Athapascan-0
Reference Manual
Version 2.4.13 - Mar 10, 1998

Jacques Briat
Ilan Ginzburg
Marcelo Pasin

17th March 1998

Abstract

This document describes the function syntax of Athapascan-0. For basic principles and guidelines to the use of Athapascan-0, please refer to the User Manual.

Please, send all your comments about this manual to the authors, so perhaps one day it would read well.

Contents

Introduction

Here is provided the description of the Athapascan-0 C language programming interface. All functions return an error code and other input and output parameters are passed as arguments.

The file `ath0.b` includes all user necessary Athapascan-0 prototypes and all header files of libraries necessary to Athapascan-0Ṅo inclusion of any other file is needed (for threads, MPI or akernel). When including header files like `stdio.h` or `errno.h`, they have to be included **after** `ath0.b`, because pthreads and MPI redefine certain basic things.

All Athapascan-0 types are prefixed by `a0t`, such as `a0tBuffer`. All variables and functions are prefixed by `a0`, as in `a0Send()` and `a0NewSemaphore()`. All preprocessed constants are prefixed by `A0`, like `A0DefaultStack`.

This is the reference manual of the C interface to the Athapascan-0 parallel runtime kernel.

a0Abort()

Aborts the whole program.

Syntax

```
a0tError   a0Abort(
                     int   code );
```

Arguments

Name	rd/wr	Description
code	write	Error code to be returned.

Description

This function aborts the program execution, in case of an user detencted irrecoverable error. It terminates all the parallel program and doesn't return to user anymore. The value `code` is passed "as is" to the Unix `exit()` function.

See Also

`a0Init()` `a0Terminate()`

a0BroadcastCondition()

Unblock all threads blocked on a condition variable `cond`.

Syntax

```
a0tError   a0BroadcastCondition(
                                        a0tCondition   *cond);
```

Arguments

Name	rd/wr	Description
cond	read	Condition to be signaled.

Description

This function unblocks all threads blocked on a condition variable `cond`.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewCondition()` `a0DisposeCondition()` `a0SignalCondition()` `a0WaitCondition()` `a0TimedWaitCondition()`

a0ClearBuffer()

Clears a buffer to reuse.

Syntax

```
a0tError   a0ClearBuffer(
                              a0tBuffer   *buffer );
```

Arguments

Name	rd/wr	Description
buffer	write	Descriptor of the buffer to clean.

Description

This function cleans `buffer`, after being sent, so the program can reuse it. After sending the data to a remote node, the buffer still contains everything in the state it was, so the buffer can be sent again, containing the same data, to other node. A buffer must be cleared before new data is packed into it. After creating or clearing a buffer, it is said to be **empty**.

After calling this function, the type and the size of the buffer do not change.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Pack()` `a0SendBuffer()` `a0StartRemoteThread()` `a0StartRemoteUrgent()` `a0DisposeBuffer()` `a0NewBuffer()`

A0DefaultPriority

The default thread scheduling priority.

Syntax

```
int   A0DefaultPriority;
```

Description

This global variable contains the Athapascan-0 default thread scheduling priority. Each scheduling rule (see `A0DefaultScheduling`) has a different range of valid priorities, varying from `0` to the maximum allowed to each rule. The maximum values are:

- `A0MaxFifoPriority`
- `A0MaxRRPriority`

See Also

`a0SetDefaultPriority()` `a0NewService()` `a0GetThreadPriority()` `a0SetThreadPriority()` `A0DefaultScheduling`

A0DefaultScheduling

The default thread scheduling rule.

Syntax

```
a0tScheduling  A0DefaultScheduling;
```

Description

This global variable contains the Athapascan-0 default thread scheduling rule. The scheduling rules of Athapascan-0 have different priorities between each other and the scheduling policy for each rule is different. The rules are the same supplied by Posix threads. There are four possible scheduling rules, presented in order of priority:

- `A0FifoScheduling`: Run to completion or blocking of the thread. Higher priority threads are scheduled before lower priority ones.
- `A0RRScheduling`: Round-robin with priorities to execute. A thread of lower priority is only scheduled when all higher priority ones are blocked or completed.
- `A0OtherScheduling`: Any vendor-specific implemented scheduling. Unfortunately, this is the only that exists in every implementation.

To the user programs find out which scheduling is supported by the version of the Athapascan-0 library being used, there are some predefined macros. They exist only when their counterpart scheduling is supported. They are as follows:

- `A0HasFifoScheduling`
- `A0HasRRScheduling`
- `A0HasOtherScheduling`

See Also

`a0SetDefaultScheduling()` `a0NewService()` `a0GetThreadScheduling()` `a0SetThreadScheduling()` `A0DefaultPriority`

A0DefaultStack

The default thread stack size.

Syntax

```
size_t   A0DefaultStack;
```

Description

This global variable contains the Athapascan-0 default stack size. this value can be changed by the user in the startup process of the program, with the option `-a0stack`.

See Also

`a0SetDefaultStack()` `a0NewService()`

a0DetachSlave()

Informs that a slave will not be waited.

Syntax

```
a0tError   a0DetachSlave(
                                a0tThread   *thread );
```

Arguments

Name	rd/wr	Description
thread	write	Address of the slave thread desciptor.

Description

This function tells to the threads kernel that the slave thread `thread` will not be waited for. The function `a0JoinSlave` will not be expected to be used and all structures allocated to the slave thread will be freed as soon it terminates.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrInvThread`	The thread descriptor is invalid.
`A0ErrOk`	Successful completion.

See Also

`a0JoinSlave()  a0NewSlave()  a0ExitThread()`

a0DisposeBuffer()

Deletes a buffer.

Syntax

```
a0tError   a0DisposeBuffer(
                              a0tBuffer   *buffer );
```

Arguments

Name	rd/wr	Description
buffer	write	Descriptor of the buffer to delete.

Description

This function deletes a buffer `buffer` previously created by a call to `a0NewBuffer()`.

The buffer automatically allocated at the creation of the thread of a Remote Thread Creation call or at the launch of a Remote Request **must not be disposed** with a call to this function, because it is disposed automatically at the thread termination.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewBuffer()`

a0DisposeCondition()

Destroys the condition variable `cond`.

Syntax

```
a0tError   a0DisposeCondition(
                                   a0tCondition   *cond);
```

Arguments

Name	rd/wr	Description
cond	read	Condition to be destroyed.

Description

This function destroys the condition variable specified by `cond`. The space for destroying condition variable is not freed.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewCondition()` `a0SignalCondition()` `a0BroadcastCondition()` `a0WaitCondition()` `a0TimedWaitCondition()`

a0DisposeMutex()

Disposes a mutex.

Syntax

```
a0tError   a0DisposeMutex(
                                 a0tMutex   *mutex );
```

Arguments

Name	rd/wr	Description
mutex	write	Mutex descriptor to dispose.

Description

This routine disposes the structures allocated to a mutex and empties its descriptor. The mutex can no longer be used afer this call.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrInvMutex`	The mutex descriptor is invalid.
`A0ErrLockedMutex`	The mutex is locked by other thread.
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewMutex()  a0LockMutex()  a0UnlockMutex()`

a0DisposePort()

Disposes a port.

Syntax

```
a0tError   a0DisposePort(
                                a0tPort   *port);
```

Arguments

Name	rd/wr	Description
port	write	Port to dispose.

