The portable C client library

kbmMW supports a number of native clients under the codename ‘Spider’.

One of then is a pure C based client which allows just about any environment to access a kbmMW based application server.

The application server must have a traditional TCP/IP server transport defined which use STANDARD streamformat, no compression, no encryption and with VerifyTransfer set to true.

This document will show how to compile the C client library and how to use it.

Compiling the C library

For Win32

The Win32 library is in the form of a lib file.. The lib file is usually delivered with the kbmMW C client install, but in case it needs recompilation the following steps must be followed.

Prerequisites:

- The kbmMW C client library souce.
  http://www.components4developers.com

- Bloodshed C++ development environment (4.9.9.2 – v5 beta 9.2 or newer)
  http://www.bloodshed.net

Installation of prerequisites:

- Create a directory in which you extract the kbmMW C client library files. e.g: `C:\work`.
- Install Bloodshed C++ development environment **or** Windows Studio .Net with C++ environment according to their installation instructions.

Compilation of the library:

Open the file **makefile.win32** in notepad or any other text editor. In the beginning of the file you will find the following lines:

```
# set compile variables for tools and environment
# use gnu compiler
CC = e:\Dev-CPP\bin\gcc
AR = e:\Dev-CPP\bin\ar
RANLIB = e:\Dev-CPP\bin\ranlib

# use full warnings
CFLAGS = -Wall -O2
```

Modify the CC, AR and RANLIB lines to reflect the correct paths to the compiler, archiver and ranlib executables which are part of the Bloodshed installation. Equally optionally modify the CFLAGS line to contain the optimizations etc. you require by the C compiler.

Save the modifications.

Then open **buildwin32.bat** in a text editor. It contains the following lines:

```
set PATH=%PATH%;e:\dev-cpp\bin


e:\dev-cpp\bin\make -f makefile.win32
```

Modify the e:\dev-cpp\bin path to match the path for your Bloodshed installation. Save the modifications.

Now everything should be ready for compilation of the client library.

To compile, open a command/DOS window and run: **buildwin32.bat**

After it has compiled, a library file will have been created. Use this library file when linking your own applications with the kbmMW C library.

Compiling and running the demo application:

Open the file **makefile.win32** from the sample directory.
The first few lines looks similar to this:

```
# Project: TestSample
# Demo project for kbmMW C client library.

# set compile variables for tools and environment
# use gnu compiler
CC = e:\Dev-CPP\bin\gcc
LIBS=..\libkbmMWClient.lib e:\Dev-CPP\lib\libwsock32.a e:\Dev-CPP\lib\libmsvcrt.a
INCS = -I"include" -I..
OBJS = main.o
EXE = TestSample.exe
```

Modify e:\dev-cpp paths to the correct path matching your Bloodshed Dev/C++ installation. Optionally modify the CFLAGS entry with required optimizations. Save your changes.

Next open **buildwin32.bat** and modify the e:\dev-cpp paths to match your Bloodshed installation. Save your changes.

Open a command/DOS window and run **buildwin32.bat.**
Now an executable should be available. To run it, first start a kbmMW based demo application server and let it listen for requests (default on *port 3000* using *STANDARD* and no compression or encryption).
Then run **TestSample.exe**. It will call the app server 1000 times asking for inventory.

For non Win32 (Linux/Unix and others)

The Linux/Unix library is in the form of an archive file (.a). To generate the archive the following steps must be followed.

Prerequisites:

- The kbmMW C client library souce.
  http://www.components4developers.com
- A Berkley Socket API compliant sockets library for TCP/IP communication.
- A standard ANSI C compliant C compiler (for example GNU C)
  http://directory.fsf.org/devel/compilers/gcc.html

Installation of prerequisites:

- Create a directory in which you extract the kbmMW C client library files.
  e.g: `/home/work`.

Compilation of the library:

Open the file **makefile.unix** in notepad or any other text editor. In the beginning of the file you will find the following lines:

```
# set compile variables for tools and environment
# use gnu compiler
CC = gcc
AR = ar
RANLIB = ranlib

# use full warnings
CFLAGS = -Wall -O2

LIBRARY=libkbmMWClient.a
```

Modify the CC, AR and RANLIB lines to reflect the correct paths to the compiler, archiver and ranlib executables if needed. Usually on a development machine, these are already in the searchpath. Equally optionally modify the CFLAGS line to contain the optimizations etc. you require by the C compiler.

Save the modifications.

Now everything should be ready for compilation of the client library.

To compile, open a terminal and in the kbmMW C client source directory, run: **make –f makefile.unix**

After it has compiled, a library file will have been created. Use this library file when linking your own applications with the kbmMW C library.

Compiling and running the demo application:

Open the file **makefile.unix** from the sample directory.
The first few lines looks similar to this:

```
# Project: TestSample
# Demo project for kbmMW C client library.

CPP  = g++
CC   = gcc
RES  =
OBJ  = main.o $(RES)
LIBS =  -L"lib" ../kbmMWClient.a
INCS =  -I"include" -I..
BIN  = TestSample
CFLAGS = $(INCS)   -fexpensive-optimizations -O3 -march=pentium4
RM = rm –f
```

Modify CC, LIBS and INCS to reflect the directories in which the relevant compiler, libraries and include files exists. Optionally modify the CFLAGS entry with required optimizations. Save your changes.

Open a terminal window and in the sample directory, run: **make –f makefile.unix**

Now an executable should be available. To run it, first start a kbmMW based demo application server and let it listen for requests (default on *port 3000* using *STANDARD* and no compression or encryption).
Then run ./**TestSample**. It will call the app server 1000 times asking for inventory.

