



AGETOR®

Integration

Table of contents

1	Preface	3
2	Language mapping.....	3
2.1	C mapping rules.....	3
2.1.1	Mapping of types to C	3
2.1.2	Creating variables of the generated types.....	5
2.1.3	Input and output parameters	5
2.1.4	Sequences	5
2.2	OCX mapping rules	5
2.2.1	Safe naming in OCX.....	6
2.2.2	Mapping IDL types to OCX/COM	6
2.2.3	Structures in OCX	6
2.2.4	Sequences in OCX.....	7
2.2.5	Methods in OCX.....	7
2.3	ABF mapping rules	8
2.3.1	Mapping of types to ABF	8
2.3.2	Structures in ABF	9
2.3.3	Sequences in ABF.....	9
3	Navision Financials Integration	10
3.1	Overall	10
3.1.1	Prerequisites	10
3.2	Architecture.....	10
3.2.1	Servicing the IDL interface	10
3.2.2	Invoking the IDL interface.....	12
3.3	Example.....	13
3.3.1	The CRM IDL	13
4	Concorde XAL Integration	16
4.1	Overall	16
4.1.1	Prerequisites	16
4.1.2	Architecture	17
4.2	Application Programming Interface.....	18
4.2.1	Control functions:	18
4.2.2	Request reading functions:.....	19
4.2.3	Response writing functions:	20
4.3	XAL pseudo-code:	21
4.4	Implementing the IDL defined protocol	22

1 Preface

This document describes how to integrate AGETOR to your existing solutions. It starts explaining the mapping rules to IDL (see document IDL guide) for the different languages. If you are mapping to Java, then you must use the IDL guide directly. Finally, the document explains how to integrate to different back-end systems.

2 Language mapping

AGETOR is based on a subset of IDL called AGETOR IDL (AIDL). Some features were removed from standard IDL to ease the mapping to different host languages.

Clients and services can be written in many different host languages which all map to the same IDL specification so procedure calls can be transmitted between these different languages. The mapping from IDL to any host language is defined by mapping rules, specifying which language types corresponds to IDL types and how IDL structs, sequences, interfaces and methods are to be represented in the host language.

2.1 C mapping rules

This section describes how to access IDL defined types, structures and operations in C.

2.1.1 Mapping of types to C

C does not support pass-by-variable parameters, so out and inout parameters to IDL generated functions must be passed by reference. Otherwise the server programs changes to these variables cannot be passed back to the client.

IDL-type	C-type (in parameter or struct member)	C-type (out/inout parameter)
Boolean	int	int*
char (8 bit)	char	char*
octet (8 bit)	int	int*
string (af char)	char*	char*
short (16 bit)	int	int*
long (32 bit)	int	int*
float (32 bit)	float	float*
double (64 bit)	double	double*

The IDL type sequence is mapped to an Array type:

<code>Array Array_create();</code>	Arrays used internally or passed as return value (not parameter) from an IDL generated function must be allocated by the application programmer.
<code>void Array_free(Array o, void freeFunc(void*));</code>	Only Arrays allocated internally are explicitly freed. Arrays passed as parameters or as return values are freed from by the IDL generated skeleton code.
<code>void Array_add(Array o, void *element);</code>	Add an element to the designated Array.
<code>void Array_remove(Array o, void *element);</code>	Remove the element from the designated Array.
<code>void Array_removeAll(Array o);</code>	Remove all elements from the Array. Thus <code>Array_size</code> will return 0 after this call.
<code>int Array_size(Array o);</code>	Returns the current number of elements in the Array.
<code>void *Array_get(Array o, int index);</code>	Returns the element at the given index. Casting to the appropriate type is necessary.
<code>void Array_set(Array o, int index, void *element);</code>	Sets the element at the given index.

The predefined type `Date` is mapped to a `Date` type:

<code>Date Date_create();</code>	Only Dates used internally or passed as return values are allocated by the application programmer.
<code>void Date_free(void *o);</code>	Only Dates allocated internally are explicitly freed. Dates passed as parameters or as return values are freed from by the IDL generated skeleton code.

