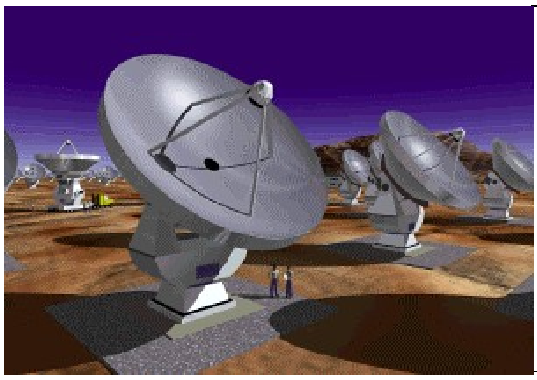

Atacama Large Millimeter

KGB-SPE-01/02

Revision: 2.5.3

2005-09-08

Software Specifications

Gasper Tkacik

ACS Basic Control Interface Specification

Software Specification

Mark Plesko (mark.plesko@ijs.si)

KGB Team, Jozef Stefan Institute

Gasper Tkacik (gasper.tkacik@ijs.si)

KGB Team, Jozef Stefan Institute

Gianluca Chiozzi (gchiozzi@eso.org)

ESO

Keywords: KGB-SPE-01/02

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

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
1.0	1999-09-16	Gasper Tkacik	All
	Created. IDL interfaces modeled after ACI (Accelerator Control Interface).		
2.0	1999-12-25	Gasper Tkacik	All
	Reorganized the document into separate Architecture and Interfaces sections. Modified almost all IDL interfaces to make them more consistent and less restrictive.		
2.1	2000-01-30	Gasper Tkacik	Small modifications to all sections, sections Groups and Descriptors added
	Specified Group interface and BACI Device-to-Group mapping. Specified descriptor structures.		
2.2	2000-03-01	Gasper Tkacik	All
	Cosmetic changes, small IDL corrections.		
2.3	2001-03-25	Gasper Tkacik	Basic Interfaces, Devices, Monitoring
	Used ACS naming convention: renamed Device to DO, added methods for characteristic access, added postponed monitoring capability, made some consistency corrections to the IDL		
2.4	2001-04-01	Gasper Tkacik	All
	Made a number of modifications suggested by Gianluca Chiozzi, separated KGB BACI specification from BACI for ACS specification.		
2.5	2003-11-21	Alessandro Caproni	All
	Partially updated to ACS v3.0		
2.5.1	2004-02-19	Gianluca Chiozzi	All
	Update for ACS 3.0.		
2.5.2	2004-09-22	Alessandro Caproni	
	More explanations about async. calls and callbacks		
2.5.3	2005-09-08	Gianluca Chiozzi	
	Minor update about component references		

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED

Table Of Contents

1 Introduction

1.1 References

[R01] http://www.omg.org/library/c2indx.html CORBA Specification

[R02] Generating COCOS, internal KGB documentation

[R03] MACI, internal KGB documentation

[R04] Mapping of BACI device types into BACI group types, internal KGB documentation

[R05] JavaBeans specification

[R06] OMG Time Service Specification, formal/97-12-21.pdf

[R07] OMG Notification Service Specification, telecom/99-07-01.pdf

[R08] Henning M., Vinoski S., Advanced CORBA Programming with C++, ISBN 0-201-37927-9

[R09] ACS Architecture

1.2 Abbreviations

CORBA	Common Object Request Broker Architecture
IDL	Interface Definition Language
OOP	Object Oriented Programming
BACI	Basic Control Interface
DII	Dynamic Invocation Interface
TBD	To Be Done
MACI	Management and Control Interface

1.3 Control Systems

A ***control system*** is an interface to ***controlled objects***. This interface is utilized by ***controllers*** to send requests to controlled objects and by the objects to report back to the controllers. Controlled objects are entities that can be at least managed by the control system, that is, objects to which a communication link can be established by the system. There are in general no restrictions as to which entity has the role of a controller and which the role of the controlled object. Therefore, we may think about the control system as a communication network between controllers and controlled objects that allows these participants to exchange well-defined messages.

In practice, control systems are often implemented as three-tier architecture systems in which computer users mostly play the role of controllers, while the controlled objects take the form of devices, e.g. machinery or other instrumentation. Naturally there are exceptions to the scheme, for instance when a device with some built-in logic responds to another device and acts as a controller, or when the role of the controlled object is played by an application instead of a physical device.

Since the controlled objects and the users can be in different locations, there must exist a middleware in the control system responsible for the communication over potentially large distances. Today, when the prevailing programming practice is object-oriented programming and the cheapest modes of broadband communication are LAN and the Internet, Common Object Request Broker Architecture (CORBA) fills the middleware niche satisfactorily.

Henceforth the discussion will be limited to the part of the control system that deals with ***network objects***, i.e. objects in the object-oriented programming (OOP) sense that are available to users through the client-server paradigm. CORBA allows the servers and clients to be hooked to these objects from either side - CORBA is therefore the middleware responsible for making the client-server communication transparent to both sides. CORBA objects are defined in terms of Interface Definition Language (IDL). IDL interfaces only declare functionality, but do not implement it **[R01]**.

1.4 Basic Control Interface

Basic Control Interface (BACI) defines a ***framework*** that forms a base for the IDL definition of the interfaces for the controlled objects. BACI itself does not define any ***specific control system*** (for instance a control system designed to manage a synchrotron light source or a network of gas pumps). It does, however, restrict the set of all definable objects with the IDL to a specific set that conforms to BACI design guidelines and uses BACI interfaces in a predefined way.

Conceptually this means that BACI introduces a certain number of entities, such as a component (the basic controlled entity); and a number of pre-defined formats of queries and commands to these entities (such as the methods used to retrieve or set a certain value in the controlled entity).

Programmatically this means that BACI introduces a certain number of interfaces with a well defined containment and hierarchy, which are extended and used by the definition of a specific control system in accordance with BACI design guidelines.

