ALOE++ Architecture

Version: 0.2

Status: Draft

1 Introduction

In this document, we briefly describe the general architecture of the ALOE++ core framework. We also indicate the adopted coding style. We hope this helps building a community of developers which may find easier to extend and improve ALOE++.

Disclaimer: We use UML models to represent relations between classes, components or elements in the system. However, we do not always follow the language rules and/or notation. The objective is only providing an illustration of concepts and ideas.

2 Definitions

Table 1. Definitions

Processor	One processing core
Node	A set of shared memory processors
Platform	A set of nodes (no shared memory)
Manager	A processor in a node that runs a program to control the entire platform
A-Node	The ALOE++ program that runs on each node to support the distributed execution
A-Node-M	Is an A-Node that acts as a master clock reference (there is only one A-Node-M in the Platform)
A-Manager	The ALOE++ program that runs on the manager
A-CF	The ALOE++ Core Framework is the union of all software elements that enable the distributed execution
Module	A module is a signal processing algorithms that follows a predefined structure imposed by ALOE++
Waveform	A set of interconnected modules
SWAPI	The Software API provides platform-independence to the Modules
HWAPI	The Hardware API provides platform-independence to the A-Node and A-Manager programs and the SWAPI
MANAPI	User interface to manage the entire platform
Module-Skeleton	Is a set of tools designed to faciliate the creation and implementation of modules.

3 General Architecture

ALOE++ is a middleware and execution environment for distributed signal processing. The first requirement is to provide an environment where object instances can be read or modified from remote processors. The A-CF is designed to allow the A-Manager to have mirrored instances of objects distributed among the different A-Nodes. The general architecture on a platform deployment is depicted in Figure 1.

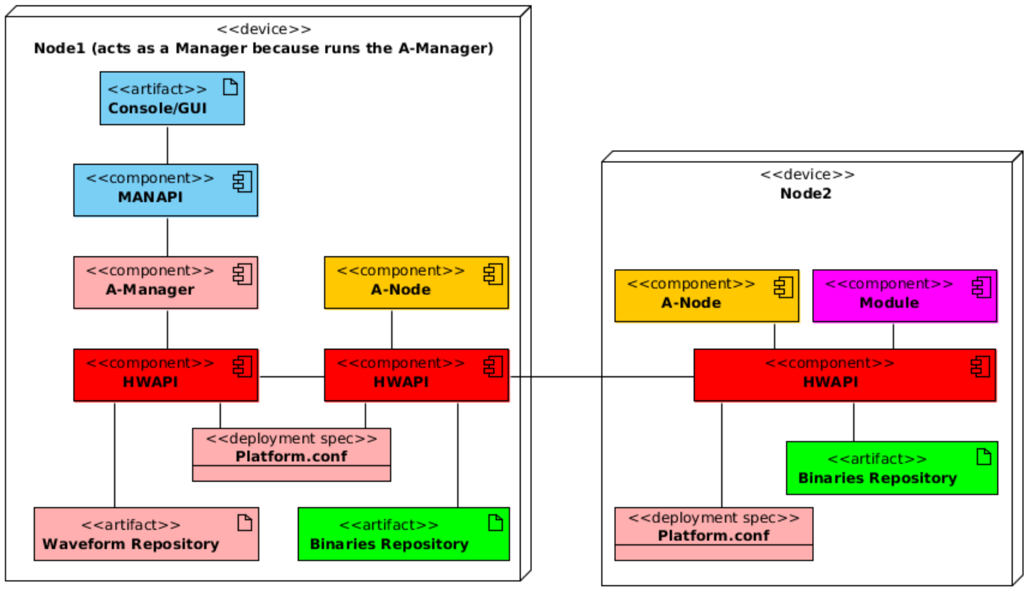

<<device>>
Node1 (acts as a Manager because runs the A-Manager)
<<artifact>> Console/GUI
<<component>> MANAPI
<<component>> A-Manager
<<component>> A-Node
<<component>> HWAPI
<<component>> HWAPI
<<deployment spec>> Platform.conf
<<artifact>> Waveform Repository
<<artifact>> Binaries Repository
<<device>>
Node2
<<component>> A-Node
<<component>> Module
<<component>> HWAPI
<<artifact>> Binaries Repository
<<deployment spec>> Platform.conf

Figure 1. ALOE++ Platform deployment.

The user interacts with the entire platform using a Console, GUI or a custom application, though the CMDAPI. The CMDAPI is part of the A-Manager component. The A-Manager component uses the HWAPI to access the A-Nodes in the distributed platform.

The platform deployment specifications are stored in the Platform.conf file. This file must be accessible by all system parts (through the HWAPI). This may be achieved using SMB or NFS file sharing or other custom tools.

The waveform models are stored in text files, which are read by the A-Manager when the user loads a new waveform. This file defines a set of modules and the interfaces between them. For each module, the file also indicates the name of the module's binary file (program or library). The set of waveform module binary files is stored in a repository called Binaries Repository. The Binaries Repository also must be accessible by all A-Nodes in the platform.

3.1 Physical Interfaces

Physical interfaces connect the different nodes between each other and with the manager. The list of interfaces is described in Table 2.

Table 2. List of physical interfaces.

ctrl_itf	Used by the manager to send commands to the nodes and transfer object instances.
probe_itf	Used by the nodes to send asynchronous messages to the manager (variable reports and module execution failures)
sync_itf	Used by the nodes to synchronize with the A-Node-M (clock reference)
data_itf	Data interfaces between nodes for inter-module communications.