Any user-defined type `type` is mapped to a C type named `type` with a function for allocating and initializing (create) and deallocating (free):

<code>type type_create();</code>	Only <code>type</code> used internally or passed as return values are allocated by the application programmer.
<code>void type_free(void *o);</code>	Only <code>type</code> allocated internally are explicitly freed. <code>type</code> passed as parameters or as return values are freed from by the IDL generated skeleton code.

2.1.2 Creating variables of the generated types

In order to correctly create variables of the types that were generated by the `idl2c` tool, you must use the `type_create()` function. This returns you a pointer to the type. Notice that you always operate on pointers to structures, because an IDL struct (named A) always declares the following C code:

```
typedef struct {  
    ... /* Declaration of struct members. */  
} AStruct;  
  
typedef AStruct *A;
```

So in order to declare a variable of this type in your code, write:

```
A a = A_create();  
... /* access the members of A. */  
A_free(a);
```

2.1.3 Input and output parameters

Both input and output parameters are allocated and freed by the skeleton code that marshals and demarshals the parameters. Thus, you should not create and free parameters declared in the IDL method. Return values are not created, because they do not (explicitly) have a name. They are however freed by the skeleton code, so you are responsible for creating the return value.

2.1.4 Sequences

IDL sequences are mapped to the Array type as described above. When declaring parameters of the sequence type the skeleton code is responsible for creating and freeing the Array (as with any parameter). When adding elements to the Array, you must however create each structure before calling `Array_add`. Note that only structures may be inserted into an Array, because the AGETOR IDL only allows sequences of structures, not simple types. The skeleton code will free all elements in the Array when your function implementation returns.

2.2 OCX mapping rules

This section describes how to access IDL defined types, structures and operations from an OCX.

When implementing/accessing IDL methods from COM enabled platforms and object-oriented languages such as Visual Basic, the COM solution is preferable. However for non object oriented languages that can access OCX but have no way of creating/accessing objects, OCX is an alternative.

As an example consider Navision Financials AL, which is a procedural language tightly integrated with the underlying relational database. This language can not instantiate or access objects, so the OCX holds and creates all structures and sequences corresponding to the IDL specified operations. AL can access simple typed variables by 'dotting' down through the runtime generated object hierarchy, setting and getting only the leaf variables.

2.2.1 Safe naming in OCX

The OCX is actually created by generating Visual Basic code and compiling this to an OCX. However the reserved keywords Visual Basic may conflict with parameter/field naming in the IDL (i.e. date) and to avoid this conflict you may specify the option -s<prefix> to the idl2ocx command. This will prefix the prefix string to all parameter/fields thus resolving the conflicts.

2.2.2 Mapping IDL types to OCX/COM

This table describes the mapping of simple parameters:

IDL-type	COM/OCX-type
Boolean	Long (0/1)
char (8 bit)	Long
octet (8 bit)	Long
string<n>	String
short (16 bit)	Long
long (32 bit)	Long
float (32 bit)	Single
double (64 bit)	Double

The predefined complex type Date is mapped to a date type:

Date	Date
------	------

2.2.3 Structures in OCX

Structures are not accessed directly but just traversed on 'the way down' to the simple types. Think of a type as a tree, where structures and sequences are internal nodes and simple types are leaves.

The example structure UserObject from the IDL guide document corresponds to a tree like this one, where simple types are printed in bold:

UserObject		
5,,,,	str	
5,,,,	number	
5,,,,	date	
5,,,,	obj	
.	Φ,,,,	number
Φ,,,,	objs	

$\Phi_{,,,,}$ number

So the five different leaf nodes may be read by the following pseudo code:

```

UserObject.str
UserObject.number
UserObject.date
UserObject.obj.number
UserObject.objs.obj.number

```

2.2.4 Sequences in OCX

Like structures, sequences are not mapped to objects, but exist as a kind of cursor. This means that a sequence parameters have methods to navigate and modify it. Navigation moves the cursor to an element, which may then be accessed through the cursor. The cursor has the same properties as the structure that the sequence contains.

