Pacemaker Development

Release 3.0.0

the Pacemaker project contributors

Jan 09, 2025

CONTENTS

Working with the Pacemaker Code Base

CHAPTER

ONE

ABSTRACT

This document has guidelines and tips for developers interested in editing Pacemaker source code and submitting changes for inclusion in the project. Start with the FAQ; the rest is optional detail.

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Frequently Asked Questions

Q Who is this document intended for?

A Anyone who wishes to read and/or edit the Pacemaker source code. Casual contributors should feel free to read just this FAQ, and consult other chapters as needed.

Q Where is the source code for Pacemaker?

A The source code for Pacemaker is kept on GitHub, as are all software projects under the ClusterLabs umbrella. Pacemaker uses Git for source code management. If you are a Git newbie, the gittutorial(7) man page is an excellent starting point. If you're familiar with using Git from the command line, you can create a local copy of the Pacemaker source code with: **git clone https://github.com/ClusterLabs/pacemaker.git**

Q What are the different Git branches and repositories used for?

A

- The main branch is used for all new development.
- The 3.0 and 2.1 branches are for the currently supported major and minor version release series. Normally, they do not receive any changes, but during the release cycle for a new release, they will contain release candidates. The main branch is pulled into 3.0 just before the first release candidate of a new release, but otherwise, separate pull requests must be submitted to backport changes from the main branch into a release branch.
- The 2.0 branch, 1.1 branch, and separate 1.0 repository are frozen snapshots of earlier release series, no longer being developed.

Q How do I build from the source code?

A See INSTALL.md in the main checkout directory.

Q What coding style should I follow?

A You'll be mostly fine if you simply follow the example of existing code. When unsure, see the relevant chapter of this document for language-specific recommendations. Pacemaker has grown and evolved organically over many years, so you will see much code that doesn't conform to the current guidelines. We discourage making changes solely to bring code into conformance, as any change requires developer time for review and opens the possibility of adding bugs. However, new code should follow the guidelines, and it is fine to bring lines of older code into conformance when modifying that code for other reasons.

Q How should I format my Git commit messages?

A An example is "Feature: scheduler: wobble the frizzle better".

- The first part is the type of change, used to automatically generate the change log for the next release. Commit messages with the following will be included in the change log:
  - **Feature** for new features
  - **Fix** for bug fixes (**Bug** or **High** also work)
  - **API** for changes to the public API

  Everything else will *not* automatically be in the change log, and so don't really matter, but types commonly used include:
  - **Log** for changes to log messages or handling
  - **Doc** for changes to documentation or comments
  - **Test** for changes in CTS and regression tests
  - **Low**, **Med**, or **Mid** for bug fixes not significant enough for a change log entry
  - **Refactor** for refactoring-only code changes
  - **Build** for build process changes
- The next part is the name of the component(s) being changed, for example, **controller** or **libcrmcommon** (it's more free-form, so don't sweat getting it exact).
- The rest briefly describes the change. The git project recommends the entire summary line stay under 50 characters, but more is fine if needed for clarity.
- Except for the most simple and obvious of changes, the summary should be followed by a blank line and a longer explanation of *why* the change was made.
- If the commit is associated with a task in the ClusterLabs project manager, you can say "Fixes T*n*" in the commit message to automatically close task T*n* when the pull request is merged.

Q How can I test my changes?

A The source repository has some unit tests for simple functions, though this is a recent effort without much coverage yet. Pacemaker's Cluster Test Suite (CTS) has regression tests for most major components; these will automatically be run for any pull requests submitted through GitHub, and are sufficient for most changes. Additionally, CTS has a lab component that can be used to set up a test cluster and run a wide variety of complex tests, for testing major changes. See cts/README.md in the source repository for details.

Q What is Pacemaker's license?

A Except where noted otherwise in the file itself, the source code for all Pacemaker programs is licensed under version 2 or later of the GNU General Public License (GPLv2+), its headers, libraries, and native language translations under version 2.1 or later of the less restrictive GNU Lesser General Public License (LGPLv2.1+), its documentation under version 4.0 or later of the Creative Commons Attribution-ShareAlike International Public License (CC-BY-SA-4.0), and its init scripts under the Revised BSD license. If you find any deviations from this policy, or wish to inquire about alternate licensing arrangements, please e-mail the developers@ClusterLabs.org mailing list. Licensing issues are also discussed on the ClusterLabs wiki.

Q How can I contribute my changes to the project?

A Contributions of bug fixes or new features are very much appreciated! Patches can be submitted as pull requests via GitHub (the preferred method, due to its excellent features), or e-mailed to the developers@ClusterLabs.org mailing list as an attachment in a format Git can import. Authors may only submit changes that they have the right to submit under the open source license indicated in the affected files.

Q What if I still have questions?

A Ask on the ClusterLabs mailing lists.

2.2 General Guidelines for All Languages

2.2.1 Copyright

When copyright notices are added to a file, they should look like this:

Note: Copyright Notice Format

Copyright *YYYY[-YYYY]* the Pacemaker project contributors

The version control history for this file may have further details.

The first *YYYY* is the year the file was *originally* published. The original date is important for two reasons: when two entities claim copyright ownership of the same work, the earlier claim generally prevails; and copyright expiration is generally calculated from the original publication date.[1]

If the file is modified in later years, add *-YYYY* with the most recent year of modification. Even though Pacemaker is an ongoing project, copyright notices are about the years of *publication* of specific content.

Copyright notices are intended to indicate, but do not affect, copyright *ownership*, which is determined by applicable laws and regulations. Authors may put more specific copyright notices in their commit messages if desired.

[1] See the U.S. Copyright Office's "Compendium of U.S. Copyright Office Practices", particularly "Chapter 2200: Notice of Copyright", sections 2205.1(A) and 2205.1(F), or "Updating Copyright Notices" for a more readable summary.

2.2.2 Terminology

Pacemaker is extremely complex, and it helps to use terminology consistently throughout documentation, symbol names and comments in code, and so forth. It also helps to use natural language when practical instead of technical jargon and acronyms.

For specific recommendations, see the *Glossary*.

2.3 Documentation Guidelines

See doc/README.md in the source code repository for the kinds of documentation that Pacemaker provides.

2.3.1 Books

The `doc/sphinx` subdirectory has a subdirectory for each book by title. Each book's directory contains .rst files, which are the chapter sources in reStructuredText format (with index.rst as the starting point).

Once you have edited the sources as desired, run `make` in the `doc` or `doc/sphinx` directory to generate all the books locally. You can view the results by pointing your web browser to (replacing PATH_TO_CHECKOUT and BOOK_TITLE appropriately):

file:///PATH_TO_CHECKOUT/doc/sphinx/BOOK_TITLE/_build/html/index.html

See the comments at the top of `doc/sphinx/Makefile.am` for options you can use.

Recommended practices:

- Use `list-table` instead of `table` for tables
- When documenting newly added features and syntax, add "*(since X.Y.Z)*" with the version introducing them. These comments can be removed when rolling upgrades from that version are no longer supported.

2.4 Python Coding Guidelines

2.4.1 Python Boilerplate

If a Python file is meant to be executed (as opposed to imported), it should have a `.in` extension, and its first line should be:

```
#!@PYTHON@
```

which will be replaced with the appropriate python executable when Pacemaker is built. To make that happen, add an entry to `CONFIG_FILES_EXEC()` in `configure.ac`, and add the file name without `.in` to `.gitignore` (see existing examples).

After the above line if any, every Python file should start like this:

```
""" <BRIEF-DESCRIPTION>
"""

__copyright__ = "Copyright <YYYY[-YYYY]> the Pacemaker project contributors"
__license__ = "<LICENSE> WITHOUT ANY WARRANTY"
```

<BRIEF-DESCRIPTION> is obviously a brief description of the file's purpose. The string may contain any other information typically used in a Python file docstring.

`<LICENSE>` should follow the policy set forth in the COPYING file, generally one of "GNU General Public License version 2 or later (GPLv2+)" or "GNU Lesser General Public License version 2.1 or later (LGPLv2.1+)".

2.4.2 Python Version Compatibility

Pacemaker targets compatibility with Python 3.6 and later.

Do not use features not available in all targeted Python versions. An example is the `subprocess.run()` function.

2.4.3 Formatting Python Code

- Indentation must be 4 spaces, no tabs.
- Do not leave trailing whitespace.
- Lines should be no longer than 80 characters unless limiting line length significantly impacts readability. For Python, this limitation is flexible since breaking a line often impacts readability, but definitely keep it under 120 characters.
- Where not conflicting with this style guide, it is recommended (but not required) to follow PEP 8.
- It is recommended (but not required) to format Python code such that `pylint --disable=line-too-long,too-many-lines,too-many-instance-attributes, too-many-arguments,too-many-statements` produces minimal complaints (even better if you don't need to disable all those checks).

2.5 C Coding Guidelines

Pacemaker is a large project accepting contributions from developers with a wide range of skill levels and organizational affiliations, and maintained by multiple people over long periods of time. Following consistent guidelines makes reading, writing, and reviewing code easier, and helps avoid common mistakes.

Some existing Pacemaker code does not follow these guidelines, for historical reasons and API backward compatibility, but new code should.

2.5.1 Code Organization

Pacemaker's C code is organized as follows:

Directory	Contents
daemons	the Pacemaker daemons (pacemakerd, pacemaker-based, etc.)
include	header files for library APIs
lib	libraries
tools	command-line tools

Source file names should be unique across the entire project, to allow for individual tracing via `PCMK_trace_files`.

