NATIONAL UNIVERSITY OF HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FALCUTY OF TRANSPORTATION ENGINEERING

DEPARMENT OF AEROSPACE ENGINEERING

-------o0o-------

Report #3.1

Integrate new controller code into ArduPilot

Student: Nguyễn Anh Quang

Student ID: V1002583

Supervisor: Prof Emmanuel GROLLEAU

Poitier, Spring 2015

Abstract

In the previous report, the method to embed new controller code into ArduPilot has been introduced. However, this solution has some drawbacks, particularly in the way changing between different controllers. This report will present a developed module to integrate new controllers, the details about how this module is implemented and how to add new control algorithm into it.

Mục lục - Table of Contents

Danh mục hình ảnh - List of Figures

Chương 1

A new module to embed new controller code

This part of the report presents the general idea of a module to help developers to integrate new controller code into ArduPilot framework. This chapter will also show the implementation of this module to help users in the future develop it further or replace it with a more practical solution if possible.

1.1 Overview

Figure 1-1 introduces the general idea of this module.

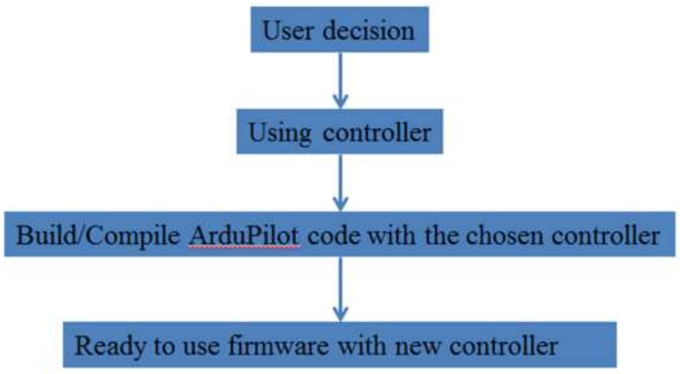

User decision
Using controller
Build/Compile ArduPilot code with the chosen controller
Ready to use firmware with new controller

Figure 1-1: How the new module should work

The most important idea about this module is that users can change the embedded controllers with just a simple switch. This solution will not only reduce the amount of future work while integrating new controllers but also simplify the controller integrating procedure, make it easier for developers to test and conclude the pros and cons of each controller.

Figure 1-2 gives an introduction about the generated module to integrate new controller into ArduPilot.

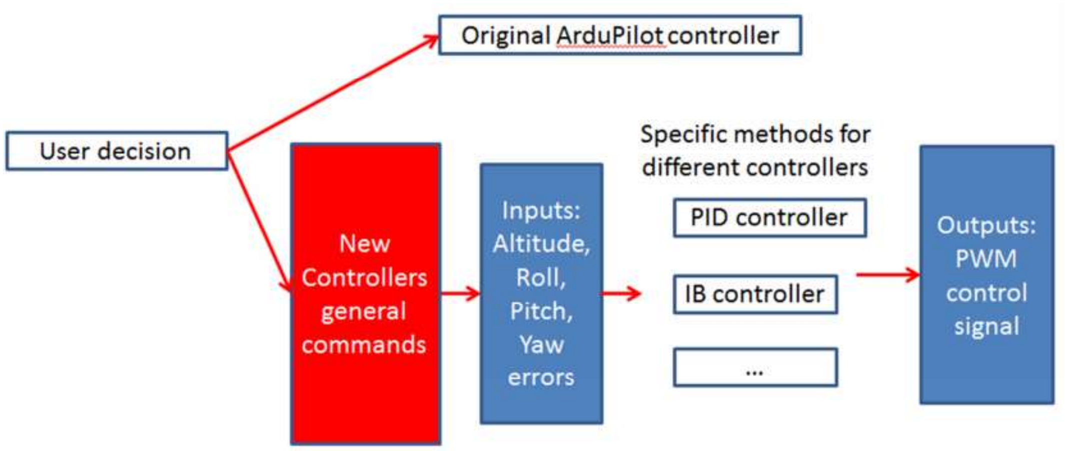

Original ArduPilot controller
User decision
New Controllers general commands
Inputs: Altitude, Roll, Pitch, Yaw errors
Specific methods for different controllers
PID controller
IB controller
...
Outputs: PWM control signal

