

Change Record

Version	Published	Changed By	Comment
CURRENT (v. 9)	Apr 13, 2019 21:56	 Jorge Avarias	
v. 8	Apr 13, 2019 21:54	 Jorge Avarias	
v. 7	Apr 13, 2019 21:39	 Jorge Avarias	
v. 6	Apr 10, 2019 22:16	 Jorge Avarias	Migration of unmigrated content due to installation of a new plugin
v. 5	Apr 10, 2019 22:16	 Jorge Avarias	Migration of unmigrated content due to installation of a new plugin
v. 4	Apr 10, 2019 22:16	 Jorge Avarias	
v. 3	Apr 10, 2019 22:12	 Jorge Avarias	
v. 2	Apr 10, 2019 22:07	 Jorge Avarias	
v. 1	Apr 10, 2019 21:57	 Jorge Avarias	

- [Introduction](#)
 - [Purpose of the Document](#)
 - [Abbreviations](#)
 - [References](#)
- [Designing a Component](#)
 - [Properties](#)
 - [Commands](#)
- [Creating IDL](#)
- [Configuration Database files](#)
 - [A simple example: the mount.](#)
- [Header files](#)
- [Implementation files](#)
- [The ContainerServices](#)
- [Logging and debugging](#)
- [Compiling](#)

Purpose of the Document

The purpose of this document is to illustrate how to create a simple device server based on ACS. A C++ example is developed step-by-step and the implementation is explained (note: the example code is taken from the ACS software module "acsexmpl" with very few modifications).

This document neither explains architecture in depth, nor the low-level communication between the device server and the physical device. However, it is advisable to get in-depth acquaintance with concepts of the Basic Control Interface, as described in detailed technical document (BACI specification *[R01])* and other documents, specified in References).

Note: these instructions are written for programming under UNIX. Programming under MS Windows does not differ very much, but there are some minor changes.

Abbreviations

ACSCo	ACS Component
BACI	Basic Control Interface
CORBA	Common Object Request Broker Architecture
IDL	Interface Definition Language
MACI	Management and Control Interface
CDB	Configuration Database

References

[R01]	ACS Basic Control Interface Specification]]></ac:plain-text-body></ac:structured-macro> (http://kgb.ijs.si/KGB/Alma/Specs/ACS_Basic_Control_Interface_Specification.pdf)
[R02]	Management and Access Control Interface Specification]]></ac:plain-text-body></ac:structured-macro> (http://kgb.ijs.si/KGB/Alma/Specs/Management_and_Access_Control_Interface_Specification.pdf)
[R03]	Definition of BACI types for ACS1.1 ([http://kgb.ijs.si/KGB/Alma/Specs/Definition_of_BACI_Types_for_ACS1.1.pdf])
[R04]	Logging and Archiving ([http://kgb.ijs.si/KGB/Alma/Specs/Logging_and_Archiving.pdf])
[R05]	Advanced CORBA Programming with (Henning, Vinoski)
[R06]	C++ Programming Language Third Edition (Stroustrup)
[R07]	ACS Software Module "acsexmpl"
[R08]	CDB Tutorial ([http://kgb.ijs.si/KGB/Alma/Docs/CDB.pdf])

First, we need to know what kind of a device (actually Component) we are writing a device server for. Is it a giant motor, which rotates telescopes, simple power supply, device which throws baked pieces of bread out of a toaster, or an object, residing only on the network, which does something useful? Our part of writing a device server should actually not distinguish much; only variable and method names should be different. Maybe also the types of variables (double, pattern, read-only, etc.) and methods (synchronous and asynchronous), but that should be all. We have to now consider properties that our Component will maintain and also all possible actions. For example: a simple power supply.

Properties

- **current** - the reference current which the power supply should deliver to the machine – we will set this property, so it should be a read-write double (RWdouble).
- **readback** - the actual current delivered by the power supply, as measured by the physical device. Ideally, the values of current and readback are equal. However, due to fluctuations in the feedback loop and measuring imprecision, the readback always differs slightly from the reference current. A case, where the current and readback differ significantly, is, when the power supply is switched off: the readback is zero, independent of what current has been requested. We cannot set the readback, so it will be a read-only double (ROdouble).
- **status** - the status is described by a series of alternative conditions: ON/OFF, LOCAL/REMOTE, ALARM/NO-ALARM, READY/NOT-READY, etc. Each condition is represented electrically by a digital input, or - in software - by one bit, respectively. Depending on the type of power supply, several conditions can be active at the same time. Therefore, the status is a collection of bits that can be called a bit-pattern. A variable of type unsigned integer is used to keep the values of this bit-pattern.

Note that in order to change the status, one can not simply change the value of a bit; e.g. to change from OFF to ON, the command on() has to be called explicitly (see below). And the transition from NO-ALARM to ALARM is completely out of the user's hands – an alarm is an external event, for example a burned fuse, a high temperature, etc. For this reason, it will be ROpattern.
Note: A property is described by more data than just its value. There are also limits (maxValue and minValue), some describing text (name, description, units) and other relevant data. They are called **characteristics**. Such data are constants and are usually read from the configuration database.

