Alien Foreign Function Interface User Guide

Eliot Miranda
2007

This document describes the Alien Foreign Function Interface (Alien FFI) for potential users. A foreign function interface is a system for interfacing to the external platform's OS and libraries which on relevant systems are defined according to a C language semantics specified by a platform's Application Binary Interface (ABI). An FFI can be used to invoke external functionality in libraries and in the operating system such as GUIs, file-system and network interfaces and so on. It must support representation of C data, making call-outs to external functions and allowing call-backs from external code into Smalltalk.

The Alien FFI is a minimal foreign function interface for Newspeak and/or Squeak on IA32 (Intel x86) platforms. The FFI is "lean and mean", providing no support beyond IA32 or 32-bits, and providing limited safety and abstraction. But it is fully functional, simple and relatively performant.

Beware all ye who enter here, there be beasties. The FFI does not provide type safety (checking that parameters to external calls are conformant with the functions called), doesn't hide many details of IA32 ABIs, and doesn't isolate the user from differences in the ABI on different OSs. The user is responsible for calling functions correctly. This document tries to describe these pitfalls, but you, dear reader, are responsible for avoiding them.

Core Classes

Instances of **Alien** represent external "C" data. There are three different kinds:

Direct: some number of bytes held in the Alien itself. Can be passed by value to external code, but cannot have its address taken since the Squeak garbage collector can and does move objects. Direct Aliens are created via new and new: dataSize <Integer>. Their data are initialized with zeros.

Indirect: some number of bytes held in the external C heap. The indirect Alien contains the size of the data and the data's address. Indirect aliens are created via newC, newC: dataSize <Integer>, newGC, newGC: dataSize <Integer>, atAddress: address <Integer> and atAddress: address <Integer> dataSize: dataSize <Integer>. Their data are initialized with zeros.

Pointer: some data of indeterminate size held in the external C heap. The indirect Alien contains the data's address, but not its size. Pointer Aliens are created via forPointer: address <Integer>. Whereas Indirect Aliens pass their data when used as arguments in FFI calls, Pointer Aliens pass their pointer, but both allow their data to be accessed via the various accessing primitives.

Alien defines a set of indexing primitives for accessing an instance's external data. Although the three kinds of Alien manage their data in different ways the data can be accessed uniformly by these primitives. The access primitives will access the Alien's data independent of the Alien being direct, indirect or pointer. However, access to pointer Aliens is not bounds-checked, which is

both convenient and dangerous. Although this could be changed we value the convenience higher. N.B. All indices are 1-relative in keeping with Smalltalk, *not* 0-relative as in keeping with C.

There are primitives for integer, floating-point and byte-oriented access:

<Alien> [**un**]**signedByteAt:** index <Integer> [**put:** value <Integer>] ^<Integer>
<Alien> [**un**]**signedShortAt:** index <Integer> [**put:** value <Integer>] ^<Integer>
<Alien> [**un**]**signedLongAt:** index <Integer> [**put:** value <Integer>] ^<Integer>
These get or put a [un]signed integer of a given byte size, failing if the indices through the datum size are out of range (pace pointer Aliens), if the value stored is out of range, or if any argument is mistyped.

<Alien> **doubleAt:** index <Integer> ^<Float>
<Alien> **doubleAt:** index <Integer> **put:** value <Float | Integer> ^<Float | Integer>
<Alien> **floatAt:** index <Integer> ^<Float>
<Alien> **floatAt:** index <Integer> **put:** value <Float | Integer> ^<Float | Integer>
These get or put a double-precision (64-bit) or single-precision (32-bit) IEEE float, failing if the indices through the datum size are out of range (pace pointer Aliens), if the value stored is out of range, or if any argument is mistyped. Integers are coerced to floating-point automatically when storing.

<Alien> **replaceFrom:** start <Integer> **to:** stop <Integer>
with: replacement <Alien | byteIndexable> **startingAt:** repStart <Integer> ^<self>
<Alien> **copyInto:** aByteObject <Alien | byteIndexable> **from:** start <Integer> **to:** stop <Integer>
in: replacement <Alien | byteIndexable> **startingAt:** repStart <Integer> ^<self>
These copy some number of bytes from one object to another, replaceFrom:... copying into the receiver, copyInto:... storing into the first argument, failing if the indices through the number of bytes to be copied are out of range (pace pointer Aliens), or if any argument is mistyped.

There are also primitives for calls out to C functions, address manipulation, memory management and instance initialization, and convenience protocol for string manipulation that we'll present later on.