Figure 1-2: Overview of the controller-integrating module

Because of the differences of ArduPilot original controller comparing with other controllers generated in the previous work of this project, which have been mentioned in **Report#3**, while users decide to use the original code, this module will be neglected and the built firmware will work as the original ArduPilot. In other case, for example, the decision is using the new PID controller, this module will redirect the calculation process as shown above. Since the inputs (altitude, roll, pitch, yaw error) and outputs (PWM signal to control the motors) of new controllers are the same, the general commands will be used to get the inputs as well as send the outputs to motors as in **figure 1-2**.

1.2 Implement the module

There are several solutions to implement this controller-integrating module. One of the solutions is the multi-layered structure. Multi-layer structure, as presented in the report about ArduPilot, is one of the most practical ways when developers need to implement a multiple targets application. As shown in **figure 1-3**, just by generating the code for the virtual controller class, developers can make the firmware for many other real controllers and the using controller can be changed just by a simple code line.

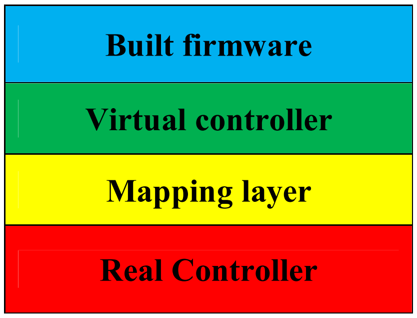


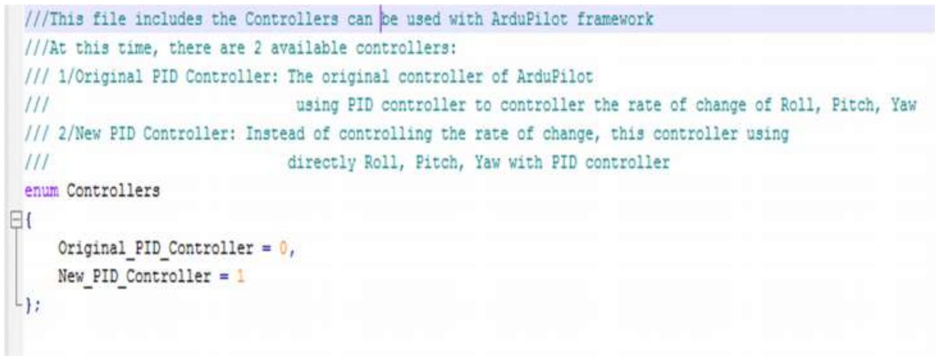
Figure 1-3: Multi-layer structure to switching between controllers

This multi-layer architecture has been successfully implemented into ArduPilot, which makes this framework become one of the most popular frameworks for compiling autopilot firmware for drones.

However, this solution has some drawbacks. The complexity of the code structure is the major disadvantage. In order to using this multi-layer structure, many code lines need to be generated, not only for the virtual class but also for the mapping layers. Due to it complication, this architecture is also not very friendly for new users, some of which are not familiar with C/C++ code. More importantly, this solution requires many modifications each time a controller need to be integrated into ArduPilot. Since this solution require the mapping layer for each controller, developers will need to do more than just adding new controllers in C/C++ code and then use them directly. Although this multi-layer structure will have many benefits as shown in the ArduPilot structure in **Report#1**, for a simple module like this, it is not very effective.