Commands

- **on**: switches the power supply on
- **off**: switches the power supply off
- **reset**: resets the power supply

(If you are using ALMA directory structure, IDL files should be located in <xxx>/ws/idl or the <xxx>/idl directory.)
Having defined the device we are now able to write the Component's interface. For this, Interface Definition Language is used. An IDL file is a list of all methods and properties that are needed to describe an interface between client and server.
It looks like this (note: code marked with **bold** should be adapted for new applications):

```
#ifndef _POWERSUPPLY_IDL_
#define _POWERSUPPLY_IDL_

#include <baci.idl>

#pragma prefix "alma"

module PS{
interfacePowerSupply : ACS::CharacteristicComponent {
    void on(in ACS::CBvoid cb, in ACS::CBDescIn desc); // @Action
    void off(in ACS::CBvoid cb, in ACS::CBDescIn desc); // @Action
    void reset(in ACS::CBvoid cb, in ACS::CBDescIn desc); // @Action
    readonly attribute ACS::Rwdouble current; // @Property
    readonly attribute ACS::ROdouble readback; // @Property
    readonly attribute ACS::ROpattern status; // @Property
};

};
#endif
```

4	Because we use RW/ROdouble, ROpattern, and other BACI components, we have to include baci.idl.
6	This is necessary for all alma interfaces so they will have the correct location within the Interface Repository.
8	And now the creation of a new module, with the name PS. A module is (by ESO specifications) a part of code, which rounds up some functionality (same as package in Java and namespace in C++).
9	Interface (like a class in C++) is a defined collection of methods and properties, specific for one object. One module can contain many interfaces with some common characteristics.
1 0 - 12	When we are defining actions, we must write its arguments and what it returns. Every asynchronous action must have "in ACS::CBvoid cb, in ACS: CBDescIn desc". Note that beside arguments there is also a word "in". Here is short explanation of all possibilities (from Advance CORBA programming with C++): • in – the in attribute indicates that the parameter is sent from client to the server • out – the out attribute indicates that the parameter is sent from the server to the client • inout – The inout attribute indicates a parameter that is initialized by the client and sent to the server. The server can modify the parameter value, so, after the operation completes, the client-supplied parameter value may have been changed by the server <ac:structured-macro ac:name="unmigrated-wiki-markup" ac:schema-version="1" ac:macro-id="01990895-c808-4c39-9fc2-2e9c8412ee94"><ac:plain-text-body><![CDATA[You may also notice the abbreviation cb, which means callback and is a means of communication for asynchronous methods (you can read more about this in [R03], part 3.1 and [R02], part 3.3).
1 3 - 15	Here we define our properties as read-only attributes, which are read-only because we do not want someone to change them uncontrollably.

When finished writing an IDL, the file must be "compiled" by an IDL compiler. In the ACS environment, **you can simply use the acsMakefile**, but this will be discussed later. At this point there is no actual need to compile the IDL since the client and servant stubs (generated by TAO from the IDL) never even need to be viewed to write the C++ class!

The Configuration Database uses these files so that distributed objects can be instantiated with pre-configured data for their properties. More about how this is done can be found in the Configuration Database and XML Schemas for Configuration Database documents. For the PowerSupply Component we need to create three different XML files containing:

- An XML schema describing the IDL interface
- CORBA object information (i.e., the Component's name, location of PowerSupply's IDL interface, name of the PowerSupply shared library, and the name of the Container which will instantiate this Component) used by Manager
- The instance's actual configuration data to be used at runtime

While a running system can have many XML files describing the configuration data for a particular instance of a Component, there can be only one XML schema per Component, defining the structure and all default values for the configuration of that Component type. We will describe the case of the power supply component as it is in the acsexmpl module. This description is often considered a bit hard to understand at a first glance. As we said before we kindly suggest to read all the CDB documentation before proceeding. If you do not have time at the moment, we have added, at the end of this chapter, the case of the mount example that is simpler than the power supply. With the mount example we aim to help the developers in speeding up the process of creating new components. For the mount component, you should at least have a look on its IDL interface. On the other hand the mount component example is not enough to fully understand the interactions between xml, xsd in the CDB. A good idea could be that of fully understand the case of the mount then read again this chapter to extend your knowledge with the power supply example.

XML schema for PowerSupply (.../config/CDB/schemas/PowerSupply.xsd)

```
<xs:schema targetNamespace="urn:schemas-cosylab-com:PowerSupply:1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="urn:schemas-cosylab-com:PowerSupply:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:import namespace="urn:schemas-cosylab-com:CDB:1.0" schemaLocation="CDB.xsd"/>
  <xs:import namespace="urn:schemas-cosylab-com:BACI:1.0" schemaLocation="BACI.xsd"/>
  <xs:complexType name="PowerSupplyType">
    <xs:sequence>

      <xs:element name="current" type="baci:Rwdouble"/>

      <xs:element name="readback" type="baci:ROdouble"/>

      <xs:element name="status" >
        <xs:complexType>
          <xs:complexContent>
            <xs:restriction base="baci:ROpattern">
              <xs:attribute name="resolution" type="xs:int" use="optional"
                default="511" />
              <xs:attribute name="bitDescription" type="xs:string" use="optional"
                default="On,Remote,Sum Failure,External Interlock,
                DC Overcurrent,Phase Failure,Not Ready,State
                Inconsistent,Ramping"/>
              <xs:attribute name="whenSet" type="xs:string" use="optional"
                default="3, 2, 0, 0, 0, 0, 1, 1, 1"/>
              <xs:attribute name="whenCleared" type="xs:string" use="optional"
                default="2, 3, 3, 3, 3, 3, 3, 3, 3"/>
            </xs:restriction>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>

    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="optional" default="0"/>
  </xs:complexType>
  <xs:element name="PowerSupply" type="PowerSupplyType"/>
</xs:schema>
```

1 & 3	This is the namespace we will be using for PowerSupply. It is arbitrary but should include the name of the Component.
4- 5	The namespace that was used for BACI and CDB. These urns are not arbitrary and can be found by looking in \$ACSCROOT/config/CDB/schemas/*.

