ArduPlane Version 2.65

This code description is based partly on APM code outline
https://docs.google.com/document/d/1OCEMtCq7Njr-YeEroSsMPQCZNP0QvMxqYPPCJy2zZHI/edit?hl=en&pli=1

Here only normal flight mode's functions with APM2.5 board are discussed. HIL modes and telemetry are out of the scope.

ArduPlane.pde

All the basic system and flight variables and structures are declared at the top of this file.
User configurations are in APM_config.h,
g defined as Parameter class entity (Parameter.h)

```
static Parameters      g;
   // PID controllers
   PID        pidNavRoll;
   PID        pidServoRoll;
   PID        pidServoPitch;
   PID        pidNavPitchAirspeed;
   PID        pidServoRudder;
   PID        pidTeThrottle;
   PID        pidNavPitchAltitude;
```

From parameters.h it's can be seen that plane's controllers are PID controllers including attitude angle controllers and servo controllers

```
// All GPS access should be through this pointer.
static GPS        *g_gps;
```

setup() Basic Initialization
Calls memcheck_init() in libraries/memcheck/memcheck.cpp and calls init_ardupilot() in system.pde to init serial ports, I2C, SPI, data flash, servos, RC and telemetry.

loop() Main system loop. Also know as the outside loop. This calls fast_loop and medium_loop.

fast_loop() This loop should be executed at 50Hz if possible to ensure servos update frequency.

medium_loop() - this loop is also called at 50Hz but with counter from 0 to 4 each branch is executed at 10 Hz, so the slow_loop is called at 10 Hz. With counter 0, 1, 2, each branch is executed at $3^1/_3$ Hz.

In the table below the main loop is shown delta_ms_fast_loop

--------Fast_loop --Medium_loop, counter=0 update_GPS()
read_radio()
ahrs.update(); calc_bearing_error()
update_current_flight_mode();
if (control_mode > MANUAL) stabilize();
set_servos();
--------Fast_loop --Medium_loop, counter=1
read_control_switch();
navigate() nav_bearing = target_bearing;
read_radio()
ahrs.update(); calc_bearing_error()
update_current_flight_mode();
if (control_mode > MANUAL) stabilize();
set_servos();
--------Fast_loop --Medium_loop, counter=2 update_alt(); update_command()
read_radio()
ahrs.update(); calc_bearing_error()
update_current_flight_mode();
if (control_mode > MANUAL) stabilize();
set_servos();
--------Fast_loop --Medium_loop, counter=3 Telemetry
read_radio()
ahrs.update(); calc_bearing_error()
update_current_flight_mode();
if (control_mode > MANUAL) stabilize();
set_servos();

dTnav

counter=0 check_long_failsafe();
--------Fast_loop --Medium_loop, counter=4 Slow_loop() counter=1 update_aux_servo_function();
counter=2 update_events();
read_radio()
ahrs.update(); calc_bearing_error()
update_current_flight_mode();
if (control_mode > MANUAL) stabilize();
set_servos();
--------Fast_loop --Medium_loop, counter=0 update_GPS()
read_radio()
ahrs.update(); calc_bearing_error()
update_current_flight_mode();
if (control_mode > MANUAL) stabilize();
set_servos();
--------Fast_loop --Medium_loop, counter=1

The control process with APM2.5 hardware can be imagined as follow:

ATMEGA 2560 radio input is PPM from ATMEGA 32-U2 with _timer5_capt_cb() for reading PPM signal, inputs/outputs are SERIAL0 to ATMEGA 32-U2 to USB (console) for loading software and running simulations, SERIAL1 (GPS data) for communicating with GPS, SERIAL2 or SERIAL0 according to SJ1/4 setting (Xbee telemetry) for telemetry, SPI_mega bus for communicating with MPU6000 motion processing unit and baro MS5011, SPI_DF for communicating with data flash chip, and I2C bus connects to magnetometer and I2C port, outputs are pwm outputs from timers to drive servos.

Outputs from RC receiver are connected to APM2.5 board inputs (max 8 channel, channel 8 for mode select). From there they are connected to ATMEGA 32-U2 . ATMEGA 32-U2 combines 8 PWM channels into 1 PPM output. Due to the fact that PPM channel frequency = servo update frequency = 50 Hz, so the PPM cycle is 20 ms and the maximum PWM pulse width is 1900us = 1.9 ms, so maximum 10 PWM channels can be combined into 1 PPM channel. This PPM channel from ATMEGA 32-U2 is used to interrupt ATMEGA 2560. Each time there's a PPM pulse, an interrupt is generated and APM_RC_APM2.cpp _timer5_capt_cb() interrupt service routine calculates time intervals between PPM pulses (the ppm pulse width) and saves in _PPM_RAW[].