Description

This function disposes a port descriptor. It allows the system to reuse the port numbers of the disposed port descritor. In the current version, this function doesn't perform any action.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewPort()`

a0DisposeSemaphore()

Disposes a semaphore.

Syntax

```
a0tError   a0DisposeSemaphore(
                                          a0tSemaphore   *semaphore );
```

Arguments

Name	rd/wr	Description
semaphore	write	Semaphore descriptor to dispose.

Description

This routine disposes the structures allocated to a semaphore and empties its descriptor. The semaphore can no longer be used after this call.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewSemaphore()` `a0PSemaphore()` `a0VSemaphore()` `a0TryPSemaphore()`

a0ExitThread()

Terminates the current thread.

Syntax

```
a0tError  a0ExitThread(
                          void  *result );
```

Arguments

Name	rd/wr	Description
result	read	The result pointer of the thread.

Description

This function asks to the threads kernel to terminate the calling thread, passing in `result` the pointer of its results. If this thread is a slave one, the function `a0JoinSlave()` must be used to wait for the result pointer.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0JoinSlave()` `a0NewSlave()` `a0DetachSlave()`

a0GetBufferPosition()

Gets current buffer's storage pointer.

Syntax

```
a0tError   a0GetBufferPosition(
                                    a0tBuffer   *buffer,
                                    long        *position );
```

Arguments

Name	rd/wr	Description
buffer	read	Buffer to get the storage pointer from.
position	write	Storage pointer of the buffer.

Description

This function extracts the current storage pointer of a given buffer.

On return of this function, the variable pointed to by `position` contains the current storage pointer.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0SetBufferPosition()` `a0Pack()` `a0Unpack()` `a0NewBuffer()` `a0NewBufferStorage()`

a0GetPackedSize()

Gets size of packed data.

Syntax

```
a0tError   a0GetPackedSize(
                               a0tDatatype   datatype,
                               size_t        *size );
```

Arguments

Name	rd/wr	Description
datatype	read	Datatype to get the size.
source	write	Size of the datatype when packed.

Description

This function calculates the amount of bytes needed to pack a given datatype. It is useful to build buffers with enough space to pack all data nedded into it.

On return of this function, the variable pointed to by `size` contains the packed size of `datatype`.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Receive()` `a0IReceive()` `a0ReceiveBuffer()` `a0IReceiveBuffer()` `a0WaitRequest()`

a0GetRequestData()

Gets information from a request.

Syntax

```
a0tError   a0GetRequestData(
                                  a0tRequest      *request,
                                  a0tRequestData  *data );
```

Arguments

Name	rd/wr	Description
request	read	Request of a completed receive.
data	write	Structure containing the information.

Description

This function must be used **after** completion of a probe or a receive request. It stores in the field `data->source` the source node of the send that matched the request, useful when doing probe or receive from `A0AnySource`. It also stores in the field `data->size` the number of bytes received.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Probe()` `a0IProbe()` `a0Receive()` `a0IReceive()` `a0ReceiveBuffer()` `a0IReceiveBuffer()` `a0WaitRequest()`

a0GetSelfContext()

Gets a thread context.

Syntax

```
a0tError   a0GetSelfContext(
                                   a0tContext   *context
                                   void         **data );
```

Arguments

Name	rd/wr	Description
context	read	Context descriptor.
data	write	Data associated with the desciptor.

Description

On return of this function, the variable pointed to by `data` contains the data previously associated to the thread context descritor `context`. Each thread returns its own private data for the same context descriptor.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewContext()` `a0SetSelfContext()`

a0GetSelfThread()

Gets the thread descriptor of the calling thread.

Syntax

```
a0tError  a0GetSelfThread(
                                  a0tThread  *thread );
```

Arguments

Name	rd/wr	Description
thread	write	The thread descritor of the caller thread.

Description

On return of this function, the variable pointed to by `thread` contains the thread descritor of the calling thread.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewThread()`

a0GetSelfThreadPriority()

Gets the priority of the calling thread.

Syntax

```
a0tError   a0GetSelfThreadPriority(
                                        int   *priority );
```

Arguments

Name	rd/wr	Description
priority	write	The priority of the caller thread.

Description

On return of this function, the variable pointed to by `priority` contains the priority of the calling thread.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`A0DefaultPriority` `a0GetThreadPriority()` `a0SetDefaultPriority()` `a0SetThreadPriority()`

a0GetSelfThreadScheduling()

Gets the scheduling rule of the calling thread.

Syntax

```
a0tError   a0GetSelfThreadScheduling(
                                            a0tScheduling   *sched );
```

Arguments

Name	rd/wr	Description
sched	write	The sceduling rule of the caller thread.

Description

On return of this function, the variable pointed to by `sched` contains the scheduling rule of the calling thread.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`A0DefaultScheduling` `a0GetThreadScheduling()` `a0SetDefaultScheduling()` `a0SetThreadScheduling()`

a0GetThreadPriority()

Gets the priority of a thread.

Syntax

```
a0tError   a0GetThreadPriority(
                                       a0tThread   *thread,
                                       int         *priority );
```

Arguments

Name	rd/wr	Description
thread	read	Thread to find out the priority
priority	write	The priority of the given thread.

Description

On return of this function, the variable pointed to by `priority` contains the priority of a given thread.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`A0DefaultPriority` `a0SetDefaultPriority()` `a0SetThreadPriority()`

a0GetThreadScheduling()

Gets the scheduling rule of a thread.

Syntax

```
a0tError   a0GetThreadScheduling(
                                         a0tThread       *thread,
                                         a0tScheduling   *sched );
```

Arguments

Name	rd/wr	Description
thread	read	Thread to find out the scheduling rule
sched	write	The sceduling rule of the given thread.

Description

On return of this function, the variable pointed to by `sched` contains the scheduling rule of a given thread.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`A0DefaultScheduling` `a0SetDefaultScheduling()` `a0SetThreadScheduling()`

a0Init()

Begin Athapascan-0 startup.

Syntax

```
a0tError  a0Init(
                   int   *argcp,
                   char  *(*argvp)[] );
```

Arguments

Name	rd/wr	Description
argcp	write	**Pointer** to C main program arguments counter .
argvp	write	**Pointer** to C main program arguments table pointer.

Description

This function begins the startup phase of an Athapascan-0 parallel program. It starts all the Athapascan-0 modules — all the initial tables are filled and all the servers are started. This function **must** be called before any other Athapascan-0 functions. All program **static** objects such as services, synchronisation objects, etc.. must be declared between this call and the call to `a0InitCommit()`. Thread creation and synchronisation are not allowed in the startup phase.

The arguments used by Athapascan-0 and its lower levels are extracted from the `argvp` table.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0InitCommit()` `a0Terminate()` `a0Abort()`

a0InitCommit()

End Athapascan-0 startup.

Syntax

```
a0tError  a0InitCommit(  void  );
```

Description

This function terminates the startup phase of an Athapascan-0 parallel program. Thisfunction **must** be called before using any Athapascan-0 objets created in the initialization step. After the return of this function, all nodes have finished their startup.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Init()`

a0IProbe()

Starts a non-blocking probe in a port.

Syntax

```
a0tError   a0IProbe(
                      a0tPort      *port,
                      int          source,
                      int          tag,
                      a0tRequest   *request );
```

Arguments

Name	rd/wr	Description
port	read	Port to probe (on caller's node).
source	read	Source node to probe.
tag	read	tag to use.
request	write	Request descriptor to hold the status.

