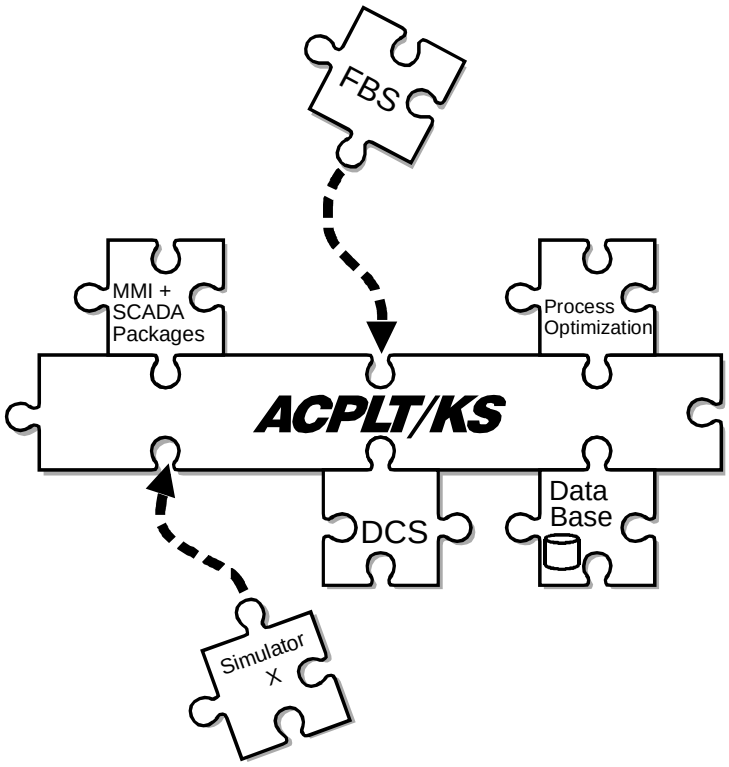
ACPLT/KS

Technology Paper No. 8:
C++ Communication Library Reference



Technology Paper #8 – Edition E97.12.1
Replaces: Edition E97.9.1
Documents: Release 1.01

Contact: H. Albrecht
Chair of Process Control Engineering, RWTH Aachen
D-52056 Aachen, Germany
Tel.: +49 241 80-7703
Fax.: +49 241 8888-238
eMail: ks@plt.rwth-aachen.de
WWW: http://www.plt.rwth-aachen.de/ks/english
(German pages: http://www.plt.rwth-aachen.de/ks)

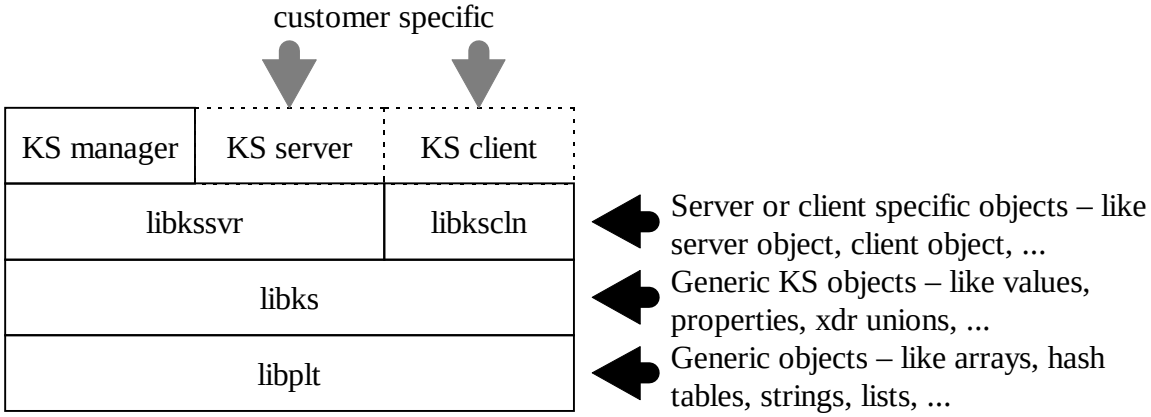
Contents

1 The C++ Communication Library

1.1 Introduction

This document describes the release 1.01 of the ACPLT/KS "C++ communication library. It helps you to integrate the open communication system ACPLT/KS into servers and applications that want to work with process data worldwide. As the name already suggests, the C++ communication library is written entirely in the C++ programming language. Due to the object-oriented structure of the library, the network communication and data representation is completely hidden inside the library, so the programmer does not need to worry about such issues as packing data into network order or the like.

The C++ communication library has been ported to various Unix platforms, like HP-UX, IRIX, Linux, and Solaris. In addition, the library has also been ported to Windows NT/95 and Open-VMS (according to an old wisdom, there are no portable applications – only applications which have been ported).



The ACPLT/KS C++ communication library consists of three levels (see figure above) of which the "libplt" library part represents the basic level implementing generic objects. The "libks" library part then represents the next level above the "libplt", and contains generic objects, which are used throughout the KS servers, clients, and managers. Above this level finally sit the server- and client-specific libraries "libkssvr" and "libkscln". The various KS (customer-specific) servers and clients are eventually build on top of these libraries.

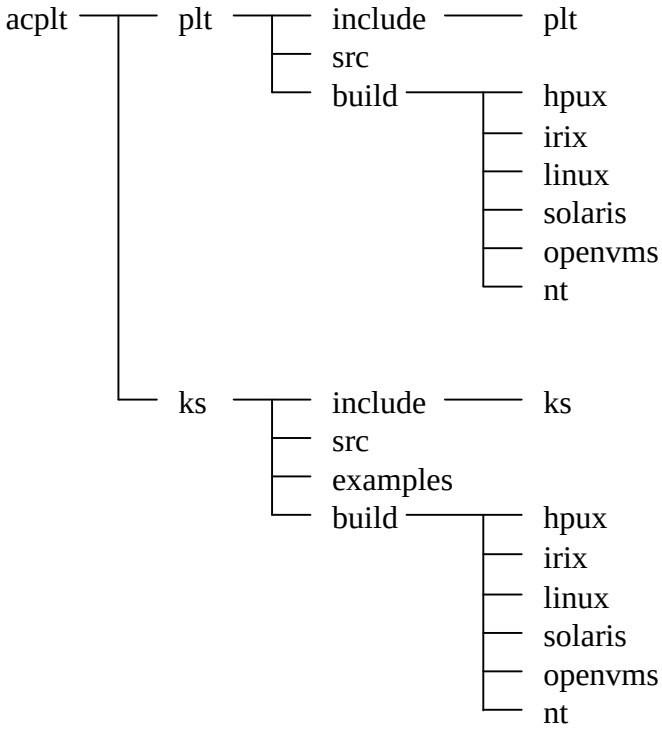
Please note that this reference only documents the public interfaces (and in some instances the protected interfaces too) of the various C++ classes contained in the C++ communication library. All interfaces not documented in this reference are considered private and must not be used by application writers. Such private interfaces are implementation specific and subject to change at any time without notice. Also, such classes which are not documented at all in this technology paper should be considered private and implementation dependent too. Using them explicitely in your own applications is solely at your own risk.

If you intend to write a KS server, then we suggest you to start reading in chapter 4 "libkssvr". This chapter explains the basic functionality provided by the communication library to server implementors. It describes how to add your customer-specific code in order to access a DCS, simulator, or other process data sources through the ACPLT/KS protocol. For this task, you'll work with various generic objects, for which you can look up detailed information (public interfaces, behaviour, use) in the previous chapters when you need to do.

If you plan to write a KS client, then start reading in chapter 5 "libkscln". Once again, detailed information about generic object, with which you'll work, can be looked up on demand in the previous chapters 2 "libplt" and 3 "libks".

1.2 Building the C++ Communication Library

The figure below shows the file tree of the C++ communication library. According to the general structur of the library, there are two main branches containing the more generic "plt" branch and the ACPLT/KS-specific "ks" branch. Within these two main branches, the header files always reside in the particular "include" subdirectories, whereas the sources can be found in the "src" subdirectories.



The directories below the "build" subdirectories contain the platform-specific makefiles. The object and library files for a particular platform are stored in these subdirectories too, so you can work with the same source tree on different operating systems at the same time, without the need to recompile the object files every time you switch the operating system.

Finally, the "examples" subdirectory of the "ks" branch contains a KS manager as well as an example of a KS server. Depending on the operating system you're using, you have to follow the appropriate steps listed below to compile the C++ communication library as well as the examples.

1.2.1 Unix

First, change to the base directory, where you put the ACPLT/KS sources files to, and do a full build of the communication library:

```
$ cd acplt
```

"*Os-name*" can (currently) be either "hpux", "irix", "linux", or "solaris".

```
$ make os-name.all
```

You should end up with the libraries "`libplt.a`", "`libks.a`", "`libkssvr.a`" and "`libkscln.a`". The examples are build in this step too.

The example applications are:

- `manager` – a Un*x-based KS manager, which catches various signals, so it can terminate graciously. In addition, it can be run as a daemon process by specifying "`--detach`" on the command line. Available command line options are printed to the console, if you start the KS manager process with the option "`--help`".
- `tmanager.exe` – a simple, operating system-independent KS manager.
- `tserver.exe` – a KS server, which implements various kinds of variables.
- `ttree.exe` – a KS client, which dumps a tree of the communication object hierarchy within a KS server. For all variables the client explores, it prints their values, too.

1.2.2 Windows NT/95

First, change to the base directory, where you put the ACPLT/KS sources files to, and do a full build of the communication library (the make process will automagically determine which development tools you're using):

```
$ cd acplt
$ make -f makefile.nt nt.all
```

You should end up with the libraries "`libplt.lib`", "`libks.lib`", "`libkssvr.lib`" and "`libkscln.lib`". The examples are build in this step too.

The example applications are:

- `ntksmanager.exe` – a NT-based KS manager, which is seamlessly integrated into NT's service mechanism.
- `w95ksmanager` – a 95-based KS manager, which can be run as a "service process" on Windows 95. In contrast to "normal" applications, service processes can be started before the user logs in and run until the system is shut down. They do not terminate when a user logs out and another user logs in.
- `tmanager.exe` – a simple, operating system-independent KS manager.
- `tserver.exe` – a KS server, which implements various kinds of variables.
- `ttree.exe` – a KS client, which dumps a tree of the communication object hierarchy within a KS server.
- `pmobile.exe` – a KS client, which shows how to work with packages when requesting large sets of variables at once.

1.2.3 OpenVMS

You'll need the OpenVMS port of the GNU make utility to build the C++ communication library. Make sure that you have at least version 3.75p1 of `gmake` as version 3.75 will **not** work. First, you need to define a symbol `GMAKE`, which executes the GNU `gmake` utility if you don't already have.

Second, define the symbol `ACPLT$REPOSITORY`, which must point to the directory for the C++ repository needed by DEC's CXX compiler. A good place for the repository directory is in your login directory and the directory is usually called `[CXX_REPOSITORY]`.

Change into the toplevel directory of the ACPLT/KS source tree and start the compilation. The examples are build in this step too.

```
$ gmake openvms.all
```

During the compilation, you can safely ignore warnings like "`%DELETE_W_SEARCHFAIL (target files are deleted before re-making them)`" and – when compiling with the debugging options turned on – "`%LINK_W_MULDEF DECC$__DL__XPV (operator delete) and DECC$__NW__XUI (operator new)`", these are intentionally redefined. As a rule of thumb, whenever you have *aborted* a compiler run, you should cleanup with "`gmake clean`", because partially created output files are not always deleted.

When you compile your own programs and link them with the C++ communication library for ACPLT/KS, **you must use the IEEE floating point format** (the UCX RPC library from DEC relies on this). Thus, make sure that you compile with the switch `"/float=ieee_float"` or ACPLT/KS participants on all other platforms will receive corrupted floating point values.

1.3 Running the Examples

After you've compiled the examples you're ready to fire up your first KS (test) server. First, start the KS manager which is in the file `tmanager.exe` (this name applies regardless of the OS platform you work with). Especially for Unix, there's a second KS manager available which is called `manager`. It is basically the same as the `tmanager.exe` but has addition code which catches some signals (most notabely `SIGINT`) and then graciously exits. On Windows NT make sure that you've first started the "portmap" service for the ONC/RPC mechanism. For more information about the installation of KS services, please refer to chapter 4.8, "NT Services".

Second, you can start the test KS server `tserver.exe`. It will register itself with the local KS manager and then accept incomming connections from KS clients. That's it! You're now ready to explore this test server. One way to do this is with the help of the "KS Shell" which is available as a small separate paket (KS-To-Tcl Interface package) from the web server of the Chair of Process Control. This KS shell works much like "ftp" clients, and can be used to walk through the tree of communication objects of a KS server, and for querying and setting variables. Another example application, `ttree.exe`, provides a quick view at the hierarchy of communication objects within a KS server. To look at the hierarchy within the KS manager, simply issue the command

```
ttree.exe localhost/MANAGER
```

at the command prompt. This `ttree` client shows how to iterate over the child communication objects of a domain and how to work with variable objects and their value objects. For each variable object it visits, it prints out the its value. In addition, you will find another KS client in the `examples` subdirectory: `pmobile.exe`. This client shows how to use packages when requesting large sets of variables at once from a ACPLT/KS server.

1.4 Template Mechanism

The template mechanism of the C++ language allows the programmer to provide a template for example for an array, which can be used to automatically create the code necessary to work with arrays of different data types. Part of this procedure is usually that you tell the compiler to create the code from the template for a particular data type. Unfortunately, there is currently no standard available how to achieve this automatically. Some compilers like Borland C++ provide an automatic mode which should relieve the programmer from this burden, but they currently all fail when working with at least mid-sized C++ projects.

As a work-around, the makefiles of the C++ communication library use the following mechanism on Unix platforms: the makefiles first try to link the applications although template instances (the code for particular data types) might be missing. If the linking process fails with unresolved symbols for template instances, then a wrapper program around the linker creates so-called

"template instantiation files". The names of these files start with the name of the first object module on the linker command line (without extension) and end with "`inst.h`" (for example, the template instance file for the application "`tserver.cpp`" has the name "`tserver_inst.h`"). This process repeats until there are no more unresolved template instances left. If you compile your KS server for the first time, this process may take around three to five round-trips. After this, you ordinarily can (re-) link your KS server in a single step, as the template instances are already there. To make this work, you must have the correct file dependencies. You can rebuild the dependency information using the command "`make depend`" (or "`gmake depend`") on unix platforms.

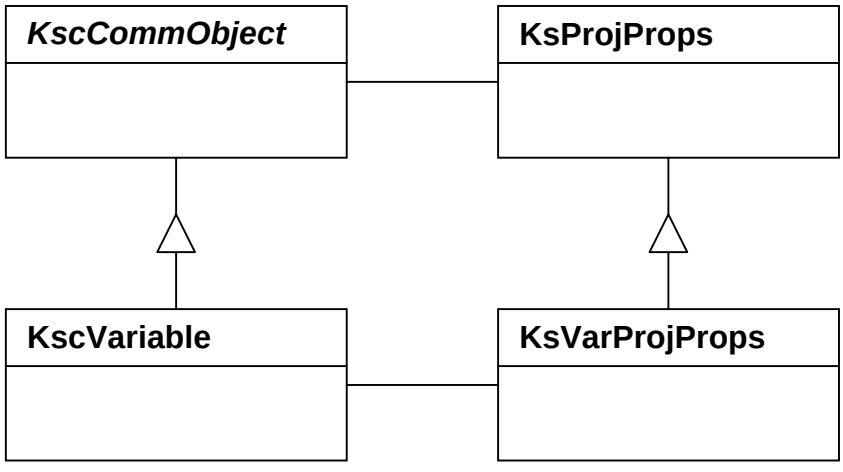
Currently, this wrapper mechanism around the linker is not supported for NT. The build directory therefore already contains all the template instance files necessary for compiling the example applications. If you write your own KS server, then you can simply copy the file with the name "`tserver_inst.h`" into another file. This template instance file contains the template instances of all template classes which are sufficient when writing a KS server. If you should delete the template instance files by accident, you can either pull of the compressed archive of the C++ communication library or copy them from the build directory of another platform.

So far, template instantiation is, er, quiet interesting – so we will not document it in every detail. Just take a look at the example sources like `tmanager.cpp`, `tserver.cpp` and `ttree.cpp`, and do it the same way. In the future, compiler vendors will have to support the C++ standard and this problem will go away. In the meantime watch the `#ifdef`'s in the example code which trigger different template instantiation dependend on the C+(+) compiler you use.

1.5 When your Compiler only knows of C+ instead of C++

The C++ Communication Library tries to support many different platforms using even more different C++ development environments. Unfortunately, these C++ environments differ in many ways. Some features can be emulated to some extend on platforms that don't support them (for example, runtime type information, or the bool data type). Other features introduce some not-so-elegant workarounds to be portable.

One particularly bad example is the overloading of return types. Specified by the C++ standard draft, this feature allows certain methods, which override base class methods to have a more specific return type than the overridden method (see the C++ standard draft for details). A very handy use of this C++ language feature can be seen with the way ACPLT/KS communication objects are associated with projected properties (see figure below).



Every ACPLT/KS communication object within the C++ Communication Library has associated so-called "projected properties", which are represented by objects of the class `KsProjProps` or a

subclass of. Like a basic communication object of the class `KscCommObject` can only have an object for its projected properties of class `KsProjProps`. A variable object of class `KscVariable` (which is derived from `KscCommObject`) can only have an object for its projected properties of class `KsVarProjProps` (which "is-a" `KsProjProps`).

You can ask any communication object for its projected properties using the `getProjProps()` member function. In case of a basic `KscCommObject` this returns a pointer to an object of class `KsProjProps`, and in case of a `KscVariable` this returns a pointer to an object of the derived class `KsVarProjProps`:

- class `KscCommObject`: `virtual KsProjProps *getProjProps();`
- class `KscVariable`: `virtual KsVarProjProps *getProjProps();`

Compilers that don't support this return type overloading must use the base class' return value in such cases, which causes substantial loss of information. To get around these restrictions, the C++ Communication Library uses a set of macros that expand into the correct behaviour on compilers supporting return type overloading and use the basic type on the others. In the following, the "correct" overloaded return type of the derived classes' method will be called the "conceptual return type".

This documentation will refer to the standard behaviour and thus avoids getting cluttered with obsolescent information. What you will see in declarations is therefore the conceptual return type.

Compiler Vendor/Platform	Supports Return Type Overloading according to C++ Standard draft?
FSF GNU g++ 2.7.x (Unix, (NT))	YEA!
Borland C++ 5.0 (Windows 95/NT)	YEA!
Microsoft Visual C+ 4.2/5.0 (Windows 95/NT)	NAY
DEC CXX 5.4 (OpenVMS)	NAY

The supporting macros are defined in "`plt/config.h`":

`PLT_RETTYPE_OVERLOADABLE`
: Defined to `1` for compilers supporting overloading of return types and to `0` for such other compilers who never have heard of the C++ language standard. This symbol is needed to provide the correct method declarations. By convention, within the C++ Communication Library, macros called *`classname`*`_THISTYPE` are used to denote the *most basic ancestor* of *`classname`*, which is the return type which has to be used with compilers without return type overloading. For compilers supporting overloading return types according to the C++ Standard draft, *`classname`*`_THISTYPE` is defined to *`classname`*. For example:

```
#define KsVarProjProps_THISTYPE KsVarProjProps
```

For compilers without proper support for return type overloading the macro is defined as:

```
#define KsVarProjProps_THISTYPE KsProjProps
```

Similarly, the C++ Communication Library uses a "`typedef` *`classname`* `THISTYPE;`" within class declarations to avoid cluttering the sources too much: this typedef defines the return type of the overloaded method according to the capabilities of the compiler.

`PLT_RETTYPE_CAST(type)`
: When you call a method with an overloaded return type, you will have to cast the return type to the conceptual return type if you have a broken compiler. If your compiler

understands return type overloading as described above, then you're fine and you do *not* have to cast.

To make the sources of the C++ Communication Library portable, this behaviour is encapsulated inside the macro `PLT_RETTYPE_CAST`, which you should use in your applications too for the sake of portability. Otherwise you'll be in deep trouble when porting forth and back from and to Visual C+ or DEC CXX Compilers.

Now, let us look at an example. If you ask a list object for an iterator, it does return a specialised iterator, which has all the methods a basic iterator has, and some additional ones. If you want to take advantage of these additional iterator methods and you work with some of the not-so-up-to-date C++ compilers, then you must cast the pointer returned to the appropriate iterator pointer type. This is hidden inside the `PLT_RETTYPE_CAST` macro:

```
PltListIterator<int> *pit =
  PLT_RETTYPE_CAST((PltListIterator<int> *)) list.newIterator();
```

Note: you **must** use double parentheses here!

And now the case if you don't need to consider backwards compatibility with incapable compilers – slightly shorter and more comprehensive:

```
PltListIterator<int> *pit = list.newIterator();
```

See also section 6.1 for a list of preprocessor symbols defined or used by the C++ communication library.

2 libplt

The "libplt" library part represents the basic level of the ACPLT/KS C++ communication library. It contains classes for commonly used objects throughout the whole communication library, like arrays, lists, strings, and handles. All classes defined within the "libplt" share the same name prefix "Plt", so you can easily tell from a classes' name to which part of the C++ communication library it belongs to.

2.1 Container

2.1.1 Class PltContainer

Containers play an important part within the C++ communication library. There are various kinds of containers available like arrays, lists, and hash tables. This abstract container class provides a common interface to these kinds of containers. Following is the public interface of the abstract container base class `PltContainer`:

```
template<class T>
class PltContainer {
public:
    virtual bool isEmpty() const;
    virtual size_t size() const = 0;
    virtual PltIterator<T> * newIterator() const = 0;
};
```

`virtual bool isEmpty() const`
: Returns `true`, if the container is empty.

`virtual size_t size() const`
: Returns the number of elements the container currently contains.

`virtual PltIterator<T> * newIterator() const`
: Returns a new iterator object, which is suitable for iterating over the elements of the container. The caller must eventually delete the iterator when she/he is done with it, or a memory leak occours.

2.1.2 Class PltHandleContainer

The so-called "handle containers" are very similiar to the other containers derived of the abstract base class `PltContainer`, but they manage their elements using handles. This way the lifetime of the elements within a handle container is controlled by the element's handle.

```
template<class T>
class PltHandleContainer {
public:
    virtual bool isEmpty() const;
    virtual size_t size() const;
    virtual PltHandleIterator<T> * newIterator() const = 0;
};
```

`virtual bool isEmpty() const`
: Returns `true`, if the container is empty.

`virtual size_t size() const`
: Returns the number of elements the container currently contains.

`virtual PltHandleIterator<T> * newIterator() const`
: Returns a new iterator object, which is suitable for iterating over the elements of the container. The caller must eventually delete the iterator when she/he is done with it, or a memory leak occours.

2.1.3 Class PltArrayed

The abstract class `PltArrayed` is the base for such containers, whose elements can be individually addressed like an array. Following is the public interface of the class `PltArrayed`:

```
template <class T>
class PltArrayed : public PltContainer<T> {
public:
    virtual size_t size() const = 0;
    virtual const T & operator[] (size_t idx) const = 0;
    virtual PltArrayIterator<T> * newIterator() const;
};
```

`virtual const T & operator[] (size_t idx) const`
: Provides access to an individual element of the container by its index `idx`.

`virtual size_t size() const`
: Returns the number of elements the container currently contains.

`virtual PltArrayIterator<T> * newIterator() const`
: Returns a new iterator object, which is suitable for iterating over the elements of the container. The caller must eventually delete the iterator when she/he is done with it, or a memory leak occours.

2.2 Iterators

2.2.1 Usage

Iterators are used to walk through (iterate over) the elements stored in a container. All iterators at least provide access to the current element within a container the iterator points to, increment operations to move on to the next element, and a check whether all elements have been visited. Finally, you can reset an iterator, so it points to the first element of a container again. You can therefore iterate over the elements within a container more than once using the same iterator.

Some iterators also provide decrement operations which move the iterator to the previous element. Whether an iterator supplies decrement operations depend on the type of container the iterator iterates over.

The following example shall enlighten the use of iterators. You can either ask a container for an iterator...

```
PltIterator<int> *pit = intlist.newIterator();
```

...or you can create one yourself:

```
PltListIterator<int> it(intlist);
```

In our little example the `intlist` above is a list object of class `PltList<int>`. It just stores integers. Now, we iterate over all elements within the list, starting with the first one and printing each, as we visit them.

```
while ( it ) {
    cout << *it << " " << **pit << endl;
    ++it;
}
```

Eventually, we must destroy any dynamically allocated iterator to save spare memory resources.

```
delete pit;
```

2.2.2 Class PltIterator

Following is the public interface of the abstract class `PltIterator`, which forms the root for most of the iterators provided by the libplt.

```
template<class T>
class PltIterator {
public:
    virtual operator const void * () const = 0;
    virtual const T & operator * () const;
    virtual const T * operator -> () const;
    virtual PltIterator & operator ++ () = 0;
    virtual void toStart() = 0;
};
```

`virtual operator const void * () const`
: Cast operator that allows to check whether the iterator has iterated over all elements. In case all elements have been iterated over, the cast operator returns `0` (the null pointer). See the "Usage" section of this chapter for an example.

`virtual const T & operator * () const`
: Provides access to the element in a container the iterator currently points to.

`virtual const T * operator -> () const`
: Provides access to the element in a container the iterator currently points to.

`virtual PltIterator & operator ++ ()`
Prefix operator that moves the iterator to the next element within a container.

`virtual void toStart()`
Resets the iterator, so it points to the first element within a container.

2.2.3 Class PltBidirIterator

The abstract class `PltBidirIterator` is derived from the `PltIterator` class and can iterate in two directions over the elements of an container. The iteration directions are commonly refered to as "forward" and "backwards".

```
template <class T>
class PltBidirIterator : public PltIterator<T>
{
public:
    virtual PltBidirIterator & operator -- () = 0;
    virtual void toEnd() = 0;
};
```

`virtual PltBidirIterator & operator -- ()`
Prefix operator that moves the iterator backwards to the previous element within a container.

`virtual void toEnd()`
Resets the iterator, so it points to the last element within a container.

2.2.4 Class PltArrayIterator

This iterator iterates over the elements within an array. In addition to the public operations of the `PltBidirIterator` class, the `PltArrayIterator` class defines the following operations:

```
template<class T>
class PltArrayIterator : public PltBidirIterator<T> {
public:
    PltArrayIterator(const PltArrayed<T> &);
};
```

`PltArrayIterator(const PltArrayed<T> &)`
Constructs an iterator that iterates over the elements of an instance of the template class `PltArrayed<T>`.

For a description of the remaining operations please refer to the interface description of the abstract base class `PltBidirIterator`.

2.2.5 Class PltHandleIterator

Instances of the abstract iterator class `PltHandleIterator` iterate over the elements of containers, which are managed by `PltPtrHandles`. The class `PltHandleIterator` behaves much like the class `PltIterator` (and in fact shares an private common superclass with it):

```
template<class T>
class PltHandleIterator {
public:
```

```
    virtual operator const void * () const = 0;
    virtual PltPtrHandle<T> operator * () const = 0;
    virtual PltHandleIterator & operator ++ () = 0;
    virtual void toStart() = 0;
}
```

`virtual operator const void * () const`
: Cast operator that allows to check whether the iterator has iterated over all elements. In case all elements have been iterated over, the cast operator returns `0` (the null pointer). See the "Usage" section of this chapter for an example.

`virtual PltPtrHandle<T> operator * () const`
: Provides access to the element in a container the iterator currently points to. The access is granted through a `PltPtrHandle`, to keep track of the lifetime of the element.

`virtual PltIterator & operator ++ ()`
: Prefix operator that moves the iterator to the next element within a container.

`virtual void toStart()`
: Resets the iterator, so it points to the first element within a container.

2.2.6 Class PltHashIterator

This iterator iterates over the associations within a hash table. In addition to the public operations of the `PltIterator` class, the `PltHashIterator` class defines the following operations:

```
template <class K, class V>
class PltHashIterator : public PltIterator< PltAssoc<K,V> >,
                        private PltHashIterator_base {
public:
    PltHashIterator(const PltHashTable<K,V> & t);
    virtual operator const void * () const;
    virtual const PltAssoc<K,V> * operator -> () const;
    virtual PltIterator< PltAssoc<K,V> > & operator ++ ();
    virtual void toStart();
};
```

`PltHashIterator(const PltHashTable<K,V> &t)`
: Constructs an iterator that iterates over the elements of an instance of the template class `PltHashTable<K,V>`. Please see the appropriate section for more information about the `PltAssoc<K, V>` class.

`virtual const PltAssoc<K,V> * operator -> () const`
: Returns the association, the iterator currently points to.

For a description of the remaining operations please refer to the interface description of the abstract base class `PltIterator`.

