libfx Documentation

Release 0.0.0

Kevin Leptons

Feb 20, 2018

Contents

CHAPTER 1

Part 1. Introduction

libfx is set of packages, an package provides documents and implementations for concepts which relates with computer programming. Documents are write by nature language and to be use as a way to get knowledge about concepts, also as guide to use implementations. Implementations are write by programming language and to be use as APIs to write programs, for specific programming language is C.

libfx describes concepts and give implementations under the **libfx** name to help programmers do thier job and do together. For example, **libfx** provides vector, hash table, thread pool or TCP handling model. Unlike in the case use other packages with different styles, **libfx** is unique clauses, it saves resources such as time, money and working. Programmers can be student, researcher, developer or any one, they can use or contribute to **libfx** if they have base knowledge about computer programming.

Packages aims to both hosted and freestanding environments.If package runs on hosted, it must be independent between hosts, called cross platform. So some packages can runs on embeded system, some packages must be runs on kernel of operating system. Let consider carefully destination environments while create new packages or use early existed packages.

Implementations by C language because C language is freedom. That mean C language provides essential concepts and let programmers decides concepts what they need. There are few other reasons but freedom is most important. Compare with other language such as C++: C++ provides three concepts are OOP, RAII, and exception together, if disable one of them to use alternative concepts then C++ is not C++. Or with Python, it hides raw memory concepts so programmers can not do work with memory exactly.

libfx exchanges freedom by hard working. Because C language provides essential concepts, more work to, it is result and can not be avoid completely. If some one need do work quickly, please consider other solutions.

Here are fork, Go to *Part 2. Specification* to use or *Part 3. Development* to contribute **libfx**.

CHAPTER 2

Part 2. Specification

2.1 libfx_inf

libfx_inf - fx infrastructure library

C is freedom programming language. C language allows programmers do every things which hardwares can. It is powerfull, also weakness because programmers can go to correct ways or wrong ways. And C language lacks few of essential concepts such as coding standard or error handling, C language lets programmers to decide that. If you are a C programmer then you know what is that when you deal with error handling model, allocate and release resources. There are many ways to do with pros/cons but you do not know what should you choice. It is wasted time, stress and painful.

This package provides solution to deal with freedom in C language. Package contains both specifications and APIs to do that and used as infrastructure for any fx's package. You can read detail concepts in next sections.

Let contribute this package. This package may be one in many ways to deal with freedom in C language. It may be good way, may be bad way but it is one way. If you think you can do it better, just let we know because we know that some things are not good is exists in this package and we are not satisfied with it but we can not find better way.

Here are concepts:

2.1.1 fx_file_system_style

NAME

fx_file_system_style - fx file system style

SYNOPSIS

```
[a-z][a-z0-9_]*\.[h|c]          // for C header and source files
[a-z][a-z0-9_]*                 // for other files, directories
```

```
[a-z][a-z0-9_]*_d[0-9]{4}       // year postfix
[a-z][a-z0-9_]*_d[0-9]{6}       // year and month postfix
[a-z][a-z0-9_]*_d[0-9]{8}       // year, month and day postfix
[a-z][a-z0-9_]*_t[0-9]{2}       // hour postfix
[a-z][a-z0-9_]*_t[0-9]{4}       // hour and minute postfix
[a-z][a-z0-9_]*_t[0-9]{6}       // hour, minute and second postfix
[a-z][a-z0-9_]*_d[0-9]{14}      // year, month, day, hour, minute
                                // and second postfix

.+                              // for special files
```

DESCRIPTION

Just lower case, numbers and underscore character are allowed, it is enough to describe something and still simple enough to read or write. Lower case and numbers characters are used to describe meaning, underscore character is used to divide part of name.

Dash character is easy to write and read than underscore character but is is conflict with identity naming in programs. Then underscore character is selected instead of dash character to have unified naming for both file system and identities in programs.

File names with date and time postfix are use for files which have same name but different created, modified time or it's content is in different date time.

Exception for special file names. There are file which uses by programs and using other file names require more cost such as time, documentation or habits. So it is not requires to compliant naming in this case.

