NAME

xcpc_set_exitproc, xcpc_push_tryctx, xcpc_pop_tryctx, xcpc_do_throw, xcpc_do_rethrow, xcpc_context_create, xcpc_context_reparent, xcpc_context_free, xcpc_context_parent, xcpc_context_root, xcpc_context_exhandler, xcpc_resource_add, xcpc_resource_movehead, xcpc_resource_movetail, xcpc_resource_moveafter, xcpc_resource_movebefore, xcpc_resource_ctx, xcpc_resource_data, xcpc_resource_del, xcpc_resource_reparent, xcpc_resource_set

XCPC_THROW, XCPC_RETHROW, XCPC_TRY, XCPC_CATCH, XCPC_CATCH_ANY, XCPC_END_TRY, XCPC_RETURNV, XCPC_RETURN, XCPC_EXCEPT_DATA, XCPC_EXCEPTION

xcpc__malloc, xcpc__realloc, xcpc__strdup, xcpc__fopen, xcpc__fdopen, xcpc__freopen, xcpc__open, xcpc__push_remove, xcpc__mmap, xcpc__write, xcpc__read, xcpc__fwrite, xcpc__fread, xcpc__stat, xcpc__fstat, xcpc__mkdir, xcpc__rmdir, xcpc__remove, xcpc__opendir, xcpc__pipe, xcpc__socket, xcpc__bind, xcpc__connect, xcpc__truncate, xcpc__ftruncate, xcpc__lseek

SYNOPSIS

Core API

```
#include <libxcpc.h>


void xcpc_set_exitproc(void (*proc)(int));
void xcpc_push_tryctx(xcpc_ctx ctx, xcpc_tryctx *tctx);
xcpc_tryctx *xcpc_pop_tryctx(xcpc_ctx ctx);
void xcpc_do_throw(xcpc_ctx ctx, int exno, void *data);
void xcpc_do_rethrow(xcpc_ctx ctx);
xcpc_ctx xcpc_context_create(xcpc_ctx pctx);
void xcpc_context_reparent(xcpc_ctx ctx, xcpc_ctx pctx);
void xcpc_context_free(xcpc_ctx ctx);
xcpc_ctx xcpc_context_parent(xcpc_ctx ctx);
xcpc_ctx xcpc_context_root(xcpc_ctx ctx);
xcpc_ctx xcpc_context_exhandler(xcpc_ctx ctx);
xcpc_res xcpc_resource_add(xcpc_ctx ctx, void *data,
              void (* free)(void *));
void xcpc_resource_movehead(xcpc_res res);
void xcpc_resource_movetail(xcpc_res res);
void xcpc_resource_moveafter(xcpc_res res, xcpc_res rres);
void xcpc_resource_movebefore(xcpc_res res, xcpc_res rres);’
xcpc_ctx xcpc_resource_ctx(xcpc_res res);
void *xcpc_resource_data(xcpc_res res);
void xcpc_resource_del(xcpc_res res, int do_free);
void xcpc_resource_reparent(xcpc_res res, xcpc_ctx pctx);
void xcpc_resource_set(xcpc_res res, int do_free, void *data,
             void (* free)(void *));
XCPC_THROW(ctx, exno, data)
XCPC_RETHROW(ctx)
XCPC_TRY(ctx)
XCPC_CATCH(exno)
XCPC_CATCH_ANY
XCPC_END_TRY
XCPC_RETURNV(expr)
XCPC_RETURN
XCPC_EXCEPT_DATA
XCPC_EXCEPTION
```