Description

This function starts a non-blocking probe on a given port `port`, from a given source node `source` (or `A0AnySource` if any node will do), with a given tag `tag`. A request is posted to probe a received a message.

A probe request is said to be completed when a message is arrived. It is useful to receive messages without knowing previously its size and origin. A probe is usually followed by a dynamic memory allocation and `a0Receive()` or **??**.

`NULL` cannot be passed as `request`.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to test or wait the completion of the probe operation or get information about it.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Probe()` `a0GetRequestData()` `a0Receive()` `a0IReceiveBuffer()`

a0IReceive()

Starts a non-blocking receive with a datatype.

Syntax

```
a0tError   a0IReceive(
                          a0tPort       *port,
                          int           source,
                          int           tag,
                          a0tRequest    *request,
                          a0tDatatype   datatype,
                          void          *address,
                          int           count );
```

Arguments

Name	rd/wr	Description
port	read	Port to listen to (on caller's node).
source	read	Source node to listen to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
datatype	read	Datatype of the data to receive.
address	write	Address where the received data goes.
count	read	Number of elements to receive.

Description

This function starts a non-blocking receive on a given port `port`, from a given source node `source` (or `A0AnySource` if any node will do), with a given tag `tag`. A request is posted to receive a message described by `datatype`.

The received data cannot be used before the completion of the request.

If a `NULL` is passed as `request`, no structures are kept after the request completion, so it cannot be tested or waited later. It is supposed that the program has other means to find out if the request has ended, as for example, a subsequent request.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to test or wait the completion of the receive operation or get information about it.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Receive()` `a0Send()` `a0ISend()` `a0IReceiveBuffer()` `a0_DATATYPES`

a0IReceiveBuffer()

Starts a non-blocking receive with a buffer.

Syntax

```
a0tError   a0IReceiveBuffer(
                                a0tPort      *port,
                                int          source,
                                int          tag,
                                a0tRequest   *request,
                                a0tBuffer    *buffer );
```

Arguments

Name	rd/wr	Description
port	read	Port to listen to (on caller's node).
source	read	Source node to listen to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
buffer	write	Receive buffer.

Description

This function starts a non-blocking receive on a given port `port`, from a given source node `source` (or `A0AnySource` if any node will do), with a given tag `tag`. A request is posted to receive a buffer in `buffer`.

The buffer cannot be used before the completion of the request.

The full contents of a buffer is replaced every time a receive is performed. There is no need to call `a0ClearBuffer()` prior to receive it.

If a `NULL` is passed as `request`, no structures are kept after the request completion, so it cannot be tested or waited later. It is supposed that the program has other means to find out if the request has ended, as for example, a subsequent request.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to test or wait the completion of the receive operation or get information about it.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0ReceiveBuffer()` `a0Receive()` `a0IReceive()` `a0ISendBuffer()` `a0Pack()` `a0Unpack()`

a0ISend()

Starts a non-blocking send with a datatype.

Syntax

```
a0tError   a0ISend(
                     a0tPort       *port,
                     int           dest,
                     int           tag,
                     a0tRequest    *request,
                     a0tDatatype   datatype,
                     void          *address,
                     int           count );
```

Arguments

Name	rd/wr	Description
port	read	Port to send to (on destination node).
dest	read	Destination node to send to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
datatype	read	Datatype of the data to send.
address	write	Address of the data to send.
count	read	Number of elements to send.

Description

This function starts a non-blocking send to a given port `port` on a given node `dest`, with a given tag `tag`. It posts a request to send a message described by `datatype`.

The sent data cannot be modified before the completion of the request.

If a `NULL` is passed as `request`, no structures are kept after the request completion, so it cannot be tested or waited later. It is supposed that the program has other means to find out if the request has ended, as for example, a subsequent request.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to test or wait the completion status of the send operation.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Send()` `a0IReceive()` `a0Receive()` `a0ISendBuffer()` `a0_DATATYPES`

a0ISendBuffer()

Starts a non-blocking send of a buffer.

Syntax

```
a0tError   a0ISendBuffer(
                                a0tPort      *port,
                                int          dest,
                                int          tag,
                                a0tRequest   *request,
                                a0tBuffer    *buffer );
```

Arguments

Name	rd/wr	Description
port	read	Port to send to (on destination node).
dest	read	Destination node to send to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
buffer	read	Send buffer.

Description

This starts performs a non-blocking send to a given port `port` on a given node `dest`, with a given tag `tag`. It posts a request to send the buffer `buffer`.

The buffer cannot be modified before the completion of the request.

Only the amount of data packed in the buffer will be sent, regardless of the total size allocated. The buffer is kept intact after being sent. It can be sent again many times or be used to pack aditional data. To clear packed data inside a buffer and reuse it to pack new data, use `a0ClearBuffer()`.

If a `NULL` is passed as `request`, no structures are kept after the request completion, so it cannot be tested or waited later. It is supposed that the program has other means to find out if the request has ended, as for example, a subsequent request.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to test or wait the completion status of the send operation.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0SendBuffer()` `a0Send()` `a0ISend()` `a0IReceiveBuffer()` `a0Pack()` `a0Unpack()`

a0IStartRemoteThread()

Creates a thread on a remote node, without blocking.

Syntax

```
a0tError   a0IStartRemoteThread(
                                    int              node,
                                    int              service,
                                    a0tScheduling    sched,
                                    int              prio,
                                    size_t           stack,
                                    a0tRequest       *request,
                                    a0tBuffer        *buffer );
```

Arguments

Name	rd/wr	Description
node	read	Node on which the thread is started.
service	read	Service to be executed by the thread.
sched	read	Scheduling rule of the thread.
prio	read	Priority of the thread.
stack	read	Stack size of thread.
request	write	Request descriptor to hold the status.
buffer	read	Buffer passed to the thread.

Description

A thread is created on node `node` to execute the service `service` with scheduling rule `sched`, priority `prio` and stack size `stack`. `A0ServiceScheduling`, `A0ServicePriority` and `A0ServiceStack` indicate that the values to be used are the ones defined with the service, at the `a0NewService()` function call. See `A0DefaultScheduling` and `A0DefaultPriority` for other possible values. This function doesn't block the calling thread until the service is executed remotely.

If no parameters are needed, a `NULL` buffer can be passed to this function.

If a `NULL` is passed as `request`, no structures are kept after the request completion, so it cannot be tested or waited later. It is supposed that the program has other means to find out if the request has ended, as for example, a subsequent request.

The created thread receives the buffer `buffer` (that must be of type `A0ThreadBufferType`) with parameters that have been packed by the caller before the call to `a0StartRemoteThread()`.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewService()` `a0StartRemoteThread()` `a0IStartRemoteUrgent()`

a0IStartRemoteUrgent()

Executes a non-blocking urgent service on a remote node.

Syntax

```
a0tError   a0IStartRemoteUrgent(
                                  int          node,
                                  int          service,
                                  a0tRequest   *request,
                                  a0tBuffer    *buffer );
```

Arguments

Name	rd/wr	Description
node	read	Node on which the service is executed.
service	read	Service to be executed.
request	write	Request descriptor to hold the status.
buffer	read	Buffer passed to the service.

Description

The service `service` is executed remotely on node `node` with parameters passed in buffer `buffer` (that must be of type `A0UrgentBufferType`). This function doesn't block the calling thread until the service is executed remotely.

If no parameters are needed, a `NULL` buffer can be passed to this function.

If a `NULL` is passed as `request`, no structures are kept after the request completion, so it cannot be tested or waited later. It is supposed that the program has other means to find out if the request has ended, as for example, a subsequent request.