Another way to generate this module, which is the chosen one in this project, is using “switch” and “if” command. Because of its simplicity, it not only reduces the effects of this new module to ArduPilot code structure but also makes it easier for future work to integrate other controllers. Additionally, it requires just basic knowledge about C/C++ of developers, unlike the multi-layer architecture above requiring highly understanding in the structure of C/C++. The procedure of creating and embedding the controller-integrating module into ArduPilot framework includes the following steps:

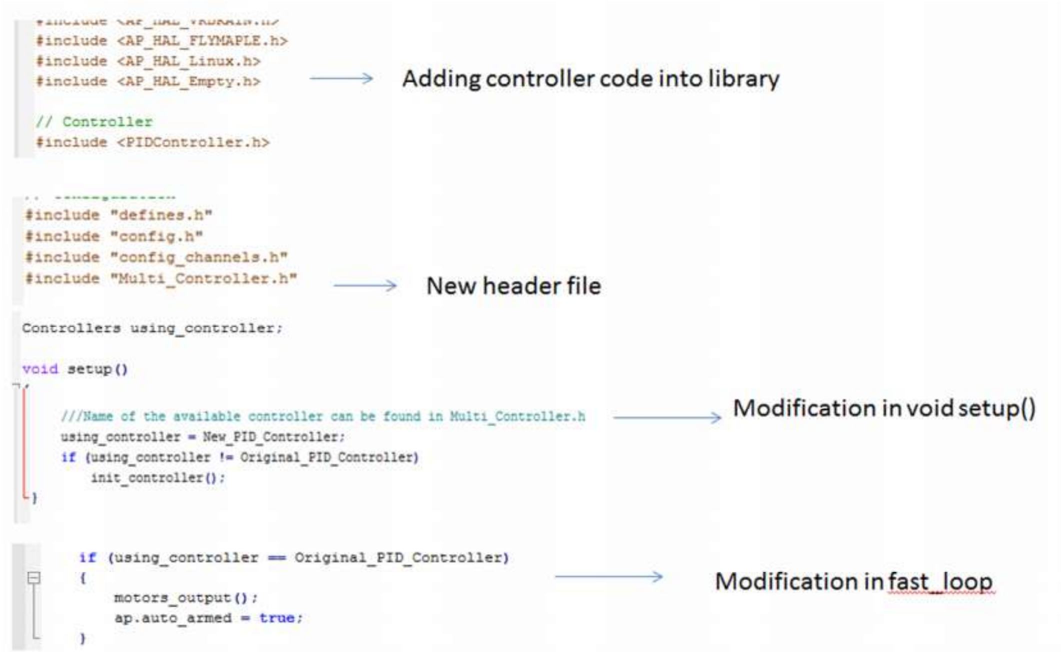
- Create a ***.h** file in the ArduCopter directory containing the controllers can be used. Until now, there are only two controllers, the default controller of ArduPilot and the new PID controller.



```
///This file includes the Controllers can be used with ArduPilot framework
///At this time, there are 2 available controllers:
/// 1/Original PID Controller: The original controller of ArduPilot
///                              using PID controller to controller the rate of change of Roll, Pitch, Yaw
/// 2/New PID Controller: Instead of controlling the rate of change, this controller using
///                              directly Roll, Pitch, Yaw with PID controller
enum Controllers
{
    Original_PID_Controller = 0,
    New_PID_Controller = 1
};
```

Figure 1-4: A header file defines the name for different controllers

- Modify code in ArduCopter.pde as shown in **figure 1-5**. Depending on the **using_controller**, the module will change the calculation process as described in **figure 1-2**.



```
#include <AP_HAL_FLYMAPLE.h>
#include <AP_HAL_Linux.h>
#include <AP_HAL_Empty.h>

// Controller
#include <PIDController.h>
```

```
#include "defines.h"
#include "config.h"
#include "config_channels.h"
#include "Multi_Controller.h"
```

