

Atacama Large Millimeter

ALMA-SW-NNNN

Revision: 1.4

2005-03-17

*Software
specification*

M. Plesko

ACS Supported BACI Types

Software specification

M. Plesko

J. Stefan Institute

G. Chiozzi

European Southern Observatory

M. Sekoranja

J. Stefan Institute

Keywords: ACS, BACI, CORBA

Author Signature:

Date:

Approved by:

Signature:

Institute:

Date:

Released by:

Signature:

Institute:

Date:

Table of Content

<u>1 Introduction.....</u>	<u>4</u>
1.1 Scope.....	4
1.2 Audience.....	4
1.3 Acronyms.....	4
1.4 References.....	4
1.5 Executive summary.....	5
<u>2 Concepts.....</u>	<u>5</u>
2.1 Design Goals of BACI.....	5
2.2 Definitions.....	7
2.3 Basic Types and Objects.....	7
<u>3 Applying BACI Patterns to ACS Property Data Types.....</u>	<u>12</u>
3.1 Callbacks.....	12
3.2 Event Sets and Alarms.....	14
3.3 Monitors.....	16
3.4 Properties.....	18
3.4.1 Methods and Attributes Common to All Properties.....	19
3.4.2 The Interface for Double Properties.....	20
3.4.3 The Interface for Ppattern.....	23
3.4.4 The interface for Pstring.....	24
<u>4 Solutions to Some Particular Requests for ACS.....</u>	<u>25</u>
4.1 Properties with Sequences.....	25
4.2 Properties with Structs.....	26
4.3 Monitor Timing.....	27
4.4 Generic Access to Characteristics.....	28

1 Introduction

1.1 Scope

This document specifies the CORBA interfaces for the Basic Control Interface (BACI) part of ACS. In order to be nearly self-sufficient, it repeats the main BACI concepts in simpler terms. In case of inconsistency, the original BACI specification [1] takes precedence.

1.2 Audience

This document is intended mainly for application programmers of clients or servers that need to know the implementation of BACI types for ACS. It also specifies some points, which are left by BACI to the implementation.

1.3 Acronyms

CORBA	Common Object Request Broker Architecture
BACI	Basic Control Interface
ACS	ALMA Control System
API	Application Programmer's Interface
IDL	Interface Definition Language
MACI	Management and Access Control Interface

1.4 References

1. ACS Basic Control Interface Specification
2. OMG Time Service Specification, formal/97-12-21.pdf
3. ACS Error System Architecture
4. Logging and Archiving
5. Abeans White Paper
6. BACI White Paper
7. Management and Control Interface Specification

1.5 Executive summary

BACI is meant to be a standardized interface so that applications and pieces of control systems can be hooked to it from either side. The BACI prescribes a model of Components that are commonly used in all control systems. It is the largest common denominator that can be found among different types of experimental facilities. As such it should enable portability of core control software and ultimately reduce the dispersed efforts at various laboratories where the same software is written all over and over.

BACI is but a definition of interface patterns with the use of IDL, not a definition of an application programmer's interface (API). A concrete API must first define interfaces for Component Properties. This is the task of the present document. The definition of different types of Components and which properties they contain is the task of people who know devices and beyond the scope of this document.

This document uses the following patterns described in the BACI specifications: callback, event set, monitor and property. It defines interfaces for these patterns on concrete data types such as double, long, string, etc. The result is an IDL module, whose components are building blocks for devices and which is implemented with a generic library. Programmers of concrete device servers use the IDL components to define their devices and use the generic library to efficiently write servers. All the common tasks of servers, such as handling and managing client requests, CORBA Object (COB) life-cycle management, dispatching monitor callbacks, initializing servant code, etc. is done by the generic libraries that are provided by ACS in the BACI and MACI (Management and Access Control Interface). The programmer only has to write the so called "business logic", i.e. the way a device behaves.

2 Concepts

Most of the discussion in this chapter is a repetition from other documents [1,6].

2.1 Design Goals of BACI

The design goals of BACI are:

- Rely on pure CORBA only: don't be language or system specific; don't assume extra functionality in an API library.
- Enforce strong type checking wherever possible. Illegal commands should be discovered already during compile time. Run-time parsing of commands through constructs like `send("command")` must be avoided. Generic applications can use the introspection capabilities of CORBA (e.g. Interface Repository, Dynamic Interface Invocation, etc.) instead.

- Exploit the object paradigm: the object itself is responsible to provide all data that are relevant to it. Avoid therefore direct access to database servers; leave this to the implementation.
- Don't try to define a generic interface for any possible control system. Specialize on the definition of experimental physics objects with the functionality that is common to all.
- Define object interfaces; don't prescribe their implementation and don't provide client-side functionality. The BACI is merely a hook to the underlying control system.
- Don't allow the client to manipulate control system behavior. Assume rather that reasonable default values are provided by the system managers through control system configuration tools.
- Use well-proven concepts from existing control systems.
- Base data transfer on asynchronous calls assuming that all client and server host operating systems are multithread capable as is necessary for GUI-based applications. Keep synchronous calls for compatibility with scripting and sequencing tools.
- Use only the core CORBA without CORBA services. Leave the use of services to the management and access control interface (MACI).
- Encourage site-specific additions through interface inheritance instead of providing generic bypasses to strong type checking. However, all interfaces that are defined in the BACI must be implemented at a given site, because client applications from other sources rely on them. Clients written on site can still use the added functionality without penalty.

Great care has been taken to be as close as possible to existing control system frameworks like EPICS, CDEV, TACO, DOOCS, etc. Many concepts were actually taken directly from one or several of those frameworks. A few examples:

- EPICS: each controlled device property has a set of standard characteristics
- CDEV: devices are objects that are a collection of properties
- TACO: a server manages the interface for one type of devices
- DOOCS: the object is keeping short term history data
- all: synchronous and asynchronous get/set calls are supported

2.2 Definitions

A **Component** is a CORBA object that corresponds to the model of a physical device, e.g. power supply, vacuum pump, current monitor, etc. The Component is the basic entity of the BACI, because it is the most natural concept for modeling physical entities.

