Hawkmoth

Release 0.17.0

Jani Nikula

Dec 10, 2023

CONTENTS

Hawkmoth is a Sphinx extension to incorporate C and C++ source code comments formatted in reStructuredText into Sphinx based documentation. It uses Clang Python Bindings for parsing, and generates C and C++ domain directives for C and C++ documentation, respectively. In short, Hawkmoth is Sphinx Autodoc for C/C++.

Hawkmoth aims to be a compelling alternative for documenting C and C++ projects using Sphinx, mainly through its simplicity of design, implementation and use.

Note: The C++ support is still in early stages of development, and lacks some fundamental features such as handling namespaces and documenting C++ specific features other than classes.

Please see the Hawkmoth project GitHub page (or README.rst in the source repository) for information on how to obtain, install, and contribute to Hawkmoth, as well as how to contact the developers.

Read on for information about Hawkmoth installation details and usage; how to configure and use the extension and how to write documentation comments, with examples.

Contents:

CHAPTER

ONE

INSTALLATION

You can install the Hawkmoth Python package and most of its dependencies from PyPI with:

```
pip install hawkmoth
```

However, you'll also need to install Clang and Clang Python Bindings through your distro's package manager; they are not available via PyPI. This is typically the biggest hurdle in getting your Hawkmoth setup to work.

1.1 Clang Distro Install

This step is necessarily distro specific.

For example, in recent Debian and Ubuntu:

```
apt install python3-clang
```

1.2 Clang Python Bindings

There are **unofficial** Clang Python Bindings available in PyPI. They may be helpful in some scenarios, but they will not include the binary `libclang`, and the provided Python Bindings might not be compatible with the library provided in your system. It's recommended to use the bindings from the distro, but if you need to install the `clang` package from PyPI, it's recommended to use the same major varsion for both system `libclang` and Python `clang`.

If the Clang Python Bindings are unable to find `libclang`, for whatever reason, there are some tricks to try:

- Set the library path in shell:

  ```
  export LD_LIBRARY_PATH=$(llvm-config --libdir)
  ```

- Set the library path in `conf.py`:

  ```
  from clang.cindex import Config
  Config.set_library_path('/path/to/clang')
  ```

- Set the library name in `conf.py`, possibly combined with `LD_LIBRARY_PATH`:

  ```
  from clang.cindex import Config
  Config.set_library_file('libclang.so')
  ```

- Set the library name with full path in `conf.py`:

```
from clang.cindex import Config
Config.set_library_file('/path/to/clang/libclang.so')
```

1.3 Virtual Environment

If you're installing Hawkmoth in a Python virtual environment, use the `--system-site-packages` option when creating the virtual environment to make the distro Clang package available to the virtual environment. For example:

```
python3 -m venv --system-site-packages .venv
```

1.4 Read the Docs

It's possible to set up Hawkmoth based documentation on Read the Docs (RTD). Use the `.readthedocs.yaml` configuration file to install system `libclang` and specify a Python `requirements.txt` file:

```
build:
  os: ubuntu-22.04
  tools:
    python: "3.11"
  apt_packages:
    - libclang-14-dev

python:
  install:
    - requirements: requirements.txt
```

In the `requirements.txt` file, specify the dependencies:

```
clang==14.0.6
hawkmoth==0.14.0
```

To ensure the system `libclang` and Python `clang` compatibility, it's recommended to specify matching major versions. RTD also recommends pinning all the versions to avoid unexpected build errors.

If the Clang Python Bindings fail to find `libclang` automatically, try adding this snippet to your `conf.py`:

```
from hawkmoth.util import readthedocs

readthedocs.clang_setup()
```

This will try to find `libclang` on RTD, and configure Clang Python Bindings to use it.

CHAPTER

TWO

AUTODOC EXTENSION

Hawkmoth provides a Sphinx extension that adds *new directives* to the Sphinx C and C++ domains to incorporate formatted C and C++ source code comments into a document. Hawkmoth is Sphinx `sphinx.ext.autodoc` for C/C++.

For this to work, the documentation comments must of course be written in correct reStructuredText. See *documentation comment syntax* for details.

Hawkmoth itself is *extensible*, and ships with some *built-in extensions*.

2.1 Usage

Add `hawkmoth` to `extensions` in `conf.py`. Note that depending on the packaging and installation directory, this may require adjusting the `PYTHONPATH`.

For example:

```
extensions.append('hawkmoth')
```

2.2 Configuration

The extension has a few configuration options that can be set in `conf.py`.

See also additional configuration options in the *built-in extensions*.

hawkmoth_root: `str`

Path to the root of the source files. Defaults to the configuration directory, i.e. the directory containing `conf.py`.

