Arducopter deals with many kinds of multicopter and a helicopter. However here I only discuss the case of quadcopter, the other cases of multicopter are similar, except for the case of traditional helicopter.
The first and main library that we're going to use is the ArduPilot Hardware Abstraction Layer (HAL) library. This library tries to hide some of the low level details about how you read and write to pins and some other things - the advantage is that the software can then be ported to new hardware by only changing the hardware abstraction layer. In the case of ArduPilot, there are two hardware platforms, APM and PX4, each of which have their own HAL library which allows the ArduPilot code to remain the same across both. If you later decide to run your code on the Raspberry Pi, you'll only need to change the HAL.
The HAL library is made up from several components:

- RCInput - for reading the RC Radio.
- RCOutput - for controlling the motors and other outputs.
- Scheduler - for running particular tasks at regular time intervals.
- Console - essentially provides access to the serial port.
- I2C, SPI - bus drivers (small circuit board networks for connecting to sensors)
- GPIO - Generial Purpose Input/Output - allows raw access to the arduino pins, but in our case, mainly the LEDs

```
const AP_HAL::HAL& hal = AP_HAL_AVR_APM2;  // Hardware abstraction layer
```

==

init

```
init_ardupilot();
scheduler.init
```

==

task scheduler

In the loop, fast_loop and scheduler are running in parallel.

```
static const AP_Scheduler::Task scheduler_tasks[] PROGMEM = {
    { update_GPS,            2,     900 },
    { update_navigation,     10,    500 },
    { medium_loop,           2,     700 },
    { update_altitude,      10,    1000 },
    { fifty_hz_loop,         2,     950 },
    { run_nav_updates,      10,     800 },
    { slow_loop,            10,     500 },
```

```
    { gcs_check_input,       2,     700 },
    { gcs_send_heartbeat,  100,     700 },
    { gcs_data_stream_send,  2,    1500 },
    { gcs_send_deferred,     2,    1200 },
    { compass_accumulate,    2,     700 },
    { barometer_accumulate,  2,     900 },
    { super_slow_loop,     100,    1100 },
    { perf_update,        1000,     500 }
};
```

There is a number of tasks in scheduler_tasks, however let's concentrate on tasks that relate to flight control.

```
          TASKS                 RUN AT
     { update_GPS,              50 Hz    }
```

Updates GPS and inits home for ground start the first time GPS_OK_FIX_3D.

```
     {update_navigation       10 Hz     }
```

Calls update_nav_mode() at 10 Hz.
Based on nav_mode will call update_circle, update_loiter, or update_wpnav but all call loiter position controller (get_loiter_position_to_velocity, then get_loiter_velocity_to_acceleration in the end get_loiter_acceleration_to_lean_angles).
updates are called at 10 Hz, in the end will compute desired_roll, desired_pitch, so these parameters are updated at 10 Hz, before calling stabilize roll-pitch controllers.

```
     fifty_hz_loop            50 Hz
    // get altitude and climb rate from inertial lib
    read_inertial_altitude();
    // Update the throttle ouput
    update_throttle_mode();
```

update_throttle_mode() - in Attitude.pde if(apply_angle_boost) {g.rc_3.servo_out = get_angle_boost(throttle_out)}.

```
    { run_nav_updates,      10 Hz      }
```

run_nav_updates() is in scheduler_tasks and is defined in navigation.pde.
It is running at 10 Hz and does the followings:

```
    calc_position();     // fetch position from inertial navigation (current_loc.lng,
                         // current_loc.lat)
```

```
    calc_distance_and_bearing();   // calculate distance and bearing for reporting and
                                   // autopilot decisions
                                  // if navigation mode NAV_LOITER, NAV_CIRCLE then bearing to
                                   // the target else (NAV_WP) to the next WP

    run_autopilot();     // run autopilot to make high level decisions about control modes
      update_commands(); // in case of AUTO mode, load the next command if the command queues
                         // are empty, and execute_nav_command() or process_cond_command()
      verify_commands(); // check if navigation and conditional commands completed
```