Function wrappers

#include <libxcpc.h>

```
void *xcpc__malloc(xcpc_ctx ctx, xcpc_res *pres, long size);
void *xcpc__realloc(xcpc_res res, long size);
char *xcpc__strdup(xcpc_ctx ctx, xcpc_res *pres, char const *str);
FILE *xcpc__fopen(xcpc_ctx ctx, xcpc_res *pres, char const *path,
            char const *mode);
FILE *xcpc__fdopen(xcpc_ctx ctx, xcpc_res *pres, int fd,
            char const *mode);
FILE *xcpc__freopen(xcpc_res res, char const *path, char const *mode);
int xcpc__open(xcpc_ctx ctx, xcpc_res *pres, char const *path,
          int flags, int mode);
void xcpc__push_remove(xcpc_ctx ctx, xcpc_res *pres, char const *path);
void *xcpc__mmap(xcpc_ctx ctx, xcpc_res *pres, void *start, size_t length,
            int prot, int flags, int fd, off_t offset);
ssize_t xcpc__write(xcpc_ctx ctx, int fd, const void *buf, size_t count);
ssize_t xcpc__read(xcpc_ctx ctx, int fd, void *buf, size_t count);
size_t xcpc__fwrite(xcpc_ctx ctx, const void *ptr, size_t size,
             size_t nmemb, FILE *stream);
size_t xcpc__fread(xcpc_ctx ctx, void *ptr, size_t size, size_t nmemb,
            FILE *stream);
void xcpc__stat(xcpc_ctx ctx, const char *path, struct stat *buf);
void xcpc__fstat(xcpc_ctx ctx, int fd, struct stat *buf);
void xcpc__mkdir(xcpc_ctx ctx, const char *path, mode_t mode);
void xcpc__rmdir(xcpc_ctx ctx, const char *path);
void xcpc__remove(xcpc_ctx ctx, const char *path);
void *xcpc__opendir(xcpc_ctx ctx, const char *path);
int xcpc__socket(xcpc_ctx ctx, int domain, int type, int protocol);
void xcpc__pipe(xcpc_ctx ctx, int * fds);
void xcpc__bind(xcpc_ctx ctx, int sfd, const struct sockaddr *addr, int alen);
void xcpc__connect(xcpc_ctx ctx, int sfd, const struct sockaddr *addr,
            int alen);
void xcpc__truncate(xcpc_ctx ctx, char const *path, off_t len);
void xcpc__ftruncate(xcpc_ctx ctx, int fd, off_t len);
off_t xcpc__lseek(xcpc_ctx ctx, int fd, off_t off, int whence);
```

