

Atacama Large Millimeter

ALMA-SW-NNNN

Revision: 1.0

2005-08-11

ACS Threading

Software Architecture and How-to manual

Bogdan Jeram (bjeram@eso.org),

ESO

Keywords:	
Author Signature:	Date:
Approved by:	Signature:
Institute:	Date:
Released by:	Signature:
Institute:	Date:

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
1.0	2005-08-12	Bogdan Jeram	All
	Created		
1.0.1	2005-12-12	Steve Harrington	All
	Edited for english/clarity		

Table of Contents

1 Introduction.....	4
1.1 Scope.....	4
1.2 Glossary.....	4
1.3 References.....	4
2 General.....	4
3 Architecture.....	5
4 ACS Thread.....	6
4.1 Thread body.....	6
4.2 Thread initialization and clean-up.....	7
4.3 Thread Functionality.....	8
5 ACS Thread Manager	9
5.1 Thread manager functionality.....	11
6 Examples.....	13

1 Introduction

1.1 Scope

This document describes how a user can create, destroy, and manipulate threads in C++ by using ACS.

1.2 Glossary

<http://www.alma.nrao.edu/development/computing/docs/joint/draft/Glossary.htm>

1.3 References

- [1] ALMA reviewed document 0005, **ALMA Common Software Technical Requirements**, G. Raffi, B. Glendenning, 2000-JUNE-05.
- [2] ALMA reviewed document 0016, **ALMA Common Software Architecture**, G.Chiozzi, B.Gustafsson, B.Jeram, 2001-SEPT-09.
- [3] **The ACE Programmers's Guide**, Stephen D. Huston, James CE Johnson, Umar Syyid

2 General

ACS has provided support for threads for some time, but in ACS 4.1 the support for threading was improved in many respects. The main aim for the changes was to simplify the effort necessary for a user to write a thread. Old ACS threads, also known as “baci threads”, were based on two classes: *baci::BACIThread* and *baci::BACIThreadManager*. With ACS 4.1, new threading classes were introduced.

This document concentrates on doing threads with the new classes that were introduced in ACS 4.1. While some of the old classes are still available, primarily for backwards compatibility, it is recommended that all new development use the new thread classes.

Here is a list of major improvements and changes in the threading support introduced in ACS 4.1:

- *baci::BACIThread* was renamed to *ACS::ThreadBase*, old name can be still used but is present only for backwards compatibility
- new class for threads was introduced: *ACS::Thread*
- *baci::BACIThreadManager* was renamed to *ACS::ThreadManagerBase*, old name can be still used for backwards compatibility

- new thread manager class was introduced: *ACS::ThreadManager*
- *baci::BACIThreadParameter* was renamed to *ACS::ThreadBaseParameter* and is no longer used with new thread class (*ACS::Thread*)
- threads can be used in all components not just in characteristic components
- threads are objects (threading is now object-oriented). This simplifies writing and dealing with threads.

3 Architecture

ACS threads are based on ACE threads 1.3.

There are two main actors in ACS threading (Figure 1):

- Thread: the actor which represents the thread
- Thread Manager: actor which groups threads together and manages them

