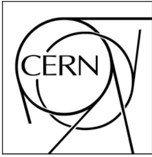
XDAQ Porting Guide

From Version 2.x to 3.0

Johannes Gutleber
Luciano Orsini
CERN, February 2005



CERN
European Organization for Nuclear Research
Department PH/CMD
1211 Geneva 23
Switzerland

Take note!
Before using this information and the product it supports, be sure to read the general information in Appendix: Special Notices on page 33.

First Edition (February 2005)
This edition applies to XDAQ V3.0 for Linux, Program Number xdaqcore_G_17559.
Comments may be addressed to:
Johannes Gutleber and Luciano Orsini
CERN, European Organization for Nuclear Research
Dept. PH/CMD Building 40 Internal Mailbox E-25810
1211 Geneva 23
Switzerland
© Copyright CERN 2005. All rights reserved.

1. Introduction *4*

2. Syntax Upgrade *5*

2.1. Include Files **6**

2.2. Namespaces **7**

2.3. Application Class Declaration **8**

2.4. Binding to Incoming Messages **9**

2.5. Exported Variables **11**

2.6. Intercepting Parameter Operations **13**

2.7. Application Instantiation **14**

2.8. Exceptions **15**

2.9. Application Identifiers **17**

2.10. Sending I_2O Messages **18**

2.11. Sending SOAP Messages **19**

2.12. Memory Management **22**

2.13. State Machines **24**

3. Makefiles *26*

4. XML Configuration *27*

5. Peer Transports *29*

6. xdaqWin *30*

7. Getting Started with XDAQ *31*

7.1. Installation **31**

7.2. Running the Process **32**

8. Appendix: Special Notices *33*

1. Introduction

The XDAQ Porting Guide is intended to help experienced developers convert existing XDAQ V2.x applications into applications that can run on XDAQ V3.0. It contains detailed information about how to adapt and build your application using the revised API as well as step-by-step instructions of the porting process.

Although V3.0 contains substantial API revision, almost no semantic changes are visible at application level. It is important to keep this in mind during the porting. Therefore the general guideline for porting applications can be condensed to the following three simple steps.

1. Only perform syntax modifications to have a compiling program for V3.0
2. Make the syntactically upgraded program run in V3.0
3. Add new features or exchange legacy features if required.

This porting guide deals with the first and biggest step that prepares you to have your application running. The second step should then be straightforward and the indications on how to run the xdaq process and the improved development/testing environment described in this document should support you in carrying out this task.

If you have not yet installed XDAQ V3.0 please see section 7. This section also gives instructions on how to start the XDAQ process.

2. Syntax Upgrade

Familiarize yourself with the XDAQ 3 API by running Doxygen on the XDAQ 3 source code and to look up the XDAQ 3 equivalents of the XDAQ 2 API. Doxygen is a program that produces HTML pages of documentation that has been embedded by the programmers into the source code. It is usually installed on the Linux operating systems at CERN. If it is not installed, please get it from www.doxygen.org and follow the usage instruction.

If you are less familiar with XDAQ, start having a look at the example programs in `TriDAS/daq/examples/include`. You will find the following programs (there may be additional ones, but the listed ones are the most important):

`HelloWorld.h`..........the simplest XDAQ application (start here)
`SimpleSOAPReceiver.h`..........how to receive a SOAP message
`SimpleWeb.h`..........how to make a Web browser viewable application
`SOAPStateMachine.h`..........Application with a state machine that can be driven using SOAP messages
`WebStateMachine.h`..........Application with a state machine that can be driven through a Web page using a browser
`AsynchronousSOAPStateMachine.h`...Application with a state machine that can be driven using SOAP messages. Transitions are performed asynchronously in a thread.

We also advise you to have a look at the *RoundTrip* example program. This program can be found in the distribution in the `TriDAS/daq/benchmark/roundtrip/` directory. Please have a look at the header and the source files in

```
TriDAS/daq/benchmark/roundtrip/include/RoundTrip.h
TriDAS/daq/benchmark/roundtrip/src/common/RoundTrip.cc
```

This example program uses most of the functionality that XDAQ 3.0 provides within a single application. It is therefore an ideal entry point to understand which items require porting.

2.1. Include Files

With the introduction of independent packages that can be used also for stand-alone applications and the usage of C++ namespaces, include files end up in subdirectories under `<package name>/include`. For example, *toolbox* general files are found in `toolbox/include/toolbox`. *Toolbox* functions in subpackages that are specific for some operations are then found in `toolbox/include/toolbox/<subpackage name>`. Accordingly, *xdaq* core include files are found in `TriDAS/daq/xdaq/include/xdaq`.

To use these new include directories, specify the

```
#include"subpackagename/filename.h"
```

in your program.

To include the XDAQ abstract `Application` class that is found in `TriDAS/daq/xdaq/include/xdaq/Application.h`, use for example

```
#include"xdaq/Application.h"
```

2.2. Namespaces

XDAQ 3 uses namespaces in all modules and libraries. Toolbox functions are for instance preceeded by the `toolbox::` qualifier. Namespaces can be stacked and the toolbox has for example sub-namespaces like `toolbox::mem` or `toolbox::net`.
A view legacy classes such as the `Task` or `SyncQueue` are not yet moved to a namespace to emphasize the change. They should be considered deprecated and serve only for coming to running XDAQ V3.0 programs. You should consider replacing them with new functionalities in the near future after you have familiarized yourself better with XDAQ V3.0.

