LCLS Fast Feedback Communication Infrastructure Interface Control Document

Till Straumann

July 10, 2009

1 Introduction

In a large distributed control system some applications require data to be shipped between remote nodes in a deterministic way with low-latency. While the standard TCP/IP network provides the necessary connectivity it is inherently non-deterministic. A lot of hardware (physical network) and software infrastructure (drivers, network stack) is shared between applications which makes the real-time behavior unpredictable.

In particular, a protocol that is designed to provide reliability (e.g., TCP/Channel-Access) is likely to exhibit non-deterministic timing because of flow-control, segmentation, retransmission and other features required to guarantee reliability.

This document describes the interfaces between a proposed IP-Multicast/UDP based communication protocol, "FCOM", and the application and between internal layers. In order to reduce latencies dedicated networking hardware and software which are both entirely decoupled from the standard TCP/IP facilities shall be employed.

2 Definition of Terms

The following terms are used throughout this document

Signal A *signal* is a control-system parameter analogous to an EPICS PV but since FCOM is not within the scope of EPICS we want to avoid the term PV.

Blob A *blob* is an elementary unit of data shipped by FCOM. It holds the "value"[1] either read from a signal or destined to be written to a signal.

[1]"Value" in this context includes attributes of data such as timestamp, type etc.

Group A *group* is a compound of blobs which is bundled together for transmission.

Set A *set* of blobs is an object comprising multiple blobs bundled for reception. A single thread can block until all requested members of the set update.

Nodes and Endpoints A *node* is defined as a computer system which is "participating" in FCOM (see sect. 3, *Connectivity*). An *endpoint* is a logical entity that is either a source or a sink of data. Multiple endpoints (of either flavor) may be present on a single node.

Frame The terms *frame* or *packet* refer to Ethernet frames.

3 Functional Requirements

The services of FCOM can be visualized as a "newsletter" publishing service. There are ''sources" of data (the editors) and "sinks" (the readers). Readers have to subscribe to receive a copy of the newsletter and they periodically have to check their mailboxes to see if a new letter has arrived. Data flow is essentially unidirectional.

More formally, the following functional requirements have been defined:

Technology FCOM shall be built on Gigabit Ethernet Technology.

Connectivity FCOM shall provide connectivity between participating nodes for communication of signals.

A node is "participating" in FCOM if it is

- physically connected to the GiGE LAN that provides the physical connections.
- running the necessary software to support the FCOM protocol.

Data Format FCOM shall ship blobs in an atomical fashion. A single blob is comprised of

- protocol version information
- a unique identifier
- data type and count information
- a timestamp
- status/quality information
- an array of "count" elements of the basic data type "type"

At least the basic types `uint32_t`, `int32_t`, `float` and `double` shall be supported.

The timestamp and other information apply to all the individual elements of a blob.

NOTE The maximum size of the representation of a blob is limited. It may not exceed the size of an Ethernet datagram excluding any protocol headers[2].

The semantics of the data are transparent to FCOM and entirely defined by the application.

Representation Given the possibility that FCOM may interconnect a heterogeneous set of computer systems the usual problems with compatibility of data representation (endian-ness, floating-point format etc.) arise. FCOM shall address these issues so that the elementary data types are converted on each node into their native representation.

Data Flow Data flow is initiated by the source endpoint and cached by FCOM on the sink node(s). Data sink endpoints retrieve data from this cache.

NOTE There is no requirement for asynchronous notification of endpoints (e.g., via callbacks) but such a feature *may* be added in the future.

NOTE There is no requirement for data-sink endpoints to be able to "enforce" the source to produce new data. In fact, data-sink endpoints require no control whatsoever over when data are produced.

Subscription The implementation *may* require data-sink endpoints to *subscribe* to FCOM prior to being able to receive data.

Subscription is considered a "configuration" feature which does not require deterministic timing.

The implementation *may* request data-sink endpoints to cancel subscription when the service is no longer needed.

Multidrop Blobs that originate at any data-source endpoint shall be delivered to any data-sink endpoint that wants to receive them. There may be multiple sinks subscribing to a single blob. The sinks may reside on any set of nodes.

Grouping Because of the Ethernet requirement the implementation *may* – for efficiency reasons – *require* the definition of *groups* of blobs which are transmitted in a single Ethernet frame.

[2]If, as explained below, for efficiency reasons blobs are bundled into groups then the size limit applies to the entire group.

Latency FCOM makes a best-effort attempt to deliver a blob from source to sink endpoint in less than 500us. Due to the complexity of Ethernet technology which was not designed with deterministic timing in mind the author does not see how a design could *guarantee* a hard limit on latency.

At the time of this writing – after consultation with other experts from SLAC and CISCO – there are still open questions as to the scalability of IP multicast on a LAN.

Error Handling The FCOM API shall define a consistent way for determining if a service request (e.g., subscription, transmission, data retrieval, etc.) was successful and shall provide some means for applications to determine the reason of failure.

Diagnostics FCOM should provide statistical information for diagnostic purposes.

4 Architectural Overview

FCOM is designed to provide the necessary software infrastructure for meeting the functional requirements. We propose to use a "layered" approach for implementing FCOM. The application layer presents an interface to application programs which hides (most of) the details of the underlying communication protocols. In fact, it may turn out that for efficiency reasons some of the lower layers might have to be changed, collapsed or eliminated but such modifications of the design are transparent to application programs.

At the application layer, the user simply writes and/or reads blobs of data without having to bother with the details of the communication, e.g., with addressing, marshalling etc.

At the lower levels, we propose to use well-known protocols from the TCP/IP world or simplified variants thereof.

5 Application Layer

The application layer defines the interface to the user. This interface should be as simple as possible to use and configure and hide as much detail as possible.

The interface can be broken into two main parts. One part that satisfies producers or "sources" of data and a second part that is used by "sinks" or consumers of data.

In both cases the same simple and yet universal and flexible data format is used.

The API to the application layer defines the elementary data layout and routines to group several blobs for subsequent transmission and to initiate the data transfer.

Consumers of data first must *subscribe* in order to receive data. This will cause FCOM to take the necessary steps to make sure the desired data are received from the network and cached in memory. The API defines a routine for consumers to retrieve data from the cache.

5.1 Design Goals

FCOM should provide a simple interface for sending data from one source to multiple receivers while hiding all the details of the underlying transport mechanism.

Strict separation of functionality was desired, i.e., no "knowledge" about the semantics of the shipped data must be built into FCOM.

All the details of "addressing" should also be transparent to applications. For sake of simplicity it was deemed acceptable to use static/manual configuration tables or files for this purpose. These configuration tables/files map a symbolic name into an ID which encodes the necessary addressing information. In the future, a directory-lookup service could be added.

5.2 Data Layout

The data layout used by FCOM was chosen so that it is essentially *self-describing*. This allows for a clear separation of functionality: FCOM can use the embedded meta-data (such as 'type' and 'size' etc.) to properly handle marshalling issues and allocate space etc. without having to know anything about the semantics of the data which are entirely defined by the application.

5.2.1 Elementary Items

The elementary entity of data or "blob" is composed of

Version Protocol version information composed of two pieces, 'major' and 'minor'. Changes of the 'major' version imply incompatible changes whereas changes of the 'minor' version imply backwards-compatible extensions.

E.g., if a receiver was built for version 2.3 it shall reject a blob of version 3.0 but shall accept a blob of version 2.4. Even though certain new features may not be available, the implementation guarantees that the subset of features of versions 2.3 and 2.4 are.

ID A universal, unique *ID* which is used by sources and sinks to refer to a particular datum. FCOM locates a blob in its cache based on the ID.

The IDs are available to application programs as symbolic constants. *Their numerical representation is defined by FCOM and opaque to the application.*

Applications must not make any assumptions about the size of IDs or their semantics.

FCOM defines a data type for IDs which applications shall use when they want to store an ID.

FCOM may define macros or subroutines for extracting information embedded in IDs as well as for generating IDs from other information.

