XS in C

Draft 2.0
July 17, 2014

1 Introduction

XS in C is the interface of xslib, a library that implements ECMAScript (ECMA-262).

Following the specifications, xslib implements only generic features that all scripts can use. Application define the specific features that its own scripts can use through C callbacks. An application that uses xslib is a host, in ECMAScript terminology.

The first part of the document, "Slot", tells how to handle ECMAScript constructs in C callbacks, with examples to hilite the correspondences between ECMAScript and the API of XS in C.

The second part of the document, "Machine", explains how to build a host step by step, how to make available C callbacks to scripts, and includes examples of a simple application.

2 Slot

In xslib, everything is stored in slots. A slot is an opaque structure that is only manipulated through XS in C.

```
typedef struct xsSlotRecord xsSlot
struct xsSlotRecord {
	void* data[4];
};
```

There are several types of slots.

```
enum {
	xsUndefinedType,
	xsNullType,
	xsBooleanType,
	xsIntegerType,
	xsNumberType,
	xsStringType,
	xsReferenceType
}
typedef char xsType;
```

The undefined, null, boolean, number and string slots correspond to the ECMAScript primitive types. The integer slot is a useful optimization, for scripts, it is equivalent to the number slot. The reference slot corresponds to the ECMAScript reference type.

The xsTypeOf macro returns the primitive type of a slot.

```
xsType xsTypeOf(xsSlot theSlot)
```

theSlotthe slot to test
returns the type of the slot

The xsTypeOf macro is similar to the ECMAScript typeof keyword.

In ECMAScript

```
switch(typeof arguments[0]) {
case "undefined": break;
/* null is object */
case "boolean": break;
/* integers are numbers */
case "number": break;
case "string": break;
case "object": break;
case "function": break;
}
```

In C

```
switch(xsTypeOf(xsArg(0))) {
case xsUndefinedType: break;
case xsNullType: break;
case xsBooleanType: break;
case xsIntegerType: break;
case xsNumberType: break;
case xsStringType: break;
case xsReferenceType: break;
/* objects and functions are references */
}
```

2.1 Primitives

The undefined, null, boolean, integer, number or string slots are direct slots or primitives. The undefined and null slots contain no value. The boolean, integer, number and string slot contain boolean, integer, number and string values.

```
typedef char xsBooleanValue;
typedef long xsIntegerValue;
typedef double xsNumberValue;
typedef char* xsStringValue;
```

When accessing the value of a slot, the slot is coerced to the requested type if necessary, then the requested value is returned.

```
xsSlot xsUndefined
```
returns an undefined slot

```
xsSlot xsNull
```
returns a null slot

```
xsSlot xsFalse
```
returns a boolean slot containing 0

```
xsSlot xsTrue
```
returns a boolean slot containing 1

```
xsSlot xsBoolean(xsBooleanValue theValue)
```
theValue the value to be contained in the slot
returns a boolean slot

```
xsBooleanValue xsToBoolean(xsSlot theSlot)
```
theSlotthe slot to coerce to boolean
returns the value contained in the slot

```
xsSlot xsInteger(xsIntegerValue theValue)
```
theValue the value to be contained in the slot
returns an integer slot

```
xsIntegerValue xsToInteger(xsSlot theSlot)
```

theSlotthe slot to coerce to integer
returns the value contained in the slot

```
xsSlot xsNumber(xsNumberValue theValue)
```
theValue the value to be contained in the slot
returns a number slot
```
xsNumberValue xsToNumber(xsSlot theSlot)
```
theSlotthe slot to coerce to number
returns the value contained in the slot

```
xsSlot xsString(xsStringValue theValue)
```
theValue the value to be contained in the slot
returns a string slot
```
xsStringValue xsToString(xsSlot theSlot)
```
theSlotthe slot to coerce to string
returns the value contained in the slot
```
xsStringValue xsToStringBuffer(xsSlot theSlot,
            xsStringValue theBuffer, xsIntegerValue theSize)
```
theSlotthe slot to coerce to string
theBuffer a buffer to copy the string value
theSize the size of the buffer
returns the buffer

A string value is a pointer to a UTF-8 C string. The virtual machine and the garbage collector of xslib manage all UTF-8 C strings used by scripts. C constants, C globals or C locals may be safely passed to the xsString macro since it duplicates its parameter.

For the sake of performance, the xsToString macro returns the value contained in the slot itself. Since xslib can compact string values anytime, the result of the xsToString macro cannot be used across or in other macros of XS in C and cannot be modified in place.

To get a copy of the string value use the xsToStringBuffer macro: the buffer has to be large enough to copy the string value.

In ECMAScript

```
undefined
null
false
true
0
0.0
"foo"
```

In C

```
xsUndefined;
xsNull;
xsFalse;
xsTrue;
xsInteger(0);
xsNumber(0.0);
xsString("foo");
```

2.2 Instances and Prototypes

The reference slots are indirect slots: they contain a reference to an instance of object, function, array, and so on... Instances themselves are made of slots, which are the properties or the items of the instance.

An instance usually has a prototype, which is an instance too. An instance overrides or shares the properties of its prototype:
-When accessing the property of an instance, the property is searched for in the instance. If the instance has the property, it is returned; otherwise the property is searched for in the prototype of the instance, and so on, until there is no prototype, in which case undefined is returned;
-When assigning the property of an instance, the property is searched for in the instance. If the instance has the property, it is set; otherwise the property is inserted into the instance;
-When deleting the property of an instance, the property is searched for in the instance. If the instance has the property, the property is removed from the instance.

