Open System

API Overview

Author: J. Costa

Date: July 28, 2019

Table of Contents

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 strings[1], 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 assume 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 compiler[2] supporting C11 and C++11 is required. The following demonstrates the minimum compiler settings to build a client application against the OSAPI library:

gcc -D*OS_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.

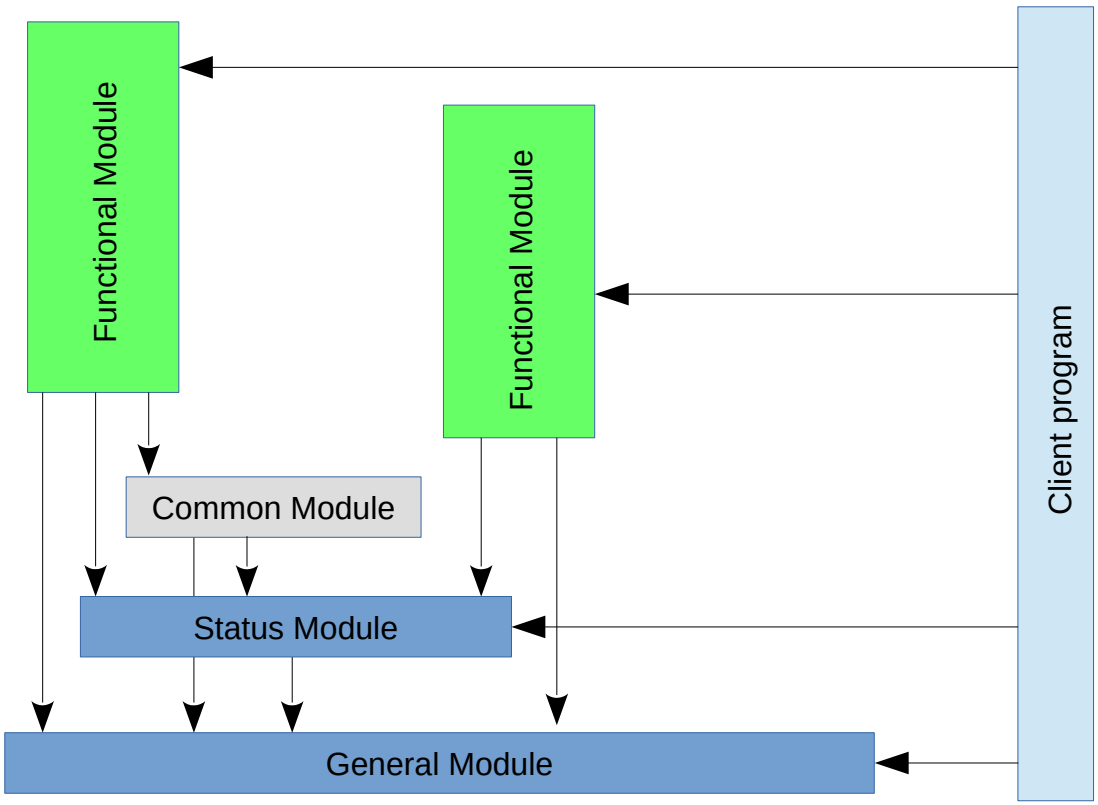

Figure 1: OSAPI Module Design

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.

3.3 - Function calls

One of the most important decisions in the API design is how the function call is structured, what are the constraints and how are these solved in the context of the API. The next sections detail these function call design decisions starting with the return information.

3.3.1 - Return information

Probably the most important decision in the API design is how information is returned by each API call. Information in the OSAPI library is returned in two separated sets. The target information of a module call, if any, is returned as function output parameters using pointers to some memory location. The result of the call itself that provides information about the operation is given by an

opaque type **t_status**. A status type is more than the information of a success of failure condition it provides the following addition information:

- Module that returned information

- Function name

- Internal or external (i.e. system) state

- Success or error code for the operation

This type is an opaque type which means that the returned information can be altered in a future revision of the library. To avoid breaking compatibility, client applications are advised to use the facilities provided by the Status module to retrieve the information from the t_status type. In particular, checking success or failure can be done using the macros **status_success** and **status_failure**. Additionally, there are calls to print a status message, to retrieve the name of the function, module, etc.