The difference with starting a remote thread is that here the service is executed directly by a daemon thread, without creation of an independent thread. The function activated by an urgent service **must not block**.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewService()` `a0IStartRemoteThread()` `a0StartRemoteUrgent()`

a0JoinSlave()

Waits a slave thread to end.

Syntax

```
a0tError   a0JoinSlave(
                            a0tThread   *thread,
                            void        **result );
```

Arguments

Name	rd/wr	Description
thread	write	Address of the slave thread desciptor.
result	read	Pointer to the result pointer of the tread.

Description

This function waits a slave thread `thread` to end and store its result pointer in the address pointed by `result`. If no result is expected, a `NULL` pointer may be used. This function blocks the calling thread until the specified slave thread ends.

On return of this function, the variable pointed to by `result` contains the pointer passed by the exiting thread, with the `a0ExitThread()` function.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrInvThread`	The thread descriptor is invalid.
`A0ErrOk`	Successful completion.

See Also

`a0NewSlave()` `a0DetachSlave()` `a0ExitThread()`

a0LockMutex()

Locks a mutex.

Syntax

```
a0tError   a0LockMutex(
                             a0tMutex   *mutex );
```

Arguments

Name	rd/wr	Description
mutex	write	Mutex descriptor to lock.

Description

This routine locks the mutex pointed by `mutex`, if it is unlocked. If the mutex is already locked, the thread blocks until it be unlocked and lock it.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrInvMutex`	The mutex descriptor is invalid.
`A0ErrLockedMutex`	Locking this mutex causes a deadlock.
`A0ErrOk`	Successful completion.

See Also

`a0NewMutex()` `a0DisposeMutex()` `a0UnlockMutex()`

a0NewBuffer()

Allocates memory and creates a new buffer.

Syntax

```
a0tError   a0NewBuffer(
                           a0tBuffer   *buffer,
                           int         buftype,
                           size_t      size );
```

Arguments

Name	rd/wr	Description
buffer	write	Descriptor of the created buffer.
buftype	read	Type of the created buffer.
size	read	Size of the buffer.

Description

This function creates a buffer `buffer` allocating `size` bytes for its storage. Depending on the value of `buftype`, the created buffer is to be used as the input buffer of a remote thread (`A0ThreadBufferType`), a remote urgent call (`A0UrgentBufferType`) or a send/receive buffer (`A0SendBufferType`).

A buffer with any type can be sent and received. Only buffers of type `A0ThreadBufferType` can be used to start remote threads, with `a0StartRemoteThread()`. Only buffers of type `A0UrgentBufferType` can be used to remote urgent calls, with `a0StartRemoteUrgent()`.

All calls to `a0Pack()` and `a0Unpack()` functions receive as parameter a buffer descriptor.

On return of this function, the variable pointed to by `buffer` contains a buffer descriptor with the requested storage size.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0ClearBuffer()` `a0DisposeBuffer()` `a0NewBufferStorage()` `a0Pack()` `a0Unpack()` `a0SendBuffer()` `a0ReceiveBuffer()` `a0StartRemoteThread()` `a0StartRemoteUrgent()`

a0NewBufferStorage()

Creates a new buffer with user storage.

Syntax

```
a0tError   a0NewBuffer(
                            a0tBuffer   *buffer,
                            int         buftype,
                            size_t      size,
                            void        *storage );
```

Arguments

Name	rd/wr	Description
buffer	write	Descriptor of the created buffer.
buftype	read	Type of the created buffer.
size	read	Size of the buffer.
storage	read	User storage to be used by the buffer.

Description

This function creates a buffer `buffer` of size `size` bytes, in the storage `storage`, allocated by the user. Depending on the value of `buftype`, the created buffer is to be used as the input buffer of a remote thread (`A0ThreadBufferType`), a remote urgent call (`A0UrgentBufferType`) or a send/receive buffer (`A0SendBufferType`).

A buffer with any type can be sent and received. Only buffers of type `A0ThreadBufferType` can be used to start remote threads, with `a0StartRemoteThread()`. Only buffers of type `A0UrgentBufferType` can be used to remote urgent calls, with `a0StartRemoteUrgent()`.

All calls to `a0Pack()` and `a0Unpack()` functions receive as parameter a buffer descriptor.

On return of this function, the variable pointed to by `buffer` contains a buffer descriptor with the requested storage size.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0ClearBuffer()` `a0DisposeBuffer()` `a0NewBuffer()` `a0Pack()` `a0Unpack()` `a0SendBuffer()` `a0ReceiveBuffer()` `a0StartRemoteThread()` `a0StartRemoteUrgent()`

a0NewCondition()

Initializes a condition variable `cond`.

Syntax

```
a0tError   a0NewCondition(
                                  a0tCondition   *cond);
```

Arguments

Name	rd/wr	Description
cond	read	Condition variable to be initialized.

Description

This function initializes the condition variable specified by `cond`. The variable condition allows only threads created within the same process operate in it.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0DisposeCondition()` `a0SignalCondition()` `a0BroadcastCondition()` `a0WaitCondition()` `a0TimedWaitCondition()`

a0NewContext()

Creates a thread context descritor.

Syntax

```
a0tError   a0NewContext(
                               a0tContext   *context );
```

Arguments

Name	rd/wr	Description
context	write	Context descriptor to fill.

Description

This routine creates a thread context descritor and stores it in `context`. All threads started after the creation of a thread context descriptor can have a thread context associated with it.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0SetSelfContext()` `a0GetSelfContext()`

a0NewMutex()

Creates a mutex.

Syntax

```
a0tError   a0NewMutex(
                            a0tMutex   *mutex );
```

Arguments

Name	rd/wr	Description
mutex	write	Mutex descriptor to fill.

Description

This routine creates a mutex and store its descriptor in `mutex`.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0LockMutex()` `a0UnlockMutex()` `a0DisposeMutex()`

a0NewPort()

Creates a new port.

Syntax

```
a0tError   a0NewPort(
                        a0tPort   *port,
                        int       tags,
                        int       type);
```

Arguments

Name	rd/wr	Description
port	write	Port created.
tags	read	Number of tags that can be used with the port.
type	read	local or global creation.

Description

This function creates a port descriptor, to be used on **any** site of the computation. If `type` is `A0LocalCreation`, then the created port is only created in the local node. If `type` is `A0GlobalCreation`, then **all** nodes must do the call to `a0NewPort()` at same time and all nodes will create the **same** port descriptor.

In other words, when a node calls `a0NewPort()` with `type==A0LocalCreation`, a port descriptor is returned that it is guaranteed not to be returned by in any other, subsequent or concurrent, call to `a0NewPort()`, on any node. (Actually, it can return the same port number of a previous call, when `a0DisposePort()` has been called to free a port descriptor). When the call to `a0NewPort()` is made with `type==A0GlobalCreation`, it is supposed to happen in all nodes at the same time. All the calls on all the nodes will return the **same** port descriptor. All nodes must do global creations of ports in the same order.

`tags` gives the number of tags associated with that port. Valid values for a tag to use with that port are in the range `0` to `tags−1`. The implementation of ports with more than one tag is done by allocating consecutive ports and returning the number of the first one. In a communication, the tag is added to the first port number.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0DisposePort()` `a0Send()` `a0SendBuffer()` `a0Receive()` `a0ReceiveBuffer()`

a0NewSemaphore()

Creates a semaphore.