EXAMPLE 01

```
// target: valid file names

error.h                                 // C header
ring_queue.h                            // C header with multi parts
ring_queue.c                            // C source

report_d2018.txt                        // report in 2018
report_d201802.txt                      // report in Feb 2018
report_d20180201.txt                    // report in Feb 01 2018

log_t23.txt                             // log at 23
log_t2305.txt                           // log at 23:05
log_t230509.txt                         // log at 23:05:09

backup_d20180201230509.txt              // backup in Feb 01 2018
                                        // at 23:05:08
```

EXAMPLE 02

```
// target: invalid file names

Error.h                                 // contain upper case
error.H
ringQueue.h
RingQueue.h
```

```
ring-queue.h                            // contain dash character
log-in-today.txt

log__in.txt                             // dash does not divide
_log_in.txt                             // anything
__log_in.txt
log_in_.txt
```

EXAMPLE 03

```
// target: valid file names for special using

Makefile                                // contain upper cases but
                                        // it is uses by "make"

CMakeLists.txt                          // containse upper cases but
                                        // it is uses by "cmake"
```

2.1.2 fx_identity_style

NAME

fx_identity_style - fx identity style

SYNOPSIS

```
[A-Z][A-Z0-9_]+                 // macros
[a-z][a-z0-9_]+                 // every things is not macros:
                                // constants, variables, functions
                                // structure, enum, union

Keep identity name as short as possible but still not too hard to
understand.
```

DESCRIPTION

Macro must be all upper case characters. Because macro's behaviors are different than C code, it is important to distinguish macro and C code.

Other things are not macro must be lower case characters. Lower cases are more quick to write and more easy to read than lower interleave upper case characters.

Keep identity name as short as possible but still not too hard to understand. This style does not specify minimum or maximum lenght of identity name because that is unreasonable, sometime long names are not avoidable, sometime short names are efficiency. Long names may be easy to understand but it cause too long lines and take long time to read. Short name enough is harmonious between understandable and readable.

EXAMPLE 01

```
// target: valid macro's name

#define FX_ERROR_H
#define FX_YES 1
#define FX_NO 0
#define FX_COE 0x1234567A
```

EXAMPLE 02

```
// target: invalid macro's name

#define fx_error_h              // not upper case
#define FX_ERROR_h              // not upper case completly
#define _FX_ERROR_H             // _ prefix are reserved names
#define __FX_ERROR_H            // __ prefix are reserved names
```

EXAMPLE 03

```
// target: valid identity names

unsigned int year;
unsigned char month;
unsigned char day_in_month;
unsigned char day_in_week;
const double pi = 3.14;

struct date {};
void now_date(struct date *d);
```

EXAMPLE 04

```
// target: invalid identity names

unsigned int _year;             // _ prefix are reserved names
unsigned int __year;            // __ prefix are reserved names

unsigned int Year;              // contain upper case
struct Date {};
void Now_date(struct Date *d);
const double PI = 3.14;
```

EXAMPLE 05

```
// target: wrong identity names - too long

const double gravity_constant = 9.8;    // g is enough
                                         // any one work with physical
                                         // theory know what is g
```

```
                                        // if not, they should
                                        // research to know

struct input_output_type {};            // io_type is engouh
                                        // any one know what is io

// too long name: interator, i is engough
for (size_t interator = 0; interator < 8; ++interator) {}
```

EXAMPLE 06

```
// target: wrong identity names - too short

double s(double x) {                    // too short to understand
        return x * x;
}

double a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
double b[8];

for (size_t i = 0; i < 8; ++i)          // we will look for s() to see
        b[i] = s(a[i]);                 // what is it doing because
                                        // we can not know from it
                                        // name, it is wasted time
```

2.1.3 fx_comment_style

NAME

fx_comment_style - fx comment style

SYNOPSIS

```
With header or source files description, use multi line comment in
top of file:

        /*
                describe here
                more description here
        */

        void do_some_thing();
        void do_some_thing_more();

With function description, use multi comment in bottom side:

        void do_some_thing();
        /*
                describe about function here
                and more here
        */

With statement, use one line comment in right side if it fits:
```

```
        do_some_thing();        // describe here
                                // more description here

With statement, use one line comment in top side if right side does
not fit:

        // describe here
        do_some_thing_long_and_long_and_so_long();

Do not abuse comments.

Keep comments to be simple.

Do not add global information to header or source files comment.
```

DESCRIPTION

Do not abuse comments. Source code itself can explain what is it do, almost but not always. Then comment helps to explain some things too hard to get from source code. If comments become too much, it is terrible to read duplicate contents.

Keep comments to be simple. Comments helps something becomes more simple to understand, not more complicate. So keep it short, clear, non duplicate content.

Do not add global information to header or source files comment. That information can be change any time then you must update all of files which related. Other hand, you can put that information to one file, it is more easy to find, read and update. For example: lincenses, copyright.