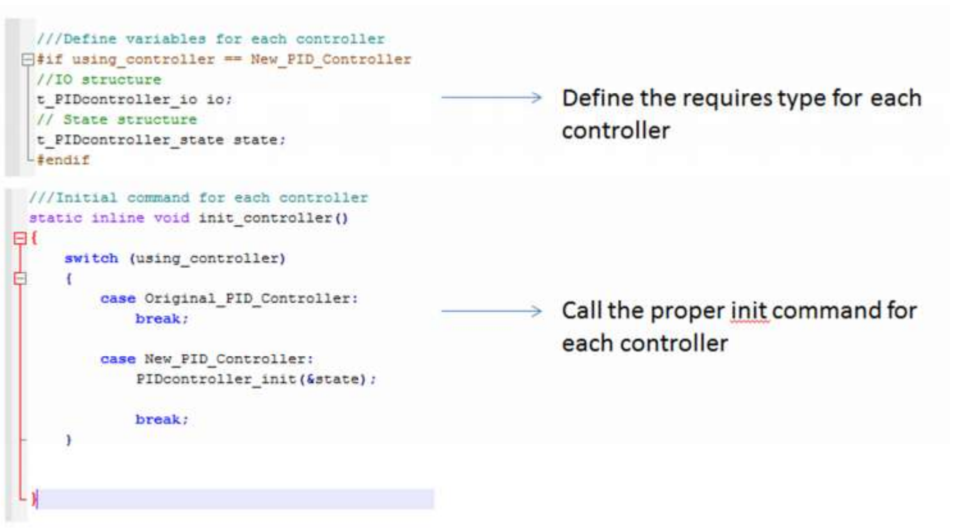
```
Controllers using_controller;

void setup()
{
    ///Name of the available controller can be found in Multi_Controller.h
    using_controller = New_PID_Controller;
    if (using_controller != Original_PID_Controller)
        init_controller();
}
```

```
    if (using_controller == Original_PID_Controller)
    {
        motors_output();
        ap.auto_armed = true;
    }
```

Adding controller code into library
New header file
Modification in void setup()
Modification in fast_loop

Figure 1-5: Some changes in ArduCopter.pde

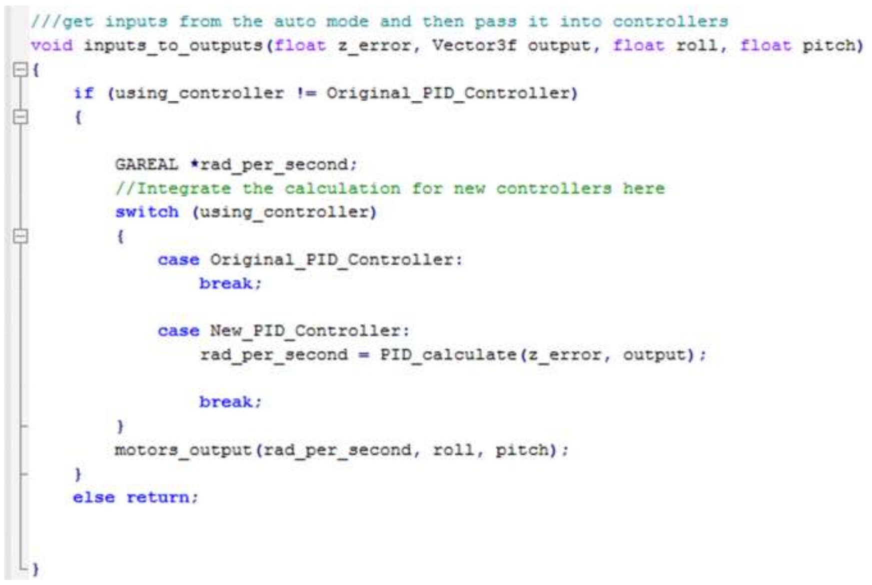
- Create the general methods for the new controllers. There are three general methods: **init_controller** defines the initial values; **inputs_to_outputs** gets the required inputs, parses them into the calculation and then send the outputs to the motors calculation; **motors_output** converts outputs of the new controller into angular speed rad/s and send them to the PWM motors calculation. All of these methods will be put into a new pde file in the ArduCopter directory. **Figure 1-6**, **fig 1-7** and **fig 1-8** shows the code of each method respectively.


