Calling C plugins from sa.engine

Stream Analyze Sweden AB
Sweden

Version 2.1
2022-01-13
`sa_C_pluginAPI_2.0.pdf`

One important property of sa.engine is that it is designed to be tightly integrated with other systems. Tightly integrated here means that sa.engine can be linked to external programs using the *plugin API* where *foreign* OSQL functions can be implemented in C. The combination is also possible where foreign functions call sa.engine back through the client API. There are predefined APIs for interfacing code in several programming languages: C99, C++, Lisp, Java, Python and JavaScript. The system furthermore provides primitives for defining APIs to any other programming language based on the C99 API. This document describes the API to define plugins that extend the sa.engine core system.

Table of contents

1. Introduction

There are two main kinds of external interfaces to sa.engine, the *client* and the *plugin* interfaces:

- With the *client interface* a program implemented in some programming language calls sa.engine. The client interface allows OSQL queries and function calls to be shipped from application programs to either i) *remote* sa.engine servers or ii) an *embedded* sa.engine system running in the same process as the application. The client interface is documented separately [2].

- With the *plugin interface* OSQL functions are implemented as code in some programming language. These *foreign OSQL functions* are executed in the same process and address space as sa.engine. The client interface can be used also in foreign OSQL function implementations. This document describes how the define foreign functions in C.

The plugin API of sa.engine for C is presented in this document through a number of example programs whose source codes are in the folders `sa.engine/demo/*/C` of an installed sa.engine system. In that folder you will find a number of examples for how to compile, use and validate C plugins using the APIs. You are assumed to be familiar with OSQL.

1.1. Object handles

All access to objects inside sa.engine from C is made through *object handles* which are indirect identifiers for physical data structures, called *storage objects*, stored in the database image of sa.engine. Object handles in general are declared as C type `ohandle` in the header file `sa_client.h`.

Notice that object handles *must always* be initialized to `nil`, declared like this:

```
ohandle myhandle = nil;
```

There are specialized object handle C data types for different kinds of objects such as object streams, tuples, or connections, for example:

```
sa_stream mystream = nil;
sa_tuple mytuple = nil;
sa_connection myconnection = nil;
```

In order to make the application code both fast and independent of the internal representation of object handles, the object handles are always manipulated through a set of C macros and functions. The interface is connected to an automatic garbage collector inside sa.engine so that data no longer used is reclaimed when using those macros/functions.

Storage objects have an associated data type called a *storage type* represented an integer. The

storage type tag of object handle `h` can be accessed with the function:

```
int sa_typetag(ohandle h);
```

For a given storage type tag `tt`, the name of the corresponding storage type is retrieved by:

```
char *sa_storagetype(int tt);
```

The following storage types are predefined as macros:

Storage type name	Type tag	Represents
INTEGER	INTEGERTYPE	Integers
REAL	REALTYPE	64-bit real numbers
STRING	STRINGTYPE	Strings
ARRAY	ARRAYTYPE	1D arrays of object handles
RECORD	RECORDTYPE	Records of key/value pairs
BINARY	BINARYTYPE	Binary areas (buffers)
OID	SURROGATETYPE	OSQL objects

1.2. Foreign functions

Foreign functions in C execute inside the sa.engine kernel. They are thus executing in the same address space as the local main memory database. The programmer has full access to the data structures accessible inside the database image described in [5]. There is no protection for destroying data inside the kernel, so the programmer must make sure that the C code is correct. This is the case if the conventions described below are followed.

For developing safe foreign functions where small mistakes will not crash the system, it is recommended to implement them in e.g. Lisp [1] [3], which will make them somewhat slower. Thus C is recommended for very high performance foreign functions and for connecting to external systems through C when Lisp cannot be used. An alternative is to write them in a storage safe language such as Java [3], but this will make them slower and they may not run on small edge devices where there is no JVM.

When possible regular OSQL-function should be used rather than implementing foreign functions. There is an OSQL-compiler that can generate efficient binary code for a certain class of OSQL-functions, making them (almost) as fast as C for numeric expressions.

The sa.engine kernel can execute many threads so the C code inside foreign functions may run in parallel to other foreign functions. For thread safety, the system puts locks around the execution of each foreign function so that C code executing in different threads does not interfere or crash, given that the conventions below are followed. The user normally does not have to manage locks.