EXAMPLE 01

```
// target: comment for header files

/*
        fx_array: array's interface - an abstract data type.

        fx_array_t store information about array.
        fx_array_init() initialize an array
        fx_array_free() release resources which uses by array
        fx_array_set() set a value into array at specific index
        fx_array_get() get a value from array at specific index
*/

typedef struct {} fx_array_t;

void fx_array_init(fx_array_t *a);
void fx_array_free(fx_array_t *a);
void fx_array_set(fx_array_t *a, size_t index, void *item);
void *fx_array_get(fx_array_t *a, size_t index);
```

EXAMPLE 02

```
// target: comment on one line in right side

const unsigned double g = 9.8;          // gravity constant
const unsigned double l = 299792458;    // speed of light
```

EXAMPLE 03

```
// target: comment for functions

struct qe2_result {
        bool has_roots;
        double x1;
        double x2;
};

void solve_qe2(double a, double b, double c, struct qe2_result *r);
/*
        Solve quadratic equation 2: y = a*x^2 + b*x + c.

        If a is zero, undefined behiviors.  If equation has root,
        r->has_roots is set to true and x1, x2 is set to corresponding
        values. If not, r->has_roots is set to false and x1, x2 is not
        change.
*/
```

2.1.4 fx_curly_bracket_style

NAME

fx_curly_bracket_style - fx curly brackets style

SYNOPSIS

```
Open and close curly brackets in same line if it is fits in one line.

Open curly brackets in same line and close curly brackets in next
line if it is not fits on one line.
```

DESCRIPTION

Open and close curly brackets in same line if it is fit. It saves space.

Open curly brackets in same line and close curly brackets in next line. It saves space and it is unified rule for open-close blocks. Maybe open block in next line for function is more easy to read but it is conflicts with rest of components such as `struct`, `enum`, so just pick one rule and it becomes unified, mean that easy to use.

EXAMPLE 01

```
// target:: open and close curly brackets on one line

enum week_day {mon, tue, wed, thus, fri, sat, sun};

unsigned char odd_numbers[] = {1, 3, 5, 7, 9};
```

EXAMPLE 02

```
// target: open and close curly brackets on multi line

struct qe2_result {
        bool has_roots;
        double x1;
        double x2;
};

void solve_qe2(double a, double b, double c, struct qe2_result *r) {

}

for (unsigned int i = 0; i < 10; ++i) {

}
```

2.1.5 fx_page_style

NAME

fx_page_style - fx page style

SYNOPSIS

```
Maximum characters per line is 78.

Indentation is 8 spaces.
```

DESCRIPTION

Maximum characters per line is 78. First, it is historical. Terminal devices can display 80 characters per line, if you need text work well on that devices, you must limit at 78 characters after you spend one character for enter line and one character for indentation with right edge screen. Second, it is printable. With 78 characters per line, it is able to print to A4 paper which popular paper size. Third, reader no need to scroll screen to see what is happens out of 80 characters wide.

Indentation is 8 spaces. First, it helps reader to distinguish indentation clearly and warns about too much nested blocks. Second, it helps printing to A4 paper correctly, not affected by custom tab size.

EXAMPLE 01

```
// target: indentation by 8 spaces

**************************************
        **********************************
        **********************************
        **********************************
        **********************************
<------> 8 space

**************************************
**************************************
**************************************
**************************************
        **********************************
        **********************************
<------> 8 space
```

REFERENCES

Character per line - Wikipedia

2.1.6 fx_loop_style

NAME

fx_loop_style - fx loop style

SYNOPSIS

```
Use for() for all of loop.
```

DESCRIPTION

Use `for()` for all of loop. C language provides three loop statements: `for()`, `while-do` and `do-while`, people are fucking crazy to research it, use it and read it. Use only `for()` statement is unified to stop care about other loop statements.

EXAMPLE 01

```
// target: finite loop

int array[8] = {1, 2, 3, 4, 5, 6, 7, 8};
for (size_t i = 0; i < 8; ++i) {

}
```

EXAMPLE 02

```
// target: infinite loop

for (;;) {

}
```

EXAMPLE 03

```
// target: do some things first then check loop condition


for (bool still_loop = true;;) {        // verify loop condition here

                                        // do some things to change
                                        // loop condition here

        if (!still_loop)                // check loop condition here

                break;
}
```

EXAMPLE 04

```
// target: change loop condition in other statements


for (bool still_loop = true; still_loop;) {     // verify loop
                                                // condition here

                                                // change loop
                                                // condition here
}
```

2.1.7 fx_hide_info_concept

NAME

fx_hide_info_concept - fx hide information concept

SYNOPSIS

```
typedef struct {} <name>_t;     // for private type
struct <name> {};               // for public type

show interfaces                 // functions, data types
hide implementation
```

DESCRIPTION

Hide information mean do not allow to access to unnecessary things. That appears when you design function's interface and implement it, then you show interface and hide any thing of implementation, that is hide information.

