Systems Biology Workbench Delphi/Kylix Programmer's Manual

Herbert Sauro, Mike Hucka, Andrew Finney, Hamid Bolouri

{hsauro,mhucka,afinney,hbolouri}@cds.caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA
http://www.cds.caltech.edu/erato

Principal Investigators: John Doyle and Hiroaki Kitano

December 12, 2004

Contents

1 Introduction

This document describes the Delphi/Kylix binding library for interfacing Delphi-based modules to the Systems Biology Workbench (SBW). A discussion of the SBW and other related issues can be found in the paper by Hucka et al. (2001).

The Delphi SBW binding library is a wrapper around the C SBW DLL (Finney et al., 2002), and in addition, the Delphi library provides a higher-level API to make it easier for developers to work with SBW. Note that the lower-level C API is still accessible via the SBW DLL. This document only describes the higher-level API; for details on the C API, readers are directed to the C API document.

The Delphi SBW binding library is supplied in the form of a non-visual component called `TSBW`. A number of properties and events can be set at design time and a host of non-design time library methods and properties are also supplied. The Delphi library relies on the C/C++ binding interface, `SBWD.DLL` or `libsbw.so` for Linux, which supplies the runtime infrastructure and handles all direct communications to the SBW Broker automatically. Thus, the Delphi component can be viewed as a wrapper around this library.

The purpose of the Delphi library is not only to allow access to the C SBW DLL, but also to simplify and reduce as much as possible the work required to interface software to the SBW environment. This allows developers to concentrate on writing the functional aspects of a module rather than get bogged down in the details of interfacing the module to SBW.

Before describing in detail the higher-level API and design-time interface supplied by `TSBW`, we will first describe a few implementation examples in Section 2.

1.1 Services, Methods and Functions

When a Delphi-based SBW module provides functionality, it does so by way of services, methods and functions. Services are simply a means by which one can categorize methods and functions into logical groupings. For example, a maths module might categorize a number of services under the titles, `trigonometric`, `logarithmic` and `hyperbolic`. Within each service one would provide the appropriate functions for each service. Thus we might add the functions `sin` and `cos` to the `trigonometric` service and `ln` and `Log2` to the `logarithmic` service.

Note that in Object Pascal there is a natural language distinction between methods which return results and methods which do not. The latter are called *functions*, and this distinction is reflected in the SBW Delphi library. In the SBW C library, if a method does not return a result, the programmer is still obliged to return nil value to the caller. If this isn't done, errors will occur during runtime. The Object Pascal approach of separating methods from functions avoid this frequent pitfall.

There is an additional categorization called service categories (not to be confused with services), but discussion of this concept is beyond the scope of this short document.

2 Work-through Example — Providing a Service

The simplest SBW module is one that provides services to other modules. The following example illustrates a simple math service provider, that is, a module which provides maths services to the SBW environment.

Our simple maths service provider will supply one service which in turn will provide the two basic trig functions, `sin` and `cos`. The following description applies to both the Windows and Linux platforms.

1. **Start a new Windows project.** From the Delphi File menu, select New and choose

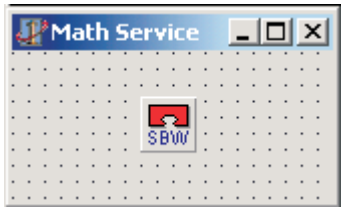

Figure 1: *Delphi Form with TSBW Component*

the window application project. This will create a basic Delphi application containing a simple form. Although our service module does not need any visual presence when running, we can use the form to host the SBW component. This makes it convenient to make changes to the SBW component at design time.

2. **Place a TSBW Component on the Form.** Go to the component palette and select the SBW component and place a copy on the application main form.

3. **Set Properties** Set the following properties on the SBW component:

   AppVisible Set to false, so that the service provider is invisible at runtime.

   AutoConnect Set to true, so that the service provider automatically connects to the SBW broker when it starts.

   DisplayName Set the display name to something meaningful.

   ModuleName Give the module a descriptive name. You might use `mathService`, for example.

   Name Set the name of the component; `sbw` is a convenient name.