MPU6000 and pressure sensor MS5011 are connected to SPI_mega bus, currently with Arduplane, ATMEGA 2560 only reads gyro and accelerator data from MPU6000 to calculate DCM based on the old DCM algorithm but in the future it may use DMP functions to calculate DCM for a beter performance. Besides, ATMEGA 2560 also reads magnetometer data on I2C interface and writes logs to data flash on SPI_DF bus.

Pulse outputs from 2560 are generated by programmable timers in micro controller as programmed in OutputCh(uint8_t ch, uint16_t pwm) in APM_RC_APM2.cpp and this is called by set_servos() in attitude.pde. Maximum 10 pulse outputs can be generated.

Fast _loop details
===
read_radio() // radio.pde

InputCh(ch) in APM_RC_APM2.cpp reads _PPM_RAW[ch]. read_radio() then reads all 8 channels and sets pwm arcordingly (radio_in=pwm).

```
g.channel_roll.set_pwm(APM_RC.InputCh(CH_ROLL));     // if not elevon mixing
g.channel_pitch.set_pwm(APM_RC.InputCh(CH_PITCH));

g.channel_throttle.set_pwm(APM_RC.InputCh(CH_3));
g.channel_rudder.set_pwm(APM_RC.InputCh(CH_4));
g.rc_5.set_pwm(APM_RC.InputCh(CH_5));
g.rc_6.set_pwm(APM_RC.InputCh(CH_6));
g.rc_7.set_pwm(APM_RC.InputCh(CH_7));
g.rc_8.set_pwm(APM_RC.InputCh(CH_8));
```

If there's a mode set on channel 8, it will be processed in medium_loopCounter=1 case
(read_control_switch();   // control_modes.pde).

==
//full DCM update
ahrs.update();   // libraries/AP_AHRS/AP_AHRS_DCM.cpp.

(Look at DCMDraft2.pdf for more information
http://code.google.com/p/gentlenav/downloads/detail?name=DCMDraft2.pdf&can=2&q
also look at   http://gentlenav.googlecode.com/files/RollPitchDriftCompensation.pdf
about driftcorrection)
AP_InertialSensor_MPU6000::update() calculates gyro and accelerator vectors used in matrix_update()

```
matrix_update(delta_t);   // Integrate the DCM matrix using gyro inputs
normalize();              // Normalize the DCM matrix
drift_correction();       // Perform drift correction
check_matrix();           // paranoid check for bad values in the DCM matrix
euler_angles();           // Calculate pitch, roll, yaw for stabilization and navigation
```

(This is the most important algorithm of the software that calculates DCM matrix. In the future DMP functions may replace it. I think most of us don't need to understand this in details, it's necessary only in case you wan to change the software basis)

==
// uses the yaw from the DCM to give more accurate turns
calc_bearing_error();   // in Navigation.pde.

navigate() (navigation.pde) - called in medium_loop as it needs GPS data, calculates
nav_bearing_cd = target_bearing_cd = get_bearing_cd(&current_loc, &next_WP)
(however for cases of LOITER, RTL, GUIDED nav_bearing_cd is changed in update_loiter() as called in update_navigation())

For calculating: bearing_error_cd = nav_bearing_cd – ahrs.yaw_sensor; nav_bearing_cd doesn't change between navigate() calls in medium_loop but ahrs.yaw_sensor yes in fast_loop, so the bearing_error_cd changes in fast_loop that are used in calc_nav_yaw and calc_nav_roll.

==
// custom code/exceptions for flight modes
// --
update_current_flight_mode();   // (Arduplane.pde)

Most cases (except FBW, STABILIZE, CIRCLE and MANUAL) call calc_nav_roll, calc_nav_pitch, calc_throttle (these are in attitude.pde) to calculate nav_roll_cd from bearing_error_cd, and also calculate throttle, nav_pitch_cd.

e.g. nav_roll_cd = g.pidNavRoll.get_pid(bearing_error_cd, dTnav, nav_gain_scaler).

In STABILIZE mode, update_current_flight_mode() sets nav_rol_cd = nav_pitch_cd = 0, so if you release stick that means read_radio() gives radio_in = radio_trim, so in stabilize() in attitude.pde
ch1_inf = ch2_inf = 1 (see ch1_inf, ch2_inf calculating in stabilize()),
g.channel_roll.pwm_to_angle() = g.channel_pitch.pwm_to_angle() = 0, and with APM_CONTROL disabled g.channel_roll.servo_out =
g.pidServoRoll.get_pid((nav_roll_cd - ahrs.roll_sensor),delta_ms_fast_loop, speed_scaler)
= g.pidServoRoll.get_pid((0 - ahrs.roll_sensor), delta_ms_fast_loop, speed_scaler)

(similiarly for g.channel_pitch.servo_out) so the plane will go back to nav_rol_cd = nav_pitch_cd = 0 level state.