Commands that are executed on a Component, like on, off or reset are referred to as methods of the Component. Each Component has a number of Component **Properties** that are controlled, e.g. electric current, status, position, etc. A Component also has a set of **Characteristics**, which are static data, usually - but not necessarily - read from a configuration database.

Properties, which are also defined as objects in the BACI, are referred to as IDL attributes of the Component. Properties are distinguished by type (integer, double, etc.) and by being read-only or read-write objects. Each such “property object” has specific characteristics, e.g. the value, the minimum, etc. The methods of a property allow to retrieve or modify these characteristics: get(), set(), min_value(), etc. While there are in principle an infinite number of Component types, one for each physical controlled device, there are very few different property types. It makes therefore sense to standardize those types, which is exactly the aim of this document.

Most of the device commands and property methods are executed asynchronously by the remote object. The results of the operations are communicated to the client by means of a **callback**. A callback is an object interface that must be implemented by the client, so that it can be invoked by the remote object. During this process, the remote object functions as a client and the client performs as a server.

A **Component servant** is a CORBA aware piece of code that implements one specific Component interface. A servant usually communicates with the hardware via VME, fieldbus, or similar. Note that devices of the same type, i.e. having the same IDL interface, can be exported by several different device servers, residing on different hosts. A typical example is a telescope complex with several identical but independent telescopes. The servant for “telescope1/motor” exports the motors of telescope 1, while the servant for “telescope2/motor” exports the motors of the other telescope. Both types of motors have the same IDL.

2.3 Basic Types and Objects

Device properties have one of the following primitive types: **long**, **double**, **pattern**, **enum**, **string**, where “**pattern**” stands for an unsigned long. The typedef pattern is used because this type will mostly be used to encode a pattern of status bits. Furthermore, “pattern” is a single word as opposed to “unsigned long”. The type pattern can be used also for raw binary data. Other primitive types are currently not supported for a single reason: simplicity. Given the CORBA overhead it makes no sense to save a byte or two by using short instead of integer. Likewise for float, which in addition holds too few decimal places than required for 16-bit precision. Should there be a compelling reason for other types, they can easily be added following the design patterns set forth in BACI. In the next generation of ACS we plan to use code generators, either from a special

template preprocessor or from UML models. Then, it will be much easier to add any new type, therefore it doesn't make much sense to spend the time now.

Several values of the same type are stored as a sequence, according to the IDL type definition, e.g.:

```
typedef sequence<double> doubleSeq;
```

Such sequences are already provided by the IDL syntax. Sequences are used when multiple devices are controlled with one method call or when a history of values of one property is requested. The use of sequences for individual values is possible but strongly discouraged, as properties are supposed to be simple objects related to one I/O channel.

The only Component properties with sequences of value types are longSeq (RO and RW), doubleSeq (RO and RW) and Seq (RO), because others have not been requested. But they can be added at any moment. The same argument as in the paragraph above holds here, too.

Each value read from the control system has an associated time-stamp. The time should be ideally represented by the CORBA time service through the UTO interface (which is not the POSIX time). As the time service is not yet part of most of the ORBs and the astronomers have other requirements on time, we use a different interface. We use the following definitions for Time and TimeInterval:

```
typedef unsigned long long Time;
```

where Time is the absolute time in 100 ns since 1582-10-15 00:00:00, see [2]. And

```
typedef unsigned long long TimeInterval;
```

where TimeInterval is used for the difference between two absolute time points.

For each operation (action, command, monitor or alarm event), the time stamps, errors and alarm codes are returned through the interface Completion, which is defined as:

```
struct Completion {

    unsigned long long timeStamp; // time stamp in 100th of ns

    ACSErr::ACSErrType type; // error type (group)

    ACSErr::ErrorCode code; // error code

    sequence<ACSErr::ErrorTrace, 1> previousError; // previos error(s) (error trace)

};
```

The completion structure is described in more detail in the BACI[1] and the ACS Error System Architecture [3] documents. Here, we will discuss the basic concepts. When there was no error, struct Completion contains the timestamp, the non-error code (ACSErrTypeOK) and type (ACSErrOK) and an empty previousError (error trace). BACI does not define values of the completion code, because it is open to particular implementations.

When there is an error, completion returns information that allows to pinpoint the reasons; previousError data structure contains detailed information about the error, either used to display or log the information - see the documents on error handling [3] and logging [4] for more.

Listing 1: Definition of ACSErrType extracted by acserr.idl:

```
#ifndef _ACSERR_IDL_
#define _ACSERR_IDL_

#pragma prefix "alma"

module ACSErr {

...

    typedef unsigned long ACSErrType;
    typedef unsigned long ErrorCode;

...

};

#endif
```

Error types and their corresponding codes are defined in XML files. For each error type, a unique XML file must be defined by the developer. The type ACSErrTypeMonitor is reserved for monitor callbacks that can be either time or value triggered.

Listing 2: The XML for ACSErrTypeMonitor (ACSErrTypeMonitor.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
  <Type xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ACSError.xsd" name="ACSErrTypeMonitor"
type="1" _prefix="alma">
  <Code name="ACSErrMonitorOnTimer" shortDescription="A regular timer monitor
triggered the event." description="A regular timer monitor triggered the event."/>
```

```

    <Code name="ACSErrMonitorOnValue" shortDescription="The value changed by a
specified amount or more." description="The value changed by a specified amount or
more."/>
</Type>

```

Code is automatically generated from the XML file.

Listing 3: Definition of ACS ACSErrTypeMonitor type.

```

#ifndef _ACSErrTypeMonitor_IDL_
#define _ACSErrTypeMonitor_IDL_

#include <acserr.idl>

#pragma prefix "alma"

module ACSErr {
    // type
    const ACSErr::ACSErrType ACSErrTypeMonitor = 1;
};

module ACSErrTypeMonitor {

    //ED: A regular timer monitor triggered the event.
    const ACSErr::ErrorCode ACSErrMonitorOnTimer = 0;
    //ED: The value changed by a specified amount or more.
    const ACSErr::ErrorCode ACSErrMonitorOnValue = 1;

    // exceptions:

}; //module

#endif

```

The type ACSErrTypeAlarm is reserved for alarm messages that originate from the property value.

Listing 4: Definition of ACS ACSErrTypeAlarm type.