Define the requires type for each controller
Call the proper init command for each controller

```
///Define variables for each controller
#if using_controller == New_PID_Controller
//IO structure
t_PIDcontroller_io io;
// State structure
t_PIDcontroller_state state;
#endif
```

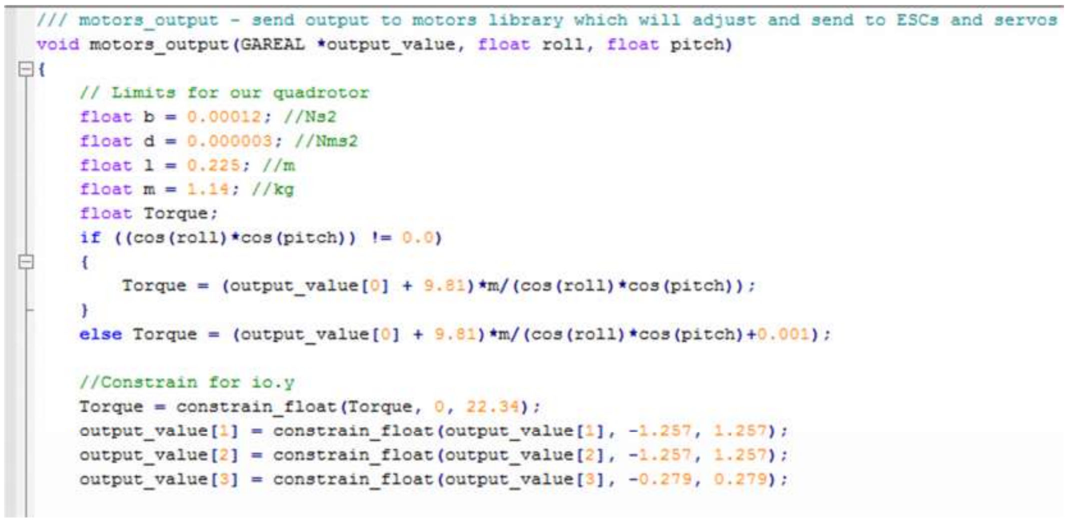
```
///Initial command for each controller
static inline void init_controller()
{
    switch (using_controller)
    {
        case Original_PID_Controller:
            break;

        case New_PID_Controller:
            PIDcontroller_init(&state);

            break;
    }

}
```

Figure 1-6: Initial code for each controller



```
///get inputs from the auto mode and then pass it into controllers
void inputs_to_outputs(float z_error, Vector3f output, float roll, float pitch)
{
    if (using_controller != Original_PID_Controller)
    {

        GAREAL *rad_per_second;
        //Integrate the calculation for new controllers here
        switch (using_controller)
        {
            case Original_PID_Controller:
                break;

            case New_PID_Controller:
                rad_per_second = PID_calculate(z_error, output);

                break;
        }
        motors_output(rad_per_second, roll, pitch);
    }
    else return;


}
```

Figure 1-7: From the general command inputs-outputs command to the specific one for each controller