Representing External Datatypes

Alien is designed to be subclassed in order to define specific external datatypes more conveniently. A subclass can provide accessors for fields, wrapping the low-level accessing primitives above, and freeing the client from details of a datatype's layout. On the class side a subclass should define the method dataSize to define the size, in bytes, of an instance's datum (we recommend the convention of including the C definition in the class comment and the class-side dataSize method). e.g. here are some C definitions from Mac OS X's GUI framework.

```
typedef struct _NSPoint {
   float x;
   float y;
} NSPoint;
```

```
typedef struct _NSSize {
   float width;
   float height;
} NSSize;

typedef struct _NSRect {
   NSPoint origin;
   NSSize size;
} NSRect;
```

These can be represented by two classes, NSPoint for `NSPoint` and `NSSize`, and NSRect for `NSRect`. As Newsqueak doesn't currently support variable-byte classes we must use Smalltalk:

```
Alien variableByteSubclass: #NSPoint
NSPoint class methods for instance creation
dataSize
   ^8

NSPoint variableByteSubclass: #NSRect
NSRect class methods for instance creation
dataSize
   ^16
```

These classes can then provide accessors for the various fields, some of which are:

```
NSPoint methods for accessing
x
   ^self floatAt: 1
x: aFloat
   self floatAt: 1 put: aFloat asFloat "asFloat to coerce fractions"
height
   ^self floatAt: 5
height: aFloat
   self floatAt: 5 put: aFloat asFloat

NSRect methods for accessing
height
   ^self floatAt: 13
height: aFloat
   self floatAt: 13 put: aFloat asFloat
```

Alien subclasses can then provide convenience methods for instance creation:

```
NSRect class methods for instance creation
x: x y: y width: width height: height
   ^self new x: x; y: y; width: width; height: height; yourself
```

This method creates a direct Alien of size 16 bytes that lives on the Smalltalk heap. The result can be passed by value to some function expecting an `NSRect struct` by value. There is no need to allocate C data for these by-value parameters, simplifying memory-management and

gaining a little performance. If the method used newC instead of new then the data would be allocated on the C heap and an indirect Alien would be returned. The storage would not be reclaimed unless the Alien is sent the message free. If the method used newGC then the storage would also be allocated on the C heap but finalization would arrange that the storage was reclaimed some time after the Alien was garbage collected.

Initializing Aliens

All instance creation methods that allocate data, new, new:, newC, newC:, newGC, newGC: (but *not* atAddress:, atAddress:dataSize:, or forPointer:), send initialize to the Alien instance after storage is allocated. The initialize method can be used to initialize the storage as required. Here is a real example from `Win32` where the `lStructSizeOffset` field of an `OPENFILENAME` is used by `Win32` to determine what version of `OPENFILENAME` is being used, and where the storage for the filename is referred to by a pointer embedded in the struct.

```
Alien variableByteSubclass: #OPENFILENAME
OPENFILENAME class methods for instance creation
dataSize
    "sizeof(OPENFILENAME) in Windows 5 (76 in Windows 4)"
    ^88
OPENFILENAME methods for instance initialization
initialize
    self
    unsignedLongAt: self lStructSizeOffset put: self class dataSize;
    unsignedLongAt: self lpstrFileOffset put: (self class Ccalloc: self MAXPATH);
    unsignedLongAt: self nMaxFileOffset put: self MAXPATH
```

Instance Creation and Memory Allocation/Reclamation

Here are the full instance creation protocol and lower-level memory allocation and reclamation facilities in Alien class.

instance creation (storage allocation)

new <^Alien>
: answer a direct instance whose data resides in the instance on the Smalltalk heap and is of the size defined by a subclass's dataSize. Raises an error if dataSize is not defined. Storage is zeroed and the instance initialized.

newC <^Alien>
: answer an indirect instance whose data resides on the C heap and is of the size defined by a subclass's dataSize. Raises an error if dataSize is not defined. Storage is zeroed and the instance initialized.

newGC
: as for newC but arrange for the finalization mechanism to free the C storage some time after the instance is garbage collected.

new:, **newC:**, **newGC:** dataSize <Integer> <^Alien>
: as per new, newC, newGC respectively but answer an Alien of the byte size requested.

rawNewC:, dataSize <Integer> <^Alien>
as per new, newC: but storage is allocated via `malloc` and hence not initialized to zero.

instance creation (pointer coercing)

atAddress: address <Integer> <^Alien>
answer an indirect instance to some preallocated data residing at address with the size bounds given by the class's dataSize method. Raises an error if dataSize is not defined. Neither the storage nor the instance is initialized.

atAddress: address <Integer> **dataSize:** dataSize <Integer> <^Alien>
answer an indirect instance to some preallocated data residing at address, with the size bounds requested. Neither the storage nor the instance is initialized.

forPointer: address <Integer> <^Alien>
answer a pointer instance to some preallocated data residing at address with *no* storage bounds. Neither the storage nor the instance is initialized.

memory management

Cmalloc: byteSize <Integer> <^Integer>
answer the address of storage allocated via C's `malloc` that is hence uninitialized.