To use paths relative to the configuration directory, use `os.path.abspath()`, for example:

```
import os
hawkmoth_root = os.path.abspath('my/sources/dir')
```

hawkmoth_transform_default: `str`

The default transform parameter to be passed to the *hawkmoth-process-docstring* event. It can be overriden with the `transform` option of the *directives*. Defaults to `None`.

hawkmoth_clang: `list`

A list of arguments to pass to `clang` while parsing the source, typically to add directories to include file search path, or to define macros for conditional compilation. No arguments are passed by default.

Example:

```
hawkmoth_clang = ['-I/path/to/include', '-DHAWKMOTH']
```

Hawkmoth provides a convenience helper for querying the include path from the compiler, and providing them as `-I` options:

```
from hawkmoth.util import compiler

hawkmoth_clang = compiler.get_include_args()
```

You can also pass in the compiler to use, for example `get_include_args('gcc')`.

hawkmoth_source_uri: *str*

A template URI to source code. If set, add links to externally hosted source code for each documented symbol, similar to the Sphinx linkcode extension. Defaults to `None`.

The template URI will be formatted using `str.format()`, with the following replacement fields:

{source}
: Path to source file relative to *hawkmoth_root*.

{line}
: Line number in source file.

Example:

```
hawkmoth_source_uri = 'https://example.org/src/{source}#L{line}'
```

cautodoc_root: *str*

Equivalent to *hawkmoth_root*.

> **Warning:** The `cautodoc_root` option has been deprecated in favour of the *hawkmoth_root* option and will be removed in the future.

cautodoc_clang: *str*

Equivalent to *hawkmoth_clang*.

> **Warning:** The `cautodoc_clang` option has been deprecated in favour of the *hawkmoth_clang* option and will be removed in the future.

CHAPTER

THREE

DIRECTIVES

Hawkmoth provides several new directives for incorporating *documentation comments* from C/C++ source files into the reStructuredText document. There are three main types of directives, for incorporating documentation from entire files, for single objects, and for composite objects optionally with members.

3.1 Source Files

The *c:autodoc* and *cpp:autodoc* directives simply include all the documentation comments from any number of files. This is the most basic and quickest way to generate documentation, but offers no control over what gets included.

.. c:autodoc:: filename-pattern [...]

.. cpp:autodoc:: filename-pattern [...]

Incorporate documentation comments from the files specified by the space separated list of filename patterns given as arguments. The patterns are interpreted relative to the *hawkmoth_root* configuration option.

:transform: (text)

Override *hawkmoth_transform_default* for the `transform` parameter value of the *hawkmoth-process-docstring* event.

See also *hawkmoth.ext.transformations*.

:clang: (text)

The `clang` option extends the *hawkmoth_clang* configuration option.

For example:

```
.. c:autodoc:: interface.h

.. c:autodoc:: api/*.[ch] interface.h
   :clang: -DHAWKMOTH
```

3.2 Variables, Types, Macros, and Functions

The *c:autovar*, *c:autotype*, *c:automacro*, and *c:autofunction* directives and their C++ domain counterparts incorporate the documentation comment for the specified object in the specified file.

The directives support all the same directive options as *c:autodoc* and *cpp:autodoc*, adding the `file` option.

.. c:autovar:: name

.. cpp:autovar:: name

Incorporate the documentation comment for the variable `name`.

If `file` is specified, look up `name` there, otherwise look up `name` in all previously parsed files in the current document.

:file: (text)

The `file` option specifies the file to look up `name` in. This is required if the file has not been parsed yet, and to disambiguate if `name` is found in multiple files.

The filename is interpreted relative to the *hawkmoth_root* configuration option.

For example:

```
.. c:autovar:: example_variable
   :file: example_file.c

.. c:autovar:: another_variable
```

.. c:autotype:: name

.. cpp:autotype:: name

Same as *c:autovar* but for typedefs.

```
.. c:autotype:: example_type_t
   :file: example_file.c
```

.. c:automacro:: name

.. cpp:automacro:: name

Same as *c:autovar* but for macros, including function-like macros.

Note: The C++ Domain does not have a `cpp:macro` directive, so all macros are always in the C Domain. This affects cross-referencing them; see *Cross-Referencing C and C++ Constructs* for details.

```
.. c:automacro:: EXAMPLE_MACRO
   :file: example_file.c
```

.. c:autofunction:: name

.. cpp:autofunction:: name

Same as *c:autovar* but for functions. (Use *c:automacro* for function-like macros.)

```
.. c:autofunction:: example_function
   :file: example_file.c
```

3.3 Structures, Classes, Unions, and Enumerations

The *c:autostruct*, *c:autounion*, and *c:autoenum* directives, their C++ domain counterparts, and the *cpp:autoclass* directive incorporate the documentation comments for the specified object in the specified file, with additional control over the structure, class or union members and enumeration constants to include.