DESCRIPTION

The **libxcpc** library implements an automatic resource cleanup and exception handling in **C**. Error handling and cleanup code (at least for code that **does** error handling) can take quite some space in terms of lines of code to be written. This not only increases the binary size, but makes the code more difficult to read. Furthermore, error handling code typically lies in untested code paths (because they get exercised only when abnormal conditions happens) that are very much error prone. Effective and complete code coverage tests to exercise all error paths, are not trivial to implement, and very often many of such paths are simply left untested. **C++** has native exception handling, and this solves part of the problem, while keeping exposed the resource cleanup one. The **libxcpc** offers **C++** like exception handling, plus automatic resource cleanup, to be used in software written in **C**. The **libxcpc** introduces three abstractions, that are *Resource*, *Container* and *Exception*. The *Resource* is every object (or action) that needs cleanup. This can be a block of allocated memory, an open file, a mapped memory region, etc... Every allocated *Resource* is owned by a *Container*. A *Resource* can be moved from a *Container* to another. A *Container* is a bucket inside which *Resource* are allocated. A *Container* can be the parent of other *Container*s, by hence introducing a parent/child relationship between *Container*s. A *Container* can be reassigned to be child of a new *Container*. By freeing a *Container* all the *Resource*s allocated inside the *Container* will be freed, and all the child *Container*s will be recursively freed too. *Resource Container*s greatly simplify the resource cleanup code, by being able to issue a single call to free a *Container* and having automatically all the *Resource*s contained by it, freed as well. *Resource*s are added/removed in a **LIFO** (Last In First Out) way, and the **libxcpc** library offers APIs to re-arrange the order of the *Resource*s inside their *Context*s. An *Exception* is (like in **C++**) any kind of abnormal condition that prevent the program to flow in its path. This can be a failed memory allocation, a failure to open a file, a failure to **mmap**(2) a portion of a file, a failure to **write**(2) a file, etc... An *Exception* is described by a unique number (*int*) and by an associated data (*void* *). *Exception*s numbers from 1 to (**XCPCE_USER**-1) are reserved to the **libxcpc** library, while the user can define their own starting from **XCPCE_USER**. *Exception*s are thrown using the **XCPC_THROW(***ctx***,** *exno***,** *data***)** statement, or re-thrown using **XCPC_RETHROW(***ctx***).** A nice feature of exception handling, is that it allows you to handle only certain kind of exceptions, and different exceptions in different points of your code tree. Using **libxcpc** library, you handle exceptions by surrounding the potentially-throwing code with the **XCPC_TRY(***ctx***)** statement. The code can then use either the **XCPC_CATCH(***exno***)** or the **XCPC_CATCH_ANY** statements to handle specific or all kind of exceptions that happened in the code bound by the **XCPC_TRY(***ctx***).** An *Exception* block must be terminated by a **XCPC_END_TRY** statement. If the current *Exception* block does not handle the current *Exception* using the **XCPC_CATCH(***exno***)** statement, and does not have a **XCPC_CATCH_ANY** statement, the **libxcpc** library backtrack to find a valid handler in the code at higher layers of the call hierarchy. It is important that at least the outer *Exception* block uses a **XCPC_CATCH_ANY** statement, so that any *Exception*s not caught by the code, is handled properly. When an *Exception* is caught by an handler, all the resources allocated by the code from the beginning of the **XCPC_TRY(***ctx***)** block, down to the place where the *Exception* is thrown, are automatically freed once the handler reaches the **XCPC_END_TRY** statement.

Structures and Types

The following types are defined:

xcpc_ctx

The **xcpc_ctx** type rapresent a *Container*, by the means described above.

xcpc_res

The **xcpc_res** type rapresent a *Resource*, by the means described above.

xcpc_tryctx

The **xcpc_tryctx** is an internal type that the caller should not care about it, for normal **libxcpc** usage.

Functions and Macros (Core API)

The following functions are defined:

void xcpc_set_exitproc(void (**proc***)(int));**

Sets the exit function for the core **libxcpc** implementation. This is called when an exception has been thrown, and noone is handling it. It default on the **exit**(3) function, on system supporting it. Systems not supporting **exit**(3) should call **void xcpc_set_exitproc(void (****proc***)(int))** at the beginning of the program, before any other **libxcpc** is called.

void xcpc_push_tryctx(xcpc_ctx *ctx*, **xcpc_tryctx ****tctx***);**

This is an internal function that is used by the *Exception* macros. It pushes a new **xcpc_tryctx** context into the stack. Normal **libxcpc** should never have to call this function.

xcpc_tryctx *xcpc_pop_tryctx(xcpc_ctx *ctx***);**

Like the **void xcpc_push_tryctx(xcpc_ctx** *ctx*, **xcpc_tryctx ****tctx***)** function, **xcpc_tryctx *xcpc_pop_tryctx(xcpc_ctx** *ctx***)** is an internal function and should not be called by the normal user. This function removes the top of the *exception* stack from the stack itself.

void xcpc_do_throw(xcpc_ctx *ctx*, **int** *exno*, **void ****data***);**

This is the function that **XCPC_THROW(***ctx*, *exno*, *data***)** relies on to throw exceptions. The user should call **XCPC_THROW(***ctx*, *exno*, *data***)** instead of calling **void xcpc_do_throw(xcpc_ctx** *ctx*, **int** *exno*, **void ****data***)** directly.

void xcpc_do_rethrow(xcpc_ctx *ctx***);**

This is the function that **XCPC_RETHROW(***ctx***)** relies on to re-throw exceptions. The user should call **XCPC_RETHROW(***ctx***)** instead of calling **void xcpc_do_rethrow(xcpc_ctx** *ctx***)** directly.

