Writing QSAS Plug-ins

A. R. Hare, S. D. Bale, A. J. Allen and M. Fränz
Astronomy Unit
Queen Mary & Westfield College
Mile End Road, London E1 4NS, U.K.
E-mail: csc_support@qmul.ac.uk
http://www.space-plasma.qmul.ac.uk/

Document Status Sheet			
1. Document Title: **QSAS Plug-ins**			
2. Document Reference Number: **CU-QMW-TN-0008**			
3. Issue	4. Revision	5. Date	6. Reason for Change
0	0	15 Jan 96	Internal doc ‘QSAS Data Object Attributes’.
1	0	09 Mar 96	Added QTPL notes and retitled document ‘Writing QSAS External Functions’.
1	1	24 Apr 97	QTPL update for QSAS 1.1; ‘external function’ renamed ‘plug-in’.
1	2	13 Nov 97	Reinstated SDB’s QSAS data-object attribute notes.
1	3	21 Nov 97	Explained use of dump to find object entry name.
1	4	10 Aug 99	Tidied up for Source release
1	5	17 Nov 99	Change to qtpl for .so entry name handling
1	6	19 Oct 2001	Revised for QDOS2 at qsas 1.6
1	7	25 Jun 2002	Revised for QSAS 1.65
1	8	2 Feb 2004	Revised for QSAS 2.0
1	9	12 May 2005	Addition of QuMultiJoin() utility and QSAS 2.1 release
1	10	7 Nov 2005	qdos and qsas class definitions available

Contents

1 About this document

This Technical Note provides sufficient information for a computer-programmer with some `C++` or `C` experience to write **plug-ins** for the QMW Science Analysis System (QSAS).
It should be used in conjunction with an up-to-date QSAS installation, including the examples of `C++` plug-in source-code and QSAS Template Language (`QTPL`) template-files supplied with QSAS by QMW. Programmers may wish to consult the doxygen generated class hierarchy for qdos and qsas available online in www.space-plasma.maths.qmul.ac.uk/QSAS/doc2.1/html/index.html or with the qsas source in QSAS_HOME/doc/doc2.1/html/index.html.

2 Related material

CU-QMW-MA-0006	QSAS User Manual
CU-QMW-MA-0009	QDOS Programmer's Manual
CU-QMW-MA-0014	QSAS HTML help pages
CU-QMW-MA-0015	QSAS and QDOS class definitions

3 Introduction

QSAS supports the dynamic loading and execution of separately-built **plug-ins**: these are software modules that accept QSAS data objects and constants as their input, and return new data objects or values as their output upon successful completion. This permits users to supplement QSAS's built-in facilities with their own specialised functions without having to rebuild the whole QSAS application. It also provides a mechanism for distributing self-contained additions to QSAS without the overhead of issuing a full new release.

QSAS plug-ins are loaded and run through a dynamic Plug-in Window which is described in the 'QSAS Help Pages', CU-QMW-MA-0014. The user first selects a template file that describes the plug-in in QSAS Template Language (`QTPL`). The template describes the arguments of the plug-in, how to find and load it, and optionally provides additional labelling, layout, and help information for the Plug-in Window.

After a plug-in template has been read successfully, the Plug-in Window is constructed with slots for the input and output arguments of the plug-in according to the template description. When the user has supplied valid entries for all mandatory input arguments, QSAS can be asked to load and run the plug-in. Arguments are communicated between QSAS and the plug-in by means of a `C++` list object containing pointers to QSAS data objects, and the plug-in return value indicates its completion status. The plug-in can write status messages and diagnostic information to a scrolling text window within its Window.
`C++` is used for the software interface between QSAS and plug-ins because the QSAS Data Object System (QDOS) is itself written in `C++`, and so `C++` methods are used to extract data from the input data objects and to create new data objects in which to return results. However it is possible to use code in other languages such as `C` and `FORTRAN` as the kernel of a plug-in by enclosing it in a `C++` 'wrapper': the wrapper takes care of the interfaces with QSAS and QDOS before and after calling the underlying non-`C++` function or subroutine with arguments of the types it requires. The details of how to bind together such mixed-programming-language components are system-dependent and outside the scope of this document.

4 QSAS Template Language (QTPL)

4.1 Example: XY2Polar.qtpl

Figure 1 shows the Plug-in Window prepared to run the plug-in `XY2Polar`. The arguments are set to their default values.
The principal features of QSAS Template Language `QTPL` may be seen by comparing the Window with the corresponding plug-in template, `XY2Polar.qtpl`, which is listed in figure 2 below. There follows a description of the principal features of `QTPL` and recommendations on their use.

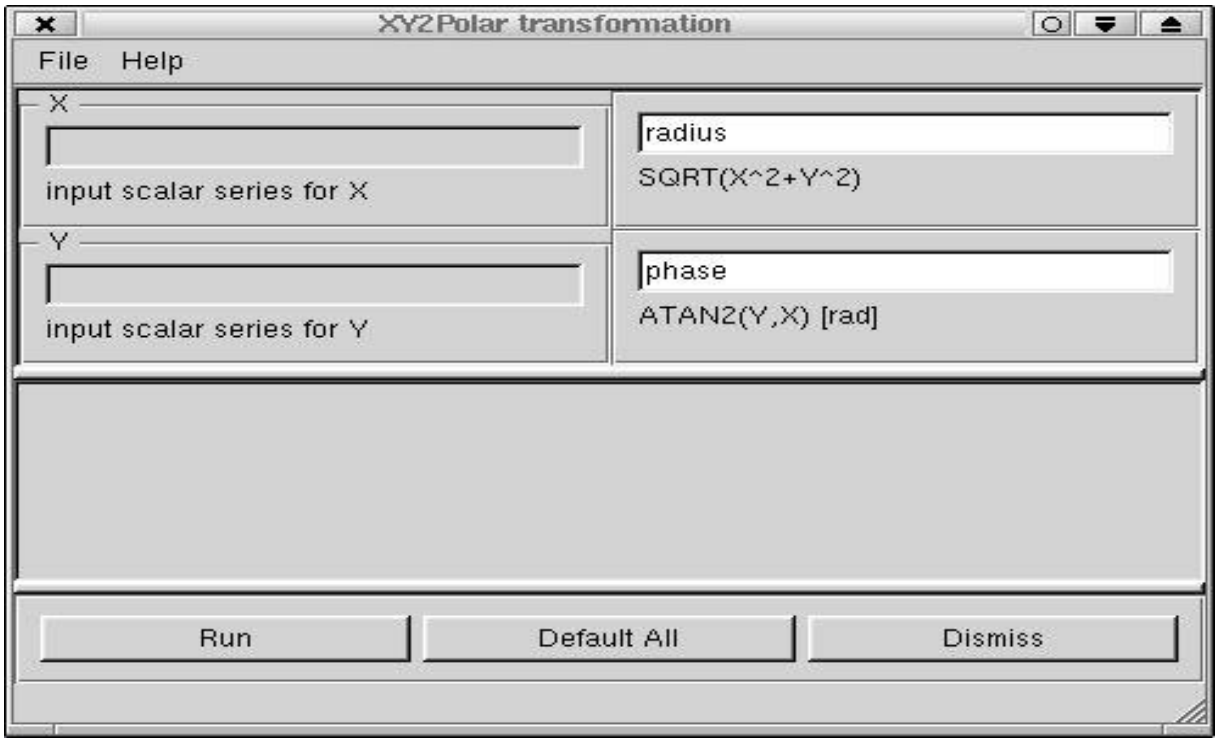

Figure 1: Plug-in Window after executing `XY2Polar.qtpl`.

```
                                          //  Plug-In Declaration:
EXTERNAL XY2Polar                         //  Type Name
 (INPUT  SCALAR_SERIES X_in,              //  Argument List
  INPUT  SCALAR_SERIES Y_in,
  OUTPUT SCALAR_SERIES radius,
  OUTPUT SCALAR_SERIES phase)

                                           //  Plug-In Body:
{XY2Polar                                  //  name variable
 {ENTRY_NAME  { "XYtoPolar" }               //  mandatory defs
  OBJECT_FILE { "XY2Polar.so" }

  PUBLIC_NAME { "XY2Polar transformation"}  //  optional attributes
  DESCRIPTION
  { "
   XY2Polar Transformation
   ======================

   This Plug In converts two scalar data series of same length
   into radius and phase.

   Input Arguments:

   Scalar series X.
   Scalar series Y.

   Output:

   Radius = SQRT(X^2+Y^2) Scalar series
   Phase  = ATAN2(Y,X) Scalar series
  " }

 }
                                           //  argument variables
 X_in { PUBLIC_NAME {"X"} DESCRIPTION {"input scalar series"} }
 Y_in { PUBLIC_NAME {"Y"} DESCRIPTION {"input scalar series"} }
 radius { DEFAULT_VALUE {"radius"} DESCRIPTION {"SQRT(X^2+Y^2)"} }
 phase {  DEFAULT_VALUE {"phase"} DESCRIPTION {"ATAN2(Y,X) [rad]"} }
}
```

Figure 2: Listing of template file `XY2Polar.qtpl`.

4.2 QTPL fundamentals: layout, punctuation, and names

`QTPL` **grammar** is loosely modelled on the `C` and `C++` languages. Blank lines and spaces may be used freely within template files to improve readability.

Multiple templates in a single file are not supported.

Comments are permitted anywhere in a template file. The `C++` comment-introducer `//` is used in `QTPL` to signify that whatever follows it until the end of the line is a comment.

Keywords (reserved words) in `QTPL` are written all-uppercase, *e.g.* `FLOAT`.

Declarations and definitions of plug-in functions and argument attributes are delimited by curly braces {}, which may be nested.

Strings of printable characters, which may include spaces and the newline-specifier `\n`, are enclosed in `ASCII` double-quotes *e.g.* `"line 1\nline 2"`. The maximum permitted string-length is 16,000 characters.

Variable names in `QTPL` are sequences of letters, digits, underscore and percent characters of arbitrary length; they must begin with a letter, and must be unique within each plug-in template.

4.3 Plug-in declaration

This has the form *plug-in-type plug-in-name argument-list*.

Plug-in-type keyword begins a plug-in declaration: it must be the first non-comment text in a template. The only type currently supported is `EXTERNAL`, as used in the example.

Plug-in name follows the type keyword, and must be a valid name for a `QTPL` variable. It is local to the template and independent of the name of the associated `C++` function, although it may be convenient for them to be similar. The example plug-in is named `XY2Polar`.

Argument list follows the plug-in name, and has the syntax

(*inout datatype varname, inout datatype varname, ...*).