4 Classes

Despite ALOE++ is implemented in ANSI C, in this document and source code, we use the object oriented (OO) notation of classes, inheritance, attributes and so forth. We think this notation is more clear and insightful, while (to some extend) it is language-independent.

4.1 Overview

There are three packages in ALOE++:

- **BASE**: Contains MANAPI library, the A-Node and A-Manager classes as well as common classes. The MANAPI library consists on all the set of public methods and attributes in this package.
- **SWAPI**: Contains the SWAPI library. The SWAPI library consists on all the set of public functions and class attributes in this package.
- **HWAPI**: Contains the HWAPI library and the kernel classes. The HWAPI library consists on all the set of public functions and class attributes in this package.

Notation: For the HWAPI and SWAPI packages, all methods and attributes names follow the rule:

```
packagename_classname_method();
packagename_classname.attribute;
```

For the BASE package, the names follow the convention:

```
classname_method()
classname.attribute;
```

The class name of the classes of the BASE package is `an_somename` or `am_somename` for A-Node and A-Manager related classes, or just `somename` for other classes.

Visibility: The visibility of attributes/methods of the UML diagrams shown in this section can be:

- *public* (+): Accessible from outside the package. Declared in a single `packagename.h` file in a public directory
- *package* (~): Accessible by all other elements of the package. Declared in a `classname.h`, which is included by any other class using it.
- *private* (-): Accessible by its class only. Declared in the `classname.c` file with static scope modifier.

Directory Structure: Each package is distributed in a separate directory. There are two subdirectories: `src/` which has all class sources and definitions, and `include/` which has the `packagename.h` file.

4.2 Serializable Classes

One of the functionalities of the A-Node and A-Manager is to provide seamless access to object instances. This is achieved through the **Serializable Interface** (in the BASE package)**.** Objects inheriting from this interface must implement two functions:

```
~serialize(pkt : packet) : serializable
```

```
~unserializeTo(pkt : packet, dest_object : serializable)
```

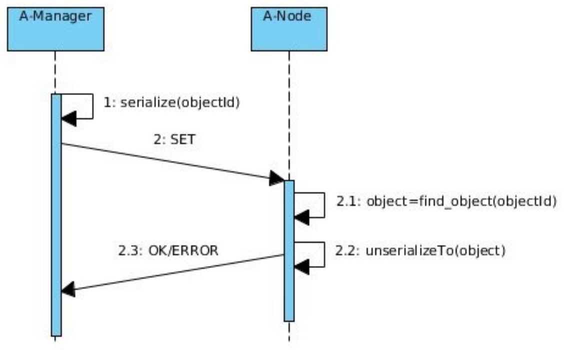
This functions are used to append an object instance to a packet that will be sent through the network. And vice versa, to save replace an object in memory by an object inside a packet received from the network.

The set of serializable classes is described in Table 2. The A-Node class extends these classes, adding the prefix `nod_` to the class name. Only public attributes are extended. The `nod_` classes have their own methods.

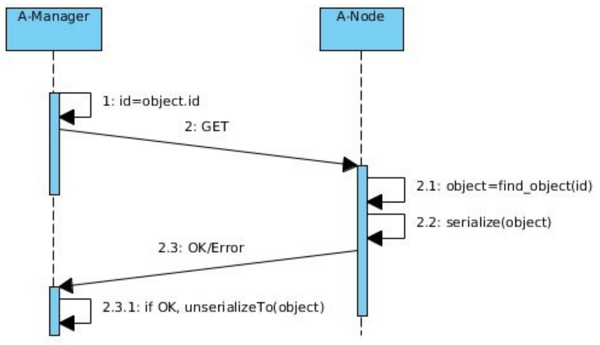
Table 2. Serializable A-Manager classes

`module`	Has information about a module name, binary file, status, interfaces, variables and so forth.
`module_interface`	Characterizes a logical interface between two modules.
`module_variable`	A variable is a value accessed by a module and the manager as well (i.e. user through the GUI or console).
`module_execinfo`	Keeps information about the module's execution properties, e.g. mean/max execution time, MOPTS, number of rt-faults, relinquished time, etc.
`waveform`	A collection of modules plus other information.
`waveform_status`	The status can be: LOADED, INIT, RUN, PAUSE, STOP. Used to coordinate the execution of the waveform.

Serializable object instances in the A-Manager have a unique id. Each object has a mirror instance in one of the A-Nodes. The A-Manager objects and A-Node replicas share the same id, providing the basic communications mechanism. The procedure to modify an object instance in an A-Node and the process to update an A-Manager object instance with the contents of the A-Node object are depicted in Figure 2.



(a)



(b)

Figure 3. SET (a) and GET (b) procedures to mirror object instances among the distributed platform.