xcpc_ctx xcpc_context_create(xcpc_ctx *pctx***);**

Creates a *Resource Container* from the parent *Container* passed in the *pctx* parameter. The **xcpc_ctx xcpc_context_create(xcpc_ctx** *pctx***)** function returns the new *Container* or throws an exception in case of errors.

void xcpc_context_reparent(xcpc_ctx *ctx*, **xcpc_ctx** *pctx***);**

As explained in the **DESCRIPTION** *Container*s form a hierarchy with each parent allowed to have many childs. The **void xcpc_context_reparent(xcpc_ctx** *ctx*, **xcpc_ctx** *pctx***)** function makes the *Context* passed in *ctx* a new child of the *Context* passed in *pctx*, by detaching *ctx* from its previous parent. A root *Context* (the one whose parent is **NULL**) cannot be re-parented and trying to do so, will generate an *Exception*. Also, *Context*s can be re-parented if they share the same root *Context*.

void xcpc_context_free(xcpc_ctx *ctx***);**

Frees the *Context* passed in the *ctx* parameter. The **void xcpc_context_free(xcpc_ctx** *ctx***)** function frees all the *Resource*s allocated inside *ctx* and also frees all the *Context s* that are rooted on *ctx*.

xcpc_ctx xcpc_context_parent(xcpc_ctx *ctx***);**

Returns the parent *Context* of *ctx*.

xcpc_ctx xcpc_context_root(xcpc_ctx *ctx***);**

Returns the root *Context* of the *ctx* dinasty.

xcpc_ctx xcpc_context_exhandler(xcpc_ctx *ctx***);**

Returns the inner (the Top Of Stack in the handlers stack) *Exception* handling *Context*.

xcpc_res xcpc_resource_add(xcpc_ctx *ctx***, void ***data***,**
void (* *free***)(void *));**

Adds a new *Resource* to the *Context* passed in the *ctx* parameter. The *data* parameter is the pointer to the *Resource*, while *free*() is the destructor to be called when the resource has to be freed. The function return the newly allocated resource, or throws an *Exception* in case of error.

void xcpc_resource_movehead(xcpc_res *res***);**

Moves the *Resource res* at the beginning of the *Resource* list in its *Context*. *Resource*s are removed (freed) from **HEAD** to **TAIL**.

void xcpc_resource_movetail(xcpc_res *res***);**

Moves the *Resource res* at the end of the *Resource* list in its *Context*. *Resource*s are removed (freed) from **HEAD** to **TAIL**.

void xcpc_resource_moveafter(xcpc_res *res***, xcpc_res** *rres***);**

Moves the *Resource res* after the *Resource rres*. This means that *Resource res* will be remove (freed) after *Resource rres*.

void xcpc_resource_movebefore(xcpc_res *res***, xcpc_res** *rres***);'**

Moves the *Resource res* before the *Resource rres*. This means that *Resource res* will be remove (freed) before *Resource rres*.

xcpc_ctx xcpc_resource_ctx(xcpc_res *res***);**

Returns the *Context* inside which the *Resource* passed in the *res* parameter is allocated.

void *xcpc_resource_data(xcpc_res *res***);**

Returns the pointer to the *Resource* associated with *res*.

void xcpc_resource_del(xcpc_res *res***, int** *do_free***);**

Deletes the *Resource* passed in the *res* parameter. If the *do_free* parameter is not zero, the corresponding *Resource* destructor is called (hence the *Resource* core data freed), otherwise the *Resource* is simply removed from its *Container* and its metadata freed). After the **void xcpc_resource_del(xcpc_res** *res***, int** *do_free***)** function returns, the *Resource res* is invalid.

void xcpc_resource_reparent(xcpc_res *res***, xcpc_ctx** *pctx***);**

Makes the *Context* passed in *pctx* the new parent of the *Resource* passed in *res*.

void xcpc_resource_set(xcpc_res *res***, int** *do_free***, void ****data***,**
void (* *free***)(void *));**

Changes the *Resource res* by setting the new *data* and *free*() parameters. If *do_free* is not zero, the previously associated *data* is freed using the previously associated destructor.