XS in C defines macros to refer to the prototypes created by xslib.

```
xsSlot xsObjectPrototype
```
returns a reference to the prototype of object instances
```
xsSlot xsFunctionPrototype
```
returns a reference to the prototype of function instances
```
xsSlot xsArrayPrototype
```
returns a reference to the prototype of array instances
```
xsSlot xsDatePrototype
```
returns a reference to the prototype of date instances
```
xsSlot xsRegExpPrototype
```
returns a reference to the prototype of regexp instances
```
xsSlot xsErrorPrototype
```
returns a reference to the prototype of error instances

To create an instance, use the xsNewInstanceOf macro. Pass the prototype, which can be a prototype defined by xslib or an instance created by the host application.

```
xsSlot xsNewInstanceOf(xsSlot thePrototype)
```
thePrototype a reference to the prototype of the instance to create
returns a reference to the new instance

To test if an instance has a prototype, directly or indirectly, use the xsIsInstanceOf macro.

```
xsBooleanValue xsIsInstanceOf(xsSlot theInstance,
            xsSlot thePrototype)
```
theInstance a reference to the instance to test
thePrototype a reference to the prototype to test
returns 1 if the prototype is a prototype of the instance, else 0

The xsNewInstanceOf and xsIsInstanceOf macros have no equivalent in ECMAScript. To create or test instances, scripts invoke constructors with new or instanceof. Constructors are like functions but have a property identified by "prototype" that is used to create instances with new or to test instances with instanceof.

In ECMAScript

```
if (xsThis instanceof Object)
      return new Object();
```

In C

```
if (xsIsInstanceOf(xsThis, xsObjectPrototype))
      xsResult = xsNewInstanceOf(xsObjectPrototype);
```

2.3 Identifiers

For scripts, the properties of an instance are identified by a string, the items of an instance are identified by a number. For XS in C, all identifiers are indexes.

```
typedef short xsIndex;
```

The properties of an instance are identified by a negative index, the items of an instance are identified by a positive index.

XS in C defines a macro to convert a string value into an index.

```
xsIndex xsID(xsStringValue theValue)
```

theValue	the string to convert
returns	the identifier

With the same virtual machine, the same string value is always converted into the same index, so frequently used indexes may be stored.

In ECMAScript

```
this.foo
this[0]
```

In C

```
xsGet(xsThis, xsID("foo"));
xsGet(xsThis, 0);
```

Use the xsIsID macro to test if a string value is an identifier.

```
xsBooleanValue xsIsID(xsStringValue theValue)
```

theValue	the string to test
returns	0 or 1

2.4 Properties and Items

2.4.1 xsHas

To test if an instance has a property, use the xsHas macro.

```
xsBooleanValue xsHasID(xsSlot theThis, xsStringValue theValue)
```

theValue	a reference to the instance to test
returns	0 or 1

It is similar to the ECMAScript in keyword.

In ECMAScript

```
if ("foo" in this)
```

In C

```
if (xsHas(xsThis, xsID("foo"))
```

2.4.2 xsGet

To get the property of an instance, use the xsGet macro. If the property is undefined by the instance or its prototypes, the xsGet macro returns undefined.

```
xsSlot xsGet(xsSlot theThis, xsIndex theIndex)
```

theThis	a reference to the instance that has the property
theIndex	the identifer of the property to get
returns	a slot with what is contained in the property

In ECMAScript

```
foo
this.foo
this[0]
```

In C

```
xsGet(xsGlobal, xsID("foo"));
xsGet(xsThis, xsID("foo"));
xsGet(xsThis, 0);
```

2.4.3 xsSet

To set the property of an instance, use the xsSet macro. If the property is undefined by the instance, the xsSet macro inserts the property in the instance.

```
voif xsSet(xsSlot theThis, xsIndex theIndex, xsSlot theParam)
```

theThis	a reference to the instance that has the property
theIndex	the identifer of the property to set
theParam	a slot with what is to be contained in the property

In ECMAScript

```
foo = 0
this.foo = 1
this[0] = 2
```

In C

```
xsSet(xsGlobal, xsID("foo"), xsInteger(0));
xsSet(xsThis, xsID("foo"), xsInteger(1));
xsSet(xsThis, 0, xsInteger(2));
```

2.4.4 xsDelete

To delete the property of an instance, use the xsDelete macro. If the property is undefined by the instance, the xsDelete macro has no effect.

```
void xsDelete(xsSlot theThis, xsIndex theIndex)
```

theThis	a reference to the instance that has the property
theIndex	the identifier of the property to delete

In ECMAScript

```
delete foo
delete this.foo
delete this[0]
```

In C

```
xsDelete(xsGlobal, xsID("foo"));
xsDelete(xsThis, xsID("foo"));
xsDelete(xsThis, 0);
```

2.4.5 xsCall

When a property is a reference to a function, you can call the function with the xsCall* macros. If the property is undefined by the instance or its prototypes, or if the property is no reference to a function, the xsCall* macro throws an exception.

```
xsSlot xsCall0(xsSlot theThis, xsIndex theIndex)
xsSlot xsCall1(xsSlot theThis, xsIndex theIndex, xsSlot theParam0)
xsSlot xsCall2(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1)
xsSlot xsCall3(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2)
xsSlot xsCall4(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3)
xsSlot xsCall5(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3,
            xsSlot theParam4)
xsSlot xsCall6(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
```

```
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3,
            xsSlot theParam4, xsSlot theParam5)
xsSlot xsCall7(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3,
            xsSlot theParam4, xsSlot theParam5, xsSlot theParam6)
```

theThis	a reference to the instance that has the property
theIndex	the identifer of the property to call
theParam0	
theParam1	
theParam2	
theParam3	
theParam4	
theParam5	
theParam6	the parameters slots to pass to the function
returns	the result slot of the function

In ECMAScript

```
foo()
this.foo(1)
this[0](2, 3)
```

In C

```
xsCall0(xsGlobal, xsID("foo"));
xsCall1(xsThis, xsID("foo"), xsInteger(1));
xsCall2(xsThis, 0, xsInteger(2), xsInteger(3));
```