The following navigation and modification methods are available on sequences:

<pre> toItem(index) toFirst() toLast() long size() clear() create() remove(index) </pre>	<p>Moves the cursor to element index assuming the index is valid. Otherwise the cursor is not moved.</p> <p>Moves the cursor to the first element.</p> <p>Moves the cursor to the last element.</p> <p>Returns the size of the sequence.</p> <p>Resets the sequence, thus removing all elements. The size of the sequence will be 0 after this call.</p> <p>Creates a new element and adds it as the last element in the sequence. The cursor is moved to this element.</p> <p>Removes the element at the given index, thereby reducing the size of the sequence by 1.</p>
--	--

The following example illustrates how to write the n'th element in the sequence `objs` from the former example:

```

UserObject.objs.toItem(n);
UserObject.objs.obj.number

```

The next example shows how to reset the sequence and create a new element in the sequence `objs` from the former example:

```

UserObject.objs.clear();
UserObject.objs.create();
UserObject.objs.obj.number

```

2.2.5 Methods in OCX

Each method in the IDL interface is mapped to a property in the OCX. Accessing the methods parameters and return value happens through this property.

The example IDL from the IDL guide document, section on Structures, generates an OCX which can be depicted like this:

test				The OCX
Φ,,,,,	method			The only method
	5,,,,,	a		Simple type short
	5,,,,,	b		Simple type string
	5,,,,,	c		Simple type Date
	.	d		Structure type UserObject
	.	5,,,,,	str	Simple type string
	.	5,,,,,	number	Simple type short
	.	5,,,,,	date	Simple type
	.	5,,,,,	obj	Structure type UserObject2
	.	.	Φ,,,,, number	Simple type short
	.	Φ,,,,,	objs	Sequence type UserObjects2
	.	.	Φ,,,,, number	Simple type short
	5,,,,,	e		Sequence type UserObjects
	.	5,,,,,	str	Simple type string
	.	5,,,,,	number	Simple type short
	.	5,,,,,	date	Simple type Date
	.	5,,,,,	obj	Structure type UserObject2
	.	.	Φ,,,,, number	Simple type
	.	Φ,,,,,	objs	Sequence type UserObjects2
	.	.	Φ,,,,, number	Simple type
	Φ,,,,,	method		Return value type short

2.3 ABF mapping rules

This section describes how to access IDL defined types, structures and operations in ABF.

2.3.1 Mapping of types to ABF

The ABF mapping is based on a template so you do not define the procedures yourself, but rather fill in the corresponding section in a template, which was generated. Parameters are mapped to variables and sequences are mapped to tables.

IMPORTANT: Variables in the IDL are mapped to lowercase names in ABF.

This table describes the mapping of simple parameters:

IDL-type	ABF-type
Boolean	int not null with default
char (8 bit)	varchar(1) not null with default
octet (8 bit)	int not null with default
string<n>	varchar(n) not null with default [n defaults to 100+1]
short (16 bit)	int not null with default
long (32 bit)	int not null with default
float (32 bit)	float not null with default
double (64 bit)	float not null with default

The predefined complex type Date is mapped to a date type:

Date	date not null with default
------	----------------------------

2.3.2 Structures in ABF

Nested structures no longer expanded as in earlier releases
Lets look at some sample IDL definitions:

```
struct A {
    string stringA;
};
struct B {
    A a;
    string stringB;
};
interface service {
    void test(
        inout string string1;
        inout short short1;
        inout Date date1;
        inout A a;
        inout B b;
    );
};
```

In your OSQ file you need to define the template section corresponding to operation test:

```
#template.test

/* Accessing simple type parameters. */
_string1 = _string1 + 'server-side';
_short1 = _short1 * 2;

/* Accessing structure a. */
_a.stringA = _a.stringA + 'server-side';

/* Accessing structure b and its substructures. */
_b.stringB = _b.stringB + 'server-side';
_b.a.stringA = _b.a.stringA + 'server-side';

#template.end
```

2.3.3 Sequences in ABF

IDL Sequences are mapped to arrays in ABF. Since structures defined in the IDL file, now are imported into ABF, SQL scripts for table generation in Ingres are no longer generated.

3 Navision Financials Integration

This section describes how to use the pre-built general Navision Financials AGETOR service as well as how to write new Navision Financials AGETOR services and clients.