execute_nav_command() in command_process.pde calls process_nav_command() in command_logic.pde which in case of MAV_CMD_NAV_WAYPOINT calls do_nav_wp() wich does the followings:

```
    // set roll-pitch mode
    set_roll_pitch_mode(AUTO_RP);
    // set throttle mode
    set_throttle_mode(AUTO_THR);
    // set nav mode
    set_nav_mode(NAV_WP);
    // Set next WP as navigation target (distances from home in cm from lon, lat)
    wp_nav.set_destination(pv_location_to_vector(command_nav_queue));
    // initialise original_wp_bearing which is used to check if we have missed the waypoint
    wp_bearing = wp_nav.get_bearing_to_destination();
    original_wp_bearing = wp_bearing;
```

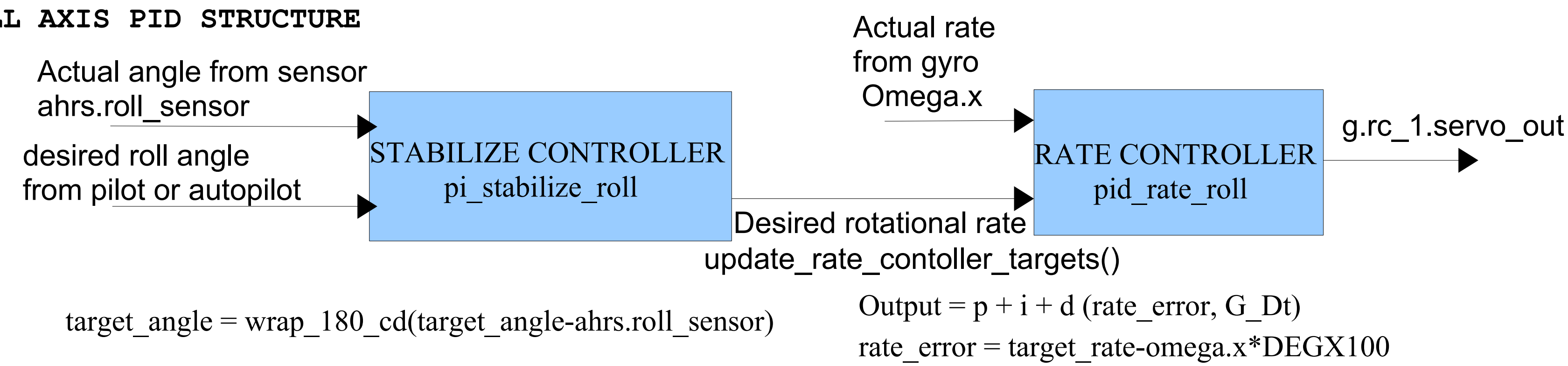
===

PIDs

```
Stabilize Roll      get_stabilize_roll
Stabilize Pitch     get_stabilize_pitch
Stabilize Yaw       get_stabilize_yaw
Rate Roll           get_rate_roll
Rate Pitch          get_rate_pitch
Rate Yaw            get_rate_yaw
Loiter PID          get_loiter_position_to_velocity
Rate Loiter         get_loiter_velocity_to_acceleration (Loiter does not require much tuning)
```

Throttle Accel	get_throttle_accel The Throttle Accel PID gains convert the acceleration error (i.e the difference between the desired acceleration and the actual acceleration) into a motor output. The 1:2 ratio of P to I (i.e. I is twice the size of P) should be maintained if you modify these parameters. These values should never be increased but for very powerful copters you may get better response by reducing both by 50% (i.e P to 0.5, I to 1.0).
Throttle Rate	get_throttle_rate The Throttle Rate (which normally requires no tuning) converts the desired climb or descent rate into a desired acceleration up or down.
Altitude Hold	get_throttle_althold The Altitude Hold P is used to convert the altitude error (the difference between the desired altitude and the actual altitude) to a desired climb or descent rate. A higher rate will make it more aggressively attempt to maintain it's altitude but if set too high leads to a jerky throttle response.