8-9	Please see 4-5.
10-38	These rows describe the type of the record in the xml file. The user is free to customize this complex type in order to add specific information (see CDB documentation for further information about this issue).
13-15	Here we just specify that current is of type RWdouble and readback is a ROdouble . Nothing else needs to be specified because these are simple types already defined in BACI.xsd.
17-34	Since status is a ROpattern (which is a complex type), minimal attributes have been specified in BACI.xsd. Because of this, it is necessary to specify a pattern or enum's attributes since they will not change for any Component instantiated from this schema.
39	We specify that the xml contain one record of the same name of the component. The type of the record is PowerSupplyType defined above (see 10-38)

The xml files are parsed against their schema definition. In PowerSupply.xsd the PowerSupply is defined to be of PowerSupplyType. By customizing the PowerSupplyType, the PowerSupply.xml will be accordingly different. If the developer would add some custom field to the configuration of the power supply, in the PowerSupply.xml file, he also has to change its schema definition. We suggest having a look on the CDB documentation about that. The acsexmplFilterWheel shows an example of a customized xsd/xml for a generic filter wheel.

Component information (\$ACS_CDB/CDB/MACI/Components/Components.xml).

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:COBs:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <_ Name="TEST_PS_1"
    Code="acsexmplPowerSupplyImpl"
    Type="IDL:alma/PS/PowerSupply:1.0"
    Container="bilboContainer"/>

  <_ Name="TEST_PS_2"
    Code="acsexmplPSOptimized"
    Type="IDL:alma/PS/PowerSupply:1.0"
    Container="bilboContainer2"/>

</Components>
```

This xml file describes the deployment information for components.

2	The namespace used by MACI Components as defined in \$ACSROOT/config/CDB/schemas/maciCOB.xsd
3-4	Please see 4-5 of the schema.
7-15	We must specify Component information used by Manager for each Component instance in the Configuration Database. That is, if we want to be able to use two PowerSupply objects named TEST_PS_1 and TEST_PS_2: entries containing the name of the PowerSupply shared library, Container that will instantiate each object, etc must exist in Components.xml
7	Name of the Component instance, which is also the name of the XML file containing the configuration data for said Component instance.
8	Specify the name of the Power Supply shared library that will be loaded by Container here.
9	Location of Power Supply's IDL interface within the interface repository. PS corresponds to the module name the PowerSupply interface is located within (inside acsexmplPowerSupply.idl). PowerSupply is simply the name of the interface to be used and 1.0 is version number of that interface.
10	The name of the Container, which will activate this CORBA object.

1	Notice we have specified acsexmplPSOptimized , which is a shared library for Power Supply that has been compiled with all optimizations turned
2	on (or even the source code for PowerSupply could be optimized). This is useful when you have a real-time system where every clock cycle counts,
-	but for development you wouldn't normally want to deal with the extra compile time. Finally, note that TEST_PS_2's Container is different from
15	TEST_PS_1. Use this when you want to take CPU overhead off of one Container (i.e., PC) and place it on another.

In this example we want to override some of the default values. One way to obtain that is to define a schema derived from the PowerSupply type. Another way might be that of customizing the xml definition. In this example we prefer to define a new schema as the same changes will be automatically propagated to all the power supply components.

We have defined the following schema that extends the PowerSupply.xsd Here you can find only a parte of the whole file. You can find the complete file in \$ACS_CDB/CDB/schemas/TestPowerSupplyACS.xsd..

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="urn:schemas-cosylab-com:TestPowerSupplyACS:1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="urn:schemas-cosylab-com:TestPowerSupplyACS:1.0"
xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0" xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
xmlns:powerSupply="urn:schemas-cosylab-com:PowerSupply:1.0" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:import namespace="urn:schemas-cosylab-com:CDB:1.0" schemaLocation="CDB.xsd"/>
  <xs:import namespace="urn:schemas-cosylab-com:BACI:1.0" schemaLocation="BACI.xsd"/>
  <xs:import namespace="urn:schemas-cosylab-com:PowerSupply:1.0" schemaLocation="PowerSupply.xsd"/>
  <xs:complexType name="TestPowerSupplyACS">
    <xs:complexContent>
      <xs:restriction base="powerSupply:PowerSupply">
        <xs:sequence>
          <xs:element name="current">
            <xs:complexType>
              <xs:complexContent>
                <xs:restriction base="baci:Rwdouble">
                  <xs:attribute name="description" type="xs:
string" use="optional" default="-" />
                  <xs:attribute name="format" type="xs:
string" use="optional" default="%9.4f" />
                  <xs:attribute name="units" type="xs:
string" use="optional" default="A" />
                  <xs:attribute name="resolution" type="xs:
int" use="optional" default="65535" />
                  <xs:attribute name="default_timer_trig"
type="xs:double" use="optional"
                  default="1.0" />
                  <xs:attribute name="min_timer_trig" type="
xs:double" use="optional"
                  default="0.001" />
                  <xs:attribute name="min_delta_trig" type="
xs:double" use="optional"
                  default="0.01526" />
                  <xs:attribute name="default_value" type="
xs:double" use="optional"
                  default="0.0" />
                  <xs:attribute name="graph_min" type="xs:
double" use="optional" default="0" />
                  <xs:attribute name="graph_max" type="xs:
double" use="optional"
                  default="1000.0" />
                  <xs:attribute name="min_step" type="xs:
double" use="optional"
                  default="0.01526" />
                  <xs:attribute name="min_value" type="xs:
double" use="optional" default="0.0" />
                  <xs:attribute name="max_value" type="xs:
double" use="optional"
                  default="1000.0" />
                </xs:restriction>
              </xs:complexContent>
            </xs:complexType>
          </xs:element>
          .....
        </xs:complexContent>
      </xs:restriction>
    </xs:complexType>
  </xs:schema>
```