To prevent blocking when accessing external systems or performing long-running or parallel computations, the programmer can put the current thread in background mode to temporarily release its lock. This is for example used when waiting for socket I/O.

There are several flavors of how foreign OSQL function can be defined:

- A *simple* foreign function that takes zero, one or several atomic arguments and return a simple object, e.g. *sqrt(Number x)->Number*. The most simple one is a "Hello World" function with signature *hello()->Charstring* that returns the string "Hello World". The implementation of the Hello World function is located in the folder `sa.engine/demo/Hello/C`. Once the Hello World function runs OK the other foreign functions usually are easy to made working as well.
- A foreign function returning a *tuple* where more than one result is computed as tuples containing several simple objects.
- A foreign function returning a *bag* where a set objects, possibly containing duplicates, is returned as an object stream.
- A foreign function returning a *stream* where possibly infinite streams of ordered objects are returned as an object stream.
- A foreign function over *vectors* where vectors are arguments and/or results.
- A foreign function over *records* where records similar to JSON objects are operated upon.
- An *aggregate function* where simple values are computed forgiven bag or stream argument(s).
- A *stream transformation function* where an input stream is transformed into another result stream.

Next follows examples of how to implement the different kinds of foreign functions in C in `sa.engine/demo/*/C`. Notice that the example foreign functions also can be implemented more easily directly in OSQL in folders `sa.engine/demo/*/OSQL`.

2. Simple foreign functions

Simple foreign functions take atomic values as input parameters and produce one or several results computed from the inputs.

2.1. A Hello World foreign function

To get started making sure that you can successfully compile and load plugins there is a trivial "Hello World" program in folder `sa.engine/demo/Hello/C/Hello.c`. The C function implementation `helloB` in `Hello.c` implements in C a trivial foreign OSQL function with signature *hello()->Charstring* that has no arguments and returns the string "Hello world!". It is compiled into a shared object `bin/hello.so` (Unix) or DLL `bin/hello.dll` (Windows):

```
#include "sa_core.h"

ohandle helloF(a_callcontext cxt)
{
  /* Bind result tuple element one to "Hello world!" handle */
  a_bind(cxt, 1, mkstring("Hello world!"));
```

```
  /* Emit result tuple */
  a_result(cxt);

  /* Always return nil from foreign C function implementations: */
  return nil;
}

// This initialization function is called when dynamically loading the shared
// object sa.engine/bin/hello.so or DLL sa.engine/bin/hello.dll:
EXPORT void a_initialize_extension()
{
  a_extimpl("hello+", helloF); // Bind implementation to symbol "C:hello+"
}
```

A foreign C function implementation *ff* always has the signature

```
ohandle ff(a_callcontext cxt)
```

The *call context* `cxt` provides an interface between the state of the host sa.engine system and the foreign function implementation. It contains a *parameter tuple* that hold both the arguments and results of the called foreign function. It allows the foreign function implementation both to access the argument parameters of the called OSQL function and to bind result parameters of one or several computed result tuples emitted to sa.engine by the implementation.

In the example, the foreign function implementation emits an object stream containing a single element, a handle to the string “Hello world!”. The macro

```
a_bind(cxt, p, h)
```

binds position `p` in the parameter tuple of `cxt` to object handle `h`. In the example the first and only element of the parameter tuple is bound to the object handle representing the result string “Hello world!”. The macro `mkstring(s)` makes an object handle of the C string `s`.

A parameter tuple is emitted by calling

```
a_result(cxt);
```

In the example a single result tuple containing the string “Hello world” is emitted by `a_result()`.

In order to define of the foreign function by an OSQL `create function` statement it has to be associated with a symbolic name. This is done in the *extension initializer* always named `a_initialize_extension` called once by sa.engine when the DLL or shared object is loaded. In the example the extension initializer calls the function:

```
a_extimpl(char *fn, external_implementation cfn)
```

The function `a_extimpl` associates the case insensitive symbol `fn` (`hello+` in the example) with a C implementation `cfn` (`helloF` in the example).

The `sa.engine/demo/Hello/Makefile` compiles `Hello.c` under Unix and makes the shared object `sa.engine/bin/Hello.so`. For Windows there is a Visual Studio solution `Hello.sln` that can be compiled from the console using the script `compile.cmd` (see `README` for details).