The Demo application explained

The source of the demo application looks similar to this:

```
#include <stdio.h>
#include <stdlib.h>
```

Include the relevant kbmMW C client include files.

```
#include "..\kbmMW_Global.h"
#include "..\kbmMW_Variant.h"
#include "..\kbmMW_Exception.h"
#include "..\kbmMW_ClientTransport.h"
#include "..\kbmMW_Client.h"
```

The main application starts here.

```
int main(int argc, char *argv[])
{
   int err;
   int i;
   long l;
```

Its very important that all kbmMW client 'objects' are initialized to KBMMW_NULL at declaration time, or anyway as early as possible before use.

```
   TkbmMWClientTransport *transport = KBMMW_NULL;
   TkbmMWSimpleClient *client = KBMMW_NULL;
   TkbmMWVariant *v = KBMMW_NULL;
   TkbmMWVariant *arg = KBMMW_NULL;

   char *result;
```

The following call must be the first call to the kbmMW C client library. It initialies certain things for the communication layer which is required on Windows platforms.

```
   kbmMW_ClientTransport_Initialize();
```

Next we allocate a client transport 'object' and connects to 'localhost' on port 3000. After trying to connect we check to see if any exceptions has occurred.

```
   transport=kbmMW_NewClientTransport();
   transport->Connect(transport,"localhost",3000);
   kbmMW_IfException(transport) {
      err=kbmMW_ErrorNo(transport);
      printf("Exception: %d - %s\n\r",err,kbmMW_ErrorMessage(transport));
      goto L_Exit;
   }
```

If no exceptions happened, then we allocate a simple client instance which we will use later on for doing the actual requests.

```
   client=kbmMW_NewSimpleClient(transport);
```

As variants are not known to normal pure C and since kbmMW depends on moving its data via variants, we have to emulate those using the kbmMW C client variant functions. We allocate one single new variant, and set its value to the string of KBMMW_INVENTORY.

```
   arg=kbmMW_NewVariant();
   kbmMW_SetVariantAsString(arg,"KBMMW_INVENTORY",-1);
```

Then we make a call to the application server 1000 times. The last argument to the SendRequest call is the variable receiving the result variant. kbmMW C library will automatically deallocate any previously allocated variant value in case v is not KBMMW_NULL before assigning a new value to it.

For each call we check for if an exception has occurred, and if so, react on that. We also show 2 different ways to obtain the status code and text from the call (eg. OK or some error or warning).

```
   for (i=0; i<1000; i++) {
     client->SendRequest(client,"KBMMW_INVENTORY","kbmMW_1.0","GET SYNTAX ABSTRACT",arg,&v);
     kbmMW_IfException(client) {
        err=kbmMW_ErrorNo(client);
        printf("Exception: %d - %s\n\r",err,kbmMW_ErrorMessage(client));
        goto L_Exit;
     }
     if (client->IsError(client)) {
        printf("Error from server: %d - %s\n\r",client->GetStatusCode(client),
                                                client->GetStatusText(client));
        goto L_Exit;
     }

     // Print result.
     printf("Result:\n\r StatusCode=%d (%s)\n\r",client->ResponseStream->StatusCode,
                                                 client->ResponseStream->StatusText);
```

Then we get the returned result and converts it to a string that we can show.

```
     kbmMW_GetVariantAsString(v,&result,&l);
     printf("Data: %s\n\r",result);
   }
L_Exit:
```

Finally we need to dispose all the allocated variants and ‘objects’. Its extremely important to do this, otherwise we will have a memory leak. In addition we disconnect from the application server, and finalize the transport layer as required by Windows setups.

```
   kbmMW_DisposeVariant(&v);
   kbmMW_DisposeVariant(&arg);
   kbmMW_DisposeSimpleClient(&client);
   transport->Disconnect(transport);
   kbmMW_DisposeClientTransport(&transport);

   kbmMW_ClientTransport_Finalize();

   return err;
}
```

Reference

Exception handling

Header file: kbmMW_Exception.h, kbmMW_Global.h

Type: TkbmMWException

For error codes, please refer to the 'General routines' section.

```
TkbmMWException *kbmMW_NewException(int aErrorNo,
                                    char *aErrorMessage);
```

Create a new exception with the given error number and message. The message string is automatically duplicated, and thus the source string can be discarded at any time. Returns the new exception object, which must be discarded later on using kbmMW_DisposeException.

```
void kbmMW_DisposeException(TkbmMWException **aException);
```

Dispose an existing exception. The exception object variable will automatically be set to KBMMW_NULL after the exception has been disposed.

```
TkbmMWException *kbmMW_RaiseException(TkbmMWException **aException,
                                      int aErrorNo,
                                      char *aErrorMessage);
```

Raises a new exception on the existing exception object variable. Any previous exception object pointed to by the variable (aException) will automatically be discarded. aException must be the address of a valid TkbmMWException * variable.

```
TkbmMWException *kbmMW_ReRaiseException(TkbmMWException **aException,
                                        TkbmMWException *aOrigException);
```

Raises a new exception based on an existing exception. aException will contain the new exception object. If it already referred to an old exception object, that will automatically be discarded before raising the new exception. aOrigException points to the exception that is to be reraised. Notice that after calling this method, two exception objects are in existance, the one referred to by aOrigException, and the new one returned by the method and also set in the aException variable.

Helper definitions:

```
#define kbmMW_DefineException TkbmMWException *E
```

Can be used when defining an exception within a structure. This makes the structure 'exception aware', and other definitions can be used to work on the exception object of the structure.

Eg.

```
struct {
  kbmMW_DefineException;
  int myint;
} mystruct;
```

```
#define kbmMW_IfException(aObject)
                    if (aObject->E)
```

Checks if an exception has been raised on a structure 'object'. It replace an if statement.

Eg.

```
kbmMW_IfException(mystruct) {
   printf("Exception raised!\n");
}
```

```
#define kbmMW_IfNotException(aObject)
                    if (!aObject->E)
```

Checks if an exception has not been raised on a structure 'object'.

Eg.

```
kbmMW_IfNotException(mystruct) {
   printf("No exception raised!\n");
}
```

```
#define kbmMW_ErrorNo(aObject)
                    (aObject->E?(aObject->E->ErrorNo):0)
```

Returns the error number of an exception on the given structure ‘object’. If no exception has been raised on the object, 0 is returned.