The `QTPL` arguments must correspond in **sequence and datatype** to the QSAS data-object pointers expected by the `C++` plug-in.

There is a limit of 30 outputs, but input slots are unlimited, and if too many slots are specified to fit on the screen the plugin interface window becomes scrollable.

Body of the template follows the argument list, and is enclosed in curly braces {} .

4.4 Plug-in argument list

4.4.1 Argument syntax

Entry in argument list comprises three tokens *inout datatype varname*.

Input/output keyword *inout* may have the value `INPUT` or `OUTPUT`. `INPUT` arguments must not be modified by the plug-in; `OUTPUT` arguments are created and returned by it. Note that all inputs must appear before any outputs in the argument list.

Datatype keyword *datatype* indicates the type of data-object or value accepted or returned by that argument. Some datatypes include a variety of data-objects, and the QPLUGUI can perform implicit type-conversions as described below in §4.4.2, and summarised in Table 1.

Variable-name token *varname* is used to refer to this argument only within the body of the template, and is independent of any names used within the plug-in source-code or elsewhere. It must be a valid `QTPL` variable-name as defined above.

4.4.2 Argument datatype

The `QTPL` argument datatype keywords are summarised in Table 1. For `INPUT` arguments some testing of the datatype will be performed against the types understood by **QSAS** as listed in table 1. For `OUTPUT` arguments the datatype information is used only for labelling the QPLUGUI slots, so incorrect `OUTPUT` datatypes may not be detected by the software but could confuse the user.
Output objects are placed automatically on the working list, with the exception of `TIME_INTERVAL` objects that are placed on the Time Interval List.
In all instances the input object available to the plug-in is of type QdObject and must be narrowed to the type expected. Because of the wide range of objects that may be narrowed to the requested data type, the plug-in code is responsible for performing the required narrow and test, and to ensure that xrefs such as FRAME or TIME_TAGS are present when required.
The datatypes `ANY_DATA_OBJECT` and `ANY_TIME_SERIES` are provided so that a single plug-in may operate analagously on a variety of types of data-object: *e.g.* a time-averaging plug-in might have an `ANY_TIME_SERIES` input and an `ANY_DATA_OBJECT` output argument, as well as other inputs to specify the details of the time-averaging, enabling it to time-average many different kinds of time-series and to return a new data-object of an appropriate kind to contain the result in each case.
Inputs requiring either a sequence of time tags, `TIMETAGS` or a time interval, `TIME_INTERVAL`, will accept any object with associated TIME_TAGS xref as well as time sequence or interval objects.
In addition, if an input is specified as `ANY_DATA_OBJECT` then the associated input slot will accept a numeric value typed into the input slot. This is converted into a FLOAT type data object before being passed to the plug-in.
Single numeric quantities `FLOAT` and `INTEGER`, and strings `STRING` accept both QSAS data objects of the appropriate type and values typed directly into the input slot. The interface

datatype in file.qtpl	QSAS data-object type accepted in INPUT slot	NOTES
ANY_DATA_OBJECT	QdObject *float value*	All qdos objects inherit from QdObject or float numeric value
ANY_TIME_SERIES	QdRScalarSeq QdRMatrixSeq	with TIME_TAGS Xref
MATRIX_TIME_SERIES	QdRMatrixSeq	with TIME_TAGS Xref
VECTOR_TIME_SERIES	QdRMatrixSeq	with TIME_TAGS & Frame Xrefs
SCALAR_TIME_SERIES	QdRScalarSeq	with TIME_TAGS Xref
MATRIX_SERIES	QdRMatrixSeq	
VECTOR_SERIES	QdRMatrixSeq	with Frame Xref
SCALAR_SERIES	QdRScalarSeq	
TIMETAGS	QdTimeSeq QdRMatrixSeq QdRScalarSeq	 uses TIME_TAGS Xref uses TIME_TAGS Xref
TIME_INTERVAL	QdTimeInterval QdTimeSeq QdRMatrixSeq QdRScalarSeq	 uses start/end times uses TIME_TAGS Xref uses TIME_TAGS Xref
FLOAT	QdRScalar *float value*	 numeric value typed in INPUT
INTEGER	QdRScalar *float value*	 numeric value (cast to int)
MATRIX	QdRMatrix	
STRING	QdString *string literal*	 string typed in INPUT

Table 1: QTPL datatypes, valid QPLUGUI inputs, and QDOS data-object types.

between QSAS and the plug-in still uses a QSAS data-object of the appropriate type, but the Plug-in Window handles such quantities as text strings instead of requiring the user to first create a data-object. Thus the user can type a number directly into a FLOAT or INTEGER input slot, and text into a STRING input slot.

4.5 Template body: definitions

4.5.1 Definition syntax

Definitions in QTPL take the form
varname { *defname1* { *defvalue1* } *defname2* { *defvalue2* } ... }
for a list of at least one name-value pair. Here *varname* is the name of the plug-in or the argument-variable for which something is being defined, *defname* is the keyword, and *defvalue* is the value defined for it expressed inside an ASCII-double-quoted string. Strings may contain any printable characters, and additionally the newline specifier \n.

4.5.2 Plug-in mandatory definitions

The plug-in name-variable must have an OBJECT_FILE and an ENTRY_NAME defined as quoted text, in order that the plug-in may be located and loaded successfully.

OBJECT_FILE is the shareable library from which to load the plug-in: it may be defined as a complete path (*i.e.* beginning with '/') which will be taken as literal, or as a filename. If no path is specified a search will be undertaken using the current working directory '.', then the user specific environment variable USER_LIB and finally the **QSAS** provided plug-in library QSAS_HOME/lib. The latter method is recommended so that templates may be portable between different users and systems without the need to edit the OBJECT_FILE definition.

ENTRY_NAME is the function name of the plug-in wrapper. It is used to determine the entry point for the shareable library OBJECT_FILE which must be known in order to load it. Note: this supercedes the earlier plug-in syntax (QSAS version 1.3 and earlier) which required that the actual entry point string be specified here.

4.5.3 Plug-in optional definitions

The definitions below may optionally be set for the plug-in name variable to influence the appearance of the Plug-in Window and to provide on-line help when a plug-in is loaded.

PUBLIC_NAME**,** if defined, is used in place of the plug-in's internal QTPL variable name for the title of the window. The PUBLIC_NAME may contain characters that would not be permitted in a QTPL variable name. This is recommended as a way to display the version number after the name, as in the example.

DESCRIPTION of a plug-in provides the text to output when the user requests Brief description on the Plug-in Window Help menu, as shown figure 1.

4.5.4 Argument definitions (all optional)

Any QTPL variable may optionally have any or all of the following defined to influence its presentation in the Plug-in Window.

PUBLIC_NAME of a variable is used to label the interface slot for that argument instead of the variable name; this permits more flexibility in composing useful labels, as a PUBLIC_NAME may contain spaces, newlines, and other characters not permitted in variable names.

DESCRIPTION of non-function arguments provides additional labelling information for plug-in slots, as shown in figure 1.

DEFAULT_VALUE provides a default for the specified argument which will be set when a template is first loaded, and subsequently if the user requests it with the QPLUGUI Default All button. Note that DEFAULT_VALUE is specified as quoted text even for numeric values, as it is copied to the corresponding plug-in slot without processing, just as if the user had typed-in the characters.

5 C++ plug-in source code

5.1 Introduction

The following code fragments illustrate the extraction of pointers to QSAS data objects from the plug-in argument, the use of QDOS methods to read the input data-object contents, and the construction and return of new output data-objects. These constitute the essential C++ wrapper that must surround a plug-in; this wrapper may call functions coded in other languages to apply algorithms to the data extracted from the QDOS data-objects.
Note that dynamic loading implies that an executing plug-in is part of the QSAS application process: this permits shared memory and so avoids copying of large data-objects simply to read them, and allows the plug-in to call any QSAS built-in procedure, not just QDOS methods. But it also permits a plug-in bug to corrupt the contents of data-objects and other parts of QSAS, and crash the whole application through a fatal error. Great care should therefore be taken to trap plug-in errors, even if only to print a generic failure message and return error status to the QPLUGUI, as this is greatly preferable to crashing QSAS.

5.2 A minimal example

Below is a minimal C++ plug-in for the function XY2Polar described by the plug-in template in Fig.2, followed by a line-by-line explanation:

```
/*****************************************************************************
* PROJECT:      Cluster UK CDHF
* COMPONENT:    Plug In library
* MODULE: XY2Polar.cc
* LANGUAGE: C++
* FUNCTION:     QSAS XY2Polar Plug In
* PURPOSE:      Calculate radius and phase of two scalar series.
* ARGUMENTS:
*    IN:    Scalar series X.
*           Scalar series Y.
*   OUT:    Radius = SQRT(X^2+Y^2) Scalar series
*           Phase  = ATAN2(Y,X) Scalar series
* RETURN:   QplugReturnStatus
*****************************************************************************/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

#include "qplug_if.h"
#include "QdUtils.h"
#include "Qdos.h"
#include "qar.h"
```

```
#include "Xrefs.h"

using namespace QSAS;

extern "C" QplugReturnStatus  XYtoPolar( QplugArgList *);

QplugReturnStatus XY2Polar(QplugArgList *call_list )
{

// unpack the input data objects
QdObject_var X_in = (* call_list)[0];
QdObject_var Y_in = (* call_list)[1];

// narrow to a scalar sequence object
QdRScalarSeq_var xx = QdRScalarSeq_var::narrow(X_in);
if(xx.is_nil() ){
  QplugAppendTextDisplay("XY2Polar ERROR: X seq type not handled\n");
  return QPLUG_FAILURE;
}
QdRScalarSeq_var yy = QdRScalarSeq_var::narrow(Y_in);
if(yy.is_nil() ){
  QplugAppendTextDisplay("XY2Polar ERROR: Y seq type not handled\n");
  return QPLUG_FAILURE;
}

long Ndata = xx->sequence_size();

// check whether input conformal and Time series joined
qar_status =  QarTestConformal(X_in, Y_in);
if(qar_status == QAR_ERR)  return QPLUG_FAILURE;

 // confirm units are equivalent
 qar_status = QarAreUnitsSame(X_in, Y_in);
 if(qar_status == QAR_ERR) {
   QplugAppendTextDisplay (
     "Arithmetic>> Operation requires inputs in same units\n");
  return QPLUG_FAILURE;
 }
 else if(qar_status == QAR_WARN){
   QplugAppendTextDisplay (
     "Arithmetic>>  units unknown for at least one input\n");
   QplugAppendTextDisplay ("Proceed with caution!\n");
 }
```

```
//define output data objects
QdRScalarSeq_var out_rad_series;
QdRScalarSeq_var out_phas_series;

//calculate radius and phase

try{
  out_rad_series = QdRScalarSeq_var::narrow(sqrt(xx*xx + yy*yy));
  out_phas_series = QdRScalarSeq_var::narrow(atan2(xx,yy));
}
catch(Exception &e){
   e.print_msg();
   return QPLUG_FAILURE;
}

//set minimal attributes by copying from input objects
out_rad_series->copy_xrefs_from(X_in);
QuSetTxtAttr("Frame", "scalar>na", (QdObject_var) out_rad_series);
sprintf(str_out,"r(%s)", QuGetAttrText("FIELDNAM", X_in));
QuSetTxtAttr("LABLAXIS", str_out, (QdObject_var) out_rad_series);
QuSetTxtAttr("FIELDNAM", str_out, (QdObject_var) out_rad_series);

QuSetTxtAttr("SI_conversion", "1>rad", (QdObject_var) out_phas_series);
sprintf(str_out,"Phase(%s)", QuGetAttrText("FIELDNAM", X_in));
QuSetTxtAttr("LABLAXIS", str_out, (QdObject_var) out_phas_series);
QuSetTxtAttr("FIELDNAM", str_out, (QdObject_var) out_phas_series);
QuSetTxtAttr("Frame", "scalar>na", (QdObject_var) out_phas_series);
QuSetTxtAttr("UNITS", "rad", (QdObject_var) out_phas_series);


//get timetags from X or Y and add to output objects
QdTimeSeq_var tags = get_timetags(X_in);
if (!tags.is_nil()) {
   set_timetags(out_rad_series, tags);
   set_timetags(out_phas_series, tags);
}

//add output objects to  QSAS working list
call_list->push_back((QdObject_var) out_rad_series);
call_list->push_back((QdObject_var) out_phas_series);
return QPLUG_SUCCESS;
}
```

