1. **Architecture Overview**

The code is a relatively simple because the purpose of the test. The architecture of the system breaks down into four main areas: **Tasks**, **Scheduler**, **Communication**, and **Hardware I/O** (devices).

* **Tasks:** this is the core of the RTOS implemented. A task is a class that will run in a different **thread** from the RTOS main thread. Created tasks could be managed by a **Scheduler** that will manage the order of the tasks to be run depending on the **Priority** and **State**. Only one task will be active at the same time.

Following some specifications about Tasks:

* + Each task is defined by a given priority: ***HIGH\_PRIORITY***, ***MID\_PRIORITY*** and ***LOW\_PRIORITY***.

// Finde different Priorities for the Tasks

#define HIGH\_PRIORITY 0x01

#define MID\_PRIORITY 0x02

#define LOW\_PRIORITY 0x04

* + Tasks can be **Stateless** or **Stateful**. For **Stateful** tasks there are several states defined:
    - **READY\_STATE**: The task is ready to be started by the **Scheduler**.
    - **WAIT\_STATE**: This state set the task to wait until the next action. This state could be used for pre-emptive interruption for High priority task.
    - **RUN\_STATE**: Current task is set to be run. This task, in pre-emptive mode, could be set to WAIT\_STATE, if higher priority task is waiting.
    - **NO\_STATE**: this is the default state after the creation of a Task. This means the task hasn’t started yet.

// Define different States for the Tasks

#define READY\_STATE 0x01

#define WAIT\_STATE 0x02

#define RUN\_STATE 0x04

#define NO\_STATE 0x08

* + For each task, a **Timer** (milliseconds) is set to be run at a certain time. This means the task will be executed at the next timer-tick. If the execution time is less than the timer, the task will wait (sleep) until the next timer-tick is triggered.
  + Task can be created by **inheritance** by overriding the function: virtual void \_default\_callback(void), or defining an custom **function** to run. Both methods are called when the **time-tick** is triggered. Custom functions has a parameter with the task itself in order to have access to all the behavior of a task.

Task watchdog("Watchdog", HIGH\_PRIORITY, 10000, WatchdogTask10000HZ);

void WatchdogTask10000HZ(Task\* task){

cout << "Verifying the System... ";

// Perform global verification of the System or Malfunctions

// Deadlocks, devices, disconnectiosns, updates, overheating, etc..

task->sleep(3000);

// System checked correctly. Shutdown? Corrective actions?

cout << "OK " << endl;

}

* + The process flow developed for the behavior of a task is implemented in the function: void \_callback\_process()

void Task::\_callback\_process(){

// Get the time-ticks for the current task

std::chrono::system\_clock::time\_point start\_time, end\_time;

uint64\_t timediff;

// Set the current Task as READY\_STATE

\_state = READY\_STATE;

// Start the thread loop for the current task

while (\_running) {

// Check if the task must be launched

if (\_state != RUN\_STATE && !\_stateless) {

//Wait until run state is on

sleep(10);

continue;

}

// Get the start time

start\_time = get\_current\_time();

// Check whether the thread has been set to run a callback

if (\_callback){

// Call to the function directly

\_callback(this);

}

else {

// Call to the default function to override

\_default\_callback();

}

// Get the end time

end\_time = get\_current\_time();

// Check until the next tick for perform the task

timediff = std::chrono::duration\_cast<std::chrono::milliseconds>

(end\_time - start\_time).count();

// Check if the Task need sleep some time before the next iteration

if (timediff < \_timer)

\_state = WAIT\_STATE;

sleep(\_timer - timediff);

// Set the current Task as READY\_STATE

\_state = READY\_STATE;

}

}

* + Source code for the class task is implemented in:
    - \src\autopilot\rtos\task.cpp
    - \src\autopilot\rtos\task.h
* **Scheduler**: The thread scheduler selects the next task to run by looking at the priority assigned to every task that is STATE\_READY. The task with the highest priority is selected to be executed.