XCPC_THROW(*ctx***,** *exno***,** *data***)**

Throws an *Exception* number *exno* with its associated *data*. The *ctx Context* is used to backtrack and first the first *Exception* handler in the *ctx* hierarchy.

XCPC_RETHROW(*ctx***)**

This statement is used to re-throw the currently handled *Exception* so to handlers in the upper layers of the hierarchy. The *ctx* parameter is the *Context* under which the current *Exception happened. Trying to perform a* **XCPC_RETHROW(***ctx***)** using a *Context* that is not the one registered in the *Exception* block by a previous **XCPC_TRY(***ctx***)** will generate a panic.

XCPC_TRY(*ctx***)**

Opens an *Exception* handler block associated with the *Context* passed in the *ctx* parameter. This is how an *Exception* block may look like:

```
XCPC_TRY(ctx) {
        /*
         * Here we have some code that may throw exceptions ...
         */
         ...
}
XCPC_CATCH(XCPCE_OPEN) {
        /*
         * Handle the XCPCE_OPEN exception ...
         */
        ...
}
XCPC_CATCH(XCPCE_WRITE) {
        /*
         * Handle the XCPCE_WRITE exception ...
         */
        ...
}
XCPC_CATCH_ANY {
        /*
         * Handle all other exceptions ...
         */
        ...
}
XCPC_END_TRY;
```

Exception handlers can be nested inside each other. Every *Resource* or *Context* allocated inside the hierarchy rooted in *ctx* will be freed in case an *Exception* happened inside the code bound by the **XCPC_TRY(***ctx***)** block.

XCPC_CATCH(*exno***)**

Follows a **XCPC_TRY(***ctx***)** statement and is used to catch the *Exception* passed in the *exno* parameter (see above example).

The code inside the handler is supposed to handle the *Exception* that happened in the associated **XCPC_TRY(***ctx***)** block (and down inside its call hierarchy). On exit from the code block inside the handler, the program will resume to the instruction following the closing **XCPC_END_TRY** statement. The handler can use the **XCPC_RETHROW(***ctx***)** statement to pass the *Exception* to handlers up in the call hierarchy

XCPC_CATCH_ANY

Handle every *Exception* not handled by the previous **XCPC_CATCH(***exno***)** blocks. In the same way as **XCPC_CATCH(***exno***)** the handler code can simply exit the code block, or can use **XCPC_RETHROW(***ctx***)** to pass the *Exception* in the upper layers of the call hierarchy.

XCPC_END_TRY

Ends an *Exception* handler started with a previous **XCPC_TRY(***ctx***)** statement. If no *Exception* happens, or if one of the handler catch the *Exception* without re-throwing, the program will continue with the next *Exception* following the **XCPC_END_TRY** statement.

XCPC_RETURNV(*expr***)**

The code inside an *Exception* handler (being it TRY or CATCH) cannot simply issue a **return** to return from the current function. The **XCPC_RETURNV(***expr***)** statement must be used to return from the current function, with *expr* being the expression to be returned.

XCPC_RETURN

Same as **XCPC_RETURNV(***expr***)** but for **void** functions.

XCPC_EXCEPT_DATA

Macro that can be used to fetch the data associated with an *Exception*.

XCPC_EXCEPTION

Macro that can be used to fetch the *Exception* number.

