Developer Notes

Table of Contents

Coding Style (General)

Various coding styles have been used during the history of the codebase, and the result is not very consistent. However, we're now trying to converge to a single style, which is specified below. When writing patches, favor the new style over attempting to mimic the surrounding style, except for move-only commits.

Do not submit patches solely to modify the style of existing code.

Coding Style (C++)

- **Indentation and whitespace rules** as specified in src/.clang-format. You can use the provided clang-format-diff script tool to clean up patches automatically before submission.
  - Braces on new lines for classes, functions, methods.
  - Braces on the same line for everything else.
  - 4 space indentation (no tabs) for every block except namespaces.
  - No indentation for `public` / `protected` / `private` or for `namespace`.
  - No extra spaces inside parenthesis; don't do `( this )`.
  - No space after function names; one space after `if`, `for` and `while`.
  - If an `if` only has a single-statement `then`-clause, it can appear on the same line as the `if`, without braces. In every other case, braces are required, and the `then` and `else` clauses must appear correctly indented on a new line.
  - There's no hard limit on line width, but prefer to keep lines to <100 characters if doing so does not decrease readability. Break up long function declarations over multiple lines using the Clang Format AlignAfterOpenBracket style option.
- **Symbol naming conventions**. These are preferred in new code, but are not required when doing so would need changes to significant pieces of existing code.
  - Variable (including function arguments) and namespace names are all lowercase and may use `_` to separate words (snake_case).
    - Class member variables have a `m_` prefix.
    - Global variables have a `g_` prefix.
  - Constant names are all uppercase, and use `_` to separate words.
  - Enumerator constants may be `snake_case`, `PascalCase` or `ALL_CAPS`. This is a more tolerant policy than the C++ Core Guidelines, which recommend using `snake_case`. Please use what seems appropriate.
  - Class names, function names, and method names are UpperCamelCase (PascalCase). Do not prefix class names with `C`.
  - Test suite naming convention: The Boost test suite in file `src/test/foo_tests.cpp` should be named `foo_tests`. Test suite names must be unique.
- **Miscellaneous**
  - `++i` is preferred over `i++`.
  - `nullptr` is preferred over `NULL` or `(void*)0`.
  - `static_assert` is preferred over `assert` where possible. Generally; compile-time checking is preferred over run-time checking.

Block style example:

```
int g_count = 0;
```

```
namespace foo {
class Class
{
    std::string m_name;

public:
    bool Function(const std::string& s, int n)
    {
        // Comment summarising what this section of code does
        for (int i = 0; i < n; ++i) {
            int total_sum = 0;
            // When something fails, return early
            if (!Something()) return false;
            ...
            if (SomethingElse(i)) {
                total_sum += ComputeSomething(g_count);
            } else {
                DoSomething(m_name, total_sum);
            }
        }

        // Success return is usually at the end
        return true;
    }
}
} // namespace foo
```

Coding Style (C++ named arguments)

When passing named arguments, use a format that clang-tidy understands. The argument names can otherwise not be verified by clang-tidy.

For example:

```
void function(Addrman& addrman, bool clear);

int main()
{
    function(g_addrman, /*clear=*/false);
}
```

Running clang-tidy

To run clang-tidy on Ubuntu/Debian, install the dependencies:

```
apt install clang-tidy bear clang
```

Then, pass clang as compiler to configure, and use bear to produce the `compile_commands.json`:

```
./autogen.sh && ./configure CC=clang CXX=clang++
make clean && bear make -j $(nproc)      # For bear 2.x
make clean && bear -- make -j $(nproc)   # For bear 3.x
```

To run clang-tidy on all source files:

```
( cd ./src/ && run-clang-tidy  -j $(nproc) )
```

To run clang-tidy on the changed source lines:

```
git diff | ( cd ./src/ && clang-tidy-diff -p2 -j $(nproc) )
```

Coding Style (Python)

Refer to /test/functional/README.md#style-guidelines.

Coding Style (Doxygen-compatible comments)

Bitcoin Core uses Doxygen to generate its official documentation.

Use Doxygen-compatible comment blocks for functions, methods, and fields.

For example, to describe a function use:

```
/**
 * ... Description ...
 *
 * @param[in]  arg1 input description...
 * @param[in]  arg2 input description...
 * @param[out] arg3 output description...
 * @return Return cases...
 * @throws Error type and cases...
 * @pre  Pre-condition for function...
 * @post Post-condition for function...
 */
bool function(int arg1, const char *arg2, std::string& arg3)
```

A complete list of `@xxx` commands can be found at https://www.doxygen.nl/manual/commands.html. As Doxygen recognizes the comments by the delimiters (`/**` and `*/` in this case), you don't *need* to provide any commands for a comment to be valid; just a description text is fine.

To describe a class, use the same construct above the class definition:

```
/**
 * Alerts are for notifying old versions if they become too obsolete and
 * need to upgrade. The message is displayed in the status bar.
 * @see GetWarnings()
 */
class CAlert
```

To describe a member or variable use:

```
//! Description before the member
int var;
```

or

```
int var; //!< Description after the member
```

Also OK:

```
///
/// ... Description ...
///
bool function2(int arg1, const char *arg2)
```

Not picked up by Doxygen:

```
//
// ... Description ...
//
```

Also not picked up by Doxygen:

```
/*
 * ... Description ...
 */
```

A full list of comment syntaxes picked up by Doxygen can be found at https://www.doxygen.nl/manual/docblocks.html, but the above styles are favored.