2 & 26	The schema defines the TestPowerSupplyACS
7-8	Import the CDB and the BACI schemas
9	Import the PowerSupply schema that we will extend.
10-43	Define the type TestPowerSupplyACS
12	Define the TestPowerSupplyACS to be of PowerSupply type
14-41	Define the current property
17	The current property is a RWdouble property
18-37	Redefine all the attributes of the property.
44	Define the xml element to be of the type defined here (we use the same name for both the element and the type here but they can be different as shown in the PowerSupply.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<TestPowerSupplyACS xmlns="urn:schemas-cosylab-com:PowerSupplyACS:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:powerSupply="urn:schemas-cosylab-com:PowerSupply:1.0"
  xmlns:testPowerSupply="urn:schemas-cosylab-com:TestPowerSupplyACS:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <current xmlns="urn:schemas-cosylab-com:TestPowerSupplyACS:1.0"/>
  <readback xmlns="urn:schemas-cosylab-com:TestPowerSupplyACS:1.0"/>
  <status xmlns="urn:schemas-cosylab-com:TestPowerSupplyACS:1.0"/>
</TestPowerSupplyACS>
```

2 & 26	PowerSupplyACS corresponds to the name specified on line 44 of the XML schema and the schema namespace is also used here.
3-4	Please see 4-5 of the schema.
8-10	current , readback and status properties are defined in TestPowerSupplyACS schema.

A simple example: the mount.

We are aware that the power supply example is a bit complicated because we wanted to show how to override some of the default values for the properties. In the acsexmpl module there are several different examples. We briefly show also the case of the CDB configuration files for the mount component example. The mount has for read-only double properties, as you can read in its IDL definition file. The schema for the mount component See acsexmpl/ws/config/CDB/schemas/MOUNT.xsd. is:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  targetNamespace="urn:schemas-cosylab-com:MOUNT:1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="urn:schemas-cosylab-com:MOUNT:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0" elementFormDefault="qualified" attributeFormDefault="unqualified"
>
  <xs:import namespace="urn:schemas-cosylab-com:CDB:1.0" schemaLocation="CDB.xsd"/>
  <xs:import namespace="urn:schemas-cosylab-com:BACI:1.0" schemaLocation="BACI.xsd"/>
  <xs:complexType name="MOUNT">
    <xs:sequence>
      <xs:element name="cmdAz" type="baci:R0double"/>
      <xs:element name="cmdEl" type="baci:R0double"/>
      <xs:element name="actAz" type="baci:R0double"/>
      <xs:element name="actEl" type="baci:R0double"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="MOUNT" type="MOUNT"/>
</xs:schema>

```

2-19	The schema definition
3	The name space schema defines
4	The type the schema defines
11-16	The sequence with the four read-only properties (see the IDL definition file)
18	Defines the element of the xml file to be of the same type defined in line 10

As expected, the xml defining each component is also very simple:

```

<?xml version="1.0" encoding="UTF-8"?>
<MOUNT xmlns="urn:schemas-cosylab-com:MOUNT:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <cmdAz />
  <cmdEl />
  <actAz />
  <actEl />
</MOUNT>

```

2-10	The element is of type MOUNT as stated in line 18 of the schema definition
3-5	Import schemas and definitions
6-9	The four properties as stated in line 12-15 of the schema. Their type and default values are argued by the schema definition

(If you are using ALMA directory structure, header files should be located in the <xxx>/ws/include or <xxx>/include directory).
 Let us now create the C++ header file for the device server. Later, this file has to be included in the implementation file (.cpp). There is not much to do here – you first have to write a standard file header and then make the corresponding "#include"s for all of the properties you will use:

```

#ifndef acsexmplPowerSupplyImpl_h
#define acsexmplPowerSupplyImpl_h

#ifndef __cplusplus
#error This is a C++ include file and cannot be used from plain C
#endif

#include <baciCharacteristicComponentImpl.h>

#include <acsexmplExport.h>
#include <acsexmplPowerSupplyS.h>

#include <baciROdouble.h>
#include <baciRWdouble.h>
#include <baciROpattern.h>

#include <baciSmartPropertyPointer.h>

using namespace baci;

```

For ACS Component with characteristic or properties usage, the `baciCharacteristicComponentImpl.h` file is needed. `acsexmplExport.h` is used for exporting this example to the Microsoft Windows environment. Also, `acsexmplPowerSupplyS.h` is automatically generated with `tao_idl` (it has the same name as the IDL with an "S" appended at the end) and header files for all properties that are used (`baciROdouble.h` and `baciROpattern.h`) must be included. The `baciSmartPropertyPointer.h` file includes the declaration for the smart pointers. They help to write the component in a clear and reliable way. The following example is written using smart pointers but it is also possible to use the properties without even if we suggest using the smart pointers with properties.

```

#define ON_ACTION    0
#define OFF_ACTION   1
#define RESET_ACTION2

/**
 * The class PowerSupply simulates the behaviour of a power supply.
 * (...doxygen comment...)
 */
class acsexmpl_EXPORT PowerSupply: public virtual CharacteristicComponentImpl,
                                   public virtual POA_PS::PowerSupply,
                                   public ActionImplementator
{

```

Asynchronous call handling implementation requires that action numbers are defined for each call (here `on()`, `off()` and `reset()`).