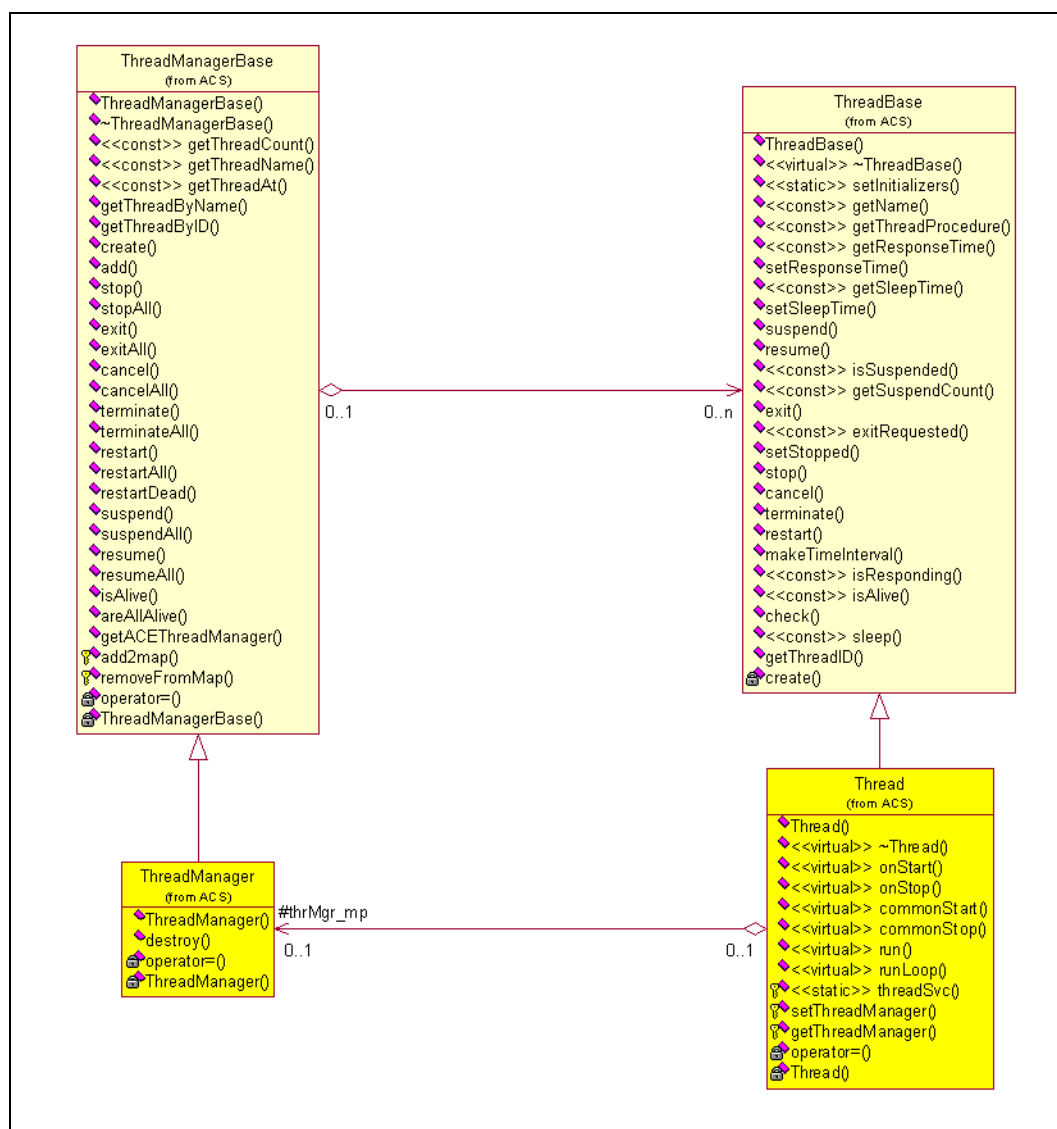


Figure 1 ACS Threading class diagram

4 ACS Thread

A thread is simply an object, where the body of the thread is a method of the (thread) class. So, to create a new thread one needs only to create an instance of a given thread class.

4.1 Thread body

The thread class must derive from *ACS::Thread* and must implement either the **run()** or **runLoop()** method. The first method (**run**) is executed just once, whereas the second method (**runLoop**) is executed periodically - it loops. In the **runLoop** case, the user does not need to implement a loop in the body of the thread; this functionality is built-in. The loop can be exited at any time by executing the **exit** or **stop** methods, resulting in thread termination. The looping frequency is defined with via the sleep parameter of the thread. The sleep value can be passed to the constructor of the *ACS::Thread* class. The default value is 100 msec.