Recommendations:

- Avoiding duplicating type and input/output information in function descriptions.
- Use backticks (`) to refer to `argument` names in function and parameter descriptions.
- Backticks aren't required when referring to functions Doxygen already knows about; it will build hyperlinks for these automatically. See https://www.doxygen.nl/manual/autolink.html for complete info.
- Avoid linking to external documentation; links can break.
- Javadoc and all valid Doxygen comments are stripped from Doxygen source code previews (`STRIP_CODE_COMMENTS = YES` in Doxyfile.in). If you want a comment to be preserved, it must instead use `//` or `/* */`.

Generating Documentation

The documentation can be generated with `make docs` and cleaned up with `make clean-docs`. The resulting files are located in `doc/doxygen/html`; open `index.html` in that directory to view the homepage.

Before running `make docs`, you'll need to install these dependencies:

Linux: `sudo apt install doxygen graphviz`

MacOS: `brew install doxygen graphviz`

Development tips and tricks

Compiling for debugging

Run configure with `--enable-debug` to add additional compiler flags that produce better debugging builds.

Show sources in debugging

If you have ccache enabled, absolute paths are stripped from debug information with the -fdebug-prefix-map and -fmacro-prefix-map options (if supported by the compiler). This might break source file detection in case you move binaries after compilation, debug from the directory other than the project root or use an IDE that only supports absolute paths for debugging.

There are a few possible fixes:

1. Configure source file mapping.

For `gdb` create or append to `.gdbinit` file:

```
set substitute-path ./src /path/to/project/root/src
```

For `lldb` create or append to `.lldbinit` file:

```
settings set target.source-map ./src /path/to/project/root/src
```

2. Add a symlink to the `./src` directory:

```
ln -s /path/to/project/root/src src
```

3. Use `debugedit` to modify debug information in the binary.

Compiling for gprof profiling

Run configure with the `--enable-gprof` option, then make.

`debug.log`

If the code is behaving strangely, take a look in the `debug.log` file in the data directory; error and debugging messages are written there.

The `-debug=...` command-line option controls debugging; running with just `-debug` or `-debug=1` will turn on all categories (and give you a very large `debug.log` file).

The Qt code routes `qDebug()` output to `debug.log` under category "qt": run with `-debug=qt` to see it.

Signet, testnet, and regtest modes

If you are testing multi-machine code that needs to operate across the internet, you can run with either the `-signet` or the `-testnet` config option to test with "play bitcoins" on a test network.

If you are testing something that can run on one machine, run with the `-regtest` option. In regression test mode, blocks can be created on demand; see test/functional/ for tests that run in `-regtest` mode.

DEBUG_LOCKORDER

Bitcoin Core is a multi-threaded application, and deadlocks or other multi-threading bugs can be very difficult to track down. The `--enable-debug` configure option adds `-DDEBUG_LOCKORDER` to the compiler flags. This inserts run-time checks to keep track of which locks are held and adds warnings to the `debug.log` file if inconsistencies are detected.

Assertions and Checks

The util file `src/util/check.h` offers helpers to protect against coding and internal logic bugs. They must never be used to validate user, network or any other input.

- `assert` or `Assert` should be used to document assumptions when any violation would mean that it is not safe to continue program execution. The code is always compiled with assertions enabled.
  - For example, a nullptr dereference or any other logic bug in validation code means the program code is faulty and must terminate immediately.
- `CHECK_NONFATAL` should be used for recoverable internal logic bugs. On failure, it will throw an exception, which can be caught to recover from the error.
  - For example, a nullptr dereference or any other logic bug in RPC code means that the RPC code is faulty and cannot be executed. However, the logic bug can be shown to the user and the program can continue to run.
- `Assume` should be used to document assumptions when program execution can safely continue even if the assumption is violated. In debug builds it behaves like `Assert` / `assert` to notify developers and testers about nonfatal errors. In production it doesn't warn or log anything, though the expression is always evaluated.
  - For example it can be assumed that a variable is only initialized once, but a failed assumption does not result in a fatal bug. A failed assumption may or may not result in a slightly degraded user experience, but it is safe to continue program execution.

Valgrind suppressions file

Valgrind is a programming tool for memory debugging, memory leak detection, and profiling. The repo contains a Valgrind suppressions file (`valgrind.supp`) which includes known Valgrind warnings in our dependencies that cannot be fixed in-tree. Example use:

```
$ valgrind --suppressions=contrib/valgrind.supp src/test/test_bitcoin
$ valgrind --suppressions=contrib/valgrind.supp --leak-check=full \
      --show-leak-kinds=all src/test/test_bitcoin --log_level=test_suite
$ valgrind -v --leak-check=full src/bitcoind -printtoconsole
$ ./test/functional/test_runner.py --valgrind
```

Compiling for test coverage

LCOV can be used to generate a test coverage report based upon `make check` execution. LCOV must be installed on your system (e.g. the `lcov` package on Debian/Ubuntu).

To enable LCOV report generation during test runs:

```
./configure --enable-lcov
make
make cov

# A coverage report will now be accessible at `./test_bitcoin.coverage/index.html`.
```

Performance profiling with perf

Profiling is a good way to get a precise idea of where time is being spent in code. One tool for doing profiling on Linux platforms is called `perf`, and has been integrated into the functional test framework. Perf can observe a running process and sample (at some frequency) where its execution is.

Perf installation is contingent on which kernel version you're running; see this thread for specific instructions.

Certain kernel parameters may need to be set for perf to be able to inspect the running process's stack.

```
$ sudo sysctl -w kernel.perf_event_paranoid=-1
$ sudo sysctl -w kernel.kptr_restrict=0
```

Make sure you understand the security trade-offs of setting these kernel parameters.

To profile a running bitcoind process for 60 seconds, you could use an invocation of `perf record` like this:

```
$ perf record \
    -g --call-graph dwarf --per-thread -F 140 \
    -p `pgrep bitcoind` -- sleep 60
```

You could then analyze the results by running:

```
perf report --stdio | c++filt | less
```

or using a graphical tool like Hotspot.