Although it seems that BACI restricts the designer in his/her freedom to adapt the system to the requirements at hand and that imposing such a number of constrains only makes the design cumbersome, there are benefits to this approach that outweigh such objections:

- BACI is in fact very general and does not presuppose any behavior of the defined objects except for the logical containment and hierarchy. More specifically, it neither deals with the life cycle and management of these objects nor with the implementation details (where the static data is stored, what is the client / server platform etc.).
- Experience in the field of control system design has shown that very often the interfaces defined for a particular control system show repetitions in terms of defined objects (instances where one would use C++ templates which do not exist in IDL) or similar functionality (different kinds of data retrieval - synchronous, asynchronous, non-blocking etc). It is a good programming practice to abstract this kind of behavior in special classes and not declare it every time for each component.
- Experience with client programming shows that by introducing a strict, non-flat hierarchy enables one to write universal and relatively simple but powerful clients.

To achieve these benefits it is necessary to design BACI from the user - client viewpoint. CORBA thus must not reveal the underlying hardware or driver details but must hide them in general constructs provided by BACI. CORBA should not be a member- and function- wise mirror of the driver or low-level API, although such access can still be provided for other reasons (debugging for instance); this is not, however, in the scope of BACI and its specific implementations.

As a consequence of client-oriented approach as much of component logic is implemented on the server side as possible.

The BACI approach to control system design is to specify as much functionality as possible formally as part of the IDL itself or with the use of MIDL syntax. Therefore BACI avoids dynamic parameter passing (by using CORBA `Any` type), generic message passing (by using constructs like `sendEvent("event name")`) and dynamic commands (for instance `Component.execute("on")`) that do not permit compile-time checks. The fact that the interfaces to the components, all their responses to commands and events that they can generate are known in advance must affect the definition of BACI and its derived control interfaces. The only reason for the adoption of a dynamic interface is the possible simplification of the IDL interface; but there is a number of drawbacks:

- it is much easier to introduce bugs in the software; errors are detected at run-time, while in strongly typed interfaces they are caught at compile-time; some bugs are very difficult to locate
- when using dynamic approach, an object can implicitly make a promise to the client that it implements a specific functionality although that is not true
- a dynamic approach needs a lot of additional documentation, enumerating the messages, events and responses for each defined object, while a strongly typed interface documents itself through method names and parameter lists
- even if the interface is strongly typed, dynamic mechanisms and clients can still be written using CORBA DII service

All the enumerated constrains and goals of BACI can be summarised in the next few general programming principles:

- BACI uses strong type checking. Interface multiplication due to this decision is not an issue since the interfaces can be automatically generated. When compile-time type checking is broken for any reason, this must occur in BACI and not in any derived (specific) control system.
- BACI and the derived control system should offer two ways of accessing an object in the containment hierarchy, or a specific data item: BACI will declare the generic approach, which will enable the user to obtain, for instance, all objects contained within a given object. The generic approach is still type-safe at compile time and the return types are BACI defined. The specific control system will declare functionality needed to access item by item separately and enforcing complete type checking (i.e. declared types are equal to the actual run-time types). BACI approach will be favored for speed during initialisation (where a lot of information will be transferred in a single remote method invocation), while specific approach will be used when needed during the client's lifetime. BACI will offer such bulk data through structures known as ***descriptors*** (to be described later).
- BACI will enable the walk down the containment hierarchy (from container to contained object) by references and up the hierarchy by names. This design principle is adopted because of security considerations and is an easy way of restricting ways to obtain object references to a few well-controlled entry points.

2 Architecture

2.1 Basic Concepts

BACI is in its scope restricted to defining containment, leaving life-cycle management, access control and other lower-level communication details to Management and Access Control Interface (MACI) **[R03]**. This means that BACI is poor in terms of lines of IDL code, but rich in concepts and design patterns. In the following sections these concepts are presented for better understanding without any accompanying IDL code. The detailed discussion of the interfaces is postponed until the next section.

2.1.1 Component

A ***Component*** is a CORBA object that corresponds to a conceptually self-contained entity: this can be a physical device or a business component that implements certain business logic. The system management of the process which underlies a component is done via a ***COB*** (CORBA object), which means that the access to it is orchestrated by the MACI. The component and the COB represent two views of the same thing: a component corresponds to the interface that the user (i.e. application program) is aware of, while the COB interface is seen only by the management part of ACS.

A component has its own life cycle, i.e. the component can only be available or unavailable as a whole. If it is available to the client it is assumed that its complete interface is available and functioning, as well as any other control system components that it references. This fact makes component the basic entity in the control system.

A component is a named entity. A name is a combination of two identifiers: a string name and a numerical identification number. Both identifiers are unique in the scope of the whole control system and are considered ***static***. The term static will refer to an attribute which is not normally expected to change or the change of which involves a lot of synchronising activity (and may involve restarting parts of the control system).

Components form an object hierarchy that must be rooted in the `ACSComponent` interface.

Components are not allowed to pass around references to other components. This would void the capability of the Manager of keeping an image of the life system and of the dependencies between Components. The Components must be passed around by CURL and the getComponent() Manager method have to be used to get their reference. See **[R09]**

It is allowed to pass around references to Offshoot objects. In this case the Component managing the Offshoot is responsible for the lifecycle of the Offshoot itself.

A component can declare any IDL functionality. However, BACI encourages the designers to construct a component from the predefined BACI components described below.

2.1.1.1 Example

Since there are almost no constraints regarding BACI component, almost every entity can be represented by it. A complex physical device, such as a fuel dispenser, can be modeled by more than one BACI device. For instance, a fuel dispenser is an entity comprised of `FuellingPoints`, which are low-level devices representing fuel nozzles, a state machine component that controls the allowed states of the `FuellingPoints` and a `DispenserCalculator` that models the interaction between components responsible for confirming payment and the `FuellingPoints`.

2.1.1.2 Logical Containment

Component contains: characteristics (*by declaring accessor method, through descriptors*), actions (*by declaring action method*), properties (*by reference, through descriptors*), event sources (*by declaring subscription methods*), components (*by reference, through descriptors*).

Component is contained by: -

2.1.2 Characteristic

A characteristic is a data item that is considered static. It is represented by a name - value pair, where value parameter is typed. A characteristic is accessed either through a single accessor method, defined by the specific control system, which returns its value, or through general BACI functionality, that encapsulates names and values into CORBA Property Service `PropertySet` interface, or through generic methods declared by components and properties. Some characteristics can be state-dependent, which means that their value can change depending on the condition the owner of the characteristic is in. This does not break the notion of a *static* item, since every possible value is prescribed in advance and is, by itself, considered static. The state only determines which of the values should be used at the moment. It is possible for the user to obtain the set of all possible values of a state-dependent characteristic through the BACI `CharacteristicComponent` interface.