The directives support all the same directive options as *c:autodoc*, *c:autovar*, *c:autotype*, *c:automacro*, and *c:autofunction*, adding the `members` option.

.. c:autostruct:: name

.. cpp:autostruct:: name

Incorporate the documentation comment for the structure `name`, optionally including member documentation as specified by `members`.

The `file` option is as in *c:autovar*. If `file` is specified, look up `name` there, otherwise look up `name` in all previously parsed files in the current document.

:members: (text)

The `members` option specifies the struct members to include:

- If `members` is not present, do not include member documentation at all.
- If `members` is specified without arguments, include all member documentation recursively.
- If `members` is specified with a comma-separated list of arguments, include all specified member documentation recursively.

For example:

```
.. c:autostruct:: example_struct
   :file: example_file.c

.. c:autostruct:: example_struct
   :members:

.. c:autostruct:: example_struct
   :members: member_one, member_two
```

.. cpp:autoclass:: name

Same as *cpp:autostruct* but for classes.

For example:

```
.. cpp:autoclass:: example_class
   :file: example_file.cpp
   :members: member_one, member_two
```

.. c:autounion:: name

.. cpp:autounion:: name

Same as *c:autostruct* but for unions.

```
.. c:autounion:: example_union
   :file: example_file.c
   :members: some_member
```

.. c:autoenum:: name

.. cpp:autoenum:: name

Same as *c:autostruct* but for enums. The enumeration constants are considered members and are included according to the `members` option.

```
.. c:autoenum:: example_enum
   :file: example_file.c
   :members:

.. c:autoenum:: example_enum
   :members: CONSTANT_ONE, CONSTANT_TWO
```

3.4 Generic Documentation Sections

The *c:autosection* and *cpp:autosection* directives incorporate generic documentation comments not attached to any objects in the specified file.

.. c:autosection:: name

.. cpp:autosection:: name

Incorporate the generic documentation comment identified by `name`.

The `name` is derived from the first sentence of the comment, and may contain whitespace. It starts from the first alphanumeric character, inclusive, and extends to the next `:`, `.`, or newline, non-inclusive.

The `file` option is as in *c:autovar*. If `file` is specified, look up `name` there, otherwise look up `name` in all previously parsed files in the current document.

For example:

```
/**
 * This is the reference. This is not. It all becomes
 * the documentation comment.
 */
```

```
.. c:autosection:: This is the reference
   :file: example_file.c
```

Note that the above does not automatically create hyperlink targets that you could reference from reStructuredText. However, reStructuredText hyperlink targets work nicely as the reference name for the directive:

```
/**
 * .. _This is the reference:
 *
 * The actual documentation comment.
 *
 * You can use :ref:`This is the reference` to reference
 * this comment in reStructuredText.
 */
```

```
.. c:autosection:: This is the reference
   :file: example_file.c
```

CHAPTER

FOUR

SYNTAX

For the *Hawkmoth autodoc directives* to work, the C or C++ source code must be documented using specific documentation comment style, and the comments must follow reStructuredText markup.

Optionally, the syntax may be *extended* to support e.g. Javadoc/Doxygen and Napoleon style comments.

See *the examples section* for a quick tour of what's possible, and read on for documentation comment formatting details.

4.1 Documentation Comments

Documentation comments are C/C++ language block comments that begin with `/**`.

Because reStructuredText is sensitive about indentation, it's strongly recommended, even if not strictly required, to follow a uniform style for multi-line comments. Place the opening delimiter `/**` and closing delimiter `␣*/` on lines of their own, and prefix the lines in between with `␣*␣`. Indent the actual documentation at the third column, to let Hawkmoth consistently remove the enclosing comment markers:

```
/**
 * The quick brown fox jumps
 * over the lazy dog.
 */
```

One-line comments are fine too:

```
/** The quick brown fox jumps over the lazy dog. */
```

All documentation comments preceding C or C++ constructs are attached to them, and result in C or C++ Domain directives being added for them accordingly. This includes macros, functions, struct and union members, enumerations, etc.

Documentation comments followed by comments (documentation or not) are included as normal paragraphs in the order they appear.

