branch: master **ardupilot** / ArduCopter / **GCS_Mavlink.cpp**

tridge 6 days ago Copter: fixed pde paths

3 contributors

1834 lines (1590 sloc) 61.288 kB Raw Blame History

```
// -*- tab-width: 4; Mode: C++; c-basic-offset: 4; indent-tabs-mode: nil -*-

#include "Copter.h"

// default sensors are present and healthy: gyro, accelerometer, barometer, rate_control, attitude_stabilization, yaw_position, al
#define MAVLINK_SENSOR_PRESENT_DEFAULT (MAV_SYS_STATUS_SENSOR_3D_GYRO | MAV_SYS_STATUS_SENSOR_3D_ACCEL | MAV_SYS_STATUS_SENSOR_ABS

// check if a message will fit in the payload space available
#define HAVE_PAYLOAD_SPACE(chan, id) (comm_get_txspace(chan) >= MAVLINK_NUM_NON_PAYLOAD_BYTES+MAVLINK_MSG_ID_ ## id ## _LEN)
#define CHECK_PAYLOAD_SIZE(id) if (txspace < MAVLINK_NUM_NON_PAYLOAD_BYTES+MAVLINK_MSG_ID_ ## id ## _LEN) return false

void Copter::gcs_send_heartbeat(void)
{
    gcs_send_message(MSG_HEARTBEAT);
}

void Copter::gcs_send_deferred(void)
{
    gcs_send_message(MSG_RETRY_DEFERRED);
}

/*
 *  !!NOTE!!
 *
 *  the use of NOINLINE separate functions for each message type avoids
 *  a compiler bug in gcc that would cause it to use far more stack
 *  space than is needed. Without the NOINLINE we use the sum of the
 *  stack needed for each message type. Please be careful to follow the
 *  pattern below when adding any new messages
 */

NOINLINE void Copter::send_heartbeat(mavlink_channel_t chan)
{
    uint8_t base_mode = MAV_MODE_FLAG_CUSTOM_MODE_ENABLED;
    uint8_t system_status = ap.land_complete ? MAV_STATE_STANDBY : MAV_STATE_ACTIVE;
    uint32_t custom_mode = control_mode;

    // set system as critical if any failsafe have triggered
    if (failsafe.radio || failsafe.battery || failsafe.gcs || failsafe.ekf)  {
        system_status = MAV_STATE_CRITICAL;
    }

    // work out the base_mode. This value is not very useful
    // for APM, but we calculate it as best we can so a generic
    // MAVLink enabled ground station can work out something about
    // what the MAV is up to. The actual bit values are highly
    // ambiguous for most of the APM flight modes. In practice, you
    // only get useful information from the custom_mode, which maps to
    // the APM flight mode and has a well defined meaning in the
    // ArduPlane documentation
    base_mode = MAV_MODE_FLAG_STABILIZE_ENABLED;
    switch (control_mode) {
    case AUTO:
    case RTL:
    case LOITER:
    case GUIDED:
    case CIRCLE:
    case POSHOLD:
    case BRAKE:
        base_mode |= MAV_MODE_FLAG_GUIDED_ENABLED;
        // note that MAV_MODE_FLAG_AUTO_ENABLED does not match what
        // APM does in any mode, as that is defined as "system finds its own goal
        // positions", which APM does not currently do
```

```
64          break;
65      }
66
67      // all modes except INITIALISING have some form of manual
68      // override if stick mixing is enabled
69      base_mode |= MAV_MODE_FLAG_MANUAL_INPUT_ENABLED;
70
71  #if HIL_MODE != HIL_MODE_DISABLED
72      base_mode |= MAV_MODE_FLAG_HIL_ENABLED;
73  #endif
74
75      // we are armed if we are not initialising
76      if (motors.armed()) {
77          base_mode |= MAV_MODE_FLAG_SAFETY_ARMED;
78      }
79
80      // indicate we have set a custom mode
81      base_mode |= MAV_MODE_FLAG_CUSTOM_MODE_ENABLED;
82
83      mavlink_msg_heartbeat_send(
84          chan,
85  #if (FRAME_CONFIG == QUAD_FRAME)
86          MAV_TYPE_QUADROTOR,
87  #elif (FRAME_CONFIG == TRI_FRAME)
88          MAV_TYPE_TRICOPTER,
89  #elif (FRAME_CONFIG == HEXA_FRAME || FRAME_CONFIG == Y6_FRAME)
90          MAV_TYPE_HEXAROTOR,