Once the foreign function implementation is compiled, the shared object `sa.engine/bin/Hello.so` can be dynamically loaded into sa.engine by the OSQL statement:

```
load_library("Hello");
```

Now the signature of the foreign OSQL function is defined by:

```
create function hello() -> Charstring
  as foreign 'hello+';
```

It can then be tested by calling

```
hello();
```

If this works you have successfully implemented, compiled, and defined your first foreign function in C. It is recommended that you make an OSQL script that validates that the foreign function works as expected, as in `sa.engine/demo/Hello/validation.osql`. That script is run by issuing this (Unix) command in Hello/C folder:

```
make test
```

Under Windows the test is made by running the batch script `test.cmd`.

2.2. Error handling

sa.engine has its own exception and error handling mechanisms. It is utilizing C's `setjmp/longjmp` functions to catch and throw exceptions when they happen. Errors can be thrown from inside foreign C functions by using one of the functions:

```
int a_throw_errormsg(const char *msg, ohandle obj);
int a_throw_errorno(int no, ohandle obj);
```

System errors have error numbers associated with corresponding error messages. Choose `a_throw_errornumber()` when you know the error number (some of them are macros defined in `C/sa_storage.h`) and `a_throw_errormsg()` when you want the system to assign the error number. Error messages are truncated to max 100 bytes. The built-in error messages are stored in an internal error table, which is looked up by `a_throw_errormsg()`. The error table can be extended when initializing the system by calling:

```
int a_register_error(const char *msg);
```

The client API [2] provides a number of primitives to access and set error codes. There you can only inspect and raise errors without actually throwing them to invoke the error handling. The corresponding functions are available in foreign functions:

```
int a_raise_errormsg(const char *msg, ohandle obj);
int a_raise_errorno(int no, ohandle obj);
```

It is then marked in the thread where the foreign function is executing that the error has occurred. To actually throw a raised error call:

```
ohandle a_throw_error(void);
```

Special care has to be taken when embedding sa.engine in programming languages or other advanced systems having its own exception handling, e.g. Java, Python, or C++. If that other system can call foreign OSQL functions and there is an error happening inside the called foreign OSQL function it is not allowed to throw an sa.engine error exception, as that would bypass the

exception handling of the embedding system, which can cause very obscure bugs to occur. Therefore, for advanced plugins where sa.engine is embedded in other systems having their own exception handling or where `setjmp/longjmp` is forbidden, delaying error handling by raising errors must be used to avoid throwing errors through the exception handling of the embedding system. This is used, e.g., in the embedding of sa.engine in the programming languages Java and C++.

2.2.1. Trapping errors

Since throwing errors immediately exits the foreign function, special care has to be taken in order to properly deallocate resources allocated in a foreign function when exceptions happen, e.g. when objects allocated with C's `malloc()` should be freed or when connections to servers must be closed when a foreign function is finished.

For this the system provides an error trapping facility, called *unwind-protection*, that provides a simple way to trap all sa.engine kernel errors and always execute clean-up code before exiting a C-block. To unwind-protect a piece of C-code use the following code skeleton:

```
unwind_protect_begin;
/* Place code to be protected here */
unwind_protect_catch;
/* This code will always be executed. If an sa.engine error happened in the
protected code the system will process the error, then trap the error and pass the
control here */
unwind_protect_end;
/* The control comes here ONLY if the execution of the protected code was
successful */
```

Make sure you always free all resources with an `unwind_protect_end` code section. Notice that even `a_result()` may fail, e.g. if a stream is terminated.

2.3. Functions with both arguments and result

The arguments of a foreign OSQL function in C are stored in the parameter tuple as well as the results. For example, the function *myconcat(Charstring x, Charstring y) -> Charstring* concatenates strings *x* and *y*. It has the following implementation in `sa.engine/demo/Basic/C/Basic.c`:

```
ohandle myconcatBBF(a_callcontext cxt)
{
  ohandle x = a_arg(cxt, 1);        // Pick up 1st argument in para tuple
  ohandle y = a_arg(cxt, 2);        // Pick up 2nd argument in para tuple
  char *dx, *dy, *res;

  IntoString(x, dx, cxt->env);      // Dereference x to string dx
  IntoString(y, dy, cxt->env);      // Dereference y to string dy
  res = alloca(strlen(dx)+strlen(dy)+1); // Stack allocate result string
  strcpy(res, dx);
  strcat(res, dy);
  a_bind(cxt, 3, mkstring(res));    // Bind result the 3rd element in para tuple
  a_result(cxt);                    // Emit result
```

```
  return nil;
}
```

In this case the arguments `x` and `y` of `myconcat(x,y)` are in positions one and two of the parameter tuple and the computed result is bound in position three.