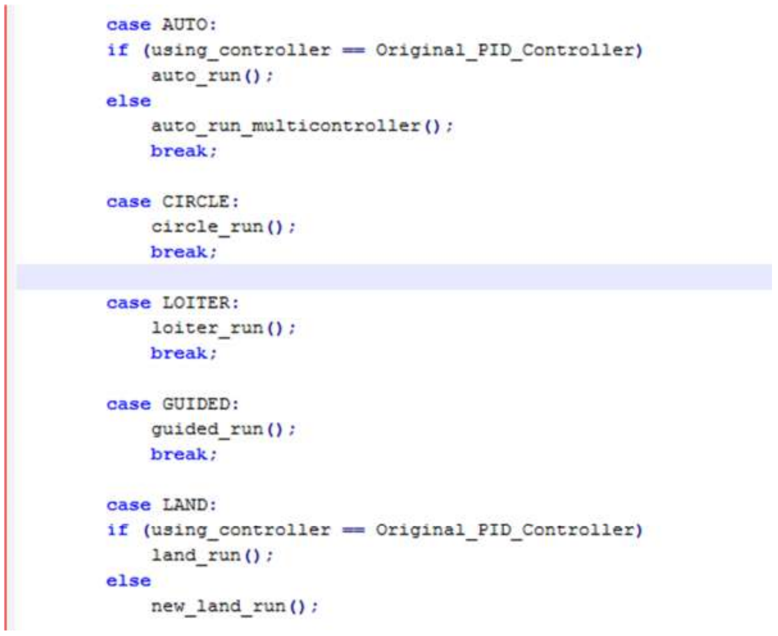


```
/// motors_output - send output to motors library which will adjust and send to ESCs and servos
void motors_output(GAREAL *output_value, float roll, float pitch)
{
    // Limits for our quadrotor
    float b = 0.00012; //Ns2
    float d = 0.000003; //Nms2
    float l = 0.225; //m
    float m = 1.14; //kg
    float Torque;
    if ((cos(roll)*cos(pitch)) != 0.0)
    {
        Torque = (output_value[0] + 9.81)*m/(cos(roll)*cos(pitch));
    }
    else Torque = (output_value[0] + 9.81)*m/(cos(roll)*cos(pitch)+0.001);

    //Constrain for io.y
    Torque = constrain_float(Torque, 0, 22.34);
    output_value[1] = constrain_float(output_value[1], -1.257, 1.257);
    output_value[2] = constrain_float(output_value[2], -1.257, 1.257);
    output_value[3] = constrain_float(output_value[3], -0.279, 0.279);
```

Figure 1-8: From outputs of controller to PWM signals (some of the code is removed)

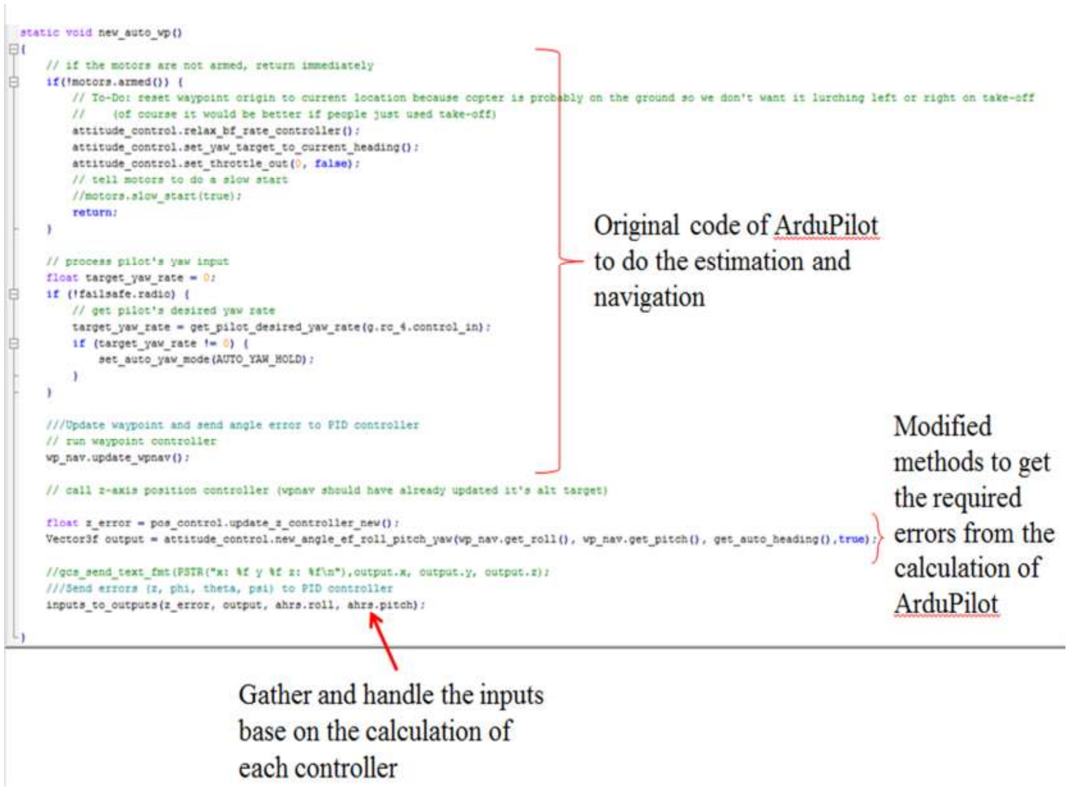
- Modify the original code of ArduPilot to get the inputs.