Scheduler Class is defined in files: src\autopilot\rtos\scheduler.h and src\autopilot\rtos\scheduler.cpp . Scheduler implements three priority queues to manage the tasks: \_high\_priority, \_mid\_priority and \_low\_priority.

class Scheduler : public Task {

public:

Scheduler(uint64\_t timer = 10);

~ Scheduler();

void add(shared\_ptr<Task> task);

private:

//Function to overwrite

void \_default\_callback();

shared\_ptr<Task> \_find\_next();

// Current Task

shared\_ptr<Task> \_current = nullptr;

// Priority queues

**queue<shared\_ptr<Task>> \_high\_priority;**

**queue<shared\_ptr<Task>> \_mid\_priority;**

**queue<shared\_ptr<Task>> \_low\_priority;**

};

The workflow is the following. RTOS creates a Scheduler instance so it will manage all the tasks and the order that will be executed depending on the priority. Tasks are created in the initialization of RTOS and the scheduler start when RTOS start. Each Task will start as soon as the scheduler detect it is the next one.

The basic process to create a Scheduler is the following:

1. Create scheduler. (Timer is set to 10 milliseconds)

Scheduler myScheduler(10);

1. Add needed Tasks to the scheduler

// TSIP task, 200Hz, Non highest nor lowest.

myScheduler.add(Task("Uplink", MID\_PRIORITY, 200, TaskUplink200Hz, packet\_queue));

// Display Task to show the time

myScheduler.add(Task("Display",LOW\_PRIORITY, 1000, UpdateDisplay1000Hz));

1. Start the scheduler

myScheduler.start();

In order the find the next task to execute, the function Task \_find\_next(), will look over the three priority queues, from the highest priority to the lowest priority, and it will look for the next Task to run. The function will remove the item from the current queue and will queue again into the back.

shared\_ptr<Task> Scheduler::\_find\_next(){

shared\_ptr<Task> \_result = nullptr;

// This function will iterate over all the list to get the next

// task to activate. The states will be READY\_STATE or NO\_STATE

for (int i=0;i<3;i++){

queue<shared\_ptr<Task>> \*current\_queue;

switch (i) {

case 0:

current\_queue = &\_high\_priority;

break;

case 1:

current\_queue = &\_mid\_priority;

break;

case 2:

current\_queue = &\_low\_priority;

break;

}

// Check the current size of the queue

if (current\_queue->size()>0) {

// For each element get the next Task to run

shared\_ptr<Task> first\_task = current\_queue->front();

do {

shared\_ptr<Task> current\_task = current\_queue->front();

// Extract the current Task from the queue

current\_queue->pop();

// Put into the queue again (At the end)

current\_queue->push(current\_task);

// Check the current state of the task

if (current\_task->get\_state()==NO\_STATE ||

current\_task->get\_state()==READY\_STATE){

\_result = current\_task;

break;

}

}while (current\_queue->front()->get\_name()!=first\_task->get\_name());

// Check if we have found a task to run

if (\_result)

break;

}

}

// Return the current result

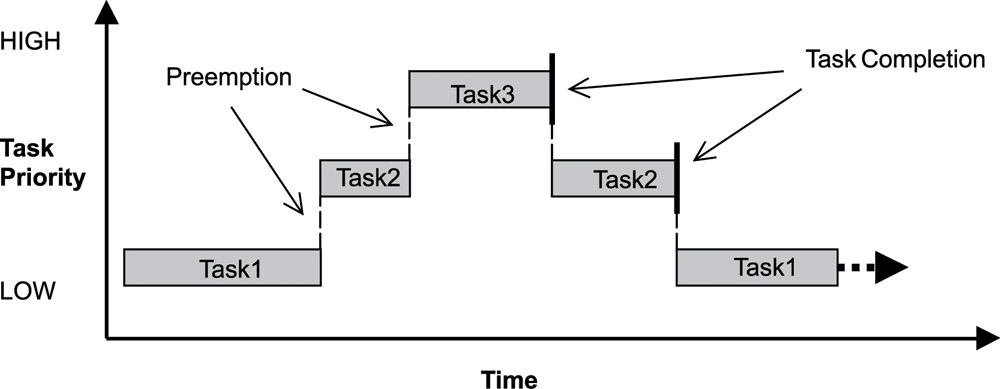
return \_result;

