Cobaro libcobaro-log0 DeveloperGuide

Release 1.0.0

cobaro.org

May 17, 2015

CONTENTS

1 Quickstart **1**

2 Installation **3**
2.1 Binary packages . . . 3
2.2 Building from source . . . 3
2.3 Dependencies . . . 3

3 Compilation and Linkage **5**
3.1 Static vs Dynamic Linkage . . . 5

4 General Philosophy **7**

5 Library Initialisation (and Finalisation) **9**
5.1 Define your messages . . . 9
5.2 Initialisation . . . 9
5.3 Finalisation . . . 10

6 Creating a Log Message **11**

7 Publishing a Log Message **13**
7.1 Using your own Queue . . . 13

8 Reporting a Log Message **15**
8.1 Logging to File . . . 15
8.2 Logging to syslog . . . 15
8.3 Logging to String . . . 15

9 Defining Log Templates **17**

10 Licensing **19**

11 References **21**

CHAPTER

ONE

QUICKSTART

A super-quick overview of how to use Cobaro Log in your program.

- Install the developer package, or build from source.
- Add *pkg-config –cflags libcobaro-log0* to CPPFLAGS
- Add *pkg-config –libs libcobaro-log0* to LDFLAGS
- Include the header:

```
#include <cobaro-log0/log.h>
```

- Create an enum and matching array for your log messages:

```
enum my_log_codes {
    MY_LOG_GOOD,
    MY_LOG_SOSO,
    MY_LOG_BAD
};

const char *my_log_msgs[] = {
    "%1? It's all good",
    "%1? It's ok, but ... %2",
    "%1? Urgh.  %2 failed %3 times."
};
```

- Create a log handle:

```
cobaro_loghandle_t lh = cobaro_log_init(my_log_messages);
```

- In a critical thread, create and send a log message:

```
cobaro_log_t log = cobaro_log_claim(lh);

log->code = MY_LOG_BAD;
log->level = COBARO_LOG_ERR;
cobaro_log_set_string(log, 1, "sample");
cobaro_log_set_string(log, 2, "badfunc");
cobaro_log_set_integer(log, 3, 42);

cobaro_log_publish(lh, log);
```

- While in a background thread, format and report messages:

```
cobaro_log_loglevel_set(lh, COBARO_LOG_NOTICE);
cobaro_log_syslog_set(lh, "my_app", LOG_PID, LOG_DAEMON);

while (run) {
    cobaro_log_t log = cobaro_log_next(lh);
    cobaro_log(lh, log);
    cobaro_log_return(lh, log);
```

```
};

cobaro_log_fini(lh);
```

- Done!

CHAPTER

TWO

INSTALLATION

Cobaro Log can be installed using the binary packages available for popular OS distributions, or from source.

2.1 Binary packages

Binary packages for a limited number of platforms (typically latest Ubuntu LTS, RedHat/CentOS, and OSX) are available from cobaro.org.

For RedHat and Debian based systems the runtime package contains only the shared libraries and need only be installed if shared library linkage is desired. The developer package contains everything needed to build statically-linked applications.

2.2 Building from source

Source releases can be obtained from github.

To initialize your build environment, or to reset it for some other reason, you should run the `bootstrap` script.

To configure your build for the host OS platform, set compiler options, and choose optional build features, you should run the `configure` script (generated by `bootstrap`). `configure --help` describes its many options.

To compile and link the main library, you should run `make` (after having run `bootstrap` and `configure`). You do not need to re-bootstrap and re-configure for each build.

To install, run `make install`, and to remove the installed files, `make uninstall`. These will target `/usr/local` by default, but you can override this with the `--prefix` option to `configure`.

To run the unit tests, run `make check`.

To build an RPM or DEB package, run `make package` on the target platform.

2.3 Dependencies

There are no runtime dependencies. Build-time dependencies vary by platform. Those listed here cover building the complete packages, including documentation.

- Build dependencies for RHEL/CentOS
  - doxygen
  - framed
  - python-sphinx
  - texlive

  - texlive-collection-xetex
  - texlive-multirow
  - texlive-threeparttable
  - texlive-wrapfig
  - texlive-xcolor
- Build dependencies for macports
  - doxygen
  - py-sphinx
  - texlive
  - texlive-fonts-recommended
  - texlive-latex-extra

CHAPTER

THREE

COMPILATION AND LINKAGE

If you can use pkgconfig, it can provide the correct compilation and linkage flags:

```
pkgconfig --cflags libcobaro-log0
pkgconfig --libs libcobaro-log0
```

To specify things manually is also simple:

```
-I /usr/local/include -pthread
-L /usr/local/lib -lcobaro-log0 -pthread
```

3.1 Static vs Dynamic Linkage

When linking the logging library, you can choose to do so by including the object code within your executable (static linking), or by referencing an installed shared libary (dynamic linking). The autotools toolchain encourages dynamic linking, and makes it quite hard to statically link a library if a shared version is installed.