Look up all functions that you use and figure out which namespace qualifier needs to be added.

XDAQ applications (applications that have class-name and instance that that you use to declare in an XML configuration file) do not have any namespace.

Do not put your XDAQ application class into a namespace.

2.3. Application Class Declaration

`xdaqApplication` used to be the class from which user applications inherited in V2.x. In V3.0 you can choose to inherit from one of two base classes:

1. `xdaq::Application`
2. `xdaq::WebApplication`

While `xdaq::Application` is the replacement for the former `xdaqApplication`, `xdaq::WebApplication` shall be used as a base for applications that wish to expose HTML pages that can be viewed using a browser.

As opposed to version 2.x, no implicit binding of pre-defined SOAP or I_2O messages is done in version 3.0. Therefore it is possible to receive messages of any protocol type in any application, whether they extend the `xdaq::Application` or the `xdaq::WebApplication` interface.

2.4. Binding to Incoming Messages

In order to intercept incoming messages, an application must *bind a callback* to a protocol/service type combination. Currently three protocol/service types are supported in V3.0

1. SOAP (XML messages over http)
2. HTTP/CGI (data encapsulated in direct http POST/GET requests)
3. I_2O (binary data over various lower transport levels)

In order to **bind to SOAP messages** do

```
#include "xoap/Method.h"
xoap::bind(applicationObject, &callback, soapMessageName, namespace );
```

`applicationObject`..................pointer to XDAQ application object (e.g. `this`)
`callback`...pointer to callback function
`soapMessageName`........................text string with incoming SOAP function to bind to
`namespace`..text string with SOAP function namespace URI

SOAP callbacks have to implement the following signature:

```
xoap::MessageReference callback (xoap::MessageReference msg)
             throw (xoap::exception::Exception);
```

XDAQ V3.0 implements the standard SOAP specifications. Therefore, namespaces are used in the SOAP XML messages. In order for an XDAQ application incercept incoming SOAP messages, it has to be informed which namespace string is used in the incoming XML message. The XDAQ core uses a predefined namespace string called `XDAQ_NS_URI` that resolves to the text string `urn:xdaq-soap:3.0`. This namespace declaration may also be used for user applications, but then the programmer must be aware of the fact that name clashes may occur any time. We recommend to use your own namespace declarations and to follow the URN notation (e.g. `urn:`<*application name*>`-soap:`<*version number*>).

Note that the callback signature defines an exception. Only this exception may be thrown in the user application callback. If your legacy callback implementation throws another type of exception, catch it and convert it using the following macro:

```
XCEPT_RETHROW (xoap::exception::Exception, "Message", yourException);
```

Note that `yourException` must inherit from the *xcept::Exception* base class. See more on the revised exception handling in the associated section 2.8.

In order to **bind to HTTP/CGI messages** do

```
#include "xgi/Method.h"
xgi::bind(applicationObject, &callback, URLPath );
```

`applicationObject`..................pointer to XDAQ application object (e.g. `this` pointer)
`callback`...pointer to callback function
`URLPath`...text string with incoming URL to bind to

HTTP/CGI callbacks have to implement the following signature:

```
void callback (xgi::Input * in, xgi::Output * out )
             throw (xgi::exception::Exception)
```

Since HTTP/CGI was not enabled in V2.x the interested reader is pointed to the example programs and the documentation of the CGICC package at *www.gnu.org/software/cgicc/cgicc.html*.

In order to **bind to I_2O messages** do

```
#include "i2o/Method.h"
i2o::bind(applicationObject, &callback, I2OFunctionCode, I2OFunctionClass );
```

Example:

```
i2o::bind (this, &RoundTrip::token, I2O_TOKEN_CODE, XDAQ_ORGANIZATION_ID);
```

`applicationObject`……pointer to XDAQ application object (e.g. this)
`callback`……pointer to callback function
`I2OFunctionCode`……Numeric identifier of the incoming I_2O message (included in the message header) used to associate the callback function
`I2OFunctionClass`……A numeric offset to associate function identifiers to regions in order to avoid clashes between applications that use the same function identifiers. The macro `XDAQ_ORGANIZATION_ID` resolves to `0x0` and therefore no offsetting will be performed when using this code.

I_2O binary callbacks have to implement the following signature:

```
void callback (toolbox::mem::Reference* arg)
             throw (i2o::exception::Exception)
```

The user in this callback using must recycle the received memory buffer with

```
arg->release();
```

The V2.x `frameFree()` no longer exists.
The `i2oListener` class no longer exists.
`i2oBindMethod()` no longer exist.

2.5. Exported Variables

Application variables that used to be exported in V2.x with the `exportParam` call are now exported by putting them into an "*Infospace*". *Infospaces* are a new feature in V3.0.

> *An Infospace is a container for variables that can be shared among XDAQ applications within a single process.*

Additions will later on extend the sharing of variables over the network to applications to different processes and computers.
By default each XDAQ application in V3.0 has an *Infospace* for the variables that it wishes to make visible. The default application *Infospace* is accessed by the following call within an application:

```
this->getApplicationInfoSpace();
```

Variables are "exported" using

```
getApplicationInfoSpace()->fireItemAvailable(
              const string & name,
              xdata::Serializable * serializable,
              void * originator = 0
              )
              throw (xdata::exception::Exception);
```

`name`.......................................the name through which the variable can be seen (no white-spaces allowed)
`serializable`..............pointer to an exportable variable (needs to be an `xdata::Serializable`)
`originator`.....................cookie to remember who published the variable

Example: to export a number and a string declare the following instance variables:

```
xdata::UnsignedLong   counter_;
xdata::String         hostname_;
```

In the constructor of your application, replace `exportParam` with

```
getApplicationInfoSpace()->fireItemAvailable("counter",&counter_);
getApplicationInfoSpace()->fireItemAvailable("hostname",&hostname_);
```

> *All exported variables are `xdata::` variables.*

You need to replace all V2.x `xdaq`*`xx`* datatypes or native C++ datatypes of exported variables. Xdata is a new package in V3.0.