4.3 BASE Package

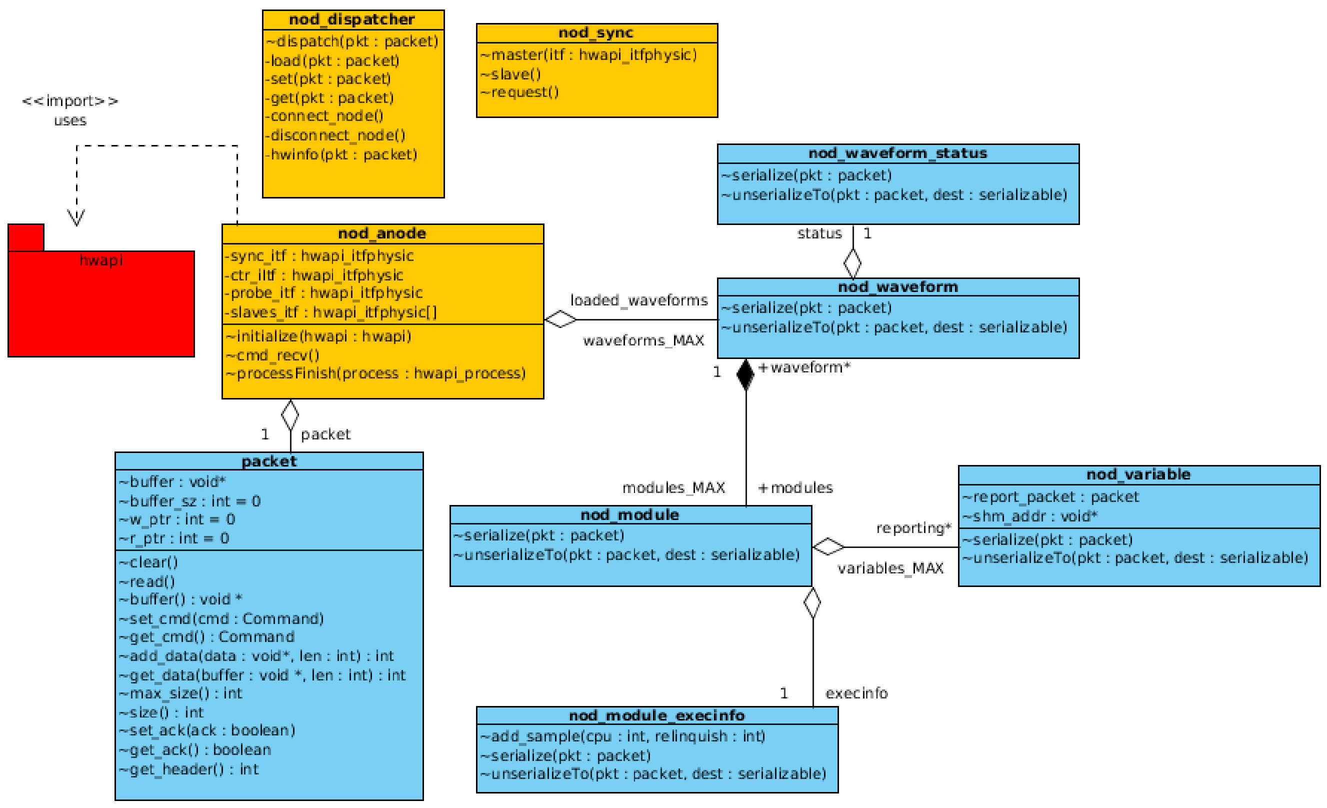


Figure 4. A-Node class diagram

One instance of the `nod_anode` class is created by the HWAPI. The `hwapi_kernel` (see Section 4.4) class creates a thread with normal priority that runs the `nod_anode_cmd_recv()`. This function is an infinite loop that receives command packets from the *ctrl_itf* and calls `nod_dispatcher_dispatch()` method for command processing. Once a waveform is loaded, it is saved into the `nod_anode.loaded_waveforms` array. Then, SET or GET commands can be issued to waveform, modules or variable object instances.

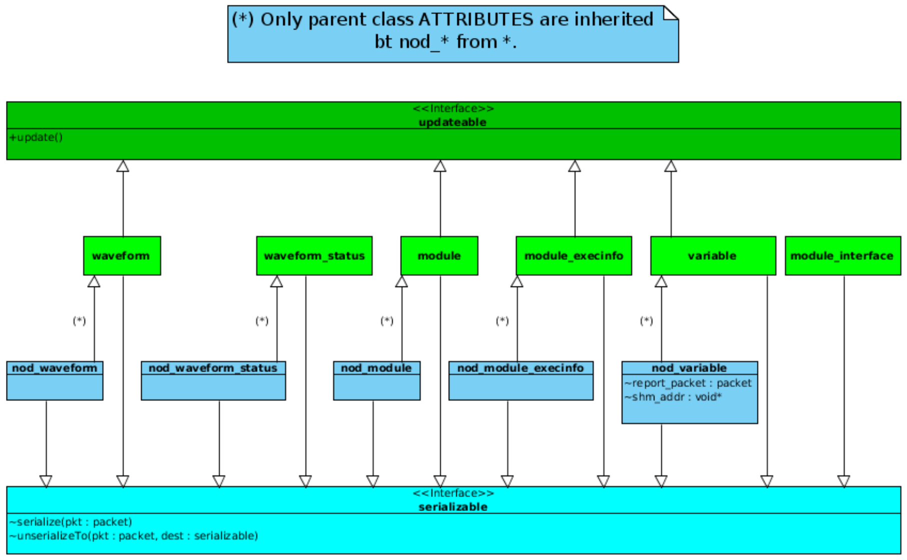


Figure 5. A-Node and A-Manager class relations

All objects managed by the `nod_anode` class implement the `serializable` interface (Figure 5), which allow to exchange their attributes with the A-Manager. A SET/GET command destination is configured using the `packet_set_dest(waveform_id, module_id, variable_id)` command. The packets have a common structure. The maximum control packet size is 64 Kbytes. A header precedes the packet data, which contains the following fields:

1) 32-bit word: Packet Info

2) 32-bit word: Destination Waveform Id

3) 32-bit word: Destination Module Id

4) 32-bit word: Destination Variable Id

The first word (Packet info) contains the following information (from LSB to MSB):

1) 0..7

MANAGER->NODE: Magic word to check if it is a control packet. 0xAF

NODE->MANAGER: 0xF0 if command successful or 0x0F if command error.