4.2 Info Field Lists

Use reStructuredText field lists for documenting function parameters, return values, and arbitrary other data. Sphinx recognizes some info fields, such as `param` and `return`, and formats them nicely.

```
/**
 * The baznicator.
 *
 * :param foo: The Foo parameter.
 * :param bar: The Bar parameter.
 * :return: 0 on success, non-zero error code on error.
 * :since: v0.1
 */
int baz(int foo, int bar);
```

4.3 Extending the Syntax

Hawkmoth supports *extending* the syntax using *built-in* and custom extensions that convert the comments to reStructuredText.

The *hawkmoth.ext.javadoc* extension provides limited support for Javadoc and Doxygen style comments, and the *hawkmoth.ext.napoleon* extension provides support for `sphinx.ext.napoleon` style comments.

4.4 Cross-Referencing C and C++ Constructs

Under the hood, the *Hawkmoth directives* generate corresponding C and C++ domain directives. For example, *`c:autovar`* produces `c:var`. Use the Sphinx C Domain Roles and C++ Domain Roles for cross-referencing accordingly.

For example:

- `:c:var:`name`` for variables.
- `:c:func:`name`` for functions and function-like macros.
- `:cpp:class:`name`` for classes.
- `:c:member:`name.membername`` for struct and union members.

The C++ Domain does not have a `cpp:macro` directive, however, so all macros generate documentation using the C Domain `c:macro` directive. This also means macros have to be referenced using the `c:macro` role, even when otherwise using C++ Domain directives.

See the Sphinx Basic Markup and generic Cross-referencing syntax for further details on cross-referencing, and how to specify the default domain for brevity.

CHAPTER

FIVE

EXAMPLES

This page showcases Hawkmoth in action.

The `[source]` links are optional, and can be enabled via the `hawkmoth_source_uri` option.

5.1 Overview

5.1.1 Source

Listing 1: overview.c

```
/**
 * The ``c:autodoc`` directive is the easiest way to extract all the
 * documentation comments from one or more source files in one go. The
 * other directives provide more fine-grained control over what to
 * document.
 *
 * This example provides a brief overview of the most common features.
```

(continues on next page)

(continued from previous page)

```
 *
 * Note that the documentation comments below are **not** good examples
 * of how to document your code. Instead, the comments primarily
 * describe the features of Hawkmoth and Sphinx.
 *
 * Source files may contain documentation comments not attached to any C
 * constructs. They will be included as generic documentation comments,
 * like this one.
 */

/**
 * Macro documentation.
 */
#define ERROR -1

/**
 * Struct documentation.
 */
struct foo {
        /**
         * Member documentation.
         */
        const char *m1;
        /**
         * Member documentation.
         */
        int m2;
};

/**
 * Enum documentation.
 */
enum bar {
        /**
         * Enumeration constant documentation.
         */
        E1,
        /**
         * Enumeration constant documentation.
         */
        E2,
};

/**
 * Function documentation.
 *
 * :param p1: Parameter documentation
 * :param int p2: Parameter documentation with type
 * :return: Return value documentation
 */
int baz(int p1, int p2);
```

5.1.2 Directive

```
.. c:autodoc:: overview.c
```

5.1.3 Output

The `c:autodoc` directive is the easiest way to extract all the documentation comments from one or more source files in one go. The other directives provide more fine-grained control over what to document.

This example provides a brief overview of the most common features.

Note that the documentation comments below are **not** good examples of how to document your code. Instead, the comments primarily describe the features of Hawkmoth and Sphinx.

Source files may contain documentation comments not attached to any C constructs. They will be included as generic documentation comments, like this one.

ERROR

Macro documentation.

struct **foo**

Struct documentation.

const char ***m1**

Member documentation.

int **m2**

Member documentation.

enum **bar**

Enum documentation.

enumerator **E1**

Enumeration constant documentation.

enumerator **E2**

Enumeration constant documentation.

int **baz**(int p1, int p2)

Function documentation.

Parameters

- **p1** – Parameter documentation
- **p2** (`int`) – Parameter documentation with type

Returns

Return value documentation

5.2 Variable

5.2.1 Source

Listing 2: variable.c

```c
/**
 * The name says it all.
 */
const int meaning_of_life = 42;

/**
 * The list of entries.
 *
 * Use :c:func:`frob` to frobnicate, always in :c:macro:`MODE_PRIMARY` mode.
 */
static struct list *entries;

/**
 * This is a sized array.
 */
const char array[10];
```

5.2.2 Directive

```
.. c:autodoc:: variable.c
```

5.2.3 Output

const int **meaning_of_life**

The name says it all.

static struct *list* ***entries**

The list of entries.

Use *frob()* to frobnicate, always in *MODE_PRIMARY* mode.

const char **array**[10]

This is a sized array.

5.2.4 Directive

```
.. c:autovar:: meaning_of_life
   :file: variable.c
```

5.2.5 Output

const int **meaning_of_life**
: The name says it all.