The macro

```
IntoString(x, dx, env)
```

dereferences (converts) an object handle `x` containing a string into a C string `dx`. The third argument `env` is used for throwing an error in case `x` is not a string, in which case the foreign function call is aborted.

The macro

```
mkstring(x)
```

creates an object handle of a C string `x`.

The implementation is bound to the symbol `myconcat--+` by calling:

```
a_extimpl("myconcat--+", myconcatBBF);
```

Compile `Basic.c` with

```
make compile
```

When the compiled shared object `Basic.so` is loaded the signature of `myconcat()` is defined by the OSQL statement:

```
create function myconcat(Charstring x, Charstring y) -> Charstring
  as foreign 'myconcat--+';
```

2.4. Functions returning tuples

Foreign functions can return more than one result through tuples. For example, the function *sqrt2(Number x) -> (Number neg, Number pos)* returns a tuple of the negative and positive square roots of *x*. It has the following definition in `sa.engine/demo/Basic/C/Basic.c`:

```
ohandle sqrt2BFF(a_callcontext cxt)
{
  ohandle x = a_arg(cxt, 1);
  double dx, root;

  IntoDouble(x, dx, cxt->env);
  root = sqrt(dx);
  if(dx >= 0)
    {
      a_bind(cxt, 2, mkreal(-root));
      a_bind(cxt, 3, mkreal(root));
      a_result(cxt);
    }
  return nil;
}
```

The macro

```
IntoDouble(x, dx, env)
```

dereferences a number handle `x` into a C double `dx`. An error is thrown if `x` is not a number.

The macro

```
mkreal(dx)
```

makes an object handle of a C double `dx`.

In this case `a_bind()` is called twice to bind the values to the two roots. The implementation is bound to the symbol `sqrt2-++` by calling:

```
a_bind("sqrt2-++", sqrtBFF);
```

The OSQL definition is:

```
create function sqrt2(Number x) -> (Number neg, Number pos)
  as foreign 'sqrt2-++';
```

3. Foreign functions over vectors

Vectors represent ordered collections of objects of any type. They are represented as object handles with type identifier `ARRAYTYPE`.

3.1. Foreign function returning a vector

The function *vsqrt2(Number x) -> Vector* returns the negative and positive square root of number *x* as a vector. It is implemented in `sa.engine/demo/Basic/C/Basic.c` as:

```
ohandle vsqrt2BF(a_callcontext cxt)
{
  ohandle x = a_arg(cxt, 1);
  double dx;
  IntoDouble(x, dx, cxt->env);
  if(dx >= 0) {
    double root = sqrt(dx);
    ohandle roots = new_array(2,nil);
    a_seta(roots, 0, mkreal(-root));
    a_seta(roots, 1, mkreal(root));
    a_bind(cxt, 2, roots);
    a_result(cxt);
  }
  return nil;
}
```

Here the vector `roots` of size 2 is created by call `new_array(2,nil)`. The elements are initialized to the handle `nil` representing `null` in OSQL. The function

```
a_seta(ohandle v, int pos, ohandle val)
```

sets element `i` of vector `v` to handle `val` (vector elements in C are enumerated from 0 and up).

The C function `vsqrt2BF` is bound to the symbol `vsqrt2-+` by calling:

```
a_extimpl("vsqrt2-+", vsqrt2BF);
```

The OSQL definition is:

```
create function vsqrt2(Number x) -> Vector roots
  as foreign 'vsqrt2-+';
```

3.2. Functions taking vectors as arguments

The function *dotprod(Vector v, Vector w) -> Number* takes two vectors of numbers, *v* and *w*, and computes their scalar product. It has the following definition in `sa.engine/demo/Basic/C/Basic.c`:

```
ohandle dotprodBBF(a_callcontext cxt)
{
  ohandle v = a_arg(cxt, 1);
  ohandle w = a_arg(cxt, 2);
  int dim, i;
  double prod=0;
  OfType(v, ARRAYTYPE, cxt->env);
  OfType(w, ARRAYTYPE, cxt->env);
  dim = a_arraysize(v);
  if(dim != a_arraysize(w))
    {
      return a_throw_errormsg(cxt->env,"Array index out of bounds", v);
 }
  for(i=0; i<dim; i++) {
    ohandle ev = a_elt(v, i); // Access v[i]
    ohandle ew = a_elt(w, i); // Access w[i]
    double dev, dew;
    IntoDouble(ev, dev, cxt->env);
    IntoDouble(ew, dew, cxt->env);
    prod = prod + dev*dew;
  }
  a_bind(cxt, 3, mkreal(prod));
  a_result(cxt);
  return nil;
}
```