Ccalloc: byteSize <Integer> <^Integer>
answer the address of storage allocated via C's `calloc` that is hence initialized with zeros.

Call-outs

Call-outs are performed using primitives that interpret the receiver Alien as a pointer to some function. Functions can be looked-up and answered as Aliens by Alien class>> lookup:inLibrary:. The call-out primitives take as arguments a result Alien into which result data is copied, or nil if no result is required, and a sequence of Aliens or integers as arguments. All call-out primitives answer the result parameter or fail if given an invalid parameter.

Here's an example from Windows using the OPENFILENAME struct defined above:

Alien class examples
exampleGetOpenFileName
```
"Call the Win32 GetOpenFileNameA function."
| openstruct |
openstruct := OPENFILENAME newC.
(self lookup: 'GetOpenFileNameA' inLibrary: 'comdlg32.dll')
      primFFICallResult: nil
      with: openstruct pointer.
^openstruct fileName
```

OPENFILENAME accessing
fileName
```
^(Alien forPointer: (self unsignedLongAt: self lpstrFileOffset)) strcpyUTF8
```

here primFFICallResult:with: calls the function pointer in the Alien answered by Alien lookup: 'GetOpenFileNameA' inLibrary: 'comdlg32.dll' with one argument, openstruct pointer, an Alien

wrapping he address of openstruct which was allocated on the C heap via OPENFILENAME newC. `GetOpenFileNameA` actually answers a Win32 `BOOL` (C `int`) but the result is ignored by using nil as the result holder.

There are versions of primCallResultWith:... that take up to 12 arguments and you can define more if you want to. There is also a version that takes its arguments from an Array allowing any number. Here is an example call of C's `printf` that answers 24 (since `printf` et al answer the number of characters printed, or a negative value on error)

```
| r s |
s := ('Hello World %d %x !!', (String with: Character lf)) asAlien.
(Alien lookup: 'printf' inLibrary: self libcName)
    primFFICallResult: (r := Alien new: 4)
    withArguments: {s pointer. 123. 48879}.
s free.
^r signedLongAt: 1
```

which uses convenience protocol on String to allocate the format string on the C heap (printf takes a pointer to a format string) and hence necessitates we free the memory to avoid a leak:

*String *Newspeak-FFI-coercing*
asAlien
```
^(Alien newC: self byteSize + 1)
      replaceFrom: 1 to: self byteSize with: self startingAt: 1
```

Parameter Passing Rules

Both direct and indirect Aliens are passed by passing their data (rounded up to a multiple of 4 bytes as mandated by the ABI). Hence to pass a structure or array by value simply pass an (in)direct Alien of the relevant byte size. For convenience one can also use integers, which are passed as 4-bytes, sign-extended if negative. Pointer Aliens pass their address in 4 bytes.

Callbacks

Call-backs are handled by associating machine code "thunks" that serve as functions for C to call with Smalltalk blocks that are activated via plumbing in response to the thunks being called. To create a callback one supplies a block and an Alien defining the callback's C signature. A call-back's block is invoked with an indirect Alien for the signature wrapping the C stack pointer at time of call, and with a special result Alien that handles passing results back from the block to C. Here's an example that calls C's `qsort` quicksort function:

Alien class examples
exampleCqsort
```
"Call the libc qsort function (which requires a callback)."
| cb rand nElements sizeofDouble values orig sort |
"create 100 doubles in random order"
rand := Random new.
values := Alien newC: (nElements := 100) * (sizeofDouble := 8).
1 to: values dataSize by: sizeofDouble do:
    [:il values doubleAt: i put: rand next].
```

```
    orig := (1 to: values dataSize by: sizeofDouble) collect: [:il values doubleAt: i].
    "create the callback"
    cb := Callback
                block:
                    [:args :resultI
                     result returnInteger: ((args first doubleAt: 1) - (args second doubleAt: 1)) sign]
                argsClass: QsortCompare.
    "sort them"
    (self lookup: 'qsort' inLibrary: self libcName)
        primFFICallResult: nil
        with: values pointer
        with: nElements
        with: sizeofDouble
        with: cb thunk.
    sort := (1 to: values dataSize by: sizeofDouble) collect: [:il values doubleAt: i].
    values free.
    ^orig -> sort
```