```

#ifndef _ACSErrTypeAlarm_IDL_
#define _ACSErrTypeAlarm_IDL_

#include <acserr.idl>

#pragma prefix "alma"

```

```

module ACSErr {
    // type
    const ACSErr::ACSErrType ACSErrTypeAlarm = 2;
};

module ACSErrTypeAlarm {

    //ED: All alarm conditions have disappeared.
    const ACSErr::ErrorCode ACSErrAlarmCleared = 0;
    //ED: At least one alarm condition remains.
    const ACSErr::ErrorCode ACSErrAlarmChanged = 1;
    //ED: Value below alarm LoLo (includes hysteresis).
    const ACSErr::ErrorCode ACSErrAlarmLow = 2;
    //ED: Value above alarm HiHi (includes hysteresis).
    const ACSErr::ErrorCode ACSErrAlarmHigh = 3;
    //ED: An alarm on the status determined by software.
    const ACSErr::ErrorCode ACSErrAlarmSoftware = 4;
    //ED: An alarm on the status from the hardware.
    const ACSErr::ErrorCode ACSErrAlarmHardware = 5;

    // exceptions:

};

```

Alarms as defined above are notifications about the state of the controlled property. Alarms are asynchronous and completely independent events. They have nothing to do with errors that occur during execution flow. In case such errors occur, the type and code values indicate this error condition, and the variable called error may contain a pointer to another completion structure. The particular values of the type and code are specific to the server implementation.

The completion structure may have a pointer to yet another completion, and so on. We see that all those completion structures form a linked list in the recursive error sequence, which called error stack. Hence it is possible to return and analyze the full stack of errors in the method call sequence, as specified in [3]. This is usually done by the ACS libraries, either in the BACI server or in the Abeans [5] client libraries. The application programmer is relieved from this work. We will therefore not go into details here. If the Completion type and code indicate an error-free condition, the error stack must be empty.

Errors and alarms are returned through the completion structure and not via CORBA exceptions, because most of the method calls are asynchronous. Callbacks, which are executed on the client in a separate thread, cannot raise exceptions. For reasons of uniformity, also synchronously executed methods that deal with control processes, return the same completion structure.

If methods on remote objects (either Component or property) fail for unexpected reasons, in particular methods that do not even return a completion, such as calls to characteristics or invocation of monitors, then, and only then, exceptions are thrown. If the remote object can not be reached, then the local ORB raises a `CORBA::NO_IMPLEMENT` exception. If the remote object can not finish the request for whatever reason, it raises `CORBA::NO_RESOURCES`. These exceptions are unchecked, i.e. they are not predeclared in the signature of the method in the IDL file.

3 Applying BACI Patterns to ACS Property Data Types

This chapter describes the implementation of the patterns described in BACI with the concrete types defined for ACS.

3.1 Callbacks

Asynchronous notification of results can be implemented with CORBA in two ways: either through callbacks or with the CORBA event/notification services. The latter is very powerful and suited also for clients that are not multi-thread capable. The advantage of callbacks on the other side is that they are simpler to use and that callbacks are implemented by all existing ORBs. Callbacks furthermore allow for type checking during compile-time, while events are always of type `CORBA::any`. We therefore use callbacks.

Program execution flow when executing callbacks is reversed to what we are used to. After the device server has finished executing a requested command, it becomes a *client* and requests execution of a callback method on the client, which for the time of this method call becomes a *server*. Two things have to be considered by the client developer:

- The client has to first create a callback object, then it invokes the remote asynchronous command, giving the reference to the callback object as one of the parameters of the call
- The callback is executed on request of the device server, completely independent of the normal execution flow on the client. The callback is therefore running in a separate thread. The thread itself is created and managed by CORBA, so there is no work for the client programmer.

BACI also prescribes the pattern, how asynchronous requests must be done:

```
request(par1, par2,...,in CB<type> cb, in CBDescIn desc);
```

We see, that when the client passes the callback to the server, it must accompany it with the `CBDescIn` structure. And both are always at the end of the parameter list. The callback object `CB<type>` is explained in detail below. The parameter `desc` (for descriptor) allows the client to describe some attributes of the callback object:

```
struct CBDescIn{
    TimeInterval normal_timeout;
```

```

TimeInterval negotiable_timeout; // not used in ACS,
Tag id_tag; // Provided by Client to uniquely tag the incoming callback call.
};

```

The TimeInterval normal_timeout is sent by the client to the server to inform it that it expects a reply in the normal_timeout period, before it will raise a timeout error condition. The server must complete the operation or send a notification that the operation is still in progress. The Tag id_tag is discussed with the CBDescOut structure.

For clients written in Java, the ABeans [5] library conveniently does all the work, so that the programmer doesn't even have to know CORBA and even less all those callbacks and descriptors.

The callback interface for simple types is illustrated with these two examples:

```

interface CBvoid: Callback{
    oneway void working(in Completion c, in CBDescOut desc);
    oneway void done(in Completion c, in CBDescOut desc);
};
interface CBdouble: Callback{
    oneway void working(in double value, in Completion c, in CBDescOut desc);
    oneway void done(in double value, in Completion c, in CBDescOut desc);
};

```

All callback methods are of type oneway, which means that the invoker (the device server in this case) does not have to wait for the callback methods to finish. The reason for this is that we do not want to influence the performance of device servers if client code is written badly.

The callback CBvoid does not return any value, only the time stamp and possible errors in the completion structure. It is used mainly to return the completion after an asynchronous command has been executed or to act as an event without data. Since BACI takes extreme care to be type-safe, there is one callback class for each data type that is transferred. The above example of CBdouble is for values of type double. Callbacks for other types are equivalent; just replace all three occurrences of "double" with the appropriate primitive type.

The CBDescOut type defines a structure that is a descriptor of the callback with information from the server:

```

struct CBDescOut{
    TimeInterval estimated_timeout;
    Tag id_tag;
};

```

The parameter estimated_timeout contains the estimated time for the callback to be returned. This value can be used by the client for timeout handling – if the callback is not received within this time plus some safety margin, the client can assume that there was a problem on the server side.

The `id_tag` is a numeric index (the Tag data type is nothing but an unsigned long), which has been provided by the client in the `CBDescIn` structure, when making the asynchronous request. This can be used by the client to distinguish callbacks from different requests.

When the callback is used for asynchronous notification in response to some action, the `done` method must be invoked when the action terminates, either with error condition or success. When the client processes the `done` invocation, it may discard the callback. If the action is time consuming, i.e. it cannot be completed before the `normal_timeout` parameter, the server must issue a working notification periodically (the server works under presupposition that each invocation resets the client's timeout timer to the `normal_timeout` period). The client must not discard the callback before `done` notification is called or one of the notifications timeout on the client side. See the BACI specifications for a detailed discussion on timeouts.

In normal cases, a “single-shot” command invokes the `done` method of the callback, while monitors regularly call `working`.

3.2 Event Sets and Alarms

The BACI specification does not explicitly define alarms. BACI specifies more general ways of user notifications, called events. Any multiple asynchronous notification that can be subscribed to is an event. Most of event types will be defined by device server programmers, who know what kind of events happen for a device and its properties. It is not the task of ACS to prescribe device events. The exception are alarms, which are a very standard kind of events. We can therefore define alarms as BACI-type events already for ACS. Alarms are passed asynchronously as usual through callbacks, which correspond to the type of the property value. As always, a callback descriptor has to be given in the parameter list. As also alarms must be type-safe, ACS foresees several alarms, one for each property type.

For each value type there is an Alarm event set. They all follow the same design pattern: Each alarm event set contains two events, `alarm_raised` and `alarm_cleared`. If the reason for an alarm changes, then `alarm_raised` can be invoked again, sending the new value and/or the new alarm code. It is not necessary that a new `alarm_raised` event should occur only after the alarm has been cleared. Each event is a callback method call that returns the value that is responsible for the event, a completion structure that contains the reasons for the alarm and a `CBDescOut` structure, which is returned with all callbacks.

```
interface Alarmpattern : Callback{
    oneway void alarm_raised(in pattern value, in Completion c, in CBDescOut desc);
    oneway void alarm_cleared(in pattern value, in Completion c, in CBDescOut desc);
};
```

```
interface Alarmdouble : Callback{
    oneway void alarm_raised(in double value, in Completion c, in CBDescOut desc);
    oneway void alarm_cleared(in double value, in Completion c, in CBDescOut desc);
```

```
};

interface Alarmlong : Callback{
    oneway void alarm_raised(in long value, in Completion c, in CBDescOut desc);
    oneway void alarm_cleared(in long value, in Completion c, in CBDescOut desc);
};
```

```
interface Alarmstring : Callback{
    oneway void alarm_raised(in string value, in Completion c, in CBDescOut desc);
    oneway void alarm_cleared(in string value, in Completion c, in CBDescOut desc);
};
```

There are no alarm types for sequences of values, because the alarm can only be triggered by a single value.

A process subscribes to alarms via a method that is part of all RO (read-only) properties. Following is the code from the ROdouble property; for other types all occurrences of double have to be replaced correspondingly:

```
Subscription new_subscription_Alarmdouble(in Alarmdouble cb, in CBDescIn desc);
```

Only properties of type RO have alarms, as RW properties should not even be able to set values in the alarm ranges. The method returns a Subscription object, which has to be used to destroy the subscription to the alarm events. The Subscription object is the event source interface. It is also the superclass for monitors (seen next section).

```
interface Subscription{
// temporarily suspends dissemination of event callbacks.
    void suspend();
// resumes dissemination of callbacks. Ignored if no previous suspend occurred.
    void resume();
// stops dissemination of event callbacks and releases all resources.
    void destroy();
};
```

A very important feature of alarms is that an alarm event is sent immediately after the client has subscribed to the alarm and when a server reconnects to existing client subscriptions after a restart. The event is used to notify the client of the current alarm status. If everything is OK, the alarm_cleared method is called, otherwise the alarm_raised method returns the code for the current alarm. The reason for this requirement is that when a client subscribes freshly to an alarm, it does not know, whether the property is in normal or alarm state. As alarm events are sent only upon changes of alarm conditions, the client might never know that the property has already raised an alarm and that further operations on the property or its device should be restricted.

This requirement is particularly useful in two cases:

- A client crashes and has to be restarted: the new client will immediately know the correct state. In object oriented speak, the alarm state is replicated as it were persistent.
- A server restarts after some downtime: while the server was down, the alarm condition might have changed. And the client would never know about it, so it must be get the actual status.

The above discussion is not necessary for monitors, because monitor callbacks are sent repeatedly and even if a few are lost due to client or server downtime, the actual situation is always restored with the new arriving callback.

Alarms can occur for any reason, which are mostly related to hardware. However, for values types that can be defined in intervals, such as double and long, most alarms will come from values that exceed some safety limits. For those value types, a RO-property contains four characteristics that define the limits for alarms:

```
readonly attribute double alarm_low_on; // below this value alarm is set
readonly attribute double alarm_low_off; // above this value alarm is cleared
readonly attribute double alarm_high_on; // above this value alarm is set
readonly attribute double alarm_high_off; // below this value alarm is cleared
```

There are two values for each interval boundary, in order to have hysteresis that prevents from constant triggering of alarms due to noisy signal levels.

3.3 Monitors

A client will often need the get the value of a property on a regular basis, either at given time intervals or whenever the value changes. A regular callback with the updated value is invoked by means of a monitor. The client creates a remote monitor on the server by calling one of the two methods of a property (again, the example is from the Pdouble property):