Timestamp A timestamp marking the time when the datum was produced or when it is to be applied (but the exact semantics are defined by the application).

Status Status information associated with the datum. The semantics are defined by the application.

Type An element-type. Currently, the C99 types `int8_t`, `int32_t`, `uint32_t`, `float` and `double` are supported.

Count An element-count. The datum consists of an array of elements which all share the same status, timestamp, type and ID.

Alignment FCOM guarantees that the array of data elements ("payload") in the cache is properly aligned for access by a *SIMD* unit if present on a particular CPU architecture.

When handing blobs to FCOM the application is responsible for the proper alignment of the blob header and payload. However, natural alignment is sufficient but maintaining "vector-alignment" *may* result in higher efficiency.

5.2.2 Grouping

Because of the Ethernet requirement the implementation may require – for efficiency reasons – the definition of *groups* of blobs which can be transmitted in a single Ethernet frame.

Groups essentially map to IP multicast groups.

A group has the following properties:

- A group has a unique *group ID*.

- Any blob with a given ID is a member of exactly one group. The group ID can therefore be a sub-part of the blob's ID.
- Multiple blobs belonging to the same group can be sent in a single Ethernet frame.
- Blobs belonging to different groups cannot be sent in a single frame.
- When a endpoint subscribes to a particular blob the node hosting the communication endpoint actually subscribes to the group and therefore FCOM has to process the entire group (but this fact is hidden from the receiving endpoint).

From these properties it follows that a smaller group size provides finer grained control at the expense of more and smaller Ethernet frames being sent (but these may not be distributed as widely as bigger groups because they are likely to have less subscribers).

A smaller group size increases the total number of groups and thus consumes more resources on the Ethernet switches which have to maintain path information for each individual group.

A larger group size reduces the protocol overhead (as long as the sink assembles as many blobs as possible into a single transmission) and the use of resources on the switches for packet routing but larger groups are likely to be more widely distributed which increases overall network load.

It shall be the responsibility of the system designer to define a suitable set of groups.

FCOM defines an upper bound on the number of groups it supports.

The group a blob is a member of shall implicitly map to a unique IP multicast address so that the data source and sink both "know" what IP multicast address to use for communication.

NOTE Because of the requirement to bundle blobs into groups a particular setpoint variable may require to be represented in multiple groups: Assume an application wants to control three parameters A, B and C which are grouped into G1. Another application (not executing at the same time) may want to control B,E and F by means of a group G2. Hence, depending on the application, B must be represented in group G1 or G2. Because a single ID can only be a member of a single group *two different IDs* are needed to deal with the situation described above, e.g., G1_B and G2_B. The first application would subscribe to the former whereas the alternate application would subscribe to the latter ID.

5.2.3 Sets

The implementation may define *blob sets* which consist of a set of IDs. An application may retrieve a set with a single operation. Sets allow an application to block for the arrival of new data for all members of the set (or a subset). Such a feature could be very useful if a single thread needs multiple blobs before processing the data contained in them.

5.2.4 Configuration

Applications need to refer to particular blobs using their IDs. Furthermore, FCOM needs to determine the group a particular blob is a member of.

It shall be permissible for the implementation to use a *static database* for this purpose of *association*. The database (e.g., header file) contains definitions of group and blob IDs which associate a symbolic name with the numerical values of the IDs. Applications use the symbolic names when referring to a particular blob.

The database also defines the composition of groups.

Blob IDs are composed of a signal part and a group part; each part ranges from zero to a maximum defined (as symbolic constants) by FCOM. FCOM provides macros to assemble a blob ID from its parts.

The numerical IDs 0..7 are reserved.

EXAMPLE The signal part of the ID could be split into a subsystem/IOC sub-part (10-bit) and a "PV" sub-part (6-bit) to be defined by the subsystem designer.

This approach partitions the numerical range of signals so that changes/additions to individual subsystems are possible without having to renumber (and recompile and restart) everything.

The group IDs could be defined numerically identical to the subsystem/IOC part of the signal part. However, note that the group ID indicates where a *value originates* and the subsystem/IOC sub-part indicates where a signal resides.

For a "detector" signal the origin and signal location are identical. Hence such a "PV" would have identical group and subsystem IDs.

On the other hand, an "actuator" signal originates on a controller IOC targetting an actuator IOC. Therefore, the group (controller IOC) ID and subsystem (actuator IOC) IDs would be different.

5.2.5 Alternative

From the previous sections it should have become obvious that grouping considerably complicates the general concept and the API.

If performance is sufficient, every signal could simply be assigned a single group and the whole issue would go away.

When configuring the IDs there would be no need to define groups and the API for transmitting data would be symmetrical to the receiving counterpart without any need for explicitly assembling groups.

Finally, without grouping an actuator could simply read a particular signal without having to switch subscription when an application controlling a different set of actuators is started. This would eliminate any conceptual difference between detectors, controllers and actuators.

As a compromise, detector signals could still be grouped while assigning every actuator signal it's own ID.

5.3 API

The C programming language according to the standard C99 shall be used. The API to FCOM is defined in a header file `fcom_api.h` (see appendix A) which shall be protected from multiple inclusion and shall be usable from C++.

Most routines return an error status with zero indicating success and nonzero an error condition. The return value specifies the type of error encountered.

5.3.1 Data Types

FCOM defines data types for IDs and blobs. A blob is a C `struct` containing a protocol version, ID, type, count, timestamp, status and a pointer to the data array.

NOTE The application *must* set the version, ID, type and count fields as well as the pointer to the "payload" data array before handing a blob to FCOM.

The timestamp, status and actual data are not interpreted or used by FCOM. Their semantics are defined by the application.

5.3.2 Error Handling

All FCOM routines which possibly can fail return an `int` value. A return value less than zero indicates failure. The possible error status values are defined in the `<fcom_api.h>` header. Successful execution is indicated by a return value of zero.

In some cases and FCOM routine may fail due to an error at a lower level, in many cases a system error. If the lower level flags an error reason by means of the `errno` variable then the `errno` value is encoded into the FCOM routine return value and may be retrieved using the `FCOM_ERR_SYS_ERRNO(status)` macro. Another macro, `FCOM_ERR_IS_SYS(status)` allows the application to test if in fact there is such an encoded `errno` value.

A routine

```
const char * fcomStrerror(int fcom_error_status);
```

converts an error status into a human-readable ASCII-string.

5.3.3 Subscription

Before an application is able to read data from the cache it must subscribe with a call to

```
int fcomSubscribe(FcomID id, int mode);
```

providing the ID of the desired blob. Subscription is not a "real-time" operation – an unspecified amount of time may elapse before data are cached.

The `mode` argument normally is `FCOM_ASYNC_GET`. Optionally, the implementation may implement `FCOM_SYNC_GET`. In this mode, `fcomGetBlob()` may (depending on it's `timeout_ms` argument) block until new data arrive.

`fcomSubscribe(id,FCOM_SYNC_GET)` returns `FCOM_ERR_UNSUPP` if the implementation does not support synchronous operation[3].

NOTE `ids` subscribed for synchronous operation may consume more resources than ordinary asynchronous subscriptions.

In order to conserve resources an application should unsubscribe a blob when it is no longer needed.

[3]The expression `(FCOM_ERR_UNSUPP != fcomSubscribe(FCOM_ID_ANY, FCOM_SYNC_GET))` is a possible run-time test for the availability of the synchronization feature

```
int fcomUnsubscribe(FcomID id);
```

5.3.4 Reception

FCOM maintains a cache of the data it receives from the network. Applications can obtain a pointer to the cached data using

```
int fcomGetBlob(FcomID id, FcomBlobRef *pp_blob, uint32_t timeout_ms);
```

NOTE Applications *must not* alter the contents of the data; they are only allowed to read them.

When fresh data arrive these are stored in a *different area* so that the application does not have to worry about locking the blob or it being updated while the application is accessing it. I.e., the sequence

```
FcomBlobRef before, after;

fcomGetBlob( SOME_ID, &before, 0 );

/* NEW DATA ARRIVE HERE */

fromGetBlob( SOME_ID, &after, 0 );
```

yields two different pointers `before` and `after`. Applications can use the timestamp fields to determine if the data are "fresh enough".