4. **Set Events**:

   OnRegister Event: If a module is to supply services and methods, it must first register them with the Broker. The `OnRegister` event is automatically called at the start of a program run and the code that resides in the event is responsible for registering all services, methods and functions supplied by the module. In our simple example we supply one service and two functions. The following code should be pasted into the `OnRegister` event procedure:

```
sbw.addService  ('trig', 'Trig Service', 'Trignometric Functions');
sbw.addFunction ('trig', MySin  'double sin(double)', 'sine of radian angle');
sbw.addFunction ('trig', MyCos, 'double cos(double)', 'cosine of radian angle');
```

   This code uses two `TSBW` methods, `addService` and `addFunction`. `addService` takes three arguments, the name of the service, a long description of the service and a help string.

   The second call, `addFunction`, registers the function associated with the service. The first argument is the name of the service, in this case, `trig`. The second argument is a pointer to the function which will carry out the required operation, `sin, cos` or whatever.

   The third argument is the function signature and gives details of the return type, argument types and function name. For example, `double sin(double)` means that the name of this function is `sin`, it takes a double argument and returns a double argument. The final argument is the help string, which may be empty if necessary. All service functions (i.e., the 2nd argument to `addFunction`) must have the following function signature:

```
function (ds : TSBWDataStream) : SBWDataBlockWriter of object;
```

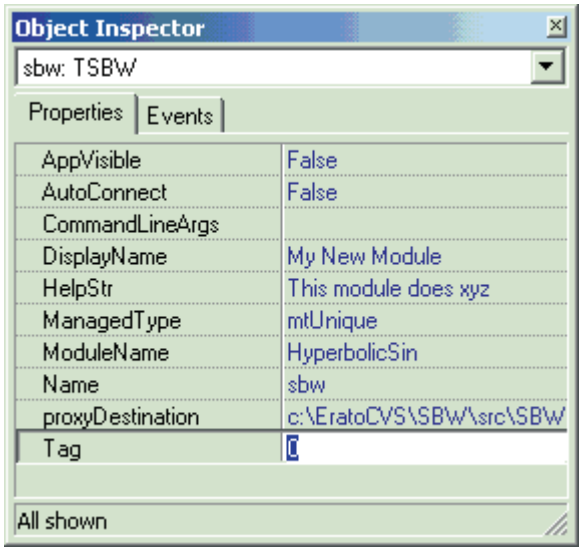

Figure 2: *Properties List for TSBW Component*

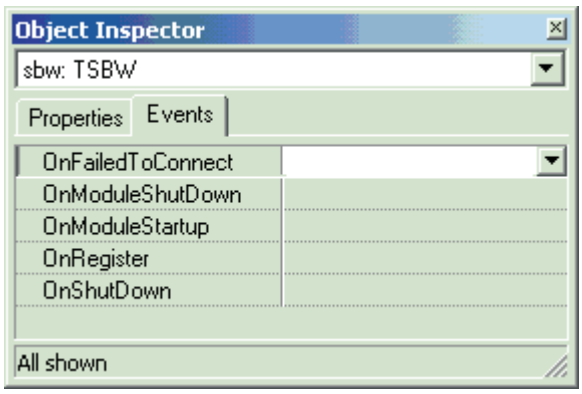

Figure 3: *Event List for TSBW Component*

2.1 Implementing SBW Functions

Function implementations must be supplied to complete the registration.

Recall that the function handler has the signature

```
function (ds : TSBWDataStream) : SBWDataBlockWriter of object;
```

This is the function that will be called to carry out the requested operations, e.g. compute sin.

The argument passed in the function handler is the data stream sent from the SBW broker and will contain the arguments you will need to carry out the function request. In this example, the data stream should contain a single double value.

As an example, here is the implementation of the `sin` function:

```
function TForm1.MySin (ds : TSBWDataStream) : SBWDataBlockWriter;
begin
  result := ds.pack (Sin (ds.getScalar ()));
end;
```

The data stream is of object type `TSBWDataStream` which has a number of useful methods. The data stream will contain the data necessary to carry out the computation. In this case we

expect one double value in the data stream. To extract the various arguments from the data stream, use the `getXXX()` methods. The specific method `getScalar()` will attempt to extract a double or integer value from the data stream. Any errors that occur during this extraction will be handled automatically by routing the error back to the Broker and on to the original caller for processing. If you need to do any of your own clean up, make sure you wrap the entire code in a try/finally block.