The macro

```
OfType(h, tp, env)
```

checks that the handle `h` has type identifier `tp` and throws an error otherwise. In the code `OfType` is called twice to check that the two arguments are vectors. The function `a_arraysize(ohandle v)` returns the number of elements of vector `v`.

If the number of elements of `v` and `w` are not the same an error is thrown by calling the C function:

```
ohandle a_throw_errormsg(bindtype env, const char *msg, ohandle v)
```

The function `a_throw_errormsg()` throws an error with message `msg` for handle `v`. The argument `env` is the binding context in which the error is thrown. In the example, the error message has a number in the *error table*. In the example the error table is looked up for the error message “Array index out of bounds [2]. Normally `a_throw_errormsg()` throws an exception and does not return.

The function

```
ohandle a_elt(ohandle v, int i)
```

accesses element `i` of vector `v` (or raises an exception if handle `v` is not a vector). The elements in the vector are enumerated starting from zero.

The C function `dotprodBBF` is bound to symbol `dotprod--+` by calling

```
a_extimpl("dotprod--+", dotprodBBF);
```

The OSQL definition is:

```
create function dotprod(Vector v, Vector w) -> Number
  as foreign 'dotprod--+';
```

4. Foreign functions over bags

Bags represent sets of objects possibly with duplicates. Foreign functions usually generate the bags iteratively by calling `a_result` several times as shown below.

4.1. Functions generating bags

The foreign function *natural(Number m, Number n) -> Bag of Number* generates a bag of the natural numbers from *m* to *n*. It is implemented in `sa.engine/demo/Basic/C/Basic.c` as

```
ohandle naturalBBF(a_callcontext cxt)
{
  int m = a_arg(cxt, 1); // Pick up first argument
  int n = a_arg(cxt, 2); // Pick up second argument
  int dm, dn, i;

  IntoInteger32(m, dm, cxt->env);
  IntoInteger32(n, dn, cxt->env);
  for(i=dm; i<=dn; i++) {
    // Bind the result element of the parameter tuple:
    a_bind(cxt, 3, mkinteger(i));
    a_result(cxt); // emit the result
  }
  return nil;
}
```

The macro `IntoInteger32` dereferences a integer handle into a 32-bit C integer and `mkinteger(i)` makes an object handle of a C integer.

Here `a_bind` and `a_result` are called `m-n+1` times, once per element of the result bag.

The call

```
a_extimpl("natural--+", naturalBBF)
```

associates the implementation with the symbol `natural--+`.

The OSQL definition is:

```
create function natural(Number m, Number n) -> Bag of Number
  as foreign 'natural--+';
```

4.2. Aggregate functions

Aggregate functions such as *sum(Bag b) -> Number* compute a single object for a given collection `b`. The foreign function *sqsum(Bag b)->Number* computes the sum of the square of the elements in the bag of number *b*. It has the following C implementation in `Basic.c`:

```
ohandle sqsumBFmapper(a_callcontext cxt, int arity, ohandle *restpl,
                      void *xa)
{
  double *sum = (double *)xa;
  double x;

  IntoDouble(restpl[0], x, cxt->env);
  // Element must be double or exception will be raised
  *sum += x*x;
  return nil; // Always return nil
}

ohandle sqsumBF(a_callcontext cxt)
{
  double sqs = 0;
  ohandle collection = a_arg(cxt, 1);

  // a_mapstream will map over both bags, streams, and vectors:
  a_mapstream(cxt, collection, sqsumBFmapper, (void *)&sqs);
  a_bind(cxt, 2, mkreal(sqs)); // Bind result to total sqs
  a_result(cxt); // Emit result tuple
  return nil;    // Always return nil
}
```

The main function `sqsumBF` initializes the square sum in `sqs` and calls the function:

```
a_mapstream(a_callcontext cxt, ohandle coll, mapper_function cb, void *xa)
```

The function `a_mapstream` calls a *mapper function* `mf` for each element of a collection `coll`, which is a bag in this case. A mapper function `mf` always has the signature:

```
ohandle mf(a_callcontex cxt, int width, ohandle tpl[], void *xa)
```

It is called for every tuple `tpl` in the collection, with `width` being the size of the tuple. The variable `xa` is passed unchanged from the `a_mapstream` call to the mapper function call.

