
libbpf

unknown

Aug 22, 2022

CONTENTS

1	API naming convention	1
2	API documentation convention	5
3	Building libbpf	7

API NAMING CONVENTION

libbpf API provides access to a few logically separated groups of functions and types. Every group has its own naming convention described here. It's recommended to follow these conventions whenever a new function or type is added to keep libbpf API clean and consistent.

All types and functions provided by libbpf API should have one of the following prefixes: `bpf_`, `btf_`, `libbpf_`, `btf_dump_`, `ring_buffer_`, `perf_buffer_`.

1.1 System call wrappers

System call wrappers are simple wrappers for commands supported by `sys_bpf` system call. These wrappers should go to `bpf.h` header file and map one to one to corresponding commands.

For example `bpf_map_lookup_elem` wraps `BPF_MAP_LOOKUP_ELEM` command of `sys_bpf`, `bpf_prog_attach` wraps `BPF_PROG_ATTACH`, etc.

1.2 Objects

Another class of types and functions provided by libbpf API is “objects” and functions to work with them. Objects are high-level abstractions such as BPF program or BPF map. They're represented by corresponding structures such as `struct bpf_object`, `struct bpf_program`, `struct bpf_map`, etc.

Structures are forward declared and access to their fields should be provided via corresponding getters and setters rather than directly.

These objects are associated with corresponding parts of ELF object that contains compiled BPF programs.

For example `struct bpf_object` represents ELF object itself created from an ELF file or from a buffer, `struct bpf_program` represents a program in ELF object and `struct bpf_map` is a map.

Functions that work with an object have names built from object name, double underscore and part that describes function purpose.

For example `bpf_object__open` consists of the name of corresponding object, `bpf_object`, double underscore and `open` that defines the purpose of the function to open ELF file and create `bpf_object` from it.

All objects and corresponding functions other than BTF related should go to `libbpf.h`. BTF types and functions should go to `btf.h`.

1.3 Auxiliary functions

Auxiliary functions and types that don't fit well in any of categories described above should have `libbpf_` prefix, e.g. `libbpf_get_error` or `libbpf_prog_type_by_name`.

1.4 ABI

libbpf can be both linked statically or used as DSO. To avoid possible conflicts with other libraries an application is linked with, all non-static libbpf symbols should have one of the prefixes mentioned in API documentation above. See API naming convention to choose the right name for a new symbol.

1.5 Symbol visibility

libbpf follow the model when all global symbols have visibility "hidden" by default and to make a symbol visible it has to be explicitly attributed with `LIBBPF_API` macro. For example:

```
LIBBPF_API int bpf_prog_get_fd_by_id(__u32 id);
```

This prevents from accidentally exporting a symbol, that is not supposed to be a part of ABI what, in turn, improves both libbpf developer- and user-experiences.

1.6 ABI versionning

To make future ABI extensions possible libbpf ABI is versioned. Versioning is implemented by `libbpf.map` version script that is passed to linker.

Version name is `LIBBPF_` prefix + three-component numeric version, starting from `0.0.1`.

Every time ABI is being changed, e.g. because a new symbol is added or semantic of existing symbol is changed, ABI version should be bumped. This bump in ABI version is at most once per kernel development cycle.

For example, if current state of `libbpf.map` is:

```
LIBBPF_0.0.1 {  
    global:  
        bpf_func_a;  
        bpf_func_b;  
    local:  
        *;  
};
```

, and a new symbol `bpf_func_c` is being introduced, then `libbpf.map` should be changed like this:

```
LIBBPF_0.0.1 {  
    global:  
        bpf_func_a;  
        bpf_func_b;  
    local:  
        *;  
};
```

(continues on next page)

(continued from previous page)

```
LIBBPF_0.0.2 {  
    global:  
        bpf_func_c;  
} LIBBPF_0.0.1;
```

, where new version LIBBPF_0.0.2 depends on the previous LIBBPF_0.0.1.

Format of version script and ways to handle ABI changes, including incompatible ones, described in details in [1].

1.7 Stand-alone build

Under <https://github.com/libbpf/libbpf> there is a (semi-)automated mirror of the mainline's version of libbpf for a stand-alone build.

However, all changes to libbpf's code base must be upstreamed through the mainline kernel tree.

API DOCUMENTATION CONVENTION

The libbpf API is documented via comments above definitions in header files. These comments can be rendered by doxygen and sphinx for well organized html output. This section describes the convention in which these comments should be formatted.

Here is an example from btf.h:

```
/**
 * @brief **btf__new() creates a new instance of a BTF object from the raw
 * bytes of an ELF's BTF section
 * @param data raw bytes
 * @param size number of bytes passed in `data`
 * @return new BTF object instance which has to be eventually freed with
 * **btf__free()
 *
 * On error, error-code-encoded-as-pointer is returned, not a NULL. To extract
 * error code from such a pointer `libbpf_get_error()` should be used. If
 * `libbpf_set_strict_mode(LIBBPF_STRICT_CLEAN_PTRS)` is enabled, NULL is
 * returned on error instead. In both cases thread-local `errno` variable is
 * always set to error code as well.
 */
```

The comment must start with a block comment of the form ‘/**’.

The documentation always starts with a @brief directive. This line is a short description about this API. It starts with the name of the API, denoted in bold like so: **api_name**. Please include an open and close parenthesis if this is a function. Follow with the short description of the API. A longer form description can be added below the last directive, at the bottom of the comment.

Parameters are denoted with the @param directive, there should be one for each parameter. If this is a function with a non-void return, use the @return directive to document it.

2.1 License

libbpf is dual-licensed under LGPL 2.1 and BSD 2-Clause.

2.2 Links

- [1] <https://www.akkadia.org/drepper/dsohowto.pdf>
(Chapter 3. Maintaining APIs and ABIs).

BUILDING LIBBPF

libelf and zlib are internal dependencies of libbpf and thus are required to link against and must be installed on the system for applications to work. pkg-config is used by default to find libelf, and the program called can be overridden with PKG_CONFIG.

If using pkg-config at build time is not desired, it can be disabled by setting NO_PKG_CONFIG=1 when calling make.

To build both static libbpf.a and shared libbpf.so:

```
$ cd src
$ make
```

To build only static libbpf.a library in directory build/ and install them together with libbpf headers in a staging directory root/:

```
$ cd src
$ mkdir build root
$ BUILD_STATIC_ONLY=y OBJDIR=build DESTDIR=root make install
```

To build both static libbpf.a and shared libbpf.so against a custom libelf dependency installed in /build/root/ and install them together with libbpf headers in a build directory /build/root/:

```
$ cd src
$ PKG_CONFIG_PATH=/build/root/lib64/pkgconfig DESTDIR=/build/root make
```

This is documentation for libbpf, a userspace library for loading and interacting with bpf programs.

All general BPF questions, including kernel functionality, libbpf APIs and their application, should be sent to bpf@vger.kernel.org mailing list. You can [subscribe](#) to the mailing list search its [archive](#). Please search the archive before asking new questions. It very well might be that this was already addressed or answered before.