

Open System

API Overview

Author: J. Costa

Date: July 10, 2018

Table of Contents

1 - Introduction.....	3
1.1 - Background.....	3
1.2 - Purpose.....	3
1.3 - Objectives.....	3
1.4 - Design goals.....	4
2 - Usage.....	6
2.1 - First client application.....	6
2.2 - Compiling a client.....	7
2.3 - Compiling the Library.....	8
3 - Design.....	9
3.1 - Baseline.....	9
3.2 - Modules.....	9
3 - Follow up.....	11

1 - Introduction

The purpose of the Open Systems API Overview document is to provide a starting point into this open source project.

The first chapter explains the project background and context, it states the project purpose, defines the project objectives and reports on the project design goals. The second chapter focus provides a short usage introduction and the third chapter details the API design. The last chapter provides follow up information.

1.1 - Background

As a software developer first and as an architect later I found how hard it was to interact with Operating Systems to use their services specially in the context of completely different platforms such as Windows and UNIX or even among different versions of the same OS. The rational for problems encountered while attempting to create software for different platforms is due to many factors starting with the origins of the platform, the initial goals of the OS, their technical constraints not to mention their design philosophies. The Operating Systems themselves where also influenced and constrained by the underlining software development language(s).

Most applications don't require an extensive or specialized interface to the OS and those that do are normally targeting a very specific range of services for which they require demanding qualitative requirements such as performance. In this case, the best course of action is to craft a specialized implementation to address such concerns. However, most applications require basic system services to operate and don't have strident technical constraints for their operations. This API is for the latter applications.

1.2 - Purpose

The Open Systems API provides a multi platform interface for essential system services developed in the C programming language. The API will be provided as a library that wraps platform dependent behavior and provides an uniform access to other C or C++ client applications. The purpose is that this API can serve as a foundation for other libraries, applications, programming infrastructures, etc.

1.3 - Objectives

In the concept stage of the API development it became clear that the API design had to be done considering the type of clients and what type of services these clients would require and what should be the selection criteria when different design solutions or incompatible alternatives where present. To that end, the following list of prioritized objectives serves a selection guide:

1. Multi-platform support

The first objective is to provide multi-platform services that can be used by insulating client applications of the specific details on how are services implemented.

2. Orthogonality

The API has to be regular, it must provide the services in the same manner independently of the OS, client applications and internal design aspects.

3. Clear design

The API must have a well defined design that makes it clear what are the dependencies, the call patterns, the module separation, etc.

4. Performance

The API should not hinder performance significantly. Generalization always introduces performance penalties since knowledge of the specific platform is missing, no performance optimizations are possible besides those obtain by the compiler but the design of the API should not introduce, in the normal usage of the API, performance hits that make the API unusable.

1.4 - Design goals

In the process of the designing the API, it became clear that some goals had to be defined to make the API consistent and to isolate the platform internal details. The following list defines those goals:

1. Module separation

The API is divided into “functional” modules and supporting modules. The functional modules are those module that provide the core of the API, these modules are those providing the services exposed to clients. Supporting modules are (in general) internal to the API.

2. Uniform function signatures

The call declarations/definitions for the functional modules have the same format:

t_status <module>_<facility>_action(input parameters, output parameters)

All functional modules return an opaque type (t_status), they all start by the module name, some facility name (think like a class name in object oriented programming) and the function name ends with an action. All service functions are composed of at least three fields.

Finally, in the parameters part, it starts with input parameters and one or more output parameters.

3. Strict internal call patterns

The API is designed using a layered approach where each functional module is independent of each other but relies on supporting modules. The call pattern is a strict layering from the functional modules to the internal modules or to the Operating System. To avoid duplicating code a common supporting module exists that allows functional modules to delegate the implementation to it.

4. Opaque types

Considering the nature of platforms that may be supported in the future, their differences, a generalization of the platform types is required in order to provide useful functionality for the library clients. To that end, opaque types are required and this also implies that basic type manipulation has to be provided by the API. As an example, a simple type comparison has to be done by the library since the client is not aware if the type is a primitive language type or a composed type.

2 - Usage

Many principles and design goals have been laid out in the previous chapter but nothing beats an example. The following section demonstrates how can the API be used by a client application.

