# $x$AMp Reference Manual (for version 3.1)

António Casimiro
INESC
casim@inesc.pt

May 25, 1994

The library provided within the $x$AMp package contains a set of functions which are specified in this document.

**Important note:** You must include the header file `xampuser.h` which defines all constants and data structures used as routine parameters.

## Initialization routines

NAME

xampOpen, xampClose

SYNOPSIS

```
int xampOpen (nonblock, async, iohandler)
    int nonblock;
    int async;
    int (*iohandler)();

int xampClose ()
```

DESCRIPTION

**xampOpen()** opens the $x$AMp daemon and associates a stream with it. If the open succeeds, it returns a descriptor that can be used to bypass the normal input routines and access the daemon's incoming messages directly.

*nonblock* specifies whether the connection is opened in a blocking ($nonblock = 0$) or non-blocking ($nonblock <> 0$) fashion.

The *async* parameter can be set to allow asynchronous handling of incoming messages. If set then the *iohandler()* function will be called asyncronously. If *iohandler()* is NULL then the *xampReadInput()* default function will be called (**Note: asynchronous message handling is not yet available in this version of the interface**).

**xampClose()** closes the descriptor associated with the $x$AMp daemon.

RETURN VALUES

On success, xampOpen() returns a descriptor enabling access to the $x$AMp daemon. On failure, it returns -1 if an error occurs while opening the descriptor, -2 if $x$AMp was already opened or -3 if communication problems occur ($x$AMp daemon unreachable, receive error, ioctl error or message status not OK). xampClose() returns 0 on success, and -1 if the $x$AMp descriptor has not been initialized.

## Input routines

NAME

xampMainloop, xampReadInput

SYNOPSIS

```
void  xampMainloop ()

int  xampReadInput ()
```

DESCRIPTION

**xampMainloop()** provides a server loop which blocks continuously on the $x$AMp descriptor, waiting for incoming messages and redirecting them to the appropriate handlers.

**xampReadInput()** reads the $x$AMp descriptor and calls the appropriate handler (according to the type of the received frame) to process each message in the queue until the queue is empty. The blocking behavior of the call is defined only with respect to the first message, and is determined by the option selected in xampOpen().

RETURN VALUES

xampReadInput() returns 0 on success, -1 if the $x$AMp descriptor has not been initialized, and -2 otherwise.

# Object management routines

## Object Unique IDentifiers

NAME

ouidNew, ouidGetUniq, ouidIsNull, ouidIsEqual, ouidBuild, ouidAccess, ouidToString, ouidFromString

SYNOPSIS

```
objUid ouidNew ()

objUid ouidGetUniq ()
    ObjUid ouid;

int ouidIsNull (ouid)
    ObjUid ouid;

int ouidIsEqual (ouid1, ouid2)
    ObjUid ouid1;
    ObjUid ouid2;

ObjUid ouidBuild (h, l)
    long    h;
    long    l;

int ouidAccess (ouid, h, l)
    ObjUid  ouid;
    long*   h;
    long*   l;

int ouidToString (ouid, buff)
    ObjUid ouid;
    char*  buff;

ObjUid ouidFromString (buff)
    char*   buff;
```

DESCRIPTION

**ouidNew()** generates a new initialized object identifier. This initialization procedure makes the identifier to become a *null identifier*.

**ouidGetUniq()** generates a new system-wide unique object identifier at each invocation.

**ouidIsNull()** tests if the identifier passed as argument is a null identifier.

**ouidIsEqual()** tests if the two values passed as arguments correspond to the same object identifier.

**ouidBuild()** takes the two long values passed as arguments and builds a new object identifier. There is a one-to-one correspondence between any two long values and an identifier.

**ouidAccess()** performs the reverse operation of ouidBuild(), that is, produces two long values with the contents of the specified identifier.

**ouidToString()** takes as input an object identifier and produces a printable string which uniquely represents that identifier.

**ouidFromString()** works the other way round, taking as input a string and producing a correspondent object identifier.

RETURN VALUES

ouidNew, ouidGetUniq, ouidBuild and ouidFromString return an object identifier (which is a null identifier in the case of ouidNew). ouidIsNull and ouidIsEqual return 1 for positive test and 0 otherwise. ouidAccess and ouidToString return 0 on success and -1 if the suplied identifier is not valid.