The `timeout_ms` argument is normally set to zero which is equivalent to *asynchronous* operation. If the `id` was subscribed for *synchronous* operation with a `mode=FCOM_SYNC_GET` then `fcomGetBlob()` accepts a nonzero `timeout_ms` argument. The routine then blocks for the given amount of milli-seconds or until fresh data arrive – whichever occurs first. The return value is `FCOM_ERR_TIMEDOUT` if no data arrive during the timeout period. A given `id` *must* have been explicitly (and successfully) subscribed for synchronous operation – otherwise an attempt to call `fcomGetBlob()` with a non-zero timeout yields `FCOM_ERR_UNSUPP`. Executing `fcomGetBlob()` with a zero timeout argument is possible and effectively (i.e., in terms of execution time) equivalent for either mode that was specified for subscription.

NOTE A synchronous `fcomGetBlob()` is *not atomical* with respect to blocking for data arriving and retrieving them.

It only guarantees that the data returned to the caller arrived after `fcomGetBlob()` was called. It is theoretically possible that the blob is updated more than once before the caller is scheduled to execute and retrieve the blob – this possible scenario depends e.g., on the task-priority of the caller and CPU load etc. A synchronous `fcomGetBlob()` does guarantee, however, that the blob was updated *at least once* since the routine was started. As in asynchronous mode the caller may use e.g., the blob's timestamp to determine if data are "valid" or "fresh enough".

Applications must release a blob when it is no longer needed so that FCOM can free up resources:

```
  fcomReleaseBlob( &before );
  fromReleaseBlob( &after );
```

5.3.5 Sets

The features and entry points described in this subsection are *optional*, i.e., they need to be enabled at compile-time and might require the availability of certain system services.

Sets are very useful if a single thread needs to obtain fresh data for a number of IDs. Without sets the application either must spawn multiple threads and let each one block for fresh data with a synchronous `fcomGetBlob()` and finally synchronize with a 'master-thread' or it must delay until it can be assumed that fresh data have arrived and then use asynchronous `fcomGetBlob()`s in order to read the data.

Using sets, the application first builds a set, then blocks for the set to "fill", i.e., for data to arrive for all member IDs and then resumes execution with the set now being populated with new blobs.

```
FcomID         member_id[] = { ID1, ID2 };
FcomBlobSetRef theSet;
fcomAllocBlobSet( member_id, 2, &theSet );
```

creates a set with two members and returns a handle in `theSet`. Note that all IDs must already be subscribed (but it is *not* necessary for them to be subscribed for synchronous operation).

The call returns zero on success and nonzero if an error occurs (returning a standard FCOM error code). The set handle is invalid if the call fails.

Note that an ID cannot be completely unsubscribed (i.e., the "nest-count" cannot drop to zero) while being member of any set.

`fcomGetBlobSet()` suspends execution of the calling thread until either the desired members update or a timeout occurs, whichever happens first.

```
FcomBlobSetMask got;
FcomBlobSetMask waitfor = (1<<0) | (1<<1);
int             flags   = FCOM_SET_WAIT_ALL;
fcomGetBlobSet( theSet, &got, waitfor, flags, timeout_ms);
```

When the routine returns (successfully) then the set is populated with new blobs which can be accessed via the set data structure:

```
typedef struct FcomBlobSetMemb {
        FcomID                  idnt;
        FcomBlobRef             blob;
        FcomBlobSetHdrRef       head; /* for INTERNAL USE ONLY */
        FcomBlobSetMembRef      next; /* for INTERNAL USE ONLY */
} FcomBlobSetMemb;

typedef struct FcomBlobSet_ {
        unsigned            nmemb;
        FcomBlobSetMemb     memb[];
} FcomBlobSet, *FcomBlobSetRef;
```

The members of the set preserve the same order that the "member_id" array had when the set was created. E.g., ID2 has member index 1 and thus the associated blob could be accessed as

```
theSet->memb[1].blob
```

(the `theSet->nmemb` field stores the number of members of the set which would be 2 in our example).

The blocking behavior is determined by the `waitfor` and `flags` (and the `timeout_ms`) arguments. The `waitfor` argument (and also the value returned in `got`) is a *bitmask* with each bit `1<<i` representing a member `i`.

- If `flags` has `FCOM_SET_WAIT_ALL` set then the call only returns when *all* members with their bit set in `waitfor` have updated.

- If `flags` has `FCOM_SET_WAIT_ALL` clear (`FCOM_SET_WAIT_ANY`) then the call returns as soon as *at least one* member with its bit set in `waitfor` has updated.
- A bitset flagging all updated members is returned to the `got` variable. If this returned value is of no interest then a NULL pointer may be passed.
- Blobs "attached" to the set logically "belong" to the set, i.e., they are released automatically either when members update as a result of passing the same set again to `fcomGetBlobSet()` or when destroying the set (`fcomFreeBlobSet()`).

  The application may "take over" a blob reference by setting the `blob` field to NULL but it is then responsible for releasing the blob.

  ```
  FcomBlobRef my_blob  = theSet->memb[1].blob;
  theSet->memb[1].blob = 0;
  fcomFreeBlobSet( theSet ); /* member # 0 is released automatically */
  fcomReleaseBlob( &my_blob );
  ```
- Only members with their bit set in `waitfor` will have their `blob` field updated.
- If the routine times out then still a subset of the requested members might have updated (and this would be reflected in the returned `got` variable).
- Sets are "single-threaded" objects, i.e., it is a programming error to let multiple threads pass the same set to `fcomGetBlobSet()` simultaneously.
- It is legal to re-use an existing set, i.e., to repeatedly pass it to `fcomGetBlobSet()` — "attached" blobs are manged automatically.
- Blob references "attached" to a set only change when the set is again passed to `fcomGetBlobSet()` (or destroyed).

A set is destroyed and all associated resources released with

```
fcomFreeBlobSet( theSet );
```

and in particular are all "attached" blobs released by this routine.

5.3.6 Sets vs. Synchronous Get Operations

Both, sets and synchronous `fcomGetBlob()` operations synchronize the execution of the calling thread with the arrival of new data. However, they serve different purposes:

- Sets are optimal if a single thread wants to wait for multiple blobs.

- Synchronous `fcomGetBlob()`s are used when multiple threads simultaneously need to wait for a single blob.

5.3.7 Transmission

An endpoint who wishes to transmit data must first allocate a group:

```
status = fcomAllocGroup( SOME_ID, &my_group );
```

The group-part of the ID passed to this routine defines the group ID. All blobs of data subsequently added to this group must belong to the same group, i.e., the GID-part of their IDs must match.

It is admissible to pass `FCOM_ID_ANY` in which case the GID of the group is defined by the first blob added with a GID different from `FCOM_GID_ANY`.

Data blobs are added to the group by calling

```
FcomBlob a_blob;
double   val;

a_blob.fc_vers = FCOM_PROTO_VERSION;
a_blob.fc_idnt = MY_BLOB_ID;
a_blob.fc_type = FCOM_EL_DOUBLE;
a_blob.fc_nelm = 1;
a_blob.fc_dbl  = &val;

val = 1.2345;

status = fcomAddGroup(my_group, &a_blob);
```

NOTE The protocol version, ID, type and element count as well as the data pointer of the blob must be set correctly before calling this routine.

`fcomAddGroup()` is then executed repeatedly to assemble the group.

It is admissible for the GID part of the ID to be `FCOM_GID_ANY` in which case it is defined by the group.

Eventually, the group is transmitted by

```
status = fcomPutGroup(my_group);
```

NOTE At this point, FCOM takes over "ownership" of the group and associated resources (`fcomAddGroup()` copies the data blobs into the group container, i.e., the user is free so reuse the blob data structure and data array area after calling `fcomAddGroup()`).