The example above includes the C++-standard libraries used by the plug-in:
`<stdio.h>, <math.h>, <string.h>, <stdlib.h>`. Then follows a minimal set of QSAS specific header files:
`"qplug_if.h", "QdUtils.h", "Qdos.h", "qar.h", "Xrefs.h"`. See below for functions available through QSAS libraries. `using namespace QSAS;` is necessary to use data object attributes (see section 7).
The 'extern' line is required by plugins for QSAS version 2.0 and later, it allows QSAS to locate the entry name in the .so library reliably...
extern "C" QplugReturnStatus XYtoPolar(QplugArgList *);
`QplugReturnStatus` can have the (self-explaining) values `QPLUG_SUCCESS`,`QPLUG_WARNING`, and `QPLUG_FAILURE`. The only argument of a plug-in is always a QSAS data object of type `QplugArgList`; it holds pointers to the data objects used by the function in the order they appear in the template file.
First we have to unpack the two input objects defined by the template (Fig.2):

```
QdObject_var X_in = (* call_list)[0];
QdObject_var Y_in = (* call_list)[1];
```

All objects are passed to the plug-in as the top level QdObject_var class (the _var means it is a safe pointer to this class). We have to convert the `QdObject_var` class var pointer to a derived class var pointer using the `narrow` method:

```
QdRScalarSeq_var xx = QdRScalarSeq_var::narrow(X_in);
if(xx.is_nil() ){
  QplugAppendTextDisplay("XY2Polar ERROR: X seq type not handled\n");
  return QPLUG_FAILURE;
}
QdRScalarSeq_var yy = QdRScalarSeq_var::narrow(Y_in);
if(yy.is_nil() ){
  QplugAppendTextDisplay("XY2Polar ERROR: Y seq type not handled\n");
  return QPLUG_FAILURE;
}
```

If the input object cannot be narrowed to the desired type the returned _var contains a NULL pointer, and can be tested using `is_nil()`.
The var pointers `xx` and `yy` can now be used for further operations. We may find the size of the sequence, using
`long Ndata = xx->sequence_size();`. This can be used to construct 'for' loops over the sequence, but is not necessary for the generic methods used here.
A single utility, QarTestConformal, is provided that checks that the sequences have the same number of elements unless one of them is a constant value (it is possible to perform most binary operations between a sequence and a constant value which is used for each entry in the sequence). If the input objects are time series, this utility also tests whether they are joined onto a common timeline.

```
 qar_status =  QarTestConformal(X_in, Y_in);
 if(qar_status == QAR_ERR)  return QPLUG_FAILURE;
```

A further check is performed since the function is only meaningful if the two input objects have the same units.

```
 qar_status = QarAreUnitsSame(X_in, Y_in);
 if(qar_status == QAR_ERR) {
   QplugAppendTextDisplay (
     "Arithmetic>>  Operation requires inputs in same units\n");
  return QPLUG_FAILURE;
 }
 else if(qar_status == QAR_WARN){
   QplugAppendTextDisplay (
     "Arithmetic>>  units unknown for at least one input\n");
   QplugAppendTextDisplay ("Proceed with caution!\n");
 }
```

This test uses the `SI_conversion` xref (attribute in cdf terminology) and reduces the unit to base SI components before comparison. It ensures that the units match as well as the scale factor (see detailed function description). Note that if the SI_conversion xref is not available for one input a warning is issued, but the operation is completed.
We could now create two output series of the same size as the input series, and fill the series with the radius and phase of the two input series using a loop over the individual elements of the input sequences,

```
QdRScalarSeq_var out_rad_series  = new QdRScalarSeq(Ndata);
QdRScalarSeq_var out_phas_series = new QdRScalarSeq(Ndata);
try{
  for (long i =0; i<Ndata; i++) {
    out_rad_series[i] =  sqrt(xx[i]*xx[i]+yy[i]*yy[i]);
    out_phas_series[i] = atan2(xx[i],yy[i]);
  }
}
catch(Exception &e){
   e.print_msg();
   return QPLUG_FAILURE;
}
```

but since the sqrt and atan2 methods are available for all qdos numeric types we may simply use,

```
QdRScalarSeq_var out_rad_series;
QdRScalarSeq_var out_phas_series;
try{
  out_rad_series = QdRScalarSeq_var::narrow(sqrt(xx*xx + yy*yy));
  out_phas_series = QdRScalarSeq_var::narrow(atan2(xx,yy));
}
```

```
catch(Exception &e){
   e.print_msg();
   return QPLUG_FAILURE;
}
```

Note that instances of the output objects are created outside of the try-catch construction to ensure they persist when the try block is exited.
The `try` and `catch` statements allow qdos to safely throw an exception if the requested operation is not permissable. For example, if the sequences xx and yy were of different types qdos would reject the operation and the returned error message would be printed to the standard output.
Also we should set the output object attributes (see section 7). In the case of the radial component most attributes are the same as for the input series and we can copy all xrefs from the input series:

```
 out_rad_series->copy_xrefs_from(X_in);
 QuSetTxtAttr("Frame", "scalar>na", (QdObject_var) out_rad_series);
```

We explicitly replaced the ”Frame” attribute with the scalar entry to be certain, since the input came from a vector component.
Note that certain standard attribute names are defined by QSAS, so we could have specified
QuSetTxtAttr(FRAME, ”scalar¿na”, (QdObject_var) out_rad_series);
in place of ”Frame”. This allows QSAS to take care of the case sensitivity of the ISTP/CSDS attribute names.
An alternative would have been to set just the minimal attributes directly, copying the SI_CONVERSION and UNITS explicitly from the input object.

```
 QuSetTxtAttr(SI_CONVERSION, QuGetAttrText("SI_conversion", X_in),
              (QdObject_var) out_rad_series);
 QuSetTxtAttr(UNITS, QuGetAttrText("UNITS", X_in),
              (QdObject_var) out_rad_series);
 QuSetTxtAttr(FRAME, "scalar>na", (QdObject_var) out_rad_series);
```

The attributes for the phase output must be set explicitly as they differ from those of the input object.

```
 QuSetTxtAttr(SI_CONVERSION, "1>rad", (QdObject_var) out_phas_series);
 QuSetTxtAttr(FRAME, "scalar>na", (QdObject_var) out_phas_series);
 QuSetTxtAttr(UNITS, "rad", (QdObject_var) out_phas_series);
```

If the input series are timeseries we would like to add the same timetags to the output series:

```
QdTimeSeq_var tags = get_timetags(X_in);
if (!tags.is_nil()) {
   set_timetags(out_rad_series, tags);
   set_timetags(out_phas_series, tags);
}
```

Note that the time tags are also Xrefs for the objects, so the earlier **copy_xrefs_from** command will have set them. Repeating this is safe, and is shown here for completeness.
Finally we are ready to put our objects on the QSAS working list by:

```
call_list->push_back((QdObject_var) out_rad_series);
call_list->push_back((QdObject_var) out_phas_series);
return QPLUG_SUCCESS;
```

The `push_back` operation puts the output objects on the argument list to be returned to QSAS. Output objects must be pushed back in the same order that they appear in the plug-in template (.qtpl) file. On successful return, the Plug-in interface adds the new data objects to the working list.

5.3 Building and Installing the Plug-In

To build a plug-in you need the `gcc` compiler version recommended in the QSAS Installation Notes (`$QSAS_HOME\doc\QSASInstallation.ps`). As an example, to build the plug-in **XY2Polar** one would follow these steps:

1. Save the plug-in code as `XY2Polar.cc` and the template as `XY2Polar.qtpl` in a local folder, say, `XY2Polar`.

2. Create a `Makefile` in this folder as shown in Fig.3. If you are including external code this should be included in the `OBJS` list and any necessary flags added to the `$(CC)` and `$(LD)` commands.

3. Copy the build script $QSAS_HOME/src/build to your local folder `XY2Polar`. This script may need editing to set library and other local paths if QSAS was installed from a binary rather than a source build.

4. Run the `build` script by typing `build` from the folder. On success the compiler will report the installation of the `.qtpl` and `.so` files.