}

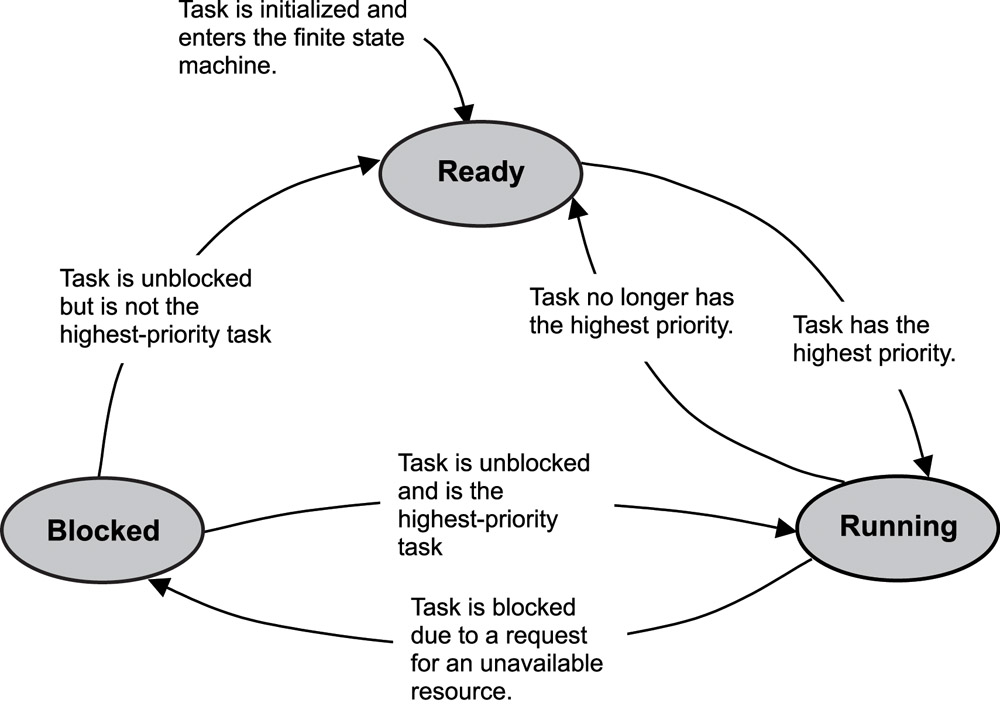
In current implementation, **Priority Inversion** are not totally implemented but could be easily integrated into the Scheduler workflow by pausing (WAIT\_STATE) the lower priority task being run, execute the higher priority, and finally continue with the execution of pending tasks.

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **priority inversion** is a problematic scenario in [scheduling](https://en.wikipedia.org/wiki/Scheduling_%28computing%29) in which a high priority [task](https://en.wikipedia.org/wiki/Task_%28computing%29) is indirectly [preempted](https://en.wikipedia.org/wiki/Preemption_%28computing%29) by a lower priority task effectively "inverting" the relative priorities of the two tasks. This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks.

Following there is a basic figure showing the proposed current solution for **pre-emption**:



There are another ways to solve the **Priority Inversion**. An alternative is the P**riority Inheritance Protocol**, a variation that uses dynamic priority adjustments. When a low-priority task acquires a shared resource, the task continues running at its original priority level. If a high-priority task requests ownership of the shared resource, the low-priority task is hoisted above the requesting task. The low-priority task can then continue executing its critical section until it releases the resource. Once the resource is released, the task is dropped back to its original low-priority level, permitting the high-priority task to use the resource it has just acquired.



* **Communication:** Communication and data transfer between tasks are accomplished by the use of the parameters and the use of the Queues implemented. Tasks use queues to send data to each other and to signal the use of critical resources using locks for critical sections. For this problem, I have used the producer-consumer pattern.

In [computing](https://en.wikipedia.org/wiki/Computing), the **producer–consumer problem**[[1]](https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem#cite_note-ostep1-1)[[2]](https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem#cite_note-ostep2-2) (also known as the **bounded-buffer problem**) is a classic example of a multi-[process](https://en.wikipedia.org/wiki/Process_%28computing%29) [synchronization](https://en.wikipedia.org/wiki/Synchronization_%28computer_science%29) problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size [buffer](https://en.wikipedia.org/wiki/Buffer_%28computer_science%29) used as a [queue](https://en.wikipedia.org/wiki/Queue_%28data_structure%29). The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Queues basically implements the following methods:

template <typename T>

class Queue {

public:

void push(T item);

T pop();

uint32\_t size();

bool empty();

};

All methods are Thread-safe, which means only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfil their design specifications without unintended interaction.

The way that Task used the Queues to communicate each other is by doing the following:

1. Create the Shared queue and the tasks that will shared the data.

Queue<string> \*packet\_queue = new Queue<string>();

// Sokets for TCPIP/COM connection

Task producer("Producer", MID\_PRIORITY, 400, Producer200Hz, packet\_queue));

// TSIP task, 200Hz, Non highest nor lowest.