2.2.7 Class PltListIterator

This iterator iterates over the elements within a list. In addition to the public operations of the `PltBidirIterator` class, the `PltListIterator` class defines the following operations:

```
template <class T>
class PltListIterator : public PltBidirIterator<T>,
```

```
                        private PltListIterator_base {
public:
    PltListIterator(const PltList<T> &list, bool startAtEnd=false);
    virtual operator const void * () const;
    virtual const T & operator * () const;
    virtual PltListIterator<T> & operator ++ ();
    virtual void toStart();
    virtual PltListIterator<T> & operator -- ();
    virtual void toEnd();
};
```

`PltListIterator(const PltList<T> &list, bool startAtEnd=false)`

Constructs an iterator which iterates over the elements of an instance of the template class `PltList<T>`. The parameter `startAtEnd` controls whether the iterator starts at the beginning or end of the list.

For a description of the remaining operations please refer to the interface description of the abstract base class `PltBidirIterator`.

2.2.8 Class PltIListIterator

This iterator iterates over the elements within an intrusive list. In addition to the public operations of the `PltBidirIterator` class, the `PltIListIterator` class defines the following operations:

```
template <class T>
class PltIListIterator : public PltBidirIterator<T>,
                         private PltListIterator_base {
public:
    PltIListIterator(const PltIList<T> &list, bool startAtEnd=false);
    virtual operator const void * () const;
    virtual const T* operator -> () const;
    virtual PltIListIterator<T> & operator ++ ();
    virtual void toStart();
    virtual PltIListIterator<T> & operator -- ();
    virtual void toEnd();
};
```

`PltIListIterator(const PltIList<T> &list, bool startAtEnd=false)`

Constructs an iterator which iterates over the elements of an instance of the template class `PltIList<T>`. The parameter `startAtEnd` controls whether the iterator starts at the beginning or end of the list.

For a description of the remaining operations please refer to the interface description of the abstract base class `PltBidirIterator`.

2.2.9 Class PltPQIterator

This iterator iterates over the elements within a priority queue. In addition to the public interface of the `PltIterator` class, the `PltPQIterator` class defines the following operations:

```
template <class T>
class PltPQIterator : public PltIterator<T> {
public:
    PltPQIterator(const PltPriorityQueue<T> &);
```

```
    virtual operator const void * () const;
    virtual const T & operator * () const;
    virtual PltIterator<T> & operator ++ ();
    virtual void toStart();
};
```

`PltPQIterator(const PltPriorityQueue<T> &)`

Constructs an iterator which iterates over the elements of an instance of the template class `PltPriorityQueue<T>`.

For a description of the remaining operations please refer to the interface description of the abstract base class `PltIterator`.

2.3 Arrays

2.3.1 Usage

The array objects provided by the communication library are fixed in size. This is no real restriction as you can use lists, if you don't know the number of entries you will work with in advance. Array objects are almost natural to use, as the following short example will show. First, we create an array, that will receive the square numbers from 0 to 99, and fill in the numbers:

```
PltArray<int> ai(100);
for (int i = 0; i < ai.size(); ++i) {
    ai[i] = i * i;
}
```

Then we print the square numbers by accessing the individual elements of the array with the familiar array syntax using square brackets:

```
for (int j = ai.size() - 1; j >= 0; --j) {
    cout << j << "*" << j << " = " << ai[j] << endl;
}
```

2.3.2 Class PltArray

Following is the public interface for the class `PltArray`:

```
template <class T>
class PltArray : public PltArrayed<T> {
public:
    PltArray(size_t size = 0);
    PltArray(size_t size, T * p, enum PltOwnership os);
    const T & operator[] (size_t i) const;
    PltArray copy() const;
    T * getPtr() const;
    T & operator[] (size_t i);
    PltArray<T> & operator = (const PltArray<T> &);
    virtual size_t size() const;
    virtual PltArrayIterator<T> * newIterator() const;
};
```

`PltArray(size_t size = 0)`

Constructs a new array that is fixed in size. If you don't specify a size, then an empty array of size 0 is constructed. Such arrays are only useful if you later copy another array into this one. Otherwise you'll get an array that holds exactly `size` entries. You then can access the individual elements using the familiar array operator "`[]`".

`PltArray(size_t size, T *p, enum PltOwnership os)`
: Constructs an array object with a fixed number of entries as indicated by `size` using an preallocated array you already got your hands on. The parameter `p` then points to the preallocated array, which has an ownership of `os`. The ownership `os` can be either `PltOsArrayNew`, `PltOsMalloc` or `PltOsUnmanaged`.

`const T & operator[] (size_t i) const`
: Provides access to individual elements within the array without modifying the contents of the array.

`T & operator[] (size_t i)`
: Provides access to individual elements within the array.

`PltArray copy() const`
: Constructs a copy of an array.

`T * getPtr() const`
: Returns a pointer to the representation of the array. You should be careful with this pointer as the lifetime of the objects is controlled by the array object.

`PltArray<T> & operator = (const PltArray<T> &)`
: Copies an array to another. Both arrays then share the elements, therefore no duplication of elements occours.

`virtual size_t size() const`
: Returns the number of elements in the array.

`virtual PltArrayIterator<T> * newIterator() const`
: Returns a new iterator suitable for iterating over the elements of the array. The caller must eventually delete the iterator after use.

2.4 Handles

2.4.1 Usage

Handles – which are in fact more appropriately reference counters – are used to manage the ownership of differend kinds of storage. A handle controls the lifetime span of the storage (for example an C++ object) it is responsible for. This way, C++ objects aren't destroyed until no-one is interested in them anymore (and the reference counter has reached zero). Whenever you assign or copy a handle to another handle, the reference counter of the handle to be set is decremented and the counter of the handle to be copied is incremented.

In order that, when the reference counter reaches zero, a handle can automatically free the memory or C++ object it owns, it must know what kind of storage it manages. If this storage is administered using the traditional C functions `malloc()` and `free()`, the ownership of the storage is said to be `PltOsMalloc`. However, C++ objects allocated with `new` and destroyed using the `delete` operator must have an ownership of `PltOsNew`. A third kind of ownership is `PltOsArrayNew`, which is used for arrays of C++ objects. In the C++ language, when you want to destroy an array of objects, you must tell this the compiler explicitly, so the compiler can generate code that calls the destructors for the individual objects within the array.

Handles can also work with "unmanaged" memory, which either comes from static storage or is owned by another library and which must not be automatically destroyed when the reference counter reaches zero. In this case you must specify an ownership of `PltOsUnmanaged`.

```
enum PltOwnership
{
    PltOsUnmanaged,
```

```
    PltOsMalloc,
    PltOsNew,
    PltOsArrayNew
};
```

In practice, the first flavour of handles, the class `PltPtrHandle`, is used like a pointer. A small example shall enlighten this. First we create an unbound handle using the template class `PltPtrHandle<>` and specify between the angle brackets the type of variable or object we want to manage. In our example this will be a simple integer – but also more challenging data types do work.

```
PltPtrHandle<int> hi;
```

At this point the handle `hi` isn't yet bound to an object. So let us now create an integer object, and put it under the reference control of the handle using the `bindTo()` operation. Because the integer object was created with `new`, you must specify `PltOsNew` as the ownership.

```
int *pi = new int(3);
hi.bindTo(pi, PltOsNew);
```

The `bindTo()` operation returns `true` if it succeeds, otherwise `false` (that's the way with today's booleans, they can only think in terms of black and white, and not shades of gray...). Such a failure can be due to the lack of memory.

If the bind operation succeeds, you must not use the `pi` pointer anymore. Instead you access the integer value using the familiar pointer syntax:

```
int i = *hi + 42;
```

If you are finally through with the handle you can simply destroy it using `delete` – in case you once have allocated the handle using `new`. In our example, the handle has been allocated in automatic storage, so it will be destroyed when the program execution leaves the current scope. The reference counter will then be decremented, and because it reaches zero, the managed object eventually gets destroyed.

You can easily "duplicate" handles if someone other is interested in a managed object too.

```
PltPtrHandle<int> another_hi;
another_hi = hi;
```

Once again, after we duplicated the handle, the managed integer isn't destroyed until **both handles** go out of scope.

Array handles (class `PltArrayHandle`) are the second flavour of handles within the communication library. They are used when you need to manage an array of objects.

```
int *pia = new int[3];
PltArrayHandle<int> hia(pia, PltOsArrayNew);
```

The individual objects within the array can be accessed using the familiar `[]` syntax.

```
hia[2] = 42;
```

2.4.2 Class PltPtrHandle

Following is the public interface for the handle class `PltPtrHandle`:

```
template<class T>
class PltPtrHandle : private PltHandle {
public:
    PltPtrHandle();
    PltPtrHandle(T *p, enum PltOwnership);
    PltPtrHandle(const PltPtrHandle &);
```

```
    operator bool () const;
    T& operator*() const;
    T* operator->() const;
    T* getPtr() const;

    bool operator == (const PltPtrHandle &rhs) const;
    bool operator != (const PltPtrHandle &rhs) const;

    PltPtrHandle & operator=(const PltPtrHandle &rhs);
    bool bindTo(T *, enum PltOwnership = PltOsNew);
};
```

`PltPtrHandle()`
: Default constructor – constructs an unbound handle.

`PltPtrHandle(T *p, enum PltOwnerShip os)`
: Constructs a bound handle for an object of type `T`. The parameter `os` describes the ownership and thus how the object was created. See the previous section for the possible types of ownership. If there is enough memory, the new handle is bound to the object `p` points to. But if there is not enough memory to bind, the object pointed to by `p` will be **destroyed immediately** by this constructor and instead an unbound handle is constructed. You can check whether the bind operation succeeded by casting the handle to `bool`. You'll then get back the value `true`, if there were no problems.

`PltPtrHandle(const PltPtrHandle &h)`
: Duplicates a handle.

`operator bool () const`
: Returns `true`, if the handle is bound (it manages memory or an object). This cast operator is necessary so you can test the "boundness" in a simple `if(handle)` statement.

`T& operator*() const`
: This operator ensures that you can dereference the bound object using the familiar pointer dereferencing syntax.

`T* operator->() const`
: Same reason as above.

`T* getPtr() const`
: Returns the pointer to the object which is bound to this handle. Do not store this pointer anywhere, or in the future you may dereference a deleted object.

`bool operator == (const PltPtrHandle &rhs) const`
: Compares two pointer handles by comparing their pointers. If both handles either point to the same object or to `0`, the operator will return `true`, otherwise `false`.

`bool operator != (const PltPtrHandle &rhs) const`
: Compares two pointer handles by comparing their pointers. If both do not point to the same object, the operator will return `true`, otherwise `false`.

`PltPtrHandle & operator=(const PltPtrHandle &rhs)`
: This operator ensures that you can assign (copy) a handle to another handle without messing up the reference counting. First, the reference counter on the right side of the assignment is incremented as the bound object is now referenced once more. Then, the reference counter of the handle on the left hand side is decremented, because the handle now points to a new object. If this reference counter reaches zero, the (old) bound object is freed.

`bool bindTo(T *p, enum PltOwnership os = PltOsNew)`

Binds the object pointed to by `p` to this handle. If the operation succeeds, then the bind operation returns `true` and you can't use `p` any longer. The operation may fail due to memory constraints and in this case it returns `false`. If the reference counter of the object being managed once will reach zero, then the object will be destroyed according to the ownership specified in `os`.

If the handle is already bound at the time you call the `bindTo()` method, the old object the handle points to is unbound and instead the new object referenced by `p` is bound to the handle. If you specify a `0` pointer for `p`, then the handle is unbound.

2.4.3 Class PltArrayHandle

The array handle class `PltArrayHandle` is much like the handle class `PltPtrHandle`. But in contrast to the latter class, the `PltArrayHandle` manages arrays of objects. The handle then allows access to individual entries of the array using the traditional indexing syntax with `[]`. Below is the public interface of the class `PltArrayHandle`:

```
template<class T>
class PltArrayHandle : private PltHandle<T> {
public:
    PltArrayHandle();
    PltArrayHandle(T *p, enum PltOwnership);
    PltArrayHandle(const PltArrayHandle &);

    operator bool () const;
    T& operator[](size_t) const;
    T* getPtr() const;

    bool operator == (const PltPtrHandle &rhs) const;
    bool operator != (const PltPtrHandle &rhs) const;

    PltArrayHandle & operator=(const PltArrayHandle &rhs);
    bool bindTo(T *, enum PltOwnership = PltOsArrayNew);
};
```

`PltArrayHandle()`

Default constructor – constructs an unbound array handle.

`PltArrayHandle(T *p, enum PltOwnership os)`

Constructs a bound array handle for the array pointed to by `p` with a memory ownership of `os`. Possible values for `os` are `PltOsArrayNew`, `PltOsMalloc`, and `PltOsUnmanaged`. If there is not enough memory to bind, the object pointed to by `p` will be **destroyed immediately** by this constructor and instead an unbound handle is constructed.

`PltArrayHandle(const PltArrayHandle &h)`

Duplicates an array handle.

`operator bool () const`

Cast operator that allows you to check whether the handle is currently bound using a typecast to the type `bool`, for example in an `if(handle)` statement. If the handle is bound, then the result is `true`.

`T& operator[](size_t idx) const`

Allows access to the element with the index `idx` of the bound array.

`T* getPtr() const`
Same as `PltPtrHandle::getPtr()`.

`bool operator == (const PltArrayHandle &rhs) const`
Compares two array handles by comparing their pointers. If both handles either point to the same array or to `0`, the operator will return `true`, otherwise `false`.

`bool operator != (const PltArrayHandle &rhs) const`
Compares two array handles by comparing their pointers. If both do not point to the same array, the operator will return `true`, otherwise `false`.

`PltArrayHandle & operator=(const PltArrayHandle &rhs)`
Assignment operator that ensures that the managed objects are properly reference-counted and possibly destroyed.

`bool bindTo(T *p, enum PltOwnership os = PltOsArrayNew)`
Binds an array `p` with an ownership of `os` to the handle. See `PltPtrHandle::bindTo()` for a description of the parameters.

2.4.4 Class PltKeyHandle

The "key handles" of the class `PltKeyHandle` are specialised `PltPtrHandle`s, which can be used with hash tables. In addition to the basic functions of a handle, the class `PltKeyHandle` implement the key interface necessary for objects which should be put into hash tables (see the chapter about hash tables below for more information). For this to work, the objects a key handle manages must also implement the key interface. Following is the public interface of the class `PltKeyHandle`:

```
template <class T>
class PltKeyHandle : public PltPtrHandle<T> {
public:
    PltKeyHandle();
    PltKeyHandle(T *p, enum PltOwnership);
    PltKeyHandle(const PltPtrHandle<T> &);
    PltKeyHandle(const PltKeyHandle &);

    unsigned long hash() const;
    bool operator == (const PltKeyHandle & h) const;
};
```

`PltKeyHandle()`
Constructs an unbound key handle.

`PltKeyHandle(T *p, enum PltOwnership)`
Constructs a bound key handle for an object of type `T` with an ownership of `PltOwnership`.

`PltKeyHandle(const PltPtrHandle<T> &)`
Creates a key handle from a ordinary pointer handle.

`PltKeyHandle(const PltKeyHandle &)`
Duplicates a key handle.

`unsigned long hash() const`
Calculates a hash key based on the object's current state the handle manages.

`bool operator == (const PltKeyHandle & h) const`
Compares two key objects and returns `true`, if they are equal. Comparing keys is different

from comparing their hash values. Two objects (keys) can return the same hash value although they have different contents.

2.4.5 Class PltHandle

The abstract superclass of `PltPtrHandle` and `PltArrayHandle`. This class must never be directly instantiated.

2.5 Hash Tables

2.5.1 Usage

Hash tables map keys to values and can therefore be seen as an associative array. The keys can be of any class as long as they provide the public interface of the `PltKey` class. The programmer is responsible for the storage used by keys and values. The hash stores copies of the key and the value objects.

In contrast to lists the hash tables normally provide fast access to individual values independent of the number of values currently stored in the table.

The following example implements a very basic dictionary, which translates swabian into english.

```
PltHashTable<PltString, PltString> dict;
PltString sw1("Käschtle"), e1("tiny box");
PltString sw2("Schwäbische Alb"), e2("Swabian's high mountains");

dict.add(sw1, e1);
dict.add(sw2, e2);

PltString term("Käschtle"), answer;

if ( dict.query(term, answer) ) {
    cout << "A '" << term << "' is a " << answer << endl;
} else {
    cout << "I don't know what a '" << term << "' means." << endl;
}
```

The class representing the keys must implement the following interface (otherwise such a key class can't be used with hash tables):

```
class someClass {
public:
    unsigned long hash() const = 0;
    bool operator == (const someClass &) const = 0;
};
```

`unsigned long hash() const`
: Calculates a hash key based on the object's current state.

`bool operator == (const someClass &) const`
: Compares two key objects and returns `true`, if they are equal. Comparing keys is different from comparing their hash values. Two objects (keys) can return the same hash value although they have different contents.

  It is very important to note, that although

`key1 == key2` ⇒ `key1.hash() == key2.hash()`

always holds, the opposite conclusion doesn't necessarily need to hold:

`key1.hash() == key2.hash()` ⇏ `key1 == key2`

2.5.2 Class PltHashTable

Following is the public interface of the hash table class:

```
template <class K, class V>
class PltHashTable : public PltDictionary<K,V>,
                     private PltHashTable_base {
public:
    PltHashTable(size_t mincap=11,
                 float highwater=0.8,
                 float lowwater=0.4);
    virtual bool query(const K&, V&) const;

    virtual bool add(const K&, const V&);
    virtual bool remove(const K&, V&);
    virtual bool update(const K&, const V&, V&, bool &);

    PltHashIterator<K,V> * newIterator() const;
    size_t size() const;
};
```

`PltHashTable(size_t mincap=11, float highwater=0.8, float lowwater=0.4)`

Default constructor that constructs an empty hash table. The table capacity will never shrink below the given minimum capacity. When the highwater mark is reached, the table capacity is automatically increased. When the lowwater mark is crossed, the table capacity is decreased again. Both watermarks are expressed as fractions of the current capacity, that is `0.8` means 80%.

`virtual bool query(const K& key, V& value) const`

Looks up a value using the given `key` and stores a reference to the value in `value`. If the value could be found using the key, the `query()` method returns `true`, otherwise `false`.

`virtual bool add(const K& key, const V& value)`

Adds a new `value` to the hash table, which can be accessed through `key`. If there's already a value stored with the same key, this key to value association isn't changed and the method returns `false`, as it would due to the lack of free memory. If the method succeeds, it returns `true`.

`virtual bool remove(const K& key, V& value)`

Removes the association bound to `key` from the hash table. If the key could be found and removed then this method returns a reference to the value associated with the key in `value` an returns `true`.

`virtual bool update(const K& key, const V& value, V& old_value, bool& old_value_valid)`

Updates (or adds) an association. If the hash table already contains an association with the key `key`, then the old value is returned in `old_value` and `old_value_valid` is set to `true`. If there can no such association be found in the hash table, `old_value_valid` is set to `false`. In every case, `key` is then associated with `value`.

`PltHashIterator<K,V> * newIterator() const`
: Constructs and returns a iterator suitable for iteration over the associations stored in the hash table. The caller eventually must delete the iterator after use, or she/he will face a memory leak.

`size_t size() const`
: Returns the number of items (associations between keys and values) currently stored in the hash table.

2.5.3 Struct PltAssoc

The hash tables manage associations which are represented by structure of type `PltAssoc`. Below is the public interface of the structure `PltAssoc`:

```
template <class K, class V>
struct PltAssoc : public PltAssoc_ {
public:
    K a_key;
    V a_value;

    PltAssoc(K k, V v);
    virtual const void * key() const;
};
```

`K a_key`
: Contains the key part of the association.

`V a_value`
: Contains the value part of the association.

`PltAssoc(K k, V v)`
: Constructs a new association between a key `k` and a value `v`.

`virtual const void * key() const`
: Returns a pointer to the key part of the association.

2.5.4 Class PltKeyPtr

The hash table class `PltHashTable` works with copy semantics, that is, when you put an item into a hash table, it stores a copy of the item. An advantage is, that you don't need to worry about memory ownership and memory leaks. If you need to manage larger items, you can either use `PltKeyPtr`s or `PltHashKey`s with hash tables. In this case, you only store the `PltKeyPtr` or the `PltHashKey` in the hash table, while these objects reference the real item.

```
template <class T>
class PltKeyPtr {
public:
    PltKeyPtr(T *p = 0);
    unsigned long hash() const;
    bool operator == (const PltKeyPtr & p) const;
    T & operator * () const;
    T * operator ->() const;
};
```

`PltKeyPtr(T *p = 0)`
Constructs a `PltKeyPtr` object which references the item `p`. The class `T` must support the key interface (see above for details).

`unsigned long hash() const`
Calculates a hash key based on the referenced object's current state.

`bool operator == (const PltKeyPtr & p) const`
Compares two key objects and returns `true`, if they are equal. Comparing keys is different from comparing their hash values. Two objects (keys) can return the same hash value although they have different contents.

`T & operator * () const`
Provides access to the object this key pointer object references.

`T * operator ->() const`
Provides access to the object this key pointer object references.

2.6 Lists

2.6.1 Usage

The C++ communication library provides two kinds of lists: non-intrusive and intrusive lists. The intrusive lists require that the elements they manage are derived from the special node class `PltListNode`. For it they are more efficient than non-intrusive lists.

With both kinds of lists you can add new items to the list at either the beginning or the end. The lists also maintain the order of the items. The following example shows this:

```
int *pi;
PltList<int> list;

for (int i = 1; i <= 5; i++ )
    list.addLast(*new int(i * i));

while ( !list.isEmpty() ) {
    pi = &list.removeFirst();
    cout << pi << endl;
    delete pi;
}
```

2.6.2 Class PltList

Below is the public interface of the non-intrusive list class `PltList`:

```
template <class T>
class PltList : public PltContainer<T>, private PltList_base {
public:
    PltList();
    virtual ~PltList();

    bool addFirst(const T & t);
    bool addLast(const T & t);
    bool remove(const T & t);
    T removeFirst();
    T removeLast();

    virtual bool isEmpty() const;
    virtual size_t size() const;
```

```
    virtual PltListIterator<T> * newIterator() const;
};
```

`PltList()`
: Default constructor. It constructs an empty list.

`~PltList()`
: Destroys the list.

`bool addFirst(const T & t)`
: Adds another item `T` to the beginning of the list.

`bool addLast(const T & t)`
: Adds another item `T` to the end of the list.

`bool remove(const T & t)`
: Removes the item `t` from the list and returns `true`, if there was such an item in the list.

`T removeFirst()`
: Removes the item at the beginning of the list and returns it. You should first make sure that the list contains at least one item (using `isEmpty()`) before calling this method.

`T removeLast()`
: Removes the item at the end of the list and returns it. You should first make sure that the list contains at least one item (using `isEmpty()`) before calling this method.

`virtual bool isEmpty() const`
: Checks whether the list is empty and in this case this method returns `true`.

`virtual size_t size() const`
: Returns the number of items in the list.

`virtual PltListIterator<T> * newIterator() const`
: Constructs an iterator object that can be used to iterate over the elements of the list. You must delete the iterator object after use (sorry, but for reasons of political correctness we advise you not delete objects but to reuse them over and over again).

2.6.3 Class PltIList

The intrusive list class `PltIList` requires that the objects it manages are derived from the special node class `PltListNode`. Following is the public interface of the intrusive list class:

```
template <class T>
class PltIList : public PltContainer<T>, private PltList_base {
public:
    PltIList();

    bool addFirst(T* p);
    bool addLast(T* p);
    T* remove(T* p);
    T* removeFirst();
    T* removeLast();

    virtual bool isEmpty() const;
    virtual size_t size() const;
    virtual PltIListIterator<T> * newIterator() const;
};
```

The public interface is the similiar to the public interface of the non-intrusive list class `PltList`, so please refer to the previous section for an explanation of the interface. The only difference is, that an `PltIList` works with pointers to items, whereas the non-intrusive list `PltList` uses copies.

2.7 Logging Facilities

2.7.1 Usage

During runtime of KS servers, clients, and managers it is sometimes necessary to log error conditions, warnings, or informational messages. Unfortunately, the logging facilities vary widely depending on the operating system and system platform used. The "libplt" therefore provides a unified interface for common logging needs through the abstract class `PltLog`.

Within your application – during startup – you should instanciate a logger object (this example assumes that you're using the system logger of the Un*x operating system):

```
PltSyslog *logger = new PltSyslog();
```

During runtime of your application, whenever you feel to issue a logged message, you simply call one of the methods `Info()`, `Debug()`, `Warning()`, `Error()`, or `Alert()` (depending of the severity of the message). You don't need to have the logger object ready at hand for these methods to call. You only must have already created a logger object.

```
PltLog::Alert("The spätzle (swabian spaghetti) are burnt.");
```

On the other side, if you have the pointer to the logger object ready at hand or you want to use more than one logger object, you can use the following call to issue a logged message:

```
logger->info("Don't worry. It's already too late.");
```

When shutting down the application, you must destroy the logger object. This is necessary, as the logger object may need to do some cleanup depending on the operating system the "libplt" is compiled for.

```
delete logger;
```

2.7.2 Class PltLog

Following is the public interface of the abstract logger class `PltLog`:

```
class PltLog {
public:
    static void SetLog(PltLog & log);
    static PltLog * GetLog();
    static void Info(const char *msg);
    static void Debug(const char *msg);
    static void Warning(const char *msg);
    static void Error(const char *msg);
    static void Alert(const char *msg);

    PltLog();
    virtual ~PltLog();
    virtual void info(const char *msg) = 0;
    virtual void debug(const char *msg) = 0;
    virtual void warning(const char *msg) = 0;
    virtual void error(const char *msg) = 0;
    virtual void alert(const char *msg) = 0;
}
```

`PltLog()`
: Constructs a logger object. If there is currently no other logger object registered as the current logger object, then this object is made the current logger.

`~PltLogger()`
: Deconstructs a logger object. If the logger object is also the current logger object, it is deregistered as the current logger object.

`static PltLog * getLog()`
: Returns a pointer to the current logger object. This static member function allows you to call the logger object even if you don't have a pointer to it ready at hand. Please refer to the "Usage" section above for an example of how to use `getLog()`.

`static void SetLog(PltLog & log)`
: Registers the logger object `log` as the current logger object, so you can refer to it using the marshalling logging functions `Info()`, `Debug()`, `Warning()`, `Error()`, and `Alert()`.

`static void Info(const char *msg)`
: Marshals the informational message `msg` using the current logger object. If there is current logger object registered, then no logging happens.

`static void Debug(const char *msg)`
: Marshals the debugging message `msg` using the current logger object. If there is current logger object registered, then no logging happens.

`static void Warning(const char *msg)`
: Marshals the warning message `msg` using the current logger object. If there is current logger object registered, then no logging happens.

`static void error(const char *msg)`
: Marshals the error message `msg` using the current logger object. If there is current logger object registered, then no logging happens.

`static void alert(const char *msg)`
: Marshals the alert message `msg` using the current logger object. If there is current logger object registered, then no logging happens.

`virtual void info(const char *msg)`
: Logs the informational message `msg` using the logger object on which you call this method.

`virtual void debug(const char *msg)`
: Logs the debug message `msg` using the logger object on which you call this method.

`virtual void warning(const char *msg)`
: Logs the warning message `msg` using the logger object on which you call this method.

`virtual void error(const char *msg)`
: Logs the error message `msg` using the logger object on which you call this method.

`virtual void alert(const char *msg)`
: Logs the alert message `msg` using the logger object on which you call this method.

2.7.3 Class PltSyslog

Following is the public interface of the class `PltSyslog`, which can be used for logging purposes on Un*x platforms based on the syslog(1) facility. This logger class can handle more than one instance, in that case the last logger object instanciated is used as the "current logger" when calling the static memeber functions `PltLog::Info()` etc. You can make use of the other

instances by calling their virtual member functions (like `info()`). If you destroy the current logger, then no further logging is possible through the static member functions like `PltLog::Info()`until you call `PltLog::SetLog()`.

```
class PltSyslog : public PltLog {
public:
    PltSyslog(const char * ident = 0,
              int logopt = LOG_PID,
              int facility = LOG_USER);
    virtual ~PltSyslog();
    virtual void info(const char *msg);
    virtual void debug(const char *msg);
    virtual void warning(const char *msg);
    virtual void error(const char *msg);
    virtual void alert(const char *msg);
};
```

```
PltSyslog(const char *ident = 0, int logopt = LOG_PID,
     int facility = LOG_USER)
```

Constructs a logger object. All logging messages are identified by `ident` (for example the server's or application's name). For the possible values of `logopt` and `facility` please refer to the man pages for openlog(3).

For a description of the remaining methods please refer to the superclass `PltLog`.

2.7.4 Class PltCerrLog

The specialised logger class `PltCerrLog` uses the `cerr` stream of an application to report logging info to the console. The advantage of this simple logger is, that it is available on all operating system platforms.

```
class PltCerrLog : public PltLog {
public:
    PltCerrLog(const char * ident = 0);
    ~PltCerrLog();
    virtual void info(const char *msg);
    virtual void debug(const char *msg);
    virtual void warning(const char *msg);
    virtual void error(const char *msg);
    virtual void alert(const char *msg);
};
```

```
PltCerrLog(const char *ident = 0)
```

Constructs a logger object. All logging messages are identified by `ident` (for example the server's or application's name) and are simply printed out on the stderr channel of the console or terminal window.