2) 8..15 command word (integer, up to 256 commands are supported)

3) 16..31 packet size in bytes (integer, up to 65536 bytes can be transferred)

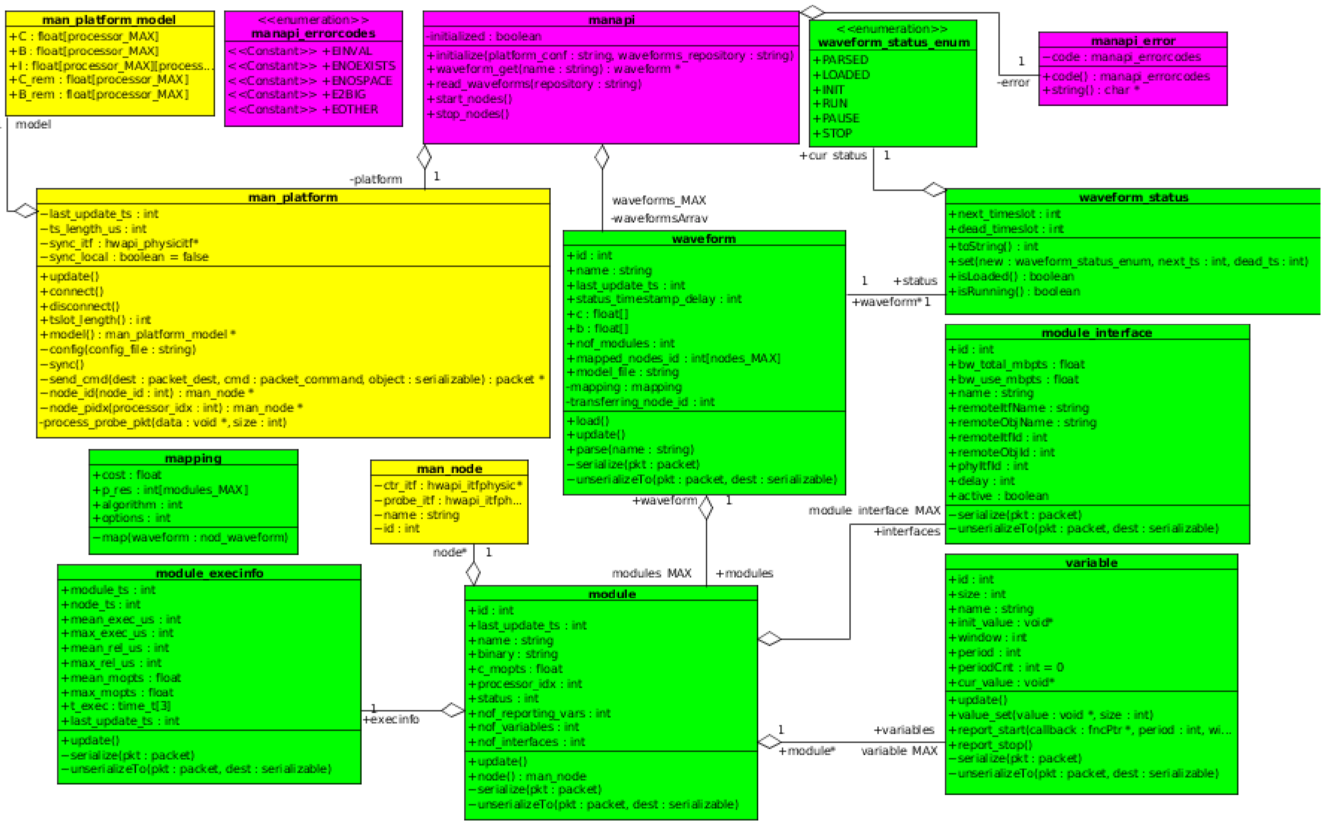


Figure 6. A-Manager class diagram

The `manapi` library is the user frontend to the A-Manager, which at the same time is a frontend to the entire platform. The set of public methods and attributes in Figure 6 forms the MANAPI. Note that a waveform can be loaded calling first `waveform_parse()` and then `waveform_load()` to a `waveform` instance given by `waveform_get()`.

The `waveform` class has a set of `module` object instances, which in turn have a set of `variable` or `module_execinfo` instances, for example. These latter classes can be used to obtain the value of a variable or the execution statistics of a module, for instance. Let us illustrate how to obtain the variable value of a module. First, we need to update the `am_module_variable` object in the manager with its mirror in the (unknown) node. Then, we can directly access the local object value, that is:

```
waveform = manapi_waveform_get(waveform_name);
object = find object_name in waveform;
variable = find variable_name in object;
variable_update(variable);
variable->cur_value // pointer to the current updated value
```