The file selection widget launched by QSAS when a plug-in is requested can be used to navigate to the user specific location, $INSTALL_QTPL holding the new .qtpl template.
When the plug-in run button is pushed qsas will attempt to load the plug-in shareable object file. It will first look for the XY2Polar.so file in the current working directory, then the location specified by the environment variable **USER_LIB** and finally the standard libraries installed with qsas in $QSAS_HOME/lib. The sample makefile installs both the template and shareable object files in the directory pointed to by the user's environment variable $USER_LIB.
You may re-install your plug-in without starting a new QSAS session. The latest `.so` file will be executed whenever you push the `Run` button on the Plug-in Window, according to the search order specified above.
The above makefile builds and installs a user plug-in into the user's own local plug-in library space. To make the XY2Polar plug-in automatically availabe to all users it would be necessary merely to copy the XY2Polar.qtpl file to $QSAS_HOME/qtpl and the shareable object library, XY2Polar.so, to $QSAS_HOME/lib and ensure all users have read permission.

```
$(USER_LIB) =       /home/me/myplugins
INSTALL_LIB =       $(USER_LIB)
INSTALL_QTPL =      $(USER_LIB)
OBJS =              XY2Polar.o
.cc.o:
                    $(CPL) $(CPLFLAGS) $(PIC) -c $*.cc
.c.o:
                    $(CC) $(CFLAGS) $(PIC) -c $*.c
all:                clean install new

XY2Polar.so:        $(OBJS)
                    $(LD) $(PIC_LD) $(PIC) -o XY2Polar.so $(OBJS) $(LDLIBS)
clean:
                    'rm' -f *.o *.so *%

new:                XY2Polar.so
                    $(INSTALL) XY2Polar.so $(INSTALL_LIB)
                    $(INSTALL) XY2Polar.qtpl $(INSTALL_QTPL)
install:
                    $(INSTALL) XY2Polar.qtpl $(INSTALL_QTPL)
```

Figure 3: Makefile for XY2Polar plug-in.

6 Some Functions and Methods for QSAS dataobjects

In the following we list some commonly used functions and methods for QSAS data objects. The QSAS data object module QDOS contains many more (e.g. mathematical operators for all numerical objects.) Use the QDOS HTML documentation to find out about these. As well as intrinsic QDOS methods there are utility functions which are defined in `QsasUtils.h` [marked (A)] and `QdUtils.h` [marked (U)].

6.1 Plug-in Interface

- `QplugReturnStatus YourPlugin(QplugArgList *object_list)`,
  This is your plug-in function that is called by qsas when the run button is pushed. The return value must be one of the enumerated values,

```
typedef enum QplugReturnStatus_e {
  QPLUG_SUCCESS,
  QPLUG_WARNING,
  QPLUG_FAILURE
} QplugReturnStatus;
```

with the obvious meanings. The function argument is a pointer to a C++ vector of QdObject_var objects. When the function is called it contains the input objects in the order they appear in the template file, and before return the output objects should be created and pushed back on to the end of this vector.

The QplugArgList has the standard methods defined for vectors in C++; useful examples are...

- `size()`, which returns an int giving the number of input objects, e.g.

```
int n_inputs = object_list->size();
if (nlist < 2){
 QplugAppendTextDisplay( "Not enough arguments from GUI\n");
 return QPLUG_FAILURE;
}  //endif
```

- `[n]`, an object on the input list can be dereferenced directly according to its position on the list (in the order they appear in the .qtpl template), starting from zero. Note, object_list is itself a pointer to the QplugArgList list and must itself be dereferenced first, (* object_list), so that `(* object_list)[n]` is the `QdObject_var` pointer at position n on the input list.

```
QdObject_var input0 = (*object_list)[0];
```

- `QdObject_var::narrow()`

The objects passed to the plug-in are all QdObject_var pointers as this is the parent class for all qdos object types. It is necessary in most cases to narrow these objects to the actual type being handled.

The following example shows three ways to unpack the input.

```
QdObject_var input0 = (*object_list)[0];
QdTimeInterval_var ti = get_timeinterval(input0);

QdRScalar_var input1 = QdRScalar_var::narrow((*object_list)[1]);


QdObject_var input2 = (*object_list)[2];
QdRScalarSeq_var SS = QdRScalarSeq_var::narrow(input2);
QdRMatrixSeq_var MS = QdRMatrixSeq_var::narrow(input2);
if( !SS.is_nil() ){
    // operate on scalar input sequence
}
else if( !MS.is_nil() ){
```

```
    // operate on matrix (or vector) input sequence
}
else return QPLUG_FAILURE;
```

The first input is a time interval, and after creating a local pointer to the input object we use a utility call to create a time interval from it. This is a special case since it is possible to derive a time interval from a time series, a series of time tags or a time interval object, so a single utility call is provided (see below).

The second input object is a single scalar value and is narrowed on the fly while dereferencing the input list.

The third object may be either a scalar sequence or matrix (or vector) sequence, and after retrieving it from the input list we try to narrow to the possible input types. Note that the test is_nil() returns true if the input object is not of the requested type, so we negate it in the if() test. Tests on cartesian three vectors are described later. Note also that physical 3-vectors such as position or magnetic field vectors are stored as QdRMatrix objects, and utilities are provided to identify such objects and manipulate them (see below).

- `push_back(QdObject_var output)`,

```
QdRScalarSeq_var output;
object_list->push_back( (QdObject_var) output);
```

places the output object on the list for return to qsas to be placed on the working list. It must be cast to a QdObject_var since this is the parent class to which all qdos objects belong.

- `void QplugAppendTextDisplay(char *text)`, allows the plug-in to write messages to the plug-in message pane. e.g.

```
#include "qplug_if.h"
QplugAppendTextDisplay("Plug-in version 1.1, Nov 2001\n");
```

- `char * QplugNewSTR( char * oldStr)`, creates a copy of the input string oldStr. It allocates the space using malloc, and the calling function is responsible for freeing the returned char * pointer afterwards. It is safe against a NULL input string, in which case it returns a null terminated empty string. The returned string is always safe to free. e.g.

```
#include "qplug_if.h"
char * text = QplugNewSTR("Plug-in version 1.1, Nov 2001\n");
QplugAppendTextDisplay(text);
free (text);
```

- `char * GetSlotText( int n)`, gets a pointer to the text held in the n^{th} output slot. This pointer should not be freed. The output slots count from 0, and are numbered in the order they appear in the .qtpl file. e.g.

```
#include "qplug_if.h"
char * out_name = GetSlotText(0);
QuSetTxtAttr("FIELDNAM", out_name, out_ptr);
```

  The output slot should contain the (possibly user entered) name of the output object to be created.

- `void SetSlotText(int n, char * text)`, Sets the text held in the n^{th} output slot that will be used to name the output object on the Working List. This function frees the previously held string before replacing it with new string `text`. The output slots count from 0, and are numbered in the order they appear in the .qtpl file. e.g.

```
#include "qplug_if.h"
char * out_name = GetSlotText(0);
if (strcmp(out_name, "") == 0)
   SetSlotText(0, "DefaultName"); // ensure valid name
```

  The output slot contains the name of the output object to be created. This function allows the user set name to be overridden. Changes do NOT appear in the GUI window, but a warning is written to the plug-in text pane when a change is detected.

- `try{} catch(Exception &e){}`, should be used whenever using qdos intrinsic operations since qdos will throw an exception whenever an operation is invalid or cannot be completed for any reason. For example,

```
#include "qplug_if.h"
#include "QdUtils.h"
#include "Qdos.h"
using namespace QSAS;
try{
  QdObject_var RadSquared = QuGetObjByName("RadiusSquared");
  rad_series = QdRScalarSeq_var::narrow(sqrt(RadSquared_var));
  if ( rad_series.is_nil() ) {
    QplugAppendTextDisplay("rad_series not a scalar series\n");
    return QPLUG_FAILURE;
  }
}
catch(Exception &e){
   e.print_msg();
```

```
    return QPLUG_FAILURE;
}
```

In this example a scalar series on the Working List called “RadiusSquared” has been retrieved (see below) and the square root taken. If either the named object does not exist or the operation sqrt cannot be performed on it an exception is thrown and execution is passed to the catch statement which prints the QDOS error message and returns from the plug-in. Note that the narrow does not throw an exception, so it is still necesary to test is_nil() after the narrow. Several calls to qdos operations can be enclosed in a single try-catch block as any throw will go immediately to the enclosing catch without trying to complete other operations in the block.

6.2 General Methods

The following methods can be used for any data object class derived from QdObject. In all cases QSAS works with a pointer class QdObject_var and other pointer classes derived from it. These pointer classes keep track of all references to the actual data object, and when an object is no longer in scope for any pointer the object is deleted. An object placed on the working list is always referenced by a _var pointer held by qsas and will remain accessible until deleted from the working list. Utility functions (see below) are provided to access objects on the Working List by name. It is forbidden to delete a qdos object as it self destructs when no-longer in use. For this reason all reference to an object should be through a _var pointer which is always safe, and creation of a new object must be either using the clone() operation or ‘new’ on a _var pointer (see below).

- `virtual bool is_nil(void)`, returns true if the QdObject_var pointer does not point to a valid QdObject. This will be the case if an object of the requested type cannot be created. e.g.

  ```
  #include "Qdos.h"
  QdObject_var obj;
  if( obj.is_nil() )return QPLUG_FAILURE;
  ```

  Note that this is a method of the _var pointer class itself, and is thus accessed using ‘.’ rather than ‘->’, and is safe even if the QdObject pointed to is NULL.

- `bool is_empty(QdObject_var& obj)`, returns true if an object is a nil pointer, a zero element series, or a timeinterval of zero length. It is a utility call that can be used to trap results from operations that returned an essentially empty object, such as requesting a sub-interval that lies entirely within a data gap(A) e.g.

  ```
  #include "qplug_if.h"
  #include "Qdos.h"
  QdScalarSeq_var ts;
  ...
  if( is_empty(ts) ){
  ```

```
    QplugAppendTextDisplay("Operation gave an empty object\n");
    return QPLUG_WARNING;
 }
```

Empty objects are trapped by qsas and will not be placed on the working list.

- `new QdObject()`. The syntax for creating a new QdObject will depend on the derived class being created, and details are provided later. It is important that when a new object is created it is the subject of a _var pointer of the correct class to ensure memory management is not compromised. A simple example is a sequence of Ndata scalar objects,

```
#include "qplug_if.h"
#include "Qdos.h"
QdRScalarSeq_var out_rad_series  = new QdRScalarSeq(Ndata);
for(int i=0; i<Ndata; i++){
  out_rad_series[i] = (double) i*0.01;
  // alternatively, out_rad_series[i] = QdRScalar(i*0.01);
}
```

Note that this sequence is created as a sequence empty QdRScalar_var pointers, and each element of type QdScalar_var is created by the subsequent code using new with an argument of type double. A distinct QdScalar_var is created each iteration of the loop.

- `Factory::clone(QdObject_var& obj)` provides a generic method for producing a duplicate of any qdos object using the Factory class. A clone of an object contains a copy of the data, where possible, in the class requested. It always returns an object of the same type as the input object, but this may be assigned to a generic QdObject_var type. The xrefs (attributes) are not copied, and this must be done separately. e.g.