The callback uses a QsortCompare Alien to define the callback's C signature. C's qsort expects the callback to have the signature `int (*)(const void *, const void *)`.

```
Alien variableByteSubclass: #QsortCompare

QsortCompare class instance creation
dataSize
    ^8

QsortCompare accessing
first
    ^Alien forPointer: (self unsignedLongAt: 1)

second
    ^Alien forPointer: (self unsignedLongAt: 5)
```

So the arguments, both pointer Aliens, are available as first and second. The callback block accesses two doubles via first and second, and answers -1, 0 or 1 via Smalltalk's sign method which meets the expectations of `qsort`. The value is answered as a 32-bit C integer by using the result's returnInteger: method. `qsort` requires a function pointer so the thunk is passed via cb thunk, which answers an Alien for the thunk's address. N.B. Allocation of callback thunks is effectively via newGC so one should hold a reference to the callback object at least as long as C code has access to the thunk. In the example above the callback is assigned to the cb temp and so is referenced until the entire method exits. If the invocation was written as

```
    (self lookup: 'qsort' inLibrary: self libcName)
        primFFICallResult: nil
        with: values pointer
        with: nElements
        with: sizeofDouble
        with: (Callback
```

```
block:
   [:args :resultI
    result returnInteger: ((args first doubleAt: 1) - (args second doubleAt: 1)) sign]
argsClass: QsortCompare) thunk
```

then here is a possibility that the callback object could be reclaimed by the time the callback machinery attempts to evaluate the callback's block.

Floating-point Return Values

The call-outs and callbacks seen so far involve only integral (integer and pointer) arguments and results. On our current platform IA32 ABIs floating-point arguments are passed on the normal stack and so don't require special processing. But since floating-point results are returned in an FPU register they do require special handling. Because Aliens contain only size information, not type information they cannot indicate this different behaviour to the callout primitive(s). To call a function returning a `double` result use primFFICallDoubleResult: et al (not fully implemented; simply copy primFFICallDoubleResult: methods and change their primitive from 'primCallOutIntegralReturn' to 'primCallOutDoubleReturn'. To return a `double` from a callback use returnDouble:. The VM code has support for single-precision if needed, use 'primCallOutFloatReturn'.

Using Pointers

As we've seen in some of the examples above one can take the address of an Alien allocated with newC et al.

<Alien> pointer <^Alien>
    answer the address of the receiver's data if it is indirect.

Whereas direct and indirect Aliens pass their data when passed as parameters, pointer Aliens pass their pointer. However, pointer Aliens provide access to the data pointed to when using the accessing primitives. Hence if one needs to manipulate data that C expects a pointer to one can simply hold onto the pointer after e.g. creating the data via newC, or having it answered from C code.

When used as a result a pointer Alien will have its address assigned with the four byte result from the called function, hence it will effectively be assigned the pointer answered by the called function.

Fetching the Results of C Function Calls

When (in)direct Aliens are used as result parameters their data is assigned, the number of bytes being the minimum of their dataSize and the function's result type. Both primFFICallResult:... and primFFICallDoubleResult:... answer up to 8 bytes of data. IA32 ABIs answer a 32-bit integral result in `%eax` and a 64-bit integer result in `%eax` (low 32-bits) and `%edx` (high 32-bits), and answer 32-bit or 64-bit floating-point in `%f0`.

The rules for structure results vary slightly by platform. Most functions returning structures expect a "hidden" first parameter holding the result struct's address. Because the FFI provides no abstraction one must pass this parameter explicitly. For example, to call some function answering a 16-byte NSRect above one could write

```
functionAlien primFFICallResult: nil with: (indirectStructAlien := Alien newC: 16) pointer
```

If the function takes parameters these simply follow the pointer argument.