Syntax

```
a0tError   a0NewSemaphore(
                                a0tSemaphore   *semaphore,
                                int            initial );
```

Arguments

Name	rd/wr	Description
semaphore	write	Semaphore descriptor to fill.
initial	read	Initial semaphore counter value.

Description

This routine creates a semaphore descriptor and initialize the semaphore counter with the integer value `initial`. This value may be any integer value.

On return of this function, the variable pointed to by `semaphore` contains a semaphore descriptor.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0PSemaphore()` `a0VSemaphore()` `a0TryPSemaphore()` `a0DisposeSemaphore()`

a0NewService()

Declares a new service.

Syntax

```
a0tError   a0NewService(
                              int                  *service,
                              a0tRemoteFunction    function,
                              a0tScheduling        sched,
                              int                  prio,
                              size_t               stack );
```

Arguments

Name	rd/wr	Description
service	write	Address of the service to set up.
function	read	Function to call at service activation.
sched	read	Default scheduling rule to be used .
prio	read	Default priority of the service thread.
stack	read	Default stack size of service thread.

Description

This function declares a function to be called remotely with `a0StartRemoteThread()` or `a0StartRemoteUrgent()` and creates a service descriptor to it. When this function is called by `a0StartRemoteThread()`, the values passed in `sched`, `prio`, and `stack` are the default values for the new service, concerning scheduling rule, priority and stack size, respectively. See `A0DefaultScheduling`, `A0DefaultPriority` and `A0DefaultStack` for the possible values.

When this function is called by `a0StartRemoteUrgent()`, the function registered will not be executed as a separate thread: all remote requests arriving at a node are executed by the same thread, sequentially. Thus, a function to be executed as a Remote Request should not block in any way. Other restrictions as to what this function can and can't do may appear later, as users report bugs...

The current implementation of this service declaration function relies that the user will call it only between `a0Init()` and `a0InitCommit()`, in the same order in every node that the program executes. After `a0InitCommit()`, all the Athapascan-0 daemons are already running and an error occurs if another node calls `a0StartRemoteThread()` to a service declared later.

The buffer passed to the service call will be received as the only parameter of the service function, so the service function passed as parameter should correspond to the following prototype:

```
a0tError my_function(a0tBuffer *my_buffer);
```

On return of this function, the variable pointed to by `service` contains a service descriptor that refers to the function passed.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0StartRemoteThread()` `a0StartRemoteUrgent()` `a0Init()` `a0InitCommit()`

a0NewSlave()

Starts a new slave thread.

Syntax

```
a0tError   a0NewSlave(
                          a0tThread        *thread,
                          a0tScheduling    sched,
                          int              prio,
                          size_t           stack,
                          a0tFunction      function,
                          void             *value );
```

Arguments

Name	rd/wr	Description
thread	write	Address of the slave thread desciptor.
sched	read	Scheduling rule to be used .
prio	read	Priority of the slave thread.
stack	read	Size of daemon thread stack.
function	read	Function to be executed by the thread.
value	read	Parameter passed to the thread.

Description

This function creates a slave thread with scheduling rule `sched`, priority `prio`, `stack` bytes at the stack. This thread executes the function `function` with argument `value`.

The scheduling rule, priority and stack size may be inherited from the calling thread by using the `A0InheritScheduling`, `A0InheritPriority` and `A0InheritStack` values, respectively. There's a little bug: when you use `A0InheritScheduling` for the scheduling, the priority passed can only be `A0InheritPriority` (i.e.: even if it's something else, it will act as if it were `A0InheritPriority` :-). See `A0DefaultScheduling`, `A0DefaultPriority` and `A0DefaultStack` for other possible values.

The `function` passed as parameter should correspond to the following prototype:
`a0tError function(void *argument);`

A slave thread can have its end waited by any other thread with `a0JoinSlave()`.

On return of this function, the variable pointed to by `thread` contains a thread descriptor to the new thread.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrNoMem`	No memory (or other resources) available.
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0JoinSlave()` `a0DetachSlave()` `a0ExitThread()`

a0NodeCount

The total number of nodes in the system.

Syntax

```
int   a0NodeCount
```

Description

This global variable contains the number of nodes existing in the system.

See Also

a0SelfNode

a0Pack()

Packs data into a buffer.

Syntax

```
a0tError   a0Pack(
                     a0tBuffer      *buffer,
                     a0tDatatype    datatype,
                     void           *inbuf,
                     int            incount );
```

Arguments

Name	rd/wr	Description
buffer	write	Buffer descriptor to pack into.
datatype	read	Datatype of the data to pack.
inbuf	read	Address of the data.
incount	read	Number of elements to pack.

Description

Packs into `buffer` the `incount` occurances of `datatype` typed data starting at address `inbuf` of memory. The new data packed are added to the end of the data already packed inside the buffer.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrFailPack`	It was not possible to pack all data.
`A0ErrOk`	Successful completion.

See Also

`a0Unpack()` `a0NewBuffer()` `a0ClearBuffer()` `a0SendBuffer()` `a0ReceiveBuffer()` `a0ISendBuffer()` `a0IReceiveBuffer()` `a0_DATATYPES`

a0Probe()

Performs a blocking probe in a port.

Syntax

```
a0tError   a0Probe(
                      a0tPort      *port,
                      int          source,
                      int          tag,
                      a0tRequest   *request );
```

Arguments

Name	rd/wr	Description
port	read	Port to probe (on caller's node).
source	read	Source node to probe.
tag	read	tag to use.
request	write	Request descriptor to hold the status.

Description

This function performs a blocking probe on a given port `port`, from a given source node `source` (or `A0AnySource` if any node will do), with a given tag `tag`. A request is posted to probe a received a message.

A probe request is said to be completed when a message is arrived. It is useful to receive messages without knowing previously its size and origin. A probe is usually followed by a dynamic memory allocation and `a0Receive()` or **??**.

`NULL` cannot be passed as `request`.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to test or wait the completion of the probe operation or get information about it.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0IProbe()` `a0GetRequestData()` `a0Receive()` `a0IReceiveBuffer()`

a0PSemaphore()

Performs P on a semaphore.

Syntax

```
a0tError   a0PSemaphore(
                              a0tSemaphore   *semaphore );
```

Arguments

Name	rd/wr	Description
semaphore	write	Semaphore descriptor to operate.

Description

This routine blocks the thread if the counter associated with the semaphore is zero or negative. If the counter is positive, the thread doesn't block. When the thread in unblocked, or if the thread has not blocked, the counter is decremented.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewSemaphore()` `a0DisposeSemaphore()` `a0TryPSemaphore()` `a0VSemaphore()`

a0Receive()

Performs a blocking receive with a datatype.

Syntax

```
a0tError  a0Receive(
                        a0tPort       *port,
                        int           source,
                        int           tag,
                        a0tRequest    *request,
                        a0tDatatype   datatype,
                        void          *address,
                        int           count );
```

Arguments

Name	rd/wr	Description
port	read	Port to listen to (on caller's node).
source	read	Source node to listen to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
datatype	read	Datatype of the data to receive.
address	write	Address where the received data goes.
count	read	Number of elements to receive.

Description

This function performs a blocking receive on a given port `port`, from a given source node `source` (or `A0AnySource` if any node will do), with a given tag `tag`. It blocks the calling thread to receive a message described by `datatype`.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to check the completion status of the receive operation and to get the node number of the sending thread.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0IReceive()` `a0Send()` `a0ISend()` `a0ReceiveBuffer()` `a0_DATATYPES`

a0ReceiveBuffer()

Performs a blocking receive with a buffer.