First advantage, help programmers focuses on things which they can do. With specifications of interface, programmers knows what is provides by interfaces exactly, then they can learn it in right way. If any things are show, programmers does not know where to start and follow what ways, that like a find way in a maze.

Second advantage, prevent wrong behaviors from programmers. Programers who use interfaces have not enough knowledge about implementation and they also do not want to know that. If they do some things with implementation then that behaviors may be wrong. Do not allow programmers to work with implementation avoid that problems.

Third advantage, keep interface is stable while implement changes freely. Because no one touch to implementation, change it will not affect to where uses interfaces. If you have new idea to make implementation better, just do it and interfaces still work well.

Use `typedef struct {} <name>_t` to define private data types. Private data types does not allows to access to member's fields directly, except functions in implementation. So when you see a type name with format `<name>_t`, do not access to it's members. Although `<name>_t` is POSIX's reserved name, we can not find any a better way to do that so we still use that format. Remember that we always put prefix before name such as `fx_<name>-t` so conflict will not happen.

Use `struct <name> {}` to define public data types. Public data types allows to access to data's fields freely.

EXAMPLE 01

```
// target: show interface, hide implementation and use interface

// in <yourlib/fx_array.h>
// show interface: data type and functions
typedef struct {} fx_array_t;
void fx_array_init(fx_array_t *a, size_t cap);
void fx_array_free(fx_array_t *a);
void fx_array_set(fx_array_t *a, size_t index, void *item);
void *fx_array_get(fx_array_t *a, size_t index);

// in <yourlib/fx_array.c>
// hide implementation, also define it
void fx_array_init(fx_array_t *a, size_t cap) {}
void fx_array_free(fx_array_t *a) {}
void fx_array_set(fx_array_t *a, size_t index, void *item) {}
void *fx_array_get(fx_array_t *a, size_t index) {}

// in <somewhere/main.c>
// use interface
#include <yourlib/fx_array.h>

fx_array_t v;
int data = 64;
int *item;

fx_array_init(&a, 128);
fx_array_set(&a, 0, &data);
item = (int *) fx_array_get(&a, 0);
```

EXAMPLE 02

```
// target: wrong access to private data type

typedef struct {                // define private type
        size_t cap;
        void **slots;
} fx_array_t;

int item = 8;
fx_array_t a;                   // define a instance
a.slots[0] = (void *) &item;    // do not do this

fx_array_set(&a, 0, &item);     // let interface do that
```

EXAMPLE 03

```
// target: work with public data type

// define interface: public data types and function
struct qe2_input {
        double a;
        double b;
        double c
};

struct qe2_output {
        bool has_roots;
        double x1;
        double x2;
};

void solve_qe2(struct qe2_input &in, struct qe2_output &out) {}

// use interface
struct qe2_input in;
struct qe2_output out;

in.a = 3;                       // set members freely
in.b = 4;
in.c = 5;
solve_qe2(&in, &out);

if (out.has_roots) {            // get members freely
        out.x1;
        out.x2;
}

/*
        That this, public data types occurs when you need to pass
        input arguments or get output data
*/
```

REFERENCES

IEEE Std 1003.1-2008 POSIX - Portable Operating System Interface

2.1.8 fx_error_concept

TITLE

fx_error_concept - fx error concept

SCOPE

This section specify error and error handling. That specifications apply for errors in general programming, not only for specific programming language. In specific programming language, it must implement follow specifications. Examples are write in pseudo code.