The name of a characteristic is case-sensitive.

A remote call that accesses the characteristic is ***synchronous***, i.e. it blocks until the value is returned to the caller, because it is presupposed that static values will be stored in a quickly accessible repository (for instance a static configuration database).

2.1.2.1 Example

An example of a characteristic that is not state dependent is the location of the gas pump. It can be modeled by a name-value pair, where name is `location` and the value is a `string` value describing the address where the pump is located.

An example of a state-dependent constant is maximum value of a current in a power supply. If the power supply is capable of two modes of operation, for instance *normal* and *ramping*, the maximum value of the current may be different in each. Nevertheless, it is still static for every mode, unlike the value of the current that is constantly changing during ramping operation.

2.1.2.2 Logical Containment

Characteristic contains: -

Characteristic is contained by: components (*by accessor methods, through descriptors*), properties (*by accessor methods, through descriptors*)

2.1.3 Event

Two distinct event models can be conceived depending on the nature of the link established between the ***event source*** and the ***event sink***. In the first case there is a source - sink communication through which events are exchanged. Both source and sink are aware of each other by holding some ID or reference of the other party. There can be many sinks for an event source, which is consequently responsible for dispatching events to all sinks. There can be one sink interested in receiving events of the same type from multiple sources. In this case it is the sink that has to maintain an active connection to all event sources. An example of this model is JavaBeans event model, described in **[R05]**.

Where the emphasis in event communication shifts from the source to the event type, another model becomes feasible. In this case the client is interested in receiving events of a certain type (for instance alarm events) and handling them regardless of their source. Since the potential number of sources is very large, it becomes very inefficient if the client must establish a connection to each potential source and register as an event sink. In this case an ***event channel*** is necessary as the mediator between sources and sinks. Sources supply the channel with events while the channel efficiently multicasts events to all sinks that are interested in receiving them. The mechanism also decouples the dispatching process from the normal source processing. OMG Notification Service **[R07]** adopts this approach.

Experience with control system design has shown that to scale well, both models are needed.

The first case, i.e. that of point-to-point event communication, must be an obligation to deliver events from the ***source*** to the ***listener***. A listener ***subscribes*** to the source and the source asynchronously sends notifications to the listener whenever the condition for an event is met. When the listener subscribes to the source it obtains a `Subscription` object through which it can terminate the subscription. Note that the approach is similar to monitoring discussed below.

In the latter case, the event channel is a process separate from the ***consumers*** and ***producers***. The interaction of the BACI components with the event channels is beyond the scope of this document. The design goal is, however, to implement BACI and service interaction in such a manner that the event delivery is independent of the transport method used. In other words, regardless of whether the event was delivered by point-to-point communication or through an event channel, most of the functionality and the code should be shared. To achieve this it will probably be necessary to prescribe a mapping from the BACI event design pattern into the events used by a particular event channel, i.e. CORBA Notification Service.

Event channel counts as BACI system-wide service. See section Service for details.

2.1.3.1 Example

A point-to-point communication using events is appropriate when a state transition of a specific object must be detected. For example, a `StateMachine` object might represent the sequencer which is starting up the devices in a factory. If a client must have knowledge of when the sequencer has finished, it will subscribe directly to the sequencer as an event listener. A channel architecture would be used for alarms or logging: when each component starts up, it contacts an event channel and registers as event producer of `AlarmEvents` and `LogEvents`. Alarm client then contacts the same channel and registers as `AlarmEvent` consumer. All alarm events reach the client through the channel. The client had to register once only compared with the first approach where the number of registrations would equal the number of sources.

2.1.3.2 Logical Containment

Events contain: -

Events are contained by: components (*by declaring event subscription methods*), properties (*by declaring event subscription methods*)

2.1.4 Action

An action is a command with the following properties:

- it is issued ***asynchronously***, i.e. after the command is received by remote object, the remote call returns immediately even if the command is still executing
- when the command finishes the execution, a notification is sent from the remote object to the object that issued the command; this notification must contain the description of the command execution status (successful or failure, reason for possible failure, timestamp)

- the action has a well defined execution path, i.e. after executing the command, it must complete its execution either with an error condition or along exactly one execution path that leads to successful completion; the action must complete the execution by itself (e.g. it is illegal to define an action, which when invoked, executes as long as the component containing it exists; the action must terminate on its own)
- the action termination state (either successful or in error condition) is reported through an universal notification (notification of which the syntax will be defined by this document)
- an action is timed - there exists a mechanism by which the remote object can obtain the timeout interval used by the caller; if the action is not completed before the timeout interval, the remote object must inform the caller that the action is still in progress

2.1.4.1 Example

An example of a simple action would be a command to a printer to start up. Since this is a lengthy procedure, the asynchronous approach seems suitable. Upon issuing the command the control is immediately returned to the caller while the start up procedure begins executing. Since the caller starts the timeout timer on the execution, the printer component must regularly send notifications that the start up is still in progress. When it is completed, the printer component sends a special *done* notification that marks the successful completion of the command.

2.1.4.2 Logical Containment

Actions contain: -

Actions are contained by: components (*by declaring action methods*), Property

A property is a typed changeable value in a context. Properties are represented by interfaces. Depending on the nature of the property, the value that the property interface encapsulates can be either read (by using property's ***accessor*** methods), set (by using property's ***mutator*** methods) or both. In this regard properties can be flagged as readable, writable or read-writable. There is a number of ways to read or set the property's value:

Mutators:

asynchronous	send command to change the value, notify asynchronously if the change is successful
synchronous	send command to change the value, block until the command resolves
step	send command to increment / decrement the value by smallest allowable amount
non-blocking	send command to change the value, return immediately, do not expect reply

Accessors:

asynchronous	send command to read the value, the value itself is returned by asynchronous notification
synchronous	send command to read the value, the value is returned after the blocking call completes
history	send command to read a stored array of property values and their corresponding timestamps, specify how many values should be obtained, return as many as are available up to the requested number

The ***access mode*** of the property (readable, writable and read-writable) is determined in the following way:

- if the property declares at least one mutator, it is writable
- if the property declares at least one accessor, it is readable
- if the property is both readable and writable, it is read-writable

Properties can also be ***monitored***. Monitoring is a special mechanism by which the property can inform interested parties of its current property value. When monitoring is requested by object *x* on property *p*, *p* returns `Monitor` object to *x*. Through the `Monitor` *x* can control when it wishes to receive the updates of *p*'s value. The conditions for notification (***triggers***) are either:

- When a certain time (delta time) has passed from the previous notification, a new notification must be sent
- When the monitored property value has changed more than a certain amount (delta threshold), a notification must be sent

Notifications can be sent on any combination of above triggers, which are settable through the `Monitor` object at run-time in the following manner:

- Delta time is a continuous variable expressed in `TimeInterval` time format; delta time trigger must not be below `min_timer_trigger`, a characteristic contained by monitored property; if such delta time is requested, `min_timer_trigger` delta should be used instead;
- Delta threshold is a variable of the same type as the property value; delta time must not be below `min_delta_trigger`, a characteristic contained by monitored property; if such delta threshold is requested, `min_delta_trigger` should be used instead

- When a monitor is created, its triggers are set to default values, accessible as the characteristics of the monitored property: `default_timer_trigger` and `default_delta_trigger`
- If the property value does not support semantically sensible comparisons (for instance, if the property represents a bit pattern), the corresponding `Monitor` object must not declare the functionality related to delta threshold trigger and neither must the property declare the `min_delta_trigger` and `default_delta_trigger` characteristics
- Regardless of the default trigger settings, a single notification is always sent immediately after the monitor is created as the side effect of monitor creation, unless a monitor is created that explicitly sets the absolute time for the first notification to the server

BACI itself cannot guarantee that the triggers requested by the user will actually be met by the control system. However, the control system should be designed so that the values supplied as the triggers are respected at least to the correct order of magnitude. The control system should use a best effort approach to deliver the notifications in the intervals specified by the triggers.

Monitors can be suspended, resumed and must be explicitly destroyed when they are not used anymore.

Characteristics of the property provide property's context. Ordinarily, at least the following characteristics will be included: a description of the property, its physical units, output format (for instance in the C `printf` style notation), bounds (minimum and maximum values), minimum step over the whole range, if the value of the property can be set etc.

There should be a small number of property interfaces, each characterised by the access mode and the property value type. Although it is possible to create two property interfaces for the same type (two property interfaces each describing a `double` value, for instance), this is strongly discouraged, since it considerably complicates the creation of generic clients but brings only small benefits to the overall consistency of the control system.

Properties can only exist in their containers - in other words, a connection cannot be established directly to the property.

Properties are named in the same way as components, i.e. by having a unique string name and identification number. The property's unique name is prescribed to be the container's name followed by '–' followed by property's name.

Properties can also be event sources, which can be useful for implementing alarms (e.g. an event is sent whenever the property value falls outside specified alarm bounds).

Properties form an object hierarchy that must be rooted in the `Property` interface declared by BACI.

2.1.4.3 Example

A current of the power supply can be regarded as the property. To model the current we can create an interface `RWdouble` that represents a read-writable value of type `double`. The `RWdouble` interface inherits from the `Property` interface and contains methods that enable the user to access or modify its value, methods that enable the user to create a monitor on the current and a number of characteristics, similar to those enumerated above.

2.1.4.4 Logical Containment

Property contains: characteristics (*by accessor methods, through descriptors*), event sources *(by subscription methods)*
Property is contained by: components (*by reference, through descriptors*)

2.1.5 Service

A service is a distributed entity that performs a certain system-wide function. It differs from components in the following points:

- Its interface can not be described in terms of BACI design patterns, since services do not declare similar functionality
- It provides the framework for other objects of the control system, it does not have a physical device that it is coupled to.
- It must be available to the components at the time of their construction
- It is not named in the same sense that the `ACSComponents` are named: there is only one name for a service performing one function (the name is thus a functional designation) regardless of where actual instances of the service run
- Services may be federated, meaning that if parts of the control system are joined (e.g. by establishing a LAN connection) the data from both partial services are meaningfully integrated

Event channel as an example of the service was already mentioned. Possible other incarnations of the service concept include the log service, archive service, naming / directory service etc.

BACI does not declare any interface that a service must implement. Details concerning the federation of services are implementation dependent. Access to the service and security guarantees are the responsibility of MACI. MIDL can, however, contain additions to its directives that simplify the interaction with the services and the automatic code generation. Since the service interfaces are not defined by this document, every addition to MIDL not contained in this document must of necessity be dependent on the choice of the actual service interface.

2.2 Summary

The basic ***containment hierarchy*** defined by BACI is the hierarchy `CharacteristicComponent` - `Property`. These components not only contain each other logically, but also programmatically (by references and by names). Each of these interfaces forms a ***type hierarchy*** of its own. Other constructs, such as characteristics and events can hardly be described by interfaces. Nevertheless, they can be described by design patterns, similar to the design patterns used by JavaBeans specification **[R05]**. The conceptual model outlined above leaves out the actual technical details of how the asynchronous communication, events, monitors and the like are actually designed. The presentation of interfaces, design patterns and technical solutions of BACI is the topic of the next section.

3 Interfaces and Design Patterns

3.1 Namepatterns

Since design patterns cannot be formalized in pure IDL, the following notation is used to represent any identifier (constructed so that it also counts as official IDL identifier): the name of the identifier is replaced by its description in cursive script. Therefore,

```
static_data_item_type static_data_item_name();
```

would represent a method returning a value of type *static_data_item_type*, called *static_data_item_name* and taking no parameters.

3.2 Basic Interfaces

Several values of the same type are stored in a ***sequence***, for example longSeq, doubleSeq etc. The name pattern for the sequences (that map to arrays in C++ or Java) is

```
typedef sequence<type> typeSeq
```

Sequences are used extensively with components that handle multiple properties of the same type as if they were arrays. Sequences defined by BACI are also used to optimize the number of network calls needed to query a remote object.