2.5.2 Pacemaker Libraries

Library	Symbol prefix	Source location	API Headers	Description
libcib	cib	lib/cib	include/crm/cib.h include/crm/cib/*	API for pacemaker-based IPC and the CIB
libcrmcluster	pcmk	lib/cluster	include/crm/cluster.h include/crm/cluster/*	Abstract interface to underlying cluster layer
libcrmcommon	pcmk	lib/common	include/crm/common/* some of include/crm/*	Everything else
libcrmservice	svc	lib/services	include/crm/services.h	Abstract interface to supported resource types (OCF, LSB, etc.)
liblrmd	lrmd	lib/lrmd	include/crm/lrmd*.h	API for pacemaker-execd IPC
libpacemaker	pcmk	lib/pacemaker	include/pacemaker*.h include/pcmki/*	High-level APIs equivalent to command-line tool capabilities (and high-level internal APIs)
libpe_rules	pe	lib/pengine	include/crm/pengine/*	Scheduler functionality related to evaluating rules
libpe_status	pe	lib/pengine	include/crm/pengine/*	Low-level scheduler functionality
libstonithd	stonith	lib/fencing	include/crm/stonith-ng.h include/crm/fencing/*	API for pacemaker-fenced IPC

Public versus Internal APIs

Pacemaker libraries have both internal and public APIs. Internal APIs are those used only within Pacemaker; public APIs are those offered (via header files and documentation) for external code to use.

Generic functionality needed by Pacemaker itself, such as string processing or XML processing, should remain internal, while functions providing useful high-level access to Pacemaker capabilities should be public. When in doubt, keep APIs internal, because it's easier to expose a previously internal API than hide a previously public API.

Internal APIs can be changed as needed.

The public API/ABI should maintain a degree of stability so that external applications using it do not need to be rewritten or rebuilt frequently. Many OSes/distributions avoid breaking API/ABI compatibility within a major release, so if Pacemaker breaks compatibility, that significantly delays when OSes can package the new version. Therefore, changes to public APIs should be backward-compatible (as detailed throughout this chapter), unless we are doing a (rare) release where we specifically intend to break compatibility.

External applications known to use Pacemaker's public C API include sbd and dlm_controld.

API Symbol Naming

Exposed API symbols (non-`static` function names, `struct` and `typedef` names in header files, etc.) must begin with the prefix appropriate to the library (shown in the table at the beginning of this section). This reduces the chance of naming collisions when external software links against the library.

The prefix is usually lowercase but may be all-caps for some defined constants and macros.

Public API symbols should follow the library prefix with a single underbar (for example, `pcmk_something`), and internal API symbols with a double underbar (for example, `pcmk__other_thing`).

File-local symbols (such as static functions) and non-library code do not require a prefix, though a unique prefix indicating an executable (controld, crm_mon, etc.) can be helpful when symbols are shared between multiple source files for the executable.

API Header File Naming

- Internal API headers should be named ending in `_internal.h`, in the same location as public headers, with the exception of libpacemaker, which for historical reasons keeps internal headers in `include/pcmki/pcmki_*.h`).
- If a library needs to share symbols just within the library, header files for these should be named ending in `_private.h` and located in the library source directory (not `include`). Such functions should be declared as `G_GNUC_INTERNAL`, to aid compiler efficiency (glib defines this symbol appropriately for the compiler).

Header files that are not library API are kept in the same directory as the source code they're included from.

The easiest way to tell what kind of API a symbol is, is to see where it's declared. If it's in a public header, it's public API; if it's in an internal header, it's internal API; if it's in a library-private header, it's library-private API; otherwise, it's not an API.

API Documentation

Pacemaker uses Doxygen to automatically generate its online API documentation, so all public API (header files, functions, structs, enums, etc.) should be documented with Doxygen comment blocks. Other code may be documented in the same way if desired, with an `\internal` tag in the Doxygen comment.

Simple example of an internal function with a Doxygen comment block:

```
/*!
 * \internal
 * \brief Return string length plus 1
 *
 * Return the number of characters in a given string, plus one.
 *
 * \param[in] s  A string (must not be NULL)
 *
 * \return The length of \p s plus 1.
 */
static int
f(const char *s)
{
   return strlen(s) + 1;
}
```

Function arguments are marked as `[in]` for input only, `[out]` for output only, or `[in,out]` for both input and output.

`[in,out]` should be used for struct pointer arguments if the function can change any data accessed via the pointer. For example, if the struct contains a `GHashTable *` member, the argument should be marked as `[in,out]` if the function inserts data into the table, even if the struct members themselves are not changed. However, an argument is not `[in,out]` if something reachable via the argument is modified via a separate argument. For example, both `pcmk_resource_t` and `pcmk_node_t` contain pointers to their `pcmk_scheduler_t` and thus indirectly to each other, but if the function modifies the resource via the resource argument, the node argument does not have to be `[in,out]`.

Public API Deprecation

Public APIs may not be removed in most Pacemaker releases, but they may be deprecated.

When a public API is deprecated, it is moved to a header whose name ends in `compat.h`. The original header includes the compatibility header only if the `PCMK_ALLOW_DEPRECATED` symbol is undefined or defined to 1. This allows external code to continue using the deprecated APIs, but internal code is prevented from using them because the `crm_internal.h` header defines the symbol to 0.

2.5.3 C Boilerplate

Every C file should start with a short copyright and license notice:

```
/*
 * Copyright <YYYY[-YYYY]> the Pacemaker project contributors
 *
 * The version control history for this file may have further details.
 *
 * This source code is licensed under <LICENSE> WITHOUT ANY WARRANTY.
 */
```

<LICENSE> should follow the policy set forth in the COPYING file, generally one of "GNU General Public License version 2 or later (GPLv2+)" or "GNU Lesser General Public License version 2.1 or later (LGPLv2.1+)".

Header files should additionally protect against multiple inclusion by defining a unique symbol of the form `PCMK__<capitalized_header_name>__H`, and declare C compatibility for inclusion by C++. For example:

```
#ifndef PCMK__MY_HEADER__H
#define PCMK__MY_HEADER__H

// put #include directives here

#ifdef __cplusplus
extern "C" {
#endif

// put header code here

#ifdef __cplusplus
}
#endif

#endif // PCMK__MY_HEADER__H
```

Public API header files should give a Doxygen file description at the top of the header code. For example:

```
/*!
 * \file
 * \brief My brief description here
 * \ingroup core
 */
```

2.5.4 Line Formatting

- Indentation must be 4 spaces, no tabs.
- Do not leave trailing whitespace.
- Lines should be no longer than 80 characters unless limiting line length hurts readability.

2.5.5 Comments

```
/* Single-line comments may look like this */

// ... or this

/* Multi-line comments should start immediately after the comment opening.
 * Subsequent lines should start with an aligned asterisk. The comment
 * closing should be aligned and on a line by itself.
 */
```

2.5.6 Operators

```
// Operators have spaces on both sides
x = a;

/* (1) Do not rely on operator precedence; use parentheses when mixing
 *     operators with different priority, for readability.
 * (2) No space is used after an opening parenthesis or before a closing
```

(continues on next page)

(continued from previous page)

```
 *     parenthesis.
 */
x = a + b - (c * d);
```

2.5.7 Control Statements (if, else, while, for, switch)

```
/*
 * (1) The control keyword is followed by a space, a left parenthesis
 *     without a space, the condition, a right parenthesis, a space, and the
 *     opening bracket on the same line.
 * (2) Always use braces around control statement blocks, even if they only
 *     contain one line. This makes code review diffs smaller if a line gets
 *     added in the future, and avoids the chance of bad indenting making a
 *     line incorrectly appear to be part of the block.
 * (3) The closing bracket is on a line by itself.
 */
if (v < 0) {
    return 0;
}

/* "else" and "else if" are on the same line with the previous ending brace
 * and next opening brace, separated by a space. Blank lines may be used
 * between blocks to help readability.
 */
if (v > 0) {
    return 0;

} else if (a == 0) {
    return 1;

} else {
    return 2;
}

/* Do not use assignments in conditions. This ensures that the developer's
 * intent is always clear, makes code reviews easier, and reduces the chance
 * of using assignment where comparison is intended.
 */
// Do this ...
a = f();
if (a) {
    return 0;
}
// ... NOT this
if (a = f()) {
    return 0;
}

/* It helps readability to use the "!" operator only in boolean
 * comparisons, and explicitly compare numeric values against 0,
 * pointers against NULL, etc. This helps remind the reader of the
 * type being compared.
 */
int i = 0;
```

(continues on next page)

(continued from previous page)

```
char *s = NULL;
bool cond = false;

if (!cond) {
    return 0;
}
if (i == 0) {
    return 0;
}
if (s == NULL) {
    return 0;
}

/* In a "switch" statement, indent "case" one level, and indent the body of
 * each "case" another level.
 */
switch (expression) {
    case 0:
        command1;
        break;
    case 1:
        command2;
        break;
    default:
        command3;
        break;
}
```

2.5.8 Macros

Macros are a powerful but easily misused feature of the C preprocessor, and Pacemaker uses a lot of obscure macro features. If you need to brush up, the GCC documentation for macros is excellent.

Some common issues:

- Beware of side effects in macro arguments that may be evaluated more than once
- Always parenthesize macro arguments used in the macro body to avoid precedence issues if the argument is an expression
- Multi-statement macro bodies should be enclosed in do…while(0) to make them behave more like a single statement and avoid control flow issues

Often, a static inline function defined in a header is preferable to a macro, to avoid the numerous issues that plague macros and gain the benefit of argument and return value type checking.

2.5.9 Memory Management

- Always use `calloc()` rather than `malloc()`. It has no additional cost on modern operating systems, and reduces the severity and security risks of uninitialized memory usage bugs.
- Ensure that all dynamically allocated memory is freed when no longer needed, and not used after it is freed. This can be challenging in the more event-driven, callback-oriented sections of code.
- Free dynamically allocated memory using the free function corresponding to how it was allocated. For example, use `free()` with `calloc()`, and `g_free()` with most glib functions that allocate objects.