```
Monitordouble create_monitor(in CBdouble cb, in CBDescIn desc);
Monitordouble create_postponed_monitor(in Time start_time, in CBdouble cb, in
CBDescIn desc);
```

The first method starts continuous monitoring (with default time interval). As in all asynchronous requests, a callback object `cb` and its descriptor `desc` have to be provided. See callbacks for more details. The second method registers a monitor whose beginning will be postponed until the specified `start_time`, which is given in absolute time. Once it starts to regularly send callbacks, it is undistinguishable from a normal monitor.

As we are already used to, also the use of monitors are strongly typed. Each property has a different signature for `create_monitor`, depending on the type of callback object it is given to. Monitors, however, are not typed, because a monitor is only managing the dissemination of callbacks and has nothing to do with the type:

```
interface Monitor: Subscription{
```

```

void set_timer_trigger(in TimeInterval timer);
void get_timer_trigger(out TimeInterval timer);
readonly attribute Time start_time;
};

```

Monitors start with a default repetition rate, which can be obtained via the property characteristic `default_timer_trigger` (see next section). A client can request different monitor rates with the method `set_timer_trigger()`, which sets the requested delta time between two consecutive monitor callbacks. In order to protect network bandwidth, there is also an absolute minimal time, given by the property characteristic `min_timer_trigger`. A client can not request timer triggers lower than that value. The timer trigger can be disabled by passing the value 0 for timer parameter. That means that monitors will be sent only if a value changes significantly and the value trigger is set (see below). Invalid values (out-of-limits) are treated as valid extremes (a time interval out-of-limits defined by the server is interpreted as maximum allowed time interval).

Note that the `TimeInterval` timer is an approximate time between two callbacks. The real time between two callbacks can be more, as real-time performance cannot be guaranteed. The `TimeInterval` is related to the monitor, not to the property, i.e. two clients, or even the same client can request monitors on a given property at different rates.

The method `get_timer_trigger` returns the current `TimeInterval` between two consecutive monitor callbacks. An interesting attribute for postponed monitors is the `readonly` attribute `Time start_time`, which returns the absolute time of the first monitor callback that will occur. This can be used to check for erroneous inputs to `create_postponed_monitor`. In case of normal monitors, `start_time` holds the absolute time of the first callback sent by the monitor.

As the monitor interface inherits from subscription, a monitor can also be suspended and resumed. When a client does not want to receive monitor event any more, it calls the `destroy()` method on the monitor. It then gets one more callback, with the method `done()` instead of `working()` – as described in the section on callbacks. After the reception of `done()`, the callback object can be safely destroyed, because there will be no monitor callbacks anymore.

Apart from the usual timer triggers, which trigger a callback at regular intervals, there are also variable triggers for the two numeric property types `double` and `long`. Those can be customized to trigger callbacks whenever a value changes by a certain amount. This amount is set and retrieved via similar methods as for the timer trigger, as can be seen in the following IDL code:

```

interface Monitordouble: ACS::Monitor{
  void set_value_trigger(in double delta, in boolean enable);
  void get_value_trigger(out double delta, out boolean enable);
};

```

On creation of a monitor, the only trigger present will be the timer trigger. Calling the `set_value_trigger` method determines the behaviour of the value trigger. The `enable` parameter determines whether the value trigger is active or not. Value 0 assigned to

value trigger means that a notification should be sent on every change of the monitored value. This is very dangerous as it could flood the network. Therefore, there is a characteristic of the property, similar to `min_timer_trigger`, called `min_delta_trigger`, which specifies the minimum allowed threshold for value change. Invalid values (out-of-limits) are treated as valid extremes (a delta trigger below `min_delta_trigger` is treated as `min_delta_trigger`).

The interval at which the callback is invoked by the monitor is determined by the default setup for the specific property, which is typically in the order of one or several second. The ACS cannot guarantee, however, that the requested time intervals are met by the control system. The given values correspond just to approximate orders of magnitude. It can happen furthermore that not all control systems are able to respond in the order of milliseconds. The values should be therefore taken as a lower limit for the time between two consecutive monitor callbacks.

3.4 Properties

Having defined all concepts, we can now put them together into the most important interface of BACI, the property. The BACI specifications define only the core method of the root property interface, meaningfully called `Property`. For ACS, we have to define concrete properties that will be used to control and supervise all points of ALMA. In order to keep the interface definitions reasonably simple, we have kept the number of different property types small.

Properties are the basic entities that are manipulated by the control system. For a discussion on the meaning of properties and their relation to Components see the BACI white paper[6]. Two basic flavors of properties exist: RO = read-only (e.g. encoder position, device status) and RW = read-write (e.g. power supply voltage, pointing coordinate). In addition, properties have a certain type - either a double, long, pattern or string.

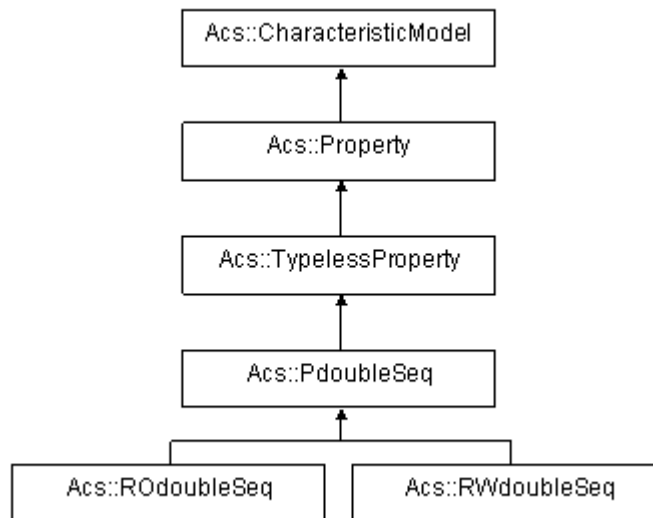


Figure 1: Inheritance diagram for *Acs::PdoubleSeq*

For the normal user of ACS, the object hierarchy of properties has little meaning (see figure 1). It may be confusing at best, because the only properties that a Component actually contains, are of the type RO and RW. However, the hierarchy is very important for the code generator (not yet provided for ACS) and to account for all BACI concepts, which include groups of Components and others. In this document we will simply state and explain all methods that all Properties have and then jump directly to the presentation of RO and RW properties.

3.4.1 Methods and Attributes Common to All Properties

The following methods and attributes are common to all properties, because they come from the interfaces higher up the inheritance hierarchy (CharacteristicModel, Property, TypelessProperty). They can be common, because they do not depend on the type of the properite value. A normal client would use only those in italics. The others are mainly used by generic clients, like the Object Explorer, and the Abeans libraries that provide Java Beans components for fast client development.