4.4 HWAPI Package

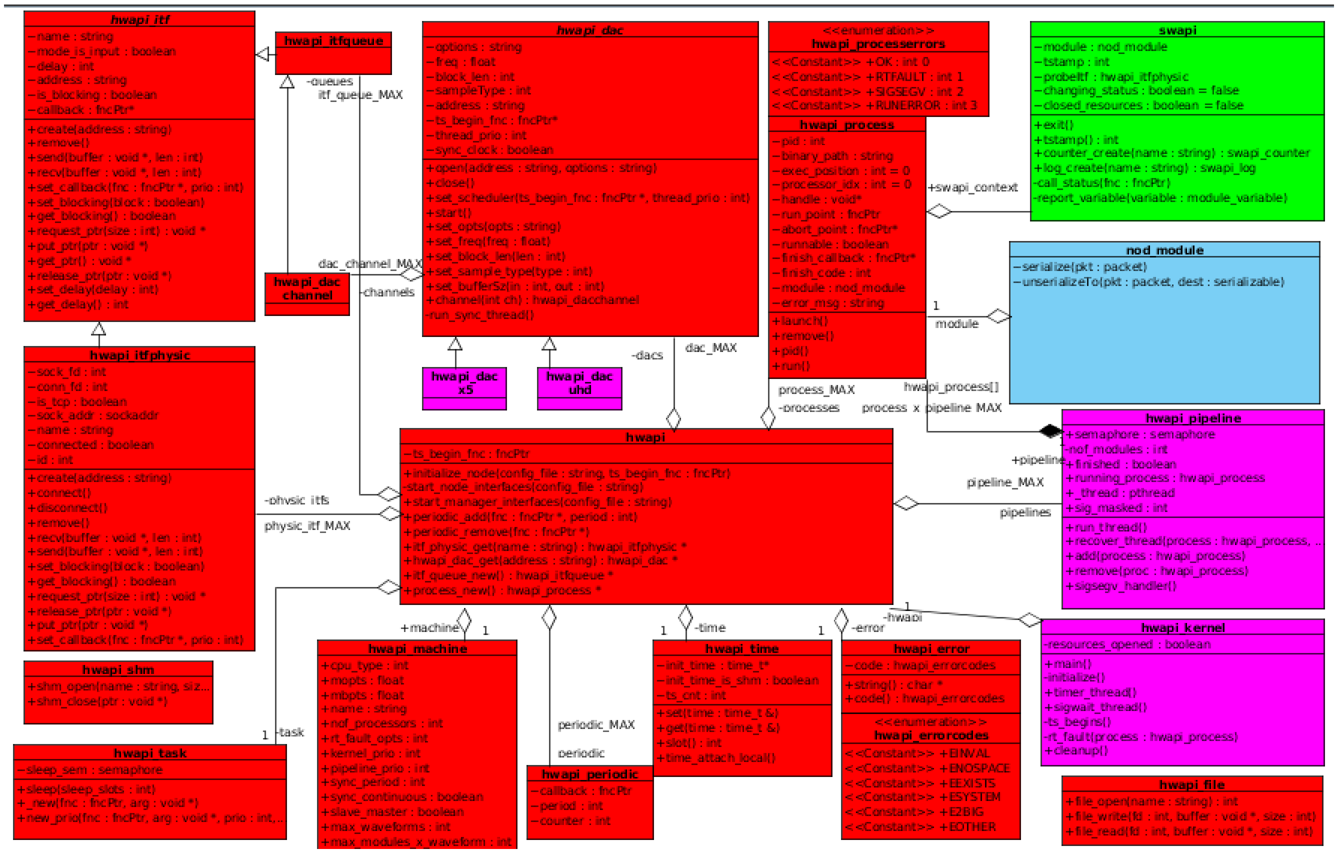


Figure 7. HWAPI class diagram

The `hwapi` provides access to platform specific services, so that the rest of the ALOE-CF code is platform-independent. These services are implemented in different classes (in red in Figure 7), the most important of them are described in Table 3. There is another part of the HWAPI (in magenta in the figure) which is inaccessible to the API user.

Table 3. Most important classes of the `hwapi` package

`hwapi`	Initialization and front-end to the creation or access to API objects
`itf`	Abstract interface to provide communications. Two classes implement it: queue and physical. A queue is a shared memory packet-based queue, used basically for inter-module data communications. A `itf_physical` is an external physical interface. Finally, the interfaces to the digital converters inherit the methods implemented by the queue interface.
`dac`	Abstract interface to the digital converter. Permits to change operating frequency, sampling frequency, frame block size and so on. A USRP implementation, for instance, implements this interface.
`file`	Access to the filesystem
`shm`	Shared memory management. Since ALOE++ is thread-oriented, this is a simple memory pool management located in a global memory.
`machine`	Keeps information about the current system and configuration options.
`process`	A `hwapi.process` is defined as dynamically loaded shared library that follows some predefined structure (see class documentation). A `hwapi_process` is associated with one signal processing module.
`pipeline`	A `hwapi.pipeline` is a set of `hwapi.process` which are executed following a given order (`hwapi.process.exec_order)` in a pipelined fashion. There is one class for every processing core in the node. Each class creates a single thread that runs the set of modules it has been allocated.
`kernel`	Contains the `main()` program function. At program start, it creates the `a_node` command processing thread, initializes the hwapi and falls to an infinite loop (`kernel.timer_thread`) that repeates every `ts_len` microsecond.

Interface (physical and queue), process and DAC objects are created statically by the library. The general usage of the API is as follows. Let for instance suppose we want to use an interface of type queue. First we request a new queue

```
hwapi_itf_queue *my_queue = hwapi_itf_queue_new();
/* check is not NULL */
```

Now I can use the queue. From the hwapi_queue class I get the functions to use it:

```
hwapi_itf_queue_send(my_queue, some_buffer, buffer_size);
```

will just send some data through the queue.

4.4.1 Scheduling

The scheduling of signal processing modules is governed by the `pipeline` class. The time slot clock is driven using four different mechanisms.

1) Local real-time timer, using the `kernel.timer_thread`
2) Digital converter sampling clock divided by the frame length
3) Active master node synchronization. The A-Node-M sends one small packet at the beginning of each time slot. Both timers are then phase misaligned by the inter-node latency (could be corrected estimating the latency using RTT and assuming it is invariant). Frequency alignment is perfect, at the expense of network overhead.
4) Passive master node synchronization. The local node runs a Local real-time timer (first mechanism) and a periodic function requests the time to the A-Node-M. The response is adjusted by computing the RTT and estimating the

one-way latency. Phase alignment is good, but frequency can be significantly misaligned in some systems. Overhead is minimal.