`PowerSupply` must inherit from the class `CharacteristicComponentImpl` because the IDL interface does, `ActionImplementator` for asynchronous methods, and `POA_<name of module in IDL>::<name of interface in IDL>` which is auto-generated by `tao_idl`. The `CharacteristicComponentImpl` class incorporates standard methods and macros necessary for the CORBA interface implementation and MACI-DLL support.

Now we can define public methods and fields (note: because of inheritance, you should put **virtual** before the destructor and other methods):

```

public:

    /**
     * Constructor
     * @param poa poa which will activate this and also all other COBs
     * @param name Component name
     */
    PowerSupply(const ACE_CString& name, maci::ContainerServices* containerServices);

    /**
     * Destructor
     */
    virtual ~PowerSupply();

    /**
     * Override component lifecycles method
     */
    virtual void execute() throw (acsErrTypeLifeCycle::LifeCycleExImpl);

```

```

/* ----- [ Action implementator interface ] ----- */

/**
 * Action dispatcher function - called from a method inherited from
 * ActionImplementator class whenever there is an asynchronous method waiting in a
 * queue.
 * (...doxygen comment...)
 */
virtual ActionRequest
invokeAction(int function, BACICComponent* cob_p, const int& callbackID,
             const CBDescIn& descIn, BACIValue* value, Completion& completion,
             CBDescOut& descOut);

/**
 * Implementation of async. on() method - called by invokeAction
 */
virtual ActionRequest
onAction(BACICComponent* cob, const int& callbackID, const CBDescIn& descIn,
        BACIValue* value, Completion& completion, CBDescOut& descOut);

/**
 * Implementation of async. off() method - called by invokeAction
 */
virtual ActionRequest
offAction(BACICComponent * cob, const int& callbackID, const CBDescIn& descIn,
        BACIValue* value, Completion& completion, CBDescOut& descOut);

/**
 * Implementation of async. reset() method - called by invokeAction
 */
virtual ActionRequest
resetAction(BACICComponent * cob, const int& callbackID, const CBDescIn& descIn,
        BACIValue* value, Completion& completion, CBDescOut& descOut);

/* ----- [ CORBA interface ] ----- */

/**
 * Switches on the power supply - registers the call to the asynch. queue
 * (...doxygen comment...)
 */
virtual void
on(ACS::CBvoid_ptr cb, const ACS::CBDescIn & desc)
    throw(CORBA::SystemException);

/**
 * Switches off the power supply - registers the call to the asynch. queue
 * (...doxygen comment...)
 */
virtual void
off(ACS::CBvoid_ptr cb, const ACS::CBDescIn & desc)
    throw(CORBA::SystemException);

/**
 * Resets the power supply - registers the call to the asynch. queue
 * (...doxygen comment...)
 */
virtual void
reset(ACS::CBvoid_ptr cb, const ACS::CBDescIn & desc)
    throw(CORBA::SystemException);

/**
 * Property current contains the commanded current of the power supply.
 */
virtual ACS::Rwdouble_ptr
current()
    throw(CORBA::SystemException);

/**
 * Property readback is the actual current obtained from the physical device.
 */

```

```

virtual ACS::R0double_ptr
readback()
    throw(CORBA::SystemException);

/**
 * Property status contains the status of the power supply.
 */
virtual ACS::R0pattern_ptr
status()
    throw(CORBA::SystemException);

```

The private fields usually consist of all variables that are in the program. We use a smart pointer for each property. The type of each smart pointer is a template based on the type of the related property. The smart pointers for the properties take care of all the CORBA detail in behalf of the developer making the programming of a device server easier, faster and safer.

```

private:

    /// Properties
    SmartPropertyPointer<R0double> m_readback_sp;
    SmartPropertyPointer<RWdouble> m_current_sp;
    SmartPropertyPointer<R0pattern> m_status_sp;

};

#endif /* acsexmplPowerSupplyImpl_h */

```

The execute method is part of the life cycle methods. There are four methods for the life cycle. They are defined in `acscomponent` and can be overridden by the developer.

```

virtual void initialize()
    throw(acsErrTypeLifeCycle::acsErrTypeLifeCycleExImpl);

virtual void execute()
    throw(acsErrTypeLifeCycle::acsErrTypeLifeCycleExImpl);
    /*
virtual void cleanUp();

virtual void aboutToAbort();

```

The **initialize** method is called by the container after instantiating the component i.e. after executing the constructor. The developer should insert here all the code related to the initialization of the component like, for example retrieve connections, read in configuration files or parameters, build up in-memory tables and so on. This method is called before any functional requests can be sent to the component.

The **execute** method is called by the container after the **initialize** to signal that the server has to be ready to accept functional requests.

The developer in the **execute** and the **initialize** can throw an exception to signal a malfunction to the container. In this case the exception is caught by the container and the component will be released and unloaded.

The **aboutToAbort** is called by the container in case of an error and an emergency situation requests the component to be destroyed. The developer should insert here the code to handle this situation trying to release resources and so on. This method is called in an emergency situation and there is no warranty that the container will wait its termination before destroying the component.