ROLL AXIS PID STRUCTURE

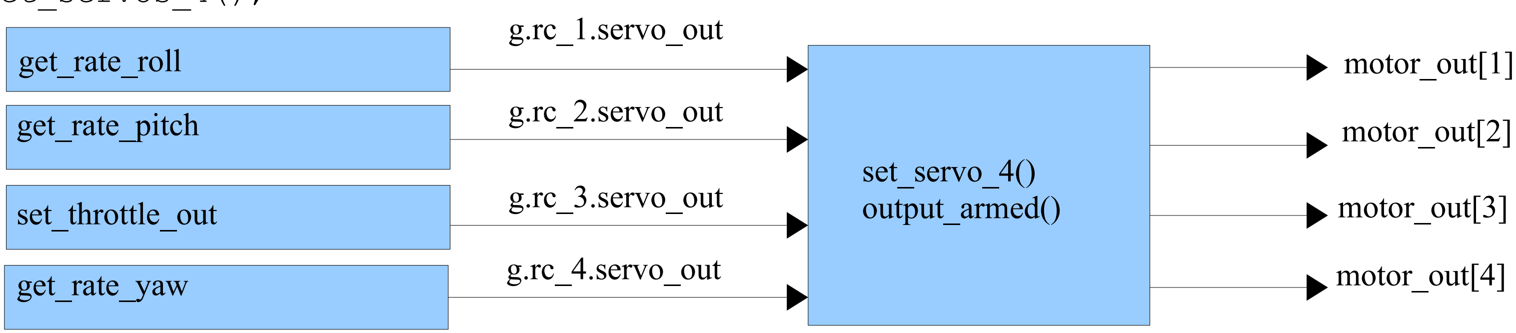


==

MAIN LOOP

```
fast_loop() running at 100 Hz in Arducopter.pde which does:
    // INPUT - IMU DCM Algorithm
    read_AHRS();    // reads IMU inputs that are necessary for running rate controllers.

    // reads all of the necessary trig functions for cameras, throttle, etc.
```

```
update_trig();  // calculates Euler angles' trig functions: cos_roll_x, cos_pitch_x,
                // cos_yaw, sin_yaw, sin_roll, sin_pitch from DCM.

// RATE CONTROLLERS - run low level rate controllers that only require IMU data
run_rate_controllers();   // runs rate controllers first, but helicopters only use rate
                          // controllers for yaw and only when not using an external gyro
                          // else (not HELI_FRAME) g.rc_1.servo_out=get_rate_roll(),
                          // g.rc_2.servo_out=get_rate_pitch(),
                          // g.rc_4.servo_out=get_rate_yaw() in attitude.pde
                          // g.rc_3.servo_out = get_angle_boost(throttle_out)


 // OUTPUT
 // write out the servo PWM values (servo_out values were calculated in rate controllers)

 set_servos_4();
```




```
 // Inertial Nav - updates velocities and positions using latest info from ahrs, ins and
 // barometer if new data is available
read_inertia();                // actually inertial_nav.update(G_Dt)

// INPUTS - Read radio and 3-position switch on radio
read_radio();
read_control_switch();
// set_mode() in system.pde - change flight mode and perform any necessary initialisation,
// returns true if mode was succesfully set. It'll set roll_pitch_mode to ROLL_PITCH_ACRO,
// ROLL_PITCH_STABLE, ROLL_PITCH_AUTO (actually set by first nav command)
```

```
// ROLL_PITCH_LOITER, ROLL_PITCH_SPORT ...

// STABILIZE CONTROLLERS include roll, pitch, yaw stabilize controllers that will compute
desired rotational rates for rate controllers
    update_yaw_mode();         // updates yaw mode, get_stabilize_yaw(nav_yaw);
    update_roll_pitch_mode(); // get_stabilize_roll, get_stabilize_pitch
                              // In case of ROLL_PITCH_AUTO
                              // nav_roll = wp_nav.get_desired_roll();
                              // nav_pitch = wp_nav.get_desired_pitch();
                              // get_stabilize_roll(nav_roll);
                              // get_stabilize_pitch(nav_pitch);
In update_roll_pitch_mode(), update_simple_mode() is called in manual modes (stabilize,
loiter,...) to apply SIMPLE mode transform.

    // update targets to rate controllers - converts earth frame rates to body frame rates for
    // rate controllers
   update_rate_contoller_targets();  // In attitude.pde
}
```