As well as accepting data as arguments, SBW function handlers must also be able to return many different kinds of data, including complex data structures, such as nested lists and others. To assist in this process, there is a helper routine associated with the DataStream object which can package arbitrary data into a form suitable for returning to the caller. This helper routine is called `pack` and takes an open variant array. In the example shown here, we are returning a single double value. The use of a variant array allows us to return any SBW data type, including multiple data items. Here is an example of a more elaborate call:

```
ds.pack ([True, 'Hello', 3.1415])
```

The other function, `cos` is handled in a similar way:

```
function TfrmMain.MyCos (ds : TSBWDataStream) : SBWDataBlockWriter;
begin
  result := ds.Pack (Cos (ds.getScalar ()));
end;
```

The implementation is now complete and the module may be deployed.

2.2 Work-through Example — Using a Remote Service

Using a remote service is very simple. As in the last example, start a new project and drag the SBW component onto the main form. Set the properties of the component as before except call the component `mathDriver`.

Place a button on the form and add the following code to the button **OnClick event**:

```
moduleId := sbw.getModuleInstance ('mathService');
mySin := sbw.getMethodObj (moduleId, 'trig', 'sin');
Answer := mySin.Call (3.4).getDouble;
```

Note the `getMethodObj` call. This function takes three arguments: the name of the module, the service we wish to use and the name of the method we wish to call. `getMethodObj` returns an object representing the method. This object in turn has the method `call()` which can be used to invoke the method.

One point worth stressing is the subtleties involved in using the call `getModuleInstance`. Unless one is careful, it is possible to inadvertently cause the generation of multiple instances of the module. The reader is strongly advised to refer to the section on module startup issues (Section 4).

In later versions of the Delphi library, there will be an option to use automatically generated module proxies which will make the calling of remote methods even simpler.

3 SBW Data Type Support

SBW communicates data between modules by way of messages encoded in binary. Each message is made up of a header followed by a payload. The header is used by the SBW Broker to correctly route the message from the originating module to the destination module. The payload contains the data that the destination module needs to carry out its operation; in turn, the destination module may return a message containing another payload to the originating module.

Although SBW places no restrictions on the structure of the payload, there are certain predefined payload types which are defined to enable modules to easily exchange data. In the current release, seven data types have been defined and are supported by all the language bindings. These predefined data types include: byte, integer, double, boolean, string, array and list. A detailed description of these types is given by (Finney et al., 2002). Note that it is also possible to send raw binary data from one module to another.

When a module calls a method in a remote module, the remote module will receive the payload containing the arguments that were passed from the sending module. One of the most important aspects of programming modules for SBW is the ability to manipulate the payload as it is received by the remote module. To assist in this work, the remote method receives a so-called `TSBWDataStream`. This is an object which represents the payload from the sending module.

In this section, we describe how to extract data from the data stream and also how to return data to the caller by using the packing methods.

3.1 Complex Type

Delphi does not have a native complex type, therefore the Delphi binding library comes with it's own complex type, `TSBWComplex`. This type is defined as:

```
TSBWComplex = record r, i : double; end;
```

A variety of methods are available that operate on the complex data type these include:

```
function complex (r, i : double) : TSBWComplex;

function geTSBWComplexRealPart (const z : TSBWComplex) : double;
function geTSBWComplexImagPart (const z : TSBWComplex) : double;

function complexAdd (a, b : TSBWComplex) : TSBWComplex;
function complexSub (a, b : TSBWComplex) : TSBWComplex;
function complexMult (a, b : TSBWComplex) : TSBWComplex;
function complexDiv (a, b : TSBWComplex) : TSBWComplex;
function complexNeg (a : TSBWComplex) : TSBWComplex;
function complexPow (a, b : TSBWComplex) : TSBWComplex;
function complexSqrt (const z : TSBWComplex) : TSBWComplex;
function complexSqr  (const z : TSBWComplex) : TSBWComplex;
function complexAbs  (const z : TSBWComplex) : double;
function complexPhase(const z : TSBWComplex) : double;

function complexSin (const z : TSBWComplex) : TSBWComplex;
function complexCos (const z : TSBWComplex) : TSBWComplex;
function complexTan (const z : TSBWComplex) : TSBWComplex;
function complexExp (const z : TSBWComplex) : TSBWComplex;
function complexLn  (const z : TSBWComplex) : TSBWComplex;
function complexLog10 (const z : TSBWComplex) : TSBWComplex;
function complexLog2 (const z : TSBWComplex) : TSBWComplex;

function complexLessThan (a, b : TSBWComplex) : boolean;
function complexGreaterThan (a, b : TSBWComplex) : boolean;
function complexLessThanOrEqual (a, b : TSBWComplex) : boolean;
function complexGreaterThanOrEqual (a, b : TSBWComplex) : boolean;

function complexNotEquals (a, b : TSBWComplex) : boolean;
function complexEquals (a, b : TSBWComplex) : boolean;
```

