To be or not to be in core

This document explains things to consider when deciding whether a component should be in core or not.

A component may be included in core as a dependency, a module, or integrated into the code base. The same arguments for including/not including in core generally apply in all of these cases.

Strong arguments for including a component in core

1. The component provides functionality that is standardized (such as a Web API) and overlaps with existing functionality.
2. The component can only be implemented in core.
3. The component can only be implemented in a performant way in core.
4. Developer experience is significantly improved if the component is in core.
5. The component provides functionality that can be expected to solve at least one common use case Node.js users face.
6. The component requires native bindings. Inclusion in core enables utility across operating systems and architectures without requiring users to have a native compilation toolchain.
7. Part or all of the component will also be re-used or duplicated in core.

Strong arguments against including a component in core

1. None of the arguments listed in the previous section apply.
2. The component has a license that prohibits Node.js from including it in core without also changing its own license.
3. There is already similar functionality in core and adding the component will provide a second API to do the same thing.
4. A component (or/and the standard it is based on) is deprecated and there is a non-deprecated alternative.
5. The component is evolving quickly and inclusion in core will require frequent API changes.

Benefits and challenges

When it is unclear whether a component should be included in core, it might be helpful to consider these additional factors.

Benefits

1. The component will receive more frequent testing with Node.js CI and CITGM.
2. The component will be integrated into the LTS workflow.
3. Documentation will be integrated with core.
4. There is no dependency on npm.

Challenges

1. Inclusion in core, rather than as an ecosystem module, is likely to reduce code merging velocity. The Node.js process for code review and merging is more time-consuming than that of most separate modules.
2. By being bound to the Node.js release cycle, it is harder and slower to publish patches.
3. Less flexibility for users. They can't update the component when they choose without also updating Node.js.