In V2.x it was possible to directly print out exported variables since it was possible to pass native C++ datatypes to `exportParam`. In V3.0 you need to cast `xdata::` datatypes explicitly to the native datatype contained before putting them as parameters to a `cout` or `printf` statement.

Example:

```
std::printf ("A number: %d, a string: %s\n",
             (unsigned long) counter_, (std::string) hostname_);
std::cout << "A number " << (unsigned long) counter_
          << ", a string: " << (std::string) hostname_);
```

Arithmetic operations on `xdata::` datatypes are not defined at this point. To increment an `xdata::UnsignedLong` for example you must do one of the following

1) `i = i + 1;`
2) `i.value_++;`

Option one is preferred at this point, since the direct access of the internal variable `value_` may be replaced by an API function in later versions.

2.6. Intercepting Parameter Operations

In V2.x very specific calls have been provided to intercept settings and retrievals of exported parameters. A special callback was provided to intercept the setting of the default parameters. In V3.0 the model has been generalized. No specific calls are needed anymore. Instead, the user attaches listeners to the *Infospace* and can intercept read or write operations at any time.

Use the Infospace API to intercept parameter access/modifications

In order to attach a listener, we recommend that the application also inherits from `xdata::ActionListener`. Then a callback with the following signature can be declared:

```
void actionPerformed (xdata::Event& e);
```

The callback can be attached to various events in the constructor, here for example to the retrieval of an "exported" parameter. For each exported variable another listener may be attached. Such the granularity can be tuned according to the applications needs. The first parameter is the text identifier of the parameter to be attached to, the second parameter is a pointer to the callback listener object (in this case the xdaq application that also inherits from `xdata::ActionListener`).

```
getApplicationInfoSpace()->addItemRetrieveListener ("counter", this);
getApplicationInfoSpace()->addItemRetrieveListener ("hostname", this);
```

It is possible to attach to the following events:

- `addItemAvailableListener(xdata::ActionListener * l)`
  listen to the addition of variables to the *Infospace*
- `addItemRevokedListener(xdata::ActionListener * l)`
  listen to the removal of variables from the *Infospace*
- `addItemChangedListener(const string& name, xdata::ActionListener * l)`
  called back when the item called "name" *has* been written
- `addItemRetrieveListener(const string& name, xdata::ActionListener * l)`
  called back when the item called `name` *is going to be read* (before it is really read – that offers the opportunity to assign values to the variable just before the variable is read)

There is not listener yet for the use case that an application wants to intercept right before the variable is going to be assigned a new value. That will be included in a further version of XDAQ. There is not specific listener anymore for the setting of the default parameters.
It is still possible to directly intercept the SOAP `ParameterGet` and `ParameterSet` calls by overriding the

- `virtual xoap::MessageReference ParameterGet (xoap::MessageReference message) throw (xoap::exception::Exception);`
- `virtual xoap::MessageReference ParameterSet (xoap::MessageReference message) throw (xoap::exception::Exception);`

functions. However, you must at the end of your implementation call the `xdaq::Application` implementations of these functions using

- `xdaq::ParameterGet(message);`

- `xdaq::ParameterSet(message);`

2.7. Application Instantiation

All application loader classes (the *application*SO class) can be removed. There is no `plugin` function anymore. Applications can already perform useful work in the constructor since the whole environment is ready at this time.

Applications are instantiated automatically.

The user must put the following macro statement in the implementation file (`.cc`) of his application

```
XDAQ_INSTANTIATE (ApplicationClassName)
```

ApplicationClassName is the exact C++ class name of the XDAQ application class that is to be defined.

Currently it is not allowed to put this class into a namespace.

2.8. Exceptions

Exceptions can be raised using the `XCEPT_RAISE` macro. `XDAQ_LOG_AND_RAISE` no longer exists. The `XCEPT_RAISE` macro takes two parameters: an exception that inherits from the `xcept::Exception` top class and a text string.

To raise an exception in a SOAP callback do the following

```
XCEPT_RAISE (xoap::exception::Exception, "This is an exception");
```

To raise an exception in an I_2O callback to the following

```
XCEPT_RAISE (i2o::exception::Exception, "This is an exception");
```

To raise an exception in a CGI callback to the following

```
XCEPT_RAISE (xgi::exception::Exception, "This is an exception");
```

Since an application must raise the exception that has been specified in the signature, any other exceptions must be converted.

XDAQ V3.0 supports the stacking of exceptions. Use it.