Each of these mechanisms, calls the `kernel.ts_begin()` method at the beginning of each time slot. This function performs a few simple management tasks and then activates *N* semaphores where *N* is the number of processors in the node. *N* `pipeline` threads are waiting to these semaphores, hence synchronizing the execution in all processing cores with the main kernel timing. Each of these threads calls the `process.run_module()` function for each allocated process, one after another. The execution order is indicate by the `process.exec_position` field.

4.4.2 Thread Protection and Real-time Control

Since several modules are executed by a single thread, the integrity of each module is subject to the correct execution of all other nodes in the same pipeline. A signal SIGSEGV, SIGILL, SIGBUS or SIGFPE signal generated by a module is handled by the `hwapi_pipeline_sigsegv_handler` function. The process then will be flagged as non-runnable, the thread will be terminated and a new thread will begin its execution, following with the next thread in the chain. The process terminates while the rest of the modules may continue their execution normally.

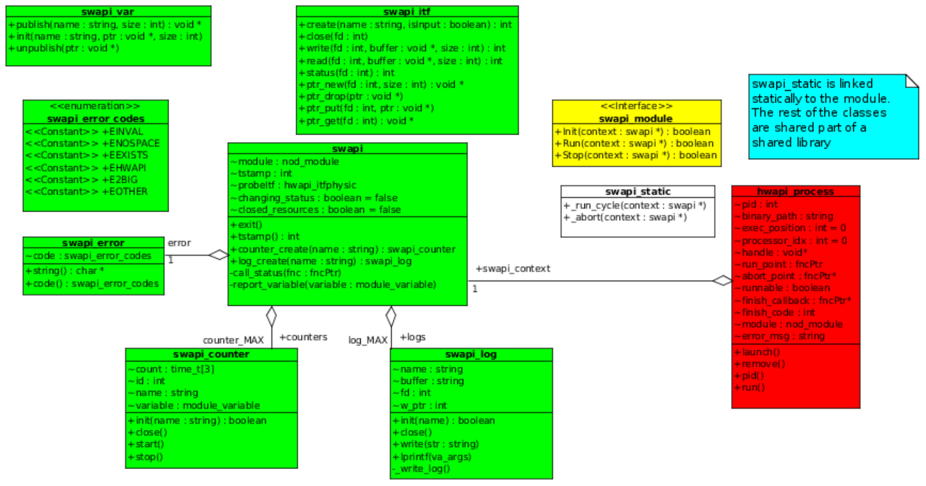
A similar procedure occurs when the `hwapi_kernel_ts_begin()` function detects that a module has violated the real-time constraints. There are two different types of real-time control, which are configured in the platform configuration file:

1) Soft real-time failure: occurs when the first module runs for longer than one time slot.
2) Hard real-time failure: occurs when all modules run for longer than one time slot.

If a real-time failure occurs, the `hwapi_kernel_rt_fault()` function is called by the kernel thread, which will terminate the pipeline thread where the module causing the failure is running. The module is flagged as non-runnable and the pipeline thread is restored. The process terminates while the rest of the modules may continue their execution normally.

In the event of a module terminating its execution abnormally (i.e. without passing through the STOP status), an asynchronous message is sent to the A-Manager through the *probe_itf*. The manager then may decide what to do. In this implementation, the default behavior is to terminate all the module's waveform by changing the status to STOP.

4.5 SWAPI Package



swapi_var
+publish(name : string, size : int) : void *
+init(name : string, ptr : void *, size : int)
+unpublish(ptr : void *)
swapi_itf
+create(name : string, isInput : boolean) : int
+close(fd : int)
+write(fd : int, buffer : void *, size : int) : int
+read(fd : int, buffer : void *, size : int) : int
+status(fd : int) : int
+ptr_new(fd : int, size : int) : void *
+ptr_drop(ptr : void *)
+ptr_put(fd : int, ptr : void *)
+ptr_get(fd : int) : void *
<<enumeration>>
swapi_error_codes
<<Constant>> +EINVAL
<<Constant>> +ENOSPACE
<<Constant>> +EEXISTS
<<Constant>> +EHWAPI
<<Constant>> +E2BIG
<<Constant>> +EOTHER
<<Interface>>
swapi_module
+Init(context : swapi *) : boolean
+Run(context : swapi *) : boolean
+Stop(context : swapi *) : boolean
swapi_static is linked statically to the module. The rest of the classes are shared part of a shared library
swapi
~module : nod_module
~tstamp : int
~probeItf : hwapi_itfphysic
~changing_status : boolean = false
~closed_resources : boolean = false
+exit()
+tstamp() : int
+counter_create(name : string) : swapi_counter
+log_create(name : string) : swapi_log
-call_status(fnc : fncPtr)
-report_variable(variable : module_variable)
swapi_static
+_run_cycle(context : swapi *)
+_abort(context : swapi *)
hwapi_process
~pid : int
~binary_path : string
~exec_position : int = 0
~processor_idx : int = 0
~handle : void*
~run_point : fncPtr
~abort_point : fncPtr*
~runnable : boolean
~finish_callback : fncPtr*
~finish_code : int
~module : nod_module
~error_msg : string
+launch()
+remove()
+pid()
+run()
swapi_error
~code : swapi_error_codes
+string() : char *
+code() : swapi_error_codes
error
1
+swapi_context
1
counter_MAX
+counters
log_MAX
+logs
swapi_counter
~count : time_t[3]
~id : int
~name : string
~variable : module_variable
+init(name : string) : boolean
+close()
+start()
+stop()
swapi_log
~name : string
~buffer : string
~fd : int
~w_ptr : int
+init(name) : boolean
+close()
+write(str : string)
+lprintf(va_args)
-_write_log()

Figure 8. SWAPI class diagram

An ALOE++ *Module* interacts with the rest of the platform through the SWAPI library. It provides inter-module communication, logging, timing and global variable management services, among others. The SWAPI library consists on two parts: a shared and an static library. The static library provides two functions called by the pipeline scheduler: `_run_cycle()` and `_abort()`. The former is the common execution function, called each time slot while the latter is used to cleanup resources in case of a hwapi-initiated program termination.