CONTEXT

1. `TASK` is object can be execute.
2. `WORKER` is object which execute tasks.
3. `ERROR` is a case when worker can not complete `TASK`.
4. `INTERNAL_ERROR` is `ERROR` which is causes by programmers.
5. `EXTERNAL_ERROR` is `ERROR` which is causes by non programmers.
6. `ERROR_IDENTITY` is information represents for `ERROR`.
7. `NONE_ERROR` is special `ERROR_IDENTITY` which represents no `ERROR` occurs.
8. `ERROR_STATE` is storage which uses to store `ERROR_IDENTITY` which occurs.
9. `DISCOVER_ERROR` mean find `ERROR` can be occurs while execute set of `TASK`.
10. `DEFINE_ERROR` mean that create `ERROR_IDENTITY` correspond with `ERROR`.
11. `DETECT_ERROR` mean that detect `TASK` will causes `ERROR`.
12. `RAISE_ERROR` mean set `ERROR_STATE` by specific `ERROR_IDENTITY`.
13. `CATCH_ERROR` mean test `ERROR_STATE` is set by specific `ERROR_IDENTITY`.
14. `SOLVE_ERROR` mean try other `TASK` to get target result.
15. `ERROR_HANDLING` is: `DISCOVER_ERROR`, `DEFINE_ERROR`, `DETECT_ERROR`, `RAISE_ERROR`, `CATCH_ERROR` and `SOLVE_ERROR`.

CLAUSES

1. **MUST**: Provide mechanism to `DEFINE_ERROR`, `DETECT_ERROR`, `RAISE_ERROR`, `CATCH_ERROR` and `SOLVE_ERROR`.
2. **MUST**: Provide `NONE_ERROR`.
3. **MUST**: Provide a `ERROR_STATE` for a `WORKER`.
4. **MUST**: Set `ERROR_STATE` by `NONE_ERROR` on `WORKER` starting.

5. **MUST**: Define all of `EXTERNAL_ERROR`.
6. **MUST**: `INTERNAL_ERROR` must be remove.
7. **MUST**: `EXTERNAL_ERROR` should be solve.
8. **MUST**: Do `RAISE_ERROR` if `EXTERNAL_ERROR` can not be solve.
9. **MUST**: Abort process if `EXTERNAL_ERROR` can not be solve ro do `RAISE_ERROR`.
10. **MUST NOT**: Do `ERROR_HANDLING` with `INTERNAL_ERROR`.
11. **MUST NOT**: Define `ERROR_IDENTITY` duplication.

EXAMPLE 01

```
// target: ERROR

devide 8 by 0.
access to 8th index of array have 7 items.
open not early exist file to read.
write to file cause full of disk storage.
connect to not early exist IP address.
play broken video file.
```

EXAMPLE 02

```
// target: INTERNAL_ERROR

array a[8] = [1, 2, 3, 4, 5, 6, 7, 8]
array b[8]

for i in [0, 8]:                        // on i = 8
        b = 2 * a[i]                    // access to 9th item
                                        // but 9th item is not exist
                                        // mean internal error occurs
```

EXAMPLE 03

```
// target: EXTERNAL_ERROR

array a[] = allocate 2.10^9 bytes       // if computer have not
                                        // enough 2.10^9 free bytes
                                        // mean external error occurs
```

EXAMPLE 04

```
// target: INTERNAL_ERROR must be remove

// back to EXAMPLE 01, change code to:

array a[8] = [1, 2, 3, 4, 5, 6, 7, 8]
array b[8]
```

```
for i in [0, 7]:                        // fixed
        b = 2 * a[i]
```

EXAMPLE 04

```
// target: EXTERNAL_ERROR should be solve

// back to EXAMPLE 02, change code to:

array a[] = allocate 2.10^9 bytes       // error occurs
if can not allocate for a[]             // then it is detect
        try other way                   // and solve
else
        continue to task                // no error occurs, do normal
```

EXAMPLE 05

```
// target: Do ``RAISE_ERROR`` if ``EXTERNAL_ERROR`` can not be solve.

array a[] = allocate 2.10^9 bytes       // error occurs
if can not allocate for a[]             // then it is detect
        set error state by full memory  // and raise
else
        continue to task                // no error occurs, do normal
```

EXAMPLE 06

MUST: Abort process if `EXTERNAL_ERROR` can not be solve ro do `RAISE_ERROR`.

```
array a[] = allocate 2.10^9 bytes       // error occurs
if can not allocate for a[]             // then it is detect
        abort process                   // but can not solve
                                        // and no parent task
                                        // abort process
else
        continue to task                // no error occurs, do normal
```

2.1.9 [+] fx_error_core

NAME

fx_error_core - fx error handling mechanism

SYNOPSIS

```
#include <fx_error/error.h>

#define FX_ERROR_DCL(error_id)
#define FX_ERROR_DEF(error_id)
```

```
typedef struct {} fx_error_t;
extern FX_ERROR_STATE_ATTR const fx_error_t *fx_error_state;

void fx_error_raise(const fx_error_t *e);
bool fx_error_catch(const fx_error_t *e);
bool fx_error_check(void);
void fx_error_clear(void);
const char *fx_error_id(const fx_error_t *e);
```

DESCRIBE

This APIs is an implementation of *fx_error_concept* called `fx_error`. `fx_error` use for all of parts of `libfx`. New code aim to part of `libfx` must be `fx_error` compliant. Code depends on `libfx` should be `fx_error` compliant to handle errors unified. This implementation also specify three more rules for compative with C language:

Comfort with C standard library. That library is specifics in ISO-IEC 9898, means APIs requires hosted environment to runs and does not guarantees working on freestanding environment.