```
readonly attribute string name; // The fully qualified name of the Property
```

```
//generic access to characteristics, even those that are not declared in the IDL
any get_characteristic_by_name(in string name) raises (NoSuchCharacteristic);
```

```
//returns a sequence of characteristic names that match the regular expressions
//reg_exp. It returns a sequence of length 0 if no match is found.
stringSeq find_characteristic(in string reg_exp);
```

```
// The name of the parent Component
readonly attribute string characteristic_component_name;
```

```
//Returns a PropertySet object containing all characteristics of the Property.
CosPropertyService::PropertySet get_all_characteristics();
```

```
readonly attribute string description; // the description of the Property
readonly attribute string format; // the format for printf in C-syntax
readonly attribute string units; // the units of the Property
readonly attribute pattern resolution; // a bitpattern of significant bits
```

Some more explanations on the definitions: The main BACI components, such as Component and Property are named: they have a name, which the manager (see [7]) can convert to object reference if the client has the correct access level (only for Component). A Property can not exist on its own - it is always a read-only attribute of a Component and thus knows the name of it. It does not keep an object reference to the Component for security reasons. The attribute pattern resolution is actually a bit pattern representing the significant bits of the word carrying the value. It is useful also for returning the resolution of analog-digital converters in case of numeric properties.

3.4.2 The Interface for Double Properties

Here comes the part of properties which is type and access specific. They all follow the same patterns. For each type, there are the interfaces P<type>, the base class of a Property, which subclasses RO<type>, a read-only version and RW<type>, a read-write version of the Property class. Additional patterns are for Properties that return sequences of values, i.e. P<type>Seq, RO<type>Seq and RW<type>Seq, which are discussed in section 4.1 of this document.

The following types are missing from the IDL for ACS, because they were not requested: PpatternSeq and PstringSeq with their subclasses. In the next version of ACS, there will probably be a generator, that will create any IDL, code and config skeleton just from design pattern definitions in Rational Rose. Therefore it does not make sense to manually define and implement all those property types already now.

Note that all Property patterns are type safe, i.e. they return the explicit type of the property value and its characteristics - no casting or use of type any is required. Even the callbacks and in some cases the Monitors are of the correct type, according to the design patterns. Actually, to get the interface for Plong, one just has to replace all occurrences of double in the following IDL definition:

Therefore, only the double type occurrence of those patterns is documented in detail, others are equivalent, unless the difference is explicitly described in the following subsections. All the explanations are part of the comments of the interfaces, because it is rather difficult to write a concise description in a separate text. The IDL code is in boldface, the comments in plain text format.

```
interface Pdouble: TypelessProperty{
```

```
/*
```

```
    Synchronous call that returns the value of the property in the correct type, no casting is required.
```

```
*/
```

```

double get_sync(
    out Completion c // The Completion structure of the request is returned as out
parameter
);
/*
    Request one value through callback. The callback cb and its descriptor desc have to be
created by the client.
    The callback is of type CBdouble, because it explicitly carries a value of type double.
*/
void get_async(in CBdouble cb, in CBDescIn desc);
/*
    Read n_last_values or all values (whichever is less) from the server's local archive. If
n_last_values
    is 0, then read complete local archive. Method returns number of actually transmitted
values.
*/
long get_history(
    in long n_last_values, // The number of last monitored values to return
    out doubleSeq vs, // a sequence returning the requested values in ascending
temporal order
    out TimeSeq ts // a sequence returning the timestamps corresponding to the returned
values
);
/*
    Start continuous monitoring (with default time interval). As in all asynchronous
requests, a callback
    object cb and its descriptor desc have to be provided. See callbacks for more details.
*/
Monitordouble create_monitor(in CBdouble cb, in CBDescIn desc);
/*
    Register a monitor whose beginning will be postponed until the specified start_time,
which is given
    in absolute time. See create_monitor for more details on monitors.
*/
Monitordouble create_postponed_monitor(in Time start_time, in CBdouble cb, in
CBDescIn desc);
    readonly attribute TimeInterval default_timer_trigger; // default time interval
between two consecutive monitor callbacks
    readonly attribute TimeInterval min_timer_trigger; // the minimum allowed time
between two consecutive time triggers
    readonly attribute double min_delta_trigger; // the minimum allowed threshold for
value change in value triggers
    readonly attribute double default_value; // default value, error-free value, etc.
    readonly attribute double graph_min; // the recommended minimum for charts and
gauges that display the value

```

```

readonly attribute double graph_max; // the recommended maximum for charts and
gauges that display the value
readonly attribute double min_step; // smallest incremental step over whole range
};

```

A pedantic reader might notice that two attributes, `default_timer_trigger` and `min_timer_trigger`, are the only ones that are not dependent on the type `double`. A nitpicking might even comment, that as such they should belong to the `TypelessProperty` interface. However, there is no code to implement them in this type. They really belong to the interface, which has the monitor factory (i.e. `create_monitor`), as they are related to the corresponding implementation code. The monitor factories can also not be moved, since `create_monitor` methods are different per each property type.

The following interface is already known to us, all its pieces have been shown in section 3.2 on alarms: it request a subscription to the alarm event set for this property. Only properties of type `RO` have alarms, as `RW` properties should not even be able to set values in the alarm ranges.