Function wrappers

void *xcpc__malloc(xcpc_ctx *ctx***, xcpc_res ***pres***, long** *size***);**

Allocates a memory block of *size* bytes. A new *Resource* will be associated with the new block of data, and the *pres* pointer will receive its value. The new *Resource* will be stored inside the *Context* passed in the *ctx* parameter. The new block pointer will be returned by the function. In case of error, an *Exception* will be thrown.

void *xcpc__realloc(xcpc_res *res*, **long** *size***);**

Realloc a *Resource* previously allocated with **void *xcpc__malloc(xcpc_ctx** *ctx*, **xcpc_res ***pres*, **long** *size***)** to the new size *size*. The reallocated block will be associated with the same *Resource res* and the function will return it. An *Exception* is thrown in case of error.

char *xcpc__strdup(xcpc_ctx *ctx*, **xcpc_res ***pres*, **char const ***str***);**

Makes a copy of the input string *str* and links the allocated *Resource* inside the *ctx Context*. Returns the newly allocated string copy, or throws an *Exception* in case of error.

FILE *xcpc__fopen(xcpc_ctx *ctx*, **xcpc_res ***pres*, **char const ***path*,
char const ***mode*);**

Opens a new file using the **fopen**(3) function and links it to a new *Resource* that will be stored in the *pres* parameter. Returns the newly opened **FILE** pointer, or throws an *Exception* in case of error.

FILE *xcpc__fdopen(xcpc_ctx *ctx*, **xcpc_res ***pres*, **int** *fd*,
char const ***mode*);**

Opens a new stream file using the **fdopen**(3) function. Returns the newly opened **FILE** pointer, or throws an *Exception* in case of error.

FILE *xcpc__freopen(xcpc_res *res*, **char const ***path*, **char const ***mode***);**

Opens a new stream file using the **freopen**(3) function. Returns the newly opened **FILE** pointer, or throws an *Exception* in case of error.

int xcpc__open(xcpc_ctx *ctx*, **xcpc_res ***pres*, **char const ***path*,
int *flags*, **int** *mode***);**

Opens a new file descriptor using the **open**(2) function. Returns the newly opened file descriptor pointer, or throws an *Exception* in case of error.

void xcpc__push_remove(xcpc_ctx *ctx*, **xcpc_res ***pres*, **char const ***path***);**

This is an example about a *Resource* that does not have any real payload, but its used only to leverage the cleanup capabilities of the *Resource* destructors. It creates a *Resource* that whose cleanup will trigger the removal of the file whose path is passed in *path*.

void *xcpc__mmap(xcpc_ctx *ctx*, **xcpc_res ***pres*, **void ***start*, **size_t** *length*,
int *prot*, **int** *flags*, **int** *fd*, **off_t** *offset***);**

Creates a new memory mapping using the **mmap**(2) function. Returns the newly created mapping address, or throws an *Exception* in case of error.

ssize_t xcpc__write(xcpc_ctx *ctx*, **int** *fd*, **const void ***buf*, **size_t** *count***);**

Maps to the **write**(2) function and throws an *Exception* if the number of bytes written are different from *count*.

ssize_t xcpc__read(xcpc_ctx *ctx*, **int** *fd*, **void ****buf*, **size_t** *count***);**

Maps to the **read**(2) function and throws an *Exception* if the number of bytes read are different from *count*.

size_t xcpc__fwrite(xcpc_ctx *ctx*, **const void ****ptr*, **size_t** *size*,
size_t *nmemb*, **FILE ****stream***);**

Maps to the **fwrite**(3) function and throws an *Exception* if the number of elements written are different from *nmemb*.

size_t xcpc__fread(xcpc_ctx *ctx*, **void ****ptr*, **size_t** *size*, **size_t** *nmemb*,
FILE ****stream*);**

Maps to the **fread**(3) function and throws an *Exception* if the number of elements read are different from *nmemb*.

void xcpc__stat(xcpc_ctx *ctx*, **const char ****path*, **struct stat ****buf***);**

Maps to the **stat**(2) function and throws an *Exception* in case of error.

void xcpc__fstat(xcpc_ctx *ctx*, **int** *fd*, **struct stat ****buf***);**

Maps to the **fstat**(2) function and throws an *Exception* in case of error.

