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_`, `xsk_`, `btf_dump_`, `ring_buffer_`, `perf_buffer_`.

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.

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`.

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`.

AF_XDP functions

AF_XDP functions should have an `xsk_` prefix, e.g. `xsk_umem__get_data` or `xsk_umem__create`. The interface consists of both low-level ring access functions and high-level configuration functions. These can be mixed and matched. Note that these functions are not reentrant for performance reasons.

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.

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.

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:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\libbpf\(linux-master)(Documentation)(bpf)(libbpf)libbpf_naming_convention.rst, line 111)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        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:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\libbpf\(linux-master)(Documentation)(bpf)(libbpf)libbpf_naming_convention.rst, line 124)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

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

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 formated.

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.

License

Links

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