Times are represented in BACI by a Time type, defined by

```
typedef unsigned long long Time
typedef long long TimeInterval
```

where long parameter represents the time as defined in **[R06]** (hundreds of nanoseconds from 1582-10-15 00:00:00). Time intervals are passed as TimeInterval values.[1] Each value read from the control system must be accompanied by its timestamp, which is returned through the Completion structure:

```
struct Completion {
   unsigned long long timeStamp;
   ACSErr::ACSErrType type;
   ACSErr::ErrorCode code;
   sequence<ACSErr::ErrorTrace, 1> previousError;
};
```

[1] The definition of the Tine structure could change with the full definition of the ACS Time System.

Parameters `type` and `code` represent the completion status of the operation (action, value retrieval and the like). Hierarchical arrangement of error codes is possible: for a defined type the code parameter will describe the condition more in detail. A more detailed discussion of completion structures and their descriptions is postponed to the latter sections.

If the `Completion` `type` and `code` values indicate an error condition, `Completion` structure may contain one `ErrorTrace` structure instance in the error sequence. If the `Completion` type and `code` indicate an error-free condition, the error stack must be empty. If the `Completion` previousError exists, it must contain the reason for its parent completion error status.

The main BACI components, such as `CharacteristicComponent` and `Property` are named: they have a name, which the manager can convert to object reference if the client has the correct access level (only for `CharacteristicComponent`).

```
exception NoSuchCharacteristic {
   string characteristic_name;
   string component_name;
};

interface ACSComponent {
   readonly attribute string name;
   readonly attribute ComponentStates componentState;
}

interface CharacteristicModel {
   any get_characteristic_by_name (in string name)
       raises (NoSuchCharacteristic);
   stringSeq find_characteristic (in string reg_exp);
   CosPropertyService::PropertySet get_all_characteristics ();
}

interface CharacteristicComponent : ACSComponent, CharacteristicModel {
   CharacteristicComponentDesc descriptor ();
};
```

BACI prescribes no name hierarchy. This task is in the domain of MACI.

Methods `find_characteristic and get_characteristic_by_name` allow a search of all static data items defined for this `CharacteristicComponent`. Method `find_characteristic` takes a regular expression as an argument, returning a sequence of all characteristic names that match the expression. It must return a sequence of length 0 if no match is found. The method `get_characteristic_by_name` returns the value in the characteristic name-value pair. The name passed as a parameter must be an exact match of the name in the name-value pair. If no characteristic with such name exists, an exception declared above must be raised.

3.3 Callbacks

A callback is the technical mechanism underlying the concepts of asynchronous notification (discussed in connection with actions) and monitoring. It is an object passed by the client to the server, so that the server can later invoke methods on the callback and thus inform the client of a change in status, completion of some operation and the like. During this notification the roles of the client and the server are reversed. Callback interfaces are defined in the following manner:

```
typedef unsigned long Tag;

typedef struct CBDescIn {
   TimeInterval normal_timeout;
   TimeInterval negotiable_timeout;
   Tag id_tag;
};

typedef struct CBDescOut {
   TimeInterval estimated_timeout;
   Tag id_tag;
};

interface Callback {
   boolean negotiate(in TimeInterval time_to_transmit, in CBDescOut
desc);
};
```

Every callback must be rooted in the `Callback` interface. When the client passes the callback to the server, it must accompany it with the `CBDescIn` structure. Upon the asynchronous notification or monitor notification, when the server invokes a method on the callback, it passes `CBDescOut` structure as one of the parameters to the remote call. Through both structures, the client programmer can exchange the `id_tag` parameter with the server to uniquely tag the remote call. The only interaction the control system has with the `id_tag` is that it passes it unchanged from `CBDescIn` to `CBDescOut`.

Callback descriptor objects provide the server and the client with the technical aspects of callback invocation. As it will be seen later, each callback must also declare at least one `Completion` structure as the parameter. Completion structures carry the context of either the value or the operation they parameterise.

Parameter `normal_timeout` in `CBDescIn` is sent by the client to the server to inform it that it expects a reply in the `normal_timeout` period specified in `TimeInterval` format, 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 mechanism of callback timeouts is explained more in detail at the end of this section.

BACI predefines the callbacks for `void` return type and other simple and sequence return types by the following design patterns:

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

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

Note that predefined callback types allow the *type* to stand for sequential types, e.g. the above pattern defines also the CBdoubleSeq callback type.

Other, custom defined callbacks should have the name pattern:

```
interface CBcallback_name : parent_callback ...
```

and should only declare `oneway` methods. Any callback used for monitoring or asynchronous notification must declare a `oneway` method with name `done.`

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. The completion condition and the timestamp are reported through the `Completion` structure as a part of the callback notification. 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). Exactly when the server must issue working notification will be discussed later. The client must not discard the callback before `done` notification is called or one of the notifications timeouts on the client side.

When the callback is used for monitoring, it must issue `working` notification with the current value of the monitored property whenever the trigger condition is satisfied. Since monitoring specifies its own timing dynamically, the server must ignore the `normal_timeout` parameter passed by the client in `CBDescIn` except in two cases:

- When the monitor is first created, the server is obliged to send a monitor notification immediately if the server is not explicitly ordered otherwise by the client; if such notification does not arrive in `normal_timeout`, the client may raise a timeout error condition, but must not discard the callback without first requiring explicit destruction of the monitor (however, it is not required to destroy the monitor - it may wait for further notifications)

- When the monitor is explicitly destroyed by the client, the client must not discard the callback until the server sends `done` notification; the server is obliged to send `done` notification before `normal_timeout` period expires after destroy was requested; otherwise - in case of timeout - the client may discard the callback

The client also ignores the `estimated_timeout` value passed in `CBDescOut`.

Sequential callbacks (callbacks that deliver a sequence of values) are used **for a single data item that is of type sequence (and is sampled once only, i.e. it is not a collection of data items from different sources).**

Clients and servers must raise `BAD_PARAM` CORBA system exception when the required callback parameters are null (callbacks and callback descriptors).