For a description of the remaining methods please refer to the superclass `PltLog`.

2.7.5 Class PltNtLog

The specialised logger class `PltNtLog` provides access to the NT system logger. Below you'll find the public interface of this logger class.

```
class PltNtLog : public PltLog {
```

```
public:
    PltNtLog(const char * ident = 0);
    virtual ~PltNtLog();
    virtual void info(const char *msg);
    virtual void debug(const char *msg);
    virtual void warning(const char *msg);
    virtual void error(const char *msg);
    virtual void alert(const char *msg);
};
```

`PltNtLog(const char * ident = 0)`

Constructs a new NT logger object and makes it the current logger object. You may supply an individual textual identification of your logger message source in `ident`. If you don't, then the default identification `"ACPLT/KS"` will be used instead.

For a description of the remaining methods please refer to the superclass `PltLog`.

2.7.6 Class PltLogStream

The class `PltLogStream` is not a logger object class, but rather is a companion to the logger classes. An object of the class `PltLogStream` inherits the output capabilities of the `ostream` C++ streams class, thus featuring various ways of formatting output. Only if you have build your logging message completely, you ask the logging stream object to send the message to a real logger object. For example, create a logger object (depending on the platform, this might be a different class than that in the example) and a logging stream:

```
PltSyslog logger; // Use PltNtLog on NT platforms
PltLogStream ls;
```

By default, the logging stream `ls` will use the default logger object, when you construct the `ls` object using the default constructor. But you're free to change this association at any time lateron. Whenever you want to log something, you just build the message as you would to it using the C++ streams:

```
ls << "Got " << packages << " packages of spätzle";
```

If the message is complete, ask the logging stream to send it to the logger object. Depending on the severity of the message, you can also use on of the other log functions (`info()`, `debug()`, `warning()`, `error()`) of the logging stream:

```
ls.alert();
```

The logging stream then is reset, so you can build the next message fresh from the start.

The public interface of the class `PltLogStream` is as follows:

```
class PltLogStream : public ostrstream {
public:
    PltLogStream();
    PltLogStream(PltLog &logger);
    void setLogger(PltLog &logger);
    PltLog *getLogger();

    void info();
    void debug();
    void warning();
    void error();
    void alert();
};
```

`PltLogStream()`
: Constructs a logging stream, which uses the default logger object. If no default logger object exists when you ask a logging stream to log its message, then the message will be lost.

`PltLogStream(PltLog &logger)`
: Constructs a logging stream, which uses the logging object specified in `logger` for its logging purposes.

`void setLogger(PltLog &logger)`
: Changes the current association, so that the logging stream now uses the logging object specified as `logger`.

`PltLog *getLogger()`
: Returns a pointer to the logger object, with which this logging stream is currently associated. If there is currently no such associaten (the logging stream uses the default logging object), then a `0` pointer is returned.

`void info()`
: Logs the current message using the severity level "info". The stream then is reset, so you can build a new message.

`void debug()`
: Logs the current message using the severity level "debug". The stream then is reset, so you can build a new message.

`void warning()`
: Logs the current message using the severity level "warning". The stream then is reset, so you can build a new message.

`void error()`
: Logs the current message using the severity level "error". The stream then is reset, so you can build a new message.

`void alert()`
: Logs the current message using the severity level "alert". The stream then is reset, so you can build a new message.

2.8 Priority Queues

2.8.1 Usage

Priority queues are somewhat similiar to lists, but in contrast to lists the items within a priority queue are partially ordered. Thus, priority queues can only manage items for which an sorting

order is defined. The position of new items which are added to the queue is therefore determined by this sorting order. The partially ordering garantuees that the first item in the priority queue is always the smallest item according to the sorting order. But if you iterate over the remaining items, there is no garantuee that they are properly ordered.

An application for priority queues are timer queues which are used within KS server objects.

2.8.2 Class PltPriorityQueue

Below is the public interface of the class `PltPriorityQueue`.

```
template <class T>
class PltPriorityQueue : public PltContainer<T> {
public:
    PltPriorityQueue(size_t growsize = 16);
    virtual ~PltPriorityQueue();
    bool isEmpty() const;
    T peek() const;
    bool add(T elem);
    T removeFirst();
    bool remove(T elem);
    PltPQIterator<T> * newIterator() const;
    size_t size() const;
};
```

`PltPriorityQueue(size_t growsize = 16)`
: Constructs a new priority queue. When adding new entries to the queue and the internal table for entries is full, the queue size (and thus the table) will grow by `growsize` elements.

`virtual ~PltPriorityQueue()`
: Disposes a priority queue.

`bool isEmpty() const`
: Returns `true`, if there are currently no entries in the priority queue.

`T peek() const`
: Returns the first entry but does not remove this entry from the priority queue. The priority queue must no be empty when calling this method.

`bool add(T elem)`
: Adds a new entry to the priority queue. Accordings to the sorting order defined for objects of class `T`, the new entry is inserted appropriately.

`T removeFirst()`
: Removes the first entry from the priority queue and returns it to the caller.

`bool remove(T elem)`
: Looks up the entry `elem`, and if it's found it is removed from the priority queue. In this case, this method returns `true`, otherwise – guess what? – `false`.

`PltPQIterator<T> * newIterator() const`
: Constructs and returns an iterator suitable for iterating over the entries of the priority queue.

`size_t size() const`
: Returns the number of entries currently stored in the priority queue.

2.8.3 Class PltPtrComparable

When working with priority queues you sometimes want to store references to objects and not copies of objects in the queue. In this case you should put objects of class `PltPtrComparable` in the priority queue, which in turn reference the real objects. The public interface of the class `PltPtrComparable` allows to compare different objects managed by this special pointers.

```
template <class T>
class PltPtrComparable {
public:
    bool operator <  ( PltPtrComparable<T> t2);
    bool operator >  ( PltPtrComparable<T> t2);
    bool operator != ( PltPtrComparable<T> t2);
    bool operator == ( PltPtrComparable<T> t2);
    bool operator <= ( PltPtrComparable<T> t2);
    bool operator >= ( PltPtrComparable<T> t2);

    operator T * ();
    T * operator ->() const;
    T & operator *() const;
    PltPtrComparable(T * p = 0);
};
```

`PltPtrComparable(T * p = 0)`
: Constructs a `PltPtrComparable` object which references the object pointed to by `p`. In contrast to a handle, these objects do not manage the storage occupied by the referenced object.

`operator T * ()`
: Cast operator which return a pointer to the object managed by this "comparable pointer".

`T * operator ->() const`
: Provides access to the object managed by this "comparable pointer".

`T & operator *() const`
: Provides access to the object managed by this "comparable pointer".

2.9 Strings

2.9.1 Usage

The class `PltString` acts much like an ordinary C string but reduces unnecessary copying by reference counting. A `PltString` can be used in most places where you otherwise would use a C string. A small example shall enlighten this. First, let us create two `PltString` objects. Whereas the contents of the first `PltString` are immediately set when the object is created, the contents of the second `PltString` are assigned lateron.

```
PltString hello("Hello, ");
PltString world;
world = "world";
```

Lateron the `PltString` can be used like any ordinary C string.

```
cout << hello << world << endl;
```

2.9.2 Class PltString

Following is the public interface of the string class `PltString`:

```
class PltString {
public:
    PltString();
    PltString(const char *, size_t len=PLT_SIZE_T_MAX);
    PltString(const PltString &);
    PltString(const char *, const char *);
    virtual ~PltString();

    PltString & operator = (const char *);
    PltString & operator = (const PltString &);

    static PltString fromInt(int v);

    virtual unsigned long hash() const;

    bool ok() const;
    size_t len() const;
    const char & operator[] (size_t) const;
    operator const char *() const;

    PltString substr(size_t first, size_t len=PLT_SIZE_T_MAX) const;

    char & operator[] (size_t);
    PltString & operator += ( const PltString & );

    friend bool operator == (const PltString &, const char *);
    friend bool operator == (const PltString &, const PltString &);
    friend bool operator != (const PltString &, const char *);
    friend bool operator != (const PltString &, const PltString &);
}
```

`PltString()`
: Constructs an empty `PltString`.

`PltString(const char *s, size_t len = PLT_SIZE_T_MAX)`
: Constructs a new `PltString` and initializes it with the given string `s`, using at most `len` characters of `s`. If `s` contains less than `len` characters, then only as much characters are copied as `s` contains. Because the contents of `s` are copied, `s` may even be located in automatic storage.

`PltString(const PltString &)`
: Constructs a new `PltString` and initializes it using the contents of another `PltString`.

`PltString(const char *s1, const char *s2)`
: Constructs a new `PltString` from `s1` and `s2` by concatenating them.

`static PltString fromInt(int v)`
: Constructs a new `PltString` from the value `v`.

`virtual ~PltString()`
: Destroys an `PltString`. The rest is silence.

`PltString & operator = (const char *)`
: Assignment operator that allows you to assign a C string to a `PltString`.

`PltString & operator = (const PltString &)`
: Assignment operator that allows you to assign a `PltString` to another `PltString`.

`friend bool operator == (const PltString &, const char *)`
Compares (case sensitive) an `PltString` with a C string and returns `true`, if both are equal.

`friend bool operator == (const PltString &, const PltString &)`
Compares (case sensitive) two `PltString`s and returns `true`, if both are equal.

`friend bool operator != (const PltString &, const char *)`
Compares (case sensitive) an `PltString` with a C string and returns `true`, if both are unequal.

`friend bool operator != (const PltString &, const PltString &)`
Compares (case sensitive) two `PltString`s and returns `true`, if both are unequal.

`PltString & operator += (const PltString &)`
Adds the contents of another `PltString` to the end of this `PltString`. You can also add any ordinary C string using the "`+=`" operator, because the C++ compiler will first construct a `PltString` from the C string and then add this temporary object to the other string.

`bool ok() const`
Returns `true`, if the last string operation succeeded (that is, the string is valid). Otherwise you should discard this string.

`size_t len() const`
Returns the length of the string.

`const char & operator[] (size_t) const`
Provides access to any character within the string using the familiar "`[]`" syntax.

`operator const char *() const`
Returns a pointer to the beginning of the string's contents. You should not store this pointer but only use it to feed it into functions like `fprintf()` etc.

`char & operator[] (size_t)`
This operator implements access to the string.

`PltString substr(size_t first, size_t len=PLT_SIZE_T_MAX) const`
Returns a substring of this string, which starts at the position `first` and is at most `len` characters long.

`virtual unsigned long hash() const`
This member function as well as the next operator allow `PltString`s to be used as keys for hash tables (see `PltHashTable`).

2.10 Time

The class `PltTime` is derived from the well known `struct timeval`, and has operations added for addition and substraction. It stores time with a (possible) maximum precission of one microsecond. Here's a little example:

```
PltTime fiveSeconds(5,0);
PltTime now(PltTime::now());
PltTime then = now + fiveSeconds;   // these two ...
PltTime then2 = PltTime::now(5,0);  // ... are about the same
```

2.10.1 Class PltTime

Following is the public interface of the class `PltTime`:

```
struct PltTime : public timeval {
    PltTime(long seconds = 0, long useconds = 0);
    PltTime(const struct timeval &tv);
    bool isZero () const;
    void normalize();
    PltTime & operator += (const PltTime &t);
    PltTime & operator -= (const PltTime &t);
    static PltTime now(long seconds=0, long useconds = 0);
    static PltTime now(const PltTime &);
};
PltTime operator + (const PltTime &t1, const PltTime &t2);
PltTime operator - (const PltTime &t1, const PltTime &t2);
bool operator == (const PltTime &t1, const PltTime &t2);
bool operator != (const PltTime &t1, const PltTime &t2);
bool operator < (const PltTime &t1, const PltTime &t2);
bool operator > (const PltTime &t1, const PltTime &t2);
bool operator <= (const PltTime &t1, const PltTime &t2);
bool operator >= (const PltTime &t1, const PltTime &t2);
```

`PltTime(long seconds = 0, long useconds = 0)`
: Constructs a new time object and initializes it to the time given in `seconds` and `useconds`.

`PltTime(struct timeval &tv)`
: Constructs a new time object using another time object or a `struct timeval`.

`bool isZero() const`
: Returns `true`, if the time held by this object is zero.

`static PltTime now(long seconds=0, long useconds = 0)`
: Returns a `PltTime` object whose time stamp lies in the future by the amount given in `seconds` and `useconds`.

`static PltTime now(const PltTime &t)`
: Returns a `PltTime` object whose time stamp lies in the future by the amount given in `t`.

`PltTime & operator += (const PltTime &t)`
: Increments time by the amount of seconds and microseconds given by `t`. Some time, Einstein will get us for that operation. After performing the addition, the time object is automatically normalized such that the microseconds are always in the range of 0...999,999.

`PltTime & operator -= (const PltTime &t)`
: Decrements time by the amount of seconds and microseconds given by `t`. Time is automatically normalized after the operation.

`PltTime operator + (const PltTime &t1, const PltTime &t2)`
: Adds the two `PltTime` objects `t1` and `t2`, and returns the sum as a new time object. Time is automatically normalized after the operation.

`PltTime operator - (const PltTime &t1, const PltTime &t2)`
: Substracts the `PltTime` object `t2` from `t1`, and returns the difference as a new time object. Time is automatically normalized after the operation.

`bool operator == (const PltTime &t1, const PltTime &t2)`
: Compares two time stamps and returns `true`, if `t1` is equal to `t2`.

`bool operator != (const PltTime &t1, const PltTime &t2)`
: Compares two time stamps and returns `true`, if `t1` is not equal to `t2`.

`bool operator < (const PltTime &t1, const PltTime &t2)`
Compares two time stamps and returns `true`, if `t1` is earlier than `t2`.

`bool operator > (const PltTime &t1, const PltTime &t2)`
Compares two time stamps and returns `true`, if `t1` is later than `t2`.

`bool operator <= (const PltTime &t1, const PltTime &t2)`
Compares two time stamps and returns `true`, if `t1` is earlier than or equal to `t2`.

`bool operator >= (const PltTime &t1, const PltTime &t2)`
Compares two time stamps and returns `true`, if `t1` is later than or equal to `t2`.

`void normalize()`
After you have messed up an `PltTime` object, you should call this method to normalize the microseconds, such that they are always in the range of 0...999,999.

3 libks

The "libks" part of the C++ communication library is build upon the basic "libplt" library part and contains commonly used objects within both KS servers and clients. In particular, some object classes of the "libks" extend classes from the underlaying "libplt" such that these classes are now "streamable": you can serialize and deserialize instances of these classes into/from XDR streams. The XDR streams are part of ONC/RPC and are used to exchange data with clients. The XDR level is responsible for the presentation layer, that is, it twists the bits and bytes in order to cope with different hardware architectures.

The identifiers used throughout the "libks" part of the communication library share a common "Ks" prefix to make them distinguishable from identifiers from the "libplt".

3.1 XDR streamability

All objects that are needed for the communication between KS servers and clients have the ability to stream themselves into XDR streams. This way, the programmer using the ACPLT/KS communication library need not to know how the communication itself is actually carried out on the lower levels. Instead the programmer can focus on how the higher functions, like reading and writing variables, have to be carried out. Thus, the communication library hides all the low-level details of the XDR presentation layer.

You can normally skip this chapter and continue reading with the next chapter 3.2 ("Properties"). The following subsections explain the mechanism for streaming objects in detail, which you only need to be familiar with, if you intend to write your own streamable objects. One application of such streamable objects could be to make objects persistent by writing them to a disk file and restoring them at the next application start. Using the ONC/XDR layer then has the advantage that such objects could be even restored on another operating system and /or hardware platform.

3.1.1 Class KsXdrAble

The ability to serialize or deserialize objects into/from XDR streams is provided through the interface of the abstract base class `KsXdrAble`. Following is this XDR public interface:

```
class KsXdrAble {
public:
    virtual bool xdrEncode(XDR *) const = 0;
    virtual bool xdrDecode(XDR *) = 0;
    // static KsXdrAble * xdrNew(XDR *);
};
```

`virtual bool xdrEncode(XDR *xdr) const`

Serializes an object – that is, writes the contents of the object into a XDR stream. If this method succeeds then it returns `true`. If an error occours – like a "dead" XDR stream or a failure to allocate the memory needed – then `xdrEncode` returns `false`.

`virtual bool xdrDecode(XDR *xdr)`

Deserializes an object – that is, reads the new contents of the object from a XDR stream. If there is no error, then this method returns `true`. Before deserializing the new contents, the old contents of the object are destroyed. If an error occours, then the object's contents may be left in the state they were when the error occoured. A class may also decide to destroy the contents if an error occours. Nevertheless, it is garantueed that the object then can be destroyed without causing a memory leak, or that the object can be reused for deserializing.

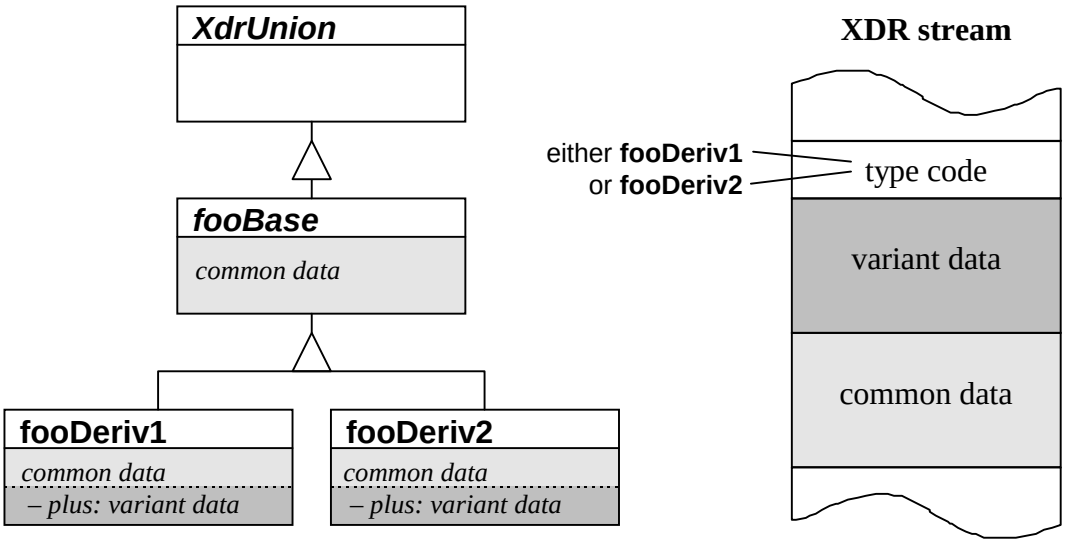
`static KsXdrAble * xdrNew(XDR *xdr)`

This is a "factory method" for constructing a new object from a XDR stream. As such a method can't be defined to be "virtual", it is here just mentioned. Derived classes implement appropriate factory methods to create new objects just out of XDR streams.

In addition to this interface, some derived classes also implement constructors, which construct objects from XDR streams. See the descriptions of the appropriate classes for more information.

3.1.2 Class XdrUnion

Within ACPLT/KS, many service requests and replies work with data structures that vary in dependance of the context. For example, the type of a communication variable controls the structure of its value. While such varying data structures are commonly represented within the ONC/RPC layer as C unions, the communications library uses the polymorphous features of the C++ language. The ability of objects to work directly with XDR streams then avoids unnecessary copying of data into C unions when serializing and deserializing objects.



The class `KsXdrUnion` extends the public interface of the class `KsXdrAble` by the ability to preserve polymorphous properties while exchanging objects between KS servers and clients over the wire. In addition the union class establishes a protected interface that must be (partly) implemented by derived classes.

```
class KsXdrUnion : public KsXdrAble {
public:
    virtual bool xdrEncode(XDR *) const;
```

```
    virtual bool xdrDecode(XDR *);
    // static KsXdrUnion * xdrNew(XDR *);
    virtual enum_t xdrTypeCode() const = 0;
protected:
    virtual bool xdrEncodeCommon(XDR *) const;
    virtual bool xdrDecodeCommon(XDR *);
    virtual bool xdrEncodeVariant(XDR *) const = 0;
    virtual bool xdrDecodeVariant(XDR *) = 0;
};
```

`virtual enum_t xdrTypeCode() const`

Returns the discriminive value which is needed to select the type of polymorphous object to be streamed using a XDR stream. This method must be supplied by derived classes. It usually returns the disciminive value used in the appropriate (RPC) protocol specification.

`virtual bool xdrEncodeCommon(XDR *xdr) const`

Serializes the **common** part of the object's contents into the XDR stream `xdr`. The default implementation of `xdrEncodeCommon()`does nothing, and always returns `true`. If a base class (like `fooBase` in the figure above) has common data which must be streamed, then it must provide its own `xdrEncodeCommon()` method which knows how to stream that common data. As with all "encode" methods, `xdrEncodeCommon()`should return `false`, if it should fail for any reason.

`virtual bool xdrDecodeCommon(XDR *xdr)`

Deserializes the **common** part of the object's contents from the XDR stream `xdr`. The default implementation of `xdrDecodeCommon()`does nothing, and always returns `true`. If a base class (like `fooBase` in the figure above) has common data which must be streamed, then it must provide its own `xdrDecodeCommon()` method which knows how to read that common data from the XDR stream. As with all "decode" methods, `xdrDecodeCommon()`should return `false`, if it should fail for any reason

`virtual bool xdrEncodeVariant(XDR *xdr) const = 0`

Serializes the **variant** part of the object's contents into the XDR stream `xdr`. Any derived class (like `fooDeriv1`, but not `fooBase`) must provide an implementation for this method. The method should return `true`, if it succeeds.

`virtual bool xdrDecodeVariant(XDR *xdr) = 0`

Deserializes the **variant** part of the object's contents from the XDR stream `xdr`. Any derived class (like `fooDeriv1`, but not `fooBase`) must provide an implementation for this method. The method should return `true`, if it succeeds.

For a description of the remaining operations please refer to the interface description of the abstract base class `KsXdrAble`. Note, that you must not supply an implementation for `xdrEncode()` or `xdrDecode()` in any subclass of `KsXdrUnion`.

To implement a polymorphous object class, you can follow the steps described below. As an example, we will implement a streamable base class `fooBase` together with two derived classes called `fooDeriv1` and `fooDeriv2` (see figure above).

First, we declare the common base class `fooBase`, which inherits the public XDR interface of the class `KsXdrUnion`. Fortunately, the macro `KS_DECL_XDRUNION` takes the burden to create the public interface of `fooBase`.

```
class fooBase : public KsXdrUnion {
public:
    // Declare any common data (member variables) here.

    KS_DECL_XDRUNION(fooBase);
```

```
};
```

If you need to stream an object from C layer functions, or from within other XDR streaming functions, you should declare the external C function `xdr_fooBase()` in your header file and implement this function using the macro `KS_IMPL_XDR_C(`*classname*`)`. But if you don't need this C layer stream function, then you can skip this step completely.

```
extern "C" bool_t xdr_fooBase(XDR *, fooBase **);
```

Then we declare the derived classes. Below is an example for the class `fooDeriv1`:

```
class fooDeriv1 : public fooBase {
public:
    fooDeriv1();
virtual enum_t xdrTypeCode() const { return 1; }

protected:
    virtual bool xdrEncodeVariant(XDR *) const;
    virtual bool xdrDecodeVariant(XDR *);

private:
    friend bool fooBase::xdrNew(XDR *);
    fooDeriv1(XDR *, bool &);
};
```

The class definition of `fooDeriv2` looks almost the same, although most notably the `xdrTypeCode()` method must return a different type code.

Now, let's proceed with the implementation part. First, we need to implement the base class `fooBase`. Fortunately, three macros make this implementation really painless. Between the macros `KS_BEGIN_IMPL_XDRUNION(`*class name*`)` and `KS_END_IMPL_XDRUNION` you list all derived classes. In our example this will be the derived classes `fooDeriv1` and `fooDeriv2`. Behind the scenes, these macros generate code for the `fooBase::xdrNew()`.

```
    KS_BEGIN_IMPL_XDRUNION(fooBase);
    KS_XDR_MAP(fooDeriv1);
    KS_XDR_MAP(fooDeriv2);
    KS_END_IMPL_XDRUNION;
```

Next, we can (optionally) implement the C interface function, which can either serialize or deserialize polymorphous objects of class `fooBase` (or any subclass of). The appropriate macro `KS_IMPL_XDR_C(`*base class name*`)` creates a C function named `xdr_fooBase()`. This C function can be called from other XDR functions and the XDR layer in general.

```
    KS_IMPL_XDR_C(fooBase);
```

So much for the macro magic – a few loose ends are still waiting to be tied up, because not everything can be generated automatically. You still have to supply the implementation for the serialization and deserialization methods `xdrEncodeVariant()` and `xdrDecodeVariant()` of both derived classes `fooDeriv1` and `fooDeriv2`.

In addition, you need to implement the constructors for these classes that constructs new objects from XDR streams. Note, that these constructors behave different for `KsXdrUnion`-derived classes: they only read the **variant** part, for example by relying on `xdrDecodeVariant()` for this task. As a consequence of this, you must at all resist any attempt to use the constructor `fooDeriv1(XDR*, bool&)` from outside the `KsXdrUnion` mechanism. But it is **perfectly okay** to use `fooBase::xdrNew()`, which behaves as expected.

If our example base class `fooBase` would declare any common data which must be streamed, then we would also need to implement the appropriate streaming functions `xdrEncodeCommon()` and `xdrDecodeCommon()` for `fooBase`. In every case, you must not supply implementations for `fooBase::xdrEncode()` or for `fooBase::xdrDecode()`.

3.1.3 Class KsArray

The array class `KsArray` is derived from the class `PltArray`, but in addition it can work with XDR streams: new `KsArray` objects can be constructed from a XDR stream, and arrays can be serialized into XDR streams. In contrast to the `PltArray` class, the elements of a `KsArray` must be streamable, that is the elements must support the interface of the class `KsXdrAble`. Following is the public interface of the class `KsArray`:

```
template <class T>
class KsArray : public PltArray<T>, public KsXdrAble {
public:
    KsArray(size_t size = 0);
    KsArray(size_t size, T * p, enum PltOwnership os);
    KsArray(XDR *, bool & success);
    virtual bool xdrEncode(XDR *) const;
    virtual bool xdrDecode(XDR *);
    static KsArray * xdrNew(XDR *);
};
```

`KsArray(size_t size = 0)`
: Constructs a new array that is fixed in size. If you don't specify a size, then an empty array of size 0 is constructed. Such arrays are only useful if you later copy another array into this one. Otherwise you'll get an array that holds exactly `size` entries. You then can access the individual elements using the familiar array operator "`[]`".

`KsArray(size_t size, T *p, enum PltOwnership os)`
: Constructs an array object with a fixed number of entries as indicated by `size` using an preallocated array you already got your hands on. The parameter `p` then points to the preallocated array, which has an ownership of `os`. The ownership `os` can be either `PltOsArrayNew`, `PltOsMalloc` or `PltOsUnmanaged`.

`KsArray(XDR *xdr, bool & success)`
: Constructs a XDR streamable array with data from the XDR stream `xdr`. If this method succeeds, then the reference parameter `success` is set to `true`. If an error occours, while the array is constructed from the XDR stream, `success` is set to `false`.

`virtual bool xdrEncode(XDR *xdr) const`
: Serializes the array's contents into the stream `xdr` and returns `true` if it succeeds.

`virtual bool xdrDecode(XDR *xdr)`
: Deserializes an array from the stream `xdr` and returns `true` if it succeeds.

`static KsArray *xdrNew(XDR *xdr)`
: Constructs a new `KsArray` from the stream `xdr`. If it succeeds, then a pointer to the new object is returned. If an error occours while deserializing the array, the new array object is immediately destroyed and `xdrNew` returns a `0` pointer.

3.1.4 Class KsPtrHandle

The class `KsPtrHandle` behaves like its `PltPtrHandle` cousin from the "libplt" – but in addition, a `KsPtrHandle` features the public interface of the class `KsXdrAble` and therefore handles streaming. If you stream a handle, then the handle will not stream itself (which would make no sense as handles act like pointers) but stream the object it points to. In consequence, the

objects, KsPtrHandles manages must support the public interface of KsXdrAble too. Below is the public interface of the class KsPtrHandle:

```
template<class T>
class KsPtrHandle : public PltPtrHandle<T>, public KsXdrAble {
public:
    KsPtrHandle();
    KsPtrHandle(T *p, enum KsOwnership);
    KsPtrHandle(const KsPtrHandle &);
    KsPtrHandle(XDR *, bool & success);
    bool xdrEncode(XDR *) const;
    bool xdrDecode(XDR *);
    static KsPtrHandle * xdrNew(XDR *);
};
```

`static KsPtrHandle * xdrNew(XDR *)`
: Constructs a new KsPtrHandle and binds to it an object of type T, which in turn is constructed from the stream xdr. If xdrNew succeeds, then a pointer to the new object is returned. If an error occours while deserializing the object of type T, the new handle and the object it should be bind to are immediately destroyed, and xdrNew returns a 0 pointer.