Eg.

```
printf(“Errornumber=%d\n”,kbmMW_ErrorNo(mystruct));
```

```
#define kbmMW_ErrorMessage(aObject)
                    (aObject->E?(aObject->E->ErrorMessage):"")
```

Returns the error message (char *) from the exception (if any is defined, else it returns a pointer to an empty string) on the given structure ‘object’.

```
#define kbmMW_Raise(aObject,aErrNo,aMessage)
                    kbmMW_RaiseException(&(aObject->E),aErrNo,aMessage)
```

Raise a new exception on the given structure ‘object’. The message given via aMessage which must be a char *, will automatically be duplicated into the internal exception object. Hence the buffer containing the aMessage can safely be discarded after the call.
If the structure ‘object’ already contain an exception ‘object’, it will be discarded first.

```
#define kbmMW_ReRaise(aObject,aSubObject)
                    kbmMW_ReRaiseException(&(aObject->E),aSubObject->E);
```

Raise the exception stored in the structure ‘object’ given by aSubObject on the structure ‘object’ given by aObject. If aObject already contains an exception, it will first be discarded.

General routines

Header file: kbmMW_Global.h

The kbmMW C client library contains several general routines and definitions which are used internally in the library, but which can also be of use for the developer.

Definitions:

```
#define KBMMW_WINDOWS
```

Is defined if the library detects that a MS Windows C compiler is being used for compiling the library.

```
#define KBMMW_ANSIC
```

Is defined if the library detects that a non MS Windows C compiler is used for compiling the library.

```
#define KBMMW_BOOL unsigned char
```

kbmMW boolean value type.

```
#define KBMMW_TRUE (0==0)
```

kbmMW boolean TRUE value.

```
#define KBMMW_FALSE (!KBMMW_TRUE)
```

kbmMW boolean FALSE value.

```
#define KBMMW_NULL ((void *)0)
```

kbmMW NULL value.

Standard error codes (used/provided by exception handling):

```
#define KBMMW_ERR_EXCEPTION         100000
```

Provided if an undefined exception has occurred in a call to a kbmMW based application server.
An error message explaining the problem is usually provided.

```
#define KBMMW_ERR_ABORT             100001
```

Provided if an abort exception has occurred in a call to a kbmMW based application server.
An error message explaining the reason is usually provided.

```
#define KBMMW_ERR_SERVICENOTAVAIL   100100
```

Provided if a call has been made to an unavailable service.
An error message explaining the problem is usually provided.

```
#define KBMMW_ERR_FUNCNOTAVAIL      100101
```

Provided if an unknown function has been called on a kbmMW based application server.
An error message explaining the problem is usually provided.

```
#define KBMMW_ERR_AUTHFAILED        100200
```

Provided if the call to the application server was not authorized.
An error message explaining the reason is optionally provided.

```
#define KBMMW_ERR_MESSAGETYPE       108000
```

Provided if an unknown message has been received from the kbmMW based application server. Currently not in use.

```
#define KBMMW_SYS_REDIRECT          110000
```

Provided if the kbmMW based application server requests the client to disconnect and reconnect to another application server. The connection string to the new application server is in the error message.

Conversion and misc. functions:

```
long xtoi(char *aPtr);
```

Converts a hexdecimal value, pointed to by aPtr to a long value. aPtr must be a zero terminated string.

```
void itox8(char *aPtr, long aValue);
```

Converts a long value to 8 hexdecimal digits which are placed into the char buffer pointed to by aPtr.

```
void itox4(char *aPtr, long aValue);
```

Converts a long value to 4 hexdecimal digits which are placed into the char buffer pointed to by aPtr.

```
void itox2(char *aPtr, long aValue);
```

Converts a long value to 2 hexdecimal digits which are placed into the char buffer pointed to by aPtr.

```
char *kbmMW_DateTimeToString(struct tm *aDate);
```

Returns a pointer to a static character buffer which contains the conversion of the date/time structure value in the format required by kbmMW. (YYYYMMDDHHNNSS)

```
void kbmMW_StringToDateTime(char *aString, struct tm *aDate);
```

Converts a kbmMW formatted date/time string to a date/time structure. The structure must already have been allocated and a pointer provided to it.

```
void kbmMW_SetString(char **aDest, char *aSource);
```

Set the char * variable pointed to by aDest to a duplicate of aSource. If aSource is NULL, the variable pointed to by aDest will be set to NULL.. It's the reponsibility of the caller to free the character buffer when no longer in use. If the variable pointed to by aDest already contained a string value, it is automatically deallocated before setting the new value.

```
KBMMW_BOOL kbmMW_IsIPAddress(char *aValue);
```

Checks if the character string provided by aValue is a reasonably valid formatted IP address. Returns KBMMW_TRUE if the format is accepted. I.e. xxx.xxx.xxx.xxx

Stream support

The C library adds support for streams, which are dynamically expandable buffers to which data can be appended to and read from.

Header file: kbmMW_Stream.h

Type:

```
typedef struct kbmMWStream TkbmMWStream;
```

Functions:

```
TkbmMWStream *kbmMW_NewStream(long initialSize);
```

Allocates a new stream 'object'. The object must be deallocated using kbmMW_DisposeStream.
An initial size can be set by initialSize. If initialSize is >0 then memory matching the size is automatically pre allocated and ready to be filled with data. Although memory has been preallocated, the stream will appear to be empty until data has been written to it.
When needed, the stream object will automatically expand to contain the data added to it.

```
void kbmMW_DisposeStream(TkbmMWStream *aStream);
```

Deallocates a stream object. All data contained in the object is deallocated and so is the object itself.

```
void kbmMW_WriteStream(struct kbmMWStream *aStream,
                       unsigned char aByte);
```

Appends a byte to the stream. If there is not enough memory space available in the stream, an additional chunk will be allocated. After appending the byte, the current position of the stream is incremented.

```
void kbmMW_WriteStreamBuffer(struct kbmMWStream *aStream,
                             unsigned char *aBytes,
                             long aSize);
```