3.3.1 Callback Timeout and Negotiation Mechanisms[2]

3.3.1.1 Callback Negotiation Mechanism

A situation can arise in which the client sends an asynchronous request to the object that is not connected to the LAN but it accessed through a link that has to be established first (for example a dialup connection). In this scenario there is a proxy object on the client LAN. Proxy implements the same interface as the remote object. Such proxy makes the dialup connection transparent. However, when the client invokes a call on the proxy, a proxy has to tell the client that, since a dialup is not active, some time will pass before the request will reach the real remote object. It does so through `negotiate`. The sequence of events is the following:

a client invokes an asynchronous request on the object (which can be proxy), passing the callback and callback in descriptor as parameters

if the object receiving the callback is the proxy, `CBDescIn` is examined and the `normal_timeout` field is accessed. If the proxy deems that the call can be invoked on the real remote object before the normal timeout expires, it just passes the call on to the remote object. Otherwise the proxy accesses the `negotiable_timeout` field. If the proxy plans to dialup the remote object before the `negotiable_timeout` expires, it must call negotiate, passing as the `time_to_transmit` parameter the time before the dialup will be established and the message queue sent to remote object. Normally, the client will return `true` which means that the asynchronous request should stay in the queue and that the client expects the callback notification when the call executes on the remote side. If, however, the proxy plans to dial the remote object after the `negotiable_timeout`, it has two choices: either to move the dial time forward so that the dialup happens before the `negotiable_timeout` expires or not. Either way, it has to call negotiate with the `time_to_transmit` parameter. If this time is longer than `negotiable_timeout` the client may return `false`, indicating that the request should be removed from the queue and that the client expects no further callbacks. In case of `true`, the scenario is the same as before.

[2] The discussion about the times is at design level. In the current implementation all the times in CBDescIn and CBDescOut are ignored.

When the remote object receives the request it performs the desired operation and calls `done` method. If, however, the execution of the method takes a long time, it should call `working` to let the client know that the request has been received and is being processed and that the client is to wait for the `done` message, after which the callback for this call may be discarded.

3.3.1.2 Callback Timeouts

The exact timeout determination sequence, if it is supported by the client, proceeds as follows.

The client invokes an action on the remote object and starts the timeout timer. A typical response time (`oneway` ping + initial processing time) `Tr` elapses until the server begins executing the request.

The server determines whether the request can be executed in less than the `normal_timeout` passed by the client in `CBDescIn`. The server will perhaps examine its static data to determine the typical processing time or typical timeout. It this datum is less than the `normal_timeout`, the server should do nothing until the request has executed and then send the `done` notification to the client.

If the server has determined that the execution will take more than `normal_timeout`, it must invoke `working` notification as soon as possible, indicating that the request has been received and is being processed. At the same time the `working` notification contains new timer data for the client, contained in the `estimated_timeout` field of `CBDescOut`. The client should then use this data item to restart the timeout timer with new `estimated_timeout` value. The client will expect the server to send the `done` or an additional `working` notification before `estimated_timeout` elapses.

If the server sends `done` notification before `estimated_timeout` elapses, the asynchronous action invocation completes successfully and the callback may be destroyed. If the server sends another `working` notification, step 3 is reiterated (the client again restarts the timer). If no notification is received, a timeout condition is raised and the callback may be destroyed.

The approach does not presuppose that the internal clocks on the client and server are synchronised, although this is necessary for the consistency of timestamps in the `Completion` structures. It is evident, however, that in this scheme the response time `Tr` is not explicitly taken into account. This fact places a condition on the selection of timeout intervals, since they should be long compared to `Tr`. A typical `normal_timeout` should probably be of the order of 5 seconds. To allow sufficient room for efficient implementation, the conformance of clients to the `estimated_timeout` parameter is relaxed: the client should never check for timeout before the `estimated_timeout` has elapsed, but may do so in the time interval between `estimated_timeout` and `estimated_timeout` + `normal_timeout`. It is not sensible to stick to the exact values, since a time interval of at least `2*Tr` is ignored anyway. Any `estimated_timeout` value that is less than normal_timeout is interpreted by the client as estimated_timeout = normal_timeout.

Custom callbacks (not one of the BACI predefined callbacks) can contain other methods except for the mandatory done method. An invocation of any other method counts as an invocation of working as far as the timeouts are concerned. Nevertheless, done method - as in normal callbacks - must be called once only (i.e. done must not be called to notify the client about the work in progress).

3.3.2 More on asynchronous calls

To better understand the functioning of the asynchronous calls have a look to the example in the acsexmpl module. The client, acsexmplAsyncCalls, connects to a component, SLOW_MOUNT, that simulates the movements of the antenna during a 30 secs time interval. The movement is slow so it is performed asynchronously and when the antenna is in position the done method in the callback is executed.

The client connects to the slow mount component through the SimpleClient object. Before executing the asynchronous call the client builds the callback and the CBDescIn object. There is no need to fill the times in the CBDescIn because times handshake is not yet implemented.

```
// Create the callback for the objfix method
   AsyncMethodCBvoid objfixCB("objfix");
   ACS::CBvoid_var objfix_CB = objfixCB._this();
   CBDescIn objfixDescIn;

   // Times handshaking is not yet implemented so
   // we do not really need to fill these fields
   //objfixDescIn.normal_timeout=10000000;
   //objfixDescIn.negotiable_timeout=5000000;

   CORBA::Double newAz=(CORBA::Double)param->az;
   CORBA::Double newEl=(CORBA::Double)param->el;

   ACS_SHORT_LOG((LM_INFO,"Calling the asynchronous objfix"))

   param->mount->objfix(newAz,newEl,objfix_CB.in(),objfixDescIn);
```

objectfix returns immediately and the client proceeds with its computation.

The objectfix method in the component registers an action in an apposite data structure and exits. There is a kind of engine in baci that repeatedly executes alll the actions registered (in our example objfixAction.) until they return a definite value. The following table reports all the values that an action is allowed to return and their meaning.

Value returned	Engine's action	Callback's method

`reqInvokeWorking`	The action will be executed again	`working`
`reqInvokeNone`	The action will be executed again	/
`reqInvokeDone`	The action is never executed again	`done`
`reqInvokeDestroy`	The action is never executed again	/

In the example, `objfixAction` returns `reqInvokeWorking` while simulating the movement. In this case the `working` method of the callback is executed in the client. The `objectfixAction` will be executed again.

