Investigation agenda

a. The general design philosophy behind FFI
b. Showcase of languages relying on FFI
c. What FFI for Node native modules would look like
    i. Supported types.
        1. Basic types
        2. Array / buffer
        3. Function pointer (opaque)
        4. Struct? (may not be needed for first cut, but needed for calling native system libraries)
        5. Wrapping / passing native refs to JS, or JS object refs to native
        6. Exceptions / error handling. Leave for version 2.
        7. String encodings. Will need to support different encogings. TBD how to specify in IDL? C signatures might need to be augmented.
    ii. Native-to-JavaScript calls. Needs function pointer signature description
    iii. Async callbacks / completion. DispatchCallback(void * state)
    iv. GC considerations
d. Possible implementations
    i. Template-based approach
    ii. Engine generated bindings
e. Achieving ABI stability
f. Showcase:
    i. One native module. Bufferutil, utf8-validate, or simple example (int add(int, int)
    ii. Http parser
        1. What's needed?
        2. Actual implementation with benchmarks run
    iii. Fs.stat
g. Action plan to make progress

Design philosophy behind FFI

In general what we have in v8 today is a form of FFI.
The design philosophy that we are proposing is:

a) Use a declarative approach and hide the marshalling glue from the end user
b) To expose to the native world little or no knowledge of the JS type system.

c) Ideally can be applied to existing native exports without modification / intermediate layers, in most common cases

Keep the interface as simple and minimalistic as possible: support C function signatures.

What about interop with other languages? Can be used for other languages but for today we focus on JS -> C.

Declarative as opposed to functional. Marshalling glue is more automatic.

Describe the C signature, not JS/v8/Chakra. No VM concepts.

Simplest type system possible, and see how far where we can go with that. Possible extensions for more advanced scenarios.

Achieving ABI compatibility / dynamic linking

JIT generated bindings based on signature declarations is the only viable way to achieve dynamic linking.

“JIT generated” could be done by VM or Node, though the VM is the best bet.

Either JIT generated. Or leverage a VM neutral API. Would have to be JIT generated to make sense as *an alternative* to VM neutral API.

How / where to describe the signatures

Can be done in multiple ways. Different facilities can be provided as syntactic sugar on top of a low-level IDL language.

1. In JS. Copy paste the C prototype into a JS string. Might fall short on certain scenarios / need augmentation (eg string encoding, byref lifecycle management) / might just be syntactic sugar on top of a richer IDL language
2. In C, with macros or templates
3. The swift approach. Just specify the function name, and it gets imported from headers https://swiftlang.ng.bluemix.net/#/repl

We don’t know what the IDL will be. We’ll look to leverage existing IDL as much possible.

Languages using FFI

- Lisp
- Haskell
- Python
- Java (JNI/JNA)
- .NET (p/invoke)
- LuaJIT

Templated based approach vs JIT generated bindings

JIT generated bindings seems like the only feasible way to achieve dynamic linking.
The value of the template based approach is how simple it is to prototype with it and get a functioning API (even though no dynamic linking). It could be used to auto-generate marshalling glue in cases where ABI compatibility is not needed, e.g. for Node internal bindings (http, etc).

It might be usable to emit IDL for JIT generated bindings, in which case it could be used as the API exposed to native modules. If not, we’ll have to adopt another way to declare signatures.

Existing IDL languages

- Blink's WebIDL https://www.chromium.org/blink/webidl
- SWIG http://www.swig.org/
- ….
- Plain C signatures copy & pasted into JS, or an external file

Engine agnostic could provide a path to interfacing with other languages in the future

Code snippets

```
int32_t AddMethod(int32_t x, int32_t y) {
    return x + y;
```

```
}

void Initialize(Handle<Object> exports) {
  NODE_SET_METHOD(exports, "add", BindNativeFunction<AddMethod>);
}


==============

#include “node_interop.h”

int32_t AddMethod(int32_t x, int32_t y) {
    return x + y;
}


NODE_MODULE(“mymodule.def”)

=== mymodule.def ===

NODE_MODULE_EXPORT("add", AddMethod)


==========

#include “node_interop.h”



int32_t AddMethod(int32_t x, int32_t y) {
    return x + y;
}

NODE_MODULE_DECL_BEGIN
NODE_MODULE_EXPORT("add", AddMethod, “int (int, int));
NODE_MODULE_DECL_END
```

Supported types.

Primitive types

JS number cannot represent int64_t with full precision.
If return value int64_t do we return it as number of as a boxed type?

Will need boxed type. For pointers, or other types that might not be fully representable in JS.
The user/IDL must specify when to use boxing. There can be defaults based on types.

Array / buffer

In v1, only support TypedArray.
Native function gets a pointer to buffer / TypedArray memory + length. No copying, internal buffer memory can be accessed.

Will need to support

- Buffer / TypedArray
- String

Will need to support both cases in which the memory is deallocated by native or JS. It's a part of the metadata.

Strings

Support most common encodings, in both directions:

- UTF8
- UTF16
- ?

Function pointer

Described in terms of C function pointer type.

Async call backs
We might need to give the native side a way to schedule callbacks on the main thread. Not very common case. Minimum support needed: DispatchCallback (provided by Node, not FFI).

Struct

May not be needed for first cut, but needed for calling some system libraries.

Need to support by value and by ref.

VM can create objects that expose the fields as properties. Implementation can be optimized to access the underlying memory.

Support nested structs.

In, Out

Must be included in metadata.
Could be implemented by a buffer. Out parameter is always a pointer (buffer or opaque type on JS side).

Unions

Hopefully not needed in v1.

Wrapping / passing native refs to JS, or JS object refs to native

Storing native refs in JS
Boxed type.
Ownership transfer can be specified as a flag on any pointer argument.

If you have a correspondence between JS and native, JS owns it.

Storing JS refs in native code
Not supported. Callbacks to JS with state can be handled by closures and function pointers.

Holding JS refs across callbacks will require explicit release. Complex case, we'll probably not support it initially.

Exceptions / error handling.

Leave for version 2.

Performance

All members of investigation team are highly confident that FFI with JIT generated bindings can be as fast or faster than the current v8 marshalling approach. As such it should be good enough for most built-in module bindings.

Stability

Given the design approach, higher level of insulation from JS/VM is expected, hence this is expected to have a higher chance of providing a **stable** interface for native modules.

Timeline

Disclaimer: highly dependant on communication and commitment between teams. FFI in Node would require buy-in and implementation from all the VMs.

On a constrained feature set:

- Spec draft / EP: 2 months
- First usable implementation: 6 months

Next steps: @ofrobots to discuss with v8 team and find out if they are willing to commit resources to the investigation / implementation. If yes, then Chakra will also commit resources.