For a description of the remaining functions please refer to the base classes PltPtrHandle and KsXdrAble.

3.1.5 Class KsList

The string class KsList is derived from the PltList class, but in addition it can work with XDR streams: new KsList objects can be constructed from a XDR stream, and lists can be serialized into XDR streams. In contrast to the PltList class, the elements of a KsList must be streamable, that is the elements must support the interface of the class KsXdrAble. Following is the public interface of the class KsList:

```
template <class T>
class KsList : public PltList<T>, public KsXdrAble {
public:
    KsList();
    KsList(XDR *, bool & success);
    virtual bool xdrEncode(XDR *) const;
    virtual bool xdrDecode(XDR *);
    static KsList * xdrNew(XDR *);
};
```

`KsList()`
: Constructs a new list object.

`KsList(XDR *xdr, bool & success)`
: Constructs a XDR streamable list with data from the XDR stream xdr. If this method succeeds, then the reference parameter success is set to true. If an error occours, while the string is constructed from the XDR stream, success is set to false.

`virtual bool xdrEncode(XDR *xdr) const`
: Serializes the list's contents into the stream xdr and returns true, if it succeeds.

`virtual bool xdrDecode(XDR *xdr) const`
: Deserializes the list's contents from the stream xdr and returns true, if it succeeds.

`static KsList * xdrNew(XDR *xdr)`
: Constructs a new `KsList` from the stream `xdr`. If it succeeds, then a pointer to the new object is returned. If an error occours while deserializing the list, the new list object is immediately destroyed, and `xdrNew` returns a `0` pointer.

3.1.6 Class KsString

The string class `KsString` is derived from the `PltString` class, but in addition it can work with XDR streams: new `KsString` objects can be constructed from a XDR stream, and strings can be serialized into XDR streams. Following is the public interface of the class `KsString`:

```
class KsString : public PltString, public KsXdrAble {
public:
    KsString();
    KsString(const char *p, size_t len = PLT_SIZE_T_MAX);
    KsString(const PltString &);
    KsString(XDR *, bool & success);
    virtual bool xdrEncode(XDR *) const;
    virtual bool xdrDecode(XDR *);
    static KsString * xdrNew(XDR *);
};
```

`KsString()`
: Default constructor for a XDR streamable string.

`KsString(const char *p, size_t len = PLT_SIZE_T_MAX)`
: Constructs a XDR streamable string from a C string, which starts at `p` and is at most `len` characters long.

`KsString(const PltString &)`
: Constructs a XDR streamable `KsString` string from another `PltString` or `KsString`.

`KsString(XDR *xdr, bool &success)`
: Constructs a XDR streamable string with data from the XDR stream `xdr`. If this method succeeds, then the reference parameter `success` is set to `true`. If there is an error, while the string is constructed from the XDR stream, `success` is set to `false`.

`virtual bool xdrEncode(XDR *xdr) const`
: Serializes the string's contents into the stream `xdr` and returns `true` if it succeeds.

`virtual bool xdrDecode(XDR *xdr)`
: Deserializes a string from the stream `xdr` and returns `true` if it succeeds.

`static KsString *xdrNew(XDR *xdr)`
: Constructs a new `KsString` from the stream `xdr`. If this method succeeds, then the new object is returned. If an error occours while deserializing the string, the new string object is immediately destroyed and `xdrNew` returns a `0` pointer.

3.1.7 Class KsTime

The class `KsTime` is the class `PltTime` extended by the XDR interface:

```
class KsTime : public PltTime, public KsXdrAble {
public:
    KsTime(long sec = 0L, long usec = 0L);
    KsTime(const PltTime &r)
```

```
    KsTime(XDR * xdr, bool & success);
    virtual bool xdrEncode(XDR * xdr) const;
    virtual bool xdrDecode(XDR * xdr);
};
```

`KsTime(long sec = 0L, long usec = 0L)`
Constructs a time object and sets its timestamp to the time given in `sec` and `usec`.

`KsTime(const PltTime &r)`
Constructs a streamable time object from a (non-streamable) time object.

`KsTime(XDR * xdr, bool & success)`
Constructs a time object from the XDR stream `xdr`. If the constructor succeeds, then it returns `true` in the reference parameter `success`.

`virtual bool xdrEncode(XDR * xdr) const`
Serializes the time object into the XDR stream `xdr` and returns `true` if it succeeds.

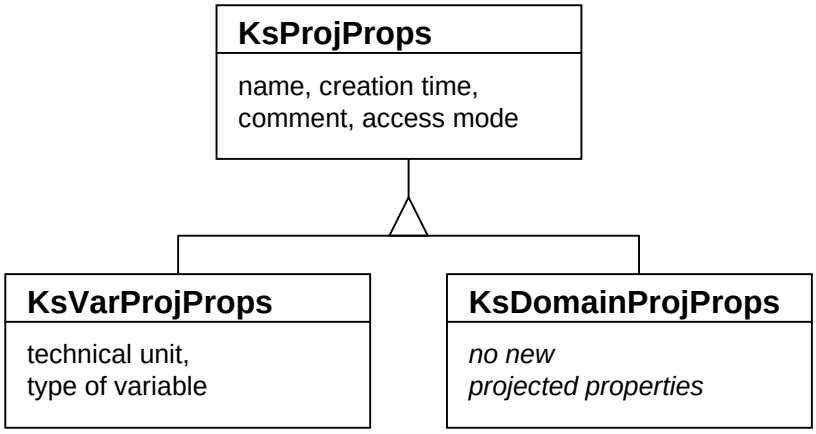
`virtual bool xdrDecode(XDR * xdr)`
Deserializes the time object from the XDR stream `xdr` and returns `true` if it succeeds.

3.2 Properties

ACPLT/KS uses communication objects, which can be either domains or variables, to represent and structure process information. Each communication object owns so-called "projected properties" as well as "current properties".

3.2.1 Projected Properties

The projected properties of ACPLT/KS communication objects are represented by the class hierarchy starting at the root class `KsProjProps` (see figure below). In this hierarchy, the class `KsProjProps` contains such projected properties, which are common to all communication objects of ACPLT/KS. The derived classes `KsVarProjProps` and `KsDomainProjProps` then contain additional properties that are specific to either variables or domains.



Following is the public interface of the projected properties class `KsProjProps`:

```
class KsProjProps : public KsXdrUnion {
public:
    KsProjProps();
    KsProjProps(const KsProjProps &);
    KsProjProps(const KsString &ident, const KsTime &ct,
                const KsString &co, KS_ACCESS am);
    virtual ~KsProjProps() {}
```

```
    KsProjProps &operator = (const KsProjProps &);

    virtual enum_t xdrTypeCode() const = 0;

    KsString identifier;
    KsTime creation_time;
    KsString comment;
    KS_ACCESS_MODE access_mode;
};
typedef KsHandle<KsProjProps> KsProjPropsHandle;
```

`KsProjProps()`
: Constructs an object representing the (generic) projected properties of a communication object.

`KsProjProps(const KsProjProps &)`
: Constructs an object representing the (generic) projected properties from another projected properties object.

`KsProjProps(const KsString &ident, const KsTime &ct, const KsString &co, KS_ACCESS am)`
: Constructs a new object representing the projected properties with the given values for the various projected properties.

`virtual ~KsProjProps()`
: Destructs an object representing the projected properties.

`KsProjProps &operator = (const KsProjProps &)`
: Copies an object containing projected properties into another, thereby making sure that the old information in the object on the left-hand-side of the expression is graciously de-allocated, so no memory leaks can occour.

`KsString identifier`
: The name of the communication object this projected property is associated with. This is **not** the full path.

`KsTime creation_time`
: Creation time of the corresponding communication object or the object within a DCS, etc. the communication object is a representation of.

`KsString comment`
: An optional comment describing the communication object.

`KS_ACCESS_MODE access_mode`
: Access mode of the corresponding communication object.

Communication objects of type "variable" have projected properties which are always represented by the subclass `KsVarProjProps`. Following is the public interface of this class:

```
class KsVarProjProps : public KsProjProps {
public:
    KsVarProjProps();
    KsVarProjProps(const KsVarProjProps &);
    KsVarProjProps(const KsString &ident, const KsTime &ct,
                   const KsString &co, KS_ACCESS am,
                   const KsString &tu, KS_VAR_TYPE tp);
    ~KsVarProjProps();
```

```
    KsVarProjProps &operator = (const KsVarProjProps &);
    virtual enum_t xdrTypeCode() const;

    KsString tech_unit;
    KS_VAR_TYPE type;
};
```

`virtual enum_t xdrTypeCode() const`
: Identifies any instance of the class `KsVarProjProps` as the projected properties of a variable communication object. This method always returns `KS_OT_VARIABLE`.

`KsString tech_unit`
: A string indicating the technical unit of the value of the associated variable object.

`KS_VAR_TYPE type`
: Type (integer, unsigned integer, string, etc.) of the associated variable object.

For a description of the constructors and other remaining public member functions please refer to the description of the superclass `KsProjProps`.

Communication objects of type "domain" do not introduce new projected properties but are nevertheless represented by their own subclass `KsDomainProjProps`. This way you can use the RTTI (run-time type information) mechanism to check whether the projected properties returned by a `GetPP` service request belong to variables or domains.

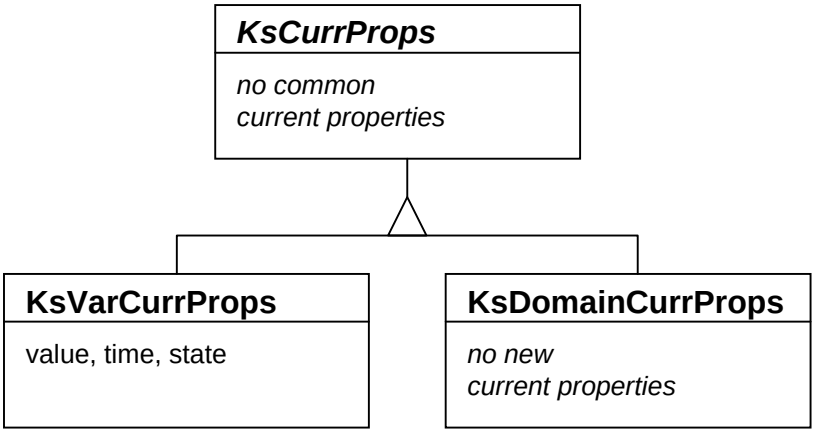
```
class KsDomainProjProps : public KsProjProps {
public:
    KsDomainProjProps();
    KsDomainProjProps(const KsDomainProjProps &);
    KsDomainProjProps(const KsString &ident, const KsTime &ct,
                      const KsString &co, KS_ACCESS am);
    ~KsDomainProjProps();

    KsDomainProjProps &operator = (const KsDomainProjProps &);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
: Identifies any instance of the class `KsDomainProjProps` as the projected properties of a domain communication object. This method always returns `KS_OT_DOMAIN`.

For a description of the constructors and other remaining public member functions please refer to the description of the superclass `KsProjProps`.

3.2.2 Current Properties

In addition to the projected properties, a ACPLT/KS communication object also has so-called "current properties". Again, these properties are represented by a class hierarchy starting with the root class `KsCurrProps` (see figure below). In this hierarchy, the class `KsCurrProps` contains all those current properties, which are common to all communication objects of the ACPLT/KS protocol. The derived classes `KsVarCurrProps` and `KsDomainCurrProps` then contain the current properties that are specific to either variables or domains.



Following is the public interface of the current properties class `KsCurrProps`. It introduces no new member variables but serves as the root for the class hierarchy of current properties objects. The interface of the class `KsCurrProps` is very similiar to the one of the class `KsProjProps`:

```
class KsCurrProps : public KsXdrUnion {
public:
    virtual enum_t xdrTypeCode() const = 0;
};
typedef KsPtrHandle<KsCurrProps> KsCurrPropsHandle;
```

Communication objects of type "variable" have current properties which are always represented by the subclass `KsVarCurrProps`. Following is the public interface of this class:

```
class KsVarCurrProps : public KsCurrProps {
public:
    KsVarCurrProps();
    KsVarCurrProps(KsValueHandle v);
    KsVarCurrProps(const KsVarCurrProps &);
    KsVarCurrProps(KsValueHandle v, const KsTime &t, KS_STATE s);
    ~KsVarCurrProps();

    virtual enum_t xdrTypeCode() const;

    KsValueHandle value;
    KsTime time;
    KS_STATE state;
};

typedef KsPtrHandle<KsValue> KsValueHandle;
```

`virtual enum_t xdrTypeCode() const`
: Identifies any instance of the class `KsVarCurrProps` as the current properties of a variable communication object. This method always returns `KS_OT_VARIABLE`.

`KsValueHandle value`
: Handle of the value object which stores the current value of the corresponding variable object.

`KsTime time`
: Time stamp of the value.

`KS_STATE state`
: Validation information/state of the value.

For a description of the constructors and other remaining public member functions please refer to the description of the superclass `KsCurrProps`.

Communication objects of type "domain" do not introduce new current properties but are nevertheless represented by their own subclass `KsDomainCurrProps`:

```
class KsDomainCurrProps : public KsCurrProps {
public:
    KsDomainCurrProps();
    ~KsDomainCurrProps();
    virtual enum_t xdrTypeCode() const;
};
```

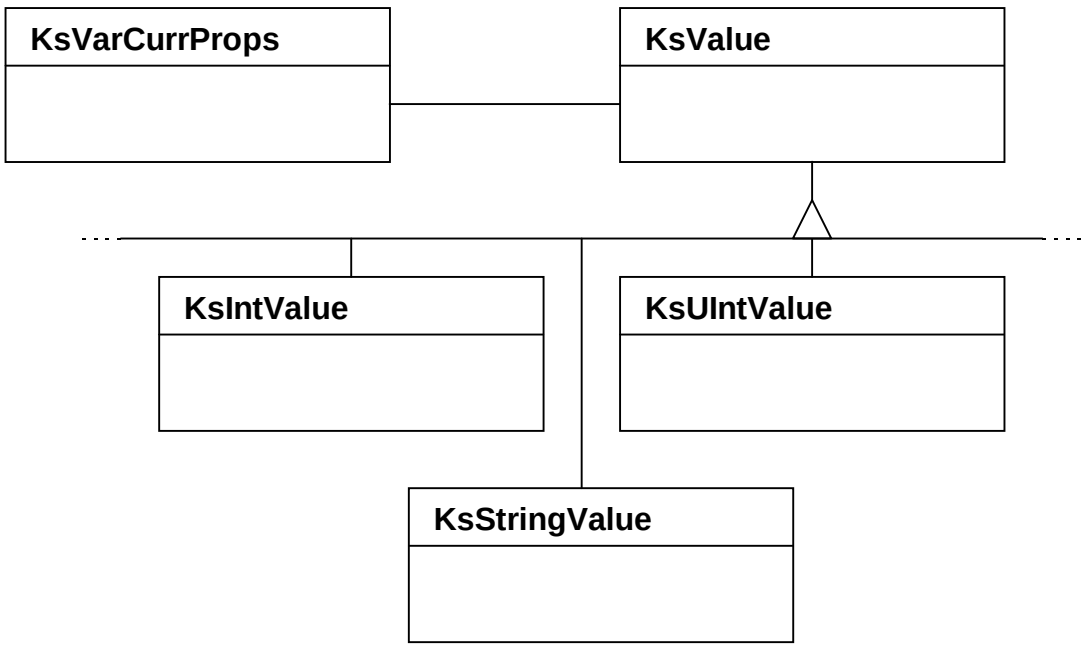
`virtual enum_t xdrTypeCode() const`

Identifies any instance of the class `KsDomainCurrProps` as the current properties of a domain communication object. This method always returns `KS_OT_DOMAIN`.

For a description of the constructors and other remaining public member functions please refer to the description of the superclass `KsCurrProps`.

3.3 Values

The class `KsValue` is the base class of value objects. It is closely related to the data structure `KS_VAR_VALUE`, which is defined in the ACPLT/KS communication protocol specification `ks.x`. Value objects of class `KsValue` or any subclass of are used by instances of the class `KsVarCurrProps`, which in turn represents the current properties of a communication object of type "variable". The various data types – like integers, floating point numbers, and strings – are represented by subclasses of the class `KsValue` (see figure below).



Following is the public interface of the abstract base class `KsValue`:

```
class KsValue : public KsXdrUnion {
public:
    static KsValue *xdrNew(XDR *xdr);
    virtual enum_t xdrTypeCode() const = 0;
};
```

```
typedef KsPtrHandle<KsValue> KsValueHandle;
```

This interface is similiar to the interface of the superclass `KsXdrUnion`, so please refer to this superclass for a description of what the member functions do. Especially, it inherits the streaming ability through the methods `xdrDecode()` and `xdrEncode()`.

3.3.1 Class KsIntValue

Following is the public interface of the value class `KsIntValue`, which represents a single 32 bit wide integer value:

```
class KsIntValue : public KsValue {
public:
    KsIntValue(long v = 0L);
    void getInt(long& v) const;
    void setInt(long v);
    operator long () const;
    KsIntValue & operator = (long v);
    virtual enum_t xdrTypeCode() const;
};
```

`KsIntValue(long v = 0L)`
: Constructs a `KsIntValue` object with the value specified by `v`.

`void getInt(long& v) const`
: Returns the current value of the `KsIntValue` object through the parameter `v`.

`void setInt(long v)`
: Sets the new value of the `KsIntValue` object to be `v`.

`operator long () const`
: Cast operator that returns the current value of the `KsIntValue` object.

`KsIntValue & operator = (long v)`
: Assigns the new value `v` to the `KsIntValue` object.

`virtual enum_t xdrTypeCode() const`
: Returns the type code of `KsIntValue` objects, which is always `KS_VT_INT`.

3.3.2 Class KsUIntValue

Following is the public interface of the value class `KsUIntValue`, which represents a single 32 bit wide unsigned integer value:

```
class KsUIntValue : public KsValue {
public:
    KsUIntValue(u_long v = 0UL);
    void getUInt(u_long& v) const;
    void setUInt(u_long v);
    operator u_long () const;
    KsUIntValue & operator = (u_long v);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
: Returns the type code of `KsUIntValue` objects, which is always `KS_VT_UINT`.

For a description of the remaining member functions please refer to the description of the value class `KsIntValue`. The access methods/operators of the class `KsUIntValue` are similiar to those of the class `KsIntValue`.

3.3.3 Class KsBoolValue

Following in the public interface of the class `KsBoolValue`, which represents a bool value:

```
class KsBoolValue : public KsValue {
public:
    KsBoolValue(bool = false);

    operator bool () const;
    KsBoolValue & operator = (bool b);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsBoolValue` objects, which is always `KS_VT_BOOL`.

For a description of the remaining member functions please refer to the description of the value class `KsIntValue`. The access methods/operators of the class `KsBoolValue` are similiar to those of the class `KsIntValue`.

3.3.4 Class KsSingleValue

Following is the public interface of the value class `KsSingleValue`, which represents a single 32 bit wide IEEE conforming floating point value:

```
class KsSingleValue : public KsValue {
public:
    KsSingleValue(float v = 0.0);

    void getSingle(float & v) const;
    void setSingle(float v);
    operator float () const;
    KsSingleValue & operator = (float v);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsSingleValue` objects, which is always `KS_VT_SINGLE`.

For a description of the remaining member functions please refer to the description of the value class `KsIntValue`. The access methods/operators of the class `KsSingleValue` are similiar to those of the class `KsIntValue`.

3.3.5 Class KsDoubleValue

Following is the public interface of the value class `KsDoubleValue`, which represents a single 64 bit wide IEEE conforming floating point value:

```
class KsDoubleValue : public KsValue {
public:
```

```
    KsDoubleValue(double v = 0.0);
    void getDouble(double &v) const;
    void setDouble(double v);
    operator double () const;
    KsDoubleValue & operator = (double v);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsDoubleValue` objects, which is always `KS_VT_DOUBLE`.

For a description of the remaining member functions please refer to the description of the value class `KsIntValue`. The access methods/operators of the class `KsDoubleValue` are similiar to those of the class `KsIntValue`.

3.3.6 Class KsStringValue

Following is the public interface of the value class `KsStringValue`, which represents a string of characters:

```
class KsStringValue : public KsString, public KsValue {
public:
    KsStringValue();
    KsStringValue(const char *);
    KsStringValue & operator = (const char *);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsStringValue` objects, which is always `KS_VT_STRING`.

For a description of the remaining member functions please refer to the description of the value class `KsIntValue` and the streamable string class `KsString`. The access methods/operators of the class `KsStringValue` are inherited from the class `KsString`.

3.3.7 Class KsTimeValue

Following is the public interface of the value class `KsTimeValue`, which represents a time stamp measured in Universal Time Coordinates (UTC):

```
class KsTimeValue : public KsTime, public KsValue {
public:
    KsTimeValue(long sec = 0L, long usec = 0L);
    KsTimeValue(const KsTime &);
    KsTimeValue & operator = (const KsTime &);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsTimeValue` objects, which is always `KS_VT_TIME`.

For a description of the remaining member functions please refer to the description of the value class `KsIntValue`. The access methods/operators of the class `KsTimeValue` are similiar to those of the class `KsTime`.

3.3.8 Class KsByteVecValue

Following is the public interface of the value class `KsByteVecValue`, which represents an array of bytes ("opaque" data type):

```
class KsByteVecValue : public KsArray<char>, public KsValue {
    KsByteVecValue(size_t size = 0);
    KsByteVecValue(size_t size, char *p, PltOwnership os);
    virtual enum_t xdrTypeCode() const;
};
```

`KsByteVecValue(size_t size = 0)`
: Constructs a new byte array (vector) with a default length of zero.

`KsByteVecValue(size_t size, char *p, PltOwnership os)`
: Constructs a byte array from data addressed by `p` with a length of `size` and an ownership of `os`.

`virtual enum_t xdrTypeCode() const`
: Returns the type code of `KsByteVecValue` objects, which is always `KS_VT_BYTE_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue`. The access methods/operators of the class `KsByteVecValue` are similiar to those of the class `KsArray<char>`.

3.3.9 Class KsIntVecValue

Following is the public interface of the value class `KsIntVecValue`, which represents an array of signed integers:

```
class KsIntVecValue : public KsArray<long>, public KsValue {
    KsIntVecValue(size_t size = 0);
    KsIntVecValue(size_t size, long *p, PltOwnership os);
    virtual enum_t xdrTypeCode() const;
};
```

`KsIntVecValue(size_t size = 0)`
: Constructs a new array (vector) of signed integers with a default length of zero.

`KsIntVecValue(size_t size, char *p, PltOwnership os)`
: Constructs a array of signed integers from data addressed by `p` with a length of `size` and an ownership of `os`.

`virtual enum_t xdrTypeCode() const`
: Returns the type code of `KsIntVecValue` objects, which is always `KS_VT_INT_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue`. The access methods/operators of the class `KsIntVecValue` are similiar to those of the class `KsArray<long>`.

3.3.10 Class KsUIntVecValue

Following is the public interface of the value class `KsUIntVecValue`, which represents an array of unsigned integers:

```
class KsUIntVecValue : public KsArray<u_long>, public KsValue {
    KsUIntVecValue(size_t size = 0);
    KsUIntVecValue(size_t size, u_long *p, PltOwnership os);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsUIntVecValue` objects, which is always `KS_VT_UINT_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue` as well as to the similiar class `KsIntVecValue`. The access methods/operators of the class `KsUIntVecValue` are similiar to those of the class `KsArray<u_long>`.

3.3.11 Class KsBoolVecValue

Following is the public interface of the value class `KsBoolVecValue`, which represents an array of booleans:

```
class KsBoolVecValue : public KsArray<bool>, public KsValue {
public:
    KsBoolVecValue(size_t size = 0);
    KsBoolVecValue(size_t size, bool *p, PltOwnership os);
    enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsBoolVecValue` objects, which is always `KS_VT_BOOL_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue` as well as to the similiar class `KsIntVecValue`. The access methods/operators of the class `KsBoolVecValue` are similiar to those of the class `KsArray<bool>`.

3.3.12 Class KsSingleVecValue

Following is the public interface of the value class `KsSingleVecValue`, which represents an array of single precision floating point penguins:

```
class KsSingleVecValue : public KsArray<float>, public KsValue {
    KsSingleVecValue(size_t size = 0);
    KsSingleVecValue(size_t size, float *p, PltOwnership os);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsSingleVecValue` objects, which is always `KS_VT_SINGLE-_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue` as well as to the similiar class `KsIntVecValue`. The access methods/operators of the class `KsSingleVecValue` are similiar to those of the class `KsArray<float>`.

3.3.13 Class KsDoubleVecValue

Following is the public interface of the value class `KsDoubleVecValue`, which represents an array of double precision floating point penguins:

```
class KsDoubleVecValue : public KsArray<double>, public KsValue {
    KsDoubleVecValue(size_t size = 0);
    KsDoubleVecValue(size_t size, double *p, PltOwnership os);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsDoubleVecValue` objects, which is always `KS_VT_DOUBLE_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue` as well as to the similiar class `KsIntVecValue`. The access methods/operators of the class `KsDoubleVecValue` are similiar to those of the class `KsArray<double>`.

3.3.14 Class KsStringVecValue

Following is the public interface of the value class `KsStringVecValue`, which represents an array of strings:

```
class KsStringVecValue : public KsArray<KsString>, public KsValue {
    KsStringVecValue(size_t size = 0);
    KsStringVecValue(size_t size, KsString *p, PltOwnership os);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsStringVecValue` objects, which is always `KS_VT_STRING_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue` as well as to the similiar class `KsIntVecValue`. The access methods/operators of the class `KsStringVecValue` are similiar to those of the class `KsArray<KsString>`.

3.3.15 Class KsTimeVecValue

Following is the public interface of the value class `KsTimeVecValue`, which represents an array of strings:

```
class KsTimeVecValue : public KsArray<KsTime>, public KsValue {
    KsTimeVecValue(size_t size = 0);
    KsTimeVecValue(size_t size, KsTime *p, PltOwnership os);
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsTimeVecValue` objects, which is always `KS_VT_TIME_VEC`.

For a description of the remaining member functions please refer to the description of superclasses `KsArray` and `KsValue` as well as to the similiar class `KsIntVecValue`. The access methods/operators of the class `KsTimeVecValue` are similiar to those of the class `KsArray<KsTime>`.

3.3.16 Class KsVoidValue

Following is the public interface of the value class `KsVoidValue`, which represents a void value ("the great nothing"):

```
class KsVoidValue : public KsValue {
public:
    KsVoidValue();
    virtual enum_t xdrTypeCode() const;
};
```

`virtual enum_t xdrTypeCode() const`
Returns the type code of `KsVoidValue` objects, which is always `KS_VT_VOID`.

For a description of the remaining functions, which are responsible for streaming `KsVoidValue` objects, please refer to the aforementioned value object classes.

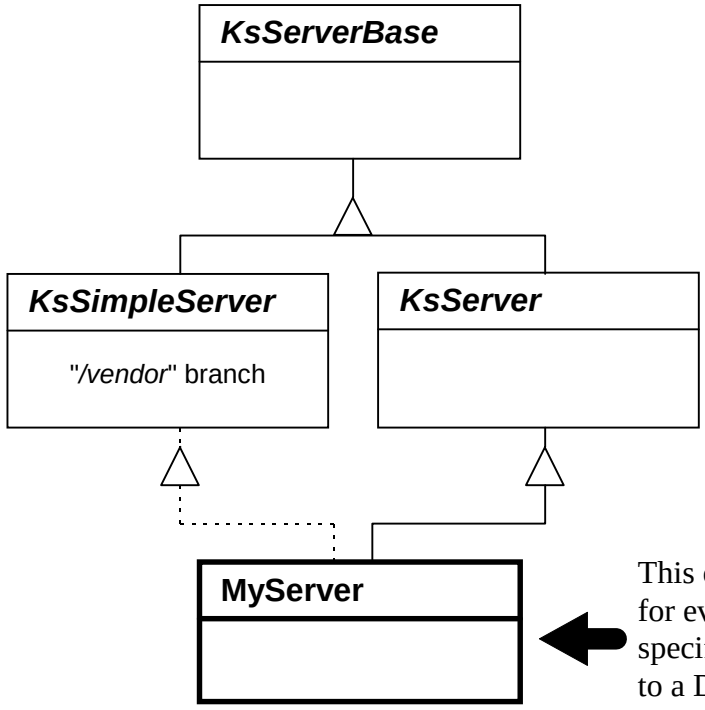
4 libkssvr

The "libkssvr" library part contains object classes which are used solely for KS servers and managers. This part of the C++ communication library sits on top of the more generic libraries "libks" and "libplt".

4.1 The Server Object

4.1.1 Building the Blocks

The abstract base class `KsServerBase` encapsulates the common tasks which are needed by both KS servers and managers. Below this base server class there are two additional abstract server classes – `KsServer` and `KsSimpleServer` – which each implement certain aspects of KS servers.



To implement a KS server, you must derive your server object at least from the abstract class `KsServer` and override some virtual member functions, most notably the "service functions". The class `KsServer` is responsible for the communication (registration) with the KS manager.

In most cases you may want to reuse a common handling of the "`/vendor`" domain branch of the communication objects in a KS server. In this case, you can inherit this behaviour from the abstract class `KsSimpleServer` (your server `MyServer` then makes use of the multiple inheritance). In addition, this class also provides convenience functions to build branches of communication objects which are solely existent within the KS server and do not correspond to anything in an underlaying DCS, simulator, or whatever.

4.1.2 At Your Service

In principle, there are two different ways a KS server serves incomming requests from KS clients. First, if a KS server possesses its own process space, you typically will use something like the following code excerpt. Second, the KS server can be part of another application which periodically gives the KS part a chance to server incomming requests.

But now for our little example. First, let us create an instance of your special server class: in this example of the class `MyServer`. Remember, that you never create an instance of `KsServer` – the C++ compiler won't let you anyway. If something is going wrong, then you should kill the object and immediately try to escape from this place...