Task consumer("Consumer", MID\_PRIORITY, 200, Consumer200Hz, packet\_queue));

1. Finally the callback for each Task, will manage the queue retrieved from the parameters:

void Producer200Hz(Task\* task){

Queue<string> \*packet\_queue = (Queue<string>\*) task->get\_parameters();

// Check the queued TSIP packets, parse them and perform the action.

packet\_queue->push(“6 (4 ounce) links sweet Italian sausage”);

packet\_queue->push(“2 tablespoons butter”);

packet\_queue->push(“1 yellow onion, sliced”);

packet\_queue->push(“1/2 red onion, sliced”);

}

void Consumer200Hz(Task\* task){

Queue<string> \*packet\_queue = (Queue<string>\*) task->get\_parameters();

// Check the queued items

while (!packet\_queue->empty()) {

// Get the next packet to parse

Cout << packet\_queue->pop() << endl;

// Do somthing after

task->sleep(10);

}

}

* **Hardware I/O:** This part is hardware-dependent so in the test specifications this is not needed to be implemented. However I have designed some classes so Hardware Interfacing could be relative easy to use.

Also this separation, it helps to **TDD** approach to work efficiently as we are mocking up the interfaces without being using the actual Hardware.

I have created a new package called “**devices**” in src\autopilot\devices, that implement the different interfaces between the devices and the RTOS. In this particular case I have implemented two modules interfaces:

* src\autopilot\devices\com.h

This module is supposed to manage the communication between the RTOS and the Serial Port. In this particular case there is only one method implemented.

/\*\*

\* \brief Read received raw data from COM port.

\*

\* This is a non-blocking read function. This means that only available received

\* data will be served. User may decide to call this function within a loop

\* until the desired amount of data is received.

\*

\* \param buf Pointer to destination buffer where to copy received data.

\* \param count Maximum number of bytes to be read.

\* \return Number of read bytes. This value will always be <= count.

\*/

int32\_t uavnComRead(uint8\_t \* const buffer, const uint32\_t count);

The function will emulate the reception of messages received from a Serial Port in **TSIP format**. The function implemented in src\autopilot\devices\com.cpp, will return a buffer with data and the number of bytes read. For this particular case, there are defined several messages that randomly it will select a few of them until fulfills the buffer limit size by using count number parameter.

* src\autopilot\devices\tsip.h

In this interface the TSIP messages will be handled. The interface defined has the following structure:

// TSIP Byte codes.

#define ETX 0x03

#define DLE 0x10

// TSIP Packet Size

#define TSIP\_SIZE 256

// Define TSIP data for packet management

struct TsipPacket {

uint8\_t data[TSIP\_SIZE];

int32\_t size = 0;

bool active = false;

};

/\*\*

\* \brief Parse a TSIP data.

\*

\* \param[in] buffer Const pointer where data is located.

\* \param[in] numberOfBytes It is filled with the number bytes in the TSIP packet.

\*/

void ParseTsipData(const uint8\_t \* const buffer, const int32\_t numberOfBytes);

The method implemented will process such buffer with the TSIP data and it will perform the parsing and the required autopilot actions. In the current implementation the code will print only the content in console terminal.

**2. RTOS**

For the current test the following function was requested to be implemented:

void TaskUplink200Hz(void) {

// Fill this function. You may use additional functions if needed.

}

Previous function (task) has been divided in two separated functions (tasks) that allow the abstraction between the raw data received from the Serial Port and the TSIP data. These two tasks are DMARead200Hz and TaskUplink200Hz.

Also, in the current RTOS there are implemented another different Tasks in order to perform systems checks for Hardware failures and display functions.

Following are the tasks that have been created:

// Create shared queue to be passed between tasks socket and tsip

Queue<TsipPacket\*> \*packet\_queue = new Queue<TsipPacket\*>()

// Initilaize the different subsystems, drivers, etc..

Task(Watchdog", HIGH\_PRIORITY, 10000, WatchdogTask10000HZ);

// Sokets for TCPIP/COM connection

Task(DMARead", MID\_PRIORITY, 400, DMARead200Hz, packet\_queue);

// TSIP task, 200Hz, Non highest nor lowest.

Task(Uplink", MID\_PRIORITY, 200, TaskUplink200Hz, packet\_queue);

// Display Task to show the time

Task(Display",LOW\_PRIORITY, 1000, UpdateDisplay1000Hz);

1. **UpdateDisplay1000Hz**

This task performs the update of a display connected to the system. For the implementation, the output will be printed into the Console.

