Using the internal/errors.js module

What is internal/errors.js

The `require('internal/errors')` module is an internal-only module that can be used to produce `Error`, `TypeError` and `RangeError` instances that use a static, permanent error code and an optionally parameterized message.

The intent of the module is to allow errors provided by Node.js to be assigned a permanent identifier. Without a permanent identifier, userland code may need to inspect error messages to distinguish one error from another. An unfortunate result of that practice is that changes to error messages result in broken code in the ecosystem. For that reason, Node.js has considered error message changes to be breaking changes. By providing a permanent identifier for a specific error, we reduce the need for userland code to inspect error messages.

Switching an existing error to use the `internal/errors` module must be considered a `semver-major` change.

Using internal/errors.js

The `internal/errors` module exposes all custom errors as subclasses of the builtin errors. After being added, an error can be found in the `codes` object.

For instance, an existing `Error` such as:

```
const err = new TypeError(`Expected string received ${type}`);
```

Can be replaced by first adding a new error key into the `internal/errors.js` file:

```
E('FOO', 'Expected string received %s', TypeError);
```

Then replacing the existing `new TypeError` in the code:

```
const { FOO } = require('internal/errors').codes;
// ...
const err = new FOO(type);
```

Adding new errors

New static error codes are added by modifying the `internal/errors.js` file and appending the new error codes to the end using the utility `E()` method.

```
E('EXAMPLE_KEY1', 'This is the error value', TypeError);
E('EXAMPLE_KEY2', (a, b) => `${a} ${b}`, RangeError);
```

The first argument passed to `E()` is the static identifier. The second argument is either a String with optional `util.format()` style replacement tags (e.g. `%s`, `%d`), or a function returning a String. The optional additional arguments passed to the `errors.message()` function (which is used by the `errors.Error`,

`errors.TypeError` and `errors.RangeError` classes), will be used to format the error message. The third argument is the base class that the new error will extend.

It is possible to create multiple derived classes by providing additional arguments. The other ones will be exposed as properties of the main class:

```
E('EXAMPLE_KEY', 'Error message', TypeError, RangeError);

// In another module
const { EXAMPLE_KEY } = require('internal/errors').codes;
// TypeError
throw new EXAMPLE_KEY();
// RangeError
throw new EXAMPLE_KEY.RangeError();
```

Documenting new errors

Whenever a new static error code is added and used, corresponding documentation for the error code should be added to the `doc/api/errors.md` file. This will give users a place to go to easily look up the meaning of individual error codes.

Testing new errors

When adding a new error, corresponding test(s) for the error message formatting may also be required. If the message for the error is a constant string then no test is required for the error message formatting as we can trust the error helper implementation. An example of this kind of error would be:

```
E('ERR_SOCKET_ALREADY_BOUND', 'Socket is already bound');
```

If the error message is not a constant string then tests to validate the formatting of the message based on the parameters used when creating the error should be added to `test/parallel/test-internal-errors.js`. These tests should validate all of the different ways parameters can be used to generate the final message string. A simple example is:

```
// Test ERR_TLS_CERT_ALTNAME_INVALID
assert.strictEqual(
  errors.message('ERR_TLS_CERT_ALTNAME_INVALID', ['altname']),
  'Hostname/IP does not match certificate\'s altnames: altname');
```

In addition, there should also be tests which validate the use of the error based on where it is used in the codebase. If the error message is static, these tests should only validate that the expected code is received and NOT validate the message. This will reduce the amount of test change required when the message for an error changes.

```
assert.throws(() => {
  socket.bind();
}, common.expectsError({
  code: 'ERR_SOCKET_ALREADY_BOUND',
  type: Error
}));
```

Avoid changing the format of the message after the error has been created. If it does make sense to do this for some reason, then additional tests validating the formatting of the error message for those cases will likely be required.

API

Object: errors.codes

Exposes all internal error classes to be used by Node.js APIs.

Method: errors.message(key, args)

- `key` {string} The static error identifier
- `args` {Array} Zero or more optional arguments passed as an Array
- Returns: {string}

Returns the formatted error message string for the given `key`.