```
MyServer *svr = new MyServer("myserver");
if ( !svr->isOk() ) {
    // do whatever you want to bail out...
}
```

Before you start the server, and thus the handling of incomming requests, you may want to register additional A/V modules or change some default settings. Eventually, you start up the server completely (trigger the final initialize).

```
svr->startServer();
```

Then you enter the main service loop of the server, so that the server starts to accept new connections from clients and processes incomming requests. The service loop is inherited, so you don't have to re-invent the wheel.

```
svr->run();
```

When the server is shut down (how this is exactly done depends on the type of operating system), program execution will return. You then should clean up any resources in use (close the connections to a DCS, free memory) by destroying the server object.

```
svr->stopServer();
delete svr;
```

That's it. Thanks to the object orientated technology, you don't need to worry about the RPC communication layer and other goodies, as they are completely hidden inside the server object's implementation.

4.2 Class KsServerBase

The functionality common to KS servers and managers is encapsulated in the abstract base class `KsServerBase`. Below is the public and protected interface of the class `KsServerBase`. The protected interface implements such functions that are used only inside the server object and thus need not to be visible outside the server object.

```
typedef KsAvTicket * (*KsTicketConstructor)(XDR *);

class KsServerBase {
public:
    KsServerBase();
    virtual ~KsServerBase();

    enum { KS_ANYPORT = 0 };

    static KsServerBase &getServerObject() const;

    virtual KsString getServerName() const = 0;
    virtual KsString getServerVersion() const = 0;
    virtual KsString getServerDescription() const = 0;
    virtual KsString getVendorName() const = 0;
    virtual u_short  getProtocolVersion() const;

    virtual bool isOk() const;
    virtual bool isGoingDown() const;

    virtual void startServer();
    virtual void run();
    virtual void downServer();
    virtual void stopServer();
    bool hasPendingEvents() const;

    bool servePendingEvents(KsTime timeout = KsTime());
    bool servePendingEvents(KsTime *pTimeout);

    bool addTimerEvent(KsTimerEvent *event);
    bool removeTimerEvent(KsTimerEvent *event);
    KsTimerEvent *removeNextTimerEvent();
    KsTimerEvent *peekNextTimerEvent() const;

    virtual void getPP(KsAvTicket &ticket,
                       const KsGetPPParams &params,
```

```
                        KsGetPPResult &result);
    virtual void getVar(KsAvTicket &ticket,
                        const KsGetVarParams &params,
                        KsGetVarResult &result);
    virtual void setVar(KsAvTicket &ticket,
                        const KsGetVarParams &params,
                        KsGetVarResult &result);
    virtual void exgData(KsAvTicket &ticket,
                         const KsExgDataParams &params,
                         KsExgDataResult &result);
};
```

`KsServerBase()`
: Constructs a server object. The new server object is registered, so you may refer to it using `getServerObject()`, if you don't have the object pointer ready at hand. You can't create more than one instance of any object subclassed from `KsServerBase`.

`virtual ~KsServerBase()`
: Destroys ("destructs") a server object. In addition, it deregisters the server object, so calls to `getServerObject()` are no longer allowed.

`enum { KS_ANYPORT = 0 }`
: An anonymous enumeration, which is used when a KS server object need not to be bound to a specific port number. In this case, you can simply specify the value `KsServer-Base::KS_ANYPORT` as the `port` argument of the constructors of derived classes.

`static KsServerBase &getServerObject() const`
: Returns a reference to the server object. This static member function allows you to call the server object even if you don't have a pointer to it ready at hand.

`virtual KsString getServerName() const = 0`
: In derived classes, you must overwrite this function to return the name of this particular KS server.

`virtual KsString getServerVersion() const = 0`
: In derived classes, you must overwrite this function to return the server version identification.

`virtual KsString getServerDescription() const = 0`
: In derived classes, you must overwrite this function to return a textual description.

`virtual KsString getVendorName() const = 0`
: In derived classes, you must overwrite this function to return the vendor name.

`virtual u_short getProtocolVersion() const`
: Returns the ACPLT/KS protocol version number, which is currently `1`.

`virtual bool isOk() const`
: Returns `true`, if the server object could be successfully constructed and is now ready to be started up.

`virtual bool isGoingDown() const`
: Returns `true`, if `downServer()` has been called and the server object is about to be shut down.

`virtual void startServer()`
: Starts the server by triggering final initialization steps. After calling the method, the server object is ready to accept incomming connections from KS clients, and to process the requests.

`virtual void run()`
: Starts the main loop of the server. This loop processes incomming requests and timer events and dispatches them appropriately. The `run` method only returns, after you have called `downServer()` or if a hard error occours, which prevents the server object from further processing requests.

`virtual void downServer()`
: Indicates that event processing should end as soon as possible. Due to the implementation of the ONC/RPC library, the main loop can only be left after all currently waiting requests have been served.

`virtual void stopServer()`
: Finally stops and properly shuts down the server object, thus indicating that this server has stopped processing requests (and is now deregistered).

`bool hasPendingEvents() const`
: Checks whether there are timer events or requests from KS clients waiting to be served. If this is the case, `hasPendingEvents()` returns `true`.

`bool servePendingEvents(KsTime timeout = KsTime())`
: Serves all waiting timer events and requests from KS clients. If there are currently neither events nor requests waiting to be served, then `servePendingEvents()` waits at most for the `timeout` span before it returns with `false`. Otherwise this method serves the waiting timer events and requests and then returns `true`.

`bool servePendingEvents(KsTime *pTimeout)`
: Same as above, but with overloaded argument.

`bool addTimerEvent(KsTimerEvent *event)`
: Adds a new timer event to the timer event queue of the server object. The caller is responsible for the event object as the server object only stores a reference to it. If `addTimerEvent()` succeeds, it returns `true`.

  If the time the event should trigger lies in the past when adding the event, the event will trigger the next time the server object checks for pending timer events. This can be either the case when program execution returns to the main loop of the server, or the method `servePendingEvents()` is called.

`bool removeTimerEvent(KsTimerEvent *event)`
: Removes the timer event from the timer event queue of the server. If there is no such timer event currently registered, then this method returns `false`.

`KsTimerEvent *peekNextTimerEvent() const`
: Returns a pointer to the next timer event that will trigger, or `0`, if there is currently no timer event in the timer event queue. The timer event is **not** removed from the timer queue.

`KsTimerEvent *removeNextTimerEvent()`
: Returns the next timer event from the queue and removes it from the queue at the same time. If there is currently no timer event in the timer event queue, then `removeNext-TimerEvent()` returns `0`.

`virtual void getPP(KsAvTicket &ticket, const KsGetPPParams &params, KsGetPPResult &result)`
: Service function for the `GetPP` service. It must be implemented in derived classes.

`virtual void getVar(KsAvTicket &ticket, const KsGetVarParams &params, KsGetVarResult &result)`
: Service function for the `GetVar` service. It must be implemented in derived classes.

```
virtual void setVar(KsAvTicket &ticket, const KsGetVarParams &params,
    KsGetVarResult &result)
```
Service function for the `SetVar` service. It must be implemented in derived classes.

```
virtual void exgData(KsAvTicket &ticket,
    const KsExgDataParams &params, KsExgDataResult &result)
```
Service function for the `ExgData` service. It must be implemented in derived classes.

4.2.1 Class KsServer

"Real" KS servers are always subclassed of the class `KsServer`, which implements special behaviour that is necessary for servers (but not for managers). Because this new behaviour does its duty "behind the scenes", the public interface of the class `KsServer` does not differ very much from that of the superclass `KsServerBase`.

```
class KsServer : virtual public KsServerBase {
public:
    KsServer(u_long ttl, int port = KS_ANYPORT);
};
```

```
KsServer(u_long ttl, int port = KS_ANYPORT)
```
Creates a server object and sets is "time to live", which is told to the KS manager lateron, to the time span in seconds given in `ttl`. The KS server will reregister itself automatically with the KS manager after three quarters of `ttl` have passed by. If you do not specify a specific port number, the KS server will be bound to an arbitrary free TCP/IP port.

The customer-specific behaviour of a KS server is then implemented by overriding the appropriate service functions in a subclass of `KsServer`.

4.2.2 Class KsSimpleServer

The class `KsSimpleServer` implements a default behaviour for creating the "`/vendor`" branch of the communication objects tree. In addition it supplies default methods for querying, reading and setting communication objects. Typically you'll inherit your particular KS server from both `KsServer` and `KsSimpleServer`. Following is the public as well as the protected interface of the class `KsSimpleServer`:

```
class KsSimpleServer : virtual public KsServerBase {
public:
    KsSimpleServer(int port = KS_ANYPORT);
    virtual void getVar(KsAvTicket &ticket,
                        const KsGetVarParams &params,
                        KsGetVarResult &result);
    virtual void setVar(KsAvTicket &ticket,
                        const KsSetVarParams &params,
                        KsSetVarResult &result);
    virtual void getPP(KsAvTicket &ticket,
                       const KsGetPPParams & params,
                       KsGetPPResult & result);

protected:
    KssSimpleDomain _root_domain;

    virtual void getVarItem(KsAvTicket &ticket,
                            const KsPath & path,
```

```
                            KsGetVarItemResult &result);
    virtual void setVarItem(KsAvTicket &ticket,
                            const KsPath & path,
                            const KsCurrPropsHandle & curr_props,
                            KsResult &result);

    bool addCommObject(const KsPath & dompath,
                       const KssCommObjectHandle & ho);

    bool addDomain(const KsPath & dompath,
                   const KsString & id,
                   const KsString & comment = KsString());

    bool addStringVar(const KsPath & dompath,
                      const KsString & id,
                      const KsString & str,
                      const KsString & comment = KsString(),
                      bool lock = true);

    bool initVendorTree();
};
```

`KsSimpleServer(int port = KS_ANYPORT)`
: Constructs an `KsSimpleServer` object and binds it to the port specified in `port` or to any arbitrary port in case you specify `KS_ANYPORT`.

`bool initVendorTree()`
: Builds an initial "`/vendor`" branch containing the following communication objects: `server_time`, `server_name`, `server_version`, `server_description` and `name`. The values of these variable objects are based on the return values of the member functions `getServerName()`, `getServerVersion()`, `getServerDescription()`, and `getVendorName()`. If `initVendorTree()` succeeds, then it'll return `true`.

  You should call `initVendorTree()` from the constructor of your KS server object and set the protected member variable `_is_okay`, if it fails. Otherwise you can carry on and create additional communication objects (not only within the "`/vendor`" branch) using `addDomain()`, `addCommObject()`, and friends.

`bool addCommObject(const KsPath &dompath,`
`const KssCommObjectHandle &ho)`
: Adds a `KssCommObject` a a child of the `KssSimpleDomain` object, whose path is given in `dompath`. If this function succeeds, then it'll return `true`. Be sure to create first a `KssSimpleDomain` object, before you add any children to it through this helper function.

`bool addDomain(const KsPath &dompath, const KsString &id,`
`const KsString &comment = KsString())`
: Convenience function that creates a `KssSimpleDomain` object (of name `id`) and adds it to the child list of the domain object with the path `dompath`. If the convenience function succeeds, then it'll return `true`.

`bool addStringVar(const KsPath &dompath, const KsString &id,`
`const KsString &str, const KsString &comment = KsString(),`
`bool lock = true)`
: Convenience function that creates a `KsSimpleVariable` object (of name `id`), which holds a value of the object class `KsStringValue`. Then variable object then is made a child of the `KssSimpleDomain` object, whose path is given in `dompath`. If this function

succeeds, then it'll return `true`. If the argument `lock` is `true`, then the value of the communication variable object is write protected.

`void getVarItem(KsAvTicket &ticket, const KsPath &path, KsGetVarItemResult &result)`
Helper function for `getVar()` which gets the value of one variable. It can be used by derived KS server classes to implement the basic acces mechanism to variables in addition to the mechanism necessary to fetch values from a DCS, simulator, or whatever.

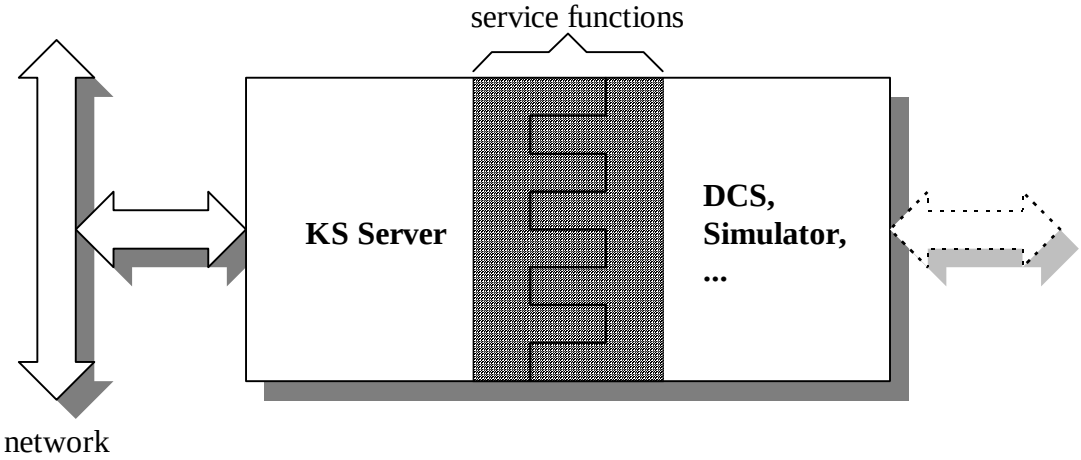
`void setVarItem(KsAvTicket &ticket, const KsPath &path, const KsCurrPropsHandle &curr_props, KsResult &result)`
Helper function for `setVar()` which sets the value of one variable. It can be used by derived KS server classes to implement the basic acces mechanism to variables in addition to the mechanism necessary to send values to a DCS, simulator, or whatever.

`KssSimpleDomain _root_domain`
The root domain of the communication tree. In particular cases, you can attach a "sister domain" to this root domain, which is then responsible for special communication objects belonging (for example) to a simulator.

In addition, the class `KsSimpleServer` provides default implementations for `getVar()`, `setVar()` and `getPP()`, which work on the communication objects created and added through `initVendorTree()` and the other convenience functions.

4.3 Service Functions

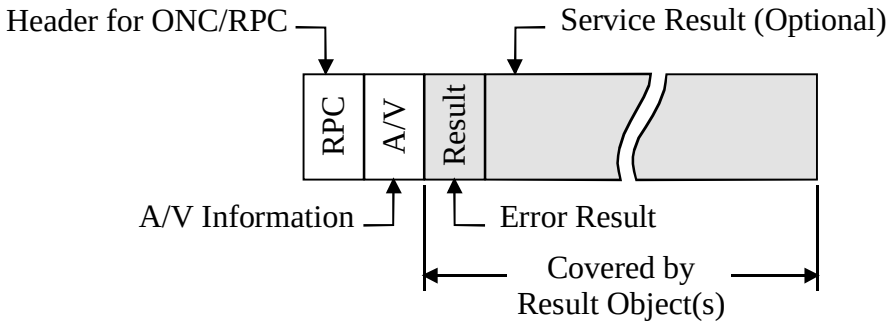
In ACPLT/KS, reading or writing variables, querying servers, etc., is carried out by "services". Each service is responsible for a certain task. Within KS servers, each service is implemented by the programmer using a different "service function". The service functions thus build the bridge between the KS server and the DCS, simulator, or other system to be connected to the network through ACPLT/KS (see figure below).



The concept of "service functions" reliefes the burdens of serializing and deserializing objects from the programmer, and it therefore hides the direct communication with KS clients. Whenever the server object receives a service request from a KS client, it creates a request object from the data sent. In addition, the server creates a result object that will receive the service's answer lateron. Then it calls the appropriate service function, which in turn now consults the request object, communicates with the underlaying DCS, or simulator (or something like that), and fills in the answer into the result object. Upon return from the service function, the server object sends back the answer (the result object) to the KS client.

4.3.1 Service Results

The abstract base class `KsResult` provides a common public interface for such object classes which store the results of calls to service funtions. In addition, result objects are sometimes used recursively – for example – to represent the results of the GetVar service. The result objects then can be easily serialized in a unified manner from the server layer below the service funtions. In particular, all service results are therefore derived from the class `KsResult`. Although the result objects are here discussed here in the context of service functions (which are only used on the server-side), the result objects are used within the "libkscln" library part as well. But they are there not visible at the API level.




```
class KsResult : public KsXdrAble {
public:
    KsResult(KS_RESULT res = -1);
    KsResult(XDR *, bool &);
    bool xdrEncode(XDR *xdr) const;
    bool xdrDecode(XDR *xdr);
    static KsServiceResult * xdrNew(XDR *xdr);

    KS_RESULT result;
};
```

`KsResult(KS_RESULT res = -1)`
: Constructs a result object and initializes it to have no result.

`KsResult(XDR *xdr, bool &success)`
: Constructs a new result object from a XDR stream. If it succeeds, then `success` is set to `true`.

`bool xdrEncode(XDR *xdr) const`
: Sends the (service) result back to the KS client through the XDR stream `xdr`. If the method succeeds, then it returns `true` (as usual). Used on the server-side.

`bool xdrDecode(XDR *xdr)`
: Receives the service result on the client-side. This method is threrefore used on the client-side.

`static KsServiceResult * xdrNew(XDR *xdr)`
: Creates a service result object from the XDR stream `xdr`. If the creation operation fails for some reason (not enough memory, broken RPC connection), then `xdrNew` deletes the object it might just have created and returns a `0` pointer. This method is used on the client-side.

`KS_RESULT result`
: Indicates the outcome of the (service) function. A service function must always set this member variable. If it is set to `KS_ERR_OK`, then the full service reply is send back to the KS client. Otherwise, only `result` is send back to the client, indicating that an error occoured while processing the service function. Derived classes then will stream their additional data only if `result` is `KS_ERR_OK`.

In the following subsections, which describe the various service parameter and reply objects, the XDR streaming methods `xdrEncode()` and `xdrDecode()` are not explicitely listed again. If factory methods are provided, they are shown in the public interfaces but not in the explaining sections below.

4.3.2 The GetPP Service

The GetPP service provides access to the projected properties of the communication objects within a KS server. In addition, the GetPP service can supply information about the childen of a domain object. Following is the function prototype of the GetPP service function:

```
virtual void KsServerBase::getPP(KsAvTicket    &ticket,
                                 KsGetPPParams &params,
                                 KsGetPPResult &result);
```

The incomming request contains information about which objects to query. In addition, the client can select which types of communication objects should be included in the query.

```
class KsGetPPParams : public KsXdrAble {
public:
    KsGetPPParams(XDR * xdr, bool & success);
    KsGetPPParams();
    static KsGetPPParams *xdrNew(XDR *);

    KsString path;
    KS_OBJ_TYPE type_mask;
    KsString name_mask;
};
```

`KsGetPPParams(XDR *xdr, bool &success)`
: Constructs a service parameter object for the GetPP service from the stream `xdr`. If the constructor succeeds, then it'll indicate this by setting the reference parameter `success` to `true`. The construction of service parameter is normally handled within the abstract base class `KsServerBase`, so the programmer of service functions doesn't need to take care of this.

`KsGetPPParams()`
: Constructs a service parameter object on the client-side. The object then must be filled with the service parameters, and finally be send to the server over the wire with `xdrEncode()`.

`KsString path`
: Contains the path to the objects to be queried.

`KS_OBJ_TYPE type_mask`
: Contains the mask which specifies what types of communication objects should be included in the query.

`KsString name_mask`
: Contains a name mask with optional wildcards which selects the communication objects to be included in the query.

The reply of the GetPP service contains the following contents:

```
class KsGetPPResult : public KsResult {
public:
    KsGetPPResult();
    KsGetPPResult(XDR *xdr, bool &success);
    static KsGetPPResult *xdrNew(XDR *);

    KsList<KsProjPropsHandle> items;
};
```

`KsGetPPResult()`
: Constructs a new object of class `KsGetPPResult` which contains only an empty list of `items`. This constructor is typically used on the server-side.

`KsGetPPResult(XDR *xdr, bool &success)`
: Used on the client-side to construct a new object from the GetPP service reply. Returns `true` in `success`, if it succeeds.

`KsList<KsProjPropsHandle> items`
: The service function must return the projected properties of all communication objects, which satisfy the query, through this list.

4.3.3 The GetVar Service

With the help of the GetVar service, a KS client can read the values of variables within a KS server. The client sends in a list with names of variables, and the server then replies with the values of these variables or error codes, if some variables could not be read. Following is the function prototype of the GetVar service function:

```
virtual void KsServerBase::getVar(KsAvTicket     &ticket,
                                  KsGetVarParams &params,
                                  KsGetVarResult &result);
```

The incomming request contains an array of variable names, which can use either absolute or relative addressing.

```
class KsGetVarParams : public KsXdrAble {
public:
    KsGetVarParams();
    KsGetVarParams(XDR * xdr, bool & success);
    KsGetVarParams(size_t num_ids);
    static KsGetVarParams *xdrNew(XDR *);

    KsArray<KsString> identifiers;
};
```

`KsGetVarParams()`
: Default constructor for a service parameter object for the GetVar service.

`KsGetVarParams(XDR *xdr, bool &success)`
: Constructs a service parameter object for the GetVar service from the stream `xdr`. If the constructor succeeds, then it'll indicate this by setting the reference parameter `success` to `true`. The construction of service parameter is normally handled within the abstract base

class KsServerBase, so the programmer of service functions doesn't need to take care of this.

`KsGetVarParams(size_t num_ids)`
: Constructs a service parameter object on the client-side. The object then must be filled with the service parameters, and finally be send to the server over the wire with `xdrEncode()`.

`KsArray<KsString> identifiers`
: An array of variable identifiers. The names are not processed, thus they can be either absolute or relative pathnames for variable objects.

The reply of the GetVar service contains the following contents:

```
class KsGetVarResult : public KsResult {
public:
    KSGetVarResult();
    KsGetVarResult(size_t size);
    KsGetVarResult(XDR * xdr, bool & success);
    static KsGetVarResult *xdrNew(XDR *);

    KsArray<KsGetVarItemResult> items;
};
```

`KsGetVarResult(size_t size)`
: Constructs a service reply object for the GetVar service. The parameter `size` indicates, how many values of variables are to be send back to the client. The programmer of service functions normally doesn't need to take care of constructing the reply object.

`KsGetVarResult(XDR * xdr, bool & success)`
: Used on the client-side to construct a new object from the GetVar service reply. Returns `true` in `success`, if it succeeds.

`KsArray<KsGetVarItemResult> items`
: An array containing the results for the individual variables from the GetVar service. The programmer of the service function must return the results through this member variable `items`. See below for a description of the class `KsGetVarItemResult`.

```
class KsGetVarItemResult : public KsResult {
public:
    KsGetVarItemResult();
    KsGetVarItemResult(XDR *, bool &);
    static KsGetVarItemResult *xdrNew(XDR *);

    KsCurrPropsHandle item;
};
```

`KsResult result`
: Derived from the superclass `KsResult`. It indicates the outcome of the getVar operation on a particular variable object.

`KsCurrPropsHandle item`
: Contains the current properties (value & co.) if the getVar operation succeeded on this particular variable object (that is, the corresponding `result` member variable is `KS_ERR_OK`).

4.3.4 The SetVar Service

The SetVar service is the counterpart of the GetVar service in that it allows to set the values (current properties) of variables. Following is the function prototype of the SetVar service function:

```
virtual void KsServerBase::setVar(KsAvTicket     &ticket,
                                  KsSetVarParams &params,
                                  KsSetVarResult &result);
```

The incomming request contains an array of variable names, which can use either absolute or relative addressing, and the current properties to be set on these variables.

```
class KsSetVarParams : public KsXdrAble {
public:
    KsSetVarParams(XDR * xdr, bool & success);
    KsSetVarParams(size_t num_items);

    KsArray<KsSetVarItem> items;
};
```

`KsSetVarParams(XDR *xdr, bool &success)`
: Constructs an service parameter object for the SetVar service from the stream `xdr`. If the constructor succeeds, then it'll indicate this by setting the reference parameter `success` to `true`. The construction of service parameter is normally handled within the abstract base class `KsServerBase`, so the programmer of service functions doesn't need to take care of this.

`KsSetVarParams(size_t num_items)`
: Constructs a service parameter object on the client-side. The object then must be filled with the service parameters, and finally be send to the server over the wire with `xdrEncode()`.

`KsArray<KsSetVarItem> items`
: Contains name-value pairs of the variables to be set. See below for a description of the entries contained in this array.

```
class KsSetVarItem : public KsXdrAble {
public:
    KsString path_and_name;
    KsCurrPropsHandle curr_props;
};
```

`KsString path_and_name`
: Contains the path (either relative or absolute) and the name of the variable which is to be set.

`KsCurrPropsHandle curr_props`
: Contains the current properties of the variable to be set.

```
class KsSetVarResult : public KsResult {
public:
    KsSetVarResult(size_t size);
    KsSetVarResult(XDR * xdr, bool & success);
```

```
    KsArray<KsResult> results;
};
```

`KsSetVarResult(size_t size)`
: Constructs a service reply object for the SetVar service. The parameter `size` indicates, how many results of access operations to variables are to be send back to the client. The programmer of service functions normally doesn't need to take care of constructing the reply object.

`KsSetVarResult(XDR * xdr, bool & success)`
: Used on the client-side to construct a new object from the SetVar service reply. Returns `true` in `success`, if it succeeds.

`KsArray<KsResult> results`
: Contains the results of the individual set-variable operations. For every variable mentioned in the request, this array contains a corresponding result, which indicates, whether the operation succeeded.

4.3.5 The ExgData Service

The ExgData (exchange data) service provides a synchronized data exchange with simulators, etc. It first sets variables as indicated in the requests, triggers a calculation, and after this is finished, the service reads a (possibly different) set of variables and returns their values.

If you don't provide the ExgData service, you don't need to implement (override) this member function. The default service function `exgData()` in the `KsServerBase` class simply sends back to the client a `KsErrNotImplemented` error.

```
virtual void KsServerBase::exgData(KsAvTicket      &ticket,
                                   KsExgDataParams &params,
                                   KsExgDataResult &result);
```

The incoming request contains a list of the variables to be set together with the values, and a list of variables to be read after the necessary processing has occoured.

```
class KsExgDataParams : public KsXdrAble {
public:
    KsExgDataParams(XDR * xdr, bool & success);
    KsExgDataParams(size_t set_num_items, size_t get_num_items);

    KsArray<KsSetVarItem> set_vars;
    KsArray<KsString> get_vars;
};
```

`KsExgDataParams(XDR *xdr, bool & success)`
: Constructs an service parameter object for the ExgData service from the stream `xdr`. If the constructor succeeds, then it'll indicate this by setting the reference parameter `success` to `true`. The construction of service parameter is normally handled within the abstract base class `KsServerBase`, so the programmer of service functions doesn't need to take care of this.

`KsExgDataParams(size_t set_num_items, size_t get_num_items)`
: Constructs a service parameter object on the client-side. The object then must be filled with the service parameters, and finally be send to the server over the wire with `xdrEncode()`.

`KsArray<KsSetVarItem> set_vars`
: Contains the names and values of the variable objects to be set. For a description of the class `KsSetVarItem` please refer to the section about the SetVar service.

`KsArray<KsString> get_vars`
: Contains the names of the variables to be read as the final part of the exchange data operation. The names can bei either relative or absolute much like the names used in the request of the GetVar service.