This integration is currently built directly on the general OCX integration. This means that no additional convenience code for the AL programming environment is generated from IDL specifications.

3.1 Overall

The integration is based on the OCX integration which provide message semantics and transmission of simple IDL types.

3.1.1 Prerequisites

The following elements must be present on the system:

- The COM DLL named "ORBLib.dll" contains a handful of proxy objects. The COM/DLL must be registered on the machine running Navision Financials.
- A Microsoft Java Virtual Machine which is usually installed with Windows NT/98/2000.
- An OCX generated from an IDL specification named according to the IDL module name.

3.2 Architecture

Navision Financials may function both as a client and a server, i.e. both processing remote procedure calls from clients, and/or invoke procedures on remote services. All remote procedures are defined by an IDL interface (see the document IDL guide).

3.2.1 Servicing the IDL interface

Writing the necessary AL code to handle remote procedure calls requires a few simple steps:

1. Defining the OCX.
2. Starting the service.
3. Awaiting a remote procedure call.
4. Processing the remote procedure call and reading/writing parameters.
5. Indicating the procedure call have been executed.
6. Looping to step 2.
7. Closing the service.

This steps are described in the next sections.

3.2.1.1 Defining the OCX

Ensure that the OCX has been registered by selecting Functions: Custom Controls and browsing for the required OCX.

Once the OCX has been registered, define a new variable by entering the service codeunit and selecting show:C/AL Globals. Then define a name for the OCX and select 'OCX' in datatype and select the OCX in the subtype field.

3.2.1.2 Starting the service

The service listens for socket connections from the Broker on a specified port. The OCX contains a separate thread that will listen on a port. This thread is started by calling:

```
ocx.startService(port);
```

This example assumes that the OCX variable name is `ocx`. The port is simply a number variable indicating which port the service is listening on.

This statement will not block the codeunit. It only starts the OCX thread which you will now be able to ping from the Broker, assuming that you have configured a service on the correct machine and port number in the Broker configuration (see the document IRE guide, the section about the Broker).


3.2.1.3 Awaiting a remote procedure call

Once the service has been started, the OCX thread will be running. To intercept remote procedure calls, the codeunit must await the OCX thread receiving a message.

This is done by calling:

```
request := ocx.awaitRequest();
```

The `request` variable has type `text` and will hold the textual name of the method invoked. The method name corresponds to the method name stated in the IDL interface.

 Note that this method call will block the codeunit until the OCX receives a remote procedure call.

Since Navision Financials (as most Windows applications) are single threaded this means that it will not respond to mouse or keyboard events. The `awaitRequest` method will return once a remote procedure call is made or it is killed by the broker. Killing the service from the Broker will terminate this Navision Financials session.

Future enhancements allow you to specify a timeout value for the blocking `awaitRequest` method, so control will return to the codeunit if a remote procedure call was not received within the timeout period.

3.2.1.4 Reading/writing parameters

Once a remote procedure call is received in parameters may be read from the OCX and out parameters may be written to the OCX. The parameters will be local to this procedure call and will not be available across multiple remote procedure calls.

Parameters may be read and written in any order.

The OCX will hold the parameters specified by the IDL and make these available according to the mapping rules. The parameter mapping rules are described in section 2.2.

3.2.1.5 Returning from the procedure call

Once the procedure call has been processed and all out parameters (including an eventual return value) has been written, the OCX thread is informed to return a message to the Broker containing the out parameters.

```
ocx.requestProcessed();
```

After this call no parameters should be read or written. At this point a service should prepare itself for a new remote procedure call and call `awaitRequest` again.

3.2.1.6 Closing the service

If the service decides to shut down it should do so by terminating the OCX thread. Simply exiting the AL code unit without terminating the OCX thread will actually leave this thread running because a Windows OCX retains its state variables once it was loaded by Navision Financials. Closing this Navision Financials session though will also terminate the OCX thread.

Explicitly closing the service, thereby terminating the OCX thread is done by calling:

```
ocx.closeService();
```

Before the service has been closed another call to `startService(port)` will fail.