void xcpc__mkdir(xcpc_ctx *ctx*, **const char ****path*, **mode_t** *mode***);**

Maps to the **mkdir**(2) function and throws an *Exception* in case of error.

void xcpc__rmdir(xcpc_ctx *ctx*, **const char ****path***);**

Maps to the **rmdir**(2) function and throws an *Exception* in case of error.

void xcpc__remove(xcpc_ctx *ctx*, **const char ****path***);**

Maps to the **remove**(3) function and throws an *Exception* in case of error.

void *xcpc__opendir(xcpc_ctx *ctx*, **const char ****path***);**

Maps to the **opendir**(3) function and throws an *Exception* in case of error.

void xcpc__pipe(xcpc_ctx *ctx*, **int *** *fds***);**

Maps to the **pipe**(2) function and throws an *Exception* in case of error.

int xcpc__socket(xcpc_ctx *ctx*, **int** *domain*, **int** *type*, **int** *protocol***);**

Maps to the **socket**(2) function and throws an *Exception* in case of error.

void xcpc__bind(xcpc_ctx *ctx*, **int** *sfd*, **const struct sockaddr** **addr*, **int** *alen***);**

Maps to the **bind**(2) function and throws an *Exception* in case of error.

void xcpc__connect(xcpc_ctx *ctx*, **int** *sfd*, **const struct sockaddr** **addr*, **int** *alen***);**

Maps to the **connect**(2) function and throws an *Exception* in case of error.

void xcpc__truncate(xcpc_ctx *ctx*, **char const** **path*, **off_t** *len***);**

Maps to the **truncate**(2) function and throws an *Exception* in case of error.

void xcpc__ftruncate(xcpc_ctx *ctx*, **int** *fd*, **off_t** *len***);**

Maps to the **ftruncate**(2) function and throws an *Exception* in case of error.

off_t xcpc__lseek(xcpc_ctx *ctx*, **int** *fd*, **off_t** *off*, **int** *whence***);**

Maps to the **lseek**(2) function and throws an *Exception* in case of error.

EXAMPLE

To see how the **libxcpc** library can simply the code, let's consider a function that copies a file. One style to write such function would be:

```
int file_copy1(char const *src, char const *dst) {
        int sfd, dfd;
        size_t count, rdy;
        char *buf;
        struct stat stb;

        if ((sfd = open(src, O_RDONLY)) == -1)
                return -1;
        if (fstat(sfd, &stb)) {
                close(sfd);
                return -2;
        }
        if ((dfd = open(dst, O_WRONLY | O_CREAT)) == -1) {
                close(sfd);
                return -3;
        }
        if ((buf = malloc(BSIZE)) == NULL) {
                close(dfd);
                close(sfd);
                return -4;
        }
        for (count = 0; count < stb.st_size;) {
                if ((rdy = stb.st_size - count) > BSIZE)
                        rdy = BSIZE;
                if (read(sfd, buf, rdy) != rdy ||
                write(dfd, buf, rdy) != rdy) {
                        free(buf);
                        close(dfd);
                        close(sfd);
                        return -5;
                }
                count += rdy;
        }
        free(buf);
        close(dfd);
        close(sfd);
        return 0;
}
```

Another style example for coding the same function is:

```
int file_copy2(char const *src, char const *dst) {
        int err, sfd, dfd;
        size_t count, rdy;
        char *buf;
        struct stat stb;

        err = -1;
        if ((sfd = open(src, O_RDONLY)) == -1)
                goto err_1;
        err = -2;
        if (fstat(sfd, &stb))
                goto err_2;
        err = -3;
        if ((dfd = open(dst, O_WRONLY | O_CREAT)) == -1)
                goto err_2;
        err = -4;
        if ((buf = malloc(BSIZE)) == NULL)
                goto err_3;
        for (count = 0; count < stb.st_size;) {
                if ((rdy = stb.st_size - count) > BSIZE)
                        rdy = BSIZE;
                err = -5;
                if (read(sfd, buf, rdy) != rdy ||
                    write(dfd, buf, rdy) != rdy)
                        goto err_4;
                count += rdy;
        }
        err = 0;

err_4:
        free(buf);
err_3:
        close(dfd);
err_2:
        close(sfd);
err_1:
        return err;
}
```