When the antenna is in position, `objfixAction` returns `reqInvokeDone`. In this case the `done` method of the callback is executed in the client. The engine will never execute `objectfixAction` again.

```
ActionRequest
SlowMount::objfixAction (BACIComponent *cob_p,
     const int &callbackID,
     const CBDescIn &descIn,
     BACIValue *value_p,
     Completion &completion,
     CBDescOut &descOut)
{
...
    // Simulate the movement of the antenna

    completion.timeStamp=getTimeStamp();
    completion.type=ACSErr::ACSErrTypeOK;
    completion.code=ACSErrTypeOK::ACSErrOK;
...
    // Check if the antenna is position
    if (actAz==param_p->az && actEl==param_p->elev) {
      // The antennza is in position
...
      return reqInvokeDone;
    } else {
      // We need a further iteration
      return reqInvokeWorking;
    }
}
```

`reqInvokeNone` and `reqInvokeDestroy` are equivalent to `reqInvokeWorking` and `reqInvoeDone` but the relative callback's methods are not executed.

Before returning a value, the completion is filled. In our example there are no errors so we set code to `ACSErrOK` and type to `ACSErrTypeOK`. This completion is also a parameter of the methods of the callback: the client checks for errors in reading the values of the fields of the completion in the methods of the callback. The following code shows how to check if the action terminated with an error.

```
void AsyncMethodCBvoid::done (const ACSErr::Completion &c,
   const ACS::CBDescOut &desc) throw (CORBA::SystemException) {
  if (c.code!=ACSErrTypeOK::ACSErrOK) {
  // error
  ....
  } else {
   ...
  }
}
```

3.4 Events

To define an event in the point-to-point style communication, both an event definition and the design pattern for the event source are needed. BACI realises events as the generalised callbacks, i.e. the conceptual act of delivering an event is implemented as the execution of a method on client's callback by the server. Events are grouped in ***event sets***, to which the client can subscribe as a listener. There is no way to subscribe to a subset of events declared by an event set. The event callbacks must follow the following design pattern:

```
interface event_set_name : Callback {
   oneway void event1_name(..., in Completion c, in CBDescOut desc);
   oneway void event2_name(..., in Completion c, in CBDescOut desc);
   ...
};
```

where ellipsis (`...`) in parameter list denotes a variable number of typed parameters. Note that event callbacks do not have to declare `done` method.

For each event interface, there will exist an interface of the same type delivering a sequence of values, namely:

```
interface event_set_nameSeq : Callback {
   oneway void event1_name(..., in Completion c, in CompletionSeq cs, in
CBDescOut desc);
   oneway void event2_name(..., in Completion c, in CompletionSeq cs, in
CBDescOut desc);
   ...
};
```

In this case the ellipsis in parameter list denotes the parameters of the CORBA sequence type, corresponding to the parameter types declared by the interface event_set_name design pattern defined above. event_set_nameSeq versions are used in event history iterators. The event source must implement the following methods:

```
exception NoSuchEvent {};

interface Subscription {
   void suspend();
   void resume();
   void destroy();
};

interface EventIterator {
   boolean next();
   boolean next_n(in long n);
   boolean next_event_type();
   void select_event_type(in string event_type) raises (NoSuchEvent);
   long count();
   void destroy();
};

/* implemented by the event source */
Subscription new_subscription_event_set_name (in event_set_name, in
CBDescIn desc);
Time event_history_event_set_name (out EventIterator iterator, in Time
t, in event_set_nameSeq, in CBDescIn desc);
```

Since the events are not timed, the timing parameters in callback descriptors may be ignored for this purpose (the tag must remain unaltered in callback descriptors). Upon registering, the event source will send events to the listener, which will be able to regulate the event flow through the Subscription interface. The design decision for such subscription style (instead of subscribe / unsubscribe methods) rests on the fact that CORBA specification does not guarantee the correct result of reference comparison (with _is_equivalent calls). For more information on this subject see **[R08]**. The event_history method allows the retrieval of all events that the event source generated since time t which is passed as the parameter. BACI does not require the persistence of events from the event source; therefore the source might not be able to return all the desired events. The Time return value will indicate the earliest event in available history (so that it is impossible to mix up the time period for which there is no history and the period for which there is history but no event was generated), while the out iterator parameter will allow the client to browse through the events. Method count returns the number of events that the iterator can deliver. Methods next and next_n deliver the required number of events in the chronological order, while methods next_event_type and select_event_type specify which event from the event set represented by the callback interface passed to the event_history method is of the interest to the client. The return value of next_event_type will indicate if there is any type remaining, i.e. in case of false return value, the next calls must return false and not invoke the callback methods. The same behaviour is prescribed for the next iterator methods when no more events are available in the history. The string parameter passed to the select_event_type must be equal (case sensitive) to the eventx_name specified in the event callback, otherwise an exception will be raised.

The order in which `next_event_type` returns events is implementation dependent. The iterator must be destroyed when it is no longer used. The behaviour of the iterator in case of the event history change during iteration is undefined, but must be thread safe. Note also that the iterator has asynchronous format, since event history retrieval is thought to be a resource consuming call that is rarely used.

3.5 Property

The `Property` interface is very simple: it contains the name of the containing `CharacteristicComponent` and allows the clients to obtain all of the static data for the property in one network call:

```
exception NoSuchCharacteristic {
   string characteristic_name;
   string component_name;
};

interface CharacteristicModel {
   any get_characteristic_by_name(in string name)
       raises (NoSuchCharacteristic);
   stringSeq find_characteristic(in string reg_exp)
   CosPropertyService::PropertySet get_all_characteristics ();
};

interface Property : CharacteristicModel {
   readonly attribute string name;
   readonly attribute string characteristic_component_name;
};
```

By calling `get_all_characteristics()` the client can download the whole set of static data for the property.

A property may be an event source; in this case it must conform to the design patterns explained in the section Events.

When the `Property` interface is extended it is given extra functionality by implementing the function templates described below ([type] signifies the type that the property represents):

Accessors

synchronous

```
type get_sync(out Completion c);
```

asynchronous

```
void get_async(in CBtype cb, in CBDescIn desc);
```

history, returns the last `nValues` or less (indicated by return value);

```
long get_history(in long nLastValues, out typeSeq v, out TimeSeq ts);
```

Mutators

synchronous

```
Completion set_sync(in type value);
```

asynchronous, returns immediately

```
void set_async(in type value, in CBvoid cb, in CBDescIn desc);
```

non-blocking

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

step

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

Skipping the intermediate abstract classes in the hierarchy, the final property classes that can appear as IDL attributes in a component declaration should have the following access mode name pattern:

`RO<type>, WO<type>, RW<type>`

where `RO` designates that the property is Readable but not Writable, `WO` is Writable but not Readable, while `RW` is both. Note that the cases where `WO` could be used are rare (one case is to declare a command that is actually a bit property: to execute it you set a bit to 1; the result is, on the other hand obtained through a `RO` pattern property status) and there should be a good reason to adopt it.