If a group is not sent, i.e., if the user decides not to execute `fcomPutGroup()` then the group can be destroyed by calling

```
fcomFreeGroup(my_group);
```

NOTE It is a programming error to call `fcomFreeGroup()` passing a group that had already been given to `fcomPutGroup()` – regardless of the status returned by the latter.

5.3.8 Statistics

The following entry points for obtaining statistics information are provided:

```
void fcomDumpStats(FILE *f);
```

prints user-readable information to a `stdio`-stream `f`, which may be `NULL`, in which case the `stdout` is used.

In some cases it is desirable for the application to obtain specific information in numerical format. For these cases the routine

```
int fcomGetStats(int n_keys, uint32_t key_arr[], uint64_t value_arr[]);
```

was implemented. This routine retrieves a bunch of values corresponding to a number of "keys". The user submits an array of dimension `n_keys` of numerical "keys" as well as a "response"-array (of the same dimension) which designates a storage area for the routine to deposit the values associated with the given keys.

The recognized keys corresponding to specific information are defined in the `<fcom_api.h>` header.

NOTE The returned information is valid only if the routine returns zero, i.e., if *all* supplied keys were recognized.

Currently, all internal counters are 32-bit wide. The routine uses 64-bit values for sake of future enhancements.

NOTE The implementation may rely on 32-bit counters being accessed/copied consistently/atomically by the CPU and thus may not use any mutual exclusion mechanism to read counters.

This means that values corresponding to multiple keys may not be read in an atomical fashion – even though they were supplied to the same call, e.g.:

```
uint32_t keys[2] = {
  FCOM_STAT_RX_NUM_BLOBS_RECV,
  FCOM_STAT_RX_NUM_BLOBS_RECV
};
uint64_t vals[2];
fcomGetStats(2, keys, vals);
if ( vals[1] != vals[2] )
  printf("No surprise\n");
```

asks for the same key – the number of blobs received – twice. However, it is possible that the two returned values differ since a new blob might have arrived/been processed between reading the counter twice.

```
int fcomDumpIDStats( FcomID idnt, int level, FILE *f);
```

prints statistics and contents of a cached blob with ID 'idnt' to a FILE. The FILE handle may be NULL in which case stdout is used.

If 'level' is nonzero then more verbose information is dumped including the payload data.

This routine is a useful diagnostic to check if data are received at all and if they update etc.

5.3.9 Initialization

Before utilizing FCOM an application must initialize the facility using

```
int fcomInit(const char *mcast_g_prefix, unsigned n_bufs);
```

The `mcast_g_prefix` argument is a string of the format

```
<multicast IP prefix> [ ':' <port number> ]
```

The multicast IP prefix must be a (valid) multicast IP address that must not overlap the range `FCOM_GID_MIN`..`FCOM_GID_MAX`. This prefix is used for all transactions by all data sinks and sources. An optional port number may also be specified. If no port number is given then `FCOM_PORT_DEFLT` is used.

NOTE All nodes participating in FCOM *must* use the same prefix and port number.

The `n_bufs` argument is used to configure the number of buffers for blobs FCOM should create. Buffers are only used by an FCOM receiver. Hence, setting `n_bufs` to zero indicates that the receiver is unused – this is recommended for applications that only wish to transmit data because memory and the FCOM port number can be saved[4].

Each blob that FCOM receives is stored in a buffer which is released when the last reference to a blob is surrendered with `fcomReleaseBlob()`.

Buffers are managed in pools of different sizes (e.g., 64, 128, 512 and 2048 bytes but the exact amount of pools and their sizes can be determined using `fcomDumpStats()` or `fcomGetStats()`). The `n_bufs` argument defines the total number of buffers that are made available. This number is divided up among the different pools with pools for smaller-sized buffers being allocated a bigger number of buffers.

Every subscribed blob requires one buffer which remains "alive" as long as a reference to the blob exists. Hence, `n_bufs` should be chosen according to the application's needs.

6 Presentation Layer

We propose to use the *XDR* format. All data are encoded to XDR before being sent and decoded after being received. This ensures interoperability of different host architectures at a relatively low cost, at least on machines which use the IEEE floating-point representation.

7 Transport Layer

We propose to use the UDP protocol without computing a checksum (relying on the link-level checksum computed/checked by Ethernet hardware).

NOTE The size of UDP messages shall be restricted to fit an Ethernet frame.

[4]Since it is usually not possible to let multiple applications running on a single node use the same port it is desirable, e.g., on a linux system with different processes executing an FCOM receiver and an FCOM transmitter, respectively, to instruct the transmitter *not* to attempt to use the reserved FCOM port since that may prevent the receiver process from using it.

A *unique port number* must be assigned to FCOM. This port number must not be used for any other service than FCOM (on the network used by FCOM). The reason is that following standard "socket" semantics multicast frames are filtered at the *interface* not at the socket. I.e., even though an interested receiver subscribes to a multicast group using BSD `setsockopt()` the group address is set on the interface, not the socket[5].

8 Network, Link and Physical Layer

State-of-the-art Gigabit Ethernet technology with a standard (non-jumbo) frame size shall be employed.

A minimal, *non-standard* (because of missing support for features such as fragmentation, options and others) IP protocol header is added. As explained in the next section, minimal support for IP is mandated by the requirement of IGMP support.

Besides that, employing IP is useful for debugging and testing purposes so that the FCOM protocol stack can be tested in a regular networking environment where many tools are available.

8.1 Multicasting

The main distribution mechanism for FCOM is *Ethernet Multicast* which must ensure that messages are delivered only to interested nodes. This is accomplished by using a switched network which is able to save bandwidth when multicast is used.

However, manual configuration of all required multicast delivery paths into the switches is extremely cumbersome and error-prone and shall therefore be avoided.

State-of-the art switches usually implement *IGMP snooping*, a technique which automatically takes care of maintaining the multicast distribution paths up to date and pruning ports where appropriate. Note, however, that the IGMP protocol is at *level 3* and therefore *requires* a minimal IP infrastructure. Note that RFC4541 (2.1.2.4) states that

> "All non-IPv4 multicast packets should continue to be flooded out to all remaining ports in the forwarding state..."

and also (2.1.2.5)

> "IP address based forwarding is preferred...".

[5]The consequence is that e.g., if application A1 receives from port P1 and application A2, which had subscribed to multicast traffic from G2, receives on port P2 then A1 still receives traffic sent to G2:P1 (even though A1 had not subscribed to group G2).

Hence, FCOM shall implement minimal IP and IGMP-v2. The network hardware infrastructure shall provide the appropriate switches and router or querier.