Minimize error definition size. Number of errors are causes by hardware resources is finite, however invalid data errors is infinite, it appears when you create new data types. If error definition size is not minimizes, it will be big problem.

A error state for a thread. This rule guarantees that errors occurs in a thread is not affect to other threads.

EXAMPLE 01

```
// target: declare, define errors

// in <yourlib/error.h>
#include <fx_error/error.h>     // use fx_error APIs

FX_ERROR_DCL(YOURLIB_E001);     // declare errors which may be occurs
FX_ERROR_DCL(YOURLIB_E002);     // in your library
FX_ERROR_DCL(YOURLIB_E003);

// in <yourlib/error.c>         // define errors
FX_ERROR_DEF(YOURLIB_E001);
FX_ERROR_DEF(YOURLIB_E002);
FX_ERROR_DEF(YOURLIB_E003);

// in <yourlib/api.h> or <yourlib/api.c>
// you can use YOURLIB_E001 with fx_error APIs
```

EXAMPLE 02

```
// target: raise, check, handle error

# include <yourlib/error.h>     // use your defined errors

// in <yourlib/do_work.h>
void do_work(void);             // describe why YOURLIB_E001 will
                                // occurs to help other people who
                                // uses your APIs check errors
```

```
// in <yourlib/do_work.c>
void do_work(void) {
        if (...) {                              // error condition
                fx_error_raise(YOURLIB_E001);   // raise error
                return;                         // stop function
        }
}

// in <somewhere/do_other_work.c>
#include <yourlib/do_work.h>            // use your APIs

void do_other_work(void) {
        do_work();                      // call your API
        if (fx_error_check()) {         // check error
                                        // handle error
        }
}
```

EXAMPLE 03

```
// target: check, handle multi error cases

// in <yourlib/do_work.h>
void do_work(void);     // describe why YOURLIB_E001, YOURLIB_E002
                        // and YOURLIB_E003 will be occurs

// in <yourlib/do_work.c>
void do_work(void) {
        if (...) {                              // error condition 1
                fx_error_raise(YOURLIB_E001);   // raise error
                return;                         // stop function
        }
        if (...) {                              // the same above
                fx_error_raise(YOURLIB_E002);
                return;
        }
        if (...) {                              // the same above
                fx_error_raise(YOURLIB_E003);
                return;
        }
}

// in <somewhere/do_other_work.c>
#include <yourlib/do_work.h>            // use your APIs

void do_other_work(void) {
        do_work();                              // call API
        if (fx_error_catch(YOURLIB_E001)) {     // check and handle
                                                // for YOURLIB_E001
        }
        if (fx_error_catch(YOURLIB_E002)) {     // check and handle
                                                // for YOURLIB_E002
        }
        if (fx_error_catch(YOURLIB_E003)) {     // check and handle
                                                // for YOURLIB_E003
```

```
        }
}
```

2.2 libfx_error

2.2.1 [+] fx_error_core

NAME

fx_error_core - mechanism for error handling for libfx

SYNOPSIS

```
#include <fx_error/error.h>
// in libfx_error

#define FX_ERROR_DCL(error_id)
// declare error identity in header file

#define FX_ERROR_DEF(error_id)
// define error identity in source file

#define FX_ERROR_ASSERT(expression)
// verify at debug mode

void fx_error_raise(const fx_error_t *e);
// set fx_error_state to e

int fx_error_catch(const fx_error_t *e);
// check specific error has occurs

int fx_error_check(void);
// check error has occurs or not

void fx_error_clear(void);
// set fx_error_state to FX_ENONE

const char *fx_error_id(const fx_error_t *e);
// get short string which describe error
```

DESCRIBE

Specify rules to handling errors efficiency in C language, also provide APIs to do that. Everything in libfx, used libfx should follow that rules to make uniform error handling, together. There are rules:

Comfort with C standard library. That library is specifics in ISO-IEC 9898, means `libfx_error` requires hosted environment to runs and does not guarantees working on freestanding environment.