Such, at the end of the exception chain, the history (comparable to the Java virtual machine stack trace) can be printed. To stack an exception use the `XCEPT_RETHROW` macro. It takes three parameters, the exception to be raised, a text string and the exception that has been caught, e.g.:

```
try
{
   do_something();
catch (xdaq::exception::Exception& e)
{
   XCEPT_RETHROW (xoap::exception::Exception, "do_something failed.", e);
}
```

The "stack" of exceptions can be printed to a string using:

```
std::string xcept::stdformat_exception_history(xcept::Exception& e)
```

The "stack" of exceptions can be formatted in HTML as a string using

```
std::string htmlformat_exception_history (xcept::Exception& e)
```

We recommend that every application defines its own exceptions that extend the `xcept::Exception` top class as follows:

Create a file *yourpackage*/include/*yourpackage*/exception/Exception.h with the following:

```
namespace yourpackage {
        namespace exception {
                class Exception: public xcept::Exception
                {
                        public:
                        Exception(  std::string name,
                                    std::string message,
                                    std::string module,
                                    int line,
                                    std::string function )
                        : xcept::Exception(name, message, module, line, function)
                        {}

                        Exception(  std::string name,
                                    std::string message,
                                    std::string module,
                                    int line,
                                    std::string function,
                                    xcept::Exception & e )
                        : xcept::Exception(name, message, module, line, function, e)
                        {}
                };
        }
}
```

You will then have an exception called *yourpackage*::exception::Exception. In the same subdirectory you may than create other specific exceptions that inherit from *yourpackage*::exception::Exception.

2.9. Application Identifiers

Applications do not require anymore an instance number and an I_2O target identifier (*tid*). If the I_2O protocol is used, an application *must* have an instance number. Every application is assigned a *tid* that is used for I_2O addressing in the XML configuration file by adding the following section:

```
<i2o:protocol xmlns:i2o="http://xdaq.web.cern.ch/xdaq/xsd/2004/I2OConfiguration-30">
   <i2o:target class="ClassName" instance="0" tid="23"/>
   <i2o:target class="ClassName" instance="1" tid="24"/>
   […]
</i2o:protocol>
```

We recommend adding this section right before the first `<Context>` declaration.
A *tid* can be any number between 1 and 2048.

To retrieve its instance, an application should do

```
if (getApplicationDescriptor()->hasInstanceNumber())
{
      getApplicationDescriptor()->getInstance();
}
```

To retrieve its own *tid* an application can do

```
i2o::utils::getAddressMap()->getTid(this->getApplicationDescriptor());
```

All addressing and sending of messages in V3.0 is based on application descriptors.

Application descriptors are objects that *represent* XDAQ applications. They are not the applications and do not contain the application objects.
The following code snippet gives an idea on how to retrieve the application descriptors of all remote XDAQ applications called *Example*:

```
try
{
   std::vector<xdaq::ApplicationDescriptor*> destination =
      getApplicationContext()->
         getApplicationGroup()->
            getApplicationDescriptors("Example");
} catch (xdaq::exception::Exception& e)
{
   XCEPT_RAISE (xcept::Exception, "Application not found");
}
```

The application group is a class that contains the descriptors of all applications that have been defined in an XML configuration file. It is retrieved using

```
getApplicationContext()->getApplicationGroup();
```

This class offers a rich variety of functions to retrieve application descriptors. See the file
`TriDAS/daq/xdaq/include/xdaq/ApplicationGroup.h`

2.10. Sending I_2O Messages

To send an I_2O message use

```
void xdaq::ApplicationContext::postFrame (
              toolbox::mem::Reference * ref,
             xdaq::ApplicationDescriptor* originator,
             xdaq::ApplicationDescriptor* destination)
                    throw (xdaq::exception::Exception)
```

The `frameSend` of V2.x no longer exists.
The `postFrame` takes a reference to a memory buffer and two application descriptors; the source (usually `this->getApplicationDescriptor()`) and the destination (to be retrieved as shown above).

Remember always to `try/catch` around this call, since an exception may be raised if the message cannot be queued for sending. Intercepting the failure of sending is more complicated and a specific user guide for this will be provided.

A complete example with buffer allocation for sending an I_2O message from within an application is outlined here:

```
try
{
   // Get a buffer of 1024 Bytes from a pool called "pool"
   toolbox::mem::Reference* ref =
      toolbox::mem::getMemoryPoolFactory()->getFrame("pool", 1024);

   // prepare frame
   PI2O_TOKEN_MESSAGE_FRAME frame = (PI2O_TOKEN_MESSAGE_FRAME) ref->getDataLocation();

   frame->PvtMessageFrame.StdMessageFrame.MsgFlags          = 0;
   frame->PvtMessageFrame.StdMessageFrame.VersionOffset     = 0;
   frame->PvtMessageFrame.StdMessageFrame.TargetAddress     =
      i2o::utils::getAddressMap()->getTid(destination);
   frame->PvtMessageFrame.StdMessageFrame.InitiatorAddress  =
      i2o::utils::getAddressMap()->getTid(this->getApplicationDescriptor());

   // Send a message of 500 Bytes
   frame->PvtMessageFrame.StdMessageFrame.MessageSize       = 500 >> 2;
   frame->PvtMessageFrame.StdMessageFrame.Function          = I2O_PRIVATE_MESSAGE;
   // The callback function code must be defined at the destination and
   // the callback function must be bound with i2o::bind to this code
   frame->PvtMessageFrame.XFunctionCode                     = I2O_CALLBACK_FUNCTION_CODE;
   frame->PvtMessageFrame.OrganizationID                    = XDAQ_ORGANIZATION_ID;

   ref->setDataSize(500); // message shall contain 500 Bytes

   getApplicationContext->postFrame(ref, getApplicationDescriptor, destination);
}
catch (xdaq::exception::Exception& e)
{
   // error handling
}
```

Do not forget to call `ref->setDataSize(size)` to tell the amount of bytes to be sent. This size must be smaller or equal to the amount of bytes allocated and it includes the I_2O message header size (it is therefore the total amount of Bytes sent).