Appendix

A The FCOM API Header

```
/* $Id: fcom_api.h,v 1.10 2010/04/22 05:09:43 strauman Exp $ */
#ifndef FCOM_API_HEADER_H
#define FCOM_API_HEADER_H

#include <inttypes.h>
#include <stdio.h>

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Major protocol version; changes of the
 * major version mark incompatible changes.
 *
 * Incompatible changes are:
 *   - change of the XDR encoded layout
 *   - change of the ID layout
 *   - change of min/max GID
 *   - change of the FCOM_EL_<type> 'enum' values
 *
 * An example for a compatible change would be
 *   - assignment of the reserved 'res3' field
 *     for a specific purpose.
 */

#define FCOM_PROTO_CATCAT(maj,min) 0x##maj##min
#define FCOM_PROTO_CAT(maj,min)     FCOM_PROTO_CATCAT(maj,min)

#define FCOM_PROTO_MAJ_1       1
#define FCOM_PROTO_MIN_1       1

#define FCOM_PROTO_VERSION_11 FCOM_PROTO_CAT(FCOM_PROTO_MAJ_1,FCOM_PROTO_MIN_1)
#define FCOM_PROTO_VERSION_1x FCOM_PROTO_CAT(FCOM_PROTO_MAJ_1,0)

#define FCOM_PROTO_VERSION      FCOM_PROTO_VERSION_11
#define FCOM_PROTO_MAJ          FCOM_PROTO_MAJ_1
#define FCOM_PROTO_MIN          FCOM_PROTO_MIN_1

#define FCOM_PROTO_MAJ_GET(x) ( (x) & ~0xf )
#define FCOM_PROTO_MIN_GET(x) ( (x) &  0xf )

/* Match major version */
#define FCOM_PROTO_MATCH(a,b) ( FCOM_PROTO_MAJ_GET(a) == FCOM_PROTO_MAJ_GET(b) )
```

```
/*
 * ID to identify a 'blob' of data -- NEVER make
 * assumptions about the size of this type; it may
 * change (e.g., use 'sizeof(FcomID)' and never
 * cast to/from 'uint32_t')
 */
typedef uint32_t FcomID;

/* Use this format string to print FCOM IDs */
#define FCOM_ID_FMT "0x%08"PRIx32

/*
 * ID to identify the 'group' a blob belongs to.
 */
typedef uint32_t FcomGID;

/* NOTE: any change of the min/max group
 *       numbers must trigger a change of
 *       the major protocol version since
 *       the format of the ID changes!
 */
#define FCOM_GID_MIN        8
#define FCOM_GID_MAX     2047     /* power of two - 1 */

/*
 * Special group (wildcard)
 */
#define FCOM_GID_ANY        0

#define FCOM_SID_MIN        8
#define FCOM_SID_MAX    65535     /* power of two - 1 */
#define FCOM_SID_ANY        0

/*
 * Special ID (wildcard)
 */
#define FCOM_ID_ANY             FCOM_MAKE_ID(FCOM_GID_ANY, FCOM_SID_ANY)

/*
 * Special ID (always invalid)
 */
#define FCOM_ID_NONE        0

/*
 * Concatenate a group ID with a 'signal' id
 * to form a FcomID.
 *
 * ALWAYS use this macro; the definition
 * (e.g., version, shift count etc.) may
 * change.
 */
#define FCOM_MAKE_ID(gid,sid)    \
    ( ((FCOM_PROTO_MAJ)<<28) | ((gid)<<16) | (sid) )
```

```
/*
 * Extract major protocol version from ID.
 *
 *
 * NOTE: Only the 'real' major number without
 *       the 0xfc prefix is encoded in the ID.
 */
#define FCOM_GET_MAJ(id) (((id)>>28) & 0xf)

/*
 * Extract GID
 *
 * NOTE: This macro is for internal use ONLY.
 */
#define FCOM_GET_GID(id) (((id)>>16) & FCOM_GID_MAX)

/*
 * Extract SID
 *
 * NOTE: This macro is for internal use ONLY.
 */
#define FCOM_GET_SID(id) ((id) & 0xffff)

#define FCOM_GID_VALID(gid) ((gid) <= FCOM_GID_MAX && (gid) >= FCOM_GID_MIN)
#define FCOM_SID_VALID(sid) ((sid) <= FCOM_SID_MAX && (sid) >= FCOM_SID_MIN)
#define FCOM_ID_VALID(id)    (    FCOM_GID_VALID(FCOM_GET_GID(id)) \
                             && FCOM_SID_VALID(FCOM_GET_SID(id)))
```

```
/*
 * Elementary data types;
 * ALWAYS use symbolic names when referring
 * to the type -- this may change if more
 * types are added.
 */
#define FCOM_EL_NONE     0
#define FCOM_EL_FLOAT    1
#define FCOM_EL_DOUBLE   2
#define FCOM_EL_UINT32   3
#define FCOM_EL_INT32    4
#define FCOM_EL_INT8     5
#define FCOM_EL_INVAL    6

#define FCOM_EL_TYPE(t) ((t) & 0xf)
#define FCOM_EL_SIZE(t) (   \
        FCOM_EL_FLOAT  == (t) ? sizeof(float)    : \
        FCOM_EL_DOUBLE == (t) ? sizeof(double)   : \
        FCOM_EL_UINT32 == (t) ? sizeof(uint32_t) : \
        FCOM_EL_INT32  == (t) ? sizeof(int32_t)  : \
        FCOM_EL_INT8   == (t) ? sizeof(int8_t)   : \
    -1 )

/*
 * A blob of data.
 */

typedef struct FcomBlobHdr_ {
    uint8_t     vers;  /* protocol vers. */
        uint8_t     type;  /* data el. type   */
        uint16_t    nelm;  /* # of elements   */
    FcomID      idnt;  /* unique ID       */
    uint32_t    res3;  /**** reserved ****/
    uint32_t    tsHi;  /* timestamp HI    */
    uint32_t    tsLo;  /* timestamp LO    */
    uint32_t    stat;  /* status of data */
} FcomBlobHdr, *FcomBlobHdrRef;

typedef struct FcomBlob_ {
        FcomBlobHdr hdr;
        union {
        void        *p_raw;
        float       *p_flt;
        double      *p_dbl;
        uint32_t        *p_u32;
        int32_t     *p_i32;
        int8_t      *p_i08;
        }           dref;  /* ptr to data     */
/*      uint8_t     pad[32 - sizeof(FcomID) - 5*4 - sizeof(void*)]; */
} FcomBlob, *FcomBlobRef;
```

```
/*
 * Helper macros to access blob fields.
 *
 * THESE MACROS SHOULD ALWAYS BE USED TO ENSURE
 * COMPATIBILITY IF THE BLOB LAYOUT CHANGES.
 *
 * E.g., if you deal with a
 * 'float' array then use
 *
 *   for ( i=0; i<my_blob.fc_nelm; i++ ) {
 *
 *      do_something( my_blob.fc_flt[i] );
 *
 *   }
 *
 * E.g.,
 *
 *  FcomBlob pb;
 *  uint32_t data[10];
 *
 *    pb.fc_vers    = FCOM_PROTO_VERSION;
 *    pb.fc_tsHi    = my_timestamp_high;
 *    pb.fc_tsLo    = my_timestamp_low;
 *    pb.fc_idnt    = MY_ID;
 *    pb.fc_stat    = 0;
 *    pb.fc_type    = FCOM_EL_UINT32;
 *    pb.fc_nelm    = 1;
 *    pb.fc_u32     = data;
 *    pb.fc_u32[0] = my_value;
 *
 *    fcomPutBlob( &pb );
 *
 */
#define fc_vers   hdr.vers

#define fc_idnt   hdr.idnt
#define fc_res3   hdr.res3
#define fc_tsHi   hdr.tsHi
#define fc_tsLo   hdr.tsLo
#define fc_stat   hdr.stat
#define fc_type   hdr.type
#define fc_nelm   hdr.nelm

#define fc_raw    dref.p_raw
#define fc_u32    dref.p_u32
#define fc_i32    dref.p_i32
#define fc_i08    dref.p_i08
#define fc_flt    dref.p_flt
#define fc_dbl    dref.p_dbl
```

```
/** ERROR HANDLING ***************************************************/

/*
 * Error return values;
 */
#define FCOM_ERR_INVALID_ID       (-1)
#define FCOM_ERR_NO_SPACE         (-2)
#define FCOM_ERR_INVALID_TYPE     (-3)
#define FCOM_ERR_INVALID_COUNT    (-4)
#define FCOM_ERR_INTERNAL         (-5)
#define FCOM_ERR_NOT_SUBSCRIBED   (-6)
#define FCOM_ERR_ID_NOT_FOUND     (-7)
#define FCOM_ERR_BAD_VERSION      (-8)
#define FCOM_ERR_NO_MEMORY        (-9)
#define FCOM_ERR_INVALID_ARG      (-10)
#define FCOM_ERR_NO_DATA          (-11)
#define FCOM_ERR_UNSUPP           (-12)
#define FCOM_ERR_TIMEDOUT         (-13)
#define FCOM_ERR_ID_IN_USE        (-14)

#define FCOM_ERR_SYS(errno)       (-((errno) | (1<<16)))
#define FCOM_ERR_IS_SYS(st)       ( (st) < 0 && ((-(st)) & (1<<16)))
#define FCOM_ERR_SYS_ERRNO(st)    ( FCOM_ERR_IS_SYS(st) ? (-(st)) & 0xffff : 0 )

/* Convert error status into string message.
 * System errors encoded in FCOM_ERR_SYS() are