Minimize number of error definitions. It means define general errors and do not define new errors with same meaning. This rules make less definitions then less time spent to research and more less time to use.

Minimize error definition size. Number of errors are causes by hardware resources is finite, however logic errors is infinite, it appears when you create new things. If storage is not minimizes, it will be big problem. Do it with `FX_ERROR_DCL` and `FX_ERROR_DCL`.

Diagnostic developer's mistakes at debug mode. Errors which causes by developers must be fix at debug mode instead of give it to other developers at release mode. If that errors occurs, show essential information to help find, fix bugs then abort process. At release mode, diagnostic must be disapear to guarantee execution speed. Do it with `FX_ERROR_ASSERT()`.

Emit non developer's errors to caller. That errors may be failer in hardware resources allocating, I/0 operations or invalid data processing which can not be fix at debug mode. This rule gives caller two choices: handle it or emit to upper caller. Do it with `fx_error_raise()`.

A error state for a thread. This rule guarantees that errors occurs in a thread is not affect to other threads. To do that, each threads have a error state called `fx_error_state`.

EXAMPLE 01

```
// target: declare, define errors

// in <yourlib/error.h>
#include <fx_error/error.h>     // use fx_error APIs

FX_ERROR_DCL(YOURLIB_E001);     // declare errors which may be occurs
FX_ERROR_DCL(YOURLIB_E002);     // in your library
FX_ERROR_DCL(YOURLIB_E003);

// in <yourlib/error.c>         // define errors
FX_ERROR_DEF(YOURLIB_E001);
FX_ERROR_DEF(YOURLIB_E002);
FX_ERROR_DEF(YOURLIB_E003);

// in <yourlib/api.h> or <yourlib/api.c>
// you can use YOURLIB_E001 with fx_error APIs
```

EXAMPLE 02

```
// target: raise, check, handle error

# include <yourlib/error.h>     // use your defined errors

// in <yourlib/do_work.h>
void do_work(void);             // describe why YOURLIB_E001 will
                                // occurs to help other people who
                                // uses your APIs check errors

// in <yourlib/do_work.c>
void do_work(void) {
        if (...) {                              // error condition
                fx_error_raise(YOURLIB_E001);   // raise error
                return;                         // stop function
        }
}

// in <somewhere/do_other_work.c>
#include <yourlib/do_work.h>            // use your APIs
```

```
void do_other_work(void) {
        do_work();                      // call your API
        if (fx_error_check()) {         // check error
                                        // handle error
        }
}
```

EXAMPLE 03

```
// target: raise error then abort process

// in <yourlib/do_work_abort.h>
void do_work_abort();           // describe why YOURLIB_E001 will
                                // be occurs and process will be
                                // terminate also

// in <yourlib/do_work_abort.c>
void do_work_abort(int arg) {
        if (...)                                    // error condition
                fx_error_abort(YOURLIB_E001);       // raise error then
                                                    // abort
}

// in <somewhere/do_other_work.c>
#include <yourlib/do_work.h>    // use your APIs

void do_other_work(void) {
        do_work_abort();        // call API
                                // if error occurs, error state will
                                // be set then terminate process with
                                // SIGABRT
}
```

EXAMPLE 04

```
// target: check, handle multi error cases

// in <yourlib/do_work.h>
void do_work(void);     // describe why YOURLIB_E001, YOURLIB_E002
                        // and YOURLIB_E003 will be occurs

// in <yourlib/do_work.c>
void do_work(void) {
        if (...) {                                  // error condition 1
                fx_error_raise(YOURLIB_E001);       // raise error
                return;                             // stop function
        }
        if (...) {                                  // the same above
                fx_error_raise(YOURLIB_E002);
                return;
        }
        if (...) {                                  // the same above
                fx_error_raise(YOURLIB_E003);
```

```
                return;
        }
}

// in <somewhere/do_other_work.c>
#include <yourlib/do_work.h>            // use your APIs

void do_other_work(void) {
        do_work();                                  // call API
        if (fx_error_catch(YOURLIB_E001)) {         // check and handle
                                                    // for YOURLIB_E001
        }
        if (fx_error_catch(YOURLIB_E002)) {         // check and handle
                                                    // for YOURLIB_E002
        }
        if (fx_error_catch(YOURLIB_E003)) {         // check and handle
                                                    // for YOURLIB_E003
        }
}
```