A thread can be created so that it is executed immediately, or it can be created as a suspended thread. The way the thread will be created is specified by passing the **suspended** flag to the *ACS::Thread* constructor as the second parameter. By default the thread is executed immediately after its creation (i.e., the default value of the suspended flag is false).

The thread can be suspended or resumed at any time by invoking the suspend or resume methods of the thread object, respectively.

The constructor of a thread class must call the constructor of the parent class, *ACS::Thread*, passing the correct parameters. Here is the signature of *ACS::Thread*'s constructor:

```
ACS::Thread::Thread (const ACE_CString & name,
                    const bool suspended = false,
                    const TimeInterval & responseTime =
                        ThreadBase::defaultResponseTime,
                    const TimeInterval & sleepTime =
                        ThreadBase::defaultSleepTime,
                    bool del=false
                    )
```

- As we can see from the signature of the constructor, the only mandatory parameter is the thread *name*. The thread name can be any string, but it must be unique (per thread manager).
- The *suspended* flag can be passed as the second parameter, which denotes whether the thread will be executed immediately (false), or will be created in a suspended mode (true). If the suspended parameter is not sent, the default value is false, meaning that the thread will be executed immediately after its creation.
- The third parameter (*responseTime*) is also optional. It represents the thread heartbeat response time in 100ns units. The default value ([*ThreadBase::defaultResponseTime*](#)) is 1 second.
- The fourth parameter (*sleepTime*) that can be passed to the *ACS::Thread* constructor represents the sleep time of the thread in 100ns units. The default value (*ThreadBase::defaultSleepTime*) is 100 milliseconds.
- The fifth and final parameter (*del*) signals whether the thread object is deleted after the thread has executed. The default value is false, which means that by default the thread object is not deleted. The user should use this flag with caution, since the thread object can be safely deleted automatically only if it was created on the heap. That is the case when the thread manager is used for creating thread objects. When the thread object is to be deleted automatically after it exits (i.e. *del = true*), a detached thread is created instead of a joinable thread.

If the thread cannot be created, the constructor of *ACS::Thread* throws an ACS exception, so the user has to handle catching this exception when creating a thread.

In general the constructor's signature for a user's thread implementation class can be arbitrary, but in order to get the functionality described above one must make sure that the right parameters are passed to the base class: *ACS::Thread*. One or more parameters of the constructor can be parameter(s) that the user wants to send to the thread. If the user wants to use the thread manger to create threads, the signature of the constructor must obey some rules. This (signature) is described in the following sections.

4.2 Thread initialization and clean-up

Sometimes a user wants to do some work before the thread is actually created and to do some clean up just after the thread is finished. This can be done by implementing the *onStart()* and *onStop()* methods.

The thread initialization code has to be put in the *onStart()* method, which is executed just before the thread body is executed, i.e. before the *run()/runLoop()* method.

Similarly, the thread clean-up code has to be put in the *onStop()* method, which is executed just after the thread body is done. The thread base class - *ACS::Thread* itself already provides some common thread initialization/clean-up functionality, such as logging system initialization/clean-up. This common behavior can be changed by overriding the *commonStart()* and *commonStop()* methods. Overriding these methods should be done with caution and only when necessary. In most cases,

implementing the *onStart()* and/or *onStop()* methods while inheriting the default implementation of *commonStart/commonStop* should be sufficient.

