FFI for Node.js Native Modules

- By @curtisman, @ofrobots, @orangemocha, @piscisaureus

Design philosophy

- The current v8 API for native modules is a form of FFI. How is this FFI proposal different?
  1. Declarative approach. Parameter marshalling is automated behind the scenes, rather than explicitly coded by user
  2. Expose to the native world little or no knowledge of the JS type system and the VM
  3. Can be used to invoke existing native libraries from JS without intermediate layers, in most common cases

Design choices

- Focused on invoking C native functions (though extensions to other languages are imaginable)
- Minimalistic functionality, to cover most common cases. Future extensions possible for more advanced use cases

Achieving ABI stability

- The hidden marshalling glue can be expressed either:
  a) In terms of a VM-neutral API (static)
  b) In terms of VM-specific bindings generated at runtime (dynamic)
- Conclusion: FFI as a viable alternative to a VM-neutral API would have to rely on VM-specific JIT-generated bindings
- We haven't really considered the static route fully in the meeting because we wanted to explore alternatives to the VM-neutral API. But instead of waiting for the VM-neutral API first, it can be explored concurrently, so that it can inform the design VM-neutral marshalling API and make sure the API is rich and robust enough to support generated marshaling glue from a FFI system.

Describing call signatures

- Needs standardized low-level IDL to be fed to the VM
- IDL could be specified in multiple places (e.g. by static or dynamic export from C, directly in JS, with a separate metadata file)
- Simple C prototypes are not enough. Signatures need to be augmented to specify important semantics: byref vs byvalue, in/out, boxed vs primitive, lifecycle management/ownership transfer, string encodings, etc.
- Actual IDL format is TBD. We should try to leverage existing, mature formats as much as possible. Examples:
  - Blink's WebIDL https://www.chromium.org/blink/webidl
  - http://www.swig.org/
- User-friendly syntaxes can be provided as syntactic sugar on top of the low level format, possibly by third-party tools/modules
- Super-intuitive approaches are not enough for 100% of use cases, though they might work for simple cases, using defaults:
  - Copy & paste C prototype into a JS string
  - Specify import name, and its signature is automatically found from system headers (https://swiftlang.ng.bluemix.net/#/repl)

Potential benefits of FFI approach

- Performance
  - JIT generated bindings can be as fast as the current v8 marshalling approach in most scenarios, and may be faster on some scenarios.
    - Hence, faster than a VM-neutral API built on top of VM-specific APIs
    - Hence, usable to replace current native binding for built-in modules (eg tcp, http_parser). Benefit: simplify the job to support multiple VMs in Node
- Ease of use
  - Provides native module developer an easier way to create and maintain bindings, instead of writing them in C/C++.
- Stability
  - Given the higher level of insulation from JS/VM internals, the FFI approach is expected to have a higher chance of providing a **stable** interface for native modules.

Concerns / Drawbacks

- This approach (without a VM neutral API) requires every VM to have the capability to generated dynamic bindings, thus imposing a barrier to entry for new VMs. Some VM might not even have a JIT which makes providing this capability even harder.
- Even though it might be possible to cover the most common cases in the long term, the cost is pretty high to implement and maintain. The first implementation may only cover a small portion of native module scenarios as compared to the VM-neutral marshalling, and we may still need to implement VM-neutral marshalling API for the scenarios that we don’t cover (and for other VM as mentioned above).

Next steps and Timeline

- Next steps: @ofrobots to discuss with v8 team and find out if they are willing to commit resources to the investigation / implementation. If so, then Chakra will also commit resources.
- Rough estimate from TBD start time (highly dependent on communication and commitment between teams)
  - Spec draft / EP: 2 months
  - First usable implementation: 6 months