## Ouid lists

NAME

ouidListAlloc, ouidListFree, ouidListAdd, ouidListSize, ouidListLength, ouidListElem, ouidInList, ouidListReadBuffer, ouidListRemBuffer

SYNOPSIS

```
OuidList* ouidListAlloc (sz)
    int sz;

int ouidListFree (olist)
    OuidList* olist;

int ouidListAdd (olist, elem)
    OuidList* olist;
    ObjUid    elem;

int ouidListSize (olist)
    OuidList* olist;

int ouidListLength (olist)
    OuidList* olist;

ObjUid ouidListElem (olist, pos)
    OuidList* olist;
    int       pos;

int ouidInList (ouid, olist)
    ObjUid    ouid;
    OuidList* olist;
```

DESCRIPTION

**ouidListAlloc** allocates the memory space that is necessary to hold an ouid list with the specified number of elements and returns a pointer to that list. This function also performs a list initialization (assures the list is empty).

**ouidListFree** frees the memory space occupied by the specified list pointer. This function must only be used to free lists previously created by the ouidListAlloc routine.

**ouidListAdd** adds an object identifier to the list. Only valid ouids (non-null) can be added to the list. If the list is full then the new element will not be added and the function will return with an error code. New elements are always added to the end of the list.

**ouidListSize** indicates the number of elements contained in the list.

**ouidListLength** indicates the size (in bytes) necessary to hold the contents of the list. This may be usefull if the user wants to treat the object identifier list as raw data and compose messages with it.

**ouidListElem** can be used to access individual elements of the list. The specific element to access is indicated by an integer correspondent to its position in the list. The position ranges from 0 to the list size minus one.

**ouidInList** tests if an object identifier is contained in a list.

RETURN VALUES

On success, ouidListAlloc returns a pointer to an object identifier list. On failure it returns a NULL pointer.

ouidListFree returns 1 on success and 0 on failure.

ouidListAdd returns 0 on success, -1 when the element to add is not valid and -2 when the list is full and the element cannot be added.

ouidListSize and ouidListLength return, respectively, the number of elements in a list and the length (in bytes) of a list.

ouidListElem returns an object identifier on success and a null identifier on failure.

ouidInList returns 1 if the ouid is found in the list and 0 otherwise.

## Group service routines

NAME

xampGroupOpen, xampGroupClose

SYNOPSIS

```
int xampGroupOpen (group_id, ouid, viewhandler, confhandler, datahandler, key)
    GroupIdentifier group_id;
    ObjUid          ouid;
    int             (*viewhandler)();
    int             (*confhandler)();
    int             (*datahandler)();
    int             key;

int xampGroupClose (group_id, ouid, key)
    GroupIdentifier group_id;
    ObjUid          ouid;
    int             key;
```

DESCRIPTION

Since $x$AMp groups are not explictly created or destroyed, xampGroupOpen and xampGroupClose have the semantics of joining and leaving a group.

**xampGroupOpen()** opens a group for communication using $x$AMp services. The interface allows a user to open up to 50 different groups, or to open the same group multiple times as long as a distinct `ouid` is specified with each request. At the interface level the call may only fail if there is not yet a connection with the $x$AMp daemon or if the limit number of concurrent requests (open or close) has been reached. This number is currently 32. At the $x$AMp level the call may fail for other reasons. The success or failure of the call at the $x$AMp level can be determined by inspecting the status of the confirmation. The status will indicate a failure in the following situations:

- There is already a group participant using the ouid supplied to the open routine — no duplicate ouids are allowed.
- The group to which the user requested to join is a reserved group. Reserved groups identifiers range from **1963** to **2002**.
- There is no internal space to old a new group. The (known) limit is 100 different groups within the same $x$AMp daemon.

*group_id* is the identification of the group to be opened. `GroupIdentifier` has the size of an integer.

*ouid* must be specified as a globally unique identifier of the object opening the group. The identification is maintained by the system, and can be used to address messages selectively to group subsets.

*key* is a user-defined unique id returned in confirmation messages which allows a user to unambiguously record any information returned upon service confirmation. For instance, if a user opens the same group twice it can use the key parameter to establish a unique correspondence between open calls and confirmations.