2.5.10 Structures

Changes to structures defined in public API headers (adding or removing members, or changing member types) are generally not possible without breaking API compatibility. However, there are exceptions:

- Public API structures can be designed such that they can be allocated only via API functions, not declared directly or allocated with standard memory functions using `sizeof`.
  - This can be enforced simply by documentating the limitation, in which case new `struct` members can be added to the end of the structure without breaking compatibility.
  - Alternatively, the structure definition can be kept in an internal header, with only a pointer type definition kept in a public header, in which case the structure definition can be changed however needed.

2.5.11 Variables

Pointers

```
/* (1) The asterisk goes by the variable name, not the type;
 * (2) Avoid leaving pointers uninitialized, to lessen the impact of
 *     use-before-assignment bugs
 */
char *my_string = NULL;

// Use space before asterisk and after closing parenthesis in a cast
char *foo = (char *) bar;
```

Globals

Global variables should be avoided in libraries when possible. State information should instead be passed as function arguments (often as a structure). This is not for thread safety – Pacemaker's use of forking ensures it will never be threaded – but it does minimize overhead, improve readability, and avoid obscure side effects.

Variable Naming

Time intervals are sometimes represented in Pacemaker code as user-defined text specifications (for example, "10s"), other times as an integer number of seconds or milliseconds, and still other times as a string representation of an integer number. Variables for these should be named with an indication of which is being used (for example, use `interval_spec`, `interval_ms`, or `interval_ms_s` instead of `interval`).

Booleans

Booleans in C can be represented by an integer type, `bool`, or `gboolean`.

Integers are sometimes useful for storing booleans when they must be converted to and from a string, such as an XML attribute value (for which `crm_element_value_int()` can be used). Integer booleans use 0 for false and nonzero (usually 1) for true.

`gboolean` should be used with glib APIs that specify it. `gboolean` should always be used with glib's `TRUE` and `FALSE` constants.

Otherwise, `bool` should be preferred. `bool` should be used with the `true` and `false` constants from the `stdbool.h` header.

Do not use equality operators when testing booleans. For example:

```
// Do this
if (bool1) {
    fn();
}
if (!bool2) {
    fn2();
}

// Not this
if (bool1 == true) {
    fn();
}
if (bool2 == false) {
    fn2();
}

// Otherwise there's no logical end ...
if ((bool1 == false) == true) {
    fn();
}
```

2.5.12 String Handling

Define Constants for Magic Strings

A "magic" string is one used for control purposes rather than human reading, and which must be exactly the same every time it is used. Examples would be configuration option names, XML attribute names, or environment variable names.

These should always be defined constants, rather than using the string literal everywhere. If someone mistypes a defined constant, the code won't compile, but if they mistype a literal, it could go unnoticed until a user runs into a problem.

String-Related Library Functions

Pacemaker's libcrmcommon has a large number of functions to assist in string handling. The most commonly used ones are:

- `pcmk__str_eq()` tests string equality (similar to `strcmp()`), but can handle NULL, and takes options for case-insensitive, whether NULL should be considered a match, etc.
- `crm_strdup_printf()` takes `printf()`-style arguments and creates a string from them (dynamically allocated, so it must be freed with `free()`). It asserts on memory failure, so the return value is always non-NULL.

String handling functions should almost always be internal API, since Pacemaker isn't intended to be used as a general-purpose library. Most are declared in `include/crm/common/strings_internal.h`. `util.h` has some older ones that are public API (for now, but will eventually be made internal).

char*, gchar*, and GString

When using dynamically allocated strings, be careful to always use the appropriate free function.

- `char*` strings allocated with something like `calloc()` must be freed with `free()`. Most Pacemaker library functions that allocate strings use this implementation.
- glib functions often use `gchar*` instead, which must be freed with `g_free()`.
- Occasionally, it's convenient to use glib's flexible `GString*` type, which must be freed with `g_string_free()`.

Regular Expressions

- Use `REG_NOSUB` with `regcomp()` whenever possible, for efficiency.
- Be sure to use `regfree()` appropriately.

2.5.13 Enumerations

- Enumerations should not have a `typedef`, and do not require any naming convention beyond what applies to all exposed symbols.
- New values should usually be added to the end of public API enumerations, because the compiler will define the values to 0, 1, etc., in the order given, and inserting a value in the middle would change the numerical values of all later values, breaking code compiled with the old values. However, if enum numerical values are explicitly specified rather than left to the compiler, new values can be added anywhere.
- When defining constant integer values, enum should be preferred over `#define` or `const` when possible. This allows type checking without consuming memory.

Flag groups

Pacemaker often uses flag groups (also called bit fields or bitmasks) for a collection of boolean options (flags/bits).

This is more efficient for storage and manipulation than individual booleans, but its main advantage is when used in public APIs, because using another bit in a bitmask is backward compatible, whereas adding a new function argument (or sometimes even a structure member) is not.

```
#include <stdint.h>

/* (1) Define an enumeration to name the individual flags, for readability.
 *     An enumeration is preferred to a series of "#define" constants
 *     because it is typed, and logically groups the related names.
 * (2) Define the values using left-shifting, which is more readable and
 *     less error-prone than hexadecimal literals (0x0001, 0x0002, 0x0004,
 *     etc.).
 * (3) Using a comma after the last entry makes diffs smaller for reviewing
 *     if a new value needs to be added or removed later.
 */
enum pcmk__some_bitmask_type {
    pcmk__some_value    = (1 << 0),
    pcmk__other_value   = (1 << 1),
    pcmk__another_value = (1 << 2),
};

/* The flag group itself should be an unsigned type from stdint.h (not
 * the enum type, since it will be a mask of the enum values and not just
```

(continues on next page)

(continued from previous page)

```
 * one of them). uint32_t is the most common, since we rarely need more than
 * 32 flags, but a smaller or larger type could be appropriate in some
 * cases.
 */
uint32_t flags = pcmk__some_value|pcmk__other_value;

/* If the values will be used only with uint64_t, define them accordingly,
 * to make compilers happier.
 */
enum pcmk__something_else {
    pcmk__whatever    = (UINT64_C(1) << 0),
};
```

We have convenience functions for checking flags (see `pcmk_any_flags_set()`, `pcmk_all_flags_set()`, and `pcmk_is_set()`) as well as setting and clearing them (see `pcmk__set_flags_as()` and `pcmk__clear_flags_as()`, usually used via wrapper macros defined for specific flag groups). These convenience functions should be preferred to direct bitwise arithmetic, for readability and logging consistency.

2.5.14 Functions

Function Naming

Function names should be unique across the entire project, to allow for individual tracing via `PCMK_trace_functions`, and make it easier to search code and follow detail logs.

Sorting

A function that sorts an entire list should have `sort` in its name. It sorts elements using a *comparison* function, which may be either hard-coded or passed as an argument.

Comparison

A comparison function for *sorting* should have `cmp` in its name and should *not* have `sort` in its name.

Constructors

A constructor creates a new dynamically allocated object. It may perform some initialization procedure on the new object.

- If the constructor always creates an independent object instance, its name should include `new`.
- If the constructor may add the new object to some existing object, its name should include `create`.

Functions that take the caller's name as an argument

Sometimes, we define a function that uses the `__FILE__`, `__func__`, and/or `__LINE__` of the caller for logging purposes, often with a wrapper macro that automatically passes them.

- The function should take those values as its first arguments.
- The function name should end in `_as()`.

- If a wrapper macro is used, its name should be the same without `_as()`.
- See `pcmk__assert_alloc()` and `pcmk__assert_alloc_as()` as examples.

Function Definitions

```
/*
 * (1) The return type goes on its own line
 * (2) The opening brace goes by itself on a line
 * (3) Use "const" with pointer arguments whenever appropriate, to allow the
 *     function to be used by more callers.
 */
int
my_func1(const char *s)
{
    return 0;
}

/* Functions with no arguments must explicitly list them as void,
 * for compatibility with strict compilers
 */
int
my_func2(void)
{
    return 0;
}

/*
 * (1) For functions with enough arguments that they must break to the next
 *     line, align arguments with the first argument.
 * (2) When a function argument is a function itself, use the pointer form.
 * (3) Declare functions and file-global variables as ``static`` whenever
 *     appropriate. This gains a slight efficiency in shared libraries, and
 *     helps the reader know that it is not used outside the one file.
 */
static int
my_func3(int bar, const char *a, const char *b, const char *c,
         void (*callback)())
{
    return 0;
}
```

Return Values

Functions that need to indicate success or failure should follow one of the following guidelines. More details, including functions for using them in user messages and converting from one to another, can be found in `include/crm/common/results.h`.

- A **standard Pacemaker return code** is one of the `pcmk_rc_*` enum values or a system errno code, as an `int`.
- `crm_exit_t` (the `CRM_EX_*` enum values) is a system-independent code suitable for the exit status of a process, or for interchange between nodes.
- Other special-purpose status codes exist, such as `enum ocf_exitcode` for the possible exit statuses of OCF resource agents (along with some Pacemaker-specific extensions). It is usually obvious when the context calls for such.

- Some older Pacemaker APIs use the now-deprecated "legacy" return values of `pcmk_ok` or the positive or negative value of one of the `pcmk_err_*` constants or system errno codes.
- Functions registered with external libraries (as callbacks for example) should use the appropriate signature defined by those libraries, rather than follow Pacemaker guidelines.

Of course, functions may have return values that aren't success/failure indicators, such as a pointer, integer count, or bool.

Comparison functions should return

- a negative integer if the first argument should sort first
- 0 if its arguments are equal for sorting purposes
- a positive integer is the second argument should sort first

Public API Functions

Unless we are doing a (rare) release where we break public API compatibility, new public API functions can be added, but existing function signatures (return type, name, and argument types) should not be changed. To work around this, an existing function can become a wrapper for a new function.

2.5.15 Logging and Output

Logging Vs. Output

Log messages and output messages are logically similar but distinct. Oversimplifying a bit, daemons log, and tools output.

Log messages are intended to help with troubleshooting and debugging. They may have a high level of technical detail, and are usually filtered by severity – for example, the system log by default gets messages of notice level and higher.