```
class KsExgDataResult : public KsResult {
public:
    KsExgDataResult(size_t size);
    KsExgDataResult(XDR * xdr, bool & success);

    KsArray<KsResult> results;
    KsArray<KsGetVarItemResult> items;
};
```

`KsExgDataResult(size_t set_size, size_t get_size)`
: Constructs a service reply object for the ExgData service. The parameter `set_size` indicates how many results of the set-variable operations must be send back to the client. The parameter `get_size` indicates, how many results of get-variable operations are available and must be send back to the client, too. The programmer of service functions normally doesn't need to take care of constructing the reply object.

`KsExgDataResult(XDR * xdr, bool & success)`
: Used on the client-side to construct a new object from the ExgData service reply. Returns `true` in `success`, if it succeeds.

`KsArray<KsResult> results`
: Contains the results indicating the outcome of the set-variable operations.

`KsArray<KsGetVarItemResult> items`
: Contains the the values (current properties) of the variables to be read. Please refer to the SetVar service for an description.

4.4 Events and Timer Events

Timer events are used within KS server and manager objects to keep the registration mechanism of KS servers running. KS servers, for example, need to reregister themselves with the manager in regular time intervals.

Following is the public interface of the abstract base class `KsEvent`:

```
class KsEvent {
public:
    virtual void trigger() = 0;
    virtual ~KsEvent(); // make sure the destructor is always virtual.
};
```

`virtual void trigger()`
: Default action to be taken, when the event triggers. Derived classes must overwrite this method – or they won't be useful at all, because the `KsEvent` class doesn't know what to do when the event happens.

Timer events (of class `KsTimerEvent`) are specialised events. They work in "hand in hand" with server objects and store information about the time they are supposed to trigger.

```
class KsTimerEvent {
public:
    KsTimerEvent(const KsTime & at);
    virtual void trigger() = 0;
    KsTime remainingTime() const;

    friend bool operator == (const KsTimerEvent & lhs,
                             const KsTimerEvent & rhs);
    friend bool operator != (const KsTimerEvent & lhs,
                             const KsTimerEvent & rhs);
    friend bool operator <= (const KsTimerEvent & lhs,
                             const KsTimerEvent & rhs);
    friend bool operator >= (const KsTimerEvent & lhs,
                             const KsTimerEvent & rhs);
    friend bool operator <  (const KsTimerEvent & lhs,
                             const KsTimerEvent & rhs);
    friend bool operator >  (const KsTimerEvent & lhs,
                             const KsTimerEvent & rhs);

    KsTime triggersAt() const;
};
```

`virtual void trigger()`
: A method that calls the server method indicated by `method` when this event triggers. Thou shalt not call this one directly, as it is only for use by the timer event manager of the server object. If you plan to write your own timer events, then you must overwrite this member function in derived classes to trigger the action, this particular event class is bound to.

`KsTime triggersAt() const`
: Time when the timer should trigger.

`KsTime remainingTime() const`
: Returns the time span before the timer event triggers.

`friend bool operator == (const KsTimerEvent & lhs, const KsTimerEvent & rhs)`
: Returns `true`, if both timer events will trigger at the same time.

`friend bool operator != (const KsTimerEvent & lhs, const KsTimerEvent & rhs)`
: Returns `true`, if both timer events will not trigger at the same time

`friend bool operator <= (const KsTimerEvent & lhs, const KsTimerEvent & rhs)`
: Returns `true`, if the first timer event triggers before or at the same time as the second timer event.

`friend bool operator >= (const KsTimerEvent & lhs, const KsTimerEvent & rhs)`
: Returns `true`, if the first timer event triggers after or at the same time as the second timer event.

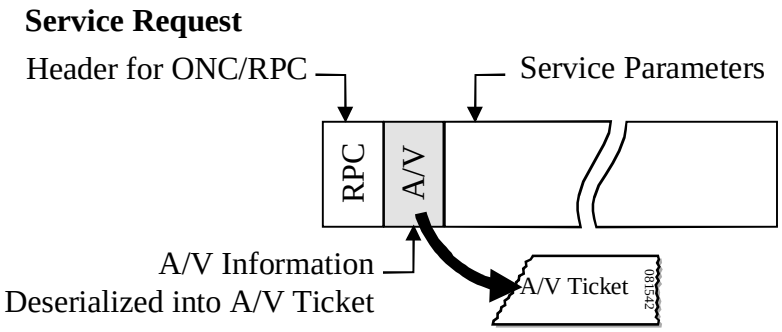
`friend bool operator < (const KsTimerEvent & lhs, const KsTimerEvent & rhs)`
: Returns `true`, if the first timer event triggers before the second timer event.

```
friend bool operator > (const KsTimerEvent & lhs,
    const KsTimerEvent & rhs)
```

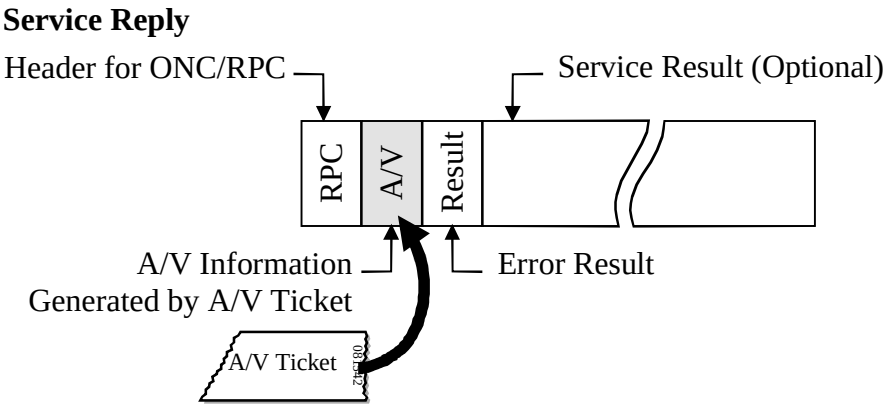
Returns `true`, if the first timer event triggers after the second timer event.

4.5 A/V Tickets

The authentification and verification mechanism of the ACPLT/KS protocol is represented within KS servers by "A/V tickets" (or "tickets" for short). Whenever a server object receives a service request from a client, it creates a ticket from the A/V data contained within the request.



Eventually, the server object sends back a reply to the client after a service function was executed. The ticket then is serialized, so it sends back to the client the necessary A/V information. After the reply is send, the ticket object is destroyed. Thus the A/V ticket mechanism within the server is stateless with respect to processing different requests.



4.5.1 Class KsAvTicket

Below is the public interface of the abstract class `KsAvTicket`:

```
typedef KsAvTicket * (*KsTicketConstructor)(XDR *);
class KsAvTicket : public XdrUnion {
public:
    static KsAvTicket *xdrNew(XDR *xdr);

    KS_RESULT result() const = 0;

    KS_ACCESS getAccess(KsString &name) const;
    virtual bool isVisible(const KsString & name) const;

    virtual bool canReadVar(const KsString &name) const = 0;
    virtual bool canWriteVar(const KsString &name) const = 0;
    virtual bool canReadVars(const KsArray<KsString> &names,
                             KsArray<bool> &canRead) const;
```

```
    virtual bool canWriteVars(const KsArray<KsString> &names,
                              KsArray<bool> &canWrite) const;

    static bool registerAvTicketType(enum_t ticketType,
                                     KsTicketConstructor ctor);
    static bool deregisterAvTicketType(enum_t ticketType);

    const struct sockaddr * getSenderAddress() const;
    struct in_addr getSenderInAddr() const;

    static KsAvTicket * emergencyTicket();
    virtual enum_t xdrTypeCode() const = 0;
};
```

`static KsAvNoneTicket *xdrNew(XDR *xdr)`
: Creates a new A/V ticket right out the XDR stream `xdr`. If it fails for any reason, the ticket is destroyed and a `0` pointer returned.

`KS_RESULT result()`
: Returns the status of the A/V ticket. If the state is set to `KS_ERR_OK`, then the ticket is valid and can be used to ask for access rights and validate variable accesses. If the state is either `KS_ERR_UNKNOWNAUTH`, `KS_ERR_BADAUTH` or `KS_ERR_GENERIC`, then the ticket is invalid and must be only used to send back an error reply to the KS client using `xdrEncode()`.

`KS_ACCESS getAccess(const KsString &name) const`
: Convenience function: asks the A/V ticket about the access mode of the communication object named `name`. The name must not be a relative path. The default implementation of this function calls `canReadVar()` as well as `canWriteVar()` to find out the access rights. If your particular server has a faster means to get the access rights of communication objects, then you should overwrite this method in a derived class.

`bool isVisible(const KsString &name) const`
: The default implementation returns `true`, if the communication object in question is at least readable and/or writable.

`virtual bool canReadVar(const KsString &name) const`
: Asks the A/V ticket, whether the communication object (of type "variable") named `name` can be read.

`virtual bool canWriteVar(const KsString &name) const`
: Asks the A/V ticket, whether the communication object (of type "variable") named `name` can be written to.

`virtual bool canReadVars(const KsArray<KsString> &names,`
`KsArray<bool> canRead) const`
: Same as `canReadVar()`, but with a list of names. This allows the ticket implementor to possibly optimize the access checks. The results of the checks are returned through the array `canRead`, which consists of `bool`s, indicating whether the respective variable can be read. If **all** variables listed in `names` can be read, only then `canReadVars()` returns `true` as a hint to the server implementor.

`virtual bool canWriteVars(const KsArray<KsString> &names,`
`KsArray<bool> &canWrite) const`
: Same as `canWriteVar()`, but with a list of names. This allows the ticket implementor to possibly optimize the access checks. The result is returned through the array `canWrite`, which consists of `bool`s, indicating whether the respective variable can be written to. If **all**

variables listed in `names` can be written to, only then `canWriteVars()` returns `true` as a hint to the server implementor.

`static bool registerAvTicketType(enum_t ticketType, KsTicketConstructor ctor)`
Registers a new A/V ticket type of `ticketType`, and returns `true`, if it succeeds. The parameter `ctor` must point to a static "factory method" that constructs a A/V ticket from a XDR stream. If a ticket type of `ticketType` is already registered, then the function will not register the new constructor factory method, and will return `false` immediately. Using this method, an implementor can add his/her own A/V mechanisms to the server. Because adding and removing A/V ticket types is done dynamically at run-time, it is possible to implement new A/V ticket types within shared libraries or dynamically loaded libraries.

`static bool deregisterAvTicketType(enum_t ticketType)`
Deregisters the A/V ticket type `ticketType` and returns `true`, if it succeeds.

`const struct sockaddr * getSenderAddress() const`
Returns the socket address of the requesting client.

`struct in_addr getSenderInAddr() const`
Returns the internet address of the requesting client or `INADDR_NONE` if the the client's address is not an internet address.

`static KsAvTicket *emergencyTicket()`
Returns a pointer to a static A/V ticket, which can be used in emergency situations where the memory is low. The ticket returned must not be freed and uses the A/V mechanism "NONE". It always send back an error result of `KS_ERR_GENERIC`.

`virtual enum_t xdrTypeCode() const`
In derived classes this member function returns the type code of the A/V mechanism used for this ticket class. For tickets using the A/V mechanism "NONE" this would be `KS_AUTH_NONE`.

4.5.2 Class KsAvNoneTicket

The class `KsAvNoneTicket` implements the A/V scheme "NONE". Following is the public interface of the class `KsAvNoneTicket`:

```
class KsAvNoneTicket : public KsAvTicket {
public:
    KsAvNoneTicket();
    KsAvNoneTicket(KS_RESULT r, KS_ACCESS a = KS_AC_NONE);
    KsAvNoneTicket(XDR *xdr, bool &success);

    virtual KS_ACCESS getAccess(const KsString &name) const;
    virtual bool canReadVar(const KsString &name) const;
    virtual bool canWriteVar(const KsString &name) const;
    virtual bool canReadVars(const KsArray<KsString> &names,
                             KsArray<bool> &canRead) const;
    virtual bool canWriteVars(const KsArray<KsString> &names,
                              KsArray<bool> &canWrite) const;

    static void setDefaultAccess(KS_ACCESS a);
    virtual enum_t xdrTypeCode() const
};
```

`KsAvNoneTicket()`
: Constructs a new A/V ticket that can be used to send back an error reply using the A/V scheme "NONE".

`KsAvNoneTicket(KS_RESULT r, KS_ACCESS a = KS_AC_NONE)`
: Constructs a new A/V ticket which will return a result of `r` and uses a default access mode of `a` for all communication objects.

`KsAvNoneTicket(XDR *xdr, bool &success)`
: Constructs a new A/V ticket from a XDR stream and returns `true` in `success` if it succeeds.

`static void setDefaultAccess(KS_ACCESS a)`
: Sets the default access mode for all communication objects, when using the A/V scheme "NONE".

`virtual enum_t xdrTypeCode() const`
: Returns the type code of the A/V scheme used, that is `KS_AUTH_NONE`.

For a description of the remaining functions please refer to the superclass `KsAvTicket`.

4.5.3 Class KsAvSimpleTicket

The class `KsAvSimpleTicket` implements the A/V scheme "Simple". You should derive a custom class from `KsAvSimpleTicket` implementing your access policy and register it with `KsAvTicket::registerAvTicketType()`. The default implementation falls back to the implementation provided by `KsAvNoneTicket`. Following is the public interface of the class `KsAvSimpleTicket`:

```
class KsAvSimpleTicket : public KsAvNoneTicket {
public:
    KsAvSimpleTicket();
    KsAvSimpleTicket(XDR *, bool &);

    virtual enum_t xdrTypeCode() const;

    KsString getId() const;
};
```

`KsString getId() const`
: Returns whatever identification string the client sent with this ticket.

`virtual enum_t xdrTypeCode() const`
: Returns the type code of the A/V scheme used, that is `KS_AUTH_SIMPLE`.

For a description of the remaining functions please refer to the superclass `KsAvTicket`.

4.6 Internet Address Sets

To ease access control checks in KS servers, the libkssvr provides classes for Internet address sets. Such sets contain IP addresses or ranges, appropriate masks, and a flag indicating whether an individual adress (range) is allowed or forbidden. A Internet address set can be created from a file, for instance, thus making it possible, to configurate servers individual to allow access from particular clients and to deny access from others. In addition, the C++ communication library provides a special Internet address set which contains all IP addresses of the host the KS server is running on.

All classes representing Internet address sets share a common public interface which is provided by the abstract interface class `KsInAddrSet` shown below.

```
class KsInAddrSet {
public:
    virtual bool isMember(in_addr addr) const = 0;
};
```

`virtual bool isMember(in_addr addr) const`
Returns `true`, if the IP address given in `addr` is contained the Internet address set and has been granted access.

4.6.1 Class KsSimpleInAddrSet

So server implementors to keep out the "bad guys", the class `KsSimpleInAddrSet` can be quite useful. Such Internet address sets store the IP addesses of clients who are granted access or who are not allowed to access a KS server. To make configuration easy, the list of friends and bad guys for a `KsSimpleInAddrSet` can be read from an ordinary file.

```
class KsSimpleInAddrSet:
    virtual public KsInAddrSet,
    private KsInAddrSet_base {
public:
    KsSimpleInAddrSet(bool defaultIsAccept = false);
    ~KsSimpleInAddrSet();

    virtual bool isMember(in_addr addr) const;

    bool accept(in_addr addr, in_addr mask);
    bool accept(in_addr addr);
    bool reject(in_addr addr, in_addr mask);
    bool reject(in_addr addr);

    void removeAll();
};
istream & operator >> (istream & istr, KsSimpleInAddrSet & set);
```

`KsSimpleInAddrSet(bool defaultIsAccept = false)`
Constructs a new and empty Internet address set. The argument `defaultIsAccept` controls, how the member function `isMember()` behaves if no suitable entry could be found in the address set. See below for an explanation.

`~KsSimpleInAddrSet()`
Destructs an Internet address set and frees all memory associated with it.

`virtual bool isMember(in_addr addr) const`
Looks up the IP address given in `addr` in the address set. If the address is found in the set and it belongs to the allowed addresses (once put into the set using `accept()`), then `true` is returned. If the address is found but belongs to the no-no addresses, then `false` is returned instead. If the address could not be found in the set, then the return value is the value specified as the argument `defaultIsAccept` when constructing the address set object. In general, you might want to use the policy "if I don't know you, then I don't trust you" and specifiy `false` for `defaultIsAccept`, thus denying all clients not explicitly listed as allowed. Also note that the set is searched from beginning to end, that is, accepted or rejected addresses inserted into the set before other addresses are tested first.

`bool accept(in_addr addr)`
Adds a single "allowed" address to the end of the set. The function `isMember()` will return `true` for this address as long as there is no other match before this one in the set which denies access.

`bool accept(in_addr addr, in_addr mask)`
Adds a range of IP address or a single address to the end of the set. The argument `mask` controls what bits of the address `addr` are significant. For example, if you want to grant access to all IP addresses in the range of `134.130.126.0` to `134.130.126.255` then you must specify the address as `134.130.126.0` and the mask must be `255.255.255.0`.

`bool reject(in_addr addr)`
Adds a single "denied" address to the end of the set.

`bool reject(in_addr addr, in_addr mask)`
Adds a range of denied IP address or a single denied address to the end of the set. Please refer to `accept()` for a description of the arguments.

`istream & operator >> (istream & istr, KsSimpleInAddrSet & set)`
Reads in the set of allowed and rejected IP addresses from an `istream`. If this fails, then the stream will be in the failed state. The format of the data supplied to the Internet address set through the `istream` must consist of zero or more lines of the following form:

`+<address>/<mask>`

or

`-<address>/<mask>`

The "+" sign at the beginning of a line introduces addresses to be accepted whereas the "–" sign indicates addresses which must be denied. Lines for IP addresses to be allowed or denied can be freely intermixed, but always a line which come first will be checked first when the `isMember()` function is called on a Internet address set. Only the "dotted IP format" is allowed, but no DNS host names. For example, if you want to cut off a particular bad guy within a subnet, use:

```
-134.130.125.33/255.255.255.255
+134.130.125.0/255.255.255.0
```

You **must** list the "bad guy" first, otherwise the more general rule "`+134.130.125.0/255.255.255.0`" will trigger first – and this wouldn't be what you've intended. So **never** do this:

~~`+134.130.125.0/255.255.255.0`~~
~~`-134.130.125.33/255.255.255.255`~~

Always specify the special rules first, then the general rules.

4.6.2 Class KsHostInAddrSet

Object instances of the class `KsHostInAddrSet` represent the set of IP addresses currently assigned to the host a KS server is running on. Thus, this class makes it possible to check easily whether a service request was issued by a KS client on the same machine as this server or from a remote client. For example, the ACPLT/KS manager uses a `KsHostInAddrSet` to make sure that only local KS servers can register and deregister themselves, and no remote process can mess around with the registration procedure. Below is the public interface of the class `KsHostInAddrSet`:

```
class KsHostInAddrSet:
    virtual public KsInAddrSet,
    private KsInAddrSet_base {
public:
    KsHostInAddrSet();
    ~KsHostInAddrSet();

    virtual bool isMember(in_addr addr) const;
    bool update();
};
```

`KsHostInAddrSet()`
: Creates an Internet address set and fills in all IP addresses for the host this object is created on. Only IP addresses of currently active interfaces (or bound interfaces on NT) are stored in this address set.

`~KsHostInAddrSet()`
: Destructs an Internet address set containing the host's IP addresses.

`virtual bool isMember(in_addr addr) const`
: Returns `true`, if the IP address given in `addr` is a currently active IP address of this host.

`bool update()`
: Rereads all IP addresses currently assigned to this host and returns `true`, if it succeeds. This function can be used to reflect configuration changes in the host's IP address list.

4.7 Simple Server Communication Objects

As every KS server (and even the KS manager) must implement at least a "`/vendor`" branch within their tree of communication objects, the C++ communication library eases this task through the "simple communication objects". These communication objects are "simple" in the sense, that they are (more or less) static objects within the process space of the KS server and are not aware that there might be a real DCS or a simulator below a particular KS server.

4.7.1 Class KssSimpleCommObject

Below is the public interface of the base class `KssSimpleCommObject`, which forms the base for the derived simple communication objects:

```
class KssSimpleCommObject {
public:
    virtual ~KssSimpleCommObject();

    KsString  getIdentifier() const;
    KsTime    getCreationTime() const;
    KsString  getComment() const;

    void setCreationTime(const KsTime &);
    void setComment(const KsString &);
};
```

`virtual ~KssSimpleCommObject()`
: Destructs a simple communication object and makes sure that this destructor is always virtual in derived classes.

`KsString getIdentifier() const`
Returns the name (identifier) of this particular simple communication object.

`KsTime getCreationTime() const`
Returns the creation date and time of this simple communication object.

`KsString getComment() const`
Returns the comment associated with this simple communication object.

`void setCreationTime(const KsTime &)`
Sets the creation date and time.

`void setComment(const KsString &)`
Sets the comment associated with this simple communication object.

4.7.2 Class KssSimpleDomain

The class `KssSimpleDomain` implements a domain object that can contain other communication objects but has no knowledge about a possible underlaying DCS, simulator, or whatever, and therefore can only handle intrinsic objects. Following is the public interface of the class `KssSimpleDomain`:

```
class KssSimpleDomain : public KssDomain, public KssSimpleCommObject {
public:
    KssSimpleDomain(const KsString &id,
                    KsTime ctime = KsTime::now(),
                    KsString comment = KsString());

    virtual KsString getIdentifier() const;
    virtual KsTime   getCreationTime() const;
    virtual KsString getComment() const;

    virtual KssSimpleDomainIterator * newIterator() const;
    size_t size() const;
    virtual KssCommObjectHandle getChildById(const KsString & id)
                const;

    bool addChild(KssCommObjectHandle hChild);
    bool addChild(KssCommObject *pChild);
    KssCommObjectHandle removeChild(const KsString &id);

    void setNextSister(KssDomainHandle hDomain);
    bool setNextSister(KssDomain *pDomain);
};
```

`KssSimpleDomain(const KsString &id, KsTime ctime = KsTime::now(), KsString comment = KsString())`
Constructs a new simple domain object with the name `id` and the given creation time and commend.

`KsString getIdentifier() const`
Returns the name (identifier) of this simple domain object.

`KsTime getCreationTime() const`
Returns the creation time of this simple domain object.

`KsString getComment() const`
Returns the comment associated with this simple domain object

`KssSimpleDomainIterator * newIterator() const`
: Returns an iterator suitable to iterate over the child communication objects of this domain object. Don't forget to dispose the iterator after use.

`size_t size() const`
: Returns the number of child communication objects this domain currently manages and therefore knows about.

`KssCommObjectHandle getChildById(const KsString &id) const`
: Returns the handle of the child with the name `id`. If this domain has no such child, then an unbound handle is returned instead.

`bool addChild(KssCommObjectHandle hChild)`
: Adds a child to this domain. The handle mechanism ensures that memory management will not free the child as long as either this domain or someone other is interested in it. If the function should fail, then it'll return `false`.

`bool addChild(KssCommObject *pChild)`
: Puts the child pointed to by `pChild` under the control of a handle (with an ownership of `PltOsNew`) and then adds it to the list of children. If the function should fail for any reason, then it'll return `false`.

`KssCommObjectHandle removeChild(const KsString &id)`
: Removes a child with the name `id` from the list of children of this domain and returns the handle to the child. If there is no such child, then an unbound handle is returned instead.

`void setNextSister(KssDomainHandle hDomain)`
: Associates a sister domain with this domain. All sister domains share the same name (identifier) which is controlled by the first domain in the sister chain. Outside the KS server these sister domains appear as a single monolithic domain. Using sister domains, you can implement different access mechanisms to the children through different domain classes and therefore often do not need to reinvent the wheel completely. This function always succeeds.

`bool setNextSister(KssDomain *pDomain)`
: Same as the function above, but this time first a handle is created for the sister domain to be attached to this domain. In case this function can not create the handle, it returns `false`.

4.7.3 Class KssSimpleDomainIterator

With the help of the class `KssSimpleDomainIterator` you can easily iterate over the child objects of a simple domain object. This iterator returns handles to the child objects of a domain if it is dereferenced, so dangling pointers are avoided. Below is the public interface of the class `KssSimpleDomainIterator`:

```
class KssSimpleDomainIterator : public KssDomainIterator {
public:
    KssSimpleDomainIterator(const KssSimpleDomain & d);
    virtual ~KssSimpleDomainIterator();

    virtual operator const void * () const;
    virtual KssCommObjectHandle operator * () const;
    virtual KssSimpleDomainIterator& operator ++ ();
    virtual void toStart();
};
```

`KssSimpleDomainIterator(const KssSimpleDomain &d)`
Creates an iterator which iterates over the children of the simple domain object given in `d`.

For a description of the remaining interface functions please refer to the descriptions chapter 2.2, "Iterators".

4.7.4 Class KssSimpleVariable

The class `KssSimpleVariable` implements a variable object that can handle a value object but has no knowledge about a possible underlaying DCS, simulator, or whatever. Following is the public interface of the class `KssSimpleVariable`:

```
class KssSimpleVariable :
    public KssVariable, public KssSimpleCommObject {
public:
    KssSimpleVariable(const KsString &id,
                      KsTime ctime = KsTime::now(),
                      const KsString & comment = KsString());

    virtual KsString  getIdentifier() const;
    virtual KsTime    getCreationTime() const;
    virtual KsString  getComment() const;
    virtual KS_ACCESS getAccessMode() const;

    virtual KsString getTechUnit() const;
    virtual KsValueHandle getValue() const;
    virtual KsTime getTime() const;
    virtual KS_STATE getState() const;

    void setTechUnit(const KsString &);
    virtual KS_RESULT setValue(const KsValueHandle &);
    virtual KS_RESULT setValue(KsValue *p);
    virtual KS_RESULT setTime(const KsTime &);
    virtual KS_RESULT setState(KS_STATE);

    bool isWriteable() const;
    void lock();
};
```

`KssSimpleVariable(const KsString &id, KsTime ctime = KsTime::now(), const KsString &comment = KsString())`
Constructs a new simple variable object with the name `id` and the given creation time and commend.

`KsString getIdentifier() const`
Returns the name (identifier) of this simple variable object.

`KsTime getCreationTime() const`
Returns the creation time of this simple variable object.

`KsString getComment() const`
Returns the comment associated with this simple variable object

`KS_ACCESS getAccessMode() const`
Returns the access mode of this simple variable object.

`KsString getTechUnit() const`
Returns the technical unit set for the value of this simple variable object.

`KsValueHandle getValue() const`
Returns a handle to the value of this simple variable object. If this variable has currently no associated value, then an unbound handle is returned.

`KsTime getTime() const`
Returns the time associated with the value.

`KS_STATE getState() const`
Returns the state of the variable's value.

`void setTechUnit(const KsString &)`
Sets the technical unit for the values of this variable object.

`KS_RESULT setValue(const KsValueHandle &)`
Sets a new value for this variable object. Due to the handle mechanisms the old value is automatically freed if no other object is any longer interested in it. If this function fails for any reason an error result is returned, otherwise `KS_ERR_OK`.

`KS_RESULT setValue(KsValue *p)`
Same as the function above, but this time the necessary handle for the value is created on the fly. If this should fail for any reason, then the function returns an appropriate error code.

`KS_RESULT setTime(const KsTime &)`
Sets the time stamp of the variable's value.

`KS_RESULT setState(KS_STATE)`
Sets the state of the variable's value.

`void lock()`
Locks this particular variable object, such that it can not be written to anymore. All further attempts to set the value are then denied.

`bool isWriteable() const`
Returns `true`, if this variable object can be written to.

4.8 NT Services

4.8.1 Usage

If you do not plan to develop for the Windows NT operating system, then you can skip this section completely.

The "NT Services" are NT's way of firing up and managing system service programs during startup and lateron. NT controls the starting sequence of services, as well as what services to start with the help of the onmipresent "registry". The registry stores the necessary information about the services installed on a NT system.

Creating a service on NT means to register a service control function during the startup phase of the service, creation of a thread for the KS server object, as well as some other housekeeping actions. Fortunately, the concept of the KS server objects within the C++ communication library makes it easy to accompany a *KS server* object with a *NT service* object. The NT service object then encapsulates the whole service mechanism and controls the KS server object.

You only have to tell your particular service object how to create your KS server object. Following is a simple example of how you can do this:

```
class NtService : public KsNtServiceServer {
```

```
public:
    NtService()
        : KsNtServiceServer("ACME_server", 7000, 7000, 7000, 7000) { }

protected:
    virtual KsServerBase *createServer(int argc, char **argv);
}; // class NtService
```

Within the default constructor `NtService()` you simply rely on the constructor of the service base class `KsNtServiceServer` to carry out the dirty work. You simply supply the service name (which must be indentical with the name used for the registry entries for this service), as well as some information about the time span this particular server needs for initializing, etc.

Next, you must tell the NT service object how to create your KS server object. In addition, in the example below, we first parse the command line (which can contain optional parameters supplied to the service at startup), and eventually create the KS service object.

```
KsServerBase *NtService::createServer(int argc, char **argv)
{
    int i;

    for ( i = 0; i < argc; i++ ) {
        if ( (stricmp(argv[i], "--v") == 0) ||
             (stricmp(argv[i], "--verbose") == 0) ) {
             _is_verbose = true;
        } else if ( (stricmp(argv[i], "--v-") == 0) ||
             (stricmp(argv[i], "--verbose-") == 0) ) {
             _is_verbose = false;
        }
    }
    verbose("Creating ACME server object");
    return new ACMEServer;
} // NtService::createServer
```

It is always a good idea to create a logger object, which should be in our case a `PltNtLog` object. This particular logger object class knows how to communicate with NT's system logger. You should use a static logger object which gets initialized, before the `main()` function of your service executable is called. This way, the logger is already in place when the real service startup will take place.