3.2.2 Invoking the IDL interface

Writing the necessary AL code to invoke procedures on remote services also requires a few simple steps:

1. Connect to a Broker.
2. Write all input parameters.
3. Invoke method and await result.
4. Read all output parameters..
5. Closing the connection.

The following sections describes these steps.

3.2.2.1 Client connect

Connecting to a Broker as an internal client (see the document IRE guide, the section about the Broker):

```
status := internalConnect(host, port);
```

The variable `status` is a number containing the status of the connection attempt. The `host` is a text variable containing the host name of the machine with the Broker. The `port` is a number containing the internal port of the Broker.

External connections may be performed by:

```
status := externalConnect(url, user, pwd);
```

All parameters are of type text. The url parameter contains the Uniform Resource Location of the webserver connecting to the desired Broker. The user and pwd are a user identification and password required for security enforcement.

3.2.2.2 Reading and writing parameters

Before invoking the remote method, input parameters must be written to the OCX by following the IDL to OCX mapping rules described in section 2.2.


After the remote method invocation returns, output parameters as well as the eventual return values can be read from the OCX following the same rules.

3.2.2.3 Invoking the remote procedure

Once the input parameters have been written to the OCX, the remote procedure can be invoked. To invoke a method obviously the name of the method is required. Further the environment which the service is configured in must also be stated.

The remote method invocation looks like this:


```
status:=invokeMethod(env, method);
```

 This invocation syntax will be extended by direct calls for each method in the IDL interface, in future releases.

The status variable of type number will contain the status of the procedure call.

Once the procedure call returns, output parameters and return value from the method will be available for reading from the OCX.

3.2.2.4 Closing the connection

 Currently no explicit connection closing is required because client connections do not start threads.

This will change in the next release to minimize resource usage.

3.3 Example

To explain the integration, an example IDL is constructed. The following IDL illustrates some simple operations within a Customer-relations-management system (CRM).

3.3.1 The CRM IDL

```
module dk.bording.idl.crm {
    struct Customer {
        string<10> id;
        string<40> name;
    }
}
```

```
    string<20> phone;
    string<50> email;
    string<50> address1;
    string<50> address2;
};
typedef sequence <Customer> Customers;

struct RecordLine {
    string<30> text;
    float amount;
    float total;
};
typedef sequence <RecordLine> RecordLines;

struct Note {
    string<100> text;
};
typedef sequence <Note> Notes;

struct CustomerReport {
    Customer customer;
    RecordLines recordLines;
};

/**
 * CustomerService allows retrieval of CustomerReports,
 * setting Customer information,
 * and getting/setting customer associated notes.
 */
interface CustomerService #env="dev" #qno=700 {

    /**
     * getCustomers returns a sequence of all Customers.
     * @returns all Customers.
     */
    Customers getCustomers(
    );

    /**
     * getReport gets the CustomerReport associated with a Customer.
     * @param id is the Customer identifier.
     * @param report is the CustomerReport incl. Customer information.
     * @returns whether the report existed.
     */
    boolean getReport(
        in string<10> id,
        out CustomerReport report
    );

    /**
     * getCustomer gets the Customer with the given id.
     * @param id is the Customer identifier.
     * @returns whether the Customer existed.
     */
}
```

```
    */
    boolean getCustomer(
        in string<10> id,
        out Customer customer
    );

    /**
     * setReport sets the Customer information.
     * @param customer is the Customer information incl. an identifier.
     */
    void setCustomer(
        in Customer customer
    );

    /**
     * getAssociatedNotes gets the Notes associated with a Customer.
     * @param id is the Customer identifier.
     * @param notes is the associated Notes.
     * @returns whether the notes existed.
     */
    boolean getAssociatedNotes(
        in string<10> id,
        out Notes notes
    );

    /**
     * setAssociatedNotes sets the notes associated with a Customer.
     * @param id is the Customer identifier.
     * @param notes the Notes that will be associated the Customer.
     */
    void setAssociatedNotes(
        in string<10> id,
        in Notes notes
    );

};
```

4 Concorde XAL Integration

This document describes how to make Concorde XAL act as a service within AGETOR. This means that Concorde XAL will respond to a set of defined requests that clients (servlets, Java applets, applications and even other services) may ask.