Output is intended to let the user know what a tool is doing, and is generally terser and less technical, and may even be parsed by scripts. Output might have "verbose" and "quiet" modes, but it is not filtered by severity.

Common Guidelines for All Messages

- When format strings are used for derived data types whose implementation may vary across platforms (`pid_t`, `time_t`, etc.), the safest approach is to use `%lld` in the format string, and cast the value to `long long`.
- Do not rely on `%s` handling `NULL` values properly. While the standard library functions might, not all functions using printf-style formatting does, and it's safest to get in the habit of always ensuring format values are non-NULL. If a value can be NULL, the `pcmk__s()` function is a convenient way to say "this string if not NULL otherwise this default".
- The convenience macros `pcmk__plural_s()` and `pcmk__plural_alt()` are handy when logging a word that may be singular or plural.

Log Levels

When to use each log level:

- **critical:** fatal error (usually something that would make a daemon exit)
- **error:** failure of something that affects the cluster (such as a resource action, fencing action, etc.) or daemon operation
- **warning:** minor, potential, or recoverable failures (such as something only affecting a daemon client, or invalid configuration that can be left to default)
- **notice:** important successful events (such as a node joining or leaving, resource action results, or configuration changes)
- **info:** events that would be helpful with troubleshooting (such as status section updates or elections)
- **debug:** information that would be helpful for debugging code or complex problems
- **trace:** like debug but for very noisy or low-level stuff

By default, critical through notice are logged to the system log and detail log, info is logged to the detail log only, and debug and trace are not logged (if enabled, they go to the detail log only).

Logging

Pacemaker uses libqb for logging, but wraps it with a higher level of functionality (see `include/crm/common/logging*h`).

A few macros `crm_err()`, `crm_warn()`, etc. do most of the heavy lifting.

By default, Pacemaker sends logs at notice level and higher to the system log, and logs at info level and higher to the detail log (typically `/var/log/pacemaker/pacemaker.log`). The intent is that most users will only ever need the system log, but for deeper troubleshooting and developer debugging, the detail log may be helpful, at the cost of being more technical and difficult to follow.

The same message can have more detail in the detail log than in the system log, using libqb's "extended logging" feature:

```
/* The following will log a simple message in the system log, like:

       warning: Action failed: Node not found

   with extra detail in the detail log, like:

       warning: Action failed: Node not found | rc=-1005 id=hgjjg-51006
*/
crm_warn("Action failed: %s " QB_XS " rc=%d id=%s",
         pcmk_rc_str(rc), rc, id);
```

Assertion Logging

`pcmk__assert(expr)` If `expr` is false, this will call `crm_err()` with a "Triggered fatal assertion" message (with details), then abort execution. This should be used for logic errors that should be impossible (such as a NULL function argument where not accepted) and environmental errors that can't be handled gracefully (for example, memory allocation failures, though returning `ENOMEM` is often better).

`CRM_LOG_ASSERT(expr)` If `expr` is false, this will generally log a message without aborting. If the log level is below trace, it just calls `crm_err()` with a "Triggered assert" message (with details). If the log level is trace, and the caller is a daemon, then it will fork a child process in which to dump core, as well as logging the message. If the log level is trace, and the caller is not a daemon, then it will behave like `pcmk__assert()` (i.e. log and abort). This should be used for logic or protocol errors that require no special handling.

`CRM_CHECK(expr, failed_action)` If `expr` is false, behave like `CRM_LOG_ASSERT(expr)` (that is, log a message and dump core if requested) then perform `failed_action` (which must not contain `continue`, `break`, or `errno`). This should be used for logic or protocol errors that can be handled, usually by returning an error status.

Output

Pacemaker has a somewhat complicated system for tool output. The main benefit is that the user can select the output format with the `--output-as` option (usually "text" for human-friendly output or "xml" for reliably script-parsable output, though `crm_mon` additionally supports "console" and "html").

A custom message can be defined with a unique string identifier, plus implementation functions for each supported format. The caller invokes the message using the identifier. The user selects the output format via `--output-as`, and the output code automatically calls the appropriate implementation function. Custom messages are useful when you want to output messages that are more complex than a one-line error or informational message, reproducible, and automatically handled by the output formatting system. Custom messages can contain other custom messages.

Custom message functions are implemented as follows: Start with the macro `PCMK__OUTPUT_ARGS`, whose arguments are the message name, followed by the arguments to the message. Then there is the function declaration, for which the arguments are the pointer to the current output object, then a variable argument list.

To output a custom message, you first need to create, i.e. register, the custom message that you want to output. Either call `register_message`, which registers a custom message at runtime, or make use of the collection of predefined custom messages in `fmt_functions`, which is defined in `lib/pacemaker/pcmk_output.c`. Once you have the message to be outputted, output it by calling `message`.

Note: The `fmt_functions` functions accommodate all of the output formats; the default implementation accommodates any format that isn't explicitly accommodated. The default output provides valid output for any output format, but you may still want to implement a specific output, i.e. xml, text, or html. The `message` function automatically knows which implementation to use, because the `pcmk__output_s` contains this information.

The interface (most importantly `pcmk__output_t`) is declared in `include/crm/common/output*h`. See the API comments and existing tools for examples.

Some of its important member functions are `err`, which formats error messages and `info`, which formats informational messages. Also, `list_item`, which formats list items, `begin_list`, which starts lists, and `end_list`, which ends lists, are important because lists can be useful, yet differently handled by the different output types.

2.5.16 XML

External Libraries

Pacemaker uses libxml2 and libxslt to process XML. These libraries implement only version 1.0 of the XML, XPath, and XSLT specifications.

Naming

Names of functions, constants, and enum values related to XML should contain substrings indicating the type of object they're used with, according to the following convention:

- `xml`: XML subtree, or XML generically

- `xe`: XML element node, including the attributes belonging to an element
- `xa`: XML attribute node
- `xc`: XML comment node

Private Data

Libxml2 data structures such as `xmlNode` and `xmlDoc` contain a `void *_private` member for application-specific data. Pacemaker uses this field to store internal bookkeeping data, such as changes relative to another XML tree, or ACLs.

XML documents, elements, attributes, and comments have private data. The private data field must be allocated immediately after the node is created and freed immediately before the node is freed.

Wrapper Functions

Pacemaker provides wrappers for a variety of libxml2 and libxslt functions. They should be used whenever possible. Some are merely for convenience. However, many perform additional, Pacemaker-specific tasks, such as change tracking, ACL checking, and allocation/deallocation of XML documents and private data.

Pacemaker assumes that every XML node is part of a document and has private data allocated. If libxml2 APIs are used directly instead of the wrapper functions, Pacemaker may crash with a segmentation fault, or change tracking and ACL checking may be incorrectly disabled.

XPaths

Separting XPath element names with `/` (specifying each level in the hierarchy explicitly) is more efficient than `//` (allowing intermediate levels to be omitted), so it should be used whenever practical.

2.5.17 Makefiles

Pacemaker uses automake for building, so the Makefile.am in each directory should be edited rather than Makefile.in or Makefile, which are automatically generated.

- Public API headers are installed (by adding them to a `HEADERS` variable in `Makefile.am`), but internal API headers are not (by adding them to `noinst_HEADERS`).

2.5.18 vim Settings

Developers who use `vim` to edit source code can add the following settings to their `~/.vimrc` file to follow Pacemaker C coding guidelines:

```
" follow Pacemaker coding guidelines when editing C source code files
filetype plugin indent on
au FileType c   setlocal expandtab tabstop=4 softtabstop=4 shiftwidth=4 textwidth=80
autocmd BufNewFile,BufRead *.h set filetype=c
let c_space_errors = 1
```

2.6 Coding Particular Pacemaker Components

The Pacemaker code can be intricate and difficult to follow. This chapter has some high-level descriptions of how individual components work.

2.6.1 Controller

`pacemaker-controld` is the Pacemaker daemon that utilizes the other daemons to orchestrate actions that need to be taken in the cluster. It receives CIB change notifications from the CIB manager, passes the new CIB to the scheduler to determine whether anything needs to be done, uses the executor and fencer to execute any actions required, and sets failure counts (among other things) via the attribute manager.

As might be expected, it has the most code of any of the daemons.

Join sequence

Most daemons track their cluster peers using Corosync's membership and *CPG* only. The controller additionally requires peers to *join*, which ensures they are ready to be assigned tasks. Joining proceeds through a series of phases referred to as the *join sequence* or *join process*.

A node's current join phase is tracked by the `join` member of `crm_node_t` (used in the peer cache). It is an `enum crm_join_phase` that (ideally) progresses from the DC's point of view as follows:

- The node initially starts at `crm_join_none`
- The DC sends the node a *join offer* (`CRM_OP_JOIN_OFFER`), and the node proceeds to `crm_join_welcomed`. This can happen in three ways:
  - The joining node will send a *join announce* (`CRM_OP_JOIN_ANNOUNCE`) at its controller startup, and the DC will reply to that with a join offer.
  - When the DC's peer status callback notices that the node has joined the messaging layer, it registers `I_NODE_JOIN` (which leads to `A_DC_JOIN_OFFER_ONE` -> `do_dc_join_offer_one()` -> `join_make_offer()`).
  - After certain events (notably a new DC being elected), the DC will send all nodes join offers (via A_DC_JOIN_OFFER_ALL -> `do_dc_join_offer_all()`).

  These can overlap. The DC can send a join offer and the node can send a join announce at nearly the same time, so the node responds to the original join offer while the DC responds to the join announce with a new join offer. The situation resolves itself after looping a bit.
- The node responds to join offers with a *join request* (`CRM_OP_JOIN_REQUEST`, via `do_cl_join_offer_respond()` and `join_query_callback()`). When the DC receives the request, the node proceeds to `crm_join_integrated` (via `do_dc_join_filter_offer()`).
- As each node is integrated, the current best CIB is sync'ed to each integrated node via `do_dc_join_finalize()`. As each integrated node's CIB sync succeeds, the DC acks the node's join request (`CRM_OP_JOIN_ACKNAK`) and the node proceeds to `crm_join_finalized` (via `finalize_sync_callback()` + `finalize_join_for()`).
- Each node confirms the finalization ack (`CRM_OP_JOIN_CONFIRM` via `do_cl_join_finalize_respond()`), including its current resource operation history (via `controld_query_executor_state()`). Once the DC receives this confirmation, the node proceeds to `crm_join_confirmed` via `do_dc_join_ack()`.