Some IA32 ABI variants return small structures in registers. On linux all struct results are returned through the hidden first argument mechanism. On Mac OS X any structure of 8 bytes or less is answered in `%eax/%edx`. On Windows any structure whose size is a power of two and is of 8 bytes or less is answered in `%eax/%edx`. Hence on Mac OS X or Windows to receive an 8 byte (e.g. `NSSize`) struct from some function one could write

```
functionAlien primFFICallResult: (directStructAlien := Alien new: 8)
```

To summarise:

`void`	primFFICallResult: nil
[`unsigned`] `char`	primFFICallResult: (Alien new: 1)
[`unsigned`] `short`	primFFICallResult: (Alien new: 2)
[`unsigned`] `int`	primFFICallResult: (Alien new: 4)
[`unsigned`] `long`	primFFICallResult: (Alien new: 4)
`void *`	primFFICallResult: (Alien new: 4 *or e.g.* Alien forPointer: 0)
[`unsigned`] `long long`	primFFICallResult: (Alien new: 8)
`struct {} foo`	primFFICallResult: nil with: (Alien newC: n) pointer
`struct {} small`	primFFICallResult: (Alien new: n) *Mac OS X and Windows only!*
`double`	primFFICallDoubleResult: (Alien new: 8)

Character Strings and Byte Data

C and Smalltalk have differing string models. C uses null-terminated strings whereas Smalltalk uses objects that store their size explicitly in a hidden field. Strings may be in various character encodings. Squeak's ByteString is an 8-byte character string in ISO9660 encoding. Squeak's WideString is a 32-bit character string mapping to Squeak's Unicode Character set. The FFI provides limited support for exchanging string data and converting between encodings.

Convenience protocol on String answers a null-terminated copy of the receiver as an Alien. protocol on Alien provides access to string data embedded in Aliens.

<ByteString> **asAlien** <^Alien>

answer an indirect Alien with the null-terminated byte contents of the receiver, i.e. a character array in ISO9660 encoding, *not* a pointer there to. Memory is allocated using newC: so the client must free to avoid a leak.

<WideString> **asAlien** <^Alien>

answer an indirect Alien with the null-terminated byte contents of the receiver, i.e. a character array with 4-bytes per character in Unicode encoding, *not* a pointer there to. Memory is allocated using newC: so the client must free to avoid a leak. This method may be useful on linux and Mac OS X where `sizeof(wchar_t)` is 4, but *not* on `Windows`, where `sizeof(wchar_t)` is 2.

<Alien> **strcpy** <^ByteString>

Answer a copy of the null-terminated string starting at the first byte of the receiver (the first byte pointed to, if a pointer Alien) as a ByteString. Assumes the string is in ISO9660 encoding. Copies only up to the size of the (in)direct Alien. Memory is allocated using newC: so the client must free to avoid a leak.

<Alien> **strcpyFrom:** index <Integer> <^ByteString>

As per strcpy but start copying from the index.

<Alien> **strcpyUTF8** <^String>

Answer a copy of the null-terminated string starting at the first byte of the receiver (the first byte pointed to, if a pointer Alien) as either a ByteString or a WideString depending on the string's contents. Assumes the string is in UTF8 encoding. Copies only up to the size of the (in)direct Alien. Memory is allocated using newC: so the client must free to avoid a leak.

<Alien> **strcpyUTF8From:** index <Integer> <^String>

As per strcpyUTF8 but start copying from the index.

<Alien> **strlenStartingAt:** index <Integer> <^Integer>

Answer the number of bytes up to a null byte or the end of the Alien starting at the index.

<Alien> **strlenThroughPointerAt:** index <Integer> <^Integer>

Answer the number of bytes up to a null byte starting at the 4 byte pointer at index in the receiver.

Memory Management

Both indirect and pointer Aliens can be freed. A side-effect of the freeing primitive is to set the Alien's address and size to zero to avoid double-freeing dangers. Hence it is safe (since Squeak primitives are atomic) to free an Alien obtained with newGC[:]. Use free rather than primFree since some Aliens (including FFICallbackThunk in the callback machinery) may need to take additional actions before actually freeing their memory. Alien class provides a naked free primitive to complement the naked Cmalloc: and Ccalloc:. For example here's the free method that complements OPENFILENAME's initialize method above

OPENFILENAME methods for memory management

free

```
self class primFree: (self unsignedLongAt: self lpstrFileOffset).
super free
```

Coda

ASHE: You still don't know what you're dealing with do you? Perfect organism. Its structural perfection is matched only by its hostility.
RIPLEY: You admire it.
ASHE: I admire its purity, its sense of survival; unclouded by conscience, remorse, or delusions of morality.
RIPLEY: I've heard enough and I'm asking you to pull the plug.

ASHE: One more word..... I can't lie to you about your chances, but... you have my sympathies.
Alien, dir. Ridley Scott, 1979