So, in the Linux packages, if you install just the developer package, no shared library is installed. This will force autotools to use the static library (libcobaro-log0.a). If you install the runtime package on your development machine, autotools will link the dynamic library (libcobaro-log0.so) by default.

You can try to work around this by passing a path to the .a file to the linker (without a -l flag), but ... that's in hairy territory and having mentioned it, we'll leave you on your own with that.

CHAPTER

FOUR

GENERAL PHILOSOPHY

The Cobaro Log package is intended for use in circumstances where a more traditional logging package (syslog, log4c, etc) is too CPU-time consuming. Specifically, it's intended to perform no I/O and no string-formatting: just to collect the relevant data, and hand it over to something else to deal with.

The library is split into two groups of functions:

- Those that create log message.
- Those that process log messages.

Log messages are represented by a structure containing the log message identifier (an integer), the log priority (from syslog, emergency to debug), and a collection of parameters to describe the event being reported.

The code that reports an event should acquire a log structure, fill out the identifier, level, and parameters, and then hand it off to have the text message generated and reported using a traditional logging system.

The text message is generated from a template, quite similar to printf(). Templates are defined in one or more arrays of strings. Only one array is needed, but you can select between different arrays if you need to support different languages.

Within the strings in the array, placeholders like "%1" specify a value from the associated array of parameter values. When the message is formatted, the values are substituted into the template to produce the final message.

The thread that generates the log message doesn't need to know about the message arrays: it simply specifies a log message _number_ (an index into the messages array), the log level, and populates the parameters array.

A service thread can subsequently look up the text template, perform the substitutions, and pass the generated text message on to general purpose logging system (like syslog).

CHAPTER

FIVE

LIBRARY INITIALISATION (AND FINALISATION)

The library provides two distinct types: `cobaro_log_t`, the log message structure; and `cobaro_loghandle_t`, a formatting infrastructure.

5.1 Define your messages

Log messages are defined with string message templates, stored in an array.

```
const char *my_log_msgs[] = {
    "Template with parameters: %1, %2, %3.",
    "Another template"
};
```

The log message templates may (optionally) contain parameter markers, which are replaced by values from the log structure when the template is formatted.

Each message template has an index in the array. The value of this index is used to select the log message in the log structure. It's often convenient to define an enum, rather than relying on the magic numbers.

```
enum my_log_codes {
    MY_LOG_THING_WITH_ARGS,
    MY_LOG_ANOTHER
};
```

5.2 Initialisation

A log *handle* contains the configured state for using the Cobaro Log system in your program. You may have any number of log handles with different configurations.

The handle is initialised with the set of defined messages.

```
cobaro_loghandle_t log_handle = cobaro_log_init(my_log_msgs);
```

Note that you can have multiple arrays of templates: in the case where your program needs to log in different languages, an array for the active language can be specified during initialisation, derived from the program's locale or other setting.

The array can be changed at runtime too:

```
cobaro_log_messages_set(log_handle, my_logs_msgs_cn);
```

5.3 Finalisation

Cleanup is simple.

```
cobaro_log_fini(log_handle);
```

This function can be used to clean up any allocated memory within the handle structure before exiting.

CHAPTER

SIX

CREATING A LOG MESSAGE

The first step to creating a log message is to acquire one. It's normally not feasible to use stack allocation, since the log structure is handed to another thread for formatting. You can simply `malloc()` one, use your own allocation pool, or use the pool implemented by the log handle type.

```
log = cobaro_log_claim(log_handle);
```

Claim a log from the handle's collection. If none are free, you'll get `NULL`.

Set the log message code and level:

```
log->code = MY_APP_LOG_MESSAGE_FOO;
log->level = MY_APP_LOG_LEVEL_FOO;
```

Set any parameters needed:

```
cobaro_log_set_string(log, 1, "boom!");
cobaro_log_set_integer(log, 2, 42);
cobaro_log_set_double(log, 3, 3.1416);
cobaro_log_set_ipv4(log, 4, ipaddr);
```

Note that the index parameter in these functions matches the parameter number in the template strings: it starts from 1, and is the index into the parameter array + 1.

Of these functions, set_string() does the most work, copying the string contents up to the size of the parameter array's strings, and terminating them correctly.

At this point you have a fully populated log structure, and need to decide what to do with it.

CHAPTER

SEVEN

PUBLISHING A LOG MESSAGE

The Cobaro Log log handle type has an in-built inter-thread queue, suitable for publishing log messages to a background thread for formatting and reporting via eg. syslog.

Alternatively, you can use your own inter-thread communications channels to hand over the log_t pointer to a service thread.

```
cobaro_log_publish(log_handle, log);
```

This function queues the provided log structure for processing by another thread sharing this handle.

The other thread should call

```
log = cobaro_log_next(log_handle);
```

to retrieve log messages from this queue, process them, and then call

```
cobaro_log_return(log_handle, log);
```

to return the structure to the handle's allocation pool (for use by future calls to cobaro_log_claim()).