Once all nodes are confirmed, the DC calls `do_dc_join_final()`, which checks for quorum and responds appropriately.

When peers are lost, their join phase is reset to none (in various places).

`crm_update_peer_join()` updates a node's join phase.

The DC increments the global `current_join_id` for each joining round, and rejects any (older) replies that don't match.

2.6.2 Fencer

`pacemaker-fenced` is the Pacemaker daemon that handles fencing requests. In the broadest terms, fencing works like this:

1. The initiator (an external program such as `stonith_admin`, or the cluster itself via the controller) asks the local fencer, "Hey, could you please fence this node?"
2. The local fencer asks all the fencers in the cluster (including itself), "Hey, what fencing devices do you have access to that can fence this node?"
3. Each fencer in the cluster replies with a list of available devices that it knows about.
4. Once the original fencer gets all the replies, it asks the most appropriate fencer peer to actually carry out the fencing. It may send out more than one such request if the target node must be fenced with multiple devices.
5. The chosen fencer(s) call the appropriate fencing resource agent(s) to do the fencing, then reply to the original fencer with the result.
6. The original fencer broadcasts the result to all fencers.
7. Each fencer sends the result to each of its local clients (including, at some point, the initiator).

A more detailed description follows.

Initiating a fencing request

A fencing request can be initiated by the cluster or externally, using the libstonithd API.

- The cluster always initiates fencing via `daemons/controld/controld_fencing.c:te_fence_node()` (which calls the `fence()` API method). This occurs when a transition graph synapse contains a `CRM_OP_FENCE` XML operation.
- The main external clients are `stonith_admin` and `cts-fence-helper`. The `DLM` project also uses Pacemaker for fencing.

Highlights of the fencing API:

- `stonith_api_new()` creates and returns a new `stonith_t` object, whose `cmds` member has methods for connect, disconnect, fence, etc.
- the `fence()` method creates and sends a `STONITH_OP_FENCE XML` request with the desired action and target node. Callers do not have to choose or even have any knowledge about particular fencing devices.

Fencing queries

The function calls for a fencing request go something like this:

The local fencer receives the client's request via an *IPC* or messaging layer callback, which calls

- `stonith_command()`, which (for requests) calls
  - `handle_request()`, which (for `STONITH_OP_FENCE` from a client) calls
    - `initiate_remote_stonith_op()`, which creates a `STONITH_OP_QUERY` XML request with the target, desired action, timeout, etc. then broadcasts the operation to the cluster group (i.e. all fencer instances) and starts a timer. The query is broadcast because (1) location constraints might prevent the local node from accessing the stonith device directly, and (2) even if the local node does have direct access, another node might be preferred to carry out the fencing.

Each fencer receives the original fencer's `STONITH_OP_QUERY` broadcast request via IPC or messaging layer callback, which calls:

- `stonith_command()`, which (for requests) calls
  - `handle_request()`, which (for `STONITH_OP_QUERY` from a peer) calls
  - `stonith_query()`, which calls
    - `get_capable_devices()` with `stonith_query_capable_device_cb()` to add device information to an XML reply and send it. (A message is considered a reply if it contains `T_STONITH_REPLY`, which is only set by fencer peers, not clients.)

The original fencer receives all peers' `STONITH_OP_QUERY` replies via IPC or messaging layer callback, which calls:

- `stonith_command()`, which (for replies) calls
  - `handle_reply()` which (for `STONITH_OP_QUERY`) calls
    - `process_remote_stonith_query()`, which allocates a new query result structure, parses device information into it, and adds it to the operation object. It increments the number of replies received for this operation, and compares it against the expected number of replies (i.e. the number of active peers), and if this is the last expected reply, calls
      - `request_peer_fencing()`, which calculates the timeout and sends `STONITH_OP_FENCE` request(s) to carry out the fencing. If the target node has a fencing "topology" (which allows specifications such as "this node can be fenced either with device A, or devices B and C in combination"), it will choose the device(s), and send out as many requests as needed. If it chooses a device, it will choose the peer; a peer is preferred if it has "verified" access to the desired device, meaning that it has the device "running" on it and thus has a monitor operation ensuring reachability.

Fencing operations

Each `STONITH_OP_FENCE` request goes something like this:

The chosen peer fencer receives the `STONITH_OP_FENCE` request via *IPC* or messaging layer callback, which calls:

- `stonith_command()`, which (for requests) calls
  - `handle_request()`, which (for `STONITH_OP_FENCE` from a peer) calls
    - `stonith_fence()`, which calls
      - `schedule_stonith_command()` (using supplied device if `F_STONITH_DEVICE` was set, otherwise the highest-priority capable device obtained via `get_capable_devices()` with `stonith_fence_get_devices_cb()`), which adds the operation to the device's pending operations list and triggers processing.

The chosen peer fencer's mainloop is triggered and calls

- `stonith_device_dispatch()`, which calls
  - `stonith_device_execute()`, which pops off the next item from the device's pending operations list. If acting as the (internally implemented) watchdog agent, it panics the node, otherwise it calls
    - `stonith_action_create()` and `stonith_action_execute_async()` to call the fencing agent.

The chosen peer fencer's mainloop is triggered again once the fencing agent returns, and calls

- `stonith_action_async_done()` which adds the results to an action object then calls its
  - done callback (`st_child_done()`), which calls `schedule_stonith_command()` for a new device if there are further required actions to execute or if the original action failed, then builds and sends an XML reply to the original fencer (via `send_async_reply()`), then checks whether any pending actions are the same as the one just executed and merges them if so.

Fencing replies

The original fencer receives the `STONITH_OP_FENCE` reply via *IPC* or messaging layer callback, which calls:

- `stonith_command()`, which (for replies) calls
  - `handle_reply()`, which calls
    - `fenced_process_fencing_reply()`, which calls either `request_peer_fencing()` (to retry a failed operation, or try the next device in a topology if appropriate, which issues a new `STONITH_OP_FENCE` request, proceeding as before) or `finalize_op()` (if the operation is definitively failed or successful).
      - `finalize_op()` broadcasts the result to all peers.

Finally, all peers receive the broadcast result and call

- `finalize_op()`, which sends the result to all local clients.

Fencing History

The fencer keeps a running history of all fencing operations. The bulk of the relevant code is in *fenced_history.c* and ensures the history is synchronized across all nodes even if a node leaves and rejoins the cluster.

In libstonithd, this information is represented by *stonith_history_t* and is queryable by the *stonith_api_operations_t:history()* method. *crm_mon* and *stonith_admin* use this API to display the history.

2.6.3 Scheduler

`pacemaker-schedulerd` is the Pacemaker daemon that runs the Pacemaker scheduler for the controller, but "the scheduler" in general refers to related library code in `libpe_status` and `libpe_rules` (`lib/pengine/*.c`), and some of `libpacemaker` (`lib/pacemaker/pcmk_sched_*.c`).

The purpose of the scheduler is to take a CIB as input and generate a transition graph (list of actions that need to be taken) as output.

The controller invokes the scheduler by contacting the scheduler daemon via local *IPC*. Tools such as `crm_simulate`, `crm_mon`, and `crm_resource` can also invoke the scheduler, but do so by calling the library functions directly. This allows them to run using a `CIB_file` without the cluster needing to be active.

The main entry point for the scheduler code is `lib/pacemaker/pcmk_scheduler.c:pcmk__schedule_actions()`. It sets defaults and calls a series of functions for the scheduling. Some key steps:

- `unpack_cib()` parses most of the CIB XML into data structures, and determines the current cluster status.
- `apply_node_criteria()` applies factors that make resources prefer certain nodes, such as shutdown locks, location constraints, and stickiness.
- `pcmk__create_internal_constraints()` creates internal constraints, such as the implicit ordering for group members, or start actions being implicitly ordered before promote actions.
- `pcmk__handle_rsc_config_changes()` processes resource history entries in the CIB status section. This is used to decide whether certain actions need to be done, such as deleting orphan resources, forcing a restart when a resource definition changes, etc.
- `assign_resources()` *assigns* resources to nodes.
- `schedule_resource_actions()` schedules resource-specific actions (which might or might not end up in the final graph).
- `pcmk__apply_orderings()` processes ordering constraints in order to modify action attributes such as optional or required.
- `pcmk__create_graph()` creates the transition graph.

Challenges

Working with the scheduler is difficult. Challenges include:

- It is far too much code to keep more than a small portion in your head at one time.
- Small changes can have large (and unexpected) effects. This is why we have a large number of regression tests (`cts/cts-scheduler`), which should be run after making code changes.
- It produces an insane amount of log messages at debug and trace levels. You can put resource ID(s) in the `PCMK_trace_tags` environment variable to enable trace-level messages only when related to specific resources.
- Different parts of the main `pcmk_scheduler_t` structure are finalized at different points in the scheduling process, so you have to keep in mind whether information you're using at one point of the code can possibly change later. For example, data unpacked from the CIB can safely be used anytime after `unpack_cib()`, but actions may become optional or required anytime before `pcmk__create_graph()`. There's no easy way to deal with this.

The Scheduler Object

The main data object for the scheduler is `pcmk_scheduler_t`, which contains all information needed about nodes, resources, constraints, etc., both as the raw CIB XML and parsed into more usable data structures, plus the resulting transition graph XML. The variable name is usually `scheduler`.