The shared library implements the rest of the services. The usage of the SWAPI library is analogous to the HWAPI (Section 4.5). Thread-safety is implemented passing a context pointer to all library function calls. This option is robust, portable and lightweight at the same time. The modules implement the `swapi_module` interface. This interface declares one method for each execution status: INIT, RUN and STOP. These methods receive a pointer to the SWAPI context, which has to be passed to any SWAPI library call. The `swapi_module` class documentation explains the functionalities that each of these functions should implement.

5 Procedures

5.1 Platform Booting

5.2 Platform Shutdown

5.3 Waveform Loading

5.4 Waveform Unloading

5.5 Waveform Status Change

5.6 Real-Time Control

5.7 Variable SET/GET

5.8 Variable Report

6 Coding Style

In this section, we briefly define the general adopted coding style. Despite being of a matter of personal choice, a uniformed coding style and naming convention facilitates a cooperative development between different programmers, one of the main goals of ALOE++.

6.1 Code Formatting

ALOE++ adopts the Linus Torvals coding style. A detailed specification can be found at https://computing.llnl.gov/linux/slurm/coding_style.pdf. We briefly summarize here the most important aspects:

- K&R general format
- 8 spaces tab

- Maximum 3 levels of indentation
- Open brackets in the same line except in functions, placed in a new line
- `} else {` to save space
- Functions maximum length of 30 lines. If longer, split work in several functions, inlining if performance-sensible
- Comments should be placed at the head of the function. Comments inside the functions should be avoided, except in very specific cases (code should be readable by itself)

And last but not least, in order to ensure 32-bit alignment, which improves performance in some processors, we impose the following conditions:

- **All data types are** 32-bit Integers or 32-bit Floats.
- **Strings** are statically allocated and there are only two classes.
  - Normal: char my_string[64];
  - Long: char my_string[512];

6.2 Naming Conventions

The names of the variables and classes should follow the rules in Table 4. **All variables are lower_case_with_underscore**

Classes	Descriptive name: `module` or `man_module`

Public methods	Descriptive name preceded by the class name:
	`module_value_get()` or `module_value_set()`
	If the class name is too long, use the first 3 words as a prefix (must be uniform in all the class). For instance, the methods of the `serializable` class:
	`ser_serialize()` and `ser_unserialize()`
	Public methods are declared in the .h file of the class

Public/Package attributes	Descriptive name: `module.current_value` and `man_module.status`
Private functions	Descriptive name: `get_value_aux()` Declared `static` in the .c file of the class.
Private attributes	Forbidden in non-instaintable classes (*)
Local variables	Short: for instance `tmp`, or `i, j, k` for loop index
Number of objects of a class	`nof_classname` (in singular), e.g. `nof_variables, nof_logs or nof_modules`

(*) See Section 6.3
(**) See Section 6.3.2

6.3 OOP in ANSI C. Adopted solution

Only a small set of OO features are adopted by ALOE++. We have written a simple example to guide developers to start coding for ALOE++ as well as provide an illustrative example of our coding style. In this example, we also provide the skeleton and a set of macros for facilitating the creation and definition of new classes.

This example can be downloaded from the ALOE++ GitHub repository at:

https://github.com/flexnets/aloe/tree/master/Documentation/aloe_classes_example

6.3.1 Classes

Conversely to common ANSI C OO styles that make an extensive use of pointers, all class attributes are visible in ALOE++.

This allows us to allocate the memory statically, preventing memory leakages and facilitating compiler optimization of loops. This solution, obviously, has its drawbacks, like a lower memory efficiency and poor information hiding (only methods can be hidden using the `static` keyword).

An array of object instances is allocated very easily. For instance

```
module modules[2];
```

creates two objects of the class module. Before accessing them, we have to assign a class to them:

```
init(modules,&CLASS(module),2);
```

Now we can access the attributes `modules[i].name` or `modules[i].id` as usual. The module class implements the methods of the `serializable` interface. They can be called simply using:

```
ser_serialize(&modules[0]);
ser_serialize(&man_modules[0]);
```

where in this example, man_modules[0] is an object of the class `man_modules`, which implements the `serializable` interface too. Note that both objects are treated indistinguishable using the same interface.

A class is defined in the file `classname.h`. In the `module.h` file, we find:

```
#define module_CODE 0x1234abcf

/* Class definition (all attributes are public */
typedef struct {
      class *class;
      int id;
      str(name);
      str(binary);
      int status;
} module;
```

The `module_CODE` constant **MUST is a unique identifier** used to run-time type checking and class inheritance. The macro `str()` is defined in Section 6.1. Except the `*class` attribute, the rest of the elements of the structure form the public attributes of the `module` class.