3.2 Extracting Arguments from the Data Stream

As previously mentioned, arguments passed to a handler come via a `TSBWDataStream` object. Not only does this object hold the argument data, it also includes a variety of helper methods to assist the programmer. In particular, there are two groups of methods: the extraction methods and the packing methods.

The extraction methods are used to extract the individual arguments sent down the data stream by the remote module. Currently there are seven extraction methods, each one tailored to extract a specific data type. Note that the arguments in the data stream are in the specific order given by the initial remote call. For example, assume that a remote call issues the call, `call (1, 3.4, "hello")` with the arguments being an integer followed by a float followed by a string. At the receiving end, the data stream will contain the arguments in the same order. Thus one should first attempt to extract an integer, followed by a float, followed by a string.

An exception is raised if the data in the data stream is not of the expected type or there are no more data items remaining to extract.

If `ds` is the data stream object, then the following extraction methods are available:

```
ds.getByte()
ds.getBoolean()
ds.getInteger()
ds.getDouble()
ds.getScalar()
ds.getComplex()
ds.getString()
ds.getArray()
ds.getList()
```

If for some reason you are uncertain of the data type to extract, then you can use the `isXXX` methods to test for specific data types. The following calls can be used to check what the next type is in the data stream object:

```
ds.isByte()    : boolean;
ds.isBoolean() : boolean;
ds.isInteger() : boolean;
ds.isDouble()  : boolean;
ds.isComplex() : boolean;
ds.isString()  : boolean;
ds.isArray()   : boolean;
ds.isList()    : boolean;
```

The extraction methods `getByte`, `getInteger`, `getDouble`, `getComplex`, `getString` and `getBoolean` will return byte, integer, double, string and boolean, respectively. The method `getScalar` returns a double, but if the argument is an integer then it will cast it and return it as a double. The methods `getArray` and `getList` require more explanation and are the subjects of the following two subsections.

3.2.1 Extracting Lists

The `getList` method attempts to extract a list from the data stream object. If successful, it returns an object of type `TSBWList`. Note that a list is a heterogenous indexable collection of SBW data types. Under the Delphi implementation, a `TSBWList` is derived from a simple `TList` type and is used to store a list of objects of type `TSBWListItem`. Since the elements of a `TSBWList` can be any valid SBW type, the items of a `TSBWList` have a data type associated with them to allow one to identify the particular type in each list item.

The following code illustrates how to construct a list of the form `[6, 3.14, 'hello']`:

```
var Lt : TSBWList;

Lt := TSBWList.Create ([6, 3.14, 'hello']);
```

Since a list is indexable, to extract a list element one can use the syntax:

```
listItem := Lt[2];
```

To check the data type of a list item, use the method `getType`; this returns a data type indicating the type stored in a list element. Here is an example:

```
if Lt[2].getType = dtDouble then
  value := Lt[2].getDouble;
```

Once a type has been identified, you can use the `getXXX` methods to extract the particular value. Alternatively, you can use the `IsXXX` test methods to determine the type, as in the following example:

```
if Lt[2].isDouble then
```

3.2.2 Extracting Arrays

Arrays are very much like lists except that they are homogeneous: whatever type they hold, every element of the array is of that type.

3.3 Returning Results to the Remote Caller

In addition to extracting the arguments upon entry to a method, it is also necessary at times to return results to the caller. This can be achieved using the packing methods from the data stream object.

To return a simple double value, you can use code such as the following:

```
result := ds.pack (3.1415926);
```

To return a number of results, enclose the values in a variant array, as in:

```
result := ds.pack ([1,2,3,4]);
```

4 Events Provided with the SBW Delphi Component

The Delphi SBW component exposes certain SBW notifications in the form of events. These events are available from the event table of the object inspector.