2.11. Sending SOAP Messages

The call `SOAPMessage frameSend (SOAPMessage & message)` no longer exists.

To send a SOAP message use

```
xoap::MessageReference postSOAP (
              xoap::MessageReference message,
             xdaq::ApplicationDescriptor* destination,
             std::string network ="")
              throw (xdaq::exception::Exception)
```

The `postSOAP` takes a reference to a `xoap` SOAP message and a destination application descriptor. In addition it is possible to choose the network over which the SOAP call should be performed (in case multiple logical TCP/IP networks have been defined). Do not use this option in XDAQ V3.0. Detailed documentation on how to use this feature will be provided. Here is an example for sending a SOAP message.

```
xoap::MessageReference msg = xoap::createMessage();
xoap::SOAPPart soap = msg->getSOAPPart();
xoap::SOAPEnvelope envelope = soap.getEnvelope();
xoap::SOAPBody body = envelope.getBody();

// Always create SOAP with namespaces, here an example for
// <xc:Command xmlns:xc="urn:my-application-soap:1.0></xc:Command>
//
xoap::SOAPName command = envelope.createName("Command","xc", "urn:my-soap:1.0");
body.addBodyElement(command);

try
{
   // Set destination ApplicationDescriptor as explained before
   xoap::MessageReference reply = getApplicationContext()->postSOAP(msg, destination);
} catch (xdaq::exception::Exception& e)
{
   LOG4CPLUS_ERROR (getApplicationLogger(), xcept::stdformat_exception_history(e));
}
```

Usually it is only possible to send SOAP messages to applications that have been declared in an XML configuration file. Only then the application descriptor can be found using the `xdaq::ApplicationGroup` API.
It is, however, sometimes desirable to send SOAP messages to arbitrary destinations, e.g. Web servers. In this case it is possible to send a message creating a messengers on the fly as outlined in the following complete example.

```
// Create SOAP message
xoap::MessageReference msg = xoap::createMessage();

// Fill SOAP message
xoap::SOAPPart soap = msg->getSOAPPart();
xoap::SOAPEnvelope envelope = soap.getEnvelope();
xoap::SOAPBody body = envelope.getBody();

// Always create SOAP with namespaces, here an example for
// <xc:Command xmlns:xc="urn:my-application-soap:1.0"></xc:Command>
//
xoap::SOAPName command = envelope.createName("Command","xc", "urn:my-soap:1.0");
body.addBodyElement(command);

pt::PeerTransportAgent* pta = pt::getPeerTransportAgent();

try
{
   // Addresses that are created on the fly are reference counted and are therefore
   // deleted automatically after usage
   std::string url = "http://hostname:port";
   pt::Address::Reference destAddress = pta->createAddress(url+"/soap");
   pt::Address::Reference localAddress =
      pt::getPeerTransportAgent()->createAddress(
         getApplicationContext()->getContextDescriptor()->getURL()+"/soap");

   // Do not merge the following two lines into one
   pt::Messenger::Reference mr = pta->getMessenger(destAddress,localAddress);
   pt::SOAPMessenger& m = dynamic_cast<pt::SOAPMessenger&>(*mr);

   // fill the SOAPAction field
   // urn is used to route a SOAP message within a server to a destination
   // For xdaq applications, an urn always looks as follows:
   // urn:xdaq-application:id=<number>
   //
   msg->getMimeHeaders()->setHeader("SOAPAction", urn);

   // send message
   xoap::MessageReference r = m.send (msg);

   xoap::SOAPBody rb = r->getSOAPPart().getEnvelope().getBody();
   if (rb.hasFault()) {
      LOG4CPLUS_ERROR (getApplicationLogger(), rb.getFault().getFaultString());
   }
} catch (pt::exception::Exception& pte)
{
   XCEPT_RETHROW(xdaq::exception::Exception, "failed to send SOAP message", pte);
}
```

Xoap SOAP messages are reference counted and associated memory is released automatically when no other variable points anymore to the message.

Do never initialize a SOAP message with

```
Xoap::MessageReference msg = 0;
```

Message references are automatically initialized when `xoap::createMessage()` is called. Assigning another `xoap::MessageReference` to an existing `xoap::MessageReference` will

release the memory used by the message that is assigned.

Remember always to `try`/`catch` around the send calls, since an exception may be raised if the message cannot be sent.

SOAP operations are synchronous.

That means that every SOAP request has a SOAP reply. It does not mean that the activated remote function performs its operation synchronously.

2.12. Memory Management

V2.x supported only a single memory pool per process. To allocate frames and to recycle them the following API functions have been frequently used:

```
BufRef * frameAlloc(unsigned long size);
void frameFree(BufRef * ref );
```

These functions together with all other pool management functions have been refurbished. The class `BufRef` has been replaced with `toolbox::mem::Reference`.

V3.0 supports multiple memory pools that can be used concurrently.