 * converted using strerror().
 *
 * RETURNS: pointer to a non-NULL static string.
 */
const char *
fcomStrerror(int fcom_error_status);
```

```
/** INITIALIZATION **************************************************/

#define FCOM_PORT_DEFLT          4586
/*
 * Initialize the FCOM facility.
 *
 * ARGUMENTS:
 *
 *   mcast_g_prefix:
 *           String of the form <mcast_ip_prefix> [ : <port_no> ]
 *
 *           The <mcast_ip_prefix> is a (valid) multicast
 *           IP address that must not overlap FCOM_GID_MIN..FCOM_GID_MAX
 *           and which is used as a prefix for all transactions.
 *
 *           An optional port number may be provided (must be the
 *           same for all applications). If this is omitted then
 *           FCOM_PORT_DEFLT is used.
 *
 *   n_bufs: Number of buffers for blobs to create. This should be a
 *           multiple of the max. # of blobs the application plans
 *           to subscribe to. If different threads 'hold' references
 *           to blobs for longer times this multiple needs to be
 *           bigger.
 *
 * RETURNS:  Zero on success, nonzero on error.
 *
 * NOTE:     This routine is NOT thread-safe.
 */
int
fcomInit(const char *mcast_g_prefix, unsigned n_bufs);
```

```
/** GROUPS **********************************************************/

/*
 * Opaque handle for a group.
 */
typedef void *FcomGroup;

/*
 * Obtain an empty group/container for
 * the group to which "id" belongs to.
 *
 * It is admissible to pass FCOM_ID_ANY.
 * In this case the group ID is defined
 * by the first blob with a GID different
 * from FCOM_GID_ANY.
 *
 * RETURNS: zero on success, nonzero on error.
 */
int
fcomAllocGroup(FcomID id, FcomGroup *p_group);

/*
 * Add a blob of data to a group. The data
 * are copied into the 'group' container.
 *
 * RETURNS: zero on success, nonzero on error
 *          (e.g., adding a blob with a GID part
 *          of its ID that doesn't match the GID
 *          of the group yields FCOM_ERR_INVALID_ID).
 *
 * NOTES: No transmission occurs.
 *
 *          All blobs added to a group must have
 *          the same GID part of their ID. Adding
 *          a new blob to a group that already
 *          contains one or more blobs with a
 *          different GID results in an error.
 *
 *          It is a programming error to add more than
 *          one blob with the same ID to a group.
 *          Behavior in this case is undefined.
 *
 */

int
fcomAddGroup(FcomGroup group, FcomBlobRef p_blob);

/*
 * Discard group; release all resources.
 *
 * NOTE: Group handle must not be used anymore.
 */
void
fcomFreeGroup(FcomGroup group);
```

```
/** TRANSMISSION *****************************************************/

/*
 * Send out group.
 *
 * RETURNS: zero on success, nonzero on error.
 */
int
fcomPutGroup(FcomGroup group);

/*
 * Write a blob of data.
 * This routine may only be used for blobs
 * that are the sole members of a group!
 *
 * This demonstrates that the API becomes
 * much simpler if grouping can be avoided.
 *
 * RETURNS: zero on success, nonzero on error.
 */

int
fcomPutBlob(FcomBlobRef p_blob);
```

```
/** SUBSCRIPTION ****************************************************/

/*
 * Subscribe.
 *
 * If the 'sync_get' argument is FCOM_SYNC_GET then
 * the subscription supports subsequent synchronous
 * 'fcomGetBlob' operations.
 *
 * This feature is optional, i.e., not enabled
 * by default for all subscriptions because it
 * consumes additional resources.
 *
 * Also, availability of synchronous operation depends
 * on compile-time configuration.
 *
 * RETURNS: zero on success, nonzero on error.
 *          FCOM_ERR_UNSUPP if FCOM was built
 *          w/o support for synchronous operation
 *          but FCOM_SYNC_GET was specified.
 *
 */

#define FCOM_SYNC_GET   1
#define FCOM_ASYNC_GET 0
int
fcomSubscribe(FcomID id, int mode);

/*
 * Cancel subscribtion.
 *
 * NOTE: Subscription nests. Subscription is only
 *       cancelled when fcomUnsubscribe() is executed
 *       as many times (on the same ID) as fcomSubscribe()
 *       had been.
 *
 * RETURNS: zero on success, nonzero on error.
 *       The last, unnesting, fcomUnsubscribe() operation
 *       may fail when attempting to unsubscribe an 'id' on
 *       which another thread is currently blocking
 *       (synchronous fcomGetBlob()).
 */
int
fcomUnsubscribe(FcomID id);
```

```
/** RECEPTION ******************************************************/

/*
 * Obtain a pointer to a blob of data from the cache.
 *
 * If the 'timeout_ms' argument is zero then an ordinary,
 * asynchronous operation is performed. If 'timeout_ms' is
 * nonzero then the calling thread is blocked for at most
 * 'timeout_ms' milliseconds or until fresh data arrive.
 * Synchronous operation is only possible if the a subscription
 * to 'id' with the FCOM_SYNC_GET attribute had been performed.
 * Also, availability of this feature depends on compile-time
 * configuration of FCOM.
 *
 * RETURNS: zero on success, nonzero on error.
 *          Pointer to retrieved blob is returned in *pp_blob.
 *          In particular, FCOM_ERR_NO_DATA may be returned if
 *          no data have arrived since subscription.
 *          FCOM_ERR_UNSUPP is returned if 'timeout_ms' is
 *          nonzero but FCOM was built w/o support for
 *          synchronous operation.
 *          FCOM_ERR_NOT_SUBSCRIBED is returned if either no
 *          subscription for 'id' exists or a blocking/
 *          synchronous operation is attempted but no
 *          subscription with the FCOM_SYNC_GET attribute exists.
 *
 * NOTES:   User must not modify/write the retrieved blob.
 *
 *          User must release the blob when done (fcomReleaseBlob()
 *          below).
 *
 *          The retrieved blob is NOT overwritten or updated
 *          when fresh data arrive.
 */

int
fcomGetBlob(FcomID id, FcomBlobRef *pp_blob, uint32_t timeout_ms);

/*
 * Release reference to a blob of data. If
 * the reference count drops to zero then
 * FCOM releases resources associated with the
 * blob.
 *
 * RETURNS: zero on success, nonzero on error.
 *
 *          NULL is stored into *pp_blob (on success).
 */
int
fcomReleaseBlob(FcomBlobRef *pp_blob);
```

```
/** BLOB SETS *****************************************************/

/*
 * Sets of blobs allow an application to block until
 * either any or all of the member blobs arrive.
 */

typedef uint32_t FcomBlobSetMask;

/* Opaque type; for FCOM internal use only */
typedef struct FcomBlobSetHdr  *FcomBlobSetHdrRef;

typedef struct FcomBlobSetMemb {
        FcomID                  idnt;
        FcomBlobRef             blob;
        FcomBlobSetHdrRef           head; /* for INTERNAL USE ONLY */
        struct FcomBlobSetMemb  *next; /* for INTERNAL USE ONLY */
} FcomBlobSetMemb, *FcomBlobSetMembRef;

typedef struct FcomBlobSet_ {
        unsigned            nmemb;