```
PltNtLog log("ACME Server");
```

Okay, you've got your KS server object class, your NT service object class, and you're now ready to fire up your KS server. Within the `main()` function of your service excutable, create the service object, check for an error condition, and if everthing is allright so far, then simply `run()` the service. That's all!

```
int main(int, char **) {
    NtService nt_service;

    if ( nt_service.isOk() ) {
        nt_service.run();
    } else {
        PltLog::Error("Failed to setup the ACME service object");
    }
    return nt_service.isOk() ? 0 : 42;
} // main
```

4.8.2 NT Registry Mumbo Jumbo

As mentioned earlier, the NT registry stores information about what services to start when NT comes up, and in which sequence the individual services must be started. The start sequence below is crucial for the KS communication system:

- TCP/IP service,
- ONC/RPC portmap service,
- ACPLT/KS manager service,
- and finally your KS server service.

Please make sure that the ONC/RPC portmap service is registered and that it depends on the TCP/IP service. Therefore, the registry must contain the following entries:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Portmap
    DependOnService: REG_MULTI_SZ: Tcpip
    DisplayName: REG_SZ: Portmap
    ErrorControl: REG_DWORD: 0x1
    ImagePath: REG_SZ_EXPAND: %SystemRoot%\System32\portmap.exe
    ObjectName: REG_SZ: LocalSystem
    Start: REG_DWORD: 0x2
    Type: REG_DWORD: 0x10
```

Next, make sure that the ACPLT/KS manager service for NT is installed, too, and that it depends on the successful startup of the ONC/RPC portmap service:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\KS_manager
    DependOnService: REG_MULTI_SZ: Tcpip Portmap
    DisplayName: REG_SZ: ACPLT/KS Manager
    ErrorControl: REG_DWORD: 0x1
    ImagePath: REG_SZ_EXPAND: %SystemRoot%\System32\ntksmanager.exe
    ObjectName: REG_SZ: LocalSystem
    Start: REG_DWORD: 0x2
    Type: REG_DWORD: 0x10
```

Finally, set up your KS server service (change the entries printed in italics accordingly to your servers's name and location of the executable):

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ACME_server
    DependOnService: REG_MULTI_SZ: Tcpip Portmap KS_manager
    DisplayName: REG_SZ: ACME KS Server
    ErrorControl: REG_DWORD: 0x1
    ImagePath: REG_SZ_EXPAND: %SystemRoot%\System32\acme.exe
    ObjectName: REG_SZ: LocalSystem
    Start: REG_DWORD: 0x2
    Type: REG_DWORD: 0x10
```

4.8.3 Class KsNtServiceServer

The abstract base class `KsNtServiceServer` encapsulates NT's service mechanism. This class must be abstract because it complements a KS server (or manager) object, and therefore it doesn't know how to create the real KS server (or manager) object. Below is the public as well as the protected interface of the class `KsNtServiceServer`:

```
class KsNtServiceServer {
public:
    KsNtServiceServer(const char *service_name,
                      unsigned int create_timeout,
                      unsigned int spinup_timeout,
```

```
                        unsigned int spindown_timeout,
                        unsigned int destroy_timeout);
    virtual ~KsNtServiceServer();

    void run();

    bool isOk() const;
    bool isVerbose() const;
    bool reportServiceStatus(DWORD curr_stat,
                             DWORD exit_code, DWORD user_code,
                             DWORD checkpoint, DWORD wait_hint);
    DWORD currentServiceStatus();

    static KsNtServiceServer *getService();

protected:
    virtual KsServerBase *createServer(int argc, char **argv) = 0;

    void lastError(const char *msg);
    void verbose(const char *msg);

    bool _is_ok;
    bool _is_verbose;
};
```

`KsNtServiceServer(const char *service_name, unsigned int create_timeout, unsigned int spinup_timeout, unsigned int spindown_timeout, unsigned int destroy_timeout)`
Constructs a NT service object for the registered service `service_name`. Because NT uses a simplified watchdog mechanisms to supervise the services, you must supply some timeout values: `create_timeout` specifies the time span within the service object must construct the KS server and setup the server object, whereas `destroy_timeout` limits the time, the service takes to destroy the server object and to clean up. The timeouts `spinup_timeout` and `spindown_timeout` specify how long it may take in the worst case to issue a `startServer()` or `stopServer()` on the server object.

`virtual ~KsNtServiceServer()`
Destructs a NT service object.

`void run()`
Starts the NT service and returns when the service fails or is shut down normally.

`bool isOk() const`
Returns `true`, if the service object is fine.

`bool isVerbose() const`
Returns `true`, if verbose reporting (debug reporting) is activated.

`bool reportServiceStatus(DWORD curr_stat, DWORD exit_code, DWORD user_code, DWORD checkpoint, DWORD wait_hint)`
Reports the current status of the service object to NT's service manager. This can be used to report either the progress or completion of a service operation. The current mode is indicated by `curr_stat`, which can be one of the symbols `SERVICE_STARTED`, `SERVICE_START_PENDING`, and so on (see the appropriate descriptions of Win32 for more details about service states). If your service fails for any reason, then you can report an error code through `exit_code` and optionally `user_code` (if `user_code` is not zero, then `exit_code` is automatically set to `ERROR_SERVICE_SPECIFIC_ERROR`). The

argument `checkpoint` is used during a lengthy operation to report progress and simply contains an arbitrary number which must change between each progress step. The `wait_hint` indicates how long the current progress step may take.

`DWORD currentServiceStatus()`
Returns the current service status, which can be one of the symbols `SERVICE_STARTED`, `SERVICE_START_PENDING`, and so on...

`static KsNtServiceServer *getService()`
Returns a pointer to the service object. There can be only one service object instantiated at the same time. If you try to instantiate another service instance, then the instantiation will fail and `isOk()` will return `False`.

`KsServerBase *createServer(int argc, char **argv) = 0`
Derived classes must implement this method and create and return within a KS server object. Additional command line arguments (which may have been set from the NT service manager) are supplied through `argc` and `argv`.

`void lastError(const char *msg)`
Sends the error message `msg` to the current logger object (through `Plt::Error()`) and appends the error code supplied by the Win32 function `GetLastError()` to the message.

`void verbose(const char *msg)`
If the NT service object is set to verbose mode (`_is_verbose` is `true`), the message `msg` is send to the current logger object (through `Plt::Debug()`).

`bool _is_ok`
Indicates whether this NT service object's state is okay. If you encouter during startup of your service or server objects any problems, you may set this member variable to `false` to indicate a failure.

`bool _is_verbose`
Indicates whether this NT service object is verbose.

4.9 Windows 95 Service Processes

4.9.1 Usage

Although "little" Windows 95 does not share NT's service mechanism, it knows at least of "service processes". Such service processes can be started when Windows comes up and they will not terminate when a user logs out and another user logs in. Service processes only terminate when the whole system is shut down. Unfortunately your service processes will not be fore-warned of pending system crashes.

4.9.2 Windows 95 Registry Mumbo Jumbo

– preliminary information –

At boot time, Windows 95 checks the registry key `"RunServices"` and the values below this key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion
  RunServices
    ArbitraryKeyname = w95ksmanager.exe
```

The values below the key `"RunServices"` can have any arbitrary name but must have a value type of string.

4.9.3 Class KsW95ServiceServer

– preliminary information –

The service process class `KsW95ServiceServer` shares some similiarities in its use with the class `KsNtServiceServer`. But due to the very different service concepts of NT and 95, there are also some differences between both service server object classes.

As an example of how to program a ACPLT/KS service process, please see the source code of `ntksmanager.cpp` (the NT and W95 ACPLT/KS manager services share the same source file).

5 libkscln

The "libkscln" library part contains object classes which are used solely for KS clients. This part of the C++ communication library sits on top of the more generic libraries "libks" and "libplt".

5.1 Communication Objects

In ACPLT/KS, information is organized and provided in a KS server through a tree of communication objects. The communication objects can be either domains (container) or variables (leaves).

5.1.1 Usage

The communication objects – like variables and domains – within ACPLT/KS servers are represented on the client-side by appropriate objects. In the following example, we first create a domain object for the server domain "`/vendor`":

```
KscDomain vendor("//ws001.plt/toaster/vendor");
```

Now we can ask this domain for its children. To achieve this, we first ask the domain for a child iterator and tell it that we want to iterate over **all** children.

```
KscChildIterator &child_it = *vendor.newChildIterator(KS_OT_ANY);
cout << "/vendor has the following children:" << endl;
```

Whether we've iterated over all children can be determined by casting the iterator's value to a value of type `void*`. Don't worry, this is the default behaviour when you dereference the iterator in a `if()` or `while()` statement.

```
while ( child_it ) {
```

When dereferencing the iterator, you normally get back the projected properties of the child communication object, the iterator currently is positioned at. Please note, that the iterator iterates over the **projected properties** of a domain's children, **and not over child objects**.

```
    cout << child_it->identifier << " (type ";
    switch ( child_it->xdrTypeCode() ) {
    case KS_OT_DOMAIN:
        cout << "domain"; break;
    case KS_OT_VARIABLE:
        cout << "variable"; break;
    default:
        cout << "unknown"; break;
    }
    cout << ")" << endl;
    if ( child_it->comment )
        cout << "  Comment: " << child_it->comment << endl;
    cout << "  Access mode(s): ";
```

```
    if ( child_it->access_mode & KS_AC_READ )
        cout << "read ";
    if ( child_it->access_mode & KS_AC_WRITE )
        cout << "write ";
    cout << endl;
```

Finally we move the iterator to the next child and end the loop.

```
    ++child_it;
}
```

Eventually, delete the iterator to free any occupied memory.

```
delete &child_it;
```

Now for another example. To ask a variable for its value, you should first create an object for it with the name of the communication object including the host and server where it is located.

```
KscVariable vendor_name("//ws001.plt/toaster/vendor/name");
```

Now you must tell the variable object to ask the KS server where the (real) communication object is located for the current value. Then you can get the value, which is represented by an value object of subclass `KsValue`:

```
vendor_name.getUpdate();
KsValueHandle value = vendor_name.getValue();
```

After we've made sure that we've indeed got back a value with an expected type of "string", we can output it:

```
KsStringValue *pv = PLT_DYNAMIC_PCAST(KsStringValue, value.getPtr());
if ( pv ) {
    cout "Vendor name is: " << (const char *) *pv << endl;
}
```

5.1.2 Class KscCommObject

Following is the public interface of the abstract base class `KscCommObject`:

```
class KscCommObject {
public:
    KscCommObject(const char *object_path);
    virtual ~KscCommObject();

    virtual KS_OBJTYPE typeCode() const = 0;

    bool hasValidPath() const;
    KsString getName() const;
    KsString getPathOnly() const;
    const KscPath &getPathAndName() const;
    KsString getHostAndServer() const;
    KsString getFullPath() const;

    KscServerBase *getServer() const;

    virtual const KsProjProps *getProjProps() const = 0;
    virtual bool getProjPropsUpdate() = 0;

    virtual void setAvModule(const KscAvModule *avm);
    virtual const KscAvModule *getAvModule() const;

    KS_RESULT getLastResult() const;

    bool operator == (const KscCommObject &);
```

```
    bool operator != (const KscCommObject &);
    bool operator <  (const KscCommObject &);
    bool operator <= (const KscCommObject &);
    bool operator >  (const KscCommObject &);
    bool operator >= (const KscCommObject &);
};
typedef PltPtrHandle<KscCommObject> KscCommObjectHandle;
```

`KscCommObject(const char *object_path)`
: Creates a communication object on the client side which represents an ACPLT/KS communication object located in an ACPLT/KS, which is adressed by the resource locator `object_path`.

`virtual ~KscCommObject()`
: Destructs a communication object.

`virtual KS_OBJTYPE typeCode() const`
: Returns the object type of this communication object, which can be either `KS_OT_DOMAIN`, `KS_OT_VARIABLE`.

`bool hasValidPath() const`
: Returns `true`, if the resource locator of this communication object is valid. If the resource locator is malformed, for example: "`rumpelpumpel/villa/kunterbunt`", then `hasValidPath()` returns `false` instead.

`virtual KsString *getName()`
: Returns the name of the communication object. If the resource locator of the communication object is "`//host/server/villa/kunterbunt`", then `getName()` returns "`kunterbunt`".

`KsString getPathOnly() const`
: Returns the path field of the resource locator of this communication object. If the resource locator is "`//host/server/villa/kunderbunt`", then `getPathOnly()` returns the path "`villa`".

`const KscPath &getPathAndName() const`
: Returns the path and name fields of the resource locator of this communication object. If the resource locator is "`//host/server/villa/kunterbunt`", then `getPathAnd-Name()` returns "`villa/kunterbunt`".

`KsString getHostAndServer() const`
: Returns the host and server fields of the resource locator where this communication object is located. If the resource locator is "`//host/server/villa/kunterbunt`", then `getHostAndServer()` returns "host`/server`".

`KsString getFullPath() const`
: Returns the full resource locator of this communication object, for example "`//host/server/villa/kunterbunt`".

`KscServerBase *getServer() const`
: Returns a pointer to a server object of subclass `KscServerBase`, this communication object is associated with. If the resource locator of this communication objects is invalid, then a `0` pointer is returned instead.

`virtual const KsProjProps *getProjProps() const`
: Returns a pointer to this communication object's projected properties. Derived class supply appropriate implementations for this member function.

`virtual bool getProjPropsUpdate()`

Asks the corresponding KS server, where this communictaion object is located, for the projected properties. Derived classes supply appropriate implementations for this member function.

`virtual void setAvModule(const KscAvModule *avm)`

Associates a new AV module with the communication object. If you don't want to associate any AV module with this particular communication object, then specify a `0` pointer as the value for the parameter `avm`. The associated AV module will be used to supply the "role" when requesting the projected properties, the GetPP service, the SetVar service, or the GetVar service from a KS server.

`virtual KscAvModule *getAvModule() const`

Returns a pointer to the AV module used for this communication object. If the object has currently no associated AV module, then `getAvModule()` returns a `0` pointer.

`KS_RESULT getLastResult() const`

Returns the result code of the last operation performed on this communication object. See section 5.4 below for a description of possible error codes.

```
bool operator == (const KscCommObject &)
bool operator != (const KscCommObject &)
bool operator <  (const KscCommObject &)
bool operator <= (const KscCommObject &)
bool operator >  (const KscCommObject &)
bool operator >= (const KscCommObject &)
```

Compares two communication objects according their resource locators.

5.1.3 Class KscVariable

Communication objects which are variables, are represented in the KS client by objects of the class `KscVariable`. Following is the public interface of the `KscVariable` class:

```
class KscVariable : public KscCommObject {
public:
    KscVariable(const char *name);
    ~KscVariable();

    KS_OBJ_TYPE typeCode() const;

    virtual bool getProjPropsUpdate();
    virtual bool getUpdate();
    virtual bool setUpdate();
    KsValueHandle getValue() const;

    virtual KsVarProjProps *getProjProps() const;
    KsVarCurrProps *getCurrProps() const;
    KsCurrPropsHandle getCurrPropsHandle();
    bool setCurrProps(KsVarCurrProps &cp);
    bool isDirty() const;

    virtual void setAvModule(const KscAvModule *avm);
    virtual const KscAvModule *getAvModule() const;
};
typedef PltCompHandle<KscVariable> KscVariableHandle;
```

`KscVariable (const char *name)`
: Constructs a variable object which represents the variable communication object on the KS server on the host specified in `name`.

`~KscVariable()`
: Destructs a variable object.

`virtual KS_OBJTYPE typeCode() const`
: Returns the object type of this communication object: `KS_OT_VARIABLE`.

`virtual bool getProjPropsUpdate()`
: Asks the corresponding KS server, where this communictaion object is located, for the projected properties.

`bool getUpdate()`
: Initiates an update of the current properties of the variable object, and thus an update of the variable's value. To accomplish this, the variable object uses a connection object to issue a GetVar service request to the KS server who publishes the variable. If the current properties could be read from the server, then `getUpdate()` returns `true`.

`bool setUpdate()`
: Writes the current properties stored within this variable object back to the KS server who publishes this variable. If the current properties could be written to the server, then `setUpdate()` returns `true`. Note that it is an error if you call `setUpdate()` for a variable object which has no value set (that is, you have either not called `setCurr-Props()` or set an unbound handle).

`KsValueHandle getValue() const`
: Returns a handle of the value object. To get the most up-to-date value, you should first call `getUpdate()` to read the variable in the KS server. If you have yet not executed `getUpdate()` at all, then this objects knows of no current properties, and thus of no value, and the call to `getValue()` will return a `0` pointer.

`KsVarProjProps *getProjProps() const`
: Returns a pointer to the variable's projected properties.

`KsVarCurrProps *getCurrProps() const`
: Returns a pointer to the variable's current properties. If you will use these current properties for a longer period of time, then you should use `getCurrPropsHandle()` instead, as this ensures that the current properties will not errorneously freed before you're though with them.

`KsCurrPropsHandle getCurrPropsHandle()`
: Returns the handle of the current properties. Using the handle ensures that the properties live at least as long as you need it.

`bool setCurrProps(const KsVarCurrProps &cp)`
: Sets the new current properties of the variable. To propagate them to the KS server, you must then call the `setUpdate()` method.

`bool isDirty() const`
: Returns `true`, if the current properties have been changed using `setCurrProps()`.

For a description of description of the remaining methods please refer to the superclass `KscCommObject`.

5.1.4 Class KscDomain

Domain objects work as containers for other domains or variables. Following is the public interface of the class `KscDomain`:

```
class KscDomain : public KscCommObject {
public:
    KscDomain(const char *name);
    ~KscDomain();

    virtual bool getProjPropsUpdate();
    virtual KsDomainProjProps *getProjProps() const;
    KscChildIterator *newChildIterator(KS_OBJTYPE mask,
                            KsString nameMask = KsString("*"));

    virtual void setAvModule(const KscAvModule *avm);
    virtual const KscAvModule *getAvModule() const;
};
typedef PltIterator<KsProjPropsHandle> KscChildIterator;
typedef PltPtrHandle<KscDomain> KscDomainHandle;
```

`KscDomain (const char *name)`
: Constructs a domain object which represents the domain communication object on the KS server on the host specified in `name`.

`~KscDomain()`
: Destructs a domain object.

`virtual bool getProjPropsUpdate()`
: Asks the corresponding KS server, where this communictaion object is located, for the projected properties.

`virtual KsDomainProjProps *getProjProps() const`
: Returns a pointer to the domain's projected properties.

`KscChildIterator *newChildIterator(KS_OBJTYPE mask, KsString nameMask = KsString("*"))`
: Returns a new iterator which can iterate over the child communication objects of this domain. The parameter `mask` can be used to select the type of objects you want to iterate over: either `KS_OT_DOMAIN` for domains, or `KS_OT_VARIABLE` for variables. If you specifiy `KS_OT_ALL` instead, then you will get back an iterator that iterates over all children of this domain. The method `newChildIterator()` may fail (for example due to memory constraints, unable to contact the KS server), then it will return a `0` pointer.

  The second argument, `nameMask`, controls what names of communication objects can match. This argument can be either a mask (with simplified regular expressions, see the Technology Paper #4 for details) or a name. In the latter case you will get at most only one child back.

  The iterator returned by `newChildIterator()` works like the other iterators you've seen in the communication library so far. Whenever you dereference it (`*child_it`), then you'll get back a handle to the projected properties (`KsProjProps*`) of a communication object. Be sure to destroy the child iterator after use. Also see the next section for a description of the class `KscChildIterator`.

For a description of description of the remaining methods please refer to the superclass `KscCommObject`.

5.1.5 Typedef KscChildIterator

A "child iterator" is returned by `KscDomain` objects and can be used to iterate over the children of a domain communication object within a KS server. A `KscChildIterator` is nothing more than an "ordinary" iterator (with the public interface of a `PltIterator`) which returns the handles to the projected properties of the children of a domain. Please note that the real iterator class for a `KscChildIterator` is hidden through the generic iterator interface of `PltIterator`, so never assume a particular class for it. `KscChildIterator` is therefore nothing more than a `typedef`:

```
typedef PltIterator<KsProjPropsHandle> KscChildIterator;
```

So let us now dive into a simple example which shows how to use such a child iterator to iterate over the children over a domain communication object, specified in the argument `branch`. In this example, we first ask this domain object for an iterator.

```
void DumpBranch(KscDomain &branch)
{
    KscChildIterator *children;

    children = branch.newChildIterator(KS_OT_ANY, "*");
    if ( !children ) {
        cout << "Can't allocate child iterator" << endl;
        return;
    }
```

Okay, we've got an iterator and this will iterate over all children regardless of their type and their name – that's why we've specified `KS_OT_ANY` and the name mask `"*"` in the call to `newChildIterator()`.

So how do we iterate? As you can see from the public interface of the abstract `PltIterator` class, asking an iterator for a bool returns a value, which is `true` as long as there are still more items in a container. Because the above call to `branch.newChildIterator()` returns a pointer to the iterator (and not the iterator itself), we must dereference this pointer whenever we want to query the iterator. Thus, the `for` loop below asks the iterator at every step "do you have more items?" and then increments it. That's all.

```
    for ( ; *children; ++*children ) {
```

Within the loop, we want to peek at the projected properties of the particular children of the domain. In order to avoid messing around with the ownership of memory and dangling pointers, the child iterator returns a handle. We just dereference the iterator (`**children`, because `*children` would just dereference the pointer to the iterator but not the iterator itself) to get this handle and copy it into the temporary handle `current`. Then we can do whatever we want to do depending on the type of child (domain, variable, ...). The function `xdrTypeCode()` is just fine for this task, as it returns the type of projected property, we're investigating through the handle `current`.

```
        KsProjPropsHandle current(**children);
        if ( current->xdrTypeCode() == KS_OT_DOMAIN ) {
            // do whatever you need to do if it is a domain...
        } else if ( current->xdrTypeCode() == KS_OT_VARIABLE ) {
            // do whatever you need to do if it is a variable...
        }
    }
```

Finally, clean up. In our little example, this involves only destroying the child iterator.

```
    delete children;
} // DumpBranch
```

5.2 Packages

5.2.1 Usage

Packages are a means to group variable objects together and to get or set their values in one batch. In the following example we first create some variables:

```
KscVariable vendor_name("//ws001/toaster/vendor/name");
KscVariable server_name("//ws001/toaster/vendor/server_name");
KscVariable server_version("//ws001/toaster/vendor/server_version");
```

In the next step, we create a package object and add the variable objects to it. This way the package can optimize the KS server access(es) when we later request an update of the variable's values. As a final note and to be more precise, you don't add the variable objects to the package but instead handles to the variable objects. This way, the lifetime of variable objects can be controlled more closely. You must only take note of using the right ownership of the variable object managed by a handle. In the following lines of code the ownership must be `PltOsUnmanaged` because we've used objects in either static or automatic storage (depending on whether the `KscVariable` objects live in the stack or the global data segment).

```
KscPackage pkg;
pkg.add(KscVariableHandle(&vendor_name, PltOsUnmanaged));
pkg.add(KscVariableHandle(&server_name, PltOsUnmanaged));
pkg.add(KscVariableHandle(&server_version, PltOsUnmanaged));
pkg.getUpate();
```

Not only variable objects can be added to a package, but also other packages (well – handles to packages, but you can guess this by now). This way you can easily build up larger packages from sub-packages suitable for special purposes.

5.2.2 Class KscPackage

Following is the public interface of the class `KsPackage`:

```
class KscPackage {
public:
    KscPackage();
    ~KscPackage();

    bool add(const KscVariableHandle var);
    bool add(const KscPackageHandle pkg);
    bool remove(const KscVariableHandle var);
    bool remove(const KscPackageHandle pkg);

    size_t sizeVariables(bool deep = false) const;
    size_t sizeSubpackages() const;
    bool isDirty() const;
    bool getUpdate();
    bool setUpdate(bool force = false);

    KS_RESULT getLastResult() const;

    PltIterator<KscVariableHandle> *newVariableIterator(
                                        bool deep=false) const;
    PltIterator<KscPackageHandle> *newSubpackageIterator() const;
```

```
    void setAvModule(const KscAvModule *avm);
    KscAvModule *getAvModule() const;
};
typedef PltPtrHandle<class KscPackage> KscPackageHandle;
```

`KsPackage()`
: Constructs a new (and yet unfilled) package.

`~KsPackage()`
: Destructs (destroys) a package.

`bool add(const KscVariableHandle var)`
: Adds a new variable object to the package. As the package uses handles for referencing the variable objects stored inside them, the lifetime of a variable object is controlled by its associated handle.

`bool add(const KscPackageHandle pkg)`
: Adds another package to this package. This way, it is possible to create packages from packages.

`bool remove(const KscVariableHandle var)`
: Removes a variable object from the package and returns `true`, if it succeeds.

`bool remove(const KscPackageHandle pkg)`
: Removes a package from the package and returns `true`, if it succeeds.

`size_t sizeVariables(bool deep = false) const`
: Returns the number of variable objects contained in this package. If the parameter `deep` is `false`, then `sizeVariables()` only returns the number of variable objects registered directly with this package and not such variables which are contained in subpackages. If parameter `deep` is `true`, then the total count of all variables is returned.

`size_t sizeSubpackages() const`
: Returns the number of subpackages contained in this package. Subpackages of subpackages are not counted.

`bool isDirty() const`
: Returns `true`, if either variable objects or subpackages have been added or removed since the last `getUpdate()` or `setUpdate()`.

`bool getUpdate()`
: Does an optimized `getUpdate()` on all variables contained within this package, whether they have been directly added to this package or any subpackage. If the operation fails then `false` is returned. In this case you can ask for the error code using `getLastResult()`. If it is `KS_ERR_OK`, then the service request itself was granted but reading individual variables might have failed. In this case, iterate over the childen of the package and check their status using using `getLastResult()`.

`bool setUpdate(bool force = false)`
: Does an optimized `setUpdate()` on all variables contained within this package, whether they have been directly added to this package or any subpackage. If `force` is `false`, then only the current properties of such variables are send back to the KS server(s), which have been changed since the last `setUpdate()`. If the operation fails then `false` is returned. In this case you can ask for the error code using `getLastResult()`. If it is `KS_ERR_OK`, then the service request itself was granted but writing individual variables might have failed. In this case, iterate over the childen of the package and check their status using

using `getLastResult()`. Note that is also an error if you call `setUpdate()` and one or more variable objects have yet no value.

`KS_RESULT getLastResult() const`
Returns the result code of the last operation performed on this package.

`PltIterator<KscVariableHandle> *newVariableIterator(bool deep = false)`
Returns an iterator suitable for iterating over the variable objects (to be more precise: handles to the variable objects) contained within this package. If the parameter `deep` is `false`, then the iterator will iterate only over such variable objects, which have been directly added to this package and not over variables contained in subpackages.

`PltIterator<KscPackageHandle> *newSubpackageIterator()`
Returns an iterator suitable for iterating over the subpackages of this package. If you want to visit all subpackages, you have to recursively descend down the subpackages by asking each subpackage again for subpackage-iterators.

`void setAvModule(const KscAvModule *avm)`
Associates a new AV module with the package. If you don't want to associate any AV module with this package object, then specify a `0` pointer as the value for the parameter `avm`. The associated AV module will be used to supply the "role" when doing operations on the contents of the package as long as individual variables or subpackages do not have their own AV modules set.

`KscAvModule *getAvModule() const`
Returns a pointer to the AV module used for this package. If the package has currently no associated AV module, then `getAvModule()` returns a `0` pointer.

5.3 Data Exchange Package

5.3.1 Usage

The data exchange package consists of two subpackages, one containing a set of variables first to be written, and the other containing a set of variables to be read afterwards. This way, the data exchange package can be used to send data to a simulator, wait for the results to become ready, and then return them. In the following example, we first create two packages:

```
KscVariable in1("//ws001/toaster/tstep");
KscVariable in2("//ws001/toaster/tstart");
KscVariable in3("//ws001/toaster/tend");
KscPackage in_pkg;
in_pkg.add(KscVariableHandle(&in1, PltOsUnmanaged));
in_pkg.add(KscVariableHandle(&in2, PltOsUnmanaged));
in_pkg.add(KscVariableHandle(&in3, PltOsUnmanaged));
// Now set the values of in1, in2, in3...

KscVariable out1("//ws001/toaster/values");
KscPackage out_pkg;
out_pkg.add(KscVariableHandle(&out1, PltOsUnmanaged));
```

Then, we create a data exchange package and specify what packages (sets of variables) to write and read. Finally, we exchange the data with a KS server.