See the functional test documentation for how to invoke perf within tests.

Sanitizers

Bitcoin Core can be compiled with various "sanitizers" enabled, which add instrumentation for issues regarding things like memory safety, thread race conditions, or undefined behavior. This is controlled with the `--with-sanitizers` configure flag, which should be a comma separated list of sanitizers to enable. The sanitizer list should correspond to supported `-fsanitize=` options in your compiler. These sanitizers have runtime overhead, so they are most useful when testing changes or producing debugging builds.

Some examples:

```
# Enable both the address sanitizer and the undefined behavior sanitizer
./configure --with-sanitizers=address,undefined

# Enable the thread sanitizer
./configure --with-sanitizers=thread
```

If you are compiling with GCC you will typically need to install corresponding "san" libraries to actually compile with these flags, e.g. libasan for the address sanitizer, libtsan for the thread sanitizer, and libubsan for the undefined sanitizer. If you are missing required libraries, the configure script will fail with a linker error when testing the sanitizer flags.

The test suite should pass cleanly with the `thread` and `undefined` sanitizers, but there are a number of known problems when using the `address` sanitizer. The address sanitizer is known to fail in sha256_sse4::Transform which makes it unusable unless you also use `--disable-asm` when running configure. We would like to fix sanitizer issues, so please send pull requests if you can fix any errors found by the address sanitizer (or any other sanitizer).

Not all sanitizer options can be enabled at the same time, e.g. trying to build with `--with-sanitizers=address,thread` will fail in the configure script as these sanitizers are mutually incompatible. Refer to your compiler manual to learn more about these options and which sanitizers are supported by your compiler.

Additional resources:

- AddressSanitizer
- LeakSanitizer
- MemorySanitizer
- ThreadSanitizer
- UndefinedBehaviorSanitizer
- GCC Instrumentation Options
- Google Sanitizers Wiki
- Issue #12691: Enable -fsanitize flags in Travis

Locking/mutex usage notes

The code is multi-threaded and uses mutexes and the `LOCK` and `TRY_LOCK` macros to protect data structures.

Deadlocks due to inconsistent lock ordering (thread 1 locks `cs_main` and then `cs_wallet`, while thread 2 locks them in the opposite order: result, deadlock as each waits for the other to release its lock) are a problem. Compile with `-DDEBUG_LOCKORDER` (or use `--enable-debug`) to get lock order inconsistencies reported in the `debug.log` file.

Re-architecting the core code so there are better-defined interfaces between the various components is a goal, with any necessary locking done by the components (e.g. see the self-contained `FillableSigningProvider` class and its `cs_KeyStore` lock for example).

Threads

- Main thread (`bitcoind`): Started from `main()` in `bitcoind.cpp`. Responsible for starting up and shutting down the application.
- ThreadImport (`b-loadblk`): Loads blocks from `blk*.dat` files or `-loadblock=<file>` on startup.
- ThreadScriptCheck (`b-scriptch.x`): Parallel script validation threads for transactions in blocks.
- ThreadHTTP (`b-http`): Libevent thread to listen for RPC and REST connections.
- HTTP worker threads(`b-httpworker.x`): Threads to service RPC and REST requests.
- Indexer threads (`b-txindex` , etc): One thread per indexer.

- SchedulerThread (`b-scheduler`) : Does asynchronous background tasks like dumping wallet contents, dumping addrman and running asynchronous validationinterface callbacks.
- TorControlThread (`b-torcontrol`) : Libevent thread for tor connections.
- Net threads:
  - ThreadMessageHandler (`b-msghand`) : Application level message handling (sending and receiving). Almost all net_processing and validation logic runs on this thread.
  - ThreadDNSAddressSeed (`b-dnsseed`) : Loads addresses of peers from the DNS.
  - ThreadMapPort (`b-upnp`) : Universal plug-and-play startup/shutdown.
  - ThreadSocketHandler (`b-net`) : Sends/Receives data from peers on port 8333.
  - ThreadOpenAddedConnections (`b-addcon`) : Opens network connections to added nodes.
  - ThreadOpenConnections (`b-opencon`) : Initiates new connections to peers.

Ignoring IDE/editor files

In closed-source environments in which everyone uses the same IDE, it is common to add temporary files it produces to the project-wide `.gitignore` file.

However, in open source software such as Bitcoin Core, where everyone uses their own editors/IDE/tools, it is less common. Only you know what files your editor produces and this may change from version to version. The canonical way to do this is thus to create your local gitignore. Add this to `~/.gitconfig` :

```
[core]
        excludesfile = /home/.../.gitignore_global
```

(alternatively, type the command `git config --global core.excludesfile ~/.gitignore_global` on a terminal)

Then put your favourite tool's temporary filenames in that file, e.g.

```
# NetBeans
nbproject/
```

Another option is to create a per-repository excludes file `.git/info/exclude` . These are not committed but apply only to one repository.

If a set of tools is used by the build system or scripts the repository (for example, lcov) it is perfectly acceptable to add its files to `.gitignore` and commit them.

Development guidelines

A few non-style-related recommendations for developers, as well as points to pay attention to for reviewers of Bitcoin Core code.

General Bitcoin Core

- New features should be exposed on RPC first, then can be made available in the GUI.
    - *Rationale*: RPC allows for better automatic testing. The test suite for the GUI is very limited.
- Make sure pull requests pass CI before merging.
    - *Rationale*: Makes sure that they pass thorough testing, and that the tester will keep passing on the master branch. Otherwise, all new pull requests will start failing the tests, resulting in confusion and mayhem.
    - *Explanation*: If the test suite is to be updated for a change, this has to be done first.