2.2.2 [+] fx_error_kernel

SYNOPSIS

```
#include <fx_error/kernel.h>

const struct fx_error *fx_error_from_kernel(int code);
// return fx_error_t * from kernel error code

void fx_error_kraise(int code);
// raise fx_error from kernel error code

// pre-defined errors corespond with kernel error codes
// for example: EPERM corespond with FX_EPERM
```

2.2.3 [+] fx_error_error

SYNOPSIS

```
#include <fx_error/error.h>

FX_ERROR_DCL(FX_EINDEX);        // invalid index accessing
FX_ERROR_DCL(FX_EKEY);          // invalid key accessing
FX_ERROR_DCL(FX_EFULL);         // full storage
```

2.3 libfx_test

2.3.1 [+] fx_test

SYNOPSIS

```
#include <fx_test/test.h>

void fx_test_init(fx_test_t *t, enum fx_test_mode mode);
// initialize testing

void fx_test_free(fx_test_t *t);
// release resources which uses by testing

void fx_test_run(fx_test_t *t, const fx_test_unit_set_t *s);
// run unit tests in parallel

void fx_test_join(fx_test_t *t, fx_test_result_t *r);
// wait for all of unit tests finished

void fx_test_result_dump(FILE *out, fx_test_result_t *r);
// print testing's result
```

2.3.2 [-] fx_test_queue

NAME

queue - ring queue.

SYNOPSIS

```
#include <fx_test/queue.h>
// in libfx_test

void fx_test_queue_init(fx_test_queue_t *q, size_t cap);
// initialize queue

void fx_test_queue_free(fx_test_queue_t *q);
// release resources which uses by queue

void fx_test_queue_push(fx_test_queue_t *q, void *item);
// add item into back of queue

void *fx_test_queue_front(fx_test_queue_t *q);
// get item in front of queue

void fx_test_queue_pop(fx_test_queue_t *q);
// remove an item in front of queue

void fx_test_queue_clear(fx_test_queue_t *q);
// make queue is empty

bool fx_test_queue_empty(fx_test_queue_t *q);
```

```
// check queue is empty or not

bool fx_test_queue_full(fx_test_queue_t *q);
// check queue is full or not
```

DESCRIPTION

Queue with fixed capability at initialization and can not change capability after that. Only one calling to memory allocator by `fx_test_queue_init()`.

ERROR

`fx_test_queue_init()`

- `FX_ENOMEM` not enough memory to allocate

UNDEFINED BEHAVIORS

`fx_test_queue_init()`

- **q** points to memory block which uses by other objects
- **q** points to memory block which is smaller than `fx_test_queue_t`

`fx_test_queue_free()`, `fx_test_queue_push()`, `fx_test_queue_front()`, `fx_test_queue_pop()`, `fx_test_queue_clear()`, `fx_test_queue_empty()`, `fx_test_queue_full()`,

- **q** is invalid to pass to `fx_test_queue_init()`
- **q** is not initialize by `fx_test_queue_init()`

`fx_test_queue_push()`

- **q** is full

`fx_test_queue_front()`, `fx_test_queue_pop()`

- **q** is empty

BUGS

EXAMPLE 01

```
// target: initialize, push, front, pop and free queue

#include <fx_test/queue.h>              // use queue APIs

fx_test_queue_t q;                      // initialize
fx_test_queue_init(&q, 128);

int one = 1;
int two = 2;
int three = 3;
fx_test_queue_push(&q, &one);           // push items to front
fx_test_queue_push(&q, &two);
```

```
fx_test_queue_push(&q, &three);

int *item;
item = fx_test_queue_front(&q);          // get front item
fx_test_queue_pop(&q);                   // remove front item

fx_test_free(&q);                        // release resources
```

CHAPTER 3

Part 3. Development

3.1 Tutorial

```
sudo apt-get install make       # to run every things automatically
make config                     # install dependency packages

make build                      # build every things
make doc-open                   # open document in browser

make test                       # test every things
make install                    # install every things to system

make pack                       # create distribution files
```

That are basic commands help you build, test and pack library. To use all of development commands, you need research about project in next sections.

3.2 File system structure

```
|
|-doc                           # document source
|-include                       # prorotype
| |-fx_error
| |-fx_test
| |-fx
|
|-src                           # source code
| |-fx_error                    # error handling library
| |-fx_test                     # testing library
| |-fx                          # data and algorithm library
|
|-dest
```

```
| |-doc
| | |-text                          # plain text document
| | |-html                          # web document
| |
| |-lib                             # output of library files
| |-bin                             # outout of executable files
|
```

3.3 Create a package

3.3.1 Create document

3.3.2 Create implementations