The file ends with these two macro calls, which define the class:

```
CLASS_EXTEND(module,serializable,(module_serialize,module_unserialize));
CLASS_DEFINE_INHERITS(module,serializable);
```

These macros indicate that the class module extends the class serializable. The implemented methods are the third argument of the first macro, which must coincide with the `METHODS(serializable)` structure defined in the file `serializable.h`. If the class does not inherit any method, the declaration is just:

```
CLASS_DEFINE(module);
```

The class `serializable` declares his methods as abstract, defining in the file `serializable.h`

```
typedef struct {
      void (*serialize) (void*);
      void (*unserialize) (void*);
} METHODS(serializable);
```

Note that the order of the methods in this structure coincides with the order of the implemented methods in the `CLASS_EXTEND()` call in the module class.

The public attributes can also be inherited. In this case, the parent class must declare the following structure in the .h file:

```
typedef struct {
      int id;
      str(name);
      str(binary);
      int status;
} ATTRIBUTES(module);
```

whose order must coincide with the elements of the class structure. Attribute inheritance is then very simple:

```
typedef struct {
      class *class;
      ATTRIBUTES(module) parent;
      int another_public_method;
} man_module;
```

Note that a pointer of type `module` to an instance of type `man_module` can access the inherited methods exactly as it was a `module` instance.

Also of important consideration is the fact that, conversely to typical OO languages, in our framework attributes and methods are inherited independently. That means that one can inherit the attributes of a parent class without having to implement the parent's methods.

6.3.2 Static Object Allocation

All objects are statically allocated in memory. By definition, ALOE++ does not require dynamic memory support. All objects are passed **as reference** to any function. Furthermore, any function returning an object returns also **a reference** to it.

The HWAPI and SWAPI libraries define in the files `hwapi_arrays.h` and `swapi_arrays.h` the number of elements of each class that will be allocated, following the convention:

```
#define classname_MAX 20
```

The arrays objects in the HWAPI and SWAPI libraries are created using a macro defined in `metaclass.h`:

```
#define ALLOC(variable, class) class variable[class##_MAX]
#define MAX(class) class##_MAX
```

The number of objects allocated by the A-Node and A-Manager, on the other hand, can be defined in the platform configuration file.

6.3.3 Error Handling

HWAPI, SWAPI and CMDAPI Libraries

There are a set of standard rules for the HWAPI and SWAPI libraries regarding the errors produced in the library functions:

- All non-void functions return either an integer or a pointer.
- All void functions never produce an error.
- All integer functions return a non-negative integer (>=0) if the call was successful or -1 otherwise, except contrary specified.
- All pointer functions return a non-null pointer if the call was successful or `NULL` otherwise, except, contrary specified.
- If a function of the HWAPI library produces an error (i.e. returns -1 or `NULL`)
    - stores an error code in the `hwapi_error.code` variable, which can be obtained using the function:

      ```
      int hwapi_error_code();
      ```
    - the list of error codes is available in the `hwapi_error_codes` enumeration (part of the `hwapi_error` class).
    - The function

      ```
      void *hwapi_error_string();
      ```

      returns a pointer to a string describing the last error produced by a call to a library function.
- If a function of the SWAPI library produces an error (i.e. returns -1 or `NULL`)
    - stores an error code in the `swapi_error.code` variable, which can be obtained using the function:

      ```
      int swapi_error_code();
      ```
    - the list of error codes is available in the `swapi_error_codes` enumeration (part of the `swapi_error` class).
    - The function

      ```
      void *swapi_error_string();
      ```

      returns a pointer to a string describing the last error produced by a call to a library function.
- If a function of the CMDAPI library produces an error (i.e. returns -1 or `NULL`)
    - stores an error code in the `swapi_error.code` variable, which can be obtained using the function:

      ```
      int cmdapi_error_code();
      ```
    - the list of error codes is available in the `cmdapi_error_codes` enumeration (part of the `cmdapi_error` class).
    - The function

      ```
      void *cmdapi_error_string();
      ```

      returns a pointer to a string describing the last error produced by a call to a library function.

A-Node and A-Manager Classes

The rest of the classes do not follow exactly a uniform convention, however, when possible it is encouraged to:

- Follow the same convention of the HWAPI/SWAPI libraries regarding the returned value of non-void integer and pointer functions
- All errors messages are printed to the standard error message by the methods producing them
- The use of `assert()` in classes is encouraged for programmer's errors
- The use of the macro CAST() is also encouraged. This macro may be disabled in production code.
- The use of the goto scheme is encouraged. Each function must de-allocate all allocated resources if any error was detected, that is:

```
int foo(int bar)
{
      int return_value = 0;
      if (do_something( bar ) == -1) {
            goto error_1;
      }
      if (init_stuff( bar ) == -1) {
            goto error_2;
      }
      if (prepare_stuff( bar ) == -1) {
            goto error_3;
      }
      return_value = do_the_thing( bar );
error_3:
      cleanup_3();
error_2:
      cleanup_2();
error_1:
      cleanup_1();
      return return_value;
}
```

7 Further Reading

More information and documentation can be found at the OSLD website:

https://sites.google.com/site/osldproject/

and the ALOE++ GitHub website:

https://github.com/flexnets/aloe

These websites hosts the following documents:

- UML models, class diagrams and documentation:
  - https://sites.google.com/site/osldproject/documents
    - File: `ALOE_Classes_Documentation.pdf` contains the documentation for all classes, their methods and attributes
    - File: aloe_UML_model.xmi contains the UML model (class diagrams, package diagrama, platform deployment diagram, etc.)
- Class code example
  - https://github.com/flexnets/aloe/tree/master/Documentation/aloe_classes_exampleh
    - Is an Eclipse project. Clone and run make in Debug subdirectory to compile the example.