Append a chunk of data to the stream. The chunk, pointed to by aBytes, will be copied into the stream at the current last position in it. The current position of the stream is incremented by the size of the appended chunk.

```
int kbmMW_ReadStream(struct kbmMWStream *aStream);
```

Read a byte from the current position of the stream. The current position of the stream is automatically incremented.
If no more data was available in the stream, -1 is returned.

```
int kbmMW_ReadStreamBuffer(struct kbmMWStream *aStream,
                           unsigned char *aBytes,
                           long aSize);
```

Read a chunk of data (sized by aSize) into the buffer pointed to by aBytes, from the current position of the stream. The actual number of bytes read and placed in the buffer pointed to by aBytes is returned. That number may very well be different to aSize if fewer bytes of data was available.
If no more data was available in the stream, -1 is returned.

```
long kbmMW_GetStreamSize(struct kbmMWStream *aStream);
```

Returns the number of bytes currently stored in the stream.

```
long kbmMW_GetStreamOfs(struct kbmMWStream *aStream);
```

Returns the current position in the stream.

```
long kbmMW_GetRemainingStreamSize(struct kbmMWStream *aStream);
```

Returns the number of bytes remaining in the stream calculated from the current position to the end of the stream.

```
void kbmMW_ClearStream(struct kbmMWStream *aStream);
```

Clear all data in the stream. The stream object is still available and can be written to again to add data to the stream.

```
void kbmMW_SizeStream(struct kbmMWStream *aStream,
                      long aSize);
```

Set the size of the stream. This automatically clears out the contents of the stream and the stream will appear empty until data has been written to it.

Variant support

Header file: kbmMW_Variant.h

Type: TkbmMWVariant

As variants is an unknown concept in pure C, a variant emulation library has been developed. A variant is a type that can contain one of many different native datatypes. This library supports variants of type Null, String, OleStr, Int, Byte, Bool, ShortInt, SmallInt, LongWord, Date, Single, Double, Decimal, Binary and array of variants.
As C do not have native types for all the supported variant types, several functions have been provided to make it possible to assign and extract C native datatypes to and from the variant variables.

```
void kbmMW_DisposeVariantContents(TkbmMWVariant *aVariant);
```

Dispose the contents of a variant, but do not dispose the variant object itself. The variant will have an undefined value after calling this function, and its value should not be used until a new value has been assigned to it.

```
void kbmMW_DisposeVariant(TkbmMWVariant **aVariant);
```

Dispose the variant object completely. The variant variable pointed to by aVariant will be set to KBMMW_NULL after the variant has been disposed.

```
TkbmMWVariant *kbmMW_NewVariant();
```

Generate a new variant object with an undefined contents. Do not obtain values from the variant until a value has been set by one of the setters.
The variant must be disposed of after use by using kbmMW_DisposeVariant.

```
TkbmMWVariant *kbmMW_NewVariantAsType(int aVarType);
```

Generate a new variant object of an defined type. The contents of the variant may be undefined depending on the type. The variant must be disposed of after use by using kbmMW_DisposeVariant.

```
TkbmMWVariant *kbmMW_NewVariantArray(int aSize, int aVarType);
```

Generates a new variant object containing an array of variants of the specified type aVarType. The elements of the array may be undefined depending on the type, and thus should be set using one of the variant setters.
The variant must be disposed of after use by using kbmMW_DisposeVariant.

```
void kbmMW_AssignVariant(TkbmMWVariant *aSourceVariant,
                         TkbmMWVariant *aDestVariant);
```

Assign the contents of the aSourceVariant to the aDestVariant. If aDestVariant already contains data, the data will automatically be disposed of. aDestVariant must already be an existing allocated variant.

```
TkbmMWVariant *kbmMW_CopyVariant(TkbmMWVariant *aVariant);
```

Returns a duplicate of the provided variant. The duplicate must be disposed of after use, using kbmMW_DisposeVariant.

```
TkbmMWVariant *kbmMW_GetVariantArrayElement(TkbmMWVariant *aVariant,
                                            int aElement);
```

Returns a pointer to the variant at index aElement (0..n) in the given variant (aVariant) which must have been defined as a variant array.

```
TkbmMWVariant *kbmMW_SetVariantArrayElement(TkbmMWVariant *aVariant,
                                            int aElement,
                                            TkbmMWVariant *aValue);
```

Set the value of an element (0..n) in a variant array to a value.
The provided aValue variant will be duplicated into the place in the variant array, given by aVariant, indexed by aElement.

```
int kbmMW_GetVariantArraySize(TkbmMWVariant *aVariant);
```

Returns the size of a variant array. If the provided variant is not an array, -1 will be returned.

```
void kbmMW_SetVariant(TkbmMWVariant **aDest, TkbmMWVariant *aSource);
```

Disposes of the existing variant pointed to by aDest, and points aDest to a new copy of the aSource variant.

```
KBMMW_BOOL kbmMW_SetVariantAsNull(TkbmMWVariant *aVariant);
```

Set an existing variant variable to the NULL value. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsType(TkbmMWVariant *aVariant,
                                  const int aType,
                                  void *aData,
                                  long aSize);
```

Set an existing variant variable to a specified variant type given the provided data pointed to by aData of the size given by aSize. If it already contained other variant data, those will be automatically disposed of.
This function is used by all the kbmMW_Setxxxx type functions, and its recommended to use those.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

Eg. To set to NULL:

```
kbmMW_SetVariantAsType(aVariant,kbmMWvarNull,NULL,-1);
```

```
KBMMW_BOOL kbmMW_SetVariantAsString(TkbmMWVariant *aVariant,
                                    char *aValue,
                                    long aSize);
```

Set an existing variant variable to contain a string value. If it already contained other variant data, those will be automatically disposed of.
aValue points to the character buffer containing the string. If aSize is <=0 then aValue must be a zero terminated string. If aSize>0 then aValue do not have to be zero terminated, and in fact the string can contain multiple binary values incl. Ascii zero. The variant will have type kbmMWvarString.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

Eg.

Setting with a zero terminated string:

```
TkbmMWVariant *aVariant = KBMMW_NULL;

aVariant=kbmMW_NewVariant();
kbmMW_SetVariantAsString(aVariant,”This is a zero terminated string”,-1);
…
kbmMW_DisposeVariant(&aVariant);
```

Setting with a buffer and length:

```
TkbmMWVariant *aVariant = KBMMW_NULL;
Char Buffer[5];

Buffer[0]=’A’;
Buffer[1]=’B’;
Buffer[2]=0;
Buffer[3]=’D’;
Buffer[4]=’E’;

aVariant=kbmMW_NewVariant();
kbmMW_SetVariantAsString(aVariant,Buffer,5);
…
kbmMW_DisposeVariant(&aVariant);
```

```
KBMMW_BOOL kbmMW_SetVariantAsOleStr(TkbmMWVariant *aVariant,
                                    char *aValue,
                                    long aSize);
```

Same as kbmMW_SetVariantAsString except that the variant type is set to be kbmMWvarOleStr which can contain 16 bit Unicode/Widestring data. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsInt(TkbmMWVariant *aVariant, int aValue);
```

Set the variant variable to an integer value (max 32 bit) . The variant will have type kbmMWvarInt. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsByte(TkbmMWVariant *aVariant,
                                  unsigned char aValue);
```

Set the variant variable to a byte value (8 bit). The variant will have type kbmMWvarByte. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsBool(TkbmMWVariant *aVariant,
                                  KBMMW_BOOL aValue);
```

Set the variant variable to a boolean value. The variant will have type kbmMWvarBoolean. Use the constants KBMMW_FALSE and KBMMW_TRUE (found in kbmMW_Global.h), or the result of a boolean expression. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsShortInt(TkbmMWVariant *aVariant,
                                      short aValue);
```

Set the variant variable to a short int (16 bit) value. The variant will have type kbmMWvarShortInt. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsSmallInt(TkbmMWVariant *aVariant,
                                      short aValue);
```

Set the variant variable to a small int (16 bit) value. The variant will have type kbmMWvarSmallInt. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsLongWord(TkbmMWVariant *aVariant,
                                      long aValue);
```

Set the variant variable to a long word (32 bit signed) value. The variant will have type kbmMWvarLongWord. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsDate(TkbmMWVariant *aVariant,
                                  struct tm aValue);
```

Set the variant variable to a date/time value. The variant will have type kbmMWvarDate. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsSingle(TkbmMWVariant *aVariant,
                                    float aValue);
```

Set the variant variable to a floating point single value. The variant will have type kbmMWvarSingle. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsDouble(TkbmMWVariant *aVariant,
                                    double aValue);
```

Set the variant variable to a floating point double value. The variant will have type kbmMWvarDouble. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsDecimal(TkbmMWVariant *aVariant,
                                     double aValue);
```

Set the variant variable to a decimal value. The variant will have type kbmMWvarDecimal. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsCurrency(TkbmMWVariant *aVariant,
                                      double aValue);
```

Set the variant variable to a decimal value. The variant will have type kbmMWvarCurrency. If it already contained other variant data, those will be automatically disposed of.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_SetVariantAsBinary(TkbmMWVariant *aVariant,
                                    char *aValue,
                                    long aSize);
```

Set the variant variable to a binary value. The variant will have type kbmMWvarBinary. If it already contained other variant data, those will be automatically disposed of.
aValue points to a buffer containing the binary data, and aSize must specify its size even if 0.
If the operation was successful, KBMMW_TRUE is returned, else KBMMW_FALSE.

```
KBMMW_BOOL kbmMW_GetVariantAsNull(TkbmMWVariant *aVariant);
```

Returns KBMMW_TRUE if the variant is NULL, else KBMMW_FALSE. This function can't be used directly on a variant array, but only on its elements.

```
KBMMW_BOOL kbmMW_GetVariantAsType(TkbmMWVariant *aVariant,
                                  const int aType,
                                  void *aData,
                                  long *aSize);
```

Returns the contents of the variant as the given type. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible. This function can't be used directly on a variant array, but only on its elements.
The function is used by most of the kbmMW_Getxxxx getter functions and should normally not be called directly.

```
KBMMW_BOOL kbmMW_GetVariantAsString(TkbmMWVariant *aVariant,
                                    char **aValue,
                                    long *aSize);
```

Returns the contents of the variant as a string. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

This function can't be used directly on a variant array, but only on its elements.

aValue is a pointer to a char pointer which will be updated with the pointer to the resulting string buffer. Any previous existing values of the pointer aValue will be overwritten. It's the responsibility of the caller to free the received buffer after use.

If the variant type is kbmMWvarBoolean, the appropriate string "TRUE" or "FALSE" will be returned.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsOleStr(TkbmMWVariant *aVariant,
                                    char **aValue,
                                    long *aSize);
```

Same as kbmMW_GetVariantAsString.

```
KBMMW_BOOL kbmMW_GetVariantAsInt(TkbmMWVariant *aVariant,
                                 int *aValue);
```

Returns the contents of the variant as an integer. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

Notice that loss of precision is very likely when converting from floatingpoint or larger integer types.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsByte(TkbmMWVariant *aVariant,
                                  unsigned char *aValue);
```

Returns the contents of the variant as a byte (8 bit unsigned) value. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

Notice that loss of precision is very likely when converting from floatingpoint or larger integer types.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsBool(TkbmMWVariant *aVariant,
                                  KBMMW_BOOL *aValue);
```

Returns the contents of the variant as a boolean value. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

If the variant is a numeric type, the boolean result will be KBMMW_TRUE if the numeric value <>0, else KBMMW_FALSE.