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"
QdObject_var ts_out = Factory::clone(ts_in);
if( ts_out.is_nil() ){
  QplugAppendTextDisplay("Unable to clone input sequence\n");
  return QPLUG_FAILURE;
}
 QdTimeSeq_var tt_out =
              (QdTimeSeq_var) Factory::clone(get_timetags(ts_in));
set_timetags(ts_out,tt_out)
```

Note that this example creates a new set of time tags for the new object, which would only normally be necessary if the time tags themselves were to be modified in any way.

If the new object must be of a specific type then clone may be used to explicitly return a new object of that type. The above example might have been...

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"

QdRMatrixSeq_var ts_out = Factory::clone( QdRMatrixSeq_var::narrow(ts_in));
if( ts_out.is_nil() ){
  QplugAppendTextDisplay("Input object not a Matrix sequence\n");
  return QPLUG_FAILURE;
}
```

- `QdObject_var::narrow(QdObject_var& obj)`, will return a _var pointer of the class requested if possible.

```
#include "qplug_if.h"
#include "Qdos.h"
QdRMatrixSeq_var ts_out = QdRMatrixSeq_var::narrow(ts_in);
if( ts_out.is_nil() ){
  QplugAppendTextDisplay("Input object not a Matrix sequence\n");
  return QPLUG_FAILURE;
}
```

The original object is pointed to by this _var pointer, and differs from clone in that it does not create a new data object. It will succeed only if the target is of the class requested. For example, a QdRMatrix_var cannot be narrowed to a QdRMatrixSeq_var, and vice versa.

- `QdObject_var QuGetObjByName(char * name)` returns a _var pointer to an object held on the working list as ‘name’.

```
#include "qplug_if.h"
#include "QdUtils.h"
#include "Qdos.h"
QdObject_var obj = QuGetObjByName("B_mag")
if ( obj.is_nil() ) return QPLUG_FAILURE;
```

The pointer is NULL if no object of the specified name can be found.(U)

- `virtual bool is_sequence(void)` returns true if the object is a sequence. This utility call is used to allow a plug-in to conditionally change between a sequence of objects or a single object.

```
#include "qplug_if.h"
#include "Qdos.h"
QdObject_var obj;
if(obj->is_sequence()){...}
else {...}
```

  It is of most use if the plug-in does not at this point need to know what type of objects the sequence contains; it is usually more efficient to narrow to the required sequence or object type directly and test `is_nil()`.

- `int sequence_size(void)` returns the number of elements in a sequence.

```
#include "qplug_if.h"
#include "Qdos.h"
QdObject_var obj;
int Ndata = obj->sequence_size();
```

  This method also works for any QdObject class and returns 1 in the case of a single object. The standard method size() will also work for the number of objects in a sequence, but fails for non-sequence type data objects.

- `bool QuConformal (QdObject_var do1, QdObject_var do2)`, returns true if the input series have the same number of entries and have the same time tags (if time series).

```
#include "qplug_if.h"
#include "QdUtils.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
QdRMatrixSeq_var out_ts;
if(QuConformal (ts1,ts2)){
  try{
    out_ts = new QdRMatrixSeq(ts1 + ts2);
  }
  catch(Exception &e){
     e.print_msg();
     return QPLUG_FAILURE;
  }
  QdTimeSeq_var tags = get_timetags(ts1);
```

```
  if (!tags.is_nil())  set_timetags(out_ts, tags);
}
```

This utility is only for use with sequence types and returns false if either input is not a sequence. Note that in this example we attach the original time tags to the new output series. If the original series is removed from the working list the time tags remain accessible so long as any object has a xref referencing them.(U)

- `bool QarTestConformal (QdObject_var do1, QdObject_var do2)`
  returns true if the input series have the same number of entries and have the same time tags (if time series), but also returns true if either object has a single entry to permit operations between a sequence and a constant. This test is advised before all arithmetic operations for time series data since the underlying qdos methods only test that the operation is well defined for the inputs. qdos methods do not test metadata, and in particular do not compare time lines.

```
#include "qplug_if.h"
#include "QdUtils.h"
#include "QsasUtils.h"
#include "Qdos.h"
#include "qar.h"
using namespace QSAS;
QdRMatrix_var Const = QdRMatrix::narrow(ts2);
QdRMatrixSeq_var out_ts;
if(QarTestConformal (ts1,Const)){
  try{
    out_ts = new QdRMatrixSeq(ts1 + Const);
  }
  catch(Exception &e){
     e.print_msg();
     return QPLUG_FAILURE;
  }
  QdTimeSeq_var tags = get_timetags(ts1);
  if (!tags.is_nil())  set_timetags(out_ts, tags);
}
```

This is a more tolerant test than QuConformal() and will permit operations between a single object and a sequence or between two sequences. If the objects are both time series, the time tags are tested to ensure either the same object is used as TIME_TAGS xref, or that tags are identical to near msec accuracy.

6.3 Time Series Handling

- `QdTimeSeq_var get_timetags(QdObject_var& obj)` returns a _var pointer to the `TIME_TAGS` xref for the object, e.g.

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
QdTimeSeq_var tags = get_timetags(ts1);
if (!tags.is_nil()) {
  ...
}
```

The returned pointer is always to an object of type QdTimeSeq_var.(A)

- `bool is_timeseries(QdObject_var& obj)` returns true if the object has a `TIME_TAGS` xref, e.g.

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
 if(is_timeseries(obj)){
   QdTimeSeq_var tags = get_timetags(obj);
    ...
 }
```

This is equivalent to `obj->xref_exists(TIME_TAGS)`.(A)

- `bool is_timetags(QdObject_var& obj)` returns true if the object is itself a sequence of time values, e.g.

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
 if(is_timetags(obj)){
    ...
 }
```

This is equivalent to a narrow to QdTimeSeq_var and test `!tt.is_nil()`. Note that a QdTimeSeq_var is a sequence of QdTime objects (time tags), whereas a time sequence is a sequence of data objects with a TIME_TAGS xref to a QdTimeSeq_var object containing the time tags.(A)

- `bool is_timeinterval(QdObject_var& obj)` returns true if the object is of type QdTimeInterval_var, e.g.

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
 if(is_timeinterval(obj)){
    ...
 }
```

This is equivalent to a narrow to QdTimeInterval_var and test `!ti.is_nil()`. Note that get_timeinterval below will return the interval associated with a time sequence and allows a more general interface.(A)

- `bool QuIsTSRegular(QdTimeSeq_var tt, double *spacing)`
  returns true if the time tags, tt, are evenly spaced, and without gaps, e.g.

```
#include "qplug_if.h"
#include "QdUtils.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
 double * spacing;
 QdTimeSeq_var tt = get_timetags(obj);
 if( QuIsTSRegular(tt, spacing) ){
    period_in_msec = spacing * 1000;
    ...
 }
```

The pointer `spacing`, returns the value of the interval between consecutive time tags in seconds.(U)

- `void set_timetags(QdObject_var obj, QdTimeSeq_var ts)`
  sets the `TIME_TAGS` xref for an object to point to the time tags, tt, e.g.

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
 QdTimeSeq_var tt = get_timetags(InObj);
 if( !tt.is_nil() )
    set_timetags(OutObj, tt);
```

Note this does not make a copy of the time tags object, and both the input and output data series will share the same time tags. The convention within QSAS is to make a copy of a set of time tags if the QdTimeSeq itself is to be changed in any way, otherwise derived objects share time tags to simplify joining and ensure arithmetic is valid. Objects that share time tags are 'joined' onto the same time line. Several objects on the working list may share a common set of time tags.(A)

- `QdTimeInterval_var get_timeinterval(QdObject_var& obj)`, returns a _var pointer to a QdTimeInterval object containing the start and end time-tag values for the object, e.g.

```
#include "qplug_if.h"
#include "QsasUtils.h"
#include "Qdos.h"
using namespace QSAS;
 QdObject_var input1 = (* object_list)[1];
 QdTimeInterval_var ti = get_timeinterval(input1);
```

Use of time interval objects is described below.(A)

6.4 Subsetting Data Series

Some specific utilities are provided for simple operations on sequences.

- `void get_subsequence(Sequence& seq,int start=0, int num=-1 )`
On return, `seq` contains a subsequence of `num` elements, starting from the index `start`. The default start index is 0, while if the number of elements to return is set to -1 (the default) the remainder of the sequence starting at index `start` is returned.

```
#include "Qdos.h"
QdRMatrixSeq_var ms = QdRMatrixSeq::narrow(ts_in);
QdRMatrixSeq_var sub_seq = new QdRMatrixSeq();
  // get first 10 elements of sequence...
ms->get_subsequence(sub_seq,0,10);
  // sub_seq contains 10 QdRMatrix objects starting from ms[0]
```

Note that in the above example we initialise the matrix sequence to be empty. A more efficient use of memory would be achieved if we had created it with the required number of elements and matrices of the correct dimensions, e.g.
`QdRMatrixSeq_var sub_seq = new QdRMatrixSeq(10,(ms[0]).dimdata());`
See Matrix Sequences below for details of this constructor.

- `QdTimeSeq_var get_timetags_inrange(QdTimeSeq_var& ts_in, QdTimeInterval_var& tint)`
Constructs and returns a subsequence of the time tags object that is delimited by the specified time interval.

```
#include "qplug_if.h"
#include "Qdos.h"
#include "QsasUtils.h"
using namespace QSAS;
 QdObject_var input1 = (* object_list)[1];
 QdObject_var input2 = (* object_list)[2];
 QdTimeInterval_var ti = get_timeinterval(input1);
 QdTimeSeq_var tt = get_timetags(input2);
 // subset time tags on interval set in input slot 1.
 QdTimeSeq_var tt_subset = get_timetags_inrange(tt, ti);
```

Note this utility returns the time-tags subset, it does not subset the data series associated (see sequence mask methods below).(A)

- `int get_num_timetags_inrange(QdTimeSeq_var& ts, QdTimeInterval_var& tint)`
  This works as for the method above, but returns the number of time tags that lie in the requested range rather than the tags themselves. `int n = get_num_timetags_inrange(ts,ti)` See also sequence masks below. (A)

6.5 Creating Sequence Masks

Subsetting data series is normally done through a sequence mask. The following methods show the construction of a sequence mask.

- `SequenceMask msk(int n_in_seq, bool accept);`
  This constructor creates a new sequence mask with n_in_seq entries, each of which takes the boolean value `accept`.