An application that wishes to allocate buffers from a pool needs to either create a pool or attach to a pool that has been created by another application. There is no default memory pool.
To create an ordinary pool of virtual heap memory use

```
try
{
   // Create a pool that may expand up to 1 MByte (1024 * 1024 Bytes)
   toolbox::mem::CommittedHeapAllocator* a =
      new toolbox::mem::CommittedHeapAllocator(1024*1024);
   toolbox::net::URN urn("toolbox-mem-pool", "MyPool");
   toolbox::mem::Pool* pool = toolbox::mem::getMemoryPoolFactory()->createPool(urn, a);
} catch (toolbox::mem::exception::Exception& e)
{
   LOG4CPLUS_ERROR (getApplicationLogger(), xcept::stdformat_exception_history(e));
}
```

Allocating physical memory is possible with similar functions. This goes, however, beyond the scope of this porting guide. Detailed guidelines on this topic will be prepared.

This pool can be retrieved later on with the function

```
toolbox::mem::Pool* MemoryPoolFactory::findPool(toolbox::net::URN&urn)
                        throw(toolbox::mem::exception::MemoryPoolNotFound)
```

Pools that an application has created must be deleted by calling

```
void MemoryPoolFactory::destroyPool(toolbox::net::URN& urn)
                        throw (toolbox::mem::exception::MemoryPoolNotFound);
```

To allocate a memory buffer from a pool use

```
toolbox::mem::Reference* MemoryPoolFactory::getFrame (
                                toolbox::mem::Pool* pool,
                                unsigned long size)
                                throw (toolbox::mem::exception::Exception);
```

To give the memory back to the pool call `release` on the reference, e.g.

```
toolbox::net::URN urn("toolbox-mem-pool", "MyPool");
Toolbox::mem::Pool* pool = toolbox::mem::getMemoryPoolFactory()->findPool(urn);
toolbox::mem::Reference* ref = toolbox::mem::getMemoryPoolFactory()->getFrame(pool,1000);
ref->release();
```

Each pool can be instrumented with high and low watermark thresholds. There exist also APIs for checking threshold violations. This topic goes beyond the scope of this guide and separate documentation will be provided.

References can be chained like in V2.x. Instead of `BufRef::chainNext` use now

```
toolbox::mem::Reference::setNextReference ( Reference * ref )
```

The `BufRef::next()` call has been superseded by

```
toolbox::mem::Reference* toolbox::mem::Reference::getNextReference()
```

`BufRef->data()` has been superseded by

```
void* toolbox::mem::Reference::getDataLocation()
```

2.13. State Machines

V2.x featured a built-in state machine. Each application could make use of a predefined state-machine that was possible to change. To improve flexibility the state-machine classes have been improved and factorized out of the xdaq application container. As a result, state machines can be used if needed in XDAQ applications and in stand-alone applications and they can be defined with more ease. If you wish, you may also re-define the legacy state machine in your ported code.

Here, we only consider the synchronous state machine. More detailed information will be provided separately for the asynchronous state machine that can easily make any synchronous operation into a co-routine.

State machines can be synchronous and asynchronous in V3.0
There is no default hard-wired state machine.

To define a state machine, declare an instance variable

```
toolbox::fsm::FiniteStateMachine fsm_;
```

Define the state machine in the constructor of your application using the following calls

```
void toolbox::fsm::FiniteStateMachine::addState(State s,
             const std::string & name,
             OBJECT * obj,
             void (OBJECT::*stateChanged)(toolbox::fsm::FiniteStateMachine &))
             throw (toolbox::fsm::exception::Exception)

void toolbox::fsm::FiniteStateMachine::setInitialState(State s)
             throw (toolbox::fsm::exception::Exception)

void toolbox::fsm::FiniteStateMachine::addStateTransition(
             State from,
             State to,
             const std::string& input,
             OBJECT * obj,
             void (OBJECT::*func)(toolbox::Event::Reference) )
             throw (toolbox::fsm::exception::Exception)
```

The `State` type corresponds currently to a simple character. Such it is possible to define human readable states (e.g. '`H`' for "Halted", '`R`' for "Running") by maintaining high efficiency when it comes to frequently querying the state (for example in polling loops).

The functions `addState` and `addStateTransition` take an `OBJECT` and a `OBJECT::*func` method pointer parameter. `OBJECT` can be any application class that extends `toolbox::lang::Class` and the method is a pointer to a member function with the signature

```
void stateChanged (toolbox::fsm::FiniteStateMachine & fsm)
             throw (toolbox::fsm::exception::Exception)
```

for a state change callback and

```
void callbackName (toolbox::Event::Reference e)
throw (toolbox::fsm::exception::Exception)
```

for a state transition callback.

Each state machine has by default a state “Failed” (name ‘`F`’) and a state transition to the “Failed” state defined. It is possible to attach callback to these two actions.

For a full working example, please consult the source code of

```
TriDAS/daq/examples/include/SOAPStateMachine.h
```

This example also explains how to connect a defined state machine for incoming SOAP messages.

3. Makefiles

Makefiles are unchanged in V3.0. Due to the modified package directories it makes however sense to provide here a re-usable Makefile template that includes most of the used include and library directories.