2.4.6 xsNew

When a property is a reference to a constructor, you can invoke the constructor with the xsNew* macros. If the property is undefined by the instance or its prototypes, or if the property is no reference to a constructor, the xsNew* macro throws an exception.

```
xsSlot xsNew0(xsSlot theThis, xsIndex theIndex)
xsSlot xsNew1(xsSlot theThis, xsIndex theIndex, xsSlot theParam0)
xsSlot xsNew2(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1)
xsSlot xsNew3(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2)
xsSlot xsNew4(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3)
xsSlot xsNew5(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3,
            xsSlot theParam4)
xsSlot xsNew6(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3,
            xsSlot theParam4, xsSlot theParam5)
xsSlot xsNew7(xsSlot theThis, xsIndex theIndex, xsSlot theParam0,
            xsSlot theParam1, xsSlot theParam2, xsSlot theParam3,
            xsSlot theParam4, xsSlot theParam5, xsSlot theParam6)
```

theThis	a reference to the instance that has the property
theIndex	the identifer of the property to invoke
theParam0	
theParam1	
theParam2	
theParam3	
theParam4	
theParam5	
theParam6	the parameters slots to pass to the constructor
returns	the result slot of the constructor

In ECMAScript

```
new foo()
new this.foo(1)
new this[0](2, 3)
```

In C

```
xsNew0(xsGlobal, xsID("foo"));
xsNew1(xsThis, xsID("foo"), xsInteger(1));
xsNew2(xsThis, 0, xsInteger(2), xsInteger(3));
```

2.4.7 xsTest

ECMAScript specifies which values are true or false depending on their type in test instructions and expressions,. Use the xsTest macro to get the same result in C.

```
xsBooleanValue xsTest(xsSlot theValue)
```

theValue	the value to test
returns	0 or 1

In ECMAScript

```
if (foo) {}
```

In C

```
if (xsTest(xsGet(xsGlobal, xsID("foo"))) {}
```

2.5 Indexes

ECMAScript provides two ways to access properties and items: the dot notation and the bracket notation. The bracket notation is the usual notation to access items of array but is also useful to access properties when the identifier is no constant.

The xsGet, xsSet and xsDelete macros can only access properties when their identifier is a constant and can only access items when their identifier is between 0 and 32767. To exceed such limits, XS in C defines the xsGetAt, xsSetAt and xsDeleteAt macros. They correspond to the ECMAScript bracket notation.

```
xsSlot xsGetAt(xsSlot theThis, xsSlot theIndex)
```

theThis	a reference to the instance that has the property
theIndex	the identifer of the property to get
returns	a slot with what is contained in the property

```
voif xsSetAt(xsSlot theThis, xsSlot theIndex, xsSlot theParam)
```

theThis	a reference to the instance that has the property
theIndex	the identifer of the property to set
theParam	a slot with what is to be contained in the property

```
void xsDeleteAt(xsSlot theThis, xsSlot theIndex)
```

theThis	a reference to the instance that has the property
theIndex	the identifier of the property to delete

In ECMAScript

```
this[2831993]
this["foo"]
this[foo]
```

In C

```
xsGetAt(xsThis, xsInteger(2831993));
xsGetAt(xsThis, xsString("foo"));
xsGetAt(xsThis, xsGet(xsGlobal, xsID("foo")));
```

2.6 Globals

Globals available to scripts are just properties of a special instance. XS in C defines the xsGlobal macro to refer to such a special instance

```
xsSlot xsGlobal
```
returns a reference to the instance made of globals

So to use globals, use the xsGet, xsSet, xsDelete, xsCall and xsNew macros with the xsGlobal macro as the this parameter.

In ECMAScript

```
foo
delete foo
foo()
new foo()
```

In C

```
xsGet(xsGlobal, xsID("foo"));
xsDelete(xsGlobal, xsID("foo"));
xsCall0(xsGlobal, xsID("foo"));
xsNew0(xsGlobal, xsID("foo"));
```

2.7 Arguments and Variables

The virtual machine of xslib uses a heap and a stack of slots. With XS in C, you can access directly stack slots and indirectly, through references, heap slots.

When a C callback is executed, the stack contains its arguments slots, its this slot, its result slot but no variables slots. To use variables slots, you have to reserve them on the stack at the beginning of the execution of the callback with the xsVars macro.

```
void xsVars(xsIntegerValue theCount)
```
theCount the number of variables slots to reserve.

Arguments and variables slots are accessed and assigned by index. An exception is thrown if the index is invalid.

Initially:
-the arguments slots are the parameters slots passed to the function or the constructor;
-if the callback is a function, the this slot refers to the instance being called and the result slot is undefined;
-if the callback is a constructor, the this and the result slots refer to the instance being created;
-the variables slots are undefined.

Nothing prevents scripts to call a constructor as a function or to invoke a function as a constructor. You can check if the result slot is initially undefined to know if the C callback is executed as a constructor or as a function.

```
xsSlot xsArgc
```
returns the integer slot that contains the number of arguments
```
xsSlot xsArg(xsIntegerValue theIndex)
```
theIndex the index of the argument, from 0 to xsArgc - 1
returns the argument slot
```
xsSlot xsThis
```
returns the this slot
```
xsSlot xsResult
```
returns the result slot
```
xsSlot xsVarc
```
returns the integer slot that contains the number of variables
```
xsSlot xsVar(xsIntegerValue theIndex)
```
theIndex the index of the variable, from 0 to xsVarc - 1

returns the variable slot

Usually you access the arguments, this, result and variables slots but you assign only the result and variables slots. What is in the result slot at the end of the execution of the callback is returned to scripts by the function or constructor.