As mentioned in the introduction of ArduPilot navigation and control in **Report#3**, some modifications have to be done to access the inputs (altitude, roll, pitch, yaw error). Since the navigation and estimation to get the desired values is different with each flight mode, in order to get the right inputs for new controllers, the flight mode code will need to be changed, as described in the following figures.



```
case AUTO:
if (using_controller == Original_PID_Controller)
    auto_run();
else
    auto_run_multicontroller();
    break;

case CIRCLE:
    circle_run();
    break;

case LOITER:
    loiter_run();
    break;

case GUIDED:
    guided_run();
    break;

case LAND:
if (using_controller == Original_PID_Controller)
    land_run();
else
    new_land_run();
```

Figure 1-9: Modifications in flight_mode.pde



```
static void new_auto_wp()
{
    // if the motors are not armed, return immediately
    if(!motors.armed()) {
        // To-Do: reset waypoint origin to current location because copter is probably on the ground so we don't want it lurching left or right on take-off
        //    (of course it would be better if people just used take-off)
        attitude_control.relax_bf_rate_controller();
        attitude_control.set_yaw_target_to_current_heading();
        attitude_control.set_throttle_out(0, false);
        // tell motors to do a slow start
        //motors.slow_start(true);
        return;
    }

    // process pilot's yaw input
    float target_yaw_rate = 0;
    if (!failsafe.radio) {
        // get pilot's desired yaw rate
        target_yaw_rate = get_pilot_desired_yaw_rate(g.rc_4.control_in);
        if (target_yaw_rate != 0) {
            set_auto_yaw_mode(AUTO_YAW_HOLD);
        }
    }

    ///Update waypoint and send angle error to PID controller
    // run waypoint controller
    wp_nav.update_wpnav();

    // call z-axis position controller (wpnav should have already updated it's alt target)

    float z_error = pos_control.update_z_controller_new();
    Vector3f output = attitude_control.new_angle_ef_roll_pitch_yaw(wp_nav.get_roll(), wp_nav.get_pitch(), get_auto_heading(),true);

    //gcs_send_text_fmt(PSTR("x: %f y %f z: %f\n"),output.x, output.y, output.z);
    ///Send errors (z, phi, theta, psi) to PID controller
    inputs_to_outputs(z_error, output, ahrs.roll, ahrs.pitch);
}
```

Original code of ArduPilot to do the estimation and navigation
Modified methods to get the required errors from the calculation of ArduPilot
Gather and handle the inputs base on the calculation of each controller