91  #elif (FRAME_CONFIG == OCTA_FRAME || FRAME_CONFIG == OCTA_QUAD_FRAME)
92          MAV_TYPE_OCTOROTOR,
93  #elif (FRAME_CONFIG == HELI_FRAME)
94          MAV_TYPE_HELICOPTER,
95  #elif (FRAME_CONFIG == SINGLE_FRAME)  //because mavlink did not define a singlecopter, we use a rocket
96          MAV_TYPE_ROCKET,
97  #elif (FRAME_CONFIG == COAX_FRAME)  //because mavlink did not define a singlecopter, we use a rocket
98          MAV_TYPE_ROCKET,
99  #else
100   #error Unrecognised frame type
101 #endif
102         MAV_AUTOPILOT_ARDUPILOTMEGA,
103         base_mode,
104         custom_mode,
105         system_status);
106 }
107
108 NOINLINE void Copter::send_attitude(mavlink_channel_t chan)
109 {
110     const Vector3f &gyro = ins.get_gyro();
111     mavlink_msg_attitude_send(
112         chan,
113         millis(),
114         ahrs.roll,
115         ahrs.pitch,
116         ahrs.yaw,
117         gyro.x,
118         gyro.y,
119         gyro.z);
120 }
121
122 #if AC_FENCE == ENABLED
123 NOINLINE void Copter::send_limits_status(mavlink_channel_t chan)
124 {
125     fence_send_mavlink_status(chan);
126 }
127 #endif
128
129
130 NOINLINE void Copter::send_extended_status1(mavlink_channel_t chan)
131 {
132     uint32_t control_sensors_present;
133     uint32_t control_sensors_enabled;
134     uint32_t control_sensors_health;
135
136     // default sensors present
137     control_sensors_present = MAVLINK_SENSOR_PRESENT_DEFAULT;
138
139     // first what sensors/controllers we have
140     if (g.compass_enabled) {
141         control_sensors_present |= MAV_SYS_STATUS_SENSOR_3D_MAG; // compass present
142     }
143     if (gps.status() > AP_GPS::NO_GPS) {
```

```
        control_sensors_present |= MAV_SYS_STATUS_SENSOR_GPS;
    }
#if OPTFLOW == ENABLED
    if (optflow.enabled()) {
        control_sensors_present |= MAV_SYS_STATUS_SENSOR_OPTICAL_FLOW;
    }
#endif
    if (ap.rc_receiver_present) {
        control_sensors_present |= MAV_SYS_STATUS_SENSOR_RC_RECEIVER;
    }

    // all present sensors enabled by default except altitude and position control and motors which we will set individually
    control_sensors_enabled = control_sensors_present & (~MAV_SYS_STATUS_SENSOR_Z_ALTITUDE_CONTROL &
                                                         ~MAV_SYS_STATUS_SENSOR_XY_POSITION_CONTROL &
                                                         ~MAV_SYS_STATUS_SENSOR_MOTOR_OUTPUTS);

    switch (control_mode) {
    case ALT_HOLD:
    case AUTO:
    case GUIDED:
    case LOITER:
    case RTL:
    case CIRCLE:
    case LAND:
    case OF_LOITER:
    case POSHOLD:
    case BRAKE:
        control_sensors_enabled |= MAV_SYS_STATUS_SENSOR_Z_ALTITUDE_CONTROL;
        control_sensors_enabled |= MAV_SYS_STATUS_SENSOR_XY_POSITION_CONTROL;
        break;
    case SPORT:
        control_sensors_enabled |= MAV_SYS_STATUS_SENSOR_Z_ALTITUDE_CONTROL;
        break;
    }

    // set motors outputs as enabled if safety switch is not disarmed (i.e. either NONE or ARMED)
    if (hal.util->safety_switch_state() != AP_HAL::Util::SAFETY_DISARMED) {
        control_sensors_enabled |= MAV_SYS_STATUS_SENSOR_MOTOR_OUTPUTS;
    }

    // default to all healthy except baro, compass, gps and receiver which we set individually
    control_sensors_health = control_sensors_present & ~(MAV_SYS_STATUS_SENSOR_ABSOLUTE_PRESSURE |
                                                         MAV_SYS_STATUS_SENSOR_3D_MAG |
                                                         MAV_SYS_STATUS_SENSOR_GPS |
                                                         MAV_SYS_STATUS_SENSOR_RC_RECEIVER);
    if (barometer.all_healthy()) {
        control