```
#include "Qdos.h"
QdRScalar_var xdo = QdRScalar::narrow((* object_list)[0]);
int ndata = xdo->sequence_size();
SequenceMask msk(ndata, true);
```

- `SequenceMask msk(const SequenceMask &mask);`
  This constructor creates a new sequence mask from the mask `mask`. It is used for creating a mask from the return value of the method `get_mask()`.

  The method `get_mask` operates on sequence types to construct a mask based on some criterion applicable to the particular sequence type. This generally takes the form
  `SequenceMask msk(QdSequence_var seq->get_mask(condition));`
  The constructor for a sequence mask here creates a mask `msk` using the get_mask() method for a QdSequence object. The possible values of argument `condition` depend on the sequence type used. Some example conditions are given below.

- `QdRScalarSeq::InRange(double x1, double x2)`
  returns true for all members of the scalar sequence which satisfy the bounding condition.

```
#include "Qdos.h"
double xmin = 0.0;
double xmax = 100.0;
QdRScalarSeq_var xdo = QdRScalarSeq::narrow(input1);
SequenceMask msk( xdo->get_mask(InRange(xmin, xmax)) );
```

- `QdTimeSeq::InRange`

```
#include "qplug_if.h"
#include "Qdos.h"
#include "QsasUtils.h"
using namespace QSAS;
QdObject_var input1 = (* object_list)[1];
QdObject_var input2 = (* object_list)[2];
QdTimeInterval_var interval = get_timeinterval(input1);
QdTimeSeq_var tt = get_timetags(input2);
// create mask based on timetags from input 2
//   that lie in interval set by input 1
SequenceMask msk(tt->get_mask(QdTimeSeq::InRange(interval->start(),
                                                 interval->end())));
```

- `EQ, NEQ, LE, GE, LT, GT`
  These operators allow masks to be constructed on a single condition, such as Greater Than (GT).

```
#include "qplug_if.h"
#include "Qdos.h"
#include "QsasUtils.h"
using namespace QSAS;
QdTime_var tc = new QdTime("02-Jan-2000 12:00:00.000");
QdTimeSeq_var tt_in = get_timetags((* object_list)[2]);
// create mask for timetags less than specified value
 SequenceMask msk(tt_in->get_mask( QdTimeSeq::LT( tc )));
```

- `(SequenceMask) msk[n] = (bool) accept;`
  Sets the mask value to the boolean value `accept`.

```
#include "Qdos.h"
QdRScalar_var xdo = QdRScalar::narrow((* object_list)[0]);
int ndata = xdo->sequence_size();
SequenceMask msk(ndata, true);
for(int i=0; i<ndata; i++)
{
  // create mask that keeps only positive values
  if( xdo[i] < 0 ) msk[i] = false;
}
```

6.6 Using Sequence Masks

The methods below show the use of sequence masks to subset both time lines and data sequences associated with them. The application of a mask to a data series has the effect that the sequence entry at index n is copied to the new sequence only if the mask value at index n is true.

- `void get_subsequence( Sequence& seq, SequenceMask &msk);`
  The returned sequence `seq` is of the same type as the parent sequence, but contains only the entries in the parent sequence for which the value of the mask is true. Note this function returns a Sequence object not a var pointer to it, and so the var pointer passed into this function must be dereferenced.

```
#include "qplug_if.h"
#include "Qdos.h"
using namespace QSAS;
QdTimeSeq_var tt_out = new QdTimeSeq();
QdObject_var input1 = (* object_list)[1];
QdObject_var input2 = (* object_list)[2];
QdTimeInterval_var interval = get_timeinterval(input1);
QdTimeSeq_var tt = get_timetags(input2);
SequenceMask msk(tt->get_mask(QdTimeSeq::InRange(interval->start(),
                                                 interval->end())));

// put timetags satisfying condition in tt_out
// Note tt_out is a var pointer, so we pass in *tt_out
tt_in->get_subsequence( *tt_out, msk);

QdRScalar_var ds_out = new QdRScalarSeq();

// put data objects satisfying condition in ds_out
input2->get_subsequence( *ds_out, msk);

set_timetags(ds_out, tt_out);
```

- `SequenceMask operator!`
  A sequence mask may be negated to invert its operation.
  `input2->get_subsequence( *ds_out, !msk);`
  will create a sub-sequence containing elements in input2 for which the mask was false.

- `SequenceMask operator &&, operator ||`
  The logical `and` and `or` operators are understood by sequence masks.
  `input2->get_subsequence( *ds_out, (msk1 && msk2) );`
  will create a sub-sequence containing elements in input2 for which the masks msk1 and msk2 were both true.

6.7 Time Object Handling

There are two new base classes defined for qdos, `Time` and `TimeInterval`. These are used for the QdTime and QdTimeInterval classes in the same way that double is the base class for QdRScalar. Any method defined for Time is also available to a QdTime object.
A QDOS Time object is a representation of a time. A Time can either be treated as a date, i.e. a specific time instant in some date system (eg: "14-Jun-1999 12:30:51.123") or as a period of time (eg: 4503.237 seconds). Knowledge of the internal represention of date/time used by Time is usually not required. The representation is the class Time::TRep, which is a double representing seconds (for time periods) or seconds from J2000.0 (Jan 1, 12:00GMT). You may apply basic mathematical operations (+,-,*,/) to time objects, e.g. `t1 += Seconds(90.0)` to add 90 seconds to `t1`. Also, you may use following QDOS functions:

- `Time Time( string& iso_time )` e.g.
  `Time t1("02-Jan-2000 12:00:00.000")` to create a time object from an ISO string.

- `Time Time(int year, int month, int day, double hour=0.0,`
  `double min=0.0, double sec=0.0 )` e.g.
  `Time t1(2000,1,2,12.0)` to create from a month/date representation. This is equivalent to `Time t1(2000,1,2,12.0, 0.0, 0.0)`, but the default argument values ensure that it is only necessary to specify down to the last non-zero argument value.

- `Days(int days),Hours(int hours),Minutes(int minutes),`
  `Seconds(double seconds), mSeconds(int msecs)`
  e.g. `t1 = t0(2000,1,1) + Days(1) + Hours(12) + Seconds(30.005)`
  to add days, hours, minutes, seconds (and milliseconds are added through the fractional part of seconds). Time units roll over, so that
  `t1 += Minutes(1) + Seconds(1.5)` is equivalent to `t1 += Seconds(61.5)`

- `Time& set_ydoy(int year, int doy, double hour=0.0, double min=0.0,`
  `double sec=0.0 ),`
  e.g. `t1.set_ydoy(2000,2,12)` to set t1 from year and day-of-year.

- `Time& set_ymd(int year, int month, int day, double hour=0.0,`
  `double min=0.0, double sec=0.0 )`
  e.g. `t1.set_ymd(2000,1,2,12)` to set t1 from year, month and day.

- `double day_of_year(void )`, e.g. `doy = t1.day_of_year()` will return 1.5.
- `void display()`, e.g. `t1.display()` will display current value of t1 on stdout.
- `string iso_srep(void )`, e.g. `tstr = t1.iso_rep()` will return ISO string.
- `void split_dhms(int& d, int& h, int& m, double& s )`,
  e.g. `t1.split_dhms(d,h,m,s)` will return a duration as days, hours, minutes, seconds.
- `void split_ydoy(int& year, int& doy, int& hour, int& min,`
  `double& sec )`
  e.g. `t1.split_ydoy(year,doy,hour)` will return year, doy, hour etc.
- `void split_ymd(int& year, int& month, int& day, int& hour,`
  `int& min, double& sec )`,
  e.g. `t1.split_ymd(year,month,day,hour)` will return year, month, day, hour of t1.

6.8 Time Interval Handling

In the examples below `ti` is an object of type TimeInterval. Since QdTimeInterval inherits from this class the methods described all work for QdTimeInterval_var pointers. For example

```
TimeInterval ti=new TimeInterval(Time(2000,1,2,12.0),
                                 Time(2000,1,2,14.0));
Time t = ti.end();
```

becomes the QdObject method

```
QdTimeInterval_var ti=new QdTimeInterval(QdTime(2000,1,2,12.0),
                                         QdTime(2000,1,2,14.0));
QdTime_var t = new QdTime(ti->end());
```

- `bool contains( Time& t )`, returns true if a time, t, is contained in the interval, e.g. `if(ti.contains(t))`
- `void display()`, will print a text representation of the interval to stdout, e.g. `ti.display()`
- `Time duration()`, returns a signed Time object holding the duration of ti, e.g.
  `Time t1 = ti.duration()`
- `Time duration_abs()`, returns a Time object holding the absolute duration of ti, e.g. `t1 = ti.duration_a`
- `bool empty()`, returns true when duration=0, e.g. `if(ti.empty()){...}`
- `Time end()`, returns a Time object holding the end time of ti, e.g. `t1=ti.end()`
- `Time end_abs()`, returns a Time object holding the later time of ti, e.g. `t1=ti.end_abs()`.
- `void force_sense(Sense option=TIME_SENSE_POSITIVE )`,
  forces the sense of ti to be either TIME_SENSE_POSITIVE or TIME_SENSE_NEGATIVE. The default is TIME_SENSE_POSITIVE, e.g. `ti.force_sense(TIME_SENSE_POSITIVE)` which is equivalent to `ti.force_sense()`

- `int get_sense()`, returns TIME_SENSE of ti, e.g. `ti.get_sense()` Possible return values are, NO_TIME_SENSE, TIME_SENSE_POSITIVE, TIME_SENSE_NEGATIVE.
- `TimeInterval int_intersect(TimeInterval& ti)`, returns a TimeInterval containing the intersection of t1 and t2, e.g. `ti=t1.int_intersect(t2)`
- `TimeInterval int_union(TimeInterval& ti)`, e.g. `ti=t1.int_union(t2)` returns union of t1 and t2.
- `void reverse()`, e.g. `t1.reverse()` reverses sense of t1.
- `void set( Time& st, Time& ed, Sense option=NO_TIME_SENSE)`, e.g. `ti.set(t1,t2)` sets ti from start time t1 and end time t2.
- `void set_duration(Time& d, SetOption option=SAME_START_TIME)`, e.g. `ti.set_duration(td,SAME_START_TIME)` sets end time of ti to `ti.start()+td`.
- `void set_duration( Time& t, Time& d, SetOption option=START_TIME)`, e.g. `ti.set_duration(t1,td,START_TIME)` sets ti from t1 and `t1+td`.
- `void set_end( Time& ed, SetOption option=SAME_START_TIME )`, e.g. `ti.set_end(t1)` sets end time of ti.
- `void set_start( Time& ed, SetOption option=SAME_SEND_TIME )`, e.g. `ti.set_start(t1)` sets start time of ti.
- `void shift_forward(double factor = 1.0 )`, e.g. `ti.shift_forward(0.5)` shifts ti forward by `0.5 * ti.duration()`, that is it shifts the time interval forward by half its own duration. The default is to shift the interval forwards by its own duration.
- `void shift_back(double factor = 1.0 )`, e.g. `ti.shift_back(2.0)` shifts ti backward by `2.0 * ti.duration()`. The default is to shift the interval by its own duration.
- `Time start()`, e.g. `t1=ti.start()` returns start time.
- `Time end()`, e.g. `t1=ti.end()` returns end time.

6.9 Matrix and Matrix Sequence handling

6.9.1 Dimensions

In order to handle the dimensions of a matrix in a coherent manner a class `Dim` is defined. Objects of this class are used when constructing matrices and enquiring about their rank and index sizes.

- `Dim(i1, i2, i3...i`*n*`)`
  This constructor creates a Dim object with rank n, and each dimension given by i1, i2 etc respectively. e.g.