In ECMAScript

```
function foo() {
	var c, i, s;
	c = arguments.length;
	s = "";
	for (i = 0; i < c; i++)
		s = s.concat(arguments[i]);
	return s;
}
```

In C

```
void xs_foo(xsMachine* the) {
	xsIntegerValue c, i;
	xsVars(1);
	c = xsToInteger(xsArgc));
	xsVar(0) = xsString("");
	for (i = 0; i < c; i++)
		xsVar(0) = xsCall1(xsVar(0), xsID("concat"), xsArg(i));
	xsResult = xsVar(0);
}
```

2.8 Garbage Collector

When xslib needs to create slots and there is not enough memory, it deletes automatically useless slots You can force xslib to delete useless slots with the xsCollectGarbage macro.

```
void xsCollectGarbage()
```

The garbage collector of xslib scans stack slots to mark and sweep heap slots. If you store slots in a C global or a C allocated structure, use the xsRemember and xsForget macro to tell xslib.

```
void xsRemember(xsSlot theSlot)
```

theSlotthe slot to remember.

```
void xsForget(xsSlot theSlot)
```

theSlotthe slot to forget.

xsRemember link and xsForget unlink a slot to and from a chain of slots. The garbage collector of xslib scans such a chain to mark the slots that the C global or the C allocated structure references.

In C

```
xsSlot gFooSlot;
void xsSetupFoo(xsMachine* the) {
	gFooSlot = xsThis;
	xsRemember(gFooSlot);
}
void xsCleanupFoo(xsMachine* the) {
	xsForget(gFooSlot);
}
```

2.9 Exceptions

To handle exceptions in C, xsLib uses setjmp, longjmp and a chain of jmp_buf.

```
typedef struct xsJumpRecord xsJump
struct JumpRecord {
      jmp_buf buffer;
      xsJump* nextJump;
      xsSlot* stack;
      xsSlot* frame;
};
```

You never need to use such a structure, XS in C defines macros to catch and throw exceptions.

To throw an exception, use the xsThrow macro.

```
void xsThrow(xsSlot theException)
```
      theException the exception slot

The xsThrow macro assigns the current exception. The xsException macro accesses the current exception.

```
xsSlot xsException
```
      returns   the exception slot

The xsTry and xsCatch macros have to be used together to catch exceptions

In ECMAScript

```
{
      try {
            /* If an exception is thrown here... */
      }
      catch(e) {
            /* ... it is caught here. */
            throw e
      }
}
```

In C

```
{
      xsTry {
            /* If an exception is thrown here... */
      }
      xsCatch {
            /* ... it is caught here. */
            xsThrow(xsException);
      }
}
```

If you catch an exception in your C callback and you want to propagate the exception to the script that invokes your constructor or calls your function, you have to throw the exception again.

2.10 Errors

Most exceptions will be thrown by C callbacks. C callbacks are the interface between scripts and systems. A lot of system calls can fail and have a way to return an error to the application.

To throw exceptions in case of errors, use the xsError macro or, to avoid redundant code in the C callbacks, the xsIfError or xsElseError macros.

```
void xsError(xsIntegerValue theError)
```
theError the error number.

The exception will be an instance of Error with a message based on theError.

In C
```
anError = FskMemPtrNew(1024, &aBuffer));
if (anError) xsError(anError);
```

The xsIfError macro is a shortcut for system calls that return zero if they succeeded and non-zero if they failed.

```
void xsIfError(xsIntegerValue theError)
```
theError if different of zero, trigger an exception.

The exception will be an instance of Error with a message based on theError.

In C
```
xsIfError(FskMemPtrNew(1024, &aBuffer));
xsIfError(FskFileRead(aFile, 1024, aBuffer, &aSize));
```

The xsElseError macro is a shortcut for system calls that return zero if they failed and non-zero if they succeeded.

```
void xsElseError(xsIntegerValue theAssertion)
```
theAssertion if equal to zero, trigger an exception.

The exception will be an instance of Error with a message based on GetLastError() on Windows and on errno elsewhere.

In C
```
xsElseError(aBuffer = malloc(1024));
xsElseError(ReadFile(aFile, aBuffer, 1024, &aSize, NULL));
```

2.11 Debugger

XS in C provides two macros to help you to debug your C callbacks.

The xsDebugger macro is equivalent to the ECMAScript debugger keyword.

```
void xsDebugger()
```

The xsTrace macro is equivalent to the ECMAScript trace function.

```
void xsTrace(xsStringValue theMessage)
```
theMessage the message to log in the debugger

In ECMAScript
```
trace("Hello xsbug!\n");
debugger;
```
In C
```
xsTrace("Hello xsbug!\n");
xsDebugger();
```

3 Machine

The main structure of xslib is its virtual machine. It is the virtual machine that parses, compiles, links and executes scripts. A virtual machine is a mostly opaque structure. Some members of the structure are available to optimize the macros of XS in C, you never need to use them.

```
typedef struct xsMachineRecord xsMachine
struct xsMachineRecord {
	xsSlot* stack;
	xsSlot* stackBotton;
	xsSlot* stackTop;
	xsSlot* frame;
	xsJump* firstJump;
};
```

A single machine does not support multiple threads. But nothing prevents you to create several machines...

Together, the examples below make a simple application. Its first argument is the script to execute. C callbacks are provided to access its next arguments and to read and write files. For instance it can run a script like:

```
src = new File(argv[2], "r");
dst = new File(argv[3], "w");
while(buf = src.getLine())
	dst.writeLine(buf.toLowerCase());
```

You can use such a simple application as a basis to experiment, to test features of xslib, to provide features to scripts with XS in C, and so on...

3.1 Allocation

To use xslib you have to create a machine with the xsNewMachine macro.