/\* UpdateDisplay1000Hz (LOW-PRIORITY)

Update the display showing the information needed.

The display will be updated every 1 second.

\*/

void UpdateDisplay1000Hz(Task\* task){

system("cls");

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\* RTOS \*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

time\_t \_tm =time(NULL );

struct tm \* curtime = localtime ( &\_tm );

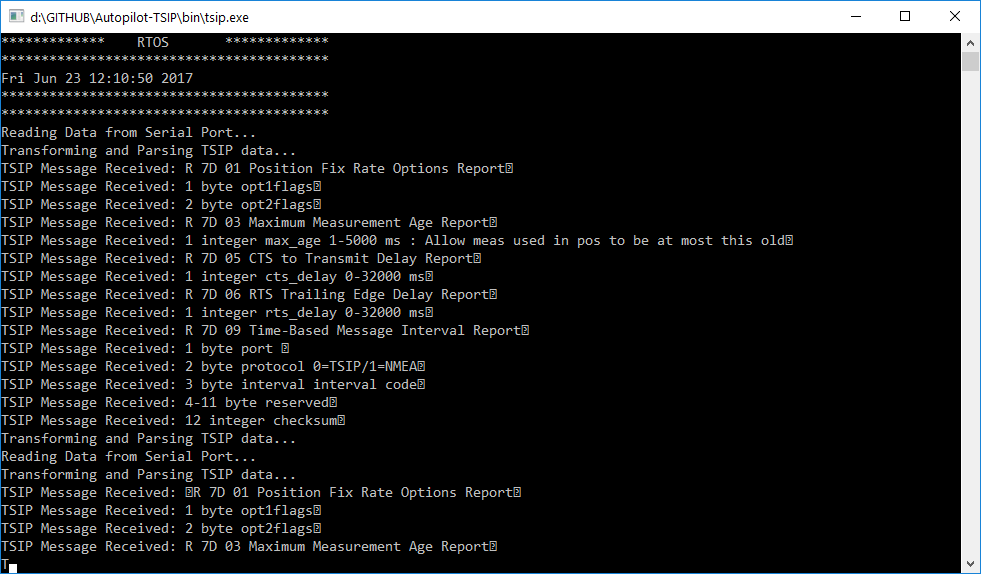
cout<< asctime(curtime);

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

}

This task is the lowest one, so it will be executed only if other tasks are not ready yet. The display will be updated each one second showing following information.



1. **WatchdogTask10000HZ**

This is the task with highest priority. It will check the entire system, such as controllers, devices, threads, etc. to ensure the system is working properly. Also this task will take actions depending on the failures founded.

/\* WatchdogTask10000HZ (HIGH-PRIORITY)

This task is used o detect and recover from computer malfunctions. During

normal operation, the computer regularly resets the watchdog timer to

prevent it from elapsing, or "timing out".The timeout signal is used to

initiate corrective action or actions. The corrective actions typically

include placing the computer system in a safe state and restoring normal

system operation.

\*/

void WatchdogTask10000HZ(Task\* task){

cout << "Verifying the System... ";

// Perform global verification of the System or Malfunctions

// Deadlocks, devices, disconnectiosns, updates, overheating, etc..

task->sleep(3000);

// System checked correctly. Shutdown? Corrective actions?

cout << "OK " << endl;

}

1. **DMARead200Hz**

This task will read the data from Serial Port, using the com.h module. This function extracts all TSIP packets from the buffer using uavnComRead and queue them into a Thread-safe queue used to share information between this task with the TaskUplink200Hz.

The task will store the information of previous execution to ensure there is no gap or missed data from different read from Serial Port. If this is the case the task will continue with previous TSIP packet that was being processed.

/\* DMARead200Hz (MEDIUM-PRIORITY)

Following task is going to read de TSIP packets from the serial Port.