Finally, the **cleanUp** method is called by the container before destroying the server in a normal situation. The developer should release all the resource and perform all the clean up here.

As a guideline, we suggest to leave the constructor and the destructor of the server empty moving the code in the life cycle method. This should help in writing more reliable servers.

There is no need to call the parent life cycle method

In the power supply component, **execute** is the only life cycle method overridden.

Together with the life cycle methods there is the concept of the state of the component i.e. the component passes through different states during its life, since it is built until it is destroyed. The state is transparent to the user and managed by the `ComponentStateManager` object that is part of the `ContainerServices`. The state of the component is managed by the container and must not be accessed nor modified by the developer. For completeness I report here the states that a component can assume, as defined in `acscomponent.idl`:

COMPSTATE_NEW	This is the initial state of the component
COMPSTATE_INITIALIZING	When the component is executing the initialize method
COMPSTATE_INITIALIZED	When the initialize method terminates
COMPSTATE_OPERATIONAL	When the execute method terminates
COMPSTATE_ERROR	An error occurred
COMPSTATE_DESTROYING	Before executing the cleanUp method
COMPSTATE_ABORTING	Before executing the aboutToAbort method
COMPSTATE_DEFUNCT	Before destroying the component

(If you are using ALMA directory structure, implementation files should be located in <xxx>/ws/src or <xxx>/src directory.)

Tao has already generated some files needed for client-server communication, but now we have to implement our device server. I will divide the C++ file into smaller parts and explain each of them. The beginning of the program is pretty much standard, so I will just write about unusual lines. (Note: code marked with **bold** should be adapted for new applications):

```
#include <vltPort.h>

static char *rcsId="@(#) $Id: $";
static void *use_rcsId = ((void)&use_rcsId,(void *) &rcsId);

#include <baciDB.h>
#include <acsexmplPowerSupplyImpl.h>

using namespace baci;
```

- | | |
|---------|---|
| 6
-7 | Because you are using BACI with database access, you have to include baciDB.h. The third include is a header file for this program, usually with same name, but ending with .h instead of .cpp. |
|---------|---|

Now we have to write a constructor:

```
PowerSupply::PowerSupply(const ACE_CString& name, maci::ContainerServices * containerServices):
    CharacteristicComponentImpl(name,containerServices)
    m_readback_sp(new R0double(name+":readback", getComponent()),this),
    m_current_sp(new R0double(name+":current", getComponent()),this),
    m_status_sp(this)
{
    ACS_TRACE("::PowerSupply::PowerSupply");

    // properties
    m_status_sp = new R0pattern(name+":status", getComponent());
}
```

- | | |
|---------|---|
| 1 | A class constructor is always the method with the name of the class.
The constructor receives the name of the component as well as a pointer to the ContainerServices as parameters. We'll say more about ContainerServices later.
There is no need to store the container services and the name in local variable as it is done by the component. The name is returned by the name() method; the container services is always available from the getContainerServices() method. |
| 3
-5 | In the constructor we create properties. Their names must be composed of the name of the server and property name (for example: TestPowerSupply1:current). In property constructor you must also provide a reference to the component. Here the getComponent() method (inherited from CharacteristicComponentImpl) is used to obtain the Component reference. Every Component has its own instance, which takes care of threads (dispatching). It must have a name and it's Unique Identification Number (UID). UID should always be the same (even when server is restarted) and it will be read from local database. There may not be two servers with the same name and/or same UID!
There are two constructors for smart pointers; both of them need a pointer to the characteristic component (this). In the first case (rows 3 and 4) in the constructor is also passed a pointer the newly created property while in row 5 you see that the smart pointer m_status_sp has no property yet. This second initialization of a smart pointer can be useful if you cannot build the property in the initialization list (for example the devIO for the property does not exist yet). |
| 10 | The created property is assigned to the smart pointer. This step is necessary if the smart property is created without a pointer to a property in the constructor (see row 5).
Smart pointers perform error checking and generate a description of the properties. |

The destructor is empty. The entire cleanup for the properties is managed by the smart pointers.

```

PowerSupply::~PowerSupply()
{
}

```

Let us come to actions now. All requests for asynchronous actions go through the Activator, which sends this request back to the device server. This is done through the `invokeAction()` method, where you define what has to be done when a call is received:

```

/* ----- [ Action implementator interface ] ----- */

ActionRequest
PowerSupply::invokeAction(int function, BACComponent* cob, const int& callbackID,
    const CBDescIn& descIn, BACValue* value, Completion& completion,
    CBDescOut& descOut)
{
    switch (function)
    {
        case ON_ACTION:
            return onAction(cob, callbackID, descIn, value, completion, descOut);
        case OFF_ACTION:
            return offAction(cob, callbackID, descIn, value, completion, descOut);
        case RESET_ACTION:
            return resetAction(cob, callbackID, descIn, value, completion, descOut);
        default:
            return reqDestroy;
    }
}

```

The implementation of the functions we defined in the previous method:

```

/* ----- [ Action implementations ] ----- */

/// implementation of async. on() method
ActionRequest
PowerSupply::onAction(BACComponent* cob, const int& callbackID,
    const CBDescIn& descIn, BACValue* value,
    Completion& completion, CBDescOut& descOut)
{
    ACS_DEBUG_PARAM("::PowerSupply::onAction", "%s", getComponent()->getName());

    .....

    completion = ACSErrTypeOK::ACSErrOKCompletion();

    // complete action requesting done invocation,
    // otherwise return reqInvokeWorking and set descOut.estimated_timeout
    return reqInvokeDone;
}

```

11	Here comes real implementation of the action – like communication with physical device, setting it on, off, etc. (for example: a <code>devIO->write(...)</code>). If action will not be completed in <code>descIn.timeout</code> or action is slow in progress, you should first return "working" (return <code>reqInvokeWorking</code>) and also set estimated timeout (<code>descOut.estimated_timeout</code>) (more about it in [R02], part 3.3.1.2).
13	To inform client about possible errors, set completion (read [R02], part 3.2). In this case everything is ok and we set completion to <code>OkCompletion</code> .
18	When action is completed, return done (return <code>reqInvokeDone</code>).