The three handlers are optionally specified. See the descriptions in the **Group Service Handler Routine** section for further details. If defined, these handlers are called in the following situations:

viewhandler(): This is called when the group view changes. In particular, this is called after a xampGroupOpen() procedure.

confhandler(): This handler is called when the confirmation of a request is received. It is used not only to receive group service confirmations but also for message service confirmations.

datahandler(): This is called when data is received from the group specified by *group_id*.

Note that the same handler may be used for multiple invocations of the xampGroupOpen() call. If this is the case, data from different groups will be received by the same handler routine which will still be able to associate the incoming data to a specific group using a *group_id* input parameter.

**xampGroupClose()** closes the group specified by *group_id*. This call must also specify the object identifier and optionally a *key* which will be returned on the confirmation.

RETURN VALUES

xampGroupOpen() and xampGroupClose() return 0 on success, -1 if the $x$AMp descriptor has not been initialized and -3 if the number of concurrent requests was exceeded. xampGroupClose() also returns -2 if the specified group-ouid combination has not been opened.

## Group Service Handler Routines

NAME

viewhandler, confhandler, datahandler

SYNOPSIS

```
void viewhandler (group_id, view)
    GroupIdentifier  group_id;
    OuidList        *view;

void confhandler (group_id, ouid, mess_id, status, data, data_size, key)
    GroupIdentifier  group_id;
    ObjUid           ouid;
    short            mess_id;
    short            status;
    char            *data;
    int              data_size;
    int              key;

void datahandler (group_id, ouid, mess_id, info, data, data_size)
    GroupIdentifier  group_id;
    ObjUid           ouid;
    short            mess_id;
    char            *info;
    char            *data;
    int              data_size;
```

DESCRIPTION

These are the formats of the optional handler routines supplied as parameters to the xampGroupOpen() routine. If provided, they are invoked by the system in response to the appropriate input events. These routines are not provided by the xamp libraries, but user-defined versions are expected to conform to the above interface specifications.

**viewhandler()** is called when the group view changes. In particular, this is called after a xampGroupOpen() or xampGroupClose() (the entity executing the xampGroupClose() will not receive the corresponding new view). Group view changes are atomically disseminated to all group members. New members are only considered to have joined the group after the new group view (or the join confirmation) is received.

*group_id* is the identification of the group to which the new view refers (note that the same handler may be specified in multiple calls to `xampGroupOpen()`).

*view* is a pointer to a `OuidList` structure containing a list of the `ouid`s of the objects in the group. This list can be manipulated using the routines described in the section **Object management routines**. The position of object ids in the view list is consistent among all view indications and may be used to select specific group members.

**confhandler()** is called when the confirmation of a request is received. It is used not only to receive group service confirmations but also for message service confirmations.

For instance, as a result of a call to `xampGroupOpen()`, the handler will be called after the group open completes. If the confirmation returns with a successful status, the user will then be allowed to send messages to that group.

*group_id* indicates the group to which the confirmation refers.

*ouid* may be used to differentiate multiple accesses to the same group (note that different objects may provide the same confirmation handler function).

*mess_id* indicates which type of request is being confirmed; it can take the following values:

```
OPEN_REQUEST
CLOSE_REQUEST
RELIABLE_SEND
ATOMIC_SEND
DELTA_SEND
TIGHT_SEND
```

*status* indicates the status of the request, i.e. whether it completed successfully or not. It can have the following two values:

```
STATUS_OK
STATUS_NOT_OK
```

If *mess_id* is set to `OPEN_REQUEST` or `CLOSE_REQUEST` then *data* is a `NULL` pointer and *data_size* is `0`. All other service confirmations have *data* pointing to the information sent.

The *key* parameter is set to the same value provided by the corresponding service request. This allows its use as a unique request identifier for distinguishing different requests to the same group.

**datahandler()** is called when data is received from the group specified by *group_id* and from the object specified by *ouid*.

*mess_id* indicates which type of data is being received; it can take the following values:

```
RELIABLE_RECV
ATOMIC_RECV
DELTA_RECV
TIGHT_RECV
```

In this version of *x*AMp *info* is always a **NULL** pointer. This parameter is only maintained in the interface for future release use.