Resources

`pcmk_resource_t` is the data object representing cluster resources. It has a couple of public members for backward compatibility reasons, but most of the implementation is in the internal `pcmk__resource_private_t` type.

A resource has a variant: *primitive*, group, clone, or *bundle*.

The private resource object has members for two sets of methods, `pcmk__rsc_methods_t` from `libcrmcommon`, and `pcmk__assignment_methods_t` whose implementation is internal to `libpacemaker`. The actual functions vary by variant.

The resource methods have basic capabilities such as unpacking the resource XML, and determining the current or planned location of the resource.

The *assignment* methods have more obscure capabilities needed for scheduling, such as processing location and ordering constraints. For example, `pcmk__create_internal_constraints()` simply calls the `internal_constraints()` method for each top-level resource in the cluster.

Nodes

Assignment of resources to nodes is done by choosing the node with the highest *score* for a given resource. The scheduler does a bunch of processing to generate the scores, then the actual assignment is straightforward.

The scheduler node implementation is a little confusing.

`pcmk_node_t` (`struct pcmk__scored_node`) is the primary object used.

It contains two sub-structs, `pcmk__node_private_t *priv` (which is internal) and `struct pcmk__node_details *details` (which is public for backward compatibility reasons), that contain all node information that is independent of resource assignment (the node name, etc.).

It contains one other (internal) sub-struct, `struct pcmk__node_assignment *assign`, which contains information particular to a specific resource being assigned.

Node lists are frequently used. For example, `pcmk_scheduler_t` has a `nodes` member which is a list of all nodes in the cluster, and the internal resource object has an `active_nodes` member which is a list of all nodes on which the resource is (or might be) active.

Only the scheduler's `nodes` list has the full, original node instances. All other node lists have shallow copies created by `pe__copy_node()`, which share `details` and `priv` from the main list (but can differ in their `assign` member).

Actions

`pcmk_action_t` is the data object representing actions that might need to be taken. These could be resource actions, cluster-wide actions such as fencing a node, or "pseudo-actions" which are abstractions used as convenient points for ordering other actions against.

Its (internal) implementation has a `flags` member which is a bitmask of `enum pcmk__action_flags`. The most important of these are `pcmk__action_runnable` (if not set, the action is "blocked" and cannot be added to the transition graph) and `pcmk__action_optional` (actions with this set will not be added to the transition graph; actions often start out as optional, and may become required later).

Colocations

`pcmk__colocation_t` is the data object representing colocations.

Colocation constraints come into play in these parts of the scheduler code:

- When sorting resources for *assignment*, so resources with highest node *score* are assigned first (see `cmp_resources()`)
- When updating node scores for resource assigment or promotion priority

- When assigning resources, so any resources to be colocated with can be assigned first, and so colocations affect where the resource is assigned
- When choosing roles for promotable clone instances, so colocations involving a specific role can affect which instances are promoted

The resource assignment functions have several methods related to colocations:

- `apply_coloc_score()`: This applies a colocation's score to either the dependent's allowed node scores (if called while resources are being assigned) or the dependent's priority (if called while choosing promotable instance roles). It can behave differently depending on whether it is being called as the *primary's* method or as the *dependent's* method.
- `add_colocated_node_scores()`: This updates a table of nodes for a given colocation attribute and score. It goes through colocations involving a given resource, and updates the scores of the nodes in the table with the best scores of nodes that match up according to the colocation criteria.
- `colocated_resources()`: This generates a list of all resources involved in mandatory colocations (directly or indirectly via colocation chains) with a given resource.

Action Relations

Ordering constraints are simple in concept, but they are one of the most important, powerful, and difficult to follow aspects of the scheduler code.

`pcmk__action_relation_t` is the data object representing an ordering, better thought of as a relationship between two actions, since the relation can be more complex than just "this one runs after that one".

For a relation "A then B", the code generally refers to A as "first" or "before", and B as "then" or "after".

Much of the power comes from `enum pcmk__action_relation_flags`, which are flags that determine how a relation behaves. There are many obscure flags with big effects. A few examples:

- `pcmk__ar_none` means the relation is disabled and will be ignored. The value is 0, meaning no flags set, so it must be compared with equality rather than `pcmk_is_set()`.
- `pcmk__ar_ordered` without any other flags set means the relation does not make either action required, so it applies only if they both become required for other reasons.
- `pcmk__ar_then_implies_first` means that if action B becomes required for any reason, then action A will become required as well.

Adding a New Scheduler Regression Test

1. Choose a test name.
2. Copy the uncompressed input CIB to cts/scheduler/xml/TESTNAME.xml. It's helpful to add an XML comment at the top describing the essential features of the test (which configuration and status scenarios are being tested).
3. Edit `cts/cts-scheduler.in` and add the test name and description to the `TESTS` array.
4. Run `cts/cts-scheduler --update --run TESTNAME` to generate the expected transition graph, scores, etc. Look over the generated files to make sure they are as expected.
5. Commit your changes.

2.7 C Development Helpers

2.7.1 Refactoring

Pacemaker uses an optional tool called coccinelle to do automatic refactoring. coccinelle is a very complicated tool that can be difficult to understand, and the existing documentation makes it pretty tough to get started. Much of the documentation is either aimed at kernel developers or takes the form of grammars.

However, it can apply very complex transformations across an entire source tree. This is useful for tasks like code refactoring, changing APIs (number or type of arguments, etc.), catching functions that should not be called, and changing existing patterns.

coccinelle is driven by input scripts called semantic patches written in its own language. These scripts bear a passing resemblance to source code patches and tell coccinelle how to match and modify a piece of source code. They are stored in `devel/coccinelle` and each script either contains a single source transformation or several related transformations. In general, we try to keep these as simple as possible.

In Pacemaker development, we use a couple targets in `devel/Makefile.am` to control coccinelle. The `cocci` target tries to apply each script to every Pacemaker source file, printing out any changes it would make to the console. The `cocci-inplace` target does the same but also makes those changes to the source files. A variety of warnings might also be printed. If you aren't working on a new script, these can usually be ignored.

If you are working on a new coccinelle script, it can be useful (and faster) to skip everything else and only run the new script. The `COCCI_FILES` variable can be used for this:

```
$ make -C devel COCCI_FILES=coccinelle/new-file.cocci cocci
```

This variable is also used for preventing some coccinelle scripts in the Pacemaker source tree from running. Some scripts are disabled because they are not currently fully working or because they are there as templates. When adding a new script, remember to add it to this variable if it should always be run.

One complication when writing coccinelle scripts is that certain Pacemaker source files may not use private functions (those whose name starts with `pcmk__`). Handling this requires work in both the Makefile and in the coccinelle scripts.

The Makefile deals with this by maintaining two lists of source files: those that may use private functions and those that may not. For those that may, a special argument (`-D internal`) is added to the coccinelle command line. This creates a virtual dependency named `internal`.

In the coccinelle scripts, those transformations that modify source code to use a private function also have a dependency on `internal`. If that dependency was given on the command line, the transformation will be run. Otherwise, it will be skipped.

This means that not all instances of an older style of code will be changed after running a given transformation. Some developer intervention is still necessary to know whether a source code block should have been changed or not.

Probably the easiest way to learn how to use coccinelle is by following other people's scripts. In addition to the ones in the Pacemaker source directory, there's several others on the coccinelle website.

2.7.2 Sanitizers

gcc supports a variety of run-time checks called sanitizers. These can be used to catch programming errors with memory, race conditions, various undefined behavior conditions, and more. Because these are run-time checks, they should only be used during development and not in compiled packages or production code.

Certain sanitizers cannot be combined with others because their run-time checks cause interfere. Instead of trying to figure out which combinations work, it is simplest to just enable one at a time.

Each supported sanitizer requires an installed libray. In addition to just enabling the sanitizer, their use can be configured with environment variables. For example:

```
$ ASAN_OPTIONS=verbosity=1:replace_str=true crm_mon -1R
```

Pacemaker supports the following subset of gcc's sanitizers:

Sanitizer	Configure Option	Library	Environment Variable
Address	–with-sanitizers=asan	libasan	ASAN_OPTIONS
Threads	–with-sanitizers=tsan	libtsan	TSAN_OPTIONS
Undefined behavior	–with-sanitizers=ubsan	libubsan	UBSAN_OPTIONS

The undefined behavior sanitizer further supports suboptions that need to be given as CFLAGS when configuring pacemaker:

```
$ CFLAGS=-fsanitize=integer-divide-by-zero ./configure --with-sanitizers=ubsan
```

For more information, see the gcc documentation which also provides links to more information on each sanitizer.

2.7.3 Unit Testing

Where possible, changes to the C side of Pacemaker should be accompanied by unit tests. Much of Pacemaker cannot effectively be unit tested (and there are other testing systems used for those parts), but the `lib` subdirectory is pretty easy to write tests for.

Pacemaker uses the cmocka unit testing framework which looks a lot like other unit testing frameworks for C and should be fairly familiar. In addition to regular unit tests, cmocka also gives us the ability to use mock functions for unit testing functions that would otherwise be difficult to test.

Organization

Pay close attention to the organization and naming of test cases to ensure the unit tests continue to work as they should.

Tests are spread throughout the source tree, alongside the source code they test. For instance, all the tests for the source code in `lib/common/` are in the `lib/common/tests` directory. If there is no `tests` subdirectory, there are no tests for that library yet.

Under that directory, there is a `Makefile.am` and additional subdirectories. Each subdirectory contains the tests for a single library source file. For instance, all the tests for `lib/common/strings.c` are in the `lib/common/tests/strings` directory. Note that the test subdirectory does not have a `.c` suffix. If there is no test subdirectory, there are no tests for that file yet.

Finally, under that directory, there is a `Makefile.am` and then various source files. Each of these source files tests the single function that it is named after. For instance, `lib/common/tests/strings/pcmk__btoa_test.c` tests the `pcmk__btoa()` function in `lib/common/strings.c`. If there is no test source file, there are no tests for that function yet.