If the variant is a string/olestring type, the boolean result will be KBMMW_TRUE if the first character of the string is either t or T, else KBMMW_FALSE.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsShortInt(TkbmMWVariant *aVariant,
                                      short *aValue);
```

Returns the contents of the variant as a short int (16 bit) value. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

Notice that loss of precision is very likely when converting from floatingpoint or larger integer types.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsSmallInt(TkbmMWVariant *aVariant,
                                      short *aValue);
```

Returns the contents of the variant as a small int (16 bit) value. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

Notice that loss of precision is very likely when converting from floatingpoint or larger integer types.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsLongWord(TkbmMWVariant *aVariant,
                                      long *aValue);
```

Returns the contents of the variant as a long word (32 bit) value. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

Notice that loss of precision is very likely when converting from floatingpoint or larger integer types.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsDate(TkbmMWVariant *aVariant,
                                  struct tm *aValue);
```

Returns the contents of the variant as a date/time value. The variant must be of a kbmMWvarDate type for the function to succeed. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarDate`

```
KBMMW_BOOL kbmMW_GetVariantAsSingle(TkbmMWVariant *aVariant,
                                    float *aValue);
```

Returns the contents of the variant as a float/single value. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

Notice that loss of precision is very likely when converting from double type.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsDouble(TkbmMWVariant *aVariant,
                                    double *aValue);
```

Returns the contents of the variant as a double value. If conversion is required, it is done automatically. Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

This function can't be used directly on a variant array, but only on its elements.

Valid conversions are from the following variant types:
`kbmMWvarNull, kbmMWvarBoolean, kbmMWvarByte, kbmMWvarShortInt, kbmMWvarSmallInt, kbmMWvarInteger, kbmMWvarLongWord, kbmMWvarSingle, kbmMWvarDouble, kbmMWvarDecimal, kbmMWvarCurrency, kbmMWvarString, kbmMWvarOleStr (no Unicode conversion done!)`

```
KBMMW_BOOL kbmMW_GetVariantAsDecimal(TkbmMWVariant *aVariant,
                                     double *aValue);
```

Same as kbmMW_GetVariantAsDouble.

```
KBMMW_BOOL kbmMW_GetVariantAsCurrency(TkbmMWVariant *aVariant,
                                      double *aValue);
```

Same as kbmMW_GetVariantAsDouble.

```
KBMMW_BOOL kbmMW_GetVariantAsBinary(TkbmMWVariant *aVariant,
                                    char **aValue,
                                    long *aSize);
```

Returns the contents of the variant as a binary value.
Returns KBMMW_TRUE if a result could be returned and KBMMW_FALSE if the variant was NULL or a conversion was not possible.

This function can't be used directly on a variant array, but only on its elements.

The binary data returned is not a copy, and thus should not be freed by the caller!
If the variant type is a string, or a conversion to a string has earlier been made using kbmMW_GetVariantAsString, then this function will return the same string value as provided by kbmMW_GetVariantAsString.

Client transport

The client transports responsibility is to manage a connection to a kbmMW based application server which is then used by the kbmMW C simple client.

Its designed to use a standard Posix compliant TCP/IP stack as the communication layer. This should make it compatible with most TCP/IP libraries for most platforms.

Header file: kbmMW_ClientTransport.h

Type: TkbmMWClientTransport

Functions:

```
void kbmMW_ClientTransport_Initialize();
```

This function must be called once before any connections is attempted to me made towards a kbmMW application server. On Windows it's a requirement, on non Windows its an option.

```
void kbmMW_ClientTransport_Finalize();
```

This function must be called once before the shutdown of an application. It finalizes ressources used by the sockets library (TCP/IP stack) on Windows. On non Windows its optional.

```
TkbmMWClientTransport *kbmMW_NewClientTransport();
```

Allocates a new client transport object which can later on be used by TkbmMWSimpleClient. The object must be deallocated after use by kbmMW_DisposeClientTransport. No connections are being created at this stage.

```
void kbmMW_DisposeClientTransport(TkbmMWClientTransport **aTransport);
```

Deallocates a client transport object. Its important to disconnect the client transport from the application server by using the Disconnect method of kbmMWSimpleClient before disposing the object in case a connection has been established.
aTransport will contain NULL after the dispose call.

Client

The clients responsibility is to provide an interface to the developer allowing requests to be made against a kbmMW based application server, and returned responses to be handled.

The client itself do not contain the actual transport layer. For that purpose the client transport is used.

Header file: kbmMW_Client.h

Types:

```
struct kbmMWSimpleClient {
   kbmMW_DefineException;

   TkbmMWClientTransport *ClientTransport;

   TkbmMWRequestTransportStream *RequestStream;
   TkbmMWResponseTransportStream *ResponseStream;

   char *UserName;
   char *Password;
   char *Token;
   char *Location;
   int StateID;
   TkbmMWVariant *Data;

   KBMMW_BOOL (*Connect)(struct kbmMWSimpleClient *Self,
                         char *Host,
                         int Port);
   KBMMW_BOOL (*Disconnect)(struct kbmMWSimpleClient *Self);
   KBMMW_BOOL (*SendRequest)(struct kbmMWSimpleClient *Self,
                             char *aServiceName,
                             char *aServiceVersion,
                             char *aFunction,
                             TkbmMWVariant *aArgs,
                             TkbmMWVariant **aResult);
   KBMMW_BOOL (*SendRequestEx)(struct kbmMWSimpleClient *Self,
                               char *aServiceName,
                               char *aServiceVersion,
                               int aStateID,
                               char *aFunction,
                               TkbmMWVariant *aArgs,
                               TkbmMWVariant **aResult);
   KBMMW_BOOL (*ReleaseState)(struct kbmMWSimpleClient *Self,
                              char *aServiceName,
                              char *aServiceVersion);
   KBMMW_BOOL (*ReleaseStateEx)(struct kbmMWSimpleClient *Self,
                                char *aServiceName,
                                char *aServiceVersion,
                                int aStateID);
   KBMMW_BOOL (*IsError)(struct kbmMWSimpleClient *Self);
   KBMMW_BOOL (*IsWarning)(struct kbmMWSimpleClient *Self);
   KBMMW_BOOL (*IsOK)(struct kbmMWSimpleClient *Self);
   int (*GetStatusCode)(struct kbmMWSimpleClient *Self);
   char *(*GetStatusText)(struct kbmMWSimpleClient *Self);
};

typedef struct kbmMWSimpleClient TkbmMWSimpleClient;
```

Functions:

```
TkbmMWSimpleClient *kbmMW_NewSimpleClient(TkbmMWClientTransport
                                          *aClientTransport);
```

Allocates a new TkbmMWSimpleClient 'object'. An already allocated TkbmMWClientTransport 'object' must be provided. The allocated TkbmMWSimpleClient must be disposed using kbmMW_DisposeSimpleClient after use.