5.3 Typedef

5.3.1 Source

Listing 3: typedef.c

```
/**
 * Typedef documentation.
 */
typedef void * list_data_t;
```

5.3.2 Directive

```
.. c:autotype:: list_data_t
   :file: typedef.c
```

5.3.3 Output

type **list_data_t**
: Typedef documentation.

5.4 Macro

5.4.1 Source

Listing 4: macro.c

```
/**
 * Failure status.
 */
#define FAILURE 13

/**
 * Terminate immediately with failure status.
 *
 * See :c:macro:`FAILURE`.
 */
#define DIE() _exit(FAILURE)

/**
```

(continues on next page)

(continued from previous page)

```
 * Get the number of elements in an array.
 *
 * :param array: An array
 * :return: Array size
 */
#define ARRAY_SIZE(array) (sizeof(array) / sizeof(array[0]))

/**
 * Variadic macros
 *
 * :param foo: regular argument
 * :param ...: variable argument
 */
#define VARIADIC_MACRO(foo, ...) (__VA_ARGS__)
```

5.4.2 Directive

```
.. c:autodoc:: macro.c
```

5.4.3 Output

FAILURE

Failure status.

DIE()

Terminate immediately with failure status.

See *FAILURE*.

ARRAY_SIZE(array)

Get the number of elements in an array.

Parameters

- **array** – An array

Returns

Array size

VARIADIC_MACRO(foo, ...)

Variadic macros

Parameters

- **foo** – regular argument
- **...** – variable argument

5.4.4 Directive

```
.. c:automacro:: DIE
   :file: macro.c
```

5.4.5 Output

DIE()

Terminate immediately with failure status.

See *FAILURE*.

5.5 Function

5.5.1 Source

Listing 5: function.c

```c
struct list;
enum mode;

/**
 * List frobnicator.
 *
 * :param list: The list to frob.
 * :param mode: The frobnication mode.
 * :return: 0 on success, non-zero error code on error.
 * :since: v0.1
 */
int frob(struct list *list, enum mode mode);

/**
 * variadic frobnicator
 *
 * :param fmt: the format
 * :param ...: variadic
 */
int frobo(const char *fmt, ...);
```

5.5.2 Directive

```
.. c:autodoc:: function.c
```

5.5.3 Output

int **frob**(struct *list* *list, enum *mode* mode)

List frobnicator.

Parameters

- **list** – The list to frob.
- **mode** – The frobnication mode.

Returns

0 on success, non-zero error code on error.

Since

v0.1

int **frobo**(const char *fmt, ...)

variadic frobnicator

Parameters

- **fmt** – the format
- **...** – variadic

5.5.4 Directive

```
.. c:autofunction:: frob
   :file: function.c
```

5.5.5 Output

int **frob**(struct *list* *list, enum *mode* mode)

List frobnicator.

Parameters

- **list** – The list to frob.
- **mode** – The frobnication mode.

Returns

0 on success, non-zero error code on error.

Since

v0.1

5.6 Struct

5.6.1 Source

Listing 6: struct.c

```
/**
 * Linked list node.
 */
struct list {
        /** Next node. */
        struct list *next;

        /** Data. */
        int data;
};
```

5.6.2 Directive

```
.. c:autodoc:: struct.c
```

5.6.3 Output

struct **list**

Linked list node.

struct *list* ***next**

Next node.

int **data**

Data.

5.6.4 Directive

```
.. c:autostruct:: list
   :file: struct.c
   :members:
```

5.6.5 Output

struct **list**

Linked list node.

struct *list* ***next**

Next node.

int **data**

Data.

5.7 Class

5.7.1 Source

Listing 7: class.cpp

```
/**
 * Circle.
 */
class Circle {
private:
        /** Radius */
        int radius;

public:
        /** Constructor */
        Circle(int radius);

        /** Destructor */
        ~Circle();

        /** Get the area. */
        virtual int area(void);
};
```

5.7.2 Directive

```
.. cpp:autoclass:: Circle
   :file: class.cpp
   :members:
```

5.7.3 Output

class **Circle**

Circle.

private int **radius**

Radius

Circle(int radius)

Constructor

~Circle(void)

Destructor

virtual int **area**(void)

Get the area.

5.8 Union

5.8.1 Source

Listing 8: union.c

```c
/**
 * Onion documentation.
 */
union onion {
        /**
         * Yellow onion.
         */
        int yellow;
        /**
         * Red onion.
         */
        int red;
        /**
         * White onion.
         */
        int white;
};
```

5.8.2 Directive

```
.. c:autounion:: onion
   :file: union.c
   :members:
```

5.8.3 Output

union **onion**

Onion documentation.

int **yellow**

Yellow onion.

int **red**

Red onion.

int **white**

White onion.