We have to describe the device server's actions (like `on()`, `off()`, `reset()`). These are asynchronous (usually they take some time and we do not want to block a client while executing) so we have to register the action to the Activator that will then do the rest of the procedure. For example, calling `on()` will just register that action inside a queue until Activator calls `PowerSupply::invokeAction` which will in turn, call `onAction()`.

The client must provide a pointer to its implementation of callbacks and the structure of type `CBDescIn` (that identifies the callback) as an argument of the `registerAction()` method. Our device server sends these parameters to Activator and also adds a pointer to itself and a description of an action (integer – `ON_ACTION` will be replaced with 1, `OFF_ACTION` with 2 and so on).

Your job is to write one block of a code for each action and change only the last argument of the method `registerAction` – an integer which is later sent back to the device server, to the method `invokeAction()`. First argument of `registerAction()` method – in this case `BACValue::type_null` – is a callback type.

```

void
PowerSupply::on(ACS::CBvoid_ptr cb, const ACS::CBDescIn & desc)
    throw(CORBA::SystemException)
{
    getComponent()->registerAction(BACIValue::type_null, cb, desc, this, ON_ACTION);
}

void
PowerSupply::off(ACS::CBvoid_ptr cb, const ACS::CBDescIn & desc)
    throw(CORBA::SystemException)
{
    getComponent()->registerAction(BACIValue::type_null, cb, desc, this, OFF_ACTION);
}

void
PowerSupply::reset(ACS::CBvoid_ptr cb, const ACS::CBDescIn & desc)
    throw(CORBA::SystemException)
{
    getComponent()->registerAction(BACIValue::type_null, cb, desc, this, RESET_ACTION);
}

```

throw(...) defines what kind of exceptions are available.

If a client wants to do something with current, readback, and other properties, it must get a reference to these properties. This is done in following code:

```

ACS::Rwdouble_ptr
PowerSupply::current()
    throw(CORBA::SystemException)
{
    if (m_current_sp==0)
        return ACS::Rwdouble::_nil();

    ACS::Rwdouble_var prop = ACS::Rwdouble::_narrow(m_current_sp->getCORBAReference());
    return prop._retn();
}

ACS::R0double_ptr
PowerSupply::readback()
    throw(CORBA::SystemException)
{
    if (m_readback_p==0)
        return ACS::R0double::_nil();

    ACS::R0double_var prop = ACS::R0double::_narrow(m_readback_sp->getCORBAReference());
    return prop._retn();
}

ACS::R0pattern_ptr
PowerSupply::status()
    throw(CORBA::SystemException)
{
    if (m_status_p==0)
        return ACS::R0pattern::_nil();

    ACS::R0pattern_var prop = ACS::R0pattern::_narrow(m_status_sp->getCORBAReference());
    return prop._retn();
}

```

One must write accessors for every property, changing the name of a property (current), variable (m_current_sp) and type of a property (replace all fields where Rwdouble appears with R0double or R0pattern).

Now we are really near the end. We have to add MACI DLL support functions. For this purpose we use the macro MACI_DLL_SUPPORT_FUNCTIONS (class_name):

```

/* ----- [ MACI DLL support functions ] ----- */

#include <maciACSComponentDefines.h>
MACI_DLL_SUPPORT_FUNCTIONS (PowerSupply)

/* ----- */

```

As we said at the beginning of the paragraph, it is possible to use the properties without smart pointers even if we discourage this implementation. We briefly describe herein the changes in the previous code if smart pointers were not used.

In the include file each property is defined without the templetized smart pointer type and, of course, the `baciSmartPropertyPointer.h` is not needed. Note that the names of the variables terminate with `_p` and not with `_sp` to highlight that we are now using pointers instead of smart pointers.

ROdouble* m_readback_p

ROdouble* m_readback_p;

RWdouble* m_current_p;

In the constructor these pointers are first initialized to 0 then each property is created in the same way they are created with smart pointers. The `CHARACTERISTIC_COMPONENT_PROPERTY` macro must be implemented for each BACI property.

The macro performs error checking and generates a description of the property. This macro must not be used with smart pointers because they take care of all the initializations automatically.

The following is the constructor without using smart pointers for the properties.

```

PowerSupply::PowerSupply(const ACE_CString& name,
                        maci::ContainerServices * containerServices):
    CharacteristicComponentImpl(name, containerServices),
    m_readback_p(0),
    m_current_p(0),
    m_status_p(0) {
    m_readback_p = new ROdouble(name+":readback", GetComponent());
    CHARACTERISTIC_COMPONENT_PROPERTY(readback, m_readback_p);

    m_current_p = new RWdouble(name+":current", GetComponent());
    CHARACTERISTIC_COMPONENT_PROPERTY (current, m_current_p);

    m_status_p = new ROpattern(name+":status", GetComponent());
    CHARACTERISTIC_COMPONENT_PROPERTY(status, m_status_p);
}

```

Another difference is in the destructor because for each property we must call the destroy method but not the delete.