Syntax

```
a0tError   a0ReceiveBuffer(
                                a0tPort      *port,
                                int          source,
                                int          tag,
                                a0tRequest   *request,
                                a0tBuffer    *buffer );
```

Arguments

Name	rd/wr	Description
port	read	Port to listen to (on caller's node).
source	read	Source node to listen to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
buffer	write	Receive buffer.

Description

This function performs a blocking receive on a given port `port`, from a given source node `source` (or `A0AnySource` if any node will do), with a given tag `tag`. It blocks the calling thread to receive data into the buffer `buffer`.

The full contents of a buffer is replaced every time a receive is performed. There is no need to call `a0ClearBuffer()` prior to receive it.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to check the completion status of the receive operation.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0IReceiveBuffer()` `a0Receive()` `a0IReceive()` `a0SendBuffer()` `a0Pack()` `a0Unpack()`

a0RemoteThreadCreate

No longer supported.

Syntax

```
a0tError  a0RemoteThreadCreate(  );
```

Description

This function is no longer supported.

See Also

`a0StartRemoteThread()` `a0StartRemoteUrgent()`

a0RemoteRequest

No longer supported.

Syntax

```
a0tError  a0RemoteRequest(  );
```

Description

This function is no longer supported.

See Also

`a0StartRemoteThread()` `a0StartRemoteUrgent()`

a0SelfNode

The number of the executing node of a thread.

Syntax

```
int   a0SelfNode;
```

Description

This state variable contains the number of the node that is running the current thread. It is greater than or equal to 0 and less than `a0NodeCount`.

See Also

`a0NodeCount`

a0Send()

Performs a blocking send with a datatype.

Syntax

```
a0tError   a0Send(
                  a0tPort       *port,
                  int           dest,
                  int           tag,
                  a0tRequest    *request,
                  a0tDatatype   datatype,
                  void          *address,
                  int           count );
```

Arguments

Name	rd/wr	Description
port	read	Port to send to (on destination node).
dest	read	Destination node to send to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
datatype	read	Datatype of the data to send.
address	write	Address of the data to send.
count	read	Number of elements to send.

Description

This function performs a blocking send to a given port `port` on a given node `dest`, with a given tag `tag`. It blocks the calling thread to send a message described by `datatype`.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to check the completion status of the send operation.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0ISend()` `a0IReceive()` `a0Receive()` `a0SendBuffer()` a0_DATATYPES

a0SendBuffer()

Performs a blocking send of a buffer.

Syntax

```
a0tError  a0SendBuffer(
                              a0tPort      *port,
                              int          dest,
                              int          tag,
                              a0tRequest   *request,
                              a0tBuffer    *buffer );
```

Arguments

Name	rd/wr	Description
port	read	Port to send to (on destination node).
dest	read	Destination node to send to.
tag	read	tag to use.
request	write	Request descriptor to hold the status.
buffer	read	Send buffer.

Description

This function performs a blocking send to a given port `port` on a given node `dest`, with a given tag `tag`. It blocks the calling thread to send data from the buffer `buffer`.

Only the amount of data packed in the buffer will be sent, regardless of the total size allocated. The buffer is kept intact after being sent. It can be sent again many times or be used to pack aditional data. To clear packed data inside a buffer and reuse it to pack new data, use `a0ClearBuffer()`.

On return of this function, the variable pointed to by `request` contains an opaque request that can be used later to check the completion status of the send operation.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0ISendBuffer()` `a0Send()` `a0ISend()` `a0ReceiveBuffer()` `a0Pack()` `a0Unpack()` `a0ClearBuffer()`

a0SetBufferPosition()

Sets current buffer's storage pointer.

Syntax

```
a0tError   a0GetBufferPosition(
                                    a0tBuffer   *buffer,
                                    long        *position );
```

Arguments

Name	rd/wr	Description
buffer	read	Buffer to get the storage pointer from.
position	write	Storage pointer of the buffer.

Description

This function modifies the current storage pointer of a given buffer. The next `a0Pack()` or **??** functions will operate at a new buffer storage position.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0GetBufferPosition()` `a0Pack()` `a0Unpack()` `a0NewBuffer()` `a0NewBufferStorage()`

a0SetDefaultPriority()

Sets the default thread priority.

Syntax

```
a0tError   a0GetDefaultPriority(
                                            int   priority );
```

Arguments

Name	rd/wr	Description
priority	write	Default priority.

Description

This function sets the default thread scheduling priority. See `A0DefaultPriority` for possible values.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`A0DefaultPriority` `a0StartRemoteThread()` `a0NewService()` `a0GetThreadPriority()` `a0SetThreadPriority()`

a0SetDefaultScheduling()

Sets the default thread scheduling rule.

Syntax

```
a0tError   a0GetDefaultSched(
                                   a0tScheduling   sched );
```

Arguments

Name	rd/wr	Description
sched	write	Default scheduling rule.

Description

This function sets the default thread scheduling rule. See `A0DefaultScheduling` for possible values.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`A0DefaultScheduling` `a0StartRemoteThread()` `a0NewService()` `a0GetThreadScheduling()` `a0SetThreadScheduling()`

a0SetDefaultStack()

Sets the default thread stack size.

Syntax

```
a0tError  a0SetDefaultStack(
                                size_t  size );
```

Arguments

Name	rd/wr	Description
size	write	Default stack size.

Description

This function sets the default thread stack size.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`A0DefaultStack` `a0StartRemoteThread()` `a0NewService()`

a0SetSelfContext()

Sets a thread context.

Syntax

```
a0tError   a0SetSelfContext(
                                   a0tContext   *context
                                   void         *data );
```

Arguments

Name	rd/wr	Description
context	read	Context descriptor.
data	read	Data to associate with the desciptor.

Description

This function must be called by all threads willing to have a private context data. It associates the pointer `data` with the thread context descriptor `context`. Subsequent calls to `a0GetSelfContext()` with the same context descriptor will return the `data` pointer. Each thread returns its own private data for the same context descriptor.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewContext()` `a0GetSelfContext()`

a0SetThreadPriority()

Sets the priority of a thread.

Syntax

```
a0tError   a0SetThreadPriority(
                                    a0tThread   *thread,
                                    int         priority );
```

Arguments

Name	rd/wr	Description
thread	read	Thread to change the priority.
priority	read	New priority to the thread.

Description

This function changes dynamically the priority of a given thread. See `A0DefaultPriority` for possible values.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0SetThreadScheduling()` `A0DefaultPriority` `a0SetDefaultPriority()` `a0StartRemoteThread()` `a0NewService()` `a0GetThreadPriority()` `a0SetDefaultPriority()`

a0SetThreadScheduling()

Sets the scheduling rule of a thread.

Syntax

```
a0tError   a0SetThreadScheduling(
                                      a0tThread       * thread,
                                      a0tScheduling   sched,
                                      int             priority );
```

Arguments

Name	rd/wr	Description
thread	read	Thread to change the priority.
sched	read	New scheduling rule to the thread.
priority	read	New priority to the thread.

Description

This function changes dynamically the scheduling rule and the priority of a given thread. See `A0DefaultScheduling` for possible values.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0SetThreadPriority()` `A0DefaultScheduling` `a0SetDefaultScheduling()` `a0StartRemoteThread()` `a0NewService()` `a0GetThreadScheduling()` `a0SetDefaultScheduling()`

a0SignalCondition()

Unblocks threads blocked on condition variable`cond`.