The following statements prepare the automatic setting of the `XDAQ_ROOT` variable. The number of `../../..` directives depends on the location of your package. It specifies the way back to the `TriDAS` directory. For instance residing in `TriDAS/daq/mypackage/mysubpackage` will require three steps: `../../..`

```
XDAQ_BACK_TO_ROOT:=../../..
include $(XDAQ_BACK_TO_ROOT)/config/mfAutoconf.rules
include $(XDAQ_ROOT)/config/mfDefs.$(XDAQ_OS)
```

The following directives determine the package name and the source files to be compiled into the specified dynamic library

```
Project=daq
Package=mypackage/mysubpackage
```

```
Sources= File1.cc File2.cc
DynamicLibrary=MyXDAQApplication
StaticLibrary=
```

The following include directories are frequently used in XDAQ applications. All include directories of `TriDAS/daq/`*`mypackage/mysubpackage`* are already automatically added.

```
IncludeDirs = \
        $(XDAQ_ROOT)/daq/extern/xerces/$(XDAQ_OS)$(XDAQ_PLATFORM)/include \
        $(XDAQ_ROOT)/daq/extern/cgicc/$(XDAQ_OS)$(XDAQ_PLATFORM)/include \
        $(XDAQ_ROOT)/daq/extern/log4cplus/$(XDAQ_OS)$(XDAQ_PLATFORM)/include \
        $(XDAQ_ROOT)/daq/toolbox/include \
        $(XDAQ_ROOT)/daq/xdaq/include \
        $(XDAQ_ROOT)/daq/pt/include \
        $(XDAQ_ROOT)/daq/xgi/include \
        $(XDAQ_ROOT)/daq/i2o/include \
        $(XDAQ_ROOT)/daq/i2o/utils/include \
        $(XDAQ_ROOT)/daq/toolbox/include/$(XDAQ_OS) \
        $(XDAQ_ROOT)/daq/toolbox/include/solaris \
        $(XDAQ_ROOT)/daq/xoap/include \
        $(XDAQ_ROOT)/daq/xdata/include \
        $(XDAQ_ROOT)/daq/xcept/include \
        $(XDAQ_ROOT)/daq \
        $(XDAQ_ROOT)/daq/extern/i2o/include/ \
        $(XDAQ_ROOT)/daq/extern/i2o/include/i2o \
        $(XDAQ_ROOT)/daq/extern/i2o/include/i2o/shared
```

Put the following line at the end of the Makefile

```
include $(XDAQ_ROOT)/config/Makefile.rules
```

4. XML Configuration

The tag names of the XML configuration have been changed. Here is the list of modifications.

Old	New
<Host id="0" url="http://hostname:port">	<Context url="http://hostname:port">
<Application class="RoundTrip" targetAddr="#" instance="#" network="name">	<Application class="name" id="#" instance="#" network="local" resident="true/false"> *network* must be given (can be "local") *resident* is for pre-loaded applications only *instance* is optional
<urlApplication> *path* </urlApplication>	<Module> *path* <Module>
<Transport class="*name*" targetAddr="*#*" instance="*#*">	Simplified: also use <Application> tag now
<Address type="*name*" hostname="*hostname*" port="*#*" network="*name*"/>	<Endpoint protocol="*name*" service="*name*" hostname="*name or IP*" port="*#*" network="*name*"/> All <Endpoint> tags have now *protocol* and *service*. For binary messages over TCP/IP select protocol="tcp", service="i2o".
<Partition>	<Partition>
<DefaultParameters> <Parameter type="unsigned long"> </Parameter> </DefaultParameters>	<properties xmlns="urn:xdaq-application:Classname" xsi:type="soapenc:Struct"> <samples xsi:type="xsd:unsignedLong"> 1000 </samples> </properties> The variable names are now encoded in the tag names. Types of variables need to be provided using *xsd* data types.
I_2O addressing was previously merged into the <Application> tag	<i2o:protocol xmlns:i2o="http://xdaq.web.cern.ch/xdaq/xsd/2004/I2OConfiguration-30"> <i2o:target class="*name*" instance="#" tid="#"/> <i2o:target class="*name*" instance="#" tid="#"/> </i2o:protocol>

<table>
<tr><th>Old</th><th>New</th></tr>
<tr><td>2. Possibilities
<pre><Unicast
    class="name"
    instance="#"
    network="name"/>

<Unicast
    class="name"
    network="name"/></pre></td><td>3. Possibilities
<pre><Unicast
    class="name"
    instance="#"
    network="name"/>

<Unicast
    class="name"
    network="name"/>

<Unicast
    url="url"
    id="#"/></pre></td></tr>
<tr><td><pre><Definitions>
    <ClassDefid="#">
        name
    </ClassDef>
</Definitions></pre></td><td>Deprecated. Numeric class identifiers are not needed.</td></tr>
</table>

XML namespaces are now used throughout the whole configurations. Please have a look at some example configurations, for instance in `TriDAS/daq/benchmark/roundtrip/TCP.xml`.

An XDAQ application is unambiguously defined by
its local numeric identifier (id) within a context (url).

Therefore the URL for an XDAQ application uniquely identifies the application:

```
http://hostname:port/urn:xdaq-application:class=ClassName;id=#
```

Local identifiers for applications in the `<Application>` tag should be higher than 10. They do not need to be consecutively ordered. They do not need to be unique within the partition (The local identifier of an application can be the same in two different contexts).

An application whose `resident` attribute is set to `true` is assumed to be present at the start-up of the XDAQ process. It will therefore not be loaded into the process. Only its descriptor is created.

5. Peer Transports

The following peer transports are delivered with XDAQ V3.0

Name	Description
PeerTransportFifo	Implements the "local" network for process internal communication
PeerTransportHTTP	Implements the protocol http with the two services SOAP and CGI. The peer transport aims at supporting the HTTP 1.1 protocol, namely with the keep-alive option to improve the efficiency of POST requests. Both POST and GET requests are supported.
PeerTransportTCP	Supports the protocol TCP and the service I_2O for binary messaging. The send operations are non-blocking at application level. Currently a single thread is used at the receiver side for each interface card on which messages are expected (for each defined endpoint at the receiver side). For sending one thread is active.