The serial port is read using DMA with a Buffer of 512

\*/

void DMARead200Hz(Task\* task){

cout << "Reading Data from Serial Port... " << endl;

// Get the current Queue to share the params

Queue<TsipPacket\*> \*packet\_queue = (Queue<TsipPacket\*>\*) task->get\_parameters();

// Create default variables

uint8\_t read\_byte, DMA\_buffer[512];

// Check if last TSIP message was interrupted

TsipPacket \*packet = nullptr;

if (task->get\_data()){

// Get the last incomplete TSIP messages from the task

packet = (TsipPacket\*)task->get\_data();

} else {

// Create a new one.

packet = new TsipPacket();

}

// Get next batch of bytes from DEM Serial Port.

int32\_t byte\_count = uavnComRead(DMA\_buffer, sizeof DMA\_buffer);

// Loop through buffer read from Serial Port

for (int32\_t i = 0; i < byte\_count; i += 1) {

// Get the current byte

read\_byte = DMA\_buffer[i];

// Check if a TSIP packet has already started

if (packet->active) {

// Check if final of the current TSIP packets

if (read\_byte == ETX) {

// Add current packet to the quaue

packet\_queue->push(packet);

// Start new TSIP packet

packet = new TsipPacket();

} else {

// Set the current byte into the TSIP buffer

packet->data[packet->size++] = read\_byte;

}

} else if (read\_byte == DLE) {

// New start TSIP packet detected

packet->active = true;

}

// End TSIP packet not detected (DLE ETX)

if (packet->size >= TSIP\_SIZE) {

// Reset current TSIP Packet from start (noise in data)

packet->size = 0;

packet->active = false;

}

}

// Append current (uncompleted) TSIP data to the task

task->set\_data(packet);

}

1. **TaskUplink200Hz**

This task will consume the information stored in the queue being managed by previous Task. This task will loop over all the items in the queue in same order that they were added. Also the items will be de-queued and removed from the collection.

For each TSIP packets the task will call to tsip.h module so the packets are processed. The function ParseTsipData will parse the TSIP packet and perform an action if needed.

/\* TaskUplink200Hz (MEDIUM-PRIORITY)

Once the TSIP data is extracted from TSIP packet. The following task

will process such TSIP data, performing the required autopilot actions.

\*/

void TaskUplink200Hz(Task\* task){

cout << "Transforming and Parsing TSIP data... " << endl;

Queue<TsipPacket\*> \*packet\_queue = (Queue<TsipPacket\*>\*) task->get\_parameters();

// Check the queued TSIP packets, parse them and perform the action.

while (!packet\_queue->empty()) {

// Get the next packet to parse

TsipPacket \*packet = packet\_queue->pop();

// Parse the current packet and perform the action

ParseTsipData(packet->data, packet->size);

// Delete current item since not a smart object

delete packet;

// Do somthing after parsing TSIP packet and perform action

task->sleep(10);

}

}

**3. Test Driver Development (TDD)**

Previous architecture was designed taking into account the **TDD** approach, that consists in write tests first and, then writing enough code to pass those tests.

The idea was to Mock the interfaces implemented for the devices or controllers, so we don’t need the actual Hardware to perform the tests.

For this purpose have been implemented two test cases in order to test the COM and TSIP functionalities.

* src\autopilot\devices\test\_tsip.cpp

In this files there is only one test that will check the process of parsing TSIP data. If exception test will fail and return fail.

/\*

Following test will parse the TSIP data. If exception test will fail

\*/

bool test\_ParseTsipData(){

try {

//TsipPacket(2, 0x8E, 0xA9); // self-survey parameters

//TsipPacket(2, 0x8E, 0x41); // manufacturing operating parameters

//TsipPacket(2, 0x8E, 0x42); // production parameters

TsipPacket packet;

packet.data[0] = 0x8E;

packet.data[1] = 0xA9;

packet.size = 2;

ParseTsipData(packet.data, packet.size);

} catch (std::ex const std::exception& e){

// Test failed to return the TSIP data

return false;

}

}

* src\autopilot\devices\test\_com.cpp

In this module there are implemented two tests. One to check if the connection fails and the other to check read data are greater than 0. These two function will ensure the Serial Port is working properly.

/\*

Check no exception reading Serial Port

\*/

bool test\_uavnComRead(){

try {

//Create an empty buffer

buffer[512];

//Call to the function

uavnComRead(buffer, sizeof buffer);

// Check the number of bytes

} catch (std::ex const std::exception& e){

// Test failed to return the TSIP data

return false;

}

}

/\*

Check if the Bytes read from Serial Port is > 0

\*/

bool test\_bytes\_read\_uavnComRead(){

try {

//Create an empty buffer

buffer[512];

//Call to the function

int32\_t count = uavnComRead(buffer, sizeof buffer);

// Check the number of bytes read from COM

if (count <= 0) return false;

return true;

} catch (std::ex const std::exception& e){

// Test failed to return the TSIP data

return false;

}

}