Wallet

- Make sure that no crashes happen with run-time option `-disablewallet`.
- Include `db_cxx.h` (BerkeleyDB header) only when `ENABLE_WALLET` is set.
    - *Rationale*: Otherwise compilation of the disable-wallet build will fail in environments without BerkeleyDB.

General C++

For general C++ guidelines, you may refer to the C++ Core Guidelines.

Common misconceptions are clarified in those sections:

- Passing (non-)fundamental types in the C++ Core Guideline.
- Assertions should not have side-effects.
    - *Rationale*: Even though the source code is set to refuse to compile with assertions disabled, having side-effects in assertions is unexpected and makes the code harder to understand.
- If you use the `.h`, you must link the `.cpp`.
    - *Rationale*: Include files define the interface for the code in implementation files. Including one but not linking the other is confusing. Please avoid that. Moving functions from the `.h` to the `.cpp` should not result in build errors.
- Use the RAII (Resource Acquisition Is Initialization) paradigm where possible. For example, by using `unique_ptr` for allocations in a function.
    - *Rationale*: This avoids memory and resource leaks, and ensures exception safety.

C++ data structures

- Never use the `std::map []` syntax when reading from a map, but instead use `.find()`.
    - *Rationale*: `[]` does an insert (of the default element) if the item doesn't exist in the map yet. This has resulted in memory leaks in the past, as well as race conditions (expecting read-read behavior). Using `[]` is fine for *writing* to a map.
- Do not compare an iterator from one data structure with an iterator of another data structure (even if of the same type).

    - *Rationale*: Behavior is undefined. In C++ parlor this means "may reformat the universe", in practice this has resulted in at least one hard-to-debug crash bug.
- Watch out for out-of-bounds vector access. `&vch[vch.size()]` is illegal, including `&vch[0]` for an empty vector. Use `vch.data()` and `vch.data() + vch.size()` instead.
- Vector bounds checking is only enabled in debug mode. Do not rely on it.
- Initialize all non-static class members where they are defined. If this is skipped for a good reason (i.e., optimization on the critical path), add an explicit comment about this.
    - *Rationale*: Ensure determinism by avoiding accidental use of uninitialized values. Also, static analyzers balk about this. Initializing the members in the declaration makes it easy to spot uninitialized ones.

```
class A
{
    uint32_t m_count{0};
}
```

- By default, declare constructors `explicit`.
    - *Rationale*: This is a precaution to avoid unintended conversions.
- Use explicitly signed or unsigned `char`s, or even better `uint8_t` and `int8_t`. Do not use bare `char` unless it is to pass to a third-party API. This type can be signed or unsigned depending on the architecture, which can lead to interoperability problems or dangerous conditions such as out-of-bounds array accesses.
- Prefer explicit constructions over implicit ones that rely on 'magical' C++ behavior.
    - *Rationale*: Easier to understand what is happening, thus easier to spot mistakes, even for those that are not language lawyers.
- Use `Span` as function argument when it can operate on any range-like container.
    - *Rationale*: Compared to `Foo(const vector<int>&)` this avoids the need for a (potentially expensive) conversion to vector if the caller happens to have the input stored in another type of container. However, be aware of the pitfalls documented in span.h.

```
void Foo(Span<const int> data);

std::vector<int> vec{1,2,3};
Foo(vec);
```

- Prefer `enum class` (scoped enumerations) over `enum` (traditional enumerations) where possible.
    - *Rationale*: Scoped enumerations avoid two potential pitfalls/problems with traditional C++ enumerations: implicit conversions to `int`, and name clashes due to enumerators being exported to the surrounding scope.
- `switch` statement on an enumeration example:

```
enum class Tabs {
    info,
    console,
    network_graph,
    peers
};

int GetInt(Tabs tab)
{
    switch (tab) {
    case Tabs::info: return 0;
    case Tabs::console: return 1;
    case Tabs::network_graph: return 2;
    case Tabs::peers: return 3;
    } // no default case, so the compiler can warn about missing cases
    assert(false);
}
```

Rationale: The comment documents skipping `default:` label, and it complies with `clang-format` rules. The assertion prevents firing of `-Wreturn-type` warning on some compilers.

Strings and formatting

- Be careful of `LogPrint` versus `LogPrintf`. `LogPrint` takes a `category` argument, `LogPrintf` does not.
    - *Rationale*: Confusion of these can result in runtime exceptions due to formatting mismatch, and it is easy to get wrong because of subtly similar naming.
- Use `std::string`, avoid C string manipulation functions.
    - *Rationale*: C++ string handling is marginally safer, less scope for buffer overflows, and surprises with `\0` characters. Also, some C string manipulations tend to act differently depending on platform, or even the user locale.
- Use `ParseInt32`, `ParseInt64`, `ParseUInt32`, `ParseUInt64`, `ParseDouble` from `utilstrencodings.h` for number parsing.
    - *Rationale*: These functions do overflow checking and avoid pesky locale issues.
- Avoid using locale dependent functions if possible. You can use the provided `lint-locale-dependence.sh` to check for accidental use of locale dependent functions.
    - *Rationale*: Unnecessary locale dependence can cause bugs that are very tricky to isolate and fix.
    - These functions are known to be locale dependent: `alphasort`, `asctime`, `asprintf`, `atof`, `atoi`, `atol`, `atoll`, `atoq`, `btowc`, `ctime`, `dprintf`, `fgetwc`, `fgetws`, `fprintf`, `fputwc`, `fputws`, `fscanf`, `fwprintf`, `getdate`, `getwc`, `getwchar`, `isalnum`, `isalpha`, `isblank`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `iswalnum`, `iswalpha`, `iswblank`, `iswcntrl`, `iswctype`, `iswdigit`, `iswgraph`, `iswlower`, `iswprint`, `iswpunct`, `iswspace`, `iswupper`, `iswxdigit`, `isxdigit`, `mblen`, `mbrlen`, `mbrtowc`,