Syntax

```
a0tError  a0SignalCondition(
                                    a0tCondition  *cond);
```

Arguments

Name	rd/wr	Description
cond	read	Condition to be signaled.

Description

This function unblocks at least one thread blocked on the condition variable specified condition variable `cond`, if any thread are blocked on `cond`.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0DisposeCondition()` `a0NewCondition()` `a0BroadcastCondition()` `a0WaitCondition()` `a0TimedWaitCondition()`

a0StartRemoteThread()

Creates a thread on a remote node.

Syntax

```
a0tError   a0StartRemoteThread(
                                      int             node,
                                      int             service,
                                      a0tScheduling   sched,
                                      int             prio,
                                      size_t          stack,
                                      a0tRequest      *request,
                                      a0tBuffer       *buffer );
```

Arguments

Name	rd/wr	Description
node	read	Node on which the thread is started.
service	read	Service to be executed by the thread.
sched	read	Scheduling rule of the thread.
prio	read	Priority of the thread.
stack	read	Stack size of thread.
request	write	Request descriptor to hold the status.
buffer	read	Buffer passed to the thread.

Description

A thread is created on node `node` to execute the service `service` with scheduling rule `sched`, priority `prio` and stack size `stack`. `A0ServiceScheduling`, `A0ServicePriority` and `A0ServiceStack` indicate that the values to be used are the ones defined with the service, at the `a0NewService()` function call. See `A0DefaultScheduling` and `A0DefaultPriority` for other possible values. This function blocks the calling thread until the service startes to execute remotely.

If no parameters are needed, a `NULL` buffer can be passed to this function.

The created thread receives the buffer `buffer` (that must be of type `A0ThreadBufferType`) with parameters that have been packed by the caller before the call to `a0StartRemoteThread()`.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewService()` `a0StartRemoteUrgent()` `a0IStartRemoteThread()`

a0StartRemoteUrgent()

Executes an urgent service on a remote node.

Syntax

```
a0tError   a0StartRemoteUrgent(
                                  int          node,
                                  int          service,
                                  a0tRequest   *request,
                                  a0tBuffer    *buffer );
```

Arguments

Name	rd/wr	Description
node	read	Node on which the service is executed.
service	read	Service to be executed.
request	write	Request descriptor to hold the status.
buffer	read	Buffer passed to the request.

Description

The service `service` is executed remotely on node `node` with parameters passed in buffer `buffer` (that must be of type `A0UrgentBufferType`). This function blocks the calling thread until the service starts to execute remotely.

If no parameters are needed, a `NULL` buffer can be passed to this function.

The difference with starting a remote thread is that here the service is executed directly by a daemon thread, without creation of an independent thread. The function activated by an urgent service **must not block**.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewService()` `a0StartRemoteThread()` `a0IStartRemoteUrgent()`

a0Terminate()

Athapascan-0 termination.

Syntax

```
a0tError   a0Terminate(   void   );
```

Description

This function terminates all the threads of Athapascan-0. It must be called in **all** nodes. A parallel barrier is done before exiting to verify that all nodes have executed the function. No Athapascan-0 functions can be used after the call of this function. This function may also terminate the parallel program, so it must be the last useful function called by every node.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0Init()` `a0Abort()`

a0TestAnyRequest()

Verifies if any request of a table is complete.

Syntax

```
a0tError  a0TestAnyRequest(
                               a0tRequest   *requestab[],
                               int          size,
                               int          *index );
```

Arguments

Name	rd/wr	Description
requestab	write	Table of requests to test on.
size	write	Size of the table.
index	write	Result of the test.

Description

This function verifies if a one of the previous requests of a table has already completed. It doesn't block the calling thread, but verifies if one of the requests specified in `requestab` is completed. The size of the table must be specified in `size`. In the case of positive completion of one of the requests, it can't be waited or tested anymore.

The table `requesttab` must be constructed by the caller thread with pointers to all the requests to be tested. After a completed request, the caller thread doesn't need to bother to rebuild the table to reuse it. The function `a0TestAnyRequest()` replaces the pointer of the completed request by `NULL` and it doesn't complaint if a pointer is `NULL` in `requestab`. Though, a `NULL` pointer is also counted in `size`.

It returns in `index` the index of the completed request in the table or `A0NoneCompleted` if none of the requests completed. In the positive case, it also inserts in the respective completed request structure its final status (see `a0GetRequestData()`).

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0TestRequest()` `a0WaitAnyRequest()` `a0ISend()` `a0IReceive()` `a0ISendBuffer()` `a0IReceiveBuffer()`

a0TestRequest()

Verifies if a request is complete.

Syntax

```
a0tError   a0TestRequest(
                                  a0tRequest   *request,
                                  int          *result );
```

Arguments

Name	rd/wr	Description
request	write	Request to test on and hold the status.
result	write	Result of the test.

Description

This function verifies if a previous request has already completed. It doesn't block the calling thread, but verifies if the request specified in `request` is completed. In the case of positive completion of the request it can't be waited or tested anymore.

It returns in `result` the values `TRUE` or `FALSE`, whether the request has completed or not. In the positive case, it also inserts in the `request` structure its final status (see `a0GetRequestData()`).

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrNullReq`	The request is `NULL`.
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0TestAnyRequest()` `a0WaitRequest()` `a0ISend()` `a0IReceive()` `a0ISendBuffer()` `a0IReceiveBuffer()`

a0threadsTotal...

Accounting on the number of threads.

Syntax

```
extern int   athreadsTotal,
             athreadsBlockedComm,
             athreadsBlockedLock,
```

Description

These three variables, readable by the user but updated only by Athapascan, reflect the current state of a node, regarding the number of (user) threads running (or waiting) on that node.

- `athreadsTotal` is the total number of user threads on a node (threads are created by Remote Thread Creates, of by locally creating a slave).
- `athreadsBlockedComm` is the number of user threads on a node blocked waiting for a communication to terminate.
- `athreadsBlockedLock` is the number of user threads on a node blocked waiting for a lock (or anything that is **not** a wait for a communication to terminate).

BUG: the update of these variables in Athapascan is **not** protected, so their values might be erroneous, due to race conditions.

a0TimedWaitCondition()

blocks thread on condition variable `cond`**.**

Syntax

```
a0tError   a0TimedWaitCondition(
                                        a0tCondition             *cond,
                                        a0tmutex                 *mutex,
                                        const struct timespec    *abstime);
```

Arguments

Name	rd/wr	Description
cond	read	Condition to be signaled.
mutex	read	Mutex associated to condition variable.
abstime	read	Maximum time to wait (time-out).

Description

This function is similar to `a0WaitCondition()` except an error is returned if the system time equals or exceeds the time specified by `abstime` before the condition `cond` is signaled or broadcasted, or if the absolute time specified by `abstime` has already passed at the time of the call. When time-outs occurs the `mutex` is released and reacquired.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0DisposeCondition()` `a0NewCondition()` `a0BroadcastCondition()` `a0WaitCondition()` `a0SignalCondition()`

a0TryPSemaphore()

Tries to perform a P on a semaphore.

Syntax

```
a0tError   a0TryPSemaphore(
                                  a0tSemaphore   *semaphore
                                  int            *succeed );
```

Arguments

Name	rd/wr	Description
semaphore	write	Semaphore descriptor to operate.
succeed	write	Flag true if a **P** was possible.