```

interface ROdouble: Pdouble{
  Subscription new_subscription_Alarmdouble(in Alarmdouble cb, in CBDescIn
desc);
  readonly attribute double alarm_low_on; // below this value alarm is set
  readonly attribute double alarm_low_off; // above this value alarm is cleared
  readonly attribute double alarm_high_on; //below this value alarm is cleared
  readonly attribute double alarm_high_off; // above this value alarm is set
};

```

The read-write interface is somewhat more complex, because of the added degree of freedom, namely setting the value of the property:

```

interface RWdouble: Pdouble{
  /*
  Synchronous setting of the property value - the returned Completion structure informs of
  the success/failure of the operation.
  */
  Completion set_sync(in double value);
  /*
  Asynchronous setting of the property value - it is equivalent to the synchronous case,
  only that the Completion structure informing of the success/failure is returned via the
  callback object cb. As it does not carry a value, the callback is of type CBvoid. As usual,
  a callback descriptor is at the end of the signature.
  */
  void set_async(in double value, in CBvoid cb, in CBDescIn desc);
  /*

```

Method for fast consecutive sets that minimizes communication overhead. No assurance exists that the command has actually arrived to the server. Therefore also no completion code is returned.

```
*/
```

```
void set_nonblocking(in double value);
```

```
/*
```

Increment the raw value by one bit. This method can be used for fine-tuning actuators controlled by the property.

```
*/
```

```
void increment(in CBvoid cb, in CBDescIn desc);
```

```
/*
```

Decrement the raw value by one bit. This method can be used for fine-tuning actuators controlled by the property.

```
*/
```

```
void decrement(in CBvoid cb, in CBDescIn desc);
```

```
readonly attribute long min_value; // minimal allowed value for set
```

```
readonly attribute long max_value; // maximal allowed value for set
```

```
};
```

3.4.3 The Interface for Ppattern

The pattern Property has no monitor with value trigger, as a difference of values makes no sense. However, it has three other attributes that provide sequences that describe the meaning of each bit in the pattern: bitDescription, whenSet and whenCleared. The latter two keep one value of type condition per bit, which is an enum that describes the meaning of a bit in the status being set (to one) or cleared (to zero). The values correspond to usual colors of status LEDs:, which describe possible conditions of a physical device or its state. GREY corresponds to the LED being off.

```
enum Condition {
  RED,
  YELLOW,
  GREEN,
  GREY
};
```

The rest of the interface is equivalent to the Pdouble property:

```
interface Ppattern: TypelessProperty{
  pattern get_sync(out Completion c);
  void get_async(in CBpattern cb, in CBDescIn desc);
  long get_history(in long n_last_values, out patternSeq vs, out TimeSeq ts);
  Monitor create_monitor(in CBpattern cb, in CBDescIn desc);
  Monitor create_postponed_monitor(in Time start_time, in CBpattern cb, in CBDescIn desc);
  readonly attribute TimeInterval default_timer_trigger;
  readonly attribute TimeInterval min_timer_trigger;
  readonly attribute pattern default_value;
  readonly attribute stringSeq bitDescription; // one description for each bit
```

```
    readonly attribute ConditionSeq whenSet; // for each bit, defines the colour of LED
when bit is set
    readonly attribute ConditionSeq whenCleared; // for each bit, defines the colour of LED
when bit is cleared
};
```

3.4.4 The interface for Pstring

The string Property has no monitor with value trigger, as a difference of values makes no sense. The rest of the interface is equivalent to the Pdouble property, as can be seen in the following IDL code:

```
interface Pstring : TypelessProperty{
    string get_sync(out Completion c);
    void get_async(in CBstring cb, in CBDescIn desc);
    long get_history(in long n_last_values, out stringSeq vs, out TimeSeq ts);
    Monitor create_monitor(in CBstring cb, in CBDescIn desc);
    Monitor create_postponed_monitor(in Time start_time, in CBstring cb, in CBDescIn
desc);
    readonly attribute TimeInterval default_timer_trigger;
    readonly attribute TimeInterval min_timer_trigger;
    readonly attribute string default_value;
};
```

4 Solutions to Some Particular Requests for ACS

During the discussions, some particular requests emerged, which are addressed explicitly in the following sections. They have all been already taken care of in the IDL (see online documentation) and also indirectly mentioned in all previous chapters and sections.

4.1 Properties with Sequences

Sequences are very easy to add: keep the Property exactly the same as for scalar types, including all characteristics. Wherever a value is returned in a scalar type, return a sequence. It is simplest to explain with an example. This is all the IDL that is related to ROdoubleSeq and RWdoubleSeq (the syntax related to a sequence of doubles is highlighted with bold typeface):

```
typedef sequence<doubleSeq> doubleSeqSeq;

interface CBdoubleSeq: Callback{
    oneway void working(in doubleSeq value, in Completion c, in CBDescOut desc);
    oneway void done(in doubleSeq value, in Completion c, in CBDescOut desc);
};

interface PdoubleSeq: TypelessProperty{
    doubleSeq get_sync(out Completion c);
    void get_async(in CBdoubleSeq cb, in CBDescIn desc);
    long get_history(in long n_last_values, out doubleSeqSeq vs, out TimeSeq ts);
    Monitordouble create_monitor(in CBdoubleSeq cb, in CBDescIn desc);
    Monitordouble create_postponed_monitor(in Time start_time, in CBdoubleSeq cb, in
    CBDescIn desc);
    readonly attribute TimeInterval default_timer_trigger;
    readonly attribute TimeInterval min_timer_trigger;
    readonly attribute double min_delta_trigger;
    readonly attribute double default_value;
    readonly attribute double graph_min;
    readonly attribute double graph_max;
    readonly attribute double min_step;
};

/* Yes, the ROdoubleSeq contains no items related to sequences. The Alarm is always
triggered for a single value exceeding the limits, therefore the scalar Alarmdouble event
is fired */
interface ROdoubleSeq: PdoubleSeq{
    Subscription new_subscription_Alarmdouble(in Alarmdouble cb, in CBDescIn desc);
    readonly attribute double alarm_low_on;
    readonly attribute double alarm_low_off;
    readonly attribute double alarm_high_on;
```