`mbsinit`, `mbsnrtowcs`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `mktime`, `putwc`, `putwchar`, `scanf`, `snprintf`, `sprintf`, `sscanf`, `stoi`, `stol`, `stoll`, `strcasecmp`, `strcasestr`, `strcoll`, `strfmon`, `strftime`, `strncasecmp`, `strptime`, `strtod`, `strtof`, `strtoimax`, `strtol`, `strtold`, `strtoll`, `strtoq`, `strtoul`, `strtoull`, `strtoumax`, `strtouq`, `strxfrm`, `swprintf`, `tolower`, `toupper`, `towctrans`, `towlower`, `towupper`, `ungetwc`, `vasprintf`, `vdprintf`, `versionsort`, `vfprintf`, `vfscanf`, `vfwprintf`, `vprintf`, `vscanf`, `vsnprintf`, `vsprintf`, `vsscanf`, `vswprintf`, `vwprintf`, `wcrtomb`, `wcscasecmp`, `wcscoll`, `wcsftime`, `wcsncasecmp`, `wcsnrtombs`, `wcsrtombs`, `wcstod`, `wcstof`, `wcstoimax`, `wcstol`, `wcstold`, `wcstoll`, `wcstombs`, `wcstoul`, `wcstoull`, `wcstoumax`, `wcswidth`, `wcsxfrm`, `wctob`, `wctomb`, `wctrans`, `wctype`, `wcwidth`, `wprintf`

- For `strprintf`, `LogPrint`, `LogPrintf` formatting characters don't need size specifiers.
  - *Rationale*: Bitcoin Core uses tinyformat, which is type safe. Leave them out to avoid confusion.
- Use `.c_str()` sparingly. Its only valid use is to pass C++ strings to C functions that take NULL-terminated strings.
  - Do not use it when passing a sized array (so along with `.size()`). Use `.data()` instead to get a pointer to the raw data.
    - *Rationale*: Although this is guaranteed to be safe starting with C++11, `.data()` communicates the intent better.
  - Do not use it when passing strings to `tfm::format`, `strprintf`, `LogPrint[f]`.
    - *Rationale*: This is redundant. Tinyformat handles strings.
  - Do not use it to convert to `QString`. Use `QString::fromStdString()`.
    - *Rationale*: Qt has built-in functionality for converting their string type from/to C++. No need to roll your own.
  - In cases where do you call `.c_str()`, you might want to additionally check that the string does not contain embedded '\0' characters, because it will (necessarily) truncate the string. This might be used to hide parts of the string from logging or to circumvent checks. If a use of strings is sensitive to this, take care to check the string for embedded NULL characters first and reject it if there are any (see `ParsePrechecks` in `strencodings.cpp` for an example).

Shadowing

Although the shadowing warning (`-Wshadow`) is not enabled by default (it prevents issues arising from using a different variable with the same name), please name variables so that their names do not shadow variables defined in the source code.

When using nested cycles, do not name the inner cycle variable the same as in the upper cycle, etc.

Threads and synchronization

- Prefer `Mutex` type to `RecursiveMutex` one

- Consistently use Clang Thread Safety Analysis annotations to get compile-time warnings about potential race conditions in code. Combine annotations in function declarations with run-time asserts in function definitions:
    - In functions that are declared separately from where they are defined, the thread safety annotations should be added exclusively to the function declaration. Annotations on the definition could lead to false positives (lack of compile failure) at call sites between the two.

```
// txmempool.h
class CTxMemPool
{
public:
    ...
    mutable RecursiveMutex cs;
    ...
    void UpdateTransactionsFromBlock(...) EXCLUSIVE_LOCKS_REQUIRED(::cs_main, cs);
    ...
}

// txmempool.cpp
void CTxMemPool::UpdateTransactionsFromBlock(...)
{
    AssertLockHeld(::cs_main);
    AssertLockHeld(cs);
    ...
}
```

```
// validation.h
class ChainstateManager
{
public:
    ...
    bool ProcessNewBlock(...) LOCKS_EXCLUDED(::cs_main);
    ...
}

// validation.cpp
bool ChainstateManager::ProcessNewBlock(...)
{
    AssertLockNotHeld(::cs_main);
    ...
    LOCK(::cs_main);
    ...
}
```

- Build and run tests with `-DDEBUG_LOCKORDER` to verify that no potential deadlocks are introduced. As of 0.12, this is defined by default when configuring with `--enable-debug`.
- When using `LOCK` / `TRY_LOCK` be aware that the lock exists in the context of the current scope, so surround the statement and the code that needs the lock with braces.

OK:

```
{
    TRY_LOCK(cs_vNodes, lockNodes);
    ...
}
```

Wrong:

```
TRY_LOCK(cs_vNodes, lockNodes);
{
    ...
}
```

Scripts

Shebang

- Use `#!/usr/bin/env bash` instead of obsolete `#!/bin/bash`.
    - *Rationale*:

      `#!/bin/bash` assumes it is always installed to /bin/ which can cause issues;

      `#!/usr/bin/env bash` searches the user's PATH to find the bash binary.

OK:

```
#!/usr/bin/env bash
```

Wrong:

```
#!/bin/bash
```

Source code organization

- Implementation code should go into the `.cpp` file and not the `.h`, unless necessary due to template usage or when performance due to inlining is critical.
    - *Rationale*: Shorter and simpler header files are easier to read and reduce compile time.
- Use only the lowercase alphanumerics (`a-z0-9`), underscore (`_`) and hyphen (`-`) in source code filenames.
    - *Rationale*: `grep` :ing and auto-completing filenames is easier when using a consistent naming pattern. Potential problems when building on case-insensitive filesystems are avoided when using only lowercase characters in source code filenames.