```
void kbmMW_DisposeSimpleClient(TkbmMWSimpleClient **aClient);
```

Deallocates a TkbmMWSimpleClient 'object'. aClient will have been set to NULL when this function finishes.
Notice that disposing the client object do not automatically disconnect the client transport from the application server, if it is connected. Hence it's good practice to call the client objects method Disconnect before disposing the client object, unless the client transport will be reused by other TkbmMWSimpleClient instances later on.

Properties and methods:

When a TkbmMWSimpleClient instance/structure has been allocated, several 'properties' and 'methods' are readily available.

```
TkbmMWClientTransport *ClientTransport;
```

Provides access to the client transport which is used for this simple client.

```
TkbmMWRequestTransportStream *RequestStream;
```

Provides access to the internal request stream instance. The request stream is used for building requests to the application server. It should generally not be operated directly.

```
TkbmMWResponseTransportStream *ResponseStream;
```

Provides access to the internal response stream instance. The response stream is used for holding the result of a request received from an application server. It should generally not be operated directly.

```
char *UserName;
```

Provides access to the current username which the client should use when contacting the application server. It can be set at any time using kbmMW_SetString. Its the responsibility of the TkbmMWSimpleClient 'object' to deallocate memory used by UserName.

Eg.

```
TkbmMWSimpleClient *client=KBMMW_NULL;

client=kbmMW_NewSimpleClient(aClientTransport);
kbmMW_SetString(client->UserName,"JENS HANSEN");

// Do the call
…
…

kbmMW_DisposeSimpleClient(&client);
```

```
char *Password;
```

Provides access to the current password which the client should use when contacting the application server. It can be set at any time using kbmMW_SetString. Its the responsibility of the TkbmMWSimpleClient 'object' to deallocate memory used by UserName.

Eg.

```
TkbmMWSimpleClient *client=KBMMW_NULL;

client=kbmMW_NewSimpleClient(aClientTransport);
kbmMW_SetString(client->Password,"SOMEPASSWORD");

// Do the call
…
…

kbmMW_DisposeSimpleClient(&client);
```

```
char *Token;
```

Provides access to the token returned from the application server, after an authentication process have happened where a username and password has been validated and accepted. As result of the validation, the application server usually returns a unique and temporary number, called a token, which identifies the current client/user session.

Until the client disconnects, the client should forward the provided Token instead of Username and Password, by remembering the Token value returned as the result of a call to the application server, and later setting the Token property with the remembered value on subsequent calls to the application server.

```
char *Location;
```

Provides access to a location string, which essentially can contain any string based value. It can be used by the client to tell the application server where the client is located, and thus the value can be used for extra security validation, logging, auditing etc. on the application server. The value can be set at any time using kbmMW_SetString. Its the responsibility of the TkbmMWSimpleClient 'object' to deallocate memory used by Location.

Eg.

```
TkbmMWSimpleClient *client=KBMMW_NULL;

client=kbmMW_NewSimpleClient(aClientTransport);
kbmMW_SetString(client->Location,"SOMEWHERE");

// Do the call
…
…

kbmMW_DisposeSimpleClient(&client);
```

```
int StateID;
```

Provides access to a state identifier. It will be –1 if the last call to the application server was a stateless call…ie. the service called is a stateless service.
In case a stateful service was called, StateID will contain a unique value which should be remembered by the caller. On later calls, the specific stateful service (currently 'owned' by the client) can be accessed by providing the state identifier returned by the application server in the StateID property. Its the responsibility of the developer to remember state id's returned from the application server and later to release the given state id's (except for state id –1).

```
TkbmMWVariant *Data;
```

Provides access to a developer defined variant which can be used to piggy back any type of data on the request calls to the application server. Similarly the application server can optionally return a piggy backed variant along with the result of the call. kbmMW do not use the Data value itself, and thus its fully available for the developers use.

Its the responsibility of the simple client object to deallocate any values contained in the Data property when the simple client object is disposed of.

Eg.

```
TkbmMWSimpleClient *client=KBMMW_NULL;
TkbmMWVariant *variant=KBMMW_NULL;

variant=kbmMW_NewVariant();
KbmMW_SetVariantAsString(&variant,"This is a test",-1);

client=kbmMW_NewSimpleClient(aClientTransport);
client->Data=variant;

// Do the call.
…
…

kbmMW_DisposeSimpleClient(&client);
```

```
KBMMW_BOOL (*Connect)(struct kbmMWSimpleClient *Self,
                      char *Host,
                      int Port);
```

Connects to a specified application server via the client transport provided when allocating the simple client object.

Self is the simple client instance itself.
Host is an IP address or internet server address to the kbmMW application server.
Port is the port number that the kbmMW application server is listening on.

Returns KBMMW_TRUE if connection succeeded, else KBMMW_FALSE.

It is important to disconnect the client if already connected, before trying to connect again.

Eg.