4.3 Thread Functionality

The ACS::Thread class provides several methods for manipulating threads:

- *suspend*- suspends the execution of the thread. If the suspend function is not supported by the underlying thread implementation, suspend is simulated.
- *resume*- resumes (continues) the execution of a previously suspended thread.
- *terminate*- terminates the thread. Terminating means calling the *stop* method and if the thread has still not stopped the *cancel* method is called.
- *cancel*- cancels (forceful terminates) the thread. Using cancel for thread termination should be avoided.
- *stop* - stops the thread. Stopping means notifying the thread by calling the *exit* method and then waiting for a while for the thread to stop, i.e. checking until the thread transitions to the stopped state
- *exit* -notifies the thread to exit its thread worker function. The thread worker function should always exit when this notification is issued
- *restart* - restarts the thread. Restarting means to terminate the thread and recreate a new thread (i.e. calling *terminate* and *create* methods).
- *setSleepTime* - sets the sleep time of the thread in 100ns unit.
- *setResponseTime* - sets the heartbeat response time in 100ns unit.
- *setStopped* - sets the thread state to stopped. This function should be called in the thread worker function.
- *sleep* – Puts the thread to sleep for given amount of time in 100ns units. If 0 is given, then the default sleep time is used.
- *makeTimeInterval* – Updates the last heartbeat time.

There are several methods to get the status of a thread:

- *getName* - gets the name of the thread
- *isSuspended* - checks if the thread is suspended.
- *isAlive* - checks if the named thread is alive (not terminated/stopped).
- *isResponding* - checks if the named thread is responsive (has a heartbeat). Having a heartbeat means that *check* was called within the last *defaultResponseTime* time.
- *getResponseTime* - gets the heartbeat response time in 100ns unit
- *getSleepTime* -gets the sleep time of the thread in 100ns unit
- *getSuspendCount* - gets the number of suspend calls that have occurred on this thread.
- *getThreadID* - returns the ACE specific thread ID of the base thread
- *check* - checks the state of the thread and updates heartbeat.
- *exitRequested* – checks if thread has already received an exit request

Details of the methods, signatures, etc. can be found in the Doxygen page for *ACS::Thread(Base)*

5 ACS Thread Manager

A thread can be created standalone as a normal C++ object (i.e. outside the context of a thread manager, by using “new” to instantiate it). In this case the user is responsible for handling the entire life cycle of the thread. A better way to create threads from within an ACS component is to use the *ThreadManager* class. *ThreadManager* acts as a thread factory and it allows the user to stop, start, suspend, etc. any threads that have been created with the thread manager. When the thread manager is destroyed, all the threads that have been created by it are also destroyed. In other words, the thread manager can be used to:

- manage the life cycle of threads (creation and destruction)
- execute common commands on a group of threads

The thread manager is normally used for creating threads inside an ACS component. To retrieve the thread manager reference (i.e. its pointer) from a component:

```
getContainerServices()->getThreadManager()
```

The thread manager provides several *create* methods for creating a new thread. They all create a thread object on the heap and return a reference to it. The object is deleted automatically by the thread manager, but it can be deleted also explicitly by the user. The *create* methods are template functions of the **ACS::ThreadManager** class. They accept either one or two **template parameters**, depending on whether we want to create a thread with or without a parameter. The first template parameter specifies the class that implements the thread. The second template parameter is used to specify the parameter type that is passed to the constructor of the thread. The thread parameter can be, for example, a reference to the component.

So, the *create* method can look like either of the following forms:

```
thrMgr->create <ThreadClass>(...)
```

```
thrMgr->create <ThreadClass, ThreadParameter>(...)
```

Here is a list of the *create* methods for creating a thread without a parameter (template functions with just a single parameter T):

```

template<class T>
T* create ( const ACE_CString name)

template<class T>
T* create ( const ACE_CString name,
           const bool suspended)

template<class T>
T* create ( const ACE_CString name,
           const bool suspended,
           const TimeInterval &responseTime,
           const TimeInterval &sleepTime)

template<class T>
T* create ( const ACE_CString name,
           const bool suspended,
           const TimeInterval &responseTime,
           const TimeInterval &sleepTime,
           bool de)