- Every `.cpp` and `.h` file should `#include` every header file it directly uses classes, functions or other definitions from, even if those headers are already included indirectly through other headers.
    - *Rationale*: Excluding headers because they are already indirectly included results in compilation failures when those indirect dependencies change. Furthermore, it obscures what the real code dependencies are.
- Don't import anything into the global namespace (`using namespace ...`). Use fully specified types such as `std::string` .
    - *Rationale*: Avoids symbol conflicts.
- Terminate namespaces with a comment (`// namespace mynamespace`). The comment should be placed on the same line as the brace closing the namespace, e.g.

```
namespace mynamespace {
...
} // namespace mynamespace

namespace {
...
} // namespace
```

- *Rationale*: Avoids confusion about the namespace context.
- Use `#include <primitives/transaction.h>` bracket syntax instead of `#include "primitives/transactions.h"` quote syntax.
    - *Rationale*: Bracket syntax is less ambiguous because the preprocessor searches a fixed list of include directories without taking location of the source file into account. This allows quoted includes to stand out more when the location of the source file actually is relevant.
- Use include guards to avoid the problem of double inclusion. The header file `foo/bar.h` should use the include guard identifier `BITCOIN_FOO_BAR_H` , e.g.

```
#ifndef BITCOIN_FOO_BAR_H
#define BITCOIN_FOO_BAR_H
...
#endif // BITCOIN_FOO_BAR_H
```

GUI

- Do not display or manipulate dialogs in model code (classes `*Model`).
    - *Rationale*: Model classes pass through events and data from the core, they should not interact with the user. That's where View classes come in. The converse also holds: try to not directly access core data structures from Views.
- Avoid adding slow or blocking code in the GUI thread. In particular, do not add new `interfaces::Node` and `interfaces::Wallet` method calls, even if they may be fast now, in case they are changed to lock or communicate across processes in the future.

Prefer to offload work from the GUI thread to worker threads (see `RPCExecutor` in console code as an example) or take other steps (see https://doc.qt.io/archives/qq/qq27-responsive-guis.html) to keep the GUI responsive.

  - *Rationale*: Blocking the GUI thread can increase latency, and lead to hangs and deadlocks.

Subtrees

Several parts of the repository are subtrees of software maintained elsewhere.

Some of these are maintained by active developers of Bitcoin Core, in which case changes should go directly upstream without being PRed directly against the project. They will be merged back in the next subtree merge.

Others are external projects without a tight relationship with our project. Changes to these should also be sent upstream, but bugfixes may also be prudent to PR against a Bitcoin Core subtree, so that they can be integrated quickly. Cosmetic changes should be taken upstream.

There is a tool in `test/lint/git-subtree-check.sh` (instructions) to check a subtree directory for consistency with its upstream repository.

Current subtrees include:

- src/leveldb
  - Subtree at https://github.com/bitcoin-core/leveldb-subtree ; maintained by Core contributors.
  - Upstream at https://github.com/google/leveldb ; maintained by Google. Open important PRs to the subtree to avoid delay.
  - **Note**: Follow the instructions in Upgrading LevelDB when merging upstream changes to the LevelDB subtree.
- src/crc32c
  - Used by leveldb for hardware acceleration of CRC32C checksums for data integrity.
  - Subtree at https://github.com/bitcoin-core/crc32c-subtree ; maintained by Core contributors.
  - Upstream at https://github.com/google/crc32c ; maintained by Google.
- src/secp256k1
  - Upstream at https://github.com/bitcoin-core/secp256k1/ ; maintained by Core contributors.
- src/crypto/ctaes
  - Upstream at https://github.com/bitcoin-core/ctaes ; maintained by Core contributors.
- src/univalue
  - Subtree at https://github.com/bitcoin-core/univalue-subtree ; maintained by Core contributors.
  - Deviates from upstream https://github.com/jgarzik/univalue.
- src/minisketch
  - Upstream at https://github.com/sipa/minisketch ; maintained by Core contributors.

Upgrading LevelDB

Extra care must be taken when upgrading LevelDB. This section explains issues you must be aware of.

File Descriptor Counts

In most configurations, we use the default LevelDB value for `max_open_files`, which is 1000 at the time of this writing. If LevelDB actually uses this many file descriptors, it will cause problems with Bitcoin's `select()` loop, because it may cause new sockets to be created where the fd value is >= 1024. For this reason, on 64-bit Unix systems, we rely on an internal LevelDB optimization that uses `mmap()` + `close()` to open table files without actually retaining references to the table file descriptors. If you are upgrading LevelDB, you must sanity check the changes to make sure that this assumption remains valid.

In addition to reviewing the upstream changes in `env_posix.cc`, you can use `lsof` to check this. For example, on Linux this command will show open `.ldb` file counts:

```
$ lsof -p $(pidof bitcoind) |\
    awk 'BEGIN { fd=0; mem=0; } /ldb$/ { if ($4 == "mem") mem++; else fd++ } END {
printf "mem = %s, fd = %s\n", mem, fd}'
mem = 119, fd = 0
```

The `mem` value shows how many files are mmap'ed, and the `fd` value shows you many file descriptors these files are using. You should check that `fd` is a small number (usually 0 on 64-bit hosts).

See the notes in the `SetMaxOpenFiles()` function in `dbwrapper.cc` for more details.

Consensus Compatibility

It is possible for LevelDB changes to inadvertently change consensus compatibility between nodes. This happened in Bitcoin 0.8 (when LevelDB was first introduced). When upgrading LevelDB, you should review the upstream changes to check for issues affecting consensus compatibility.