The `_test` suffix on the test source file is important. All tests have this suffix, which means all the compiled test cases will also end with this suffix. That lets us ignore all the compiled tests with a single line in `.gitignore`:

```
/lib/*/tests/*/*_test
```

Adding a test

Testing a new function in an already testable source file

Follow these steps if you want to test a function in a source file where there are already other tested functions. For the purposes of this example, we will add a test for the `pcmk__scan_port()` function in `lib/common/strings.c`. As you can see, there are already tests for other functions in this same file in the `lib/common/tests/strings` directory.

- cd into `lib/common/tests/strings`
- Add the new file to the `check_PROGRAMS` variable in `Makefile.am`, making it something like this:

```
check_PROGRAMS = \
       pcmk__add_word_test             \
       pcmk__btoa_test                 \
       pcmk__scan_port_test
```

- Create a new `pcmk__scan_port_test.c` file, copying the copyright and include boilerplate from another file in the same directory.
- Continue with the steps in *Writing the test*.

Testing a function in a source file without tests

Follow these steps if you want to test a function in a source file where there are not already other tested functions, but there are tests for other files in the same library. For the purposes of this example, we will add a test for the `pcmk_acl_required()` function in `lib/common/acls.c`. At the time of this documentation being written, no tests existed for that source file, so there is no `lib/common/tests/acls` directory.

- Add to `AC_CONFIG_FILES` in the top-level `configure.ac` file so the build process knows to use directory we're about to create. That variable would now look something like:

```
dnl Other files we output
AC_CONFIG_FILES(Makefile                                        \
                ...
                lib/common/tests/Makefile                       \
                lib/common/tests/acls/Makefile                  \
                lib/common/tests/agents/Makefile                \
                ...
)
```

- cd into `lib/common/tests`
- Add to the `SUBDIRS` variable in `Makefile.am`, making it something like:

```
SUBDIRS = agents acls cmdline flags operations strings utils xpath results
```

- Create a new `acls` directory, copying the `Makefile.am` from some other directory. At this time, each `Makefile.am` is largely boilerplate with very little that needs to change from directory to directory.
- cd into `acls`
- Get rid of any existing values for `check_PROGRAMS` and set it to `pcmk_acl_required_test` like so:

```
check_PROGRAMS = pcmk_acl_required_test
```

- Double check that the following includes are at the top of `Makefile.am`:

```
include $(top_srcdir)/mk/common.mk
include $(top_srcdir)/mk/tap.mk
include $(top_srcdir)/mk/unittest.mk
```

- If necessary, settings from those includes can be overridden like so:

```
AM_TESTS_ENVIRONMENT += PCMK_CTS_CLI_DIR=$(top_srcdir)/cts/cli
AM_CPPFLAGS += -I$(top_srcdir)
LDADD += $(top_builddir)/lib/pengine/libpe_status_test.la
```

- Follow the steps in *Testing a new function in an already testable source file* to create the new `pcmk_acl_required_test.c` file.

Testing a function in a library without tests

Adding a test case for a function in a library that doesn't have any test cases to begin with is only slightly more complicated. In general, the steps are the same as for the previous section, except with an additional layer of directory creation.

For the purposes of this example, we will add a test case for the `lrmd_send_resource_alert()` function in `lib/lrmd/lrmd_alerts.c`. Note that this may not be a very good function or even library to write actual unit tests for.

- Add to `AC_CONFIG_FILES` in the top-level `configure.ac` file so the build process knows to use directory we're about to create. That variable would now look something like:

```
dnl Other files we output
AC_CONFIG_FILES(Makefile                                        \
                ...
                lib/lrmd/Makefile                               \
                lib/lrmd/tests/Makefile                         \
                lib/services/Makefile                           \
                ...
)
```

- cd into `lib/lrmd`
- Create a `SUBDIRS` variable in `Makefile.am` if it doesn't already exist. Most libraries should not have this variable already.

```
SUBDIRS = tests
```

- Create a new `tests` directory and add a `Makefile.am` with the following contents:

```
SUBDIRS = lrmd_alerts
```

- Follow the steps in *Testing a function in a source file without tests* to create the rest of the new directory structure.
- Follow the steps in *Testing a new function in an already testable source file* to create the new `lrmd_send_resource_alert_test.c` file.

Adding to an existing test case

If all you need to do is add additional test cases to an existing file, none of the above work is necessary. All you need to do is find the test source file with the name matching your function and add to it and then follow the instructions in *Writing the test*.

Writing the test

A test case file contains a fair amount of boilerplate. For this reason, it's usually easiest to just copy an existing file and adapt it to your needs. However, here's the basic structure:

```
/*
 * Copyright 2021 the Pacemaker project contributors
 *
 * The version control history for this file may have further details.
 *
 * This source code is licensed under the GNU Lesser General Public License
 * version 2.1 or later (LGPLv2.1+) WITHOUT ANY WARRANTY.
 */

#include <crm_internal.h>

#include <crm/common/unittest_internal.h>

/* Put your test-specific includes here */

/* Put your test functions here */

PCMK__UNIT_TEST(NULL, NULL,
                /* Register your test functions here */)
```

Each test-specific function should test one aspect of the library function, though it can include many assertions if there are many ways of testing that one aspect. For instance, there might be multiple ways of testing regular expression matching:

```
static void
regex(void **state) {
    const char *s1 = "abcd";
    const char *s2 = "ABCD";

    assert_true(pcmk__strcmp(NULL, "a..d", pcmk__str_regex) < 0);
    assert_true(pcmk__strcmp(s1, NULL, pcmk__str_regex) > 0);
    assert_int_equal(pcmk__strcmp(s1, "a..d", pcmk__str_regex), 0);
}
```

Each test-specific function must also be registered or it will not be called. This is done with `cmocka_unit_test()` in the `PCMK__UNIT_TEST` macro:

```
PCMK__UNIT_TEST(NULL, NULL,
                cmocka_unit_test(regex))
```

Most unit tests do not require a setup and teardown function to be executed around the entire group of tests. On occassion, this may be necessary. Simply pass those functions in as the first two parameters to `PCMK__UNIT_TEST` instead of using NULL.

Assertions

In addition to the assertions provided by cmocka, `unittest_internal.h` also provides `pcmk__assert_asserts`. This macro takes an expression and verifies that the expression aborts due to a failed call to `pcmk__assert()` or some other similar function. It can be used like so:

```
static void
null_input_variables(void **state)
{
    long long start, end;

    pcmk__assert_asserts(pcmk__parse_ll_range("1234", NULL, &end));
    pcmk__assert_asserts(pcmk__parse_ll_range("1234", &start, NULL));
}
```

Here, `pcmk__parse_ll_range` expects non-NULL for its second and third arguments. If one of those arguments is NULL, `pcmk__assert()` will fail and the program will abort. `pcmk__assert_asserts` checks that the code would abort and the test passes. If the code does not abort, the test fails.

Running

If you had to create any new files or directories, you will first need to run `./configure` from the top level of the source directory. This will regenerate the Makefiles throughout the tree. If you skip this step, your changes will be skipped and you'll be left wondering why the output doesn't match what you expected.

To run the tests, simply run `make check` after previously building the source with `make`. The test cases in each directory will be built and then run. This should not take long. If all the tests succeed, you will be back at the prompt. Scrolling back through the history, you should see lines like the following:

```
PASS: pcmk__strcmp_test 1 - same_pointer
PASS: pcmk__strcmp_test 2 - one_is_null
PASS: pcmk__strcmp_test 3 - case_matters
PASS: pcmk__strcmp_test 4 - case_insensitive
PASS: pcmk__strcmp_test 5 - regex
============================================================================
Testsuite summary for pacemaker 2.1.0
============================================================================
# TOTAL: 33
# PASS:  33
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
============================================================================
make[7]: Leaving directory '/home/clumens/src/pacemaker/lib/common/tests/strings'
```

The testing process will quit on the first failed test, and you will see lines like these:

```
PASS: pcmk__scan_double_test 3 - trailing_chars
FAIL: pcmk__scan_double_test 4 - typical_case
PASS: pcmk__scan_double_test 5 - double_overflow
PASS: pcmk__scan_double_test 6 - double_underflow
ERROR: pcmk__scan_double_test - exited with status 1
PASS: pcmk__starts_with_test 1 - bad_input
============================================================================
```

(continues on next page)

(continued from previous page)

```
Testsuite summary for pacemaker 2.1.0
============================================================================
# TOTAL: 56
# PASS:  54
# SKIP:  0
# XFAIL: 0
# FAIL:  1
# XPASS: 0
# ERROR: 1
============================================================================
See lib/common/tests/strings/test-suite.log
Please report to users@clusterlabs.org
============================================================================
make[7]: *** [Makefile:1218: test-suite.log] Error 1
make[7]: Leaving directory '/home/clumens/src/pacemaker/lib/common/tests/strings'
```

The failure is in `lib/common/tests/strings/test-suite.log`:

```
ERROR: pcmk__scan_double_test
=============================

1..6
ok 1 - empty_input_string
PASS: pcmk__scan_double_test 1 - empty_input_string
ok 2 - bad_input_string
PASS: pcmk__scan_double_test 2 - bad_input_string
ok 3 - trailing_chars
PASS: pcmk__scan_double_test 3 - trailing_chars
not ok 4 - typical_case
FAIL: pcmk__scan_double_test 4 - typical_case
# 0.000000 != 3.000000
# pcmk__scan_double_test.c:80: error: Failure!
ok 5 - double_overflow
PASS: pcmk__scan_double_test 5 - double_overflow
ok 6 - double_underflow
PASS: pcmk__scan_double_test 6 - double_underflow
# not ok - tests
ERROR: pcmk__scan_double_test - exited with status 1
```

At this point, you need to determine whether your test case is incorrect or whether the code being tested is incorrect. Fix whichever is wrong and continue.