4.1 Overall

The XAL language itself does not support neither complex types nor sequences, so IDL defined interfaces are not directly transformable into XAL language constructs. XAL does allow embed SQL-like statements to iterate database tables etc. However reading and writing simple types in the IDL specified order will allow for a simple integration into XAL, so this document describes which operations in a Windows DLL XAL should invoke to behave properly as an AGETOR server.

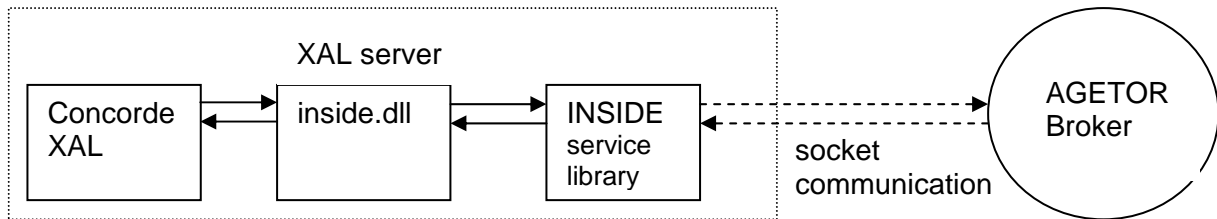
4.1.1 Prerequisites

The following files (Windows DLL's and Java Archives (JAR)) must be present in the directory where XAL is running from:

- inside.dll Containing the functions that XAL invoke to read requests and write responses.
- javai.dll Java Virtual Machine (JVM).
- net.dll The Java networking library.
- symcjit.dll Symantec Just-In-Time (JIT) compiler for JVM.
- inside.jar AGETOR service library.
- tools.jar AGETOR tools library.

4.1.2 Architecture

The inside.dll consists of wrapper functions written in C which invoke the Java Virtual Machine and utilizes the AGETOR service library to read and write simple types from the network communication.



XAL loads the inside.dll and starts the AGETOR service library by indicating the port number from which it will be servicing requests. This port number must correspond to the port number assigned to the service on the AGETOR broker.

The broker will now forward any requests to this service to the indicated port number.

XAL must now await an incoming request, which means that control is left to the AGETOR service library, and only returned once a request is available. Due to XAL single threaded architecture this will leave the XAL Window non-responsive and not updated until a request is available.

Once the AGETOR service library receives the request it will return control to XAL indicating the request type. The IDL defines which parameter types the request consists of and XAL must now read the input parameters and write the output parameters in the defined order. When this is done XAL notifies the AGETOR service library that the response for the request has been completed. The AGETOR service library then returns the response to the broker. XAL is now ready to process the next request and must await this.

4.2 Application Programming Interface

When calling functions in a DLL from within XAL, these functions must adhere to the following signature:

```
unsigned char* methodname(long x, unsigned char* y);
```

Thus functions must take two arguments: a long integer (called *x*) and a null terminated string (called *y*). Functions will always return a null terminated string.

Below is a description of the available functions. The description describes arguments and return values if these are used. Otherwise they are not mentioned.

4.2.1 Control functions:

<code>XAL_startService</code>	Starts the service on the given port number. Parameter X: The port number.
<code>XAL_awaitRequest</code>	Block calling thread until request from client is available. Parameters may now be read from the request using the <code>get<type></code> functions. Also parameters may be written to the reply using the <code>put<type></code> functions. The reply is only sent when the <code>requestProcessed</code> function is called.
<code>XAL_requestProcessed</code>	Send the generated reply. No more parameters can be read from the request or written to the reply, before a new request is available (<code>awaitRequest</code>). If protocol errors were encountered in the request, no reply is generated, and the status is cleared, thus preparing the service for a new request.
<code>XAL_getStatus</code>	Returns the status of the last operation. Return value: The status of the last operation