The call

```
a_extimpl("sqsum-+", sqsumBF);
```

associates the implementation with the symbol `sqsum-+`.

The OSQL definition is:

```
create function sqsum(Bag b) -> Number
  as foreign 'sqsum-+';
```

4.3. Aggregation over vectors

A foreign aggregate function implementation over bags can also be used for aggregating over vectors. For example, the aggregate function `sqsum-+` above can also be used for computing the sum of the square of numbers in vector by defining the function:

```
create function sqsum(Vector v) -> Number
  as foreign 'sqsum-+';
```

4.4. Aggregation over finite streams

A foreign aggregate function implementation over bags can also be used for aggregating over finite streams. For example, the aggregate function `sqsum-+` above can also be used for computing the sum of the square of numbers in stream `s` by defining the function:

```
create function sqsum(Stream s) -> Number
  as foreign 'sqsum-+';
```

5. Foreign functions over records

Objects of OSQL type *Record* are collections representing sets of attribute/vale pairs. They are represented as handles with type id `recordtype`.

5.1. Functions returning records

The function *rsqrt2(Number x) -> Record roots* returns a record of the square roots of number *x*, `{"neg": -sqrt(x), "pos": sqrt(x)}`. It has the following implementation in `sa.engine/demo/Basic/C/Basic.c`:

```
ohandle rsqrt2BF(a_callcontext cxt)
{
  ohandle x = a_arg(cxt, 1);
  double dx;
  IntoDouble(x, dx, cxt->env);
  if(dx >= 0) {
    double root = sqrt(dx);
    ohandle r = new_record();
    record_put(r, "neg", mkreal(-root));
    record_put(r, "pos", mkreal(root));
    a_bind(cxt, 2, r);
    a_result(cxt);
  }
  return nil;
}
```

A handle holding a new empty record if created by calling `new_record()`.
The function

```
ohandle record_put(ohandle r, char *a, ohandle v)
```

Sets the attribute `a` of record `r` to value `v`.

The call

```
a_extimpl("rsqrt2-+", rsqrt2BF);
```

associates symbol `rsqrt2-+` with the implementation.

The OSQL definition is:

```
create function rsqrt2(Number x) -> Record roots
  as foreign 'rsqrt2-+'
```

5.2. Functions accessing records

The function *getnum(Record r, Charstring a) -> Number* accesses attribute a of record r as a number. It has the following implementation in `sa.engine/demo/Basic/C/Basic.c`:

```
ohandle getnumBBF(a_callcontext cxt)
{
  ohandle r = a_arg(cxt, 1);
  ohandle attr = a_arg(cxt, 2);
  ohandle val = record_get(r, getstring(attr));
  double x;
  if(val==nil) return nil;
  IntoDouble(val, x, cxt->env); // Just a type check
  a_bind(cxt, 3, val);          // Handle val is result
  a_result(cxt);
  return nil;
}
```

The function

```
ohandle record_get(ohandle r, char *a)
```

retrieves the value of attribute `a` in record `r`. The symbol `nil` is returned if there is no attribute `a` in `r`.

The call

```
a_extimpl("getnum--+", getnumBBF);
```

associates symbol `getnum--+` with the implementation.

The OSQL definition is:

```
create function getnum(Record r, Charstring field) -> Number
  as foreign 'getnum--+';
```

6. Foreign functions over streams

Objects of type stream are represented in C by handles having the type identifier `generatortype`.

6.1. Functions returning infinite streams

The same mechanism to iteratively generate elements of bags can be used for returning (possibly infinite) streams of elements. For example, the function *negative_numbers() -> Stream of Number* returns an infinite stream of the negetive numbers (integers from -1 and down). It is implemented in `sa.engine/demo/Streams/C/Streams.c` as:

```
ohandle negative_numbersF(a_callcontext cxt)
```

```
{
  int i;
  for(i=1;;i++)
    {
      a_bind(cxt,1,mkinteger(i));
      a_result(cxt);
    }
  return nil; // Never reached
}
```

The call

```
a_extimpl("negative-numbers+", negative_numbersF);
```

associates the symbol `natural_numbers+` with the implementation.