BACI provides a generic way of returning all characteristics of the property by calling `get_all_characteristics` method. However, each interface extending the `Property` interface must declare exactly one method per static data item that returns the current value of the characteristic. Each characteristic accessible through statically typed method must also be accessible through generic methods. The name pattern is the following:

```
readonly attribute static_data_item_type static_data_item_name;
```

Therefore for each method with *`static_data_item_name=key`* and *`type`*`=type_of` *`value,`* `there is a` *`key - value`* mapping returned in

CosPropertyService::PropertySet. When the property contains a state-dependent constant, the function returns the current value of that constant.

3.5.1 Monitoring and related objects

The concept of monitoring has already been explained. The monitor interface through which the client can control the flow of monitoring notifications is declared as follows:

If the property that declares monitoring capability supports the sensible comparisons on its value, it will declare the following functionality:

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

interface Monitortype : ACS::Monitor {
   void set_value_trigger(in type delta, in boolean enable);
   void get_value_trigger(out type delta, out boolean enable);
};
```

If the comparisons are not possible, value triggers are not defined for that monitor type.

In order for the property interface to declare monitoring capability, it must contain the following methods:

```
Monitortype create_monitor(in CBtype cb, in CBDescIn desc);
Monitortype create_postponed_monitor(in Time start_time, in CBtype cb,
in CBDescIn desc);
readonly attribute TimeInterval default_timer_trigger;
readonly attribute TimeInterval min_timer_trigger;
readonly attribute type min_delta_trigger;
```

The last method will only be declared if the property supports the comparison operations on its value. On creation, the only trigger present will be the timer trigger. The enable parameter determines whether the delta trigger is active or not (if the property is of type integer and can actually take all the possible values, there is no way to disable the delta trigger, since any delta value is meaningful). Value 0 assigned to delta trigger means that a notification should be sent on every change of the monitored value. Timer trigger can be disabled by passing the value 0 for timer parameter. Invalid values (out-of-limits) are treated as valid extremes (a delta trigger below min_delta_trigger is treated as min_delta_trigger, a time interval out-of-limits defined by the server is interpreted as maximum allowed time interval).

A special monitor creation is allowed through create_postponed_monitor. Here the user is free to specify, in absolute time, when the first callback notification should occur. Since the user passes the time explicitly, the server must refrain from the general rule that it must respond with working within the normal_timeout period. If the time specified as an argument to the create_postponed_monitor has already passed, this method invocation counts as a normal `create_monitor`. BACI cannot guarantee the delivery of the first notification exactly on time, but will provide best-effort service not to deliver any notification before the specified time. Attribute `start_time` in Monitor interface returns the time of the first callback invocation, regardless of whether it has already occurred or if it is still scheduled.

Because monitor creation and other asynchronous invocations consume a certain amount of resources on the server machine, it is possible for them to fail in case the resources run out. The standard BACI approach is for the server to raise CORBA::NO_RESOURCES exception. The unchecked exception was preferred to checked exception in such cases since:

Exception made for this purpose already exists and is CORBA standard.

Lack of resources is a system problem and not programmers fault. Exception in this case is not a specific response (and return path) from a specific method, but a statement of the system state as a whole. It is almost impossible for the client to be able to do any sensible error recovery in this case. It can still catch an unchecked exception and do some action, but the client programmer can let it propagate to the (more generic) exception handler higher in the calling stack.

3.6 Components

All Components share a common abstract interface, which is made of the methods that place the Component into the containment hierarchy:

```
interface ACSComponent {
   readonly attribute string name;
   readonly attribute ComponentStates componentState;
};

interface CharacteristicModel {
   any get_characteristic_by_name(in string name)
       raises (NoSuchCharacteristic);
   stringSeq find_characteristic(in string reg_exp);
   CosPropertyService::PropertySet get_all_characteristics();
};

interface CharcteristicComponent : ACSComponent, CharacteristicModel {
   CharacteristicComponentDesc descriptor ();
};
```

Method `get_all_characteristics()` returns the static attributes of the distributed object. Actual components consist of properties, actions and static data. For each property of type `<type>` the component should declare the following template:

```
readonly attribute property_type property_name;
```

For each action, the following template should be used:

```
void action_name(..., in CBtype cb, in CBDescIn desc);
```

where ellipsis `(...)` designates a variable number of typed parameters.

To access components contained within a given component container, the same method pattern is used as for the characteristics:

```
<contained_type> get_<contained_name>();
```

For each static data item, the same templates should be used as defined under the section `Property` interface.

A component may be an event source; in this case it must conform to the design patterns explained in the section Events.

3.7 Descriptors

A descriptor is an IDL structure that contains key data used by the typical clients and enables the clients to retrieve this data in a single network call. A descriptor is an implementation detail and has no conceptual value in itself. The following rules pertain to descriptors:

- A descriptor will contain named members that are declared by BACI and not by the specific control system. Features of the specific control systems are returned in sequences of BACI defined root types (i.e. a component descriptor for a power supply contains a sequence of Properties and not each specific Property by name). Therefore all existing descriptors are declared by BACI and not by the specific control systems.
- Every member accessible through a descriptor must be accessible through an ordinary accessor method declared by the object being described.
- Descriptors will nest: thus a component descriptor will not contain Property references but Property descriptor structures.
- A descriptor is accessible only from the `CharacteristicComponent` that can be directly accessed by the client.

```
struct CharacteristicComponentDesc {
   ACS::CharacteristicComponent characteristic_component_ref;
   string name;
   PropertyDescSeq properties;
   CosPropertyService::PropertySet characteristics;
};

typedef struct PropertyDesc {
   Property property_ref;
   string name;
   CosPropertyService::PropertySet characteristics;
};
```