5.9 Enum

5.9.1 Source

Listing 9: enum.c

```c
/**
 * Frobnication modes for :c:func:`frob`.
 */
enum mode {
        /**
         * The primary frobnication mode.
         */
        MODE_PRIMARY,
        /**
         * The secondary frobnication mode.
         *
         * If the enumerator is initialized in source, its value will also be
         * included in documentation.
         */
        MODE_SECONDARY = 2,
};
```

5.9.2 Directive

```
.. c:autoenum:: mode
   :file: enum.c
   :members:
```

5.9.3 Output

enum **mode**

Frobnication modes for *frob()*.

enumerator **MODE_PRIMARY**

The primary frobnication mode.

enumerator **MODE_SECONDARY** = 2

The secondary frobnication mode.

If the enumerator is initialized in source, its value will also be included in documentation.

5.10 Generic Documentation Section

5.10.1 Source

Listing 10: autosection.c

```
/**
 * .. _Hyperlink Target:
 *
 * This is a generic documentation comment.
 *
 * Because generic documentation comments aren't attached to any symbols, the
 * comment itself has to contain a name that can be referenced from the
 * ``c:autosection`` or ``cpp:autosection`` directive.
 *
 * The name shall be from the first alphanumeric character in the comment,
 * inclusive, to the next :, ., or newline, non-inclusive. This means
 * reStructuredText hyperlink targets become reference names, like in this case,
 * but it does not have to be a hyperlink target. It could just be the first
 * sentence in the comment.
 *
 * No hyperlink targets are generated automatically. If you need to reference
 * the comment from reStructuredText, you need to add one yourself.
 */
```

5.10.2 Directive

```
.. c:autosection:: Hyperlink Target
   :file: autosection.c
```

5.10.3 Output

This is a generic documentation comment.

Because generic documentation comments aren't attached to any symbols, the comment itself has to contain a name that can be referenced from the `c:autosection` or `cpp:autosection` directive.

The name shall be from the first alphanumeric character in the comment, inclusive, to the next :, ., or newline, non-inclusive. This means reStructuredText hyperlink targets become reference names, like in this case, but it does not have to be a hyperlink target. It could just be the first sentence in the comment.

No hyperlink targets are generated automatically. If you need to reference the comment from reStructuredText, you need to add one yourself.

5.11 Preprocessor

5.11.1 Source

Listing 11: preprocessor.c

```
/**
 * Answer to the Ultimate Question of Life, The Universe, and Everything.
 */
#ifdef DEEP_THOUGHT
#define MEANING_OF_LIFE 42
#else
#error "Does not compute."
#endif
```

5.11.2 Directive

```
.. c:autodoc:: preprocessor.c
   :clang: -DDEEP_THOUGHT
```

5.11.3 Output

MEANING_OF_LIFE
Answer to the Ultimate Question of Life, The Universe, and Everything.

5.12 Napoleon-style comments

5.12.1 Source

Listing 12: napoleon.c

```
/**
 * Custom comment transformations.
 *
 * Documentation comments can be processed using the hawkmoth-process-docstring
 * Sphinx event. You can use the built-in extensions for this, or create your
 * own.
 *
 * In this example, hawkmoth.ext.napoleon built-in extension is used to support
 * Napoleon-style documentation comments.
 *
 * Args:
 *     foo: This is foo.
 *     bar: This is bar.
 *
 * Return:
 *     Status.
```

(continues on next page)

(continued from previous page)

```
 */
int napoleon(int foo, char *bar);
```

5.12.2 Directive

```
.. c:autodoc:: napoleon.c
   :transform: napoleon
```

5.12.3 Output

int **napoleon**(int foo, char *bar)

Custom comment transformations.

Documentation comments can be processed using the hawkmoth-process-docstring Sphinx event. You can use the built-in extensions for this, or create your own.

In this example, hawkmoth.ext.napoleon built-in extension is used to support Napoleon-style documentation comments.

Parameters

- **foo** – This is foo.
- **bar** – This is bar.

Returns

Status.