```
KscExchangePackage ex(
    KscPackageHandle(&in_pkg, PltOsUnmanaged),
    KscPackageHandle(&out_pkg, PltOsUnmanaged));
ex.doExchange();
```

5.3.2 Class KscExchangePackage

Following is the public interface of the class `KscExchangePackage`:

```
class KscExchangePackage {
public:
    KscExchangePackage();
    KscExchangePackage(KscPackageHandle setPkg,
                       KscPackageHandle getPkg);
    void setPackages(KscPackageHandle setPkg,
                     KscPackageHandle getPkg);
    void getPackages(KscPackageHandle &setPkg,
                     KscPackageHandle &getPkg) const;
    bool doExchange(bool force = true);

    KS_RESULT getLastResult() const;

    void setAvModule(const KscAvModule *avm);
    KscAvModule *getAvModule() const;
};
```

`KscExchangePackage()`
: Constructs a new data exchange package. The package has no associated packages yet.

`KscExchangePackage(KscPackageHandle setPkg, KscPackageHandle getPkg)`
: Constructs a new data exchange package and sets the two packages to be used for the data exchange with a KS server.

`void setPackages(KscPackageHandle setPkg, KscPackageHandle getPkg)`
: Sets the "set package" and the "get package". The "set package" contains the variables to be written to the KS server, whereas the "get package" contains these variables that are to be read afterwards.

`void getPackages(KscPackageHandle &setPkg, KscPackageHandle &getPkg) const`
: Returns references to the two packages currently used as the "set package" and the "get package".

`bool doExchange(bool force = true)`
: Executes an ExgData service on the appropriate KS server. If `force` is `true`, then all variables within the "set package" are always written to the KS server, regardless of whether they have changed or not since the last `doExchange()`. If the operation fails then `false` is returned. In this case you can ask for the error code using `getLastResult()`. Note that is also an error to call `doExchange()` and one or more variable objects in the "get package" have yet no value.

  If the operation failed, but `getLastResult()` returns `KS_ERR_OK`, then the data exchange request itself was granted but the KS server returned errors for individual variables in the set and get package. Iterate over the variables in the set and get packages of the data exchange package and ask them using `getLastResult()` to find out which variables could not be set or read.

`KS_RESULT getLastResult() const`
: Returns the result code of the last operation performed on the data exchange packages.

`void setAvModule(const KscAvModule *avm)`
: Associates a new AV module with the data exchange object. If you don't want to associate any AV module with this data exchange object, then specify a `0` pointer as the value for the

parameter `avm`. The associated AV module will be used to supply the "role" when requesting the DataExg service from a KS server for such packages and variables contained in the data exchange object which don't have their own AV module set.

`KscAvModule *getAvModule() const`
Returns a pointer to the AV module used for this data exchange object. If the data exchange object has currently no associated AV module, then `getAvModule()` returns a `0` pointer.

5.4 Error Codes

In an ideal world, there are probably no errors. But back to reality, you must be aware that operations on Variables and Domains may fail for various reasons. In addition to those error codes which are already defined by the core ACPLT/KS protocol, the client library introduces some additional error codes.

Following is a list of ACPLT/KS error codes (core protocol and client library) which can be returned through the `getLastResult()` methods of the classes `KscCommObject` (or its subclasses `KscVariable`, `KscDomain`), `KscPackge`, and `KscExchangePackage`:

`KS_ERR_OK = 0x0000`
No error occoured. Fine.

`KS_ERR_GENERIC = 0x0001`
A generic error happened, which may be have been caused for example by memory or resource constraints.

`KS_ERR_BADAUTH = 0x0002`
The client couldn't identify itself properly with an ACPLT/KS server, so the service request was denied.

`KS_ERR_UNKNOWNAUTH = 0x0005`
The client used an authentification/verification scheme which isn't supported by an ACPLT/KS server.

`KS_ERR_NOTIMPLEMENTED = 0x0003`
An ACPLT/KS server does not support a particular operation. For example, you may see this error code if you try to do a `doExchange()` on a `KscExchangePackage` and the KS server does not support the "Data Exchange" service.

`KS_ERR_BADPARAM = 0x0004`
One or more parameters supplied to a service request were invalid.

`KS_ERR_BADNAME = 0x0010`
A resource locator or search path contains invalid characters.

`KS_ERR_BADPATH = 0x0011`
An invalid resource locator or search path was used, for example, a communication object with such a resource locator does not exist.

`KS_ERR_BADMASK = 0x0012`
An invalid mask was specified.

`KS_ERR_NOACCESS = 0x0013`
The KS client has no access permission to a communication object in a ACPLT/KS server.

`KS_ERR_BADTYPE = 0x0014`
A new value was written to a ACPLT/KS server and this value was of a different data type than the value within the server and the server could not convert between these data types.

`KS_ERR_CANTSYNC = 0x0015`
: A "Data Exchange" operation is not possible with the variables stored in the packages of a `KscExchangePackage`.

`KS_ERR_NOREMOTE = 0x0020`
: An operation is not permitted to be requested by a remote client (or server).

`KS_ERR_SERVERUNKNOWN = 0x0021`
: No such ACPLT/KS server is available on the host specified in the resource locator of a communication object.

`KS_ERR_MALFORMEDPATH = 0x1001`
: Client error. A communication object (`KscVariable`, `KscDomain`) was created using an invalid path name, for example "`in/valid/resource/locator`". Resource locators must start with a double slash, then hostname, servername, path and object name, all separated by slashes.

`KS_ERR_NETWORKERROR = 0x1002`
: Client error. A network error occured during the communication with a KS server. This error can also be caused by attempts to send the current value of a `KscVariable` object to a KS server and you have yet not set the current value of a variable communication object.

`KS_ERR_TYPEMISMATCH = 0x1003`
: Client error.

`KS_ERR_HOSTUNKNOWN = 0x1004`
: Client error. A communication object (`KscVariable`, `KscDomain`) was created using an unknown host name (thus, the DNS lookup was not successful).

`KS_ERR_CANTCONTACT = 0x1005`
: Client error. An ACPLT/KS server or manager could not contacted, for example it refused the new connection.

`KS_ERR_TIMEOUT = 0x1006`
: Client error. There was no answer from a server within the timeout period.

`KS_ERR_NOMANAGER = 0x1007`
: Client error. No ACPLT/KS manager is available on the host specified in the resource locator of a communication object.

5.5 A/V Modules

The ACPLT/KS protocol optionally uses so-called "A/V mechanisms" for authentification, whenever a KS client wants to read or write variables, or to list communication objects within the object tree of a KS server with restricted access. In the "C++ Communication Library" objects of class `KscAvModule` (or any subclass of) handle the protocol details and represent particular A/V mechanisms.

5.5.1 Usage

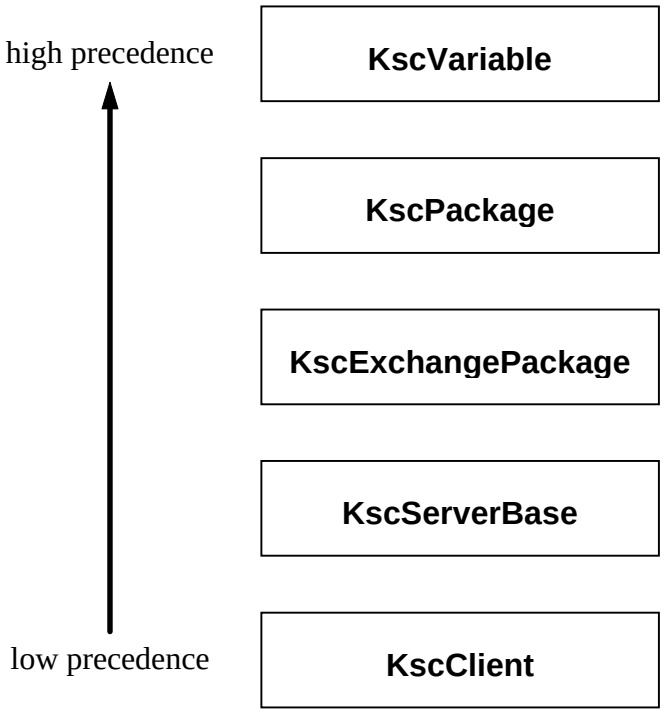
To use the A/V mechanism you simply create an apropriate A/V module object. One example of such a A/V module object is the class `KscAvSimpleModule`, which represents the so-called A/V mechanism "Simple". When creating such an object, you supply an identification string to the constructor:

```
KscAvSimpleModule simple_av("secret");
```

From now on, you simply attach such an A/V module to every object which should use this particular A/V mechanism. The A/V modules can be attached to communication objects, packages, and exchange packages. In addition, you can use an A/V module on a per-server or even per-client by clamping it to a server or client object.

```
KscVariable v("//host/server/rumpelpumpel/villa/kunterbunt");
v.setAvModule(&simple_av);
```

There is no restriction about what variables in which packages use A/V (even different) modules. Just attach A/V module objects as needed. You only have to note the precedence order shown in the figure below, if you – for example – attach A/V modules to a package and a variable within that package. In this case the A/V module attached to the variable takes precedence over the A/V module associated with the package. If the other variables in that package don't have an associated A/V module, then these variables will use the A/V module of the package.



5.5.2 Class KscAvModule

Below is the public interface of the abstract base class `KscAvModule`:

```
class KscAvModule {
public:
    virtual ~KscAvModule();
    virtual KS_AUTH_TYPE typeCode() const = 0;
};
```

`virtual KS_AUTH_TYPE typeCode() const`

Returns the object type of this A/V module object, which can be `KS_AUTH_NONE` or `KS_AUTH_SIMPLE`. Additional (and optional) A/V module object classes will define and return addition type codes.

As mentioned above, the C++ Communication Library already comes with specialized A/V module classes for the A/V mechanisms "None" and "Simple". Normally, you don't need to explicitly create A/V None modules and clamp them to unsuspecting communication objects. If

you don't associate any A/V module with a communication object, it will use the A/V None mechanism by default.

5.5.3 Class KscAvNoneModule

Following is the public interface of the class `KscAvNoneModule`:

```
class KscAvNoneModule : public KscAvModule
{
public:
    KscAvNoneModule();
    virtual ~KscAvSimpleModule();
    virtual KS_AUTH_TYPE typeCode() const;
};
```

`KscAvNoneModule()`
Constructs an A/V module which uses the A/V mechanism "None".

`virtual ~KscAvSimpleModule()`
Destructs an A/V module.

`virtual KS_AUTH_TYPE typeCode() const`
Returns the object type of this A/V module object, which is `KS_AUTH_NONE`.

5.5.4 Class KscAvSimpleModule

Following is the public interface of the class `KscAvSimpleModule`:

```
class KscAvSimpleModule : public KscAvModule
{
public:
    KscAvSimpleModule(const char *id);
    virtual ~KscAvSimpleModule();
    virtual KS_AUTH_TYPE typeCode() const;
};
```

`KscAvSimpleModule(const char *id)`
Constructs an A/V module which uses the A/V mechanism "Simple". The argument `id` specifies the identification string to be used for authentification with the KS server.

`virtual ~KscAvSimpleModule()`
Destructs an A/V module.

`virtual KS_AUTH_TYPE typeCode() const`
Returns the object type of this A/V module object, which is `KS_AUTH_SIMPLE`.

5.6 Special Objects

Every communication object is loosely associated with a client object (of class `KscClient` or a subclass of) and a server proxy object (of class `KscServerBase` or a subclass of). These two object classes work mainly behind the scenes and are responsible for the underlaying communication. Ordinarily, you don't need to worry about them at all. You may have to deal with them, if you either want to associate an A/V mechanism globally with either the client or with a particular KS server.

5.6.1 Class KscServerBase

For each KS server, for which you create communication objects, the C++ communication library automatically creates a server proxy object, which is of a specialized class derived from the class `KscServerBase`. Note that these server proxy object classes have no public constructor, so you can't create them – and in fact you don't need to. Their main purpose to the application programmer which uses the C++ communication library is to associate a particular A/V mechanism with a particular KS server. In addition, you can get some status information.

Following is the public interface of the class `KscServerBase`:

```
class KscServerBase {
public:
    KsString getHost() const;
    KsString getName() const;
    KsString getHostAndName() const;

    KS_RESULT getLastResult() const;

    virtual void setAvModule(const KscAvModule *);
    virtual const KscAvModule *getAvModule() const;
};
```

`KsString getHost() const`
: Returns the name of the host (either the DNS name or a dotted IP address) where the KS server for this server proxy object is located.

`KsString getName() const`
: Returns the name of the KS server for this server proxy object.

`KsString getHostAndName() const`
: Returns the host name and server name as "*`//host/server`*".

`KS_RESULT getLastResult() const`
: Returns the result code of the last operation performed on this KS server.

`void setAvModule(const KscAvModule *avm)`
: Associates a new AV module with the server proxy object. If you don't want to associate any AV module with this server proxy object, then specify a `0` pointer as the value for the parameter `avm`. The associated AV module will be used to supply the "role" when communicating with a KS server when no AV module has been set for particular packages or variables.

`KscAvModule *getAvModule() const`
: Returns a pointer to the AV module used for this server proxy object. If it has currently no associated AV module, then `getAvModule()` returns a `0` pointer.

5.6.2 Class KscServer

The class `KscServer` is a specialised `KscServerBase`, which handles the communication with "real" ACPLT/KS servers. Whenever you create a variable object which refers to an "ordinary" ACPLT/KS server (or manager), then such a `KscServer` is created (one for each server the client has created communication objects for).

```
class KscServer : public KscServerBase {
public:
```

```
    void setTimeouts(const PltTime &rpc_timeout,
                     const PltTime &retry_wait,
                     size_t tries);
};
```

```
void setTimeouts(const PltTime &rpc_timeout,
    const PltTime &retry_wait, size_t tries)
```

Controls through the argument `rpc_timeout` how long a KS client waits for a service answer from a KS server after it has successfully sent out a service request to the server. If either the request times out or the network connection to the server breaks, the KS client will try to reconnect to the server. It will try at most `tries` times (including the first request), waiting the amount of time specified in `retry_wait` between reconnection tries. If this all fails, then the operation, which triggered the service request, will fail with an error code of `KS_ERR_TIMEOUT`.

Depending on the configuration of the TCP/IP network transport, there can be delays up to two minutes before the underlaying operating system recognized a broken TCP/IP connection. If a KS server is multi-homed (has multiple IP addresses on different interfaces), then the C++ communication library will try to connect to such a server using a different IP address after a communication failure.

5.6.3 Class KscClient

Every ACPLT/KS client application using the C++ communication library has exactly one client object. This client object is allocated automatically the first time you create any client communication object. But you can also create this object yourself – if you want. There can't be no more than one such client object at any time. Normally, you only need to deal with such a client object is when you want to globally set a A/V mechanism for all communication objects and packages, which don't have their own A/V mechanism set.

Below is the public interface of the class `KscClient`:

```
class KscClient {
public:
    virtual ~KscClient();

    static KscClient *getClient();
    static bool setClient(KscClient *cl, KsOwnership os);

    KscServerBase *getServer(const KsString &host_and_name);

    void setAvModule(const KscAvModule *);
    const KscAvModule *getAvModule() const;

    void setTimeouts(const PltTime &rpc_timeout,
                     const PltTime &retry_wait,
                     size_t tries);
};
```

```
virtual ~KscClient()
```

Ensures that client objects are always properly destructed.

```
static KscClient *getClient()
```

Returns a pointer to the client object used for this ACPLT/KS client application. If the client object has yet not been created, then a `0` pointer is returned.

`static bool setClient(KscClient *cl, KsOwnership os)`

If you want to use your own specialized client object, then you can register it using `setClient()`. The argument `os` indicates whether the specialized client object should be destroyed automaticaly when the application is terminated (`os` set to `PltOsNew`), or should be left alone (`os` set to `PltOsUnmanaged`). Other values for the argument `os` are not allowed. If the function succeeds, then it returns `true`. You can only set the client object to use *once* and *before* the first communication object has been created.

`KscServerBase *getServer(const KsString &host_and_name)`

If there is a server object currently associated with the KS server on the host as specified by the argument `host_and_name`, then a pointer to the object will be returned. Otherwise, you'll get the `0` pointer.

`void setAvModule(const KscAvModule *avm)`

Associates a new AV module with the client object. If you don't want to associate any AV module with this client object, then specify a `0` pointer as the value for the parameter `avm`. The associated AV module will be used to supply the "role" when communicating with a KS server when no AV module has been set for particular servers, packages or variables.

`KscAvModule *getAvModule() const`

Returns a pointer to the AV module used for this client object. If it has currently no associated AV module, then `getAvModule()` returns a `0` pointer.

`void setTimeouts(const PltTime &rpc_timeout,`
`const PltTime &retry_wait, size_t tries)`

Controls through the argument `rpc_timeout` how long a KS client waits for a service answer from a KS server after it has successfully sent out a service request to the server. If either the request times out or the network connection to the server breaks, the KS client will try to reconnect to the server. It will try at most `tries` times (including the first request), waiting the amount of time specified in `retry_wait` between reconnection tries. If this all fails, then the operation, which triggered the service request, will fail with an error code of `KS_ERR_TIMEOUT`.

Depending on the configuration of the TCP/IP network transport, there can be delays up to two minutes before the underlaying operating system recognized a broken TCP/IP connection. If a KS server is multi-homed (has multiple IP addresses on different interfaces), then the C++ communication library will try to connect to such a server using a different IP address after a communication failure.

6 Appendices

6.1 Preprocessor Symbols

The C++ communication library defines some preprocessor symbols or relies on symbols defined by the development environment in order to provide a unified and mostly platform-independent view to the C++ sources.

To compile source code which uses header files of the C++ communication library, you must define exactly one "platform symbol". For the example programs included in the library, you'll find the definitions in the platform-specific makefiles. Currently, the following platform symbols are available:

`PLT_SYSTEM_HPUX`
: Define this symbol as `1`, if you're compiling for the HP-UX operating system by Hewlett Packard.

`PLT_SYSTEM_IRIX`
: Define this symbol as `1`, if you're compiling for the IRIX operating system by Silicon Graphics.

`PLT_SYSTEM_LINUX`
: Define this symbol as `1`, if you're compiling for the Linux operating system by Linus Thorwalds et al.

`PLT_SYSTEM_SOLARIS`
: Define this symbol as `1`, if you're compiling for the Solaris operating system by Sun Microsystems.

`PLT_SYSTEM_NT`
: Define this symbol as `1`, if you're compiling for either Windows NT or Windows 95.

`PLT_SYSTEM_OPENVMS`
: Define this symbol as `1`, if you're compiling for the OpenVMS operating system by DEC.

These symbols come handy if you're developing multi-platform applications. In this case, you can compile in different code according to the platform you're compiling the source for. If you forget to define exactly one of these platform symbols, then compilation will be aborted with an error message by the header file `"config.h"`.

The following symbols are available to control build-options of the C++ Communication Library:

`NDEBUG`
: Define this symbol, if you want to disable debugging features like memory tracking and diagnostic messages. For production code, define this symbol, because some diagnosis methods used in debugging mode can slow down the code.

Depending on the compiler you are using, the C++ communication library defines exactly one of the following compiler symbols with a value different from zero. The other compiler symbols are defined as zero. For example, if you're using the GNU C++-Compiler `g++`, then the symbol `PLT_COMPILER_GCC` will contain the compiler's version number and all other compiler symbols will be defined as zero (`"0"`).

`PLT_COMPILER_DECCXX`
: Version number of the DEC `CXX` compiler, or `0` if another compiler is used.

`PLT_COMPILER_GCC`
: Version number of the `g++` compiler, or `0` if another compiler is used.

`PLT_COMPILER_MSVC`
: Version number of the MS Visual C+ compiler, or `0` if another compiler is used.

`PLT_COMPILER_BORLAND`
: Version number of the Borland `C++` compiler, or `0` if another compiler is used.

Depending on the compiler you're using for compilation, different code option defines will be automatically set as soon as you include the first of any C++ communication library header file. These code options defines are used to adjust the source code accordingly to the varying levels of C++ standard conformance of the various C+(+) compilers. The following code option defines are available:

`PLT_SIMULATE_BOOL`
: Defined as `1`, if a C++ compiler does not support the `bool` data type. In this case, the C++ communication library defines its own boolean data type named `"bool"`:

```
typedef int bool;
enum { false=0, true=1 };
```

`PLT_SIMULATE_RTTI`
: Defined as `1`, if a C++ compiler does not support run-time type information (RTTI). RTTI can be used at the run-time of an application to find out of what class an object is.

`PLT_RETTYPE_OVERLOADABLE`
: Defined as `1`, if a C++ compiler supports overloading of a function's return value type. See section 1.5 for details about return type overloading. To assist the application programmer in writing cross-platform code, the macro `PLT_RETTYPE_CAST(typ)` is defined. Its use is shown in the following example:

```
KssSimpleDomain *psd;
KssDomainIterator *pit =
        PLT_RETTYPE_CAST((KssDomainIterator *))
            psd->newIterator();
```

Here we ask an object of class `KssSimpleDomain` (or any subclass of) for an iterator to iterate over the children of the domain object. If a C++ compiler supports return type overloading (`PLT_RETTYPE_OVERLOADABLE = 1`), then `newIterator()` will return a pointer to an iterator of class `KssDomainIterator`. This class is in turn a subclass of `PltIterator`. In this case, the macro `PLT_RETTYPE_CAST` will just boil down to nothing, so the compiler eventually sees this source code:

```
KssSimpleDomain *psd;
KssDomainIterator *pit = psd->newIterator();
```

If a C++ compiler does not support return type overloading, then the pointer returned by `newIterator()` must be of `"PltIterator *"` instead of `"KssDomainIterator *"`. In this case, the macro `PLT_RETTYPE_CAST` will boil down to its argument, which should always be enclosed in double brackets to avoid trouble with template class typecasts. The compiler will eventually see this source code:

```
KssSimpleDomain *psd;
KssDomainIterator *pit = (KssDomainIterator *) psd->newIterator();
```

`PLT_INSTANTIATE_TEMPLATES`
: Defined as `1`, if a C++ compiler can't handle automatic template instantiation. In this case, you must explicitely instantiate the code for these templates that are used in your project. For instance, if you use a template class `foo<double>`, then you must instantiate the appropriate code for it using a line of code like this:

```
template class foo<double>;
```

`PLT_SEE_ALL_TEMPLATES`
: Defined as `1`, if a C++ compiler must see the full template class definitions when generating code.

6.2 Header Files

The next tables show which identifiers or symbols are defined in which header files. For some classes you'll see two header files mentioned in the "Header File" column, the second file name ending in "`_impl.h`". This second header file is used only to instantiate this template and you only need it to include once in exactly one module of you project. The last column "Library" shows you against which libraries you need to link you ACPLT/KS application.

Identifier	Header File	Library
`KS_ACCESS` (enum)	`ks/ks.h`	–
`KS_AC_`*xxx* (→ enum `KS_ACCESS`)	`ks/ks.h`	–
`KS_AUTH_TYPE` (enum)	`ks/ks.h`	–
`KS_AUTH_`*xxx* (→ enum `KS_AUTH`)	`ks/ks.h`	–
`KS_ERR_`*xxx* (→ enum `KS_RESULT`)	`ks/ks.h`	–
`KS_RESULT` (enum)	`ks/ks.h`	–
`KS_OBJTYPE` (enum)	`ks/ks.h`	–
`KS_OT_`*xxx* (→ enum `KS_OBJTYPE`)	`ks/ks.h`	–
`KS_STATE` (enum)	`ks/ks.h`	–
`KS_ST_`*xxx* (→ enum `KS_STATE`)	`ks/ks.h`	–
`KS_VAR_TYPE` (enum)	`ks/ks.h`	–
`KS_VT_`*xxx* (→ enum `KS_VAR_TYPE`)	`ks/ks.h`	–
`KsArray` (class)	`ks/array.h` `ks/arry_impl.h`	`libks`
`KsAvTicket` (class)	`ks/avticket.h`	`libks`
`KsAvNoneTicket` (class)	`ks/avticket.h`	`libks`
`KsBoolValue` (class)	`ks/value.h`	`libks`
`KsBoolVecValue` (class)	`ks/value.h`	`libks`
`KsByteVecValue` (class)	`ks/value.h`	`libks`
`KscChildIterator` (class)	`ks/commobject.h`	`libkscln`
`KscClient` (class)	`ks/client.h`	`libkscln`
`KscCommObject` (abstract class)	`ks/commobject.h`	`libkscln`
`KscCommObjectHandle` (typedef)	`ks/commobject.h`	–
`KscDomain` (class)	`ks/commobject.h`	`libkscln`

KscDomainHandle (typedef)	ks/commobject.h	–
KscExchangePackage (class)	ks/package.h	libkscln
KscPackage (class)	ks/package.h	libkscln
KscPackageHandle (typedef)	ks/package.h	–
KscServer (class)	ks/client.h	libkscln
KscServerBase (abstract class)	ks/client.h	libkscln
KscVariable (class)	ks/commobject.h	libkscln
KscVariableHandle (typedef)	ks/commobject.h	–
KsCurrProps (class)	ks/props.h	libks
KsCurrPropsHandle (typedef)	ks/props.h	–
KsDomainCurrProps (class)	ks/props.h	libks
KsDomainProjProps (class)	ks/props.h	libks
KsDoubleValue (class)	ks/value.h	libks
KsDoubleVecValue (class)	ks/value.h	libks
KsEvent (abstract class)	ks/event.h	libks
KsHostInAddrSet (class)	ks/hostinaddrset.h	libkssvr
KsInAddrSet (abstract class)	ks/inaddrset.h	–
KsIntValue (class)	ks/value.h	libks
KsIntVecValue (class)	ks/value.h	libks
KsList (class)	ks/list.h ks/list_impl.h	–
KsProjProps (class)	ks/props.h	libks
KsProjPropsHandle (typedef)	ks/props.h	–
KsPtrHandle (class)	ks/handle.h ks/handle_impl.h	–
KsResult (class)	ks/register.h	libks
KsServerBase (abstract class)	ks/serverbase.h	libkssvr
KsServerDesc (class)	ks/register.h	libks
KsSimpleInAddrSet (class)	ks/inaddrset.h	libkssvr
KsSimpleServer (abstract class)	ks/simpleserver.h	libkssvr
KsSingleValue (class)	ks/value.h	libks
KsSingleVecValue (class)	ks/value.h	libks
KssSimpleCommObject (class)	ks/svrsimpleobjects.h	libkssvr
KssSimpleDomain (class)	ks/svrsimpleobjects.h	libkssvr
KssSimpleDomainIterator (class)	ks/svrsimpleobjects.h	libkssvr

KssSimpleVariable (class)	ks/svrsimpleobjects.h	libkssvr
KsString (class)	ks/string.h	libks
KsStringValue (class)	ks/value.h	libks
KsStringVecValue (class)	ks/value.h	libks
KsTicketConstructor (typedef)	ks/serverbase.h	—
KsTime (class)	ks/time.h	libks
KsTimeValue (class)	ks/value.h	libks
KsTimeVecValue (class)	ks/value.h	libks
KsTimerEvent (abstract class)	ks/event.h	libks
KsUIntValue (class)	ks/value.h	libks
KsUIntVecValue (class)	ks/value.h	libks
KsValue (abstract class)	ks/value.h	libks
KsValueHandle (typedef)	ks/value.h	—
KsVarCurrProps (class)	ks/props.h	libks
KsVarProjProps (class)	ks/props.h	libks
KsVoidValue (class)	ks/value.h	libks
KsXdrAble (class)	ks/xdr.h	libks
KsXdrUnion (abstract class)	ks/xdr.h	libks
PltAssoc (class)	plt/assoc.h	libplt
PltArray (class)	plt/array.h plt/array_impl.h	libplt
PltArrayed (abstract class)	plt/container.h plt/container_impl.h	libplt
PltArrayIterator (class)	plt/array.h plt/array_impl.h	libplt
PltArrayHandle (class)	plt/handle.h plt/handle_impl.h	libplt
PltCerrLog (class)	plt/log.h	libplt
PltContainer (abstract class)	plt/container.h plt/container_impl.h	libplt
PltBidirIterator (abstract class)	plt/container.h plt/container_impl.h	libplt
PltHandleContainer (abstract class)	plt/container.h plt/container_impl.h	libplt
PltHandleIterator (abstract class)	plt/container.h plt/container_impl.h	libplt
PltHashIterator (class)	plt/hashtable.h plt/hashtable_impl.h	libplt

PltHashTable (class)	plt/hashtable.h plt/hashtable_impl.h	libplt
PltIterator (abstract class)	plt/container.h plt/container_impl.h	libplt
PltIListIterator (class)	plt/list.h plt/list_impl.h	libplt
PltList (class)	plt/list.h plt/list_impl.h	libplt
PltIList (class)	plt/list.h plt/list_impl.h	libplt
PltKeyHandle (class)	plt/hashtable.h plt/hashtable_impl.h	libplt
PltKeyPtr (class)	plt/hashtable.h plt/hashtable_impl.h	libplt
PltListIterator (class)	plt/list.h plt/list_impl.h	libplt
PltLog (abstract class)	plt/log.h	libplt
PltLogStream (class)	plt/logstream.h	libplt
PltNtLog (class)	plt/log.h	libplt
PltPQIterator (class)	plt/priorityqueue.h plt/priorityqueue_impl.h	libplt
PltPriorityQueue (class)	plt/priorityqueue.h plt/priorityqueue_impl.h	libplt
PltPtrComparable (class)	plt/comparable.h	libplt
PltPtrHandle (class)	plt/handle.h plt/handle_impl.h	libplt
PltString (class)	plt/string.h	libplt
PltSyslog (class)	plt/log.h	libplt
PltTime (class)	plt/time.h	libplt

6.3 Index

L

N

P

S

T

V

X