Description

This routine tests if the counter associated with the semaphore is zero or negative and returns the result in `succeed`. The thread doesn't block, but when `succeed` is true the counter is decremented, as if a real **P** operation had succeeded.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewSemaphore()` `a0DisposeSemaphore()` `a0PSemaphore()` `a0VSemaphore()`

a0UnlockMutex()

Unlocks a mutex.

Syntax

```
a0tError   a0UnlockMutex(
                             a0tMutex   *mutex );
```

Arguments

Name	rd/wr	Description
mutex	write	Mutex descriptor to unlock.

Description

This routine unlocks the mutex pointed by `mutex`. If is there any blocked thread waiting to lock this mutex, one is unblocked.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrInvMutex`	The mutex descriptor is invalid.
`A0ErrOk`	Successful completion.

See Also

`a0NewMutex()` `a0DisposeMutex()` `a0LockMutex()`

a0Unpack()

Unpacks data from a buffer.

Syntax

```
a0tError  a0Unpack(
                    a0tBuffer     *buffer,
                    a0tDatatype   datatype,
                    void          *outbuf,
                    int           outcount );
```

Arguments

Name	rd/wr	Description
buffer	read	Buffer descriptor to unpack from.
datatype	read	Datatype of the data to unpack.
outbuf	write	Address to unpack to.
outcount	read	Number of elements to unpack.

Description

Unpacks from `buffer` the `outcount` occurances of `datatype` typed data and stores it in memory, starting at address `outbuf`. Successive unpacks extract data from the beginning of the buffer, until no more data exist.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrFailPack`	It was not possible to unpack all data.
`A0ErrOk`	Successful completion.

See Also

`a0Pack()` `a0NewBuffer()` `a0SendBuffer()` `a0ReceiveBuffer()` `a0ISendBuffer()` `a0IReceiveBuffer()` `a0_DATATYPES`

a0VSemaphore()

Performs V on a semaphore.

Syntax

```
a0tError   a0VSemaphore(
                              a0tSemaphore   *semaphore );
```

Arguments

Name	rd/wr	Description
semaphore	write	Semaphore descriptor to operate.

Description

This routine increments the counter associated with the semaphore and if there are blocked threads, one is unblocked.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0NewSemaphore()` `a0DisposeSemaphore()` `a0PSemaphore()` `a0TryPSemaphore()`

a0WaitAnyRequest()

Waits for any request of a table to complete.

Syntax

```
a0tError   a0WaitAnyRequest(
                                  a0tRequest   *requestab[],
                                  int          size,
                                  int          *index );
```

Arguments

Name	rd/wr	Description
requestab	write	Table of requests to wait for.
size	write	Size of the table.
index	write	Result of the wait.

Description

This function **blocks** the calling thread until one of the previous requests specified in the table `requestab` has already completed. The size of the table must be specified in `size`. After the completion of one of the requests, it can't be waited or tested anymore.

The table `requesttab` must be constructed by the caller thread with pointers to all the requests to be tested. After a completed request, the caller thread doesn't need to bother to rebuild the table to reuse it. The function `a0WaitAnyRequest()` replaces the pointer of the completed request by `NULL` and it doesn't complaint if a pointer is `NULL` in `requestab`. Though, a `NULL` pointer is also counted in `size`.

It returns in `index` the index of the completed request in the table. It also inserts in the respective completed request structure its final status (see `a0GetRequestData()`).

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0WaitRequest()` `a0TestAnyRequest()` `a0ISend()` `a0IReceive()` `a0ISendBuffer()` `a0IReceiveBuffer()`

a0WaitCondition()

blocks thread on condition variable `cond`**.**

Syntax

```
a0tError   a0TimedWaitCondition(
                                        a0tCondition   *cond,
                                        a0tmutex       *mutex);
```

Arguments

Name	rd/wr	Description
cond	read	Condition to be signaled.
mutex	read	Mutex associated to condition variable.

Description

This function blocks on a condition variable `cond` which atomically releases the mutex pointed by
mutex an causes the calling thread block on the condition variable pointed by `cond`. The blocked thread may be awakened by `aoSignalCondition`, `a0BroadcastCondition`, or interrupted by a UNIX signal.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0 DisposeCondition()` `a0NewCondition()` `a0BroadcastCondition()` `a0TimedWaitCondition()` `a0SignalCondition()`

a0WaitRequest()

Waits for a request to complete.

Syntax

```
a0tError   a0WaitRequest(
                                a0tRequest   *request );
```

Arguments

Name	rd/wr	Description
request	write	Request to wait to and hold the status.

Description

This function **blocks** the calling thread until the request specified in `request` is completed. After this call, the request it can't be waited or tested anymore.

It inserts in the `request` structure its final status (see `a0GetRequestData()`).

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrNullReq`	The request is `NULL`.
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

See Also

`a0ISend()` `a0IReceive()` `a0ISendBuffer()` `a0IReceiveBuffer()` `a0TestRequest()`

a0YieldThread()

Try to schedule another thread.

Syntax

```
a0tError  a0YieldThread(  void  );
```

Description

This function request the scheduler to choose a new thread in the same priority level of the current thread (which his the highest priority thread). If not, the current thread continues.

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0Err....`	Some error occurred.
`A0ErrOk`	Successful completion.

a0_DATATYPES

Data types known by Athapascan-0.

Syntax

```
a0tDatatype   a0_CHAR,
              a0_SHORT,
              a0_INT,
              a0_LONG,
              a0_LONG_LONG_INT,
              a0_UNSIGNED_CHAR,
              a0_UNSIGNED_SHORT,
              a0_UNSIGNED,
              a0_UNSIGNED_LONG,
              a0_FLOAT,
              a0_DOUBLE,
              a0_LONG_DOUBLE,
              a0_BYTE,
              a0_PORT;
```

Description

These variables define the basic data types recognized by Athapascan-0. They represent all the scalar types of the C language.

Error Codes

Every Athapascan-0 function returns `A0ErrOk` if no errors occured during its execution. If an error is found, one of the following codes is returned:

Return Values

If an error condition occurs, it returns the error code, otherwise it returns `A0ErrOk`. Possible values are as follows:

Error Code	Description
`A0ErrNotImplem`	Function not implemented.
`A0ErrNoMem`	System memory is exhausted.
`A0ErrFailPack`	No more data exist in the buffer to unpack or the buffer is already full to pack.
`A0ErrLockedMutex`	Mutex is still locked, it can't be disposed.
`A0ErrInvFormat`	Format passed is infalid.
`A0ErrInvMutex`	Mutex passed is invalid.
`A0ErrInvThread`	Thread passed is invalid.
`A0ErrInvPriority`	Priority passed is invlalid, at least for the concerned scheduling rule.
`A0ErrInvService`	Service passed is invalid.
`A0ErrInvBuffer`	Buffer passed is invalid.
`A0ErrInvTag`	Tag passed is invalid.
`A0ErrInvalid`	There is an invalid parameter.
`A0ErrInvBuffType`	Type of the buffer passed is invalid.
`A0ErrInvReq`	Request passed is invalid.
`A0ErrInvNode`	Node passed is invalid.
`A0ErrNullReq`	Request has already been waited or tested for.
`A0ErrFailSend`	Lower layer failed to send a message (User must abort properly).
`A0ErrFailRecv`	Lower layer failed to receive a message (User must abort properly).
`A0ErrFailComm`	Lower layer failed to communicate (User must abort properly).
`A0ErrOk`	Successful completion.