In CIRCLE mode it sets nav_roll_cd = g.roll_limit_cd / 3; nav_pitch_cd = 0; So when calling stabilize() the error in get_pid() = g.roll_limit_cd / 3 - ahrs.roll_sensor is comming to 0 i.e. ahrs.roll_sensor is comming to g.roll_limit_cd / 3 = constant, the plane circles.

In FBW modes, nav_roll_cd = g.channel_roll.norm_input() * g.roll_limit_cd and depends only on stick angle. The nav_pitch_cd depends on A or B modes.

In MANUAL mode (SOFTWARE MANUAL), servo_out values are calculated directly from stick inputs and then sent to servos without calling stabilize().

```
g.channel_roll.servo_out = g.channel_roll.pwm_to_angle();
g.channel_pitch.servo_out = g.channel_pitch.pwm_to_angle();
g.channel_rudder.servo_out = g.channel_rudder.pwm_to_angle();
```

===

```
// apply desired roll, pitch and yaw to the plane
// ---------------------------------------------
if (control_mode > MANUAL) stabilize();          // in Attitude.pde.
```

In stabilize() if there's an input from sticks in other modes (not FBW), it is mixed with servo_out as ch1_inf, ch2_inf < 1 g.channel_roll.servo_out =
g.pidServoRoll.get_pid((nav_roll_cd - ahrs.roll_sensor),delta_ms_fast_loop, speed_scaler)

```
        tempcalc = nav_pitch_cd +
        fabs(ahrs.roll_sensor * g.kff_pitch_compensation) +
        (g.channel_throttle.servo_out * g.kff_throttle_to_pitch) -
        (ahrs.pitch_sensor – g.pitch_trim_cd);
 g.channel_pitch.servo_out = g.pidServoPitch.get_pid(tempcalc, speed_scaler);
          g.channel_roll.servo_out *= ch1_inf;
          g.channel_pitch.servo_out *= ch2_inf;
          g.channel_roll.servo_out +=     g.channel_roll.pwm_to_angle();
          g.channel_pitch.servo_out +=    g.channel_pitch.pwm_to_angle();
```

==

```
// write out the servo PWM values
// ------------------------------
set_servos();                          // in Attitude.pde
```

Calculates channel_roll.radio_out, channel_pitch.radio_out, channel_throttle.radio_out and channel_rudder.radio_out from channel_...servo_out and mix_mode then sends values to the PWM timers for output

```
---------------------------------------
        APM_RC_APM2.OutputCh(CH_1, g.channel_roll.radio_out); // send to Servos
        APM_RC_APM2.OutputCh(CH_2, g.channel_pitch.radio_out); // send to Servos
        APM_RC_APM2.OutputCh(CH_3, g.channel_throttle.radio_out); // send to Servos
        APM_RC_APM2.OutputCh(CH_4, g.channel_rudder.radio_out); // send to Servos
        // Route configurable aux. functions to their respective servos
        g.rc_5.output_ch(CH_5);
        g.rc_6.output_ch(CH_6);
        g.rc_7.output_ch(CH_7);
        g.rc_8.output_ch(CH_8);
```

Medium_loop details:

```
==============================================================
 update_GPS();
```

Updates GPS data and GPS light on board and clarifies ground start or air start at the beginning.

```
// Read 6-position switch on radio
// -------------------------------
read_control_switch();
```

Reads 6 control modes from channel 8 pulse width and sets flight control modes accordingly

```
// calculate the plane's desired bearing
// -------------------------------------
navigate();
```

In auto mode calculates navigation bearing to the next waypoint and then verifies if waypoint is reached or missed so that the program can search for the next command (nav or non nav). In loiter relating flight modes (LOITER, RTL, GUIDED) it updates navigation bearing based on the distance to the waypoint.

```
// Read altitude from sensors
// ------------------
update_alt();
```

Calculates current location altitude based on GPS data and barometer data.

```
// altitude smoothing
// ------------------
```

```
   if (control_mode != FLY_BY_WIRE_B)
      calc_altitude_error();
```
Calculates altitude error from the measured altitude and the target one.

```
// perform next command
// --------------------
   update_commands();
```
In auto mode it calls process_next_command(). The code of process_next_command() includes 2 parts, the first one searches for the next navigation command, the second one searches for the next non navigation command. In any case, if the current command is executed, then the next command is searched and if found then it is processed.

Log_Write_
Writes logs to flash on SPI_DF bus based on log bitmask.

slow_loop();