The following events are available:

OnRegister This event is fired when it is time to register any services, methods and functions. If a module does not provide any of these, then this event can be ignored.

OnModuleStartup This event is fired when a remote module is starting up. The event has the following procedure declaration:

```
procedure TfrmMain.sbwModuleShutDown(Sender: TObject; Id: Integer);
```

The argument `Id` is the identification handle of the module starting up. This event is useful if a module needs to carry out updating activities in response to a module startup.

OnModuleShutdown This event is fired when a remote module is shutting down. The event has the following procedure declaration:

```
procedure TfrmMain.sbwModuleShutDown(Sender: TObject; Id: Integer);
```

The argument `Id` is the identification handle of the module shutting down. This event is useful if a module needs to carry out updating activities in response to a module shutdown.

OnFailedToConnect This event is fired if a module failed to connect to SBW when it starts up.

OnShutDown This event is fired when the current module shuts down.

5 Module Startup Issues

The method call `getModuleInstance()` is one of the most important but also potentially misunderstood calls in the SBW API.

In the most general sense, `getModuleInstance()` is responsible for returning a module identification handle to the module we wish to interact with. Two criteria determine how the method `getModuleInstance()` acquires the module identification handle. These two criteria are:

- Management type—**UNIQUE** or **SELF_MANAGED**
- Whether the module is 'registered' or not using the `-sbwregister` command line option on a module

The purpose of 'registration' is to inform the SBW Broker of the location of, and services provided by, a particular module. This information, once set, is persistent from one Broker session to another.

The other criterion is the management status of the module. This determines the number of possible separate instances of module that can co-exist at any one time. Put simply, if the management status of a module is set to UNIQUE, then this means only a single instance of this module can be running at any one time. Clearly this has an effect on the outcome of a call to `getModuleInstance()`.

If `getModuleInstance()` is called on a module with a UNIQUE management status, then the following two outcomes are possible:

1. If the module in question is already running, then `getModuleInstance()` simple returns the identification handle for the running module.
2. If the module is not running **and** the module is 'registered', then the SBW Broker will attempt to start up an instance of this module. If the module is not 'registered' then the Broker will fail to acquire an identification handle for the module.

If `getModuleInstance()` is called on a module with a SELF_MANAGED management status, and the module is 'registered', the SBW Broker will attempt to start up a new instance of the module. Thus, repeated calls to `getModuleInstance()` will result in multiple instances of the module being started up, and each instance will be identified by a unique identification handle as returned by `getModuleInstance()`.

If you choose to use `getModuleInstance()` on a self-managed module, then it is **your** responsibility to shut-down the module once you have finished using it. If you do not this then the module will remain running in memory even though the original module which started the instance has long since closed. In a later release of this library, the use of automatically-generated proxies will perform this operation for you when you close your module. However, there may be instances when you will purposely wish to leave a remote module running, in which case, there is a need to close the module explicitly.

6 API Reference

The following API list is divided into four broad groups.

- Accessing and using services in another module
- Implementing and registering services from a module
- SBW environment exploratory calls
- Miscellaneous and house-keeping calls

6.1 Accessing and Using Services in Another Module

function getModuleInstance(moduleName : string) : integer

Obtains a Reference to a Remote Module. The call returns an integer Id.

```
Id = sbw.getModuleInstance ('mathModule')
```

The effect of this call depends on the management type of the module.

If the module is managed as **SELF_MANAGED**, then a new instance of the module is started up and the method returns a unique module Id for the new instance.

If the module is managed as **UNIQUE**, then if an existing instance is available the Id for this module is returned. Otherwise, a new instance of the module is started up and a new Id is returned.

Note that in order for SBW to automatically start up a module, the module must have been previously registered using the `-sbwregister` command line option.

Registered modules can be inspected using the Inspector module.

6.1.1 Calling Methods and Functions in Remote Modules

function call (moduleId, serviceId, methodId : integer, args : array of variant) : TSBWCallResult;

Low-level API call. Calls a method in a remote module and returns the result to the caller in the form of a `TSBWCallResult`. Note that this is a blocking call.

```
res = sbw.call(MathId, TrigId, sinId, [30.4])
```

function getMethodObj (moduleId : integer, serviceName : string, methodName : string) : TSBWCallObject;

Returns the Method Calling Object for a given Module Id, Service Id and Method Name.