```
#include "Qdos.h"
Dim dVec(3);
Dim dRotMatx(3,3);
// sets up the dimension for a 3-vector and a 3x3 rotn matrix
```

- `size())`
  this method returns the number of elements in a matrix with this dimensionality, i.e.
  i1 * i2 * ... * i*n*.
  e.g. `n_elems = dRotMatx.size();`
  In the above `n_elems` is 9 for a 3x3 Dim object.

- `rank())`
  this method returns the number of dimensions in a matrix with this dimensionality, i.e. *n. e.g.*
  `rank = dRotMatx.rank();`
  In the above `rank` *is 2 for a 3x3 Dim object.*

- `[]`
  The size of each index can be accessed by dereferencing using `[ ]`.

```
#include "Qdos.h"
Dim dRotMatx(3,3);
iSize = dRotMatx[0];
jSize = dRotMatx[1];
for(i=0; i<iSize; i++)
  printf("\n", i, j);
  for (j=0; j<jSize; j++)
    printf("%d,%d  ", i, j);
```

The above example prints
0,0 0,1 0,2
1,0 1,1 1,2
2,0 2,1 2,2

6.9.2 Matrix Class

Construction of a QdRMatrix object is achieved using a Dim object as first argument in the constructor. The elements in a QdRMatrix are all of type double.

- `QdRMatrix(const Dim &dim)`

```
#include "Qdos.h"
QdRMatrix_var Rmatx = new QdRMatrix (const Dim(3,3) );
```

This example creates a var pointer to a 3x3 matrix (of doubles).

- `QdRMatrix( Dim &dim, double value)`
creates a new QdRMatrix object with dimensions specified by the Dim object and all elements initialised to the value specified.

```
#include "Qdos.h"
QdRMatrix_var Rmatx = new QdRMatrix ( Dim(3,3), 0.0 );
```

This example creates a var pointer to a 3x3 matrix and initialises all elements to the value 0.

- `[i1]...[in]`
The elements of a matrix may be accessed directly by dereferencing using [ ].

```
#include "Qdos.h"
QdRMatrix_var mat = new QdRMatrix(Dim(10,5));
QdRMatrix_var dif = new QdRMatrix(Dim(10));
for(i=0; i<10; i++)
  (*dif)[i] = (*mat)[i][0] - (*mat)[i][4];
```

Note we dereference the QdRMatrix_var pointer to the QdRMatrix using (*matrix) before accessing the elements using [n]. This example finds the difference between the last and first colums in a 10 x 5 matrix.

- `dimdata()`
This method returns a dimension object of type Dim that describes the dimensionality of this QdRMatrix, e.g.

```
#include "Qdos.h"
QdRMatrix_var mat = QdRMatrix_var::narrow(obj);
// handle 2D matrix iSize x kSize
Dim mDim = mat->dimdata();

if ( !mat.is_nil() && mDim.rank() != 2 ) {
  // handle 2D array
}
```

```
else{
  QplugAppendTextDisplay("Input object not a 2D Matrix\n");
  return QPLUG_FAILURE;

}
```

- `dims()`
  This method returns an array with as many elements as the rank of the matrix, with the index size of each dimension as elements, e.g.

```
#include "Qdos.h"
QdRMatrix_var mat = QdRMatrix_var::narrow(obj);
// handle 2D matrix iSize x kSize
if ( !mat.is_nil() && (mat->dimdata()).rank() != 2 ) {
   iSize = (mat->dims())[0];
   kSize = (mat->dims())[1];

   double sumk[iSize];
   for(i=0; i<iSize; i++){
     sumk[i] = 0;
     for(k=0; k<kSize; k++)
       sumk[i] += mat[i][k];
  }
}
else{
  QplugAppendTextDisplay("Input object not a 2D Matrix\n");
  return QPLUG_FAILURE;

}
```

  The above example finds the sizes of the two indices of a 2D matrix and them sums over the second index.

- `rank()`
  It is possible to access the rank of a matrix directly, without first getting the corresponding Dim object. For example, the line
  `if ( !mat.is_nil() && (mat->dimdata()).rank() != 2 ){`
  above could more simply be written
  `if ( !mat.is_nil() && mat->rank() != 2 ){`

6.9.3 Matrix Sequences

A sequence of QdRMatrix objects is a QdRMatrixSeq, and this in turn may take a var pointer, QdRMatrixSeq_var.

- `QdRMatrixSeq()`
  This constructor returns an empty sequence of type QdRMatrixSeq_var.

```
#include "Qdos.h"
QdRMatrixSeq_var out_seq = new QdRMatrixSeq();
```

The above example creates a new empty matrix sequence.

- `QdRMatrixSeq(int size, const Dim & dim)`
  This constructor returns a QdRMatrixSeq_var with `size` entries each of which is a QdRMatrix with dimensions given by the Dim object dim.

```
#include "Qdos.h"
QdRMatrixSeq_var in_seq = QdRMatrixSeq_var::narrow(dobj);
int sz = in_seq->sequence_size();
QdRMatrixSeq_var out_seq = new QdRMatrixSeq(sz, Dim(3));
```

The above example creates a new sequence with the same number of entries as in_seq, but with dimension Dim(3).

- `[]`
  The QdRMatrix object at the i^{th} place in the sequence can be accessed using the simple dereference []. e.g.

```
#include "Qdos.h"
QdRMatrixSeq_var in_seq = QdRMatrixSeq_var::narrow(dobj);
int sz = in_seq->sequence_size();
QdRMatrixSeq_var out_seq = new QdRMatrixSeq();
for (int i=0; i<sz; i++)
  if( UserTest(in_seq[i]) ) out_seq->push_back(in_seq[i]);
```

The above example creates a new empty sequence and appends each QdRMatrix object from the input QdRMatrixSeq sequence that satisfies the user created function UserTest. The first [i] dereference on a QdRMatrixSeq_var accesses the QdRMatrix at index i, but subsequent sets of braces will extract the corresponding element inside the matrix, e.g.

```
#include "Qdos.h"
QdRMatrixSeq_var in_seq = QdRMatrixSeq_var::narrow(dobj);
if( (in_seq[r]).rank() == 2){
  double val_ij = in_seq[r][i][j];
}
```

The above example sets val_ij equal to the value of the i,j element in the 2D array for the r^{th} matrix in the sequence (which corresponds to the r^{th} time tag in the attached TIME_TAGS xref). Note that the rank of a matrix is a property of each matrix and not the sequence (which MAY be inhomogeneous), so we access the r^{th} element of the sequence, `in_seq[r]`, before asking for the rank().

Note that QdRMatrixSeq_var objects share the same generic methods as other QdObject_var objects and sequences, such as Factory::clone() and sequence_size(), etc.

6.10 Utilities for Handling Cartesian and Polar Vectors

- `bool Qu_is3Vector(QdObject_var &obj)`,
  returns true if the input object is a QdRMatrix_var or rank 1 and size 3, and has the FRAME attribute identifying it as a vector e.g. `if(Qu_is3Vector(b_xyz)){...}`
  The syntax for FRAME is described below.(U)

- `QdObject_var QuGetVectorXYZ(QdObject_var dov)`
  This returns a QdObject_var of the same underlying type as the input object, but as a cartesian xyz representation. It returns the input object if it is already in cartesian. It uses Qu_is3Vector() to ensure input is a vector with well defined representation and frame. Returns null var pointer (test with is_nil()) if a cartesian vector cannot be constructed. It is used to protect cartesian routines against polar input representations. e.g.

```
#include "Qdos.h"
#include "QdUtils.h"

QdObject_var vec_xyz = QuGetVectorXYZ((QdObject_var)vec_in);
QdRMatrixSeq_var vecSeq = QdRMatrixSeq_var::narrow(vec_xyz);
if ( vecSeq.is_nil() ) return QPLUG_FAILURE;
```

- `QdObject_var QuGetVectorComponent(QdObject_var dov, char *vcomp)`,
  e.g. `bx = QuGetVectorComponent(b_xyz,"x")` to extract a component from a vector series, where `vcomp one of {"x","y","z","r(mag)","theta(rad)","phi(rad)", "theta(deg)","phi(deg)","Lat(deg)","Lat(rad)"}`(U)

- `char * Qu_GetVectorRep(QdObject_var& obj)`, e.g.
  `rep = Qu_GetVectorRep(b_xyz)`
  returns the ‘xyz’, ‘rtp’ or ‘rlp’ string of the FRAME xref.(U)

- `int QarAreFramesSame(QdObject_var ptr1, QdObject_var ptr2)`,
  returns QAR_OK if the input objects are both vectors in the same frame and representation (FRAME attributes are identical), e.g.