3.3.2 - Function name

Most system APIs don't have any naming call convention which makes it difficult to remember the available facilities. The lack of a convention requires extra care when using the provided API since each function is independent and there is no uniform handling of return types and input/output parameters. To improve consistency the OSAPI as a function name convention:

<module name>_<facility>_<action>

The unnatural function name convention arises from the need to place the module name in the name of the function to acts as a name space space so that two modules can have calls with the same name (in this case the name can be similar for the facility and action). This naming convention allows a high number of calls to exist without clashes due to the partitioning done by the module name. This wouldn't be required if the C language had a generalized namespace like C++.

3.3.3 - Calling convention

Apart from the function name convention, there is also a calling convention for the "functional" module operations. The function parameters start with input parameters followed by output parameters. Output parameters are always declared explicitly as pointers to types.

3.3.4 – Memory handling

Another important topic in the API design is how memory is management, i.e. who allocates memory and who releases that memory when am operation is executed. The convention for memory allocation and deallocation gives the client of the API the sole responsibility for memory management. The client allocates memory, passes a pointer to that memory and the size of that memory. He is also responsible for the memory deallocation.

Considering the nature of the API types which in some cases are system specific, the API must provide functions to retrieve the size of the API types in order for clients to perform the correct memory allocation.

3.3.5 - Generalization

The support of several different kinds of OSs forces the usage of opaque types in order to encapsulate system specific information in the same API types. For instance, a **t_uid** type is a type that contains the required information to represent a User ID. Such type may be a positive number in one OS and a string in another. To accommodate such OS differences the API defines it's own type to provide an abstraction of the underlining OS type. This generation comes with the cost that normal operations with a type can only be performed by the API itself and not by the clients of the API. The consequence is that the API has to be more extensive since it must provide type handling operations such as checking equality, type size, etc.

Another problem with the encapsulation of system specific behavior is that some facilities in the different supported OSs are very difficult to generalize due to their dissimilitudes. For these, two options are available:

1) Create a more general model that encompasses all supported Oss

   This option implies the creation of higher level abstraction to support not only the OSs currently supported but also all future supported OSs. It implies also a higher level of complexity and as a consequence higher effort.

2) Accept the differences and delegate the differences to the client of the API

   Another approach is to accept the differences between systems and pass the burden of handling the differences to the API client. It is a less sophisticated approach but also a lesser complex one.

Both approaches are possible in the API but the second is the preferred since it provides the bonus of allowing the API to be parameterized through strings. Client applications can have different string files depending on the target OS. As an example, any system log is open using the following call:

```
log_system_open( source, target, options )
```

All input parameters are strings (or array of strings in the case of the log options) and these strings are system specific and can be defined by clients, for UNIX, as shown in section 2.1. The following is an extract of that code with the generalization through strings example:

```
// 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 );
```

The same example in section 2.1 contains an example of the first generalization option above:

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

The **log_info_write** call example is a generalization of all different OSs log levels present in the **log** module API:

```
t_status log_debug_write      ( t_log, t_log_message  );
t_status log_info_write       ( t_log, t_log_message  );
t_status log_warning_write    ( t_log, t_log_message  );
t_status log_error_write      ( t_log, t_log_message  );
t_status log_fatal_write      ( t_log, t_log_message  );
```

All the above calls are a generalized model of all system log levels. These calls in turn call the more generic function:

```
t_status log_system_write     ( t_log, t_log_level, t_log_message );
```

Client applications can use both generation strategies by calling the first functions which uses a generic system log model or using the second approach by using directly the more general and more abstract form of the API.

3 - Follow up

<TO-DO>