Finally, *data* is a pointer to a buffer containing the transmitted data and *data_size* indicates the size of the data buffer.

## Message service routines

NAME

atomicSend, reliableSend, tightSend, deltaSend

SYNOPSIS

```
int reliableSend (group_id, ouid, to, data, size, clab, key)
    GroupIdentifier group_id;
    ObjUid          ouid;
    OuidList*       to;
    char*           data;
    int             size;
    int             clab;
    int             key;

int atomicSend (group_id, ouid, to, data, size, clab, key)
    GroupIdentifier group_id;
    ObjUid          ouid;
    OuidList*       to;
    char*           data;
    int             size;
    int             clab;
    int             key;

int tightSend (group_id, ouid, to, data, size, clab, prio, key)
    GroupIdentifier group_id;
    ObjUid          ouid;
    OuidList*       to;
    char*           data;
    int             size;
    int             clab;
    int             prio;
    int             key;

int deltaSend (group_id, ouid, to, data, size, key)
    GroupIdentifier group_id;
    ObjUid          ouid;
    OuidList*       to;
    char*           data;
    int             size;
    int             key;
```

DESCRIPTION

The message service provides a set of message primitives in order to allow transmission with variable qualities of service.

*group_id* specifies the $x$AMp group to which communication is directed.

*ouid* is a unique user-specified object identifier used to differentiate multiple accesses to the same group.

*to* is used to specify a subset of the group members to be contacted. The subset may be specified in the following three different ways:

- **All group members** may be addressed by setting *to* to the predifined value **ALL_OBJ**;
- **Only one object** can be addressed if *to* is a pointer to the ouid of the recipient object (*to* may be either a pointer to ObjUid or to OuidList);

- **A group subset** is addressed by providing a list containing the object ids. The list can be built using the object lists manipulation facilities.

*data* is a pointer to a buffer containing the data to be transmitted.

*size* indicates the size of the data buffer. The current maximum message size is 1024 bytes.

*key* is used as a user-specified unique identifier to facilitate matching of confirmations to requests.

*clab* is a user-defined **c**ausal **lab**el. All messages with the same label are guaranteed to be delivered in FIFO order.

*prio* is an integer priority value, with increasing numerical values indicating increasing priority, used to reorder messages in the context of the `tightSend` routine.

**reliableSend()** reliably sends the specified message to all of the reachable members of the group or group subset, even on sender failure. The reliable quality of service preserves FIFO ordering of messages for groups of messages having the same `clab` value.

**atomicSend()** reliably sends the specified message to all or none of the members of the indicated group or group subset. The atomic quality of service assumes a total ordering for messages within the same group having the same `clab` value.

**tightSend()** provides an atomic quality of service with user-specified priorities. As part of the two-phase accept protocol, the message is assigned a slot in the queue corresponding to its specified priority.

**deltaSend()** provides service with a global total and causal order using on a clock-driven approach.

RETURN VALUES

The various send routines return 0 on successful completion, -1 if the $x$AMp descriptor has not been initialized, -2 when ouid-group combination does not exists and -3 if the specified message size exceeds the size limit.

Success indicates acceptance of the message by the underlying medium. The associated confirmation message indicates success or failure of the high-level operation.

## MGS routines

NAME

xampMgsJoin, xampMgsLeave, xampMgsCheck

SYNOPSIS

```
int xampMgsJoin (ouid, mgshandler)
    ObjUid  ouid;
    int (*mgshandler)();

int xampMgsCheck (ouid)
    ObjUid  ouid;

int xampMgsLeave (ouid)
    ObjUid  ouid;
```

DESCRIPTION

The Mgs routines provide a site-level view of the multicast groups. Users need not concern themselves with these primitives unless they are specifically interested in notification of site-level changes. The $x$AMp daemon does an implicit `xampMgsJoin()` operation at startup.

*ouid* is a unique user-specified object identifier used to differentiate multiple accesses to the same group.

**xampMgsJoin()** notifies the system that the caller is interested in receiving Mgs site-level updates by providing a `mgshandler()` routine to process the notificiations.

**xampMgsCheck()** requests a current Mgs view from the system. The view is provided asynchronously by a subsequent invocation of the `msghandler()` specified in the `xampMgsJoin()` operation when the new view message arrives.