For example, if LevelDB had a bug that accidentally prevented a key from being returned in an edge case, and that bug was fixed upstream, the bug "fix" would be an incompatible consensus change. In this situation, the correct behavior would be to revert the upstream fix before applying the updates to Bitcoin's copy of LevelDB. In general, you should be wary of any upstream changes affecting what data is returned from LevelDB queries.

Scripted diffs

For reformatting and refactoring commits where the changes can be easily automated using a bash script, we use scripted-diff commits. The bash script is included in the commit message and our CI job checks that the result of the script is identical to the commit. This aids reviewers since they can verify that the script does exactly what it is supposed to do. It is also helpful for rebasing (since the same script can just be re-run on the new master commit).

To create a scripted-diff:

- start the commit message with `scripted-diff:` (and then a description of the diff on the same line)
- in the commit message include the bash script between lines containing just the following text:
  - `-BEGIN VERIFY SCRIPT-`
  - `-END VERIFY SCRIPT-`

The scripted-diff is verified by the tool `test/lint/commit-script-check.sh`. The tool's default behavior, when supplied with a commit is to verify all scripted-diffs from the beginning of time up to said commit. Internally, the tool passes the first supplied argument to `git rev-list --reverse` to determine which commits to verify script-diffs for, ignoring commits that don't conform to the commit message format described above.

For development, it might be more convenient to verify all scripted-diffs in a range `A..B`, for example:

```
test/lint/commit-script-check.sh origin/master..HEAD
```

Suggestions and examples

If you need to replace in multiple files, prefer `git ls-files` to `find` or globbing, and `git grep` to `grep`, to avoid changing files that are not under version control.

For efficient replacement scripts, reduce the selection to the files that potentially need to be modified, so for example, instead of a blanket `git ls-files src | xargs sed -i s/apple/orange/`, use `git grep -l apple src | xargs sed -i s/apple/orange/`.

Also, it is good to keep the selection of files as specific as possible — for example, replace only in directories where you expect replacements — because it reduces the risk that a rebase of your commit by re-running the script will introduce accidental changes.

Some good examples of scripted-diff:

- scripted-diff: Rename InitInterfaces to NodeContext uses an elegant script to replace occurrences of multiple terms in all source files.
- scripted-diff: Remove g_connman, g_banman globals replaces specific terms in a list of specific source files.
- scripted-diff: Replace fprintf with tfm::format does a global replacement but excludes certain directories.

To find all previous uses of scripted diffs in the repository, do:

```
git log --grep="-BEGIN VERIFY SCRIPT-"
```

Release notes

Release notes should be written for any PR that:

- introduces a notable new feature
- fixes a significant bug
- changes an API or configuration model
- makes any other visible change to the end-user experience.

Release notes should be added to a PR-specific release note file at `/doc/release-notes-<PR number>.md` to avoid conflicts between multiple PRs. All `release-notes*` files are merged into a single /doc/release-notes.md file prior to the release.

RPC interface guidelines

A few guidelines for introducing and reviewing new RPC interfaces:

- Method naming: use consecutive lower-case names such as `getrawtransaction` and `submitblock`.
  - *Rationale*: Consistency with the existing interface.
- Argument naming: use snake case `fee_delta` (and not, e.g. camel case `feeDelta`)
  - *Rationale*: Consistency with the existing interface.

- Use the JSON parser for parsing, don't manually parse integers or strings from arguments unless absolutely necessary.
    - *Rationale*: Introduces hand-rolled string manipulation code at both the caller and callee sites, which is error-prone, and it is easy to get things such as escaping wrong. JSON already supports nested data structures, no need to re-invent the wheel.
    - *Exception*: AmountFromValue can parse amounts as string. This was introduced because many JSON parsers and formatters hard-code handling decimal numbers as floating-point values, resulting in potential loss of precision. This is unacceptable for monetary values. **Always** use `AmountFromValue` and `ValueFromAmount` when inputting or outputting monetary values. The only exceptions to this are `prioritisetransaction` and `getblocktemplate` because their interface is specified as-is in BIP22.
- Missing arguments and 'null' should be treated the same: as default values. If there is no default value, both cases should fail in the same way. The easiest way to follow this guideline is to detect unspecified arguments with `params[x].isNull()` instead of `params.size() <= x`. The former returns true if the argument is either null or missing, while the latter returns true if is missing, and false if it is null.
    - *Rationale*: Avoids surprises when switching to name-based arguments. Missing name-based arguments are passed as 'null'.
- Try not to overload methods on argument type. E.g. don't make `getblock(true)` and `getblock("hash")` do different things.
    - *Rationale*: This is impossible to use with `bitcoin-cli`, and can be surprising to users.
    - *Exception*: Some RPC calls can take both an `int` and `bool`, most notably when a bool was switched to a multi-value, or due to other historical reasons. **Always** have false map to 0 and true to 1 in this case.
- Don't forget to fill in the argument names correctly in the RPC command table.
    - *Rationale*: If not, the call cannot be used with name-based arguments.
- Add every non-string RPC argument `(method, idx, name)` to the table `vRPCConvertParams` in `rpc/client.cpp`.
    - *Rationale*: `bitcoin-cli` and the GUI debug console use this table to determine how to convert a plaintext command line to JSON. If the types don't match, the method can be unusable from there.
- A RPC method must either be a wallet method or a non-wallet method. Do not introduce new methods that differ in behavior based on the presence of a wallet.
    - *Rationale*: As well as complicating the implementation and interfering with the introduction of multi-wallet, wallet and non-wallet code should be separated to avoid introducing circular dependencies between code units.
- Try to make the RPC response a JSON object.
    - *Rationale*: If a RPC response is not a JSON object, then it is harder to avoid API breakage if new data in the response is needed.