```
    readonly attribute double alarm_high_off;
};
```

```
interface RWdoubleSeq: PdoubleSeq{
    Completion set_sync(in doubleSeq value);
    void set_async(in doubleSeq value, in CBvoid cb, in CBDescIn desc);
    void set_nonblocking(in doubleSeq value);
    void increment(in CBvoid cb, in CBDescIn desc);
    void decrement(in CBvoid cb, in CBDescIn desc);
    readonly attribute double min_value;
    readonly attribute double max_value;
};
```

4.2 Properties with Structs

WARNING: the following section has not been implemented yet. It is added merely for completeness, as we have just discussed the properties with sequences.

A similar approach as above is taken for properties with structs. Here, the main dilemma is how to display the structure, i.e. the interpretation of formats and units, which is already defined in the interface `TypelessProperty`, which is a superclass of all properties. We propose to keep the `TypelessProperty` superclass as it is, and define the following interpretations of its attributes:

- `format`: a C format string for the display of the **complete** struct with `printf`
- `units`: a string with units of all members in the struct, separated by commas; if a member has no units, then no character is used
- `resolution`: this attribute is ignored for structs

where the variables `format` and `units` have references to members of the struct in the **same** order as they appear in the IDL (which may not be the same for a given language binding).

Note that the monitor object used for struct properties is the base monitor interface. Then the IDL for struct properties looks rather straightforward, as we show in the example for `ROmyStruct` and `RWmyStruct` (the syntax related to the struct is highlighted with bold typeface), where the example struct is composed of an integer and a double:

```
typedef struct {int a; double b} myStruct;
```

```
typedef sequence<myStruct> myStructSeq;
```

```
interface CBmyStruct: Callback{
    oneway void working(in myStruct value, in Completion c, in CBDescOut desc);
    oneway void done(in myStruct value, in Completion c, in CBDescOut desc);
};
```

```

interface AlarmmyStruct : Callback {
    oneway void alarm_raised(in myStruct value, in Completion c, in CBDescOut desc);
    oneway void alarm_cleared(in myStruct value, in Completion c, in CBDescOut desc);
};

interface PmyStruct: TypelessProperty{
    myStruct get_sync(out Completion c);
    void get_async(in CBmyStruct cb, in CBDescIn desc);
    long get_history(in long n_last_values, out myStructSeq vs, out TimeSeq ts);
    //use only the base Monitor object
    Monitor create_monitor(in CBmyStruct cb, in CBDescIn desc);
    Monitor create_postponed_monitor(in Time start_time, in CBmyStruct cb, in CBDescIn
desc);
    readonly attribute TimeInterval default_timer_trigger;
    readonly attribute TimeInterval min_timer_trigger;
    readonly attribute myStruct default_value;
    // yes, no characteristics for display limits
};

/* Yes, the ROmyStruct contains no alarm limits, but in case of alarm the structured
AlarmmyStruct event is fired */
interface ROmyStruct: PmyStruct{
    Subscription new_subscription_AlarmmyStruct(in AlarmmyStruct cb, in CBDescIn
desc);
};

interface RWmyStruct: PmyStruct{
    Completion set_sync(in myStruct value);
    void set_async(in myStruct value, in CBvoid cb, in CBDescIn desc);
    void set_nonblocking(in myStruct value);
    // no inc/dec and no set limits; makes no sense for structs
};

```

Although those two cases for sequenced and structured properties look rather straightforward, it has to be noted that they each form a special case in the IDL generating templates used by “meta IDL”, i.e. the templates for primitive typed properties can not be used for complex typed properties.

4.3 Monitor Timing

ALMA has a central timing system for all hardware. Therefore it is interesting to be able to register a monitor at any time but to postpone the first callback to a predefined time. This requires that BACI allows to send an absolute time to the Component at monitor creation. It is simplest to just add another method to the property:

```

interface P<type>: TypelessProperty{

```

```

...
    Monitor<type> create_postponed_monitor(in Time start_time, in CB<type> cb, in
    CBDescIn desc);
...
}

```

where `start_time` corresponds to an absolute time such that the monitor callback will not be sent before this time. It can generally not be guaranteed that the callback is sent exactly at the specified time. It is the responsibility of the user to specify a reasonable start time. If the specified time is smaller than the actual time, the method is equivalent to the conventional `create_monitor()`.

This new postponed monitor has been already included in the example sequence and struct properties in section 3 of this document.

In order to be able to check the requested start time, the monitor interface is extended:

```

interface Monitor: Subscription{
    void set_timer_trigger(in TimeInterval timer);
    void get_timer_trigger(out TimeInterval timer);
    readonly attribute Time start_time;
};

```

In case of postponed monitors, which use the same interface, `start_time` returns the absolute time of the first scheduled callback. The interpretation of `start_time` for conventional monitors is that it returns the absolute time of the first monitor callback that has been invoked.

4.4 Generic Access to Characteristics

At ALMA there will be some hardware and low level device characteristics in the configuration database that will never appear in the IDL, interesting only for the driver. However, some clients may want to access this information. For such cases a generic access to characteristics is required. One way would be through the `PropertySet` interface, which is returned by the method `get_all_characteristics()` defined both for the `Component` and the base `Property` (see section 3.4.1), however, this returns all characteristics that are defined in the IDL, for fast initialization of client APIs like `Abeans`. So it does not make sense to use `PropertySet` just to get the value of one characteristic.

Therefore two extra methods have been defined for the interface `CharacteristicModel`, which is the ancestor of `Components` and `Properties`, as both have their own set of characteristics:

```

exception NoSuchCharacteristic {
    /** The name of the requested characteristic */
    string characteristic_name;
    /** The name of the NamedComponent which raised the exception */
    string component_name;
}

```

```
};
```

```
any get_characteristic_by_name(in string name) raises (NoSuchCharacteristic);  
stringSeq find_characteristic(in string reg_exp);
```

The first method returns the value of a characteristic named `name` as a type any object. It is up to the user of this value to convert it to whichever type is necessary. The second method allows generic tools to find some characteristics that are hidden from the IDL. The string `reg_exp` is a regular expression that is compared to all available characteristics in the configuration database. A sequence of matching characteristics names is returned as a sequence of strings. In order to simplify the implementation, it may be sufficient to provide wildcard search (with '*' and '?') and not support a full regular expression parser. It is doubtful whether more than wildcard search is necessary, in particular as the names of the characteristics are relatively straight forward.