This is the high-level equivalent of the above.

```
MySin = sbw.getMethodObj (moduleId, 'trig', 'sin')
```

function call (arguments : array of variant) : TSBWCallResult

Calls the method associated with the CallObject and returns the result to the caller. Note that this is a blocking call.

```
MySin := sbw.getMethodObj (moduleId, 'trig', 'sin');

res = MySin.Call([30.4])
```

6.2 Implementing and Registering Services in a Module

6.2.1 Registration Calls

function addService (moduleName : string) : integer;

Adds a service to the current module and return a service Id.

```
Id = sbw.addService('bifurcation')
```

function addMethod(moduleName, servceName : string; method : MethodPtr; helpStr : string) : boolean;

Add a method to a particular service in the current module. Returns true if successful, false if the call fails. The methodPtr must be of type `MethodPtr = procedure (ds : TSBWDataStream) of object;`

```
Ok = sbw.addMethod('trig', 'doIt', MethodPtr, 'void doIt (int)',
'doIt with integer argument')
```

funnction addFunction (moduleName, serviceName : string; func : FunctionPtr; helpStr : string) : boolean;

Adds a function name to a particular service in the current module. The `func` argument must be of type

```
FunctionPtr = function (ds : TSBWDataStream) : SBWDataBlockWriter of
object;

Ok = sbw.addFunction('trig', 'sin', FunctionPtr, 'double sin
(double)', 'Compute sin of radian angle')
```

6.3 SBW Exploratory Calls

function getModuleList() : TStringList

Returns a list containing the names of the currently connected
modules. Note it is the responsibility of the caller to free the stringlist once the caller no longer needs the list.

```
list = sbw.getModules()
```

function getServiceList(moduleName : string) : TStringList

Returns a list of supported services for a given module. Note it is the responsibility of the caller to free the stringlist once the caller no longer needs the list

```
list = sbw.getServices('MyModule')
```

function getMethodList(moduleName, serviceName : string) : TStringList;

Returns a list of available method names for a particular module and service. Note it is the responsibility of the caller to free the stringlist once the caller no longer needs the list

```
list = sbw.getMethods('MyModule', 'trig')
```

function getMethodSignature (moduleName, serviceName, methodName : string) : string

Returns a string encoding the return and argument types for a method.

```
sig = sbw.getMethodSignature('TrigModule', 'Trig', 'sin')
```

function getMethodHelpStr(moduleName, serviceName, methodName : string) : string;

Returns the help string, if any, associated with a particular module, service and method.

```
helpStr = sbw.getMethodHelpStr('TrigModule', Trig', 'sin')
```

function getNumModules() : integer

Returns the number of currently connected Modules.

```
n = sbw.getNumModules()
```

function getNumServices(moduleId : integer) : integer

Returns the number of services associated with a module

```
n = sbw.getNumServices(moduleId)
```

function getNumMethods (moduleId, serviceId : integer) : integer

Returns the number of methods and functions associated with a module and service.

```
n = sbw.getNumMethods(moduleId, serviceId)
```

function getModuleName (moduleId : integer) : string;

Returns the name of the module with a given Id

```
name = sbw.getModuleName(moduleId)
```

function getServiceName (moduleId, serviceId : integer) : string;

Returns the name of the service given a module Id and service Id

```
name = sbw.getServiceName(moduleId, serviceId)
```

function getUniqueModuleId (moduleName : string) : integer;

Returns the module handle for the specified module name. This call will only search for modules which are managed as `UNIQUE`.

```
Id = sbw.getModuleId('MyModule')
```

function getServiceId (moduleId : string; serviceName : integer) : integer;

Returns the Service Id for a given Module Id and a Service Name.

```
Id = sbw.getServiceId(3, 'trig')
```

function getMethodId (moduleId, serviceId : integer; methodName : string) : integer;

Returns the Method or Function Id for a given Module Id and Service Id.

```
Id = sbw.getMethodId(ModuleId, ServiceId, 'sin')
```

6.4 Miscellaneous and Housekeeping calls

procedure shutDownModule(moduleId : integer);