```
#include "Qdos.h"
#include "qar.h"
if(QarAreFramesSame(b_xyz,c_xyz) == QAR_OK){ // OK go ahead...}
```

- See also the Qrtn plug-in code for frame rotations.
- See also the QDOS HTML documentation for cartesian to polar conversions.

6.11 Joining Time Series Data

A utility to join several time series objects onto a single timeline is provided in qdos, and a simple utility call to access it is provided which also ensures the metadata and ttimelines are set correctly for the joined objects.

- `#include qdutil.h;`

```
QuMultiJoin( QdObjectSeq_var        ToJoin_obj_seq,
             QdTimeSeq_var          tt_target,
             QdObjectSeq_var        Joined_obj_seq,
             vector<KVDataBase_var> options,
             int                    triage_cnt = -1);
```

If we have N qdos time series objects to join the arguments would be:

`ToJoin_obj_seq` is a sequence of N QdRScalarSeq_var and/or QdRMatrixSeq_var objects (multijoin can accept data series of different type and dimensionality), and holds the N input data series that are to be joined onto a new timeline.

`tt_target` is a single QdTimeSeq_var holding the new set of timetags that the all time series are to be joined onto.

`Joined_obj_seq` is a QdObjectSeq_var pointer that will return a sequence of length N holding the output data series - on return each entry will be a data series of the corresponding type and dimensionality as the input series at the same index. This sequence does not need to be populated before calling QuMultiJoin(), but if it does contain objects, they must match the dimensionality of the corresponding input objects, and these objects will be overwritten.

`options` is a vector of N KVDataBase_var objects holding the join options selected independently for each input object. See example for how to set these options. They cover the `join algorithm` (LINEAR, BOXCAR, and NEAREST_NEIGHBOUR), `gap_value` (the width in seconds to be used as the maximum input time tag separation above which the data is deemed to have a gap), `gap_handler` (the action to be taken if a gap is detected in the corresponding data

series. REMOVE_GAP, ZERO_FILL_GAP, FILL_GAP (with fillval), LINEAR_INTERP_GAP or NEAREST_NEIGHBOUR_GAP). This object also holds the new name to be used for the output object.

Note that `join_method`, `gap_value`, and `gap_handler` must all be set for each of the input data series, and need not be the same for each series.

`triage_cnt` determines how many of the input series must contain data at any time tag in order for that point to be included in the output sequences. Hence if any input series has gap_handler set to remove gaps, then `triage_cnt` is set to the number of input data series, in all other cases it is set to zero.

Note also that multijoin will work with any dependent variable, not just time, but the utility QuMultiJoin is specific to time series joining.

```
#include "Qdos.h"
#include "qdutil.h"
// join 4 data series onto time tags (5th arg in PlugIn list)
int nSeq = 4;
int triage_cnt = 0;
QdObjectSeq_var ToJoin_obj_seq;
QdObjectSeq_var Joined_obj_seq;
vector<KVDataBase_var> options;

QdTimeSeq_var qdtt = get_timetags((*arglist)[4]); // safe against null obj

if ( qdtt.is_nil() )
{
   QplugAppendTextDisplay ("invalid target time tags\n");
   return QPLUG_FAILURE;
}
for ( int i=0; i<nSeq; i++)
{
   QdObject_var obj = (*arglist)[i];

   // set options for object

   KVDataBase_var obj_options;

// fill value
QdRScalar_var    fillval_ptr;
if ( obj->xref_exists( "FILLVAL" ) )
{
QdObject_var fill_xref = obj->get_xref( "FILLVAL" );
fillval_ptr            = QdRScalar_var::narrow( fill_xref );
}
```

```
if ( fillval_ptr.is_nil() )
{
fillval_ptr = new QdRScalar( ( double ) 0.1e-30 );
}
obj_options->set( FILL_VALUE, fillval_ptr );

// gap tolerance in seconds
QdTime_var gap_ptr = new QdTime( Seconds( 6 ) );
obj_options->set( GAP_TOLERANCE, gap_ptr );

// boxcar width in seconds
QdTime_var boxcar_ptr = new QdTime( Seconds(60 ) );
obj_options->set( BOXCAR_WIDTH, boxcar_ptr );

// interpolation method
QdString_var interp_ptr = new QdString( LINEAR );
obj_options->set( INTERP_METHOD, interp_ptr );

// gap handling option
QdString_var gap_opt_ptr = new QdString( ZERO_FILL_GAP,  );
obj_options->set( GAP_METHOD, gap_opt_ptr );

QdString_var new_obj_name = new QdString( getSlotText(i) + "_joined");
obj_options->set( NEW_NAME, new_obj_name );

    ToJoin_obj_seq->push_back( obj );
    options.push_back( obj_options );

 }


 // Do actual join
 try
 {
    QuMultiJoin( ToJoin_obj_seq, qdtt, Joined_obj_seq, options );
 }
 catch( Exception &e )
 {
    QplugAppendTextDisplay (e.msg().c_str());
    return QPLUG_FAILURE;
 }


 for(int i=0;i<nSeq;i++)
```

```
{
  // return each object to qsas
  call_list->push_back((QdObject_var) Joined_obj_seq[i]);
}
```

Note that using the QuMultiJoin() utility it is no longer necessary to attatch the target time tags or input sequence Xrefs to the output objects as this is done by the utility.

7 Data Object Attributes

QSAS Data Objects are C++ class types used internally in QSAS to hold and manipulate data. When a Data Object is created, for example by ingestion of a CDF variable, it inherits certain metadata (attributes) of the data set. These are stored as cross-references ('xrefs') of the data object.
In the case of CDF or cef variable ingestion, the variable attributes should be available to the Data Object. The Global Attributes are available only via the browse interface and are not stored as xrefs to the data object. Data Objects created within QSAS are required to carry the 'minimal' set of attributes listed as essential in Tab.1. Details about these attributes are given in document DS-QMW-TN-0003. Attributes are accessed by name, so the user is free to invent and set new attributes. Users should avoid re-inventing the attributes named in Tab.1.
Attributes are only required for Data Objects that are visible to QSAS; in particular, objects that are to be added to the working list should meet the requirements below. Data Objects that exist only within functions or modules, as working space, aren't required to hold Attributes. The onus falls on the programmer of individual modules to ensure that any Data Object added to the working list has the required attributes.
Note that missing even "essential" attributes would not prove fatal, but can inhibit successful operation of other tools within the QSAS package (such as type checking in arithmetic and auto-labelling of plots). Attributes can be added to an object *after* it as been placed on the working list by double clicking the object and using the attribute browser/editor that pops up. This can be useful for "quick and dirty" plug-ins that are for use-once experiments, but is not generally recommended.
The time tags on a time series object are also attached to the object as a xref (TIME_TAGS), and may be accessed in the same way as any other xref. The names used for minimal qsas xrefs are defined in Xrefs.h and these defined strings are accessed through the names given in the second column in the table.

7.1 Getting and Setting Attributes

Attributes are held as QdObjects of the appropriate type, so that, for example VALIDMAX for a scalar data series would be accessed via a xref to a QdRScalar_var. Since the essential attributes are stored as xrefs in QdString format you will need only the following functions (QdUtils.h) to get and set these attributes:

Attribute	Xref	Type	Description	Example
Frame [*+]	FRAME	QdString	dim and frame	"scalar>na" or "vector>gse_xyz"
SI_conversion [*+]	SI_CONVERSION	QdString	conversion factor	"1.0e-9>T"
UNITS [+]	UNITS	QdString	unit string	"nT"
LABLAXIS [+]	LABLAXIS	QdString	plot axis label	"B"
FIELDNAM [+]	FIELDNAM	QdString	data identifier	"DC Magnetic Field"
SCALEMIN [+]	SCALEMIN	QdString	plot scaling	"0.0"
SCALEMAX [+]	SCALEMAX	QdString	plot scaling	"10.0"
VALIDMIN	VALIDMIN	QdString	plot scaling	"0.0"
VALIDMAX	VALIDMAX	QdString	plot scaling	"10.0"
FILLVAL	FILL_VALUE	QdString	missing data flag	"0.0"
SCALETYP	SCALETYP	QdString	plot scaling	"lin"

Table 1: Standard attributes for Qsas data objects. [*] means should be guaranteed by an analysis routine. [+] means should be set on input where ever possible, and should be set by an analysis routine, if possible.

- `char * QuGetAttrText(char * name, QdObject_var obj);`
  This returns a pointer to the string value of a text attribute called `name`. This string should not be deleted.

- `void  QuSetTxtAttr(char *name, char * value, QdObject_var ptr);`
  This function sets the content of the text xref called `name` to be `value` for the QSAS object pointed to by `ptr`. It makes a copy of the input string `value` which may then be safely destroyed.

```
#include "Qdos.h"
#include "QdUtils.h"
QuSetTxtAttr(FRAME, "scalar>na", (QdObject_var) out_series);
QuSetTxtAttr(SI_CONVERSION, QuGetAttrText("SI_conversion", X_in),
   (QdObject_var) out_series);
QuSetTxtAttr("Method", "Magnitude of XY components only",
             (QdObject_var) out_series);
```

The above example shows the creation of a new FRAME attribute with value “scalar_na”, the copying of the attribute SI_CONVERSION from the input object to the output series, and the creation of a new user defined attribute to be called `Method` to record useful information with the data. The attribute names shown in upper case are standard strings defined in Xrefs.h and listed in the table (1).

In addition you can use following xref-methods to manipulate and copy general xrefs of a QSAS data object, and these may be of any QdObject type. The object `Data_obj` is a var pointer of any type derived from QdObject_var.

```
Data_obj->change_xref(string& xref_name, QdObject_var xref_obj)
```

```
Data_obj->copy_xrefs_from(QdObject_var from_obj)
Data_obj->copy_xrefs_to(QdObject_var to_obj)
Data_obj->delete_xref(string& xref_name)
Data_obj->get_xref(string& xref_name)
Data_obj->list_xref_names(vector<string>& sl)
Data_obj->set_xref(string& xref_name, QdObject_var xref_obj)
bool Data_obj->xref_exists(string& xref_name)
```

It is sufficient to pass a char * string in place of the string literal into these methods for the xref_name input arguments.

Useful functions for calculating attributes In addition to the QDOS methods above there are some functions defined in `qar.h` which are useful for mathematical operations with QSAS data objects (qar.h):

- `int QarAreUnitsSame(QdObject_var ptr1, QdObject_var ptr2)`,
  e.g. `if(QarAreUnitsSame(obj1,obj2) == QAR_OK){...}` checks whether units attributes are identical.
- `char * QarUNITS_product(QdObject_var in1_ptr, QdObject_var in2_ptr)`,
  e.g. `newunits = QarUNITS_product(obj1,obj2)` to multiply units.
- `char * QarSIConv_product(QdObject_var in1_ptr, QdObject_var in2_ptr)`,
  e.g. `newsiconv = QarSIConv_product(obj1,obj2)` to multiply SI_conversion factors.
- `char * QarSIConv_inverse(QdObject_var in1_ptr)`,
  e.g. `newsiconv = QarSIConv_inverse(obj1)` to inverse SI_conversion factor.
- `int  QarTestConformalForVecProds(QdObject_var in1_ptr,`
  e.g. `if(QarTestConformalForVecProds(b_xyz,c_xyz) == QAR_OK){...}` checks whether series are 3-component with identical frames.
- `int QarAreFramesSame(QdObject_var ptr1, QdObject_var ptr2)`,
  e.g. `if(QarAreFramesSame(b_xyz,c_xyz) == QAR_OK){...}` checks whether series have same frame.
- `int QarTestJoined(QdObject_var do1,QdObject_var do2)`,
  e.g. `if(QarTestJoined(ss1,ss2) == QAR_OK){...}`
  to check whether two series have same size and - if timeseries - same timetags.