7.1 Using your own Queue

To use your own communication channel between the source thread and the reporting thread, you can take advantage of the cobaro_log_t->id header. This is a four-byte field at the start of the log_t structure that has no use in the Cobaro Log system, and is intended to be populated with header information for an external communications system if required.

For instance, if you have a queue between multiple threads already in use for control messages, usage reporting, etc, log messages can also be passed via this path. In some cases, the pointer could be used directly together with the id header to identify this pointer as a log message, rather than a control message. In other cases, it'll be necessary to wrap the log_t pointer in a suitable envelope structure.

```
log->id = MY_APP_LOG_MESSAGE;
my_queue_append(my_queue, (void *)log);
```

Note that in this case you also need to ensure that the memory management is taken care of. The log handle's free list is small (to reduce cache pressure), so you need to ensure that cobaro_log_return() is called as soon as possible if you're using the log handle's allocation pool.

CHAPTER

EIGHT

REPORTING A LOG MESSAGE

The log handle has support for logging to file, to syslog, and a generic function for formatting the message string for use with any logging system.

In the most simple configuration, you select a file:

```
FILE *f = fopen("/var/log/myapp.log", "w');
cobaro_log_file_set(log_handle, f);
```

or, *syslog* facility, with your application's name and syslog options (see syslog(3)):

```
cobaro_log_syslog_set(log_handle, "my_app", LOG_PID, LOG_DAEMON);
```

And then call

```
cobaro_log(log_handle, log);
```

to actually report a log message.

If you want more flexibility, you can call the underlying functions directly.

8.1 Logging to File

Directly log a message to an open file. This ignores any configuration of a destination via the handle, but does check the handle's log level.

```
cobaro_log_to_file(log_handle, log, file);
```

Logs written to a file are preceded by a local timestamp with displayed microseconds. Note that this is the time of reporting, not time of occurrence.

8.2 Logging to syslog

Directly send a log message to syslog.

```
cobaro_log_to_syslog(log_handle, log);
```

Calling this function assumes that you've ensured openlog(3) has already been called with your desired identification, options, and facility.

8.3 Logging to String

Finally, you can log a message directly to a character buffer, and subsequently do whatever you like with it. The file and syslog functions use this function internally to prepare the message. It performs parameter substitution on the template, and writes the resulting message to the provided buffer.

```
cobaro_log_to_string(log_handle, log, buffer, buflen);
```

The result buffer will always be correctly terminated, and will not overflow.

CHAPTER

NINE

DEFINING LOG TEMPLATES

We recommend that messages be defined as part of template files. As an example from our test code here is a fragment from our test header template file (see https://github.com/cobaro/liblog/blob/master/test/messages.h)

```
/// Enumeration of message identifiers
///
/// These are array looked up so must range from 0 to COBARO_MSG_COUNT
enum cobaro_test_message_ids {
   COBARO_TEST_MESSAGE_NULL = 0,
   COBARO_TEST_MESSAGE_TYPES = 1,

   COBARO_TEST_MSG_COUNT
};

extern char *cobaro_messages_en[];
extern char *cobaro_messages_klingon[];
```

and from the corresponding source file (see https://github.com/cobaro/liblog/blob/master/test/messages.c)

```
/// The RULES
/// You have eight parameters to play with represented as %1 through %8.
/// To print a '%' (percentage) symbol use %%, otherwise they're dropped.
/// Arguments can be used multiple times in any order you choose so
/// "%4 %3 %4" is fine.
/// We don't give you options for pretty formatting of leading zeros etc.

/// Catalog of log messages
char *cobaro_messages_en [COBARO_TEST_MSG_COUNT + 1] = {
   // COBARO_TEST_MESSAGE_NULL
   // string literal
   "%1",

   // COBARO_TEST_MESSAGE_TYPES
   // string, int, float, ip
   "s:%1, i:%2, f:%3, ip:%4, percent:%%",


   // so trailing commas always work
   ""
};

char *cobaro_messages_klingon [COBARO_TEST_MSG_COUNT + 1] = {
   // COBARO_TEST_MESSAGE_NULL
   // string literal
   "%1",

   // COBARO_TEST_MESSAGE_TYPES
   // string, int, float, ip
   "i:%2, s:%1, hoch:%1, f:%3, ip:%4, chipath:%%",
```

```
   // so trailing commas always work
   ""
};
```

You can then initialize a log handle with a specific language:

```
cobaro_loghandle_t lh = cobaro_log_init(cobaro_messages_en);
```

and you could change languages at runtime:

```
void cobaro_log_messages_set(lh, cobaro_messages_klingon);
```

CHAPTER

TEN

LICENSING

Cobaro Log is licensed using the MIT license. You are free to include this code in commercial products, and to modify it as you require:


```
MIT License
-----------

Copyright (C) 2015, Cobaro.org.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```


Contributions to the project are welcomed. Please create a GitHub issue with patch attached, or send a pull request.

CHAPTER

ELEVEN

REFERENCES

See also:

- Reference Guide (doxygen)
- README
- github