2.1 - First client application

Let's assume that we need to write an application to write to the OS log some entries in a certain log level. To simplify, the client is a C program.

```
// Include the OS API header
#include "osapi.h"

int main( int argc, char * argv[] )
{
    t_log log;    // Define opaque type

    // My Client application name
    const char * source= "client";

    // The following entries are specific to each platform
    const char * target  = "LOG_LOCAL0";           // UNIX facility
    const char * options[] = { "LOG_PID", "LOG_CONS", NULL }; // UNIX log options

    // Open the platform log
    log_system_open( source, target, options, &log );

    // Write to the system log in Info level
    log_info_write( log, "My log entry" );

    // Finally close the system log
    log_system_close( log );
}
```

The first thing that can be seen is the inclusion of the header `osapi.h`, which imports the declarations of all modules, including the log module. The second thing that pops up is that despite the client being platform independent still defines some settings that are UNIX specific (in the example). While the API functions are generic they still operate under a specific OS. The approach followed is to parametrize the API through strings¹, in this case, the `LOG_LOCAL0` which indicates a certain target log facility and specific syslog options such as to log the current process PID (`LOG_PID`), to log to the console if the system log is not available (`LOG_CONS`) and the options array will finish with a `NULL` pointer to indicate an end of the list.

¹ The other option would require the creation of a super-generic model to support all possible OSs which is unfeasible.

In the example above, it is assumed that no errors occur while calling system services but since every functional module returns a status of the operation it is trivial to check for errors and to report on those errors using the following approach:

```
// Define the status type
t_status st;

// Call some function
st = machine_host_getName( size, name );

// Check the status
if( status_success( st ) )
    // Call succeed, print host name
    printf("Host name is:%s\n", name );
else
    // An error occurred, print error message
    status_message_print( st );
```

The output in case of error will be something like:

Module **MACHINE**, function **machine_host_getName** with status: **File name too long**.

The printed status message includes the module name (MACHINE), the function name (machine_host_getName) and the status string itself (File name too long). The error, in the example, was simulated by reducing the size parameter.

A status can also be checked using the macro status_failure:

```
if( status_failure( status_type ) )
    printf( "Error description..." );
```

2.2 - Compiling a client

To compile a client application, a GCC compiler² supporting C11 and C++11 is required. The following demonstrates the minimum compiler settings to build a client application against the OSAPI library:

```
gcc -DOS_LINUX -I<OSAPI_DIR>/code -L<OSAPI_DIR>/Release -o <client name> <client
source name> -losapi
```

² Any compiler supporting the library baseline can be used but the example uses GCC. Also, some compiler pragmas also consider the usage of the GCC or a compiler supporting those GCC constructions.

Where:

- OSAPI_DIR is the directory into where the source code was downloaded
- client name is the name of the final executable
- client source name is the client application source file
- OS_LINUX is the symbol that defines the target Operating System

2.3 - Compiling the Library

Currently there is only support for Linux in the library and, therefore, the library production requires besides GCC, also GNU make. After downloading the library, move to the osapi/Release directory and execute:

```
make clean all
```

In the Release folder, the shared library **libosapi.so** should be present.

In order to use the library remember to set the **LD_LIBRARY_PATH** to include the location where the library is located.

3 - Design

When design a system API to support heterogeneous OSs the first design decision is which OSs to support and which versions of those OSs since each OS has evolved through time and not all legacy OS constructions are worth or feasible to support. The concept of a baseline is required to define not only the OS versions to support but more generically the full development and runtime environment.

3.1 - Baseline

The first thing that needs to be specified in an API is how versions are to be managed and how these versions form a baseline that can be used securely by clients. The OS API baseline is composed of several versions for both development and runtime:

1. The library version itself

Besides the GIT library source version, the library has a logical version so that client applications can check if there is a compatibility either during compilation and/or runtime.

2. The programming language version

The C supported version is C11 and C++11. While the library is written in C, C++ clients can also call the library.

3. The Operating System version

Each OS or more concretely the **libc** library has a minimum support version that must be available for the OSAPI library to work. In case, of UNIX OSs, there are two baselines, the POSIX support version and the version of the **libc** library that defines the OS services that are system specific, i.e. which are not part of POSIX.