```

Here is a list of the *create* methods for creating a thread with a parameter of type P (template functions with two parameters, T and P). The methods are identical to those for creating threads w/o a parameter, except that there is a reference to a parameter of type P as the second parameter.

```

template<class T, class P>
T * create (const ACE_CString name,
           P &param)

template<class T, class P>
T * create (const ACE_CString name,
           P &param,
           const bool suspended)

template<class T, class P>
T * create (const ACE_CString name,
           P &param,
           const bool suspended,
           const TimeInterval &responseTime,
           const TimeInterval &sleepTime)

template<class T, class P>
T * create (const ACE_CString name,
           P &param,
           const bool suspended,

```

```

const TimeInterval &responseTime,
const TimeInterval &sleepTime,          bool
de)

```

In order for a thread to be created using the thread manager, the signature of the constructor of the thread must match the signature of the *create* methods (at least one of them). The signature and parameters are the same as those in the constructor of the **ACS::Thread** class – the base class for threads. The thread manager just passes the parameters in the same order to the constructor of the thread object of type T.

A thread object can be created simply by invoking the create method on the thread manager. What is returned is a pointer to the thread object. The thread manager also takes care of deleting the threads that have been created by it. All threads created by the thread manager are automatically destroyed when the thread manager is destroyed.

In an ACS component all threads that have been created by the component's thread manager are (attempted to be) stopped in the methods `cleanUp()` or `aboutToAbort()` as part of the component's lifecycle, and are subsequently destroyed.

A thread can be destroyed explicitly by the user by simply deleting the thread object or by asking the thread manager to do this using the destroy method, passing a pointer to the thread object:

```
void destroy (ACS::Thread *thr)
```

5.1 Thread manager functionality

The thread manager provides a lot of functionality. Some of the functions are dedicated to do something on a single thread owned by the thread manager, while others can be used for doing something on all the threads belonging to the manager.

There are several ways to retrieve a thread from the thread manager:

- `const ACS::ThreadBase * getThreadByName (ACE_CString name)`
- `ACS::ThreadBase * getThreadByID (ACE_thread_t id)`

Actions on single thread owned by the thread manager:

- `bool add (const ACE_CString name, ThreadBase *acsBaseThread)` – explicitly adds a new thread (**acsBaseThread*) with name (*name*) to the thread manager. All threads that are created with the **create** method are automatically added to the thread manager.
- `bool stop (const ACE_CString name)` – stops the thread with name (*name*)

- **void [exit](#) (const ACE_CString name)** – exits the thread with name (*name*)
- **bool [cancel](#) (const ACE_CString name)** – cancels the thread with name (*name*)
- **bool [terminate](#) (const ACE_CString name)** – terminates the thread with name (*name*)
- **bool [restart](#) (const ACE_CString name)** – restarts the thread with name (*name*)
- **bool [suspend](#) (const ACE_CString name)** – suspends the thread with name (*name*)
- **bool [resume](#) (const ACE_CString name)** – resumes the suspended thread with name (*name*)
- **bool [isAlive](#) (const ACE_CString name)** – checks if the thread with name (*name*) is alive or not
- **void [destroy](#) (ACS::Thread *thr)** – explicitly destroys the thread with name (*name*). All threads created by the thread manager are automatically destroyed by the thread manager.

The actions which act on all threads that belong to the thread manager are:

- **int [getThreadCount](#) ()** – returns the number of threads managed by the thread manager
- **bool [stopAll](#) ()** – stops all the threads managed by the thread manager
- **void [exitAll](#) ()** – exits all the threads managed by the thread manager
- **bool [cancelAll](#) ()** – cancels all the threads managed by the thread manager
- **bool [terminateAll](#) ()** – terminates all the threads managed by the thread manager
- **bool [restartAll](#) ()** – restarts all the threads managed by the thread manager
- **bool [restartDead](#) ()** – restarts all the dead threads managed by the thread manager
- **bool [suspendAll](#) ()** – suspends all the threads managed by the thread manager
- **bool [resumeAll](#) ()** – resumes all the threads managed by the thread manager
- **bool [areAllAlive](#) ()** – checks if all the threads managed by the thread manager are alive

6 Examples