        FcomBlobSetMemb     memb[];
} FcomBlobSet, *FcomBlobSetRef;

/*
 * Allocate a set of blobs. You must pass a list of IDs and will obtain
 * a set with all memb[i].blob references == NULL.
 * All IDs must previously have been subscribed and none of them
 * can be unsubscribed (i.e. the nest count cannot drop to zero)
 * while being member of a set.
 *
 * After use, the set can be destroyed with fcomFreeBlobSet().
 *
 * RETURNS: Zero on success error status on failure. A valid set reference
 *          is only returned in *pp_set if the call is successful.
 *
 * NOTES:   Sets are 'single threaded', i.e., it is a programming error
 *          if multiple threads pass the same set to fcomGetBlobSet()
 *          simultaneously.
 *          It is also a programming error to free a set while a
 *          fcomGetBlobSet() operation is in progress.
 */
int
fcomAllocBlobSet(FcomID member_id[], unsigned num_members, FcomBlobSetRef *pp_set);

/*
 * Destroy a blob set. Any remaining (non-NULL) blob references
 * p_set->memb[i].blob are automatically released.
 *
 * RETURNS: zero on success, nonzero (status) on error.
 */
int
fcomFreeBlobSet(FcomBlobSetRef p_set);
```

```
/*
 * Wait for a set of blobs to arrive.
 *
 * The 'waitfor' and '*p_res' arguments are bitmasks with bit (1<<i) corresponding
 * to set member 'i' in p_set->memb[i].
 *
 * If the FCOM_SET_WAIT_ANY flag is passed then the call returns if any of the members
 * with its corresponding bit in 'waitfor' is updated.
 * If FCOM_SET_WAIT_ALL is passed then the call only returns when all members with
 * their bits set in 'waitfor' are updated (or the timeout expires).
 *
 * All members that were updated while waiting will have their bit set in *p_res
 * upon return.
 *
 * RETURNS: zero on success or nonzero failure status.
 *
 * NOTES:   Even if the timeout expires (FCOM_ERR_TIMEDOUT) some members still
 *          may have been updated (with their bit set in p_res).
 *
 *          It is legal to repeatedly call fcomGetBlobSet() with the same 'p_set'
 *          reference. Any non-null blob references that are 'attached' to the set
 *          are automatically released.
 *
 *          If the user wants to 'preserve' a blob reference then the corresponding
 *          memb[i].blob pointer must be set to NULL but it is then the user's
 *          responsibility to execute fcomReleaseBlob() explicitly.
 *
 *          Only blobs that were requested in the 'waitfor' mask are updated.
 *
 */
#define FCOM_SET_WAIT_ANY       0
#define FCOM_SET_WAIT_ALL       1

int
fcomGetBlobSet(
    FcomBlobSetRef p_set,
    FcomBlobSetMask *p_res,
    FcomBlobSetMask waitfor,
    int flags,
    uint32_t timeout_ms
);
```

```
/** STATISTICS ****************************************************/

/*
 * Dump statistics to a FILE. If a NULL file pointer is passed then
 * 'stdout' is used.
 */
void
fcomDumpStats(FILE *f);

/*
 * Dump statistics and data associated with a ID to 'f' (stdout if
 * NULL). If 'level' is nonzero then more verbose information is
 * printed (including payload data).
 *
 * RETURNS: Number of characters printed or (negative) error status.
 */
int
fcomDumpIDStats(FcomID idnt, int level, FILE *f);

/*
 * Like fcomDumpIDStats but dump info about a given blob.
 */
int
fcomDumpBlob(FcomBlobRef blob, int level, FILE *f);

/*
 * Obtain statistics information.
 *
 * Note that most if not all counters are internally only 32-bits.
 * The 64-bit exchange data type is used for future enhancements.
 *
 * RETURNS: 0 on success, FCOM_ERR_UNSUPP when asking for an
 *          unknown key.
 *
 * NOTE:    There might be no locking implemented, i.e., values
 *          belonging to two different keys might be inconsistent;
 *          it is assumed that 32-bit quantities can be accessed
 *          atomically by the CPU provided that they are properly
 *          aligned.
 *          This facility is intended for informational/diagnostic
 *          purposes only.
 */
int
fcomGetStats(int n_keys, uint32_t key_arr[], uint64_t value_arr[]);

/* Test if a given key gives 32 or 64-bit values          */
#define FCOM_STAT_IS_32(key)   (0 == ((key) & (4<<24)))
#define FCOM_STAT_IS_64(key)   (0 != ((key) & (4<<24)))

/* These macros are for internal use only                 */
#define FCOM_STAT_IS_RX(key)   (1 == (((key)>>24) & 3))
#define FCOM_STAT_IS_TX(key)   (2 == (((key)>>24) & 3))
#define FCOM_STAT_IS_V1(key)   (FCOM_PROTO_MAJ_1 == (((key)>>28) & 0xf))
#define FCOM_STAT_KIND(key)    ((key)&0xffff)
```

```
#define FCOM_RX_32_STAT(n)  ((FCOM_PROTO_MAJ_1<<28)|(1<<24)|((n)<<16))
#define FCOM_TX_32_STAT(n)  ((FCOM_PROTO_MAJ_1<<28)|(2<<24)|((n)<<16))

/* Keys for RX statistics          */

/* Number of blobs received                                  */
#define FCOM_STAT_RX_NUM_BLOBS_RECV        FCOM_RX_32_STAT(1)
/* Number of messages/groups received                        */
#define FCOM_STAT_RX_NUM_MESGS_RECV        FCOM_RX_32_STAT(2)
/* Failed attempts to allocate buffer (lack of buffers)      */
#define FCOM_STAT_RX_ERR_NOBUF             FCOM_RX_32_STAT(3)
/* XDR decoder errors                                        */
#define FCOM_STAT_RX_ERR_XDRDEC            FCOM_RX_32_STAT(4)
/* Number of blobs with bad/unknown version received         */
#define FCOM_STAT_RX_ERR_BAD_BVERS         FCOM_RX_32_STAT(5)
/* Number of msgs/groups with bad/unknown version received   */
#define FCOM_STAT_RX_ERR_BAD_MVERS         FCOM_RX_32_STAT(6)
/* Number of failed synchronous or set member broadcasts     */
#define FCOM_STAT_RX_ERR_BAD_BCST          FCOM_RX_32_STAT(7)
/* Number of subscribed blobs                                */
#define FCOM_STAT_RX_NUM_BLOBS_SUBS        FCOM_RX_32_STAT(8)
/* Max. supported number of subscribed blobs                 */
#define FCOM_STAT_RX_NUM_BLOBS_MAX         FCOM_RX_32_STAT(9)

/* Keys for RX buffer statistics   */

/* Number of different kinds (sizes) of buffers supported    */
#define FCOM_STAT_RX_NUM_BUF_KINDS         FCOM_RX_32_STAT(10)
/* Separate statistics/properties for each buffer kind       */
#define FCOM_STAT_RX_BUF_SIZE(kind)        (FCOM_RX_32_STAT(11)  | FCOM_STAT_KIND(kind))
/* Total number of buffers of a particular kind/size         */
#define FCOM_STAT_RX_BUF_NUM_TOT(kind)     (FCOM_RX_32_STAT(12)  | FCOM_STAT_KIND(kind))
/* Number of available/free buffers of a particular kind     */
#define FCOM_STAT_RX_BUF_NUM_AVL(kind)     (FCOM_RX_32_STAT(13)  | FCOM_STAT_KIND(kind))
/* Guaranteed alignment of payload of a buffer kind          */
#define FCOM_STAT_RX_BUF_ALIGNED(kind)     (FCOM_RX_32_STAT(14)  | FCOM_STAT_KIND(kind))

/* Keys for TX statistics          */

/* Number of blobs sent                                      */
#define FCOM_STAT_TX_NUM_BLOBS_SENT        FCOM_TX_32_STAT(1)
/* Number of messages/groups sent                            */