5.13 Javadoc/Doxygen-style comments

5.13.1 Source

Listing 13: javadoc.c

```
struct list;
enum mode;

/**
 * Custom comment transformations.
 *
 * Documentation comments can be processed using the hawkmoth-process-docstring
 * Sphinx event. You can use the built-in extensions for this, or create your
 * own.
 *
 * In this example, <tt>hawkmoth.ext.javadoc</tt> built-in extension is used to
 * support Javadoc/Doxygen-style documentation comments. You can use both \@ and
 * \\ for the commands.
 *
 * \note
```

(continues on next page)

(continued from previous page)

```
 * While the most common commands and inline markup \a should work, the
 * Javadoc/Doxygen support is nowhere near complete.
 *
 * The support should be good enough for basic API documentation, including
 * things like code blocks:
 *
 * \code
 * ¯\_()_/¯
 * \endcode
 *
 * And parameter and return value descriptions, and the like:
 *
 * @param list The list to frob.
 * @param[in] mode The frobnication mode.
 * @return 0 on success, non-zero error code on error.
 * @since v0.1
 */
int frob2(struct list *list, enum mode mode);
```

5.13.2 Directive

```
.. c:autodoc:: javadoc.c
   :transform: javadoc
```

5.13.3 Output

int **frob2**(struct *list* *list, enum *mode* mode)

Custom comment transformations.

Documentation comments can be processed using the hawkmoth-process-docstring Sphinx event. You can use the built-in extensions for this, or create your own.

In this example, `hawkmoth.ext.javadoc` built-in extension is used to support Javadoc/Doxygen-style documentation comments. You can use both @ and \ for the commands.

Note: While the most common commands and inline markup *should* work, the Javadoc/Doxygen support is nowhere near complete.

The support should be good enough for basic API documentation, including things like code blocks:

```
¯\_()_/¯
```

And parameter and return value descriptions, and the like:

Parameters

- **list** – The list to frob.
- **mode** – **[in]** The frobnication mode.

Returns

0 on success, non-zero error code on error.

Since
v0.1

CHAPTER

SIX

EXTENDING

Hawkmoth is a Sphinx extension that can be further extended with other Sphinx extensions.

6.1 Events

See `sphinx.application.Sphinx.connect()` on how to connect events.

hawkmoth-process-docstring

func(*app*, *lines*, *transform*, *options*)

Parameters

- **app** (`sphinx.application.Sphinx`) – The Sphinx application object
- **lines** (*list[str]*) – The comment being processed
- **transform** (*str*) – Transformation
- **options** (*dict*) – The directive options

This is similar to the `autodoc-process-docstring` event in the `sphinx.ext.autodoc` extension.

The *lines* argument is the documentation comment, with the comment markers removed, as a list of strings that the event handler may modify in-place.

The *transform* argument is the `transform` option of the *directive* being processed, defaulting to *hawkmoth_transform_default*, which defaults to `None`. The event handler may use this to decide what, if anything, should be done to *lines*.

The *options* argument is a dictionary with all the options given to the directive being processed.

Note: Please note that this API is still somewhat experimental and in development. In particular, new arguments may be added in the future.

CHAPTER

SEVEN

BUILT-IN EXTENSIONS

Hawkmoth is *extensible*, and ships with some built-in extensions.

7.1 hawkmoth.ext.javadoc

This extension converts Javadoc and Doxygen comments to reStructuredText, using the *hawkmoth-process-docstring* event.

The most commonly used commands are covered, including some inline markup, using either @ or \ command character. The support is not complete, and mainly covers the basic API documentation needs.

Note that this does not change the comment block format, only the contents of the comments. Only the `/** ... */` format is supported.

Installation and configuration in `conf.py`:

```
extensions.append('hawkmoth.ext.javadoc')
```

hawkmoth_javadoc_transform: *str*

Name of the transformation to handle. Defaults to `'javadoc'`. Only convert the comment if the `transform` option matches this name, otherwise do nothing. Usually there's no need to modify this option.

For example:

Listing 1: conf.py

```
extensions.append('hawkmoth.ext.javadoc')
hawkmoth_transform_default = 'javadoc'  # Transform everything
```

hawkmoth_transform_default sets the default for the `transform` option.

Listing 2: file.c

```
/**
 * The baznicator.
 *
 * @param foo The Foo parameter.
 * @param bar The Bar parameter.
 * @return 0 on success, non-zero error code on error.
 * @since v0.1
 */
int baz(int foo, int bar);
```

Listing 3: api.rst

```
.. c:autofunction:: baz
   :file: file.c
```

7.2 hawkmoth.ext.napoleon

This extension provides a bridge from Hawkmoth to the `sphinx.ext.napoleon` extension, using the *hawkmoth-process-docstring* event, to support Napoleon style documentation comments.

Installation and configuration in `conf.py`:

```
extensions.append('hawkmoth.ext.napoleon')
```

hawkmoth_napoleon_transform: `str`

Name of the transformation to handle. Defaults to `'napoleon'`. Only convert the comment if the `transform` option matches this name, otherwise do nothing. Usually there's no need to modify this option.