Let’s see how it looks using the **libxcpc** library:

```
int file_copy3(xcpc_ctx ctx, char const *src, char const *dst) {
        xcpc_ctx wctx;
        int sfd, dfd;
        size_t count, rdy;
        char *buf;
        struct stat stb;

        wctx = xcpc_context_create(ctx);
        sfd = xcpc__open(wctx, NULL, src, O_RDONLY, 0);
        xcpc__fstat(wctx, sfd, &stb);
        dfd = xcpc__open(wctx, NULL, dst, O_WRONLY | O_CREAT, 0644);
        buf = xcpc__malloc(wctx, NULL, BSIZE);
        for (count = 0; count < stb.st_size;) {
                if ((rdy = stb.st_size - count) > BSIZE)
                        rdy = BSIZE;
                xcpc__read(wctx, sfd, buf, rdy);
                xcpc__write(wctx, dfd, buf, rdy);
                count += rdy;
        }
        xcpc_context_free(wctx);
        return err;
}
```

The code using *file_copy3*(), or using code that uses *file_copy3*(), will can then handle the exceptions in the *proper* place, like:

```
XCPC_TRY(ctx) {
        file_copy3(ctx, ...);
}
XCPC_CATCH(XCPCE_OPEN) {
        ...
}
XCPC_CATCH(XCPCE_READ) {
        ...
}
XCPC_CATCH_ANY {
        ...
}
XCPC_END_TRY;
```

The Pros of the *Exception* handling code against the *in function* error handling, is that you can handle exception wherever it makes sense for your program. In the simplest case, a program could have a single *Exception* block in the **main**() function, and handle everything in there.

PERFORMANCE

Compared to C++ *Exception* handling, **libxcpc** performance is very good. You can compare yourself using the **xcpc_bench** binary inside the **test** subdirectory, against the analoguous bench program at:

http://www.xmailserver.org/cpp-exbench.cpp

In my machine, **libxcpc** performs almost ten times faster than standard C++ *Exception* handling in the *throwing* case. While **GCC** *Exception* handling is about four times faster than **libxcpc** *Exception* handling in the *non throwing* case. Those are micro-benchmarks though, and the effective cost of the **libxcpc** *Exception* handling is negligible when used in software does does some real work besides calling **XCPC_TRY(***ctx***).**

NOTES

Exception Handler Bounds

When an *Exception* block is opened using **XCPC_TRY(***ctx***)** all the new *Resource*s and *Context*s that are rooted to the *Context ctx* will be freed in case of *Exception*. If a new *Resource* or *Context* is allocated on a *Context* that is not rooted in *ctx*, they will not be freed by the *Excption* handling mechanism (unless the handler does not handle the *Exception* or re-throws, and the upper layer handle uses a *Context* that is root for the allocated *Resource*s or *Context*s).

Multi Threading And Re-Entrancy

The **libxcpc** library is re-entrant by nature, since it does not use any writeable global variable. The **libxcpc** is thread-safe as long as two different threads do not work at the same *Context* hierarchy at the same time. It is perfetcly legal to allocated a *Context* hierarchy in one thread and pass it to another one. As long as two threads do not use it at the same time.

LICENSE

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. A copy of the license is available at:

http://www.gnu.org/copyleft/lesser.html

AUTHOR

Developed by Davide Libenzi <**davidel@xmailserver.org**>

AVAILABILITY

The latest version of the **libxcpc** library can be found at:

http://www.xmailserver.org/libxcpc-lib.html

BUGS

There are no known bugs. Bug reports and comments to Davide Libenzi <**davidel@xmailserver.org**>