6. xdaqWin

The Java development and testing client *xdaqWin* is deprecated. Although it is comptabile with XDAQ V3.0 it is not further developed and may become out of date quickly.

All development and test operations can be carried out using *HyperDAQ* facilities. For this purpose the user may connect with a Web browser to a running XDAQ process to perform the same operations that *xdaqWin* offered.

xdaqWin provided an embedded *Tcl* interpreter for automatizing some repetitive tasks. Scripting is now facilitated. Through SOAP and *HyperDAQ* functionality any scripting language that provides SOAP or HTTP requests can be used for scripting.
In addition scripting can be performed on any UNIX command line shell using the `wget` and `curl` commands.

The HyperDAQ facilities in V3.0 supports the following browsers

- Microsoft Internet Explorer V5.2 on Mac OS X
- Microsoft Internet Explorer V6.0 on Windows XP

Java 1.4.2 is required for using the Web applets.

Other browsers and browser platforms will soon be supported.

7. Getting Started with XDAQ

7.1. Installation

This section contains generic instructions for creating a basic XDAQ installation.

The Makefile attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to perform the build operations in each directory of the packages. The simplest way to install the package is:

1. Unpack the tarball (creates a directory named `TriDAS`)
   `tar -xvzf xdaqcore_G_17559_V3_0.tgz`

2. Set the XDAQ_ROOT environment variable to the directory of the unpacked TriDAS directory, e.g. for tcsh
   `setenv XDAQ_ROOT <path to current directory>/TriDAS`
   for bash:
   `export  XDAQ_ROOT=<path to current directory>/TriDAS`

3. `cd` to the `$XDAQ_ROOT/daq` directory containing the project's C++ Makefile and type
   `make Set=external`
   to build all external packages that are required for XDAQ.

4. Type `make Set=xdaq3` to compile the XDAQ package.
   Note: Ignore all warning output - these are placeholders for future developments and serve as a reminder that the functionalities are not in place in this version.

5. Check your system's Java installation by tpying `java –version`
   Make sure that the reported Java version number is 1.4.2_xx.
   Make sure that the java compiler can be invoked by typing `javac`

6. `cd` to the `$XDAQ_ROOT/java` directory containing the project's Java Makefile and type `make Set=xdaq3` to build all Java packages that are required for XDAQ.

Compilers and Systems for XDAQ Version 3.0:
The required C and C++ compiler is gcc version 3.2.3. The required Linux version is Scientific Linux CERN 3. This release has been compiled on kernel version 2.4.21-15.0.3.EL.cernsmp. You can check your kernel version of Linux by typing `uname -r`

7.2. Running the Process

To run the XDAQ process perform the following steps:

1. Set your `XDAQ_ROOT` environment variable to the directory of the `TriDAS` distribution
   In tcsh: `setenv XDAQ_ROOT ~/TriDAS`
   In bash: `export XDAQ_ROOT=~/TriDAS`
2. `cd` to `the $XDAQ_ROOT/daq/xdaq/bin/linux/x86` directory.
3. Run the XDAQ process by typing `./xdaq.sh`
   This script takes a number of parameters that it forwards to the `xdaq.exe` process.
   a. `-p` <*number*>
      port numbert on which the XDAQ process shall listen for HTTP and SOAP requests. By default the port used by the process is 40000.
   b. `-h` <*IP or name*>
      hostname or IP number that the XDAQ process shall use to listen for HTTP and SOAP requests
   c. `-l` <*label*>
      debug level to be used by default for log messages. Possible labels are `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`. The default log level is `INFO`.

When the process starts successfully you should see something like this:

```
[xdaq@LXCMDXYZ x86]$ ./xdaq.sh
Run xdaq with options
01-25-05 10:45:19,712 [3067717984] INFO http://lxcmdXYZ:40000 <xdaq> - xdaq Version: 3.0
01-25-05 10:45:19,713 [3067717984] INFO http://lxcmdXYZ:40000 <xdaq> - xdaq URL:
http://lxcmdXYZ:40000
Work loop name: fifo/PeerTransport
Work loop name: xrelay
Work loop name: urn:toolbox-task-workloop:http://lxcmdXYZ:40000
01-25-05 10:45:19,808 [3067717984] INFO http://lxcmdXYZ:40000 <xdaq> - XDAQ Ready.
```

8. Appendix: Special Notices

This publication is intended to help application developers port their applications to the XDAQ V3.0 software platform. The information in this publication is not intended as the specification of any programming interfaces that are provided by the XDAQ V3.0 software platform. See the XDAQ Web site at http://cern.ch/xdaq for more information about what publications are considered to be product documentation.
References in this publication to products, programs or services do not imply that we intend to make these available. Any reference to a product, program, or service is not intended to state or imply that only this product, program, or service may be used. Any functionally equivalent program may be used instead of the referenced product, program or service.
Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.
The information contained in this document has not been submitted to any formal test and is distributed AS IS. The use of this information or the implementation of any of these techniques is your responsibility and depends on your ability to evaluate and integrate them into your operational environment. While each item may have bee reviewed by us for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Users attempting to adapt these techniques to their own environments do so at their own risk.
Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.
Microsoft, Windows, Windows XP, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.
UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.
Intel and Itanium are trademarks of the Intel Corporation.
Other company, product, and service names may be trademarks or service marks of others.