4.2.2 Request reading functions:

XAL_getChar	Reads a single char from the request. Return value: The read char.
XAL_getDouble	Reads a double floating point number from the request. Return value: The read double represented as a string.
XAL_getFloat	Reads a floating point number from the request. Return value: The read float represented as a string.
XAL_getInt	Reads an integer number from the request. Return value: The read integer represented as a string.
XAL_getString	Reads a string from the request. Parameter X: The maximum length of the string. Return value: The read string.
XAL_getBoolean	Reads a boolean (0 or 1) from the request. Return value: The read boolean represented as a string.
XAL_getDate	Reads a date from the request. The date is returned as a 64 bit integer representing the number of milliseconds passed since January 1, 1970, 00:00:00 GMT. Return value: The read date represented as a string.
XAL_getDoubleAsInt	Reads a double floating point number from the request. Converts the double to an integer, by multiplying with parameter multiply, and rounding. Parameter X: The multiplication factor. Return value: The multiplied and rounded integer represented as a string.
XAL_getFloatAsInt	Reads a floating point number from the request. Converts the float to an integer, by multiplying with parameter multiply, and rounding. Parameter X: The multiplication factor. Return value: The multiplied and rounded integer represented as a string.

4.2.3 Response writing functions:

XAL_putChar	Write a single char to the reply. Parameter Y: A string which first character is written.
XAL_putDouble	Write a double floating point number to the reply. Parameter Y: A string representing the double to write.
XAL_putFloat	Write a floating point number to the reply. Parameter Y: A string representing the float to write.
XAL_putInt	Write an integer number to the reply. Parameter Y: A string representing the integer to write.
XAL_putString	Write a string to the reply. Parameter X: The maximum size to write. Parameter Y: The string to write.
XAL_putBoolean	Write a boolean (0 or 1) to the reply. Parameter Y: A string representing the boolean to write.
XAL_putDate	Write a date number to the reply. The date is a 64 bit integer representing the number of milliseconds passed since January 1, 1970, 00:00:00 GMT. Parameter Y: A string representing the 64 bit integer to write.
XAL_putDoubleFromInt	Write a double floating point number to the reply. The integer in Y is divided by the multiplication factor (in X) and written as a double. Parameter X: The multiplication factor. Parameter Y: A string representing the integer to write as a double.
XAL_putFloatFromInt	Write a floating point number to the reply. The integer in Y is divided by the multiplication factor (in X) and written as a float. Parameter X: The multiplication factor. Parameter Y: A string representing the integer to write as a float.

4.3 XAL pseudo-code:

This section contains some XAL pseudo-code to illustrate how to use the API from within XAL.

```
{Load DLL}
    SET &DLLHandle = DLLOPEN( "INSIDE.DLL" )
    IF( &DLLHandle == 0 ) THEN
        SET &ErrorString = "Error using DLL:= "+NUM2STR(&PROCRET,1,0,0,0)
        SET BOX(1, &ErrorString,0)
    ELSE
{Start Service}
        SET &Status = DLLCALL(&DLLHandle, "XAL_startService", &PortNo, "")
        SET &Status = "0"
{Loop until status not OK }
        WHILE (Str2Int(&Status)==0)
{Await incoming request}
            SET &Question = DLLCALL( &DLLHandle, "XAL_awaitRequest", 255, "" )
            #Switch(&Question)
{Process request 'method'}
            #Case("method")
{Read request and write response, may be intermixed.}
                SET &a = DLLCALL( &DLLHandle, "XAL_getString", 0, "" )
                SET &Status = DLLCALL( &DLLHandle, "XAL_putInt", 0, "42")
                SET &b = DLLCALL( &DLLHandle, "XAL_getInt", 0, "" )
                SET &temp = DLLCALL( &DLLHandle, "XAL_putString", 0, "Hello world")
{Send constructed response}
                SET &Status = DLLCALL( &DLLHandle, "XAL_requestProcessed", 0, "" )
                #Endswitch
            END
        ENDIF
```

4.4 Implementing the IDL defined protocol

Once an interface has been defined in IDL the server and client must both adhere to this interface. This is done by writing and reading arguments in the correct order. In Java and C this is handled by auto-generated stubs and skeletons which relieves the application programmer from the burden of strict protocol adherence. However the current integration of XAL in AGETOR requires the application programmer to get all input parameters and put all output parameters of requests.

By applying the `idl2html` tool to an IDL a HTML description of the protocol is generated. From the HTML it is straightforward to get the input parameters and put the output parameters.