- Wallet RPCs call BlockUntilSyncedToCurrentChain to maintain consistency with `getblockchaininfo`'s state immediately prior to the call's execution. Wallet RPCs whose behavior does *not* depend on the current chainstate may omit this call.
    - *Rationale*: In previous versions of Bitcoin Core, the wallet was always in-sync with the chainstate (by virtue of them all being updated in the same cs_main lock). In order to maintain the behavior that wallet RPCs return results as of at least the highest best-known block an RPC client may be aware of prior to entering a wallet RPC call, we must block until the wallet is caught up to the chainstate as of the RPC call's entry. This also makes the API much easier for RPC clients to reason about.
- Be aware of RPC method aliases and generally avoid registering the same callback function pointer for different RPCs.
    - *Rationale*: RPC methods registered with the same function pointer will be considered aliases and only the first method name will show up in the `help` RPC command list.
    - *Exception*: Using RPC method aliases may be appropriate in cases where a new RPC is replacing a deprecated RPC, to avoid both RPCs confusingly showing up in the command list.
- Use *invalid* bech32 addresses (e.g. in the constant array `EXAMPLE_ADDRESS`) for `RPCExamples` help documentation.
    - *Rationale*: Prevent accidental transactions by users and encourage the use of bech32 addresses by default.
- Use the `UNIX_EPOCH_TIME` constant when describing UNIX epoch time or timestamps in the documentation.
    - *Rationale*: User-facing consistency.
- Use `fs::path::u8string()` and `fs::u8path()` functions when converting path to JSON strings, not `fs::PathToString` and `fs::PathFromString`
    - *Rationale*: JSON strings are Unicode strings, not byte strings, and RFC8259 requires JSON to be encoded as UTF-8.

Internal interface guidelines

Internal interfaces between parts of the codebase that are meant to be independent (node, wallet, GUI), are defined in `src/interfaces/`. The main interface classes defined there are `interfaces::Chain`, used by wallet to access the node's latest chain state, `interfaces::Node`, used by the GUI to control the node, and `interfaces::Wallet`, used by the GUI to control an individual wallet. There are also more specialized interface types like `interfaces::Handler` `interfaces::ChainClient` passed to and from various interface methods.

Interface classes are written in a particular style so node, wallet, and GUI code doesn't need to run in the same process, and so the class declarations work more easily with tools and libraries supporting interprocess communication:

- Interface classes should be abstract and have methods that are pure virtual. This allows multiple implementations to inherit from the same interface class, particularly so one implementation can execute functionality in the local process, and other implementations can forward calls to remote processes.

- Interface method definitions should wrap existing functionality instead of implementing new functionality. Any substantial new node or wallet functionality should be implemented in `src/node/` or `src/wallet/` and just exposed in `src/interfaces/` instead of being implemented there, so it can be more modular and accessible to unit tests.

- Interface method parameter and return types should either be serializable or be other interface classes. Interface methods shouldn't pass references to objects that can't be serialized or accessed from another process.

  Examples:

  ```
  // Good: takes string argument and returns interface class pointer
  virtual unique_ptr<interfaces::Wallet> loadWallet(std::string filename) = 0;

  // Bad: returns CWallet reference that can't be used from another process
  virtual CWallet& loadWallet(std::string filename) = 0;
  ```

  ```
  // Good: accepts and returns primitive types
  virtual bool findBlock(const uint256& hash, int& out_height, int64_t&
  out_time) = 0;

  // Bad: returns pointer to internal node in a linked list inaccessible to
  // other processes
  virtual const CBlockIndex* findBlock(const uint256& hash) = 0;
  ```

  ```
  // Good: takes plain callback type and returns interface pointer
  using TipChangedFn = std::function<void(int block_height, int64_t
  block_time)>;
  virtual std::unique_ptr<interfaces::Handler> handleTipChanged(TipChangedFn
  fn) = 0;

  // Bad: returns boost connection specific to local process
  using TipChangedFn = std::function<void(int block_height, int64_t
  block_time)>;
  virtual boost::signals2::scoped_connection connectTipChanged(TipChangedFn fn)
  = 0;
  ```

- For consistency and friendliness to code generation tools, interface method input and inout parameters should be ordered first and output parameters should come last.

  Example:

  ```
  // Good: error output param is last
  virtual bool broadcastTransaction(const CTransactionRef& tx, CAmount max_fee,
  std::string& error) = 0;

  // Bad: error output param is between input params
  virtual bool broadcastTransaction(const CTransactionRef& tx, std::string&
  error, CAmount max_fee) = 0;
  ```

- For friendliness to code generation tools, interface methods should not be overloaded:

  Example:

  ```
  // Good: method names are unique
  virtual bool disconnectByAddress(const CNetAddr& net_addr) = 0;
  virtual bool disconnectById(NodeId id) = 0;

  // Bad: methods are overloaded by type
  virtual bool disconnect(const CNetAddr& net_addr) = 0;
  virtual bool disconnect(NodeId id) = 0;
  ```

- For consistency and friendliness to code generation tools, interface method names should be `lowerCamelCase` and standalone function names should be `UpperCamelCase`.

  Examples:

  ```
  // Good: lowerCamelCase method name
  virtual void blockConnected(const CBlock& block, int height) = 0;

  // Bad: uppercase class method
  virtual void BlockConnected(const CBlock& block, int height) = 0;
  ```

  ```
  // Good: UpperCamelCase standalone function name
  std::unique_ptr<Node> MakeNode(LocalInit& init);

  // Bad: lowercase standalone function
  std::unique_ptr<Node> makeNode(LocalInit& init);
  ```

  Note: This last convention isn't generally followed outside of `src/interfaces/`, though it did come up for discussion before in #14635.