Figure 1-10: A simple modified flight mode to use new controllers

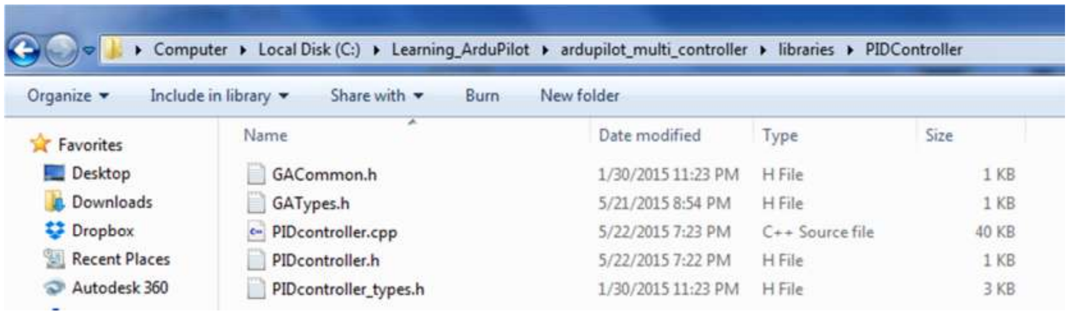
- Create a conversion method to transfer from angular speed into PWM signal. The idea of this command can be found in the previous report.

After finishing the above steps, a new module to integrate new controllers into ArduPilot has been created as well as fully embedded. The work from now on will be more simply, which is adding new controllers into this module and testing them with the simulation or real flight tests.

1.3 Add new controller into the new module.

In order to integrate a new controller into the described module, developers should follow the below procedure:

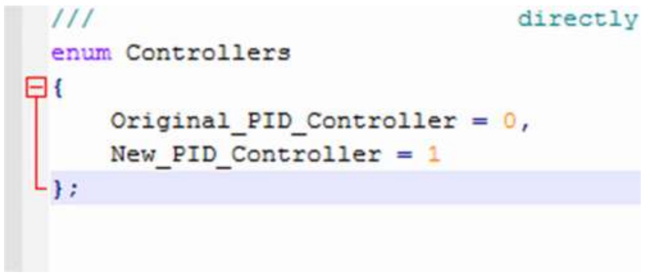
- Generate C/C++ code for the control algorithm and put the code into the library directory of ArduPilot. These code files can be programmed manually or using a support tools to generate the code from a control model such as Gene-auto or the Code generator of Matlab.



Name	Date modified	Type	Size
GACommon.h	1/30/2015 11:23 PM	H File	1 KB
GATypes.h	5/21/2015 8:54 PM	H File	1 KB
PIDcontroller.cpp	5/22/2015 7:23 PM	C++ Source file	40 KB
PIDcontroller.h	5/22/2015 7:22 PM	H File	1 KB
PIDcontroller_types.h	1/30/2015 11:23 PM	H File	3 KB

Figure 1-11: C/C++ code of the PID controller in the library of ArduPilot

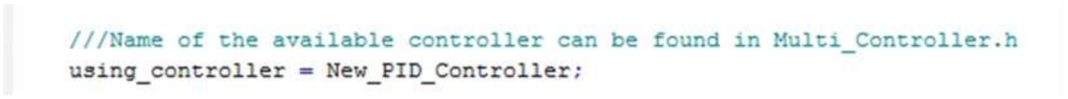
- Add the name of this controller in the **Multi_Controller.h** file as in figure 1-12



```
///                                    directly
enum Controllers
{
    Original_PID_Controller = 0,
    New_PID_Controller = 1
};
```

Figure 1-12: Decide the name to call the new Controller

- Change the using_controller in the main code of ArduCopter.pde



```
///Name of the available controller can be found in Multi_Controller.h
using_controller = New_PID_Controller;
```

Figure 1-13: Choose the controller

- Add the specific methods of the new controller into the structure of the general command as described in **figure 1-6** and **fig 1-7** above.

The integration is done and now the new controller has become a part of the controller-integrating module. By switching the **using_controller** in the main code, developers can compile and build the new firmware using any embedded controller. As can be seen, although the procedure creating the tool to integrate the controllers is complicate, the utilization of the new tool is remarkable since it has minimized the amount of work to integrate new controllers in the future. Moreover, the result about switching the controller around with just a simple variable is very impressive since with just a simple modification with the MAVLink protocol, users can change the control algorithm instantly. This achievement will be crucial in case of testing and comparing new controllers with the others as it would save a lot of time.

References

There are no sources in the current document.