The rest of the code remains the same apart for the renaming of the variables.

```

PowerSupply::~PowerSupply() {
    ...
    (m_readback_p) { m_readback_p->destroy(); m_readback_p = 0; }
    (m_current_p) { m_current_p->destroy(); m_current_p = 0; }
    if (m_status_p) { m_status_p->destroy(); m_status_p = 0; }
}

```

As we saw writing the constructor of the component, it receives a pointer to a `maci::ContainerServices` object. Really, `maci::ContainerServices` is an interface and the component receives an implementation of that interface defined in the `maci` module. The developer is free to replace this default implementation with another one that better fits its needs.

Throw the `ContainerServices` object the `Containers` offers the component a set of services freeing the developer of the details of their implementation.

The `getContainerServices()` method returns a pointer to the container services object so there is no need to store the container services object received in the constructor in a local variable.

Here I report a brief description of the services of the default implementation of the `ContainerServices` in the `maci` module. For further details, have a look at the `doxygen` documentation.

```

ACE_CString getName() // Return the name of the component

PortableServer::POA_var getPOA() // Return a reference to the POA

// Get a Component of type T
template<class T> T* getComponent(const char *name)

// Get a dynamic component of type T
template<class T> T* getDynamicComponent(maci::ComponentSpec
                                         compSpec, bool markAsDefault);

// Get the default component (of type T) that implements the idl interface
template<class T> T* getDefaultComponent(const char* idlType);

// get the descriptor of a component
maci::ComponentInfo getComponentDescriptor(const char* componentName)

// Find components using their name or their type
ACE_CString_Vector findComponents(const char *nameWildcard, const char *typeWildcard)

void releaseComponent(const char *name) // release the component with the given name

void releaseAllComponents() // Release all the components

CDB::DAL_ptr getCDB() // Return a reference to DAL that allows accessing the CDB

PortableServer::POA_var getOffShootPOA() // Return the offShoot POA

// Activate a CORBA servant that implements the offshoot interface
ACS::OffShoot_ptr activateOffShoot(PortableServer::Servant cbServant)

// Deactivate a CORBA offShoot servant
void deactivateOffShoot(PortableServer::Servant cbServant)

```

```

ACS_TRACE("::PowerSupply::~~PowerSupply");

ACS_DEBUG("::PowerSupply::~~PowerSupply", "COB destroyed");

ACS_LOG(LM_RUNTIME_CONTEXT, "PowerSupply/DLLOpen",
        (LM_ERROR, "Failed to create Component - constructor returned 0!"));

```

Find out more about this in [Logging and Archiving](#)

To compile, please use the acsMakefile and the ALMA directory structure. All files should be in subdirectories of <xxxx>/ws/ or <xxxx>/, where <xxxx> represents the name of module. If you have made it this far, you should have these files (although the names should be different...):

- <xxxx>/idl/acsexmplPowerSupply.idl
- <xxxx>/src/acsexmplPowerSupplyImpl.cpp
- <xxxx>/include/acsexmplPowerSupplyImpl.h
- <xxxx>/config/CDB/schemas/PowerSupply.xsd
- <xxxx>/test/CDB/MACI/Components/Components.xml
- <xxxx>/test/CDB/alma/TEST_PS_1/TEST_PS_1.xml

Next, create a Makefile (or preferably just change the existing one). It should be located in: <xxxx>/src/Makefile

Since the acsMakefile is rather large and most targets will not be modified, I am going to write only the parts where something has to be written/changed.

Name of main header file – the one, which is described in this document:

```

#
# Includes (.h) files (public and local)
# -----
INCLUDES      = acsexmplPowerSupplyImpl.h
INCLUDES_L

```

The name, which is written behind "LIBRARIES =", will be the name of the created library. For each library "<name of library>_OBJECTS" has to be defined:

```
#
# Libraries (public and local)
# -----
LIBRARIES      = acsexmplPowerSupplyImpl
LIBRARIES_L    =
acsexmplPowerSupplyImpl_OBJECTS = acsexmplPowerSupplyImpl \
                                acsexmplPowerSupplyCurrentImpl acsexmplPowerSupplyDLL\
acsexmplPowerSupplyImpl_LIBS    = acsexmplPowerSupplyStubs
```

Name of database configuration file:

```
#
# Configuration Database Files
# -----
CDB_SCHEMAS = PowerSupply
```

Name of IDL file:

```
#
# IDL FILES
#
IDL_FILES = acsexmplPowerSupply
USER_IDL =
```

Then you have to run it:

```
make clean all man install
```

When the compiler finishes, it should create files lib<name of libraries>.so and .a (in our case libacsexmplPowerSupply.so and libacsexmplPowerSupply.a). When you use the "install" tag, binaries, libraries documentation files, etc., will be copied into the \$INTROOT directory.

Appendix

A number of insignificant details have been removed from the PowerSupply example in this document. For the most up-to-date and complete documentation, please go to <http://www.eso.org/~gchiozzi/AlmaAcs/OnlineDocs/acsexmpl/ws/doc/html/index.html>. This webpage also contains documentation on even more examples that can be found in the ALMA CVS repository (ACS/LGPL/CommonSoftware/acsexmpl/*).