#define FCOM_STAT_TX_NUM_MESGS_SENT        FCOM_TX_32_STAT(2)
/* Number of failed attempts to send (TCP/IP stack errors)   */
#define FCOM_STAT_TX_ERR_SEND              FCOM_TX_32_STAT(3)
```

```
/** EXAMPLES ******************************************************/

/*
 * Example for the use of FCOM:
 *
 * A] Header files defining IDs
 *
 * A.1] Global Header <groups.h> Defining Group/IOC IDs
 *
 *     #include <fcom_api.h>
 *
 *     // Define group IDs; one for each IOC and one
 *     // for each controller:
 *
 *     #define GID_IOC_ABC        (FCOM_GID_MIN + 0)
 *     #define GID_IOC_DEF        (FCOM_GID_MIN + 1)
 *     #define GID_IOC_XYZ        (FCOM_GID_MIN + 2)
 *
 *     #define GID_LOOP_1         (FCOM_GID_MIN + 2)
 *
 *     // Define macro to assemble the signal part
 *     // of an ID from subsystem/IOC sub-part and
 *     // "PV" part defined by subsystem designer.
 *
 *     #define MAKE_SID(sys,sig)     (((sys)<<6)|(sig))
 *
 * A.2] Detector (XYZ) IOC specific header <detector_xyz.h>:
 *
 *     #include <groups.h>
 *
 *     // ID Definitions for detector IOC.
 *     // Detector IOC subsystem designer
 *     // defines this:
 *
 *     // Two signals acquired by IOC XYZ
 *     #define SIG_XYZ_TEMP_1      MAKE_SID(GID_IOC_XYZ, 1)
 *     #define SIG_XYZ_PRESSURE    MAKE_SID(GID_IOC_XYZ, 2)
 *
 *     // Define complete IDs including group.
 *     // Group of detector signals is detector IOC GID:
 *     #define XYZ_TEMP_1          FCOM_MAKE_ID(GID_IOC_XYZ, SIG_XYZ_TEMP_1)
 *     #define XYZ_PRESSURE        FCOM_MAKE_ID(GID_IOC_XYZ, SIG_XYZ_PRESSURE)
 *
```

```
* A.3] Actuator IOC (ABC) specific definitions
*
*    #include <groups.h>
*
*    // Actuator signals residing on ABC:
*    #define SIG_ABC_CURR_1     MAKE_SID(GID_IOC_ABC, 1)
*
* A.4] Actuator IOC (DEF) specific definitions
*
*    #include <groups.h>
*
*    // Actuator signals residing on DEF:
*    #define SIG_DEF_CURR_2     MAKE_SID(GID_IOC_DEF, 1)
*
* A.5] Definitions for loop controller 1
*
*    #include <groups.h>
*    #include <actuator_abc.h>
*    #include <actuator_def.h>
*
*    // Group of actuator signals is GID
*    // of controller/loop:
*    #define LOOP_1_CURR_1      FCOM_MAKE_ID(GID_LOOP_1, SIG_ABC_CURR_1)
*    #define LOOP_1_CURR_2      FCOM_MAKE_ID(GID_LOOP_1, SIG_DEF_CURR_2)
*
```

```
* B] Data acquisition system sends two blobs of data
*
*    #include <fcom_api.h>
*    #include <detector_xyz.h>
*
*    FcomGroup group = 0;
*    FcomBlob  blob;
*    float     data[1];
*    int       status;
*
*      // obtain a group; use any ID belonging
*      // to the target group.
*      if ( (status = fcomAllocGroup(XYZ_TEMP_1, &group)) )
*        goto bail;
*
*      // fill-in header info
*      blob.fc_vers     = FCOM_PROTO_VERSION;
*      getTimestamp(&blob);
*      blob.fc_stat     = 0;
*      blob.fc_type     = FCOM_EL_FLOAT;
*      blob.fc_flt      = data;
*
*      // fill-in data
*      blob.fc_nelm     = 1;
*      blob.fc_flt[0]   = readADC_1();
*
*      // tag with ID
*      blob.fc_idnt     = XYZ_TEMP_1;
*
*      // add to group
*      if ( (status = fcomAddGroup(group, &blob)) )
*        goto bail;
*
*      // use same version, timestamp, type, data
*      // area and status for second blob:
*      blob.fc_flt[0]   = readADC_2();
*
*      // tag with ID
*      blob.fc_id       = XYZ_PRESSURE;
*
*      // add to group
*      if ( (status = fcomAddGroup(group, &blob)) )
*        goto bail;
*
*      // done; send off
*      status = fcomPutGroup(group);
*
*      // group is now gone, even if sending failed
*      group = 0;
*
*      bail:
*         // print message if there was an error
*         if ( status )
*           fprintf(stderr,"FCOM error: %s\n", fcomStrerror(status));
*         fcomFreeGroup( group );
* C] Receiver subscribes to XYZ_TEMP_1
*
*      #include <fcom_api.h>
*      #include <detector_xyz.h>
*
*      // during initialization
*      fcomSubscribe(XYZ_TEMP_1, FCOM_ASYNC_GET);
*
*      ...
*
*      FcomBlobRef p_blob;
*
*      // normal execution; get data from cache
*      if ( 0 == fcomGetBlob( XYZ_TEMP_1, &p_blob, 0 ) ) {
*        // access data; assume we know the version, type and
*        // count but could verify...
*        if ( p_blob->fc_stat ) {
*          // handle bad status
*        } else {
*          // good data
*          consumeData( p_blob->fc_flt[0] );
*        }
*        // done -- release blob
*        fcomReleaseBlob( &p_blob );
*      } else {
*        // error handling
*      }
*
*      // if XYZ_TEMP_1 is never needed again we may
*      // unsubscribe.
*      fcomUnsubscribe( XYZ_TEMP_1 );
*
```

```
* D] Block for multiple blobs to update (using sets):
*
*     FcomID           member_id[] = { ID1, ID2, ID3 };
*     FcomBlobSetMask  waitfor, got;
*     FcomBlobSetRef   the_set;
*
*     Build a set:
*
*       fcomAllocBlobSet( member_id, 3, &the_set );
*
*     Build a 'waitfor' mask containing all our members:
*
*       waitfor = (1<<0) | (1<<1) | (1<<2) ;
*
*     Block for all members to arrive (timeout 1000 ms):
*
*       fcomGetBlobSet( the_set, &got, waitfor, FCOM_SET_WAIT_ALL, 1000 );
*
*     Dump all blobs:
*
*       for ( i=0; i<the_set->nmemb; i++ ) {
*          fcomDumpBlob( the_set->memb[i].blob, 0, 0 );
*       }
*
*     Now wait for any of ID0, ID2 to update:
*
*       waitfor = (1<<0) | (1<<2);
*       fcomGetBlobSet( the_set, &got, waitfor, FCOM_SET_WAIT_ANY, 1000 );
*
*     At this point, member #1 has not changed (bit not set in waitfor)
*     but any of #0, #2 could have (the ones that changed have their
*     bit set in 'got').
*
*     Again we can dump:
*
*       for ( i=0; i<3; i++ )
*         fcomDumpBlob( the_set->memb[i].blob, 0, 0 );
*
*     Dump only changed/updated blobs:
*
*       for ( i=0; got; i++, got>>=1 )
*         if ( got & 1 ) fcomDumpBlob( the_set->memb[i].blob, 0, 0);
*
*     Destroy set and release all contained blobs.
*
*       fcomFreeBlobSet( the_set );
```

```
 * E] Get statistics; find out how many available buffers of
 *    size >= 512 bytes there are:
 *
 *    Obtain number of different buffer kinds:
 *
 *     uint32_t key = FCOM_STAT_RX_NUM_BUF_KINDS;
 *     uint64_t nkinds;
 *     fcomGetStats(1, &key, &nkinds);
 *
 *    Obtain size and number of free buffers for all kinds.
 *
 *     uint32_t *keys = malloc(sizeof(uint32_t)*2*nkinds);
 *     uint64_t *vals = malloc(sizeof(uint64_t)*2*nkinds);
 *
 *    Generate keys for all kinds of buffers
 *     for ( i=0; i<nkinds; i++ ) {
 *       keys[i]         = FCOM_STAT_RX_BUF_SIZE(i);
 *       keys[i+nkinds] = FCOM_STAT_RX_BUF_NUM_AVL(i);
 *     }
 *
 *    Get stats
 *     fcomGetStats(nkinds*2, keys, vals);
 *
 *    Count available buffers >= 512
 *
 *    unsigned count = 0;
 *    for ( i=0; i<nkinds; i++ ) {
 *      if ( vals[i] >= 512 )
 *        count += vals[i+nkinds];
 *      }
 *    }
 *
 */

#ifdef __cplusplus
}
#endif

#endif
```