For example:

Listing 4: conf.py

```
extensions.append('hawkmoth.ext.napoleon')
# Uncomment to transform everything, example below uses :transform: option
# hawkmoth_transform_default = 'napoleon'
```

Listing 5: file.c

```
/**
 * The baznicator.
 *
 * Args:
 *     foo: The Foo parameter.
 *     bar: The Bar parameter.
 *
 * Returns:
 *     0 on success, non-zero error code on error.
 */
int baz(int foo, int bar);
```

Listing 6: api.rst

```
.. c:autofunction:: baz
   :file: file.c
   :transform: napoleon
```

7.3 hawkmoth.ext.transformations

This extension handles the *cautodoc_transformations* feature, using the *hawkmoth-process-docstring* event.

Note: Going forward, it's recommended to handle transformations using the event directly instead of *cautodoc_transformations*. This built-in extension provides backward compatibility for the functionality.

For now, this extension is loaded by default, and the installation step below is not strictly necessary. This will change in the future.

Installation and configuration in `conf.py`:

```
extensions.append('hawkmoth.ext.transformations')
```

cautodoc_transformations: *dict*

Transformation functions for the *c:autodoc* directive `transform` option. This is a dictionary that maps names to functions. The names can be used in the directive `transform` option. The functions are expected to take a (multi-line) comment string as a parameter, and return the transformed string. This can be used to perform custom conversions of the comments, including, but not limited to, Javadoc-style compat conversions.

The special key `None`, if present, is used to convert everything, unless overridden in the directive `transform` option. The special value `None` means no transformation is to be done.

For example, this configuration would transform everything using `default_transform` function by default, unless overridden in the directive `transform` option with `javadoc` or `none`. The former would use `javadoc_transform` function, and the latter would bypass transform altogether.

```
cautodoc_transformations = {
    None: default_transform,
    'javadoc': javadoc_transform,
    'none': None,
}
```

The example below shows how to use Hawkmoth's existing compat functions in `conf.py`.

```
from hawkmoth.util import doccompat
cautodoc_transformations = {
    'javadoc-basic': doccompat.javadoc,
    'javadoc-liberal': doccompat.javadoc_liberal,
    'kernel-doc': doccompat.kerneldoc,
}
```

CHAPTER

EIGHT

TIPS AND TRICKS

Here is a small collection of tips and tricks on how to use Sphinx and Hawkmoth for documenting C and C++ code.

8.1 Function Parameter Direction

Sphinx does not have a dedicated way of expressing the parameter direction similar to Doxygen @param[dir] command. One approach to emulate this is to define reStructuredText replacement texts, and use them.

For example:

Listing 1: conf.py

```
rst_prolog = '''
.. |in| replace:: **[in]**
.. |out| replace:: **[out]**
.. |in,out| replace:: **[in,out]**
'''
```

Listing 2: source code

```
/**
 * :param foo: |in| Foo parameter.
 */
void bar(char *foo);
```

By using replacement text, the direction stands out in the source code, you get warnings for typos, and you can modify the appearance across documentation in one place. Instead of `**[in]**`, you might use , or whatever you prefer.

8.2 Including Source Code Blocks

Doxygen has the @include and @snippet commands to include a source file or a fragment of one into documentation as a block of code.

The Sphinx alternative is `literalinclude`. The `:start-after:` and `:end-before:` options can be used to mimic the `block_id` of `@snippet`, but there's plenty more.

```
.. literalinclude:: path/to/source.c
   :start-after: Adding a resource
   :end-before: Adding a resource
```

8.3 Using sphinx-autobuild with Hawkmoth

sphinx-autobuild is a handy tool to automatically rebuild Sphinx documentation when you modify the `.rst` files, with live-reload in the browser.

It's possible to have it auto rebuild and live-reload source documentation on source code changes by adding `--watch <source root>` option to `sphinx-autobuild`, where `<source root>` matches *hawkmoth_root*.

CHAPTER

NINE

TROUBLESHOOTING

Things not working? Here are some things to try isolate the problem.

9.1 Use the parser directly

Hawkmoth comes with a command-line debug tool to extract the documentation comments from source without Sphinx. This can be useful in figuring out if the problem is in the parser or in the Sphinx extension.

```
hawkmoth path/to/file.c
```

See the help for command-line options:

```
hawkmoth --help
```

9.2 Get verbose output from Sphinx

Pass the `-v` option to `sphinx-build` to get more verbose output, and see if anything stands out.

```
sphinx-build -v SOURCEDIR OUTPUTDIR
```

or

```
make SPHINXOPTS=-v html
```

You can also use `-vv` for even more verbose output.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

B

C

D

E

F

H

L

M

N

O

P

V