The OSQL definition is:

```
create function negative_numbers() -> Stream of Number
  as foreign "negative-numbers+";
```

If you call `negative_numbers()` from the console REPL the system will print natural numbers until you interrupt it with CTRL-C. The call `section(negative_numbers(), -10, -20)` will return a finite stream.

6.2. Stream transformation functions

A *stream transformation function* takes a stream as argument and produces a new transformed stream as result. For example, the function *power_stream(Stream s,Number n) -> Stream of Number* generates a stream of x^n of the numbers *x* in stream *s*. It has the following implementation in `sa.engine/demo/Streams/C/Streams.c`:

```
ohandle power_streamBBFmapper(a_callcontext cxt, int width, ohandle tuple[],
                              void *xa)
{
  double x;                   // Stream element
  double *exp = (double *)xa; // The exponent

  if(width > 0) {
    IntoDouble(tuple[0], x, cxt->env);
    a_bind(cxt, 3, mkreal(pow(x,*exp)));
    a_result(cxt);
  }
  return nil;
}
ohandle power_streamBBF(a_callcontext cxt)
{
  ohandle s = a_arg(cxt, 1);
  ohandle n = a_arg(cxt, 2);
  double exp;
  IntoDouble(n, exp, cxt->env);
  a_mapstream(cxt, s, power_streamBBFmapper, (void *)&exp);
  return nil;
}
```

Here, a_mapstream calls power_streamBBFmapper for each tuple in stream s. The parameter xa of a_mapstream is used for passing the address of number exp to the mapper.

The call

```
a_extimpl("power-stream--+", power_streamBBF);
```

associates the symbol power-stream--+ with the implementation power_streamBBF.

The OSQL definition is:

```
create function power_stream(Stream s, Number n) -> Stream of Number
  as foreign 'power-stream--+';
```

7. Non-blocking polling functions

The image is locked by the kernel while a foreign function is running so that the programmer can assume that all data in the image is available without further locking. Thus, foreign functions will block while waiting for some resource, making the system perform badly or even hang when foreign functions in several threads wait for some resource. This is not acceptable if a foreign function waits for some event to occur.

To circumvent this, sa.engine provides the ability for C-code sections in foreign functions to run in parallel in the *background*. Two functions are provided for this, a_enterbg() and a_leavebg(). C-code executed in-between calls to these functions is NOT locked and can therefore execute in parallel if several instances of the function is called in different threads. For example, non-blocking polling of events and sockets can be executed in such background sections. The typical code pattern is

```
ohandle mypollerBBF(a_callcontext cxt)
{
  ohandle event = a_arg(cxt, 1);
  int eid; // Numeric identifier for some event
  int vid; // Polled value of eid
  IntoInteger(event, eid, cxt->env),
  …
  a_enterbg();
  // Background code to poll eid and set vid to a received value
  a_leavebg();
  a_bind(cxt, 2, mkinteger(vid))
  return nil;
}
```

An important limitation is that sa.engine kernel functions accessing the database image are *not allowed* to be called in background code sections of foreign functions. This include object de-referencing, which is why eid is not used in the example background section above. Mainly only thread safe error handling functions are allowed in background code.

It should also be noted that a_enterbg() and a_leavebg() are rather slow so that it should be avoided to do background polling too often.

8. References

[1] aLisp User's Guide, Version 2.0, Stream Analyze Sweden AB, *sa_Lisp_2.0.pdf*, 2020.

[2] *Calling sa.engine from C applications*, Version 2.0, Stream Analyze Sweden, AB, *sa_C_pluginAPI_2.0.pdf*.

[3] *sa.engine Java Interfaces*, Version 2.2, Stream Analyze Sweden AB, *sa_JavaAPI_2.2.pdf*, 2020,

[4] sa.engine Lisp Interfaces, Version 2.1, Stream Analyze Sweden AB, *sa_LispAPI_2.1.pdf*, 2020.

[5] sa.Storage 2.0 - A main-memory storage manager, Version 2.0, Stream Analyze Sweden AB, *sa_Storage_2.0.pdf*, 2020