Shuts down the module with module Id, `moduleId`. If this call fails to shut down the specified module, then an exception is raised.

```
sbw.shutDownModule(moduleId)
```

function getBrokerVersion () : string

Gets the version string for the currently running broker

```
sbw.getBrokerVersion()
```

function getApiVersion() : string;

Gets the version string for the currently running C Api

```
sbw.getApiVersion()
```

7 Example Code

SBW modules come in a variety of flavours:

- Modules which provide services
- Modules which use other module services
- Modules which provide and use services

7.1 Providing Services

The following example shows how to build a module which provides some form of computational service, in this case a simple set of trigonometric functions. The module is built from only three

procedures, two procedures which define the trig functions and one function that carries out the registration. The procedures which implement method functionality, that is `sin` and `cos`, must have a specific function type, namely

```
function (ds : TSBWDataStream) : SBWDataBlockWriter}
```

The type `TSBWDataStream` is a reference to a data stream object and will contain the arguments required to carry out the computation. In the case of the trigonometric functions, the data stream will contain a single double argument representing the value of the radian angle. `SBWDataBlockWriter` is a simple pointer to an array of bytes and allows returning a variety of different types without having to declare a different function type for each data type we may wish to return. A set of utility functions are available to pack particular data items to and from a `SBWDataBlockWriter`, see below.

In the following example, `sbw` is the name of the client module object.

```
function TForm1.MySin (ds : TSBWDataStream) : SBWDataBlockWriter;
var x, y :
double;
begin
  x := ds.getScalar();
  y = sin (x);
  result := ds.pack (y);
end;

// Or a much shorter version
function TForm1.MyCos (ds : TSBWDataStream) : SBWDataBlockWriter;
begin
  result := ds.pack (cos (ds.getScalar()));
end;

procedure TForm1.OnRegister (Sender: TObject);
begin
  sbw.addService ('TrigService', 'Trig Services' '');
  sbw.addFunction ('TrigService', MySin, 'double sin (double)', 'Sine function');
  sbw.addFunction ('TrigService', MyCos, 'double cos (double)', 'Cosine function');
end;
```

If we refer to the function, `MySin`, we see that the first operation is to extract the double argument value from the data stream object. This is achieved by calling the method `getScalar`. Once we have the double value we can compute the sine. The final operation is to return the computed value to the caller. The return type is a pointer to an array of bytes; however, this would be tedious to construct, therefore the data stream object supplies a packing method which will take care of this detail. The pack method takes a variant array as an argument; this allows use to pack any SBW data type.

7.2 Calling Services

Up to now we have only discussed how we go about providing services. In this section we will describe how one can use the services of another module.

There are two approaches to accessing services in other modules from Delphi:

- Low-Level Access
- High-Level Access

7.2.1 Low-Level Access

It is possible to call remote methods and functions using the so-called low level access API. This is achieved by obtaining the remote handles to the module, service and required method and then making a direct call.

```
TrigModuleId := sbw.getModuleId ('TrigModule');
TrigId := sbw.getServiceId ('TrigModule', 'TrigService');
SinId := sbw.getMethodId ('TrigModule', TrigId, 'sin');
```

The remote handles are represented using integers and are internally generated by the broker during registration.

Once the remote handles have been obtained we can make a call to the desired function using the `Call method`. This method takes the list of module handles together with a variant list of arguments which the caller needs. Note that `Call` has been designed to be generic in the sense that it can return any SBW type. To obtain the data returned it is necessary to specifically request the type of data, in this case using a call to `getDouble`.

```
x := sbw.Call (TrigModuleId, TrigId, sinId, [5.4])).getDouble;
```

7.2.2 High Level Access

A more convenient approach is to use the `getMethodObj` method; this returns a `TSBWCallObject` from which it is possible to call the module method directly. For example:

```
moduleId := sbw.getModuleInstance ('TrigModule');
MySin := sbw.getMethodObj (moduleId, 'trig', 'sin');

Answer := MySin.Call (3.4).getDouble;
```

References

Finney, A., Hucka, M., Sauro, H. M., and Bolouri, H. (2002). Systems Biology Workbench C programmer's manual. Available via the World Wide Web at `http://www.cds.caltech.edu/erato/sbw/docs/`.