

**FFI**

Igor Stasenko

RMoD Team  
INRIA Lille Nord-Europe

INRIA-CEA-EDF School Deep into Smalltalk  
March 2011

# Outline

---



what is FFI?  
how it is done?  
when do you need it?  
why not writing plugin instead?  
security?  
existing implementations

## What is FFI?

---



FFI stands for **F**oreign **F**unction **I**nterface  
helps connecting our wonderful universe with outer  
world (which in most cases are C ;)

## How it is done?



- Dynamic libraries (.dll, .so, .dylib) have to follow conventions (known as ABI), so applications could use them.
- Therefore, by knowing these conventions and honoring them, applications can use dynamic libraries.
- FFI is a layer that provides that.

# When do you need it?



- When there is no other way to access certain functionality
- Which is often happens because VM can't do everything you want :)

## Why not writing plugin instead?



- In a longer perspective, well written plugin could be better.
- But plugins are harder to develop
- You have to build own version of VM
- For prototyping and extreme style development FFI is winner for sure.
- I believe that statically generated code is counting its last days on Earth's computers :)

# Security concerns?



- Lets face it: Smalltalk in general is a wide open architecture and therefore there are little or no security.
- Does using FFI makes malicious code more dangerous than malicious smalltalk one?
- Let us grow up: Jailing kid in a room with no way to get out doesn't guarantees that kid won't become a criminal later.
- Developers are not kids. They can decide for themselves.

# Existing implementations



- FFIPlugin ([source.squeak.org/FFI](http://source.squeak.org/FFI))
- Alien ([squeaksource.com/Alien](http://squeaksource.com/Alien))
- NativeBoost ([squeaksource.com/NativeBoost](http://squeaksource.com/NativeBoost))

# FFIPlugin



```
apiGetEnvironmentVariable: lpName with: lpBuffer with: nSize
```

```
<apicall: ulong 'GetEnvironmentVariableA' (char* byte* ulong)  
module: 'kernel32.dll'>
```

```
^self externalCallFailed
```

Parser parses a definition to ExternalLibraryFunction instance and placing it to method's literals.

A special primitive #120 (primitiveExternalCall) set for a method, which knows how to handle the ExternalLibraryFunction

## Callout spec



<**apicall:** ulong 'GetEnvironmentVariable'  
(char\* byte\* ulong) **module:** 'kernel32.dll'>

Three parts

- **apicall:** or **cdecl:** denotes call convention to use.

- <**returnType**> <**name**> (<**arg types**>...) function prototype.

a function name could be string or number and used for looking up in an external library.

- **module:** a named of external library to search for given function

# Call conventions



apicall AKA stdcall AKA pascal call:

- pushing arguments from right to left
- callee cleans the stack
- mostly used by Windows OS system libraries

cdecl AKA C call convention (`__cdecl`).

- pushing arguments from right to left
- caller cleans the stack after return
- a default call convention used by C compilers

[http://en.wikipedia.org/wiki/X86\\_calling\\_conventions#cdecl](http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl)

# Basic types



FFI types matching the primitive types of C  
see `ExternalType>>initializeAtomicTypes`

"name	atomic id	byte size"
('void'	0	0)
('bool'	1	1)
('byte'	2	1)
('sbyte'	3	1)
('ushort'	4	2)
('short'	5	2)
('ulong'	6	4)
('long'	7	4)
('ulonglong'	8	8)
('longlong'	9	8)
('char'	10	1)
('schar'	11	1)
('float'	12	4)
('double'	13	8)

# Pointer types\*



- A `<name>*` denotes a pointer type.
- You can use pass instances of `variableByte` or `variableWord` classes (like `ByteArray`, `WordArray`, `FloatArray`, `String-s`) to pass them as a pointer argument.
- The pointer to first indexable field is pushed on stack.
- A pointer value returned for return type.

# Struct types



```
struct abc {  
  int a;  
  char b;  
  float c;  
};
```

```
ExternalStructure subclass: #ABCStruct  
instanceVariableNames: ""  
classVariableNames: ""  
poolDictionaries: ""  
category: 'Example'
```

```
ABCStruct class>>fields  
"define fields"  
^#(  
  (a 'int')  
  (b 'char')  
  (c 'float'))
```

# Making calls programmatically



```
fn := ExternalLibraryFunction new.
```

```
... fill argument types, module etc...
```

```
fn invokeWithArguments: { un. deux. trois. }
```

# Example



```
ExternalObject subclass: #MacOSShell
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'FFI-MacOS-Examples'
```

```
getenv: aString
<apical: char* 'getenv' (char*) module: 'libSystem.dylib'>
self externalCallFailed
```

# I want More!

---



- Callbacks
- Threaded calls