2.7.4 Fuzz Testing

Pacemaker is integrated with the OSS-Fuzz project. OSS-Fuzz calls selected Pacemaker APIs with random argument values to catch edge cases that might be missed by other forms of testing.

The OSS-Fuzz project has a contact address for Pacemaker in projects/pacemaker/project.yaml that will receive bug reports. The address must have been used to commit to Pacemaker, and should be tied to a Google account.

Open reports that aren't security-related can be seen at OSS-Fuzz testcases.

Fuzzers

Each fuzz-tested library has a fuzzers subdirectory (for example, `lib/common/fuzzers`). That directory has a file for each fuzzed source file, named the same except ending in `_fuzzer.c` (for example, `lib/common/fuzzers/strings_fuzzer.c` has fuzzing for `lib/common/strings.c`). Those files are not built or distributed as part of Pacemaker but are used by OSS-Fuzz (see `projects/pacemaker/build.sh` in the OSS-Fuzz repository).

By default, fuzzing uses libFuzzer. Only Pacemaker APIs that accept any input and do not exit can be fuzzed. Ideally, fuzzed functions will not modify global state or vary code paths by anything other than the fuzzed input (such as environment variable values, date/time, etc.).

Local Fuzzing

You can use OSS-Fuzz locally to run fuzz testing or reproduce issues reported by OSS-Fuzz.

To prep a test host:

1. If podman is installed, it will conflict with Docker, so remove it first. Example for RHEL-like OSes:
   - `dnf remove runc`
1. Install and start Docker. Example for RHEL-like OSes:
   - `dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.repo`
   - `dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin`
   - `usermod -a -G docker $USER`
2. Clone the OSS-Fuzz repository:
   - `cd` to wherever you want to put it
   - `git clone https://github.com/google/oss-fuzz.git`
   - `cd oss-fuzz`
3. Specify the Pacemaker source you want to test:
   - Edit `projects/pacemaker/Dockerfile` and replace the last `git clone` with the source that you want to test. For example, if you have a branch `my-fuzzing-branch` that you've pushed to your GitHub account, you could use: `git clone -b my-fuzzing-branch --single-branch --depth 1 https://github.com/$USER/pacemaker`.

To fuzz the code:

1. Ensure Docker is running:
   - `systemctl start docker`
2. Build the necessary Docker containers:
   - `python3 infra/helper.py build_image pacemaker`
3. Build the fuzzers. Choose a sanitizer (for example, `SANITIZER=address`). There are three possible sanitizers: address, memory, and undefined. The memory sanitizer requires special preparation and is generally not used. If you are reproducing an OSS-Fuzz-reported issue, the issue will list the sanitizer that was used.
   - `python3 infra/helper.py build_fuzzers --sanitizer $SANITIZER pacemaker`

4. Ensure the build succeeded (use the same sanitizer as the previous step):
   - `python3 infra/helper.py check_build --sanitizer $SANITIZER pacemaker`
5. If you want to run fuzzing yourself, choose a fuzzer (for example, `FUZZER=iso8601_fuzzer`). Create a temporary directory for the fuzzer's outputs, then run the fuzzing command, which will fuzz for 25 seconds then time out:
   - `rm -rf /tmp/corpus >/dev/null 2>&/dev/null`
   - `mkdir /tmp/corpus`
   - `python3 infra/helper.py run_fuzzer --corpus-dir=/tmp/corpus pacemaker $FUZZER`
   - This can be repeated with different fuzzers. The `build_fuzzers` step can also be repeated with a different sanitizer, and the fuzzers tested again.
6. If you want to reproduce an OSS-Fuzz-reported issue, make a note of the fuzzer that was used (`$FUZZER` in this example) and download the provided reproducer test case file (`$TESTCASE` in this example), then run:
   - `python3 infra/helper.py reproduce pacemaker $FUZZER $TESTCASE`

For details, see the OSS-Fuzz documentation.

2.7.5 Code Coverage

Figuring out what needs unit tests written is the purpose of a code coverage tool. The Pacemaker build process uses `lcov` and special make targets to generate an HTML coverage report that can be inspected with any web browser.

To start, you'll need to install the `lcov` package which is included in most distributions. Next, reconfigure the source tree:

```
$ ./configure --with-coverage
```

Then run `make -C devel coverage`. This will do the same thing as `make check`, but will generate a bunch of intermediate files as part of the compiler's output. Essentially, the coverage tools run all the unit tests and make a note if a given line if code is executed as a part of some test program. This will include not just things run as part of the tests but anything in the setup and teardown functions as well.

Afterwards, the HTML report will be in `coverage/index.html`. You can drill down into individual source files to see exactly which lines are covered and which are not, which makes it easy to target new unit tests. Note that sometimes, it is impossible to achieve 100% coverage for a source file. For instance, how do you test a function with a return type of void that simply returns on some condition?

Note that Pacemaker's overall code coverage numbers are very low at the moment. One reason for this is the large amount of code in the `daemons` directory that will be very difficult to write unit tests for. For now, it is best to focus efforts on increasing the coverage on individual libraries.

Additionally, there is a `coverage-cts` target that does the same thing but instead of testing `make check`, it tests `cts/cts-cli`. The idea behind this target is to see what parts of our command line tools are covered by our regression tests. It is probably best to clean and rebuild the source tree when switching between these various targets.

2.7.6 Debugging

gdb

If you use `gdb` for debugging, some helper functions are defined in `devel/gdbhelpers`, which can be given to `gdb` using the `-x` option.

From within the debugger, you can then invoke the `pcmk` command that will describe the helper functions available.

2.8 Evolution of the project

This section will not generally be of interest, but may occasionally shed light on why the current code is structured the way it is when investigating some thorny issue.

2.8.1 Origin in Heartbeat project

Pacemaker can be considered as a spin-off from Heartbeat, the original comprehensive high availability suite started by Alan Robertson. Some portions of code are shared, at least on the conceptual level if not verbatim, till today, even if the effective percentage continually declines.

Before Pacemaker 2.0, Pacemaker supported Heartbeat as a cluster layer alternative to Corosync. That support was dropped for the 2.0.0 release (see commit 55ab749bf).

An archive of a 2016 checkout of the Heartbeat code base is shared as a read-only repository. Notable commits include:

- creation of Heartbeat's "new cluster resource manager," which evolved into Pacemaker
- deletion of the new CRM from Heartbeat after Pacemaker had been split off

Regarding Pacemaker's split from heartbeat, it evolved stepwise (as opposed to one-off cut), and the last step of full dependency is depicted in The Corosync Cluster Engine paper, fig. 10. This article also provides a good reference regarding wider historical context of the tangentially (and deeper in some cases) meeting components around that time.

Influence of Heartbeat on Pacemaker

On a closer look, we can identify these things in common:

- extensive use of data types and functions of GLib
- Cluster Testing System (CTS), inherited from initial implementation by Alan Robertson
- …

2.8.2 Notable Restructuring Steps in the Codebase

File renames may not appear as notable … unless one runs into complicated `git blame` and `git log` scenarios, so some more massive ones may be stated as well.

- watchdog/'sbd' functionality spin-off:
  - start separating, eb7cce2a1
  - finish separating, 5884db780
- daemons' rename for 2.0 (in chronological order)

- start of moving daemon sources from their top-level directories under new /daemons hierarchy, 318a2e003
- attrd -> pacemaker-attrd, 01563cf26
- lrmd -> pacemaker-execd, 36a00e237
- pacemaker_remoted -> pacemaker-remoted, e4f4a0d64
- crmd -> pacemaker-controld, db5536e40
- pengine -> pacemaker-schedulerd, e2fdc2bac
- stonithd -> pacemaker-fenced, 038c465e2
- cib daemon -> pacemaker-based, 50584c234

2.9 Glossary

assign In the scheduler, this refers to associating a resource with a node. Do not use *allocate* for this purpose.

bundle The collective resource type associating instances of a container with storage and networking. Do not use *container* when referring to the bundle as a whole.

cluster layer The layer of the *cluster stack* that provides membership and messaging capabilities (such as Corosync).

cluster stack The core components of a high-availability cluster: the *cluster layer* at the "bottom" of the stack, then Pacemaker, then resource agents, and then the actual services managed by the cluster at the "top" of the stack. Do not use *stack* for the cluster layer alone.

CPG Corosync Process Group. This is the messaging layer in a Corosync-based cluster. Pacemaker daemons use CPG to communicate with their counterparts on other nodes.

container This can mean either a container in the usual sense (whether as a standalone resource or as part of a bundle), or as the container resource meta-attribute (which does not necessarily reference a container in the usual sense).

dangling migration Live migration of a resource consists of a **migrate_to** action on the source node, followed by a **migrate_from** on the target node, followed by a **stop** on the source node. If the **migrate_to** and **migrate_from** have completed successfully, but the **stop** has not yet been done, the migration is considered to be *dangling*.

dependent In colocation constraints, this refers to the resource located relative to the *primary* resource. Do not use *rh* or *right-hand* for this purpose.

IPC Inter-process communication. In Pacemaker, clients send requests to daemons using libqb IPC.

message This can refer to log messages, custom messages defined for a **pcmk_output_t** object, or XML messages sent via *CPG* or *IPC*.

metadata In the context of options and resource agents, this refers to OCF-style metadata. Do not use a hyphen except when referring to the OCF-defined action name *meta-data*.

primary In colocation constraints, this refers to the resource that the *dependent* resource is located relative to. Do not use *lh* or *left-hand* for this purpose.

primitive The fundamental resource type in Pacemaker. Do not use *native* for this purpose.

score An integer value constrained between **-PCMK__SCORE__INFINITY** and **+PCMK__SCORE__INFINITY**. Certain strings (such as **PCMK__VALUE__INFINITY**) parse as particular score values. Do not use *weight* for this purpose.

self-fencing When a node is chosen to execute its own fencing. Do not use *suicide* for this purpose.

CHAPTER

THREE

INDEX

- genindex
- search

INDEX