The POSIX minimum required version is 200809L and the XOPEN version is 700.

After the establishment of a baseline the next step is to define the structure and call patterns between modules.

3.2 - Modules

A client application that wants to use the OSAPI library needs to be aware of the calling patterns and conventions. All functional modules that expose an OS service follow the API convention described in section 1.4, this means that these library calls will have the same return type, same call name format and input/output parameter sequence. Also, no inter-(functional) module calls are allowed. To avoid duplicating functionality a common module aggregates the implementation of cross module functions. A Status and a General module exist to provide functionality for the implementation of the **t_status** type and general library definitions and types, respectively. The following figure illustrates the overall module design.

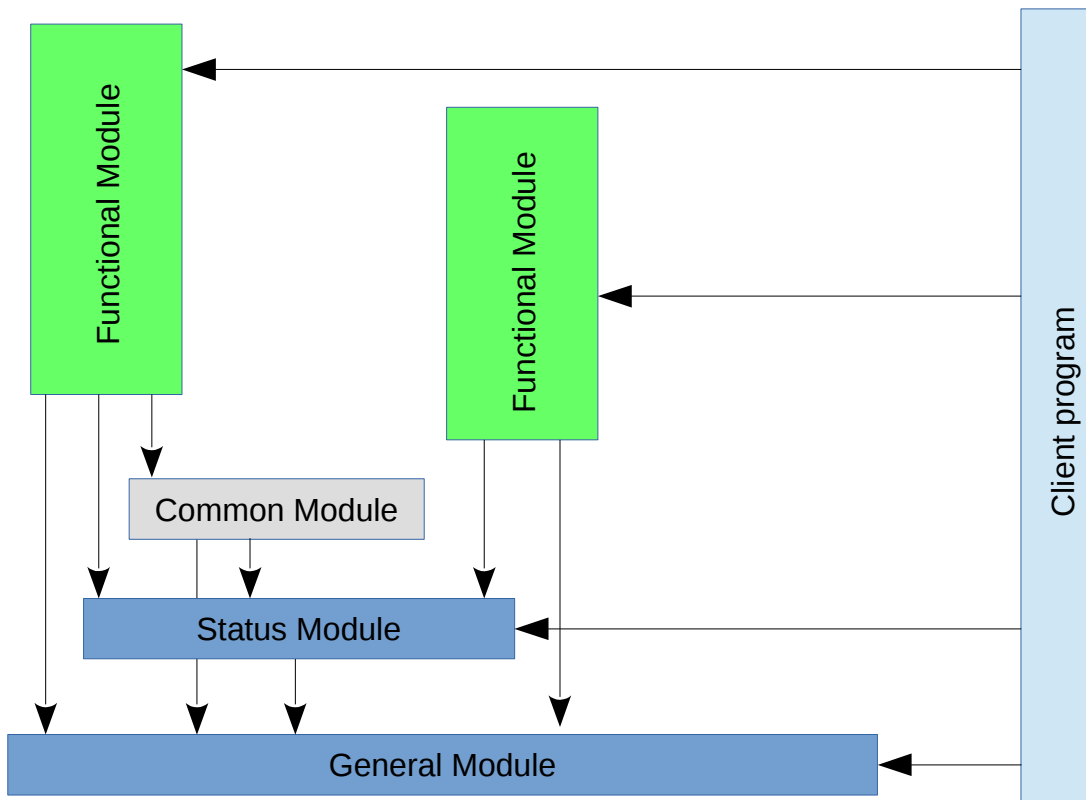


Figure 1: OSAPI Module Design

As seen in the above figure, a client application can also call the Status module to retrieve more information concerning a given returned status such as module and call name, status/error message, etc. A client can also call the General module to obtain library wide information such as API version. There is also a pseudo Error module that aggregates all functional module errors that logically is between the General and the Status modules. The reasoning of why such module exists, in the first place, is that in order to allow the Status module to operate on errors of functional modules and not breaking the strict layering pattern prescribed by the module design, there is a need to aggregate all errors in a single module that can both be called by it's own module but also by the Status module.

3 - Follow up

<TO-DO>