```
TkbmMWSimpleClient *client=KBMMW_NULL;

client=kbmMW_NewSimpleClient(aClientTransport);
if (!client->Connect(client,"127.0.0.1",3000)) {
   printf("Connection failed\n");
}

// Do the call.
…
…

client->Disconnect(client);
kbmMW_DisposeSimpleClient(&client);
```

```
KBMMW_BOOL (*Disconnect)(struct kbmMWSimpleClient *Self);
```

Disconnect a client from the application server.

Self is the simple client instance itself.
Returns KBMMW_TRUE if connection succeeded, else KBMMW_FALSE.

See Connect for an example.

```
KBMMW_BOOL (*SendRequest)(struct kbmMWSimpleClient *Self,
                          char *aServiceName,
                          char *aServiceVersion,
                          char *aFunction,
                          TkbmMWVariant *aArgs,
                          TkbmMWVariant **aResult);
```

Sends request to the application server. The client must already be connected to call this 'method'.

Self is the client instance itself.
aServiceName is a string identifying the name of the service to call.
aServiceVersion is a string identifying the version of the service to call.
aFunction is a string identifying the name of the function to call.
aArgs is a variant containing an argument, or an array of arguments. If no arguments is required for the call KBMMW_NULL can be provided.
aResult is a pointer to a variant that is to hold the result of the call.

The SendRequest call will either result in a result value or an exception.

Either use the exception handline methods described earlier in this document to test if an exception has been raised and to get its message and code, or use the IsError, IsOk, IsWarning, GetStatusCode and GetStatusText methods of the simple client .

SendRequest returns KBMMW_TRUE if it was possible to send the request to the application server, and KBMMW_FALSE if not. The return value do not identify if the result of the request was ok or not.

An application server can return several error codes. Please check the Global routines section for more information.

Eg.

```
TkbmMWSimpleClient *client=KBMMW_NULL;
TkbmMWVariant *result=KBMMW_NULL;
char *resultstring;
long resultstringlength;

client=kbmMW_NewSimpleClient(aClientTransport);
if (!client->Connect(client,"127.0.0.1",3000)) {
   printf("Connection failed\n");
}

// Make the call.
if (!client->SendRequest(client,
                         "KBMMW_INVENTORY",
                         "kbmMW_1.0",
                         "LIST",
                         KBMMW_NULL,
                         &result)) {
   printf("Call to application server not successful. Is it connected?");
   goto L_Exit;
}

// Check for server exception.
if (!client->IsOK(client)) {
   printf("Server signals exception to us. Exceptioncode=%d, Text=%s\n",
          client->GetStatusCode(client),
          client->GetStatusText(client));
   goto L_Exit;
}

// Get result.
kbmMW_GetVariantAsString(result,&resultstring,&resultstringlength);
printf("Result=%s\n",resultstring);

L_Exit:

client->Disconnect(client);
kbmMW_DisposeSimpleClient(&client);
```

Its possible to send a datastream with the call to the server. To do that, one have to access the DataStream element of the RequestStream ancestor before making the call, in the following way:

```
kbmMW_WriteStreamBuffer(client->RequestStream->super->DataStream,
                        somebuffer,
                        somelength);
```

or another of the stream manupulation functions as described in the 'Streams' section.

Similarly one can receive a stream as a result from the application server by accessing the DataStream element of the ResponseStream ancestor after the call in the following way:

```
char *somebuffer;
long *somebufferlength;

kbmMW_ReadStreamBuffer(client->ResponseStream->super->DataStream,
                       &somebuffer,
                       &somelength);
```

```
KBMMW_BOOL (*SendRequestEx)(struct kbmMWSimpleClient *Self,
                            char *aServiceName,
                            char *aServiceVersion,
                            int aStateID,
                            char *aFunction,
                            TkbmMWVariant *aArgs,
                            TkbmMWVariant **aResult);
```

Same as SendRequest except that its possible to send a specific state identifier which do not have to be the same as the one stored within the client object itself.

```
KBMMW_BOOL (*ReleaseState)(struct kbmMWSimpleClient *Self,
                           char *aServiceName,
                           char *aServiceVersion);
```

Release the state of the instance identified by aServiceName, aServiceVersion and the currently set StateID of the client object.

After a stateful service has been called its very important to release the state as soon as the stateful service is not to be used by the client any longer. Not releasing it can result in serious server ressource drain which can lead to the server not being able to serve requests for the specific service, until the application server times the relevant services out (optional server side feature).

Eg.

```
printf("releasing state with state id=%ld\n",client->StateID);
client->ReleaseState(client,"SOMESTATEFULSERVIC","ver1.0");
```

```
KBMMW_BOOL (*ReleaseStateEx)(struct kbmMWSimpleClient *Self,
                             char *aServiceName,
                             char *aServiceVersion,
                             int aStateID);
```

Same as ReleaseState except its possible to provide a state identifier directly in the call.

```
KBMMW_BOOL (*IsError)(struct kbmMWSimpleClient *Self);
```

Returns KBMMW_TRUE if an error exception has been raised on the client after last call to server. Else returns KBMMW_FALSE.

```
KBMMW_BOOL (*IsWarning)(struct kbmMWSimpleClient *Self);
```

Returns KBMMW_TRUE if a warning exception has been raised on the client after last call to server. Else returns KBMMW_FALSE.

```
KBMMW_BOOL (*IsOK)(struct kbmMWSimpleClient *Self);
```

Returns KBMMW_TRUE if the last call to the application server resulted in no errors or exceptions. Else returns KBMMW_FALSE.

```
int (*GetStatusCode)(struct kbmMWSimpleClient *Self);
```

Returns the status code provided from the last call to the application server. If no error or warning was raised, 0 is returned. See the Global routines section for a description of the predefined error codes a kbmMW application server can return (in addition to 0).

```
char *(*GetStatusText)(struct kbmMWSimpleClient *Self);
```

Returns the status text provided from the last call to the application server. If no error or warning was raised, the string “OK” is returned.

Eg.

```
char *s;

s:=client->GetStatusText(client);
printf(“Last status from server was %s\n”,s);
```

This concludes the whitepaper about the kbmMW C client library.

Kim Madsen
Components4Developers