**xampMgsLeave()** leaves the Mgs notification groups and cleans up the corresponding system state.

RETURN VALUES

`xampMgsJoin()`, `xampMgsCheck()`, and `xampMgsLeave()` return 0 on success, -1 if the $x$AMp descriptor has not been initialized, -2 if the corresponding msghandler is not found, and -3 if the number of concurrent requests was exceeded.

# MGS service handler

NAME

mgshandler

SYNOPSIS

```
int mgsHandler (id, status, state)
    short      id;
    short      status;
    SiteEntry  state[32];
```

DESCRIPTION

**mgsHandler()** is called when confirmation of an mgs request is received. This handler is also used for the reception of site level view/state informations.

*id* indicates which type of mgs service is being confirmed or indicated; it can take the following values:

```
MGS_JOIN
MGS_LEAVE
MGS_CHECK
MGS_NEW
```

*status* indicates the status of the request, i.e. whether it completed successfully or not. It can have the following two values:

```
STATUS_OK
STATUS_NOT_OK
```

Finally, *state* is a pointer to an `SiteEntry` structure. Note that for MGS_LEAVE confirmations this pointer is NULL. The SiteEntry structure is defined as follows:

```
typedef     struct siteEntry
{
    char       used;
    char       active;
    SitePname  pname;
} SiteEntry;

typedef     struct StationPname
{
    unsigned short  hh;
    unsigned short  hl;
    unsigned short  ll;
} StationPname;
```

The SiteEntry structure fields have the following meaning:

*used* is set to indicate that the entry has already been used but is now inactive. If the site that previously used the entry restarts, then it will use the same position in the SiteEntry table.

*active* is set to indicate that the entry is currently being used by the site indicated by the field *pname*.

*pname* contains the address of a station. The address corresponds to a 48 bit Ethernet number. This number may be used along with system ether functions to get the ASCII representation of the station name.

# Timing Routines

NAME

xampTiming, xampTimingMode

SYNOPSIS

```
int xampTiming (ouid, timinghandler)
    ObjUid  ouid;
    int     (*timinghandler)();

int xampTimingMode (ouid, mode)
    ObjUid  ouid;
    int     mode;
```

DESCRIPTION

**xampTiming()** and **xampTimingMode()** are provided for debugging purposes.

**Note: This functions will be removed from the interface in the next release to increase *x*AMp performance.**

**xampTiming()** is used to toggle timing of *x*AMp messages. If the timinghandler() argument is non-null, timing is turned on and the system records timestamps when messages are sent and when confirmations and indications are received. Then the system invokes the specified timinghandler() to indicate the results. If the timinghandler() argument is not provided (is set to NULL), timing is turned off.

*ouid* is a unique user-specified object identifier used to differentiate multiple accesses to the same group.

The *timinghandler* argument is a user-supplied routine conforming to the following specification which will be invoked asynchronously when timing information is available.

```
void timinghandler (type, timestamp)
    short   type;
    TimeVal timestamp;

typedef struct TimeVal
{
    long   tv_sec;
    long   tv_usec;
} TimeVal;
```

The *type* parameter of the timinghandler can take one of the following values:

```
IO_DATA
IO_CONFIRMATION
IO_TIME
```

The system does not guarantee any order in the invocation of the handler function, so it is possible to receive two **IO_DATA** timestamps before any **IO_CONFIRMATION** one.

**xampTimingMode()** is used to set the system level at which timing measurements will be performed. Two timing modes are available:

```
T_APP
T_XAMP
```

When mode is set to **T_APP** the measurements will be performed at the application level, thus outside the $x$AMp daemon.

The xampTimingMode() routine can only be used after the xampTiming() routine.

RETURN VALUES

**xampTiming()** and **xampTimingMode()** return 0 on success, -1 if the $x$AMp descriptor has not been initialized and -2 if an error occurs related with the timinghandler() routine. xampTimingMode() returns -3 if the mode parameter is not correct.

## Time Service Routines

NAME

tsRead

SYNOPSIS

```
SyncType tsRead ()
```

DESCRIPTION

**tsRead()** requests a time update. The time is returned directly (note that SyncType is a typedef for `double`) as a number in seconds.

RETURN VALUES

`tsRead()` returns the time on success, and -1 on failure.