Its first argument is a structure with various members to allocate the machine. Pass NULL if you want to use the defaults. Its second argument is the grammar to load documents. It is a structure generated by xsc. Pass NULL if you want to load no documents. Its third argument is a context you can get and set in your callbacks. Pass NULL if you want no context initially.

```
xsMachine* xsNewMachine(xsAllocation* theAllocation,
		xsGrammar* theGrammar, void* theContext)
```

theAllocation	the parameters of the machine
theGrammar	the main grammar of the machine or NULL
theContext	the initial context of the machine or NULL
returns	a machine if successful, else NULL

```
typedef struct {
	xsIntegerValue initialChunkSize;
	xsIntegerValue incrementalChunkSize;
	xsIntegerValue initialHeapCount;
	xsIntegerValue incrementalHeapCount;
	xsIntegerValue stackCount;
	xsIntegerValue symbolCount;
	xsIntegerValue symbolModulo;
} xsAllocation;
```

A machine manages strings and bytecodes in chunks. The initial chunk size is the initial size of the memory allocated to chunks. The incremental chunk size tells xslib how to grow the memory allocated to chunks.

A machine uses a heap and a stack of slots. The initial heap count is the initial number of slots allocated to the heap. The incremental heap count tells xslib how to grow the number of slots allocated to the heap. The stack count is the number of slots allocated to the stack.

The symbol count is the number of symbols the machine will use. The symbol modulo is the size of of the hash table the machine will use for symbols. A symbol binds a string value and an identifier, see the xsID macro.

When you are done with a machine, you have to free it with the xsDeleteMachine macro. The destructors of all the host objects are executed and all the memory allocated by the machine is freed.

```
void xsDeleteMachine(xsMachine* the)
```
the	a machine

Example

```
int main(int argc, char* argv[])
{
	xsAllocation anAllocation = {
		32 * 1024,		/* initialChunkSize */
		16 * 1024,		/* incrementalChunkSize */
		2048,			/* initialHeapCount */
		1024,			/* incrementalHeapCount */
		1024,			/* stackCount */
		4096,			/* symbolCount */
		1993			/* symbolModulo */
	};
	xsMachine* aMachine;

	aMachine = xsNewMachine(&anAllocation, NULL, NULL);
	if (aMachine) {
		mainContext(aMachine, argc, argv); /* see 3.2 */
		xsDeleteMachine(aMachine);
	}
	else
		fprintf(stderr, "### Cannot allocate machine\n");
	return 0;
}
```

3.2 Context

Your C code will be used by the machine mostly though callbacks. You can get and set a context from and to the machine to retrieve and store information in your callbacks.

```
void* xsGetContext(xsMachine* the)
```
the	a machine
returns	a context

```
void xsSetContext(xsMachine* the, void* theContext)
```
the	a machine
theContext	a context

Example

```
typedef struct {
	int argc;
	char** argv;
} Context;

void xsMainContext(xsMachine* theMachine, int argc, char* argv[])
{
	Context* aContext;

	aContext = malloc(sizeof(Context));
	if (aContext) {
		aContext->argc = argc;
		aContext->argv = argv;
		xsSetContext(theMachine, aContext);
		xsMainHost(theMachine, argc, argv); /* see 3.3 */
		xsSetContext(theMachine, NULL);
		free(aContext);
	}
	else
		fprintf(stderr, "### Cannot allocate context\n");
}
```

3.3 Host

3.3.1 xsBuildHost

Once you have a machine and a context, you need to build the host with the xsBuildHost macro.

The xsBuildHost macro setup the stack, executes the callback you pass and cleanup the stack.

```
typedef void (*xsCallback)(xsMachine* the);
```

It is in such a callback that you can build instances like host constructors, host functions or host objects and make them available to scripts.

You can use the xsBuildHost macro at the beginning of your application to initialize the host, and during the execution of the application to change the host.

```
xsBooleanValue xsBuildHost(xsMachine* the, xsCallback theCallback)
```

the	a machine
theCallback	a callback to execute to build the host
returns	1 if succesful, else 0

3.3.2 xsNewHostContructor

For a script, a host constructor is just like a constructor. But when a script invokes a host contructor, a callback is executed.

```
typedef void (*xsCallback)(xsMachine* the);
```

To create a host constructor, use the xsNewHostContructor macro.

```
xsSlot xsNewHostContructor(xsCallback theCallback,
		xsIntegerValue theLength, xsSlot thePrototype)
```

theCallback	the callback to execute
theLength	the number of parameters expected by the callback
thePrototype	a reference to the prototype of the instance to create
returns	a reference to the host contructor

3.3.3 xsNewHostFunction

For a script, a host function is just like a function. But when a script calls a host function, a callback is executed.

```
typedef void (*xsCallback)(xsMachine* the);
```

To create a host function, use the xsNewHostFunction macro.

```
xsSlot xsNewHostFunction(xsCallback theCallback,
            xsIntegerValue theLength);
```

theCallback	the callback to execute
theLength	the number of parameters expected by the callback
returns	a reference to the host function

3.3.4 xsNewHostObject

A host object is a special kind of object with data that can only be handled in C. Such data is invisible to scripts.

When the garbage collector is about to get rid of a host object, it executes its destructor, if any. No reference to the host object is passed to the destructor: a destructor can only destroy data.

```
typedef void (xsDestructor)(void* theData);
```

To create a host object, use the xsNewHostObject macro. You can pass its destructor or NULL if the host object does not need a destructor.

```
xsSlot xsNewHostObject(xsDestructor theDestructor)
```

xsDestructor	the destructor to be executed by the garbage collector
returns	a reference to the host object

To get and set the data of a host object, use the xsGetHostData and xsSetHostData macros. Both throw an exception if the this parameter does not refer to a host object.

```
void* xsGetHostData(xsSlot theThis)
```

theThis	a reference to a host object
returns	the data

```
void xsSetHostData(xsSlot theThis, void* theData)
```

theThis	a reference to a host object
theData	the data

3.3.5 xsNewHostProperty

To create properties of host objects, to create globals available to scripts, use the xsNewHostProperty macro. Like the xsSet macro, if the property is undefined by the instance, the xsNewHostProperty macro inserts the property in the instance. But, unlike the xsSet macro, the xsNewHostProperty macro changes the attributes of the property.

```
enum {
      xsDefault = 0,
      xsDontDelete = 2,
      xsDontEnum = 4,
      xsDontScript = 8
      xsDontSet = 16,
      xsIsGetter = 32,
      xsIsSetter = 64,
      xsChangeAll = 30
}
typedef unsigned char xsAttribute;
```

```
void xsNewHostProperty(xsSlot theThis, xsIndex theIndex, xsSlot theValue,
            xsAttribute theAttributes, xsAttribute theMasks)
```

theThis	a reference to the instance that will have the property
theIndex	the identifer of the property to create
theValue	a slot with what will be contained in the property
theAttributes	a combination of attributes to set
theMasks	a combination of attributes to set or to clear

The delete, enum and set attributes are the ECMAScript attributes. The script attribute defines if the property is visible or invisible to scripts. By default a property can be deleted, enumerated and set, and can be used by scripts.

When a property is created, if the prototype of the instance has a property with the same name, its attributes are inherited, else, by default, a property can be deleted, enumerated and set, and can be used by scripts.

The xsNewHostProperty macro can also be used to create getters and setters. See "XS" for details. Then the value parameter has to be a function and both the attributes and the masks parameter have to be xsIsGetter or xsIsSetter.

Example

```
void xsMainHost(xsMachine* theMachine, int argc, char* argv[])
{
      if (xsBuildHost(theMachine, xsMainHostCallback))
            xsMainExecute(theMachine, argc, argv); /* see 3.4 */
      else
            fprintf(stderr, "### Cannot build host\n");
}

void xsMainHostCallback(xsMachine* the)
{
      xsContext* aContext = xsGetContext(the);
      int argi;

      /* Build an array with the arguments... */
      xsResult = xsNewObject(xsArrayPrototype);
      for (argi = 0; argi < aContext->argc; argi++)
            xsSet(xsResult, argi, xsString(aContext->argv[argi]));
      /* ...and make it available as argv. */
      xsSet(xsGlobal, xsID("argv"), xsResult);

      /* Build a host object to embed a file... */
      xsResult = xsNewHostObject(xs_FileDestructor);
      /* ...with properties to get and put lines... */
      xsNewHostProperty(xsResult, xsID("getLine"),
            xsNewHostFunction(xs_FileGetLine, 0),
                  xsDontDelete | xsDontEnum | xsDontSet, xsChangeAll);
      xsNewHostProperty(xsResult, xsID("putLine"),
            xsNewHostFunction(xs_FilePutLine, 0),
                  xsDontDelete | xsDontEnum | xsDontSet, xsChangeAll);
      /* ...and make it available as a File constructor. */
      xsNewHostProperty(xsGlobal, xsID("File"),
            xsNewHostConstructor(xs_File, 2, xsResult),
                  xsDontDelete | xsDontEnum | xsDontSet, xsChangeAll);
}

void xs_File(xsMachine* the)
{
```

```
	FILE* aFile = fopen(xsToString(xsArg(0)), xsToString(xsArg(1)));
	xsSetHostData(xsThis, aFile);
}

void xs_FileDestructor(void* theData)
{
	if (theData)
		fclose(theData);
}

void xs_FileGetLine(xsMachine* the)
{
	FILE* aFile = xsGetHostData(xsThis);
	char aLine[1024];
	if (fgets(aLine, sizeof(aLine) - 1, aFile))
		xsResult = xsString(aLine);
}

void xs_FilePutLine(xsMachine* the)
{
	FILE* aFile = xsGetHostData(xsThis);
	fputs(xsToString(xsArg(0)), aFile);
}
```

3.3.6 xsBeginHost and xsEndHost

When it is inconvenient to retrieve and store information through the context, for instance in a system callback, use the xsBeginHost and xsEndHost macros instead of the xsBuildHost macro.

The xsBeginHost macro setup the stack and the xsEndHost macro cleanup the stack, so that you can use all the macros of XS in C in the block between xsBeginHost and xsEndHost.

Example

```
long FAR PASCAL xsWndProc(HWND hwnd, UINT m, UINT w, LONG l)
{
	long result = 0;
	xsMachine* aMachine = GetWindowLongPtr(hwnd, GWL_USERDATA);
	xsBeginHost(aMachine);
	{
		result = xsToInteger(xsCall3(xsGlobal, xsID("dispatch"),
				xsInteger(m), xsInteger(w), xsInteger(l)));
	}
	xsEndHost(aMachine);
	return result;
}
```

3.4 Scripts

Once you created a machine and built a host, you can execute scripts with the xsExecute macro. The xsExecute macro takes a stream and a getter. If the script is a file, the stream can be a FILE* and the getter can be fgetc, but it can be anything else with an equivalent behavior.

```
typedef int (*xsGetter)(void* theStream);

xsBooleanValue xsExecute(xsMachine* the, void* theStream,
		xsGetter theGetter, xsStringValue thePath,
		xsIntegerValue theLine)
```

the	a machine
theStream	where to get characters
theGetter	how to get characters
thePath	the current path of the stream
theLine	the current line of the stream
returns	1 if successful, else 0

The current path and line are only used to report errors and warnings. Pass NULL and 0 they do not make sense in your application.

Example

```
void xsMainExecute(xsMachine* theMachine, int argc, char* argv[])
{
	FILE* aFile;

	aFile = fopen(argv[1], "r");
	if (aFile) {
		if (xsExecute(theMachine, aFile, fgetc, argv[1], 1)))
			fprintf(stderr, "### %s: O.K.\n", argv[1]);
		else
			fprintf(stderr, "### Cannot execute %s\n", argv[1]);
		fclose(aFile);
	}
	else
		fprintf(stderr, "### Cannot open %s\n", argv[1]);
}
```

3.5 Grammars

3.5.1 xsLink

A grammar is the structure generated by xsc when you compile and link XS and C files into an application or a dynamic library. See "XS" for details.

```
typedef struct {
	xsCallback callback;
	xsStringValue symbols;
	xsIntegerValue symbolsSize;
	xsStringValue code;
	xsIntegerValue codeSize;
	xsStringValue name;
} xsGrammar;
```

Grammars prototype ECMAScript instances, parse XML documents into ECMAScript instances and serialize ECMAScript instances into XML documents.

The xsNewMachine macro takes one grammar, the grammar of the application. To link grammars dynamically, for instance grammars of plug-ins, use the xsLink macro.

```
void xsLink(xsGrammar* theGrammar)
```

theGrammar the grammar to link

If the machine cannot link the grammar, for instance if the already linked grammars are incompatible, the xsLink macro throws an exception.

To use the xsLink macro, first load the dynamic library, then look for the grammar in the symbols of the dynamic library and pass it to the xsLink macro.

Example

```
void linkLibrary(xsMachine* the, char* thePath, char* theName)
{
#if mxWindows
	HINSTANCE anInstance;
	char aName[256];
	xsGrammar* aGrammar;

	anInstance = LoadLibrary(thePath);
	xsElseError(anInstance != NULL);
	strcpy(aName, theName);
	strcat(aName, "Grammar");
	aGrammar = (xsGrammar*)GetProcAddress(anInstance, aName);
	xsElseError(aGrammar != NULL);
	xsLink(aGrammar);
#elif mxMacOSX
	const struct mach_header *anImage;
	char aName[256];
	NSSymbol aSymbol;
	xsGrammar* aGrammar;
	NSLinkEditErrors errors;
	int aNumber;
	const char* aFile;
	const char* aMessage;

	anImage = NSAddImage(thePath, NSADDIMAGE_OPTION_RETURN_ON_ERROR);
	if (anImage == NULL) goto error;
	strcpy(aName, "_");
	strcat(aName, theName);
	strcat(aName, "Grammar");
	aSymbol = NSLookupSymbolInImage(anImage, aName,
			NSLOOKUPSYMBOLINIMAGE_OPTION_BIND_NOW |
			NSLOOKUPSYMBOLINIMAGE_OPTION_RETURN_ON_ERROR);
	if (aSymbol == NULL) goto error;
	aGrammar = NSAddressOfSymbol(aSymbol);
	if (aGrammar == NULL) goto error;
	xsLink(aGrammar);
	return;
error:
	NSLinkEditError(&errors, &aNumber, &aFile, &aMessage);
	xsThrow(xsNew1(xsGlobal, xsID("Error"), xsString((char*)aMessage)));
#else
	void* aHandle;
	char aName[256];
	xsGrammar* aGrammar;

	aHandle = dlopen(thePath, RTLD_NOW);
	if (aHandle == NULL) goto error;
	strcpy(aName, theName);
	strcat(aName, "Grammar");
	aGrammar = dlsym(aHandle, aName);
	if (aGrammar == NULL) goto error;
	xsLink(aGrammar);
	return;
error:
	xsThrow(xsNew1(xsGlobal, xsID("Error"), xsString(dlerror())));
#endif
}
```

Alternatively, if the plug-in needs no C callback, xsc can build a binary file instead of a dynamic library. The binary file is atom based. To use the xsLink macro, first load the binary file, then fill a grammar with the atoms and pass it to the xsLink macro.

Example

```
void linkBinary(xsMachine* the, char* thePath)
{
#define XS_ATOM_CONTAINER 0x58533131 /* 'XS11' */
#define XS_ATOM_SYMBOLS 0x53594D42 /* 'SYMB' */
#define XS_ATOM_CODE 0x434F4445 /* 'CODE' */
	typedef struct {
		long atomSize;
		unsigned long atomType;
	} Atom;
	FILE* aFile = NULL;
	size_t aSize;
	char* aBuffer = NULL;
	xsGrammar aGrammar;
	char* aPointer;
	Atom anAtom;

	xsTry {
		aFile = fopen(thePath, "rb");
		xsElseError(aFile != NULL);
		xsElseError(fseek(aFile, 0, SEEK_END) == 0);
		aSize = ftell(aFile);
		xsElseError(fseek(aFile, 0, SEEK_SET) == 0);
		aBuffer = malloc(aSize);
		xsElseError(aBuffer != NULL);
		xsElseError(fread(aBuffer, aSize, 1, aFile) == 1);
		aPointer = aBuffer;
		anAtom.atomSize = ntohl(((Atom*)aPointer)->atomSize);
		anAtom.atomType = ntohl(((Atom*)aPointer)->atomType);
		if (anAtom.atomType != XS_ATOM_CONTAINER)
			xsError(EINVAL);
		aPointer += sizeof(anAtom);
		anAtom.atomSize = ntohl(((Atom*)aPointer)->atomSize);
		anAtom.atomType = ntohl(((Atom*)aPointer)->atomType);
		if (anAtom.atomType != XS_ATOM_SYMBOLS)
			xsError(EINVAL);
		aGrammar.symbols = aPointer + sizeof(anAtom);
		aGrammar.symbolsSize = anAtom.atomSize - sizeof(anAtom);
		aPointer += anAtom.atomSize;
		anAtom.atomSize = ntohl(((Atom*)aPointer)->atomSize);
		anAtom.atomType = ntohl(((Atom*)aPointer)->atomType);
		if (anAtom.atomType != XS_ATOM_CODE)
			xsError(EINVAL);
		aGrammar.code = aPointer + sizeof(anAtom);
		aGrammar.codeSize = anAtom.atomSize - sizeof(anAtom);
		aGrammar.callback = NULL;
		aGrammar.name = NULL;
		xsLink(&aGrammar);
		free(aBuffer);
		aBuffer = NULL;
		fclose(aFile);
		aFile = NULL;
	}
	xsCatch {
		if (aBuffer != NULL)
			free(aBuffer);
```

```
            if (aFile != NULL)
                  fclose(aFile);
            xsThrow(xsException);
      }
}
```

3.5.2 xsParse and xsParseBuffer

If you provided grammars with the xsNewMachine and/or xsLink macros, you can load XML documents with the xsParse or xsParseBuffer macros.

The xsParse macro takes a stream and a getter and returns the root of the XML document. If the document is a file, the stream can be a FILE* and the getter can be fgetc, but it can be anything else with an equivalent behavior.

```
typedef int (*xsGetter)(void* theStream);

enum {
      xsSourceFlag = 1,
      xsNoErrorFlag = 2,
      xsNoWarningFlag = 4,
      xsDebugFlag = 128
};
typedef unsigned char xsFlag;

xsSlot xsParse(void* theStream, xsGetter theGetter,
            xsStringValue thePath, xsIntegerValue theLine,
            xsFlag theFlags)
```

the	a machine
theStream	where to get characters
theGetter	how to get characters
thePath	the current path of the stream
theLine	the current line of the stream
theFlags	a combination of flags
returns	the root of the document

The xsParseBuffer macro takes a buffer and a size and returns the root of the XML document.

```
xsSlot xsParseBuffer(xsStringValue theBuffer, xsIntegerValue theSize,
            xsStringValue thePath, xsIntegerValue theLine,
            xsBooleanValue theFlags)
```

the	a machine
theBuffer	the characters
theSize	the number of characters
thePath	the current path of the buffer
theLine	the current line of the buffer
theFlags	a combination of flags
returns	the root of the document

The current path and line are only used to report errors and warnings. Pass NULL and 0 if they do not make sense in your application. Use the xsSourceFlag to keep the sources of the parsed functions, see “XS” for details. Use the xsDebugFlag to see the sources of the parsed functions in xsbug.

The parser detects unknown namespaces, unknown elements, unknown attributes and unknown processing instructions; redundant elements and redundant attributes; missing namespaces in elements, attributes and processing instructions. Depending on the flags

passed to xsParse or xsParseBuffer, xslib signals an error, reports a warning or ignore the problem.

Example

```
xsResult = xsParse(aFile, fgetc, aPath, aLine,
                   xsSourceFlag | xsNoErrorFlag | xsNoWarningFlag);
```

If the XML document cannot be parsed by the grammars or if the machine has no grammar, the xsParse and xsParseBuffer macros throw an exception.

The xsParse* and xsParseBuffer* macros (* is 1 to 8) are variants of the xsParse and xsParseBuffer macros. The additional parameters are prototypes that constraint the kind of XML document accepted by the parser. Such prototypes must have a root pattern or xslib signals an error.

3.5.3 xsSerialize and xsSerializeBuffer

If you provided grammars with the xsNewMachine and/or xsLink macros, you can serialize XML documents with the xsSerialize or xsSerializeBuffer macros.

The xsSerialize macro takes a stream and a putter. If the XML document is a file, the stream can be a FILE* and the putter can be fputs, but it can be anything else with an equivalent behavior.

```
typedef int (*xsPutter)(xsStringValue theString, void* theStream);

void xsSerialize(xsSlot theRoot, void* theStream, xsPutter thePutter)
```

theRoot	the root of the document
theStream	where to put strings
thePutter	how to put strings

The xsSerializeBuffer macro takes a buffer and a size and returns the required size. The buffer can be NULL and the size can be 0 to measure the XML document.

```
xsIntegerValue xsSerializeBuffer(xsSlot theRoot,
            void* theBuffer, xsIntegerValue theSize)
```

theRoot	the root of the document
theBuffer	the characters
theSize	the number of characters
returns	the required number of characters

If the XML document cannot be serialized by the grammars or if the machine has no grammar, the xsSerialize and xsSerializeBuffer macros throw an exception.

3.5.4 xsScript and xsSandbox

If you provided grammars with the xsNewMachine and/or xsLink macros, the xlib library can execute code out of the sandbox (framework code) or in the sandbox (script code), see XS for details.

At runtime, a callback can test if it has been called by script code, directly or indirectly with the xsScript macro.

```
xsIntegerValue xsScript()
```

returns	the depth of the script code call, 0 if no script call

The result of the test is a number, 0 means called by framework code, 1 means called directly by script code, more than 1 means called indirectly by script code.

In C, framework code can use properties created at runtime by script code with the xsSandbox macro. It is similar to the Object.prototype.sandbox property in ECMAScript, see XS for details.

```
xsSlot xsSandbox(xsSlot theThis)
```

theThis	a reference to an instance
returns	a handle to use runtime properties created by code

The handle can be cached in a variable like any slot.

In ECMAScript

```
this.sandbox.foo
delete this.sandbox.foo
this.sandbox.foo()
new this.sandbox.foo()
```

In C

```
xsGet(xsSandbox(xsThis), xsID("foo"));
xsDelete(xsSandbox(xsThis), xsID("foo"));
xsCall0(xsSandbox(xsThis), xsID("foo"));
xsNew0(xsSandbox(xsThis), xsID("foo"));
```

3.6 "the"

The xsNewMachine, xsDeleteMachine, xsGetContext, xsSetContext, xsBuildHost, xsBeginHost, xsEndHost and xsExecute macros are the only macros that have an explicit machine parameter named "the" because they are the only macros that can be used outside of a callback and can throw no exception.

All callbacks must name their machine parameter "the" since all other macros have an implicit parameter named "the". Because of terseness of course, but also because such a trivial convention emphasizes the fact that all other macros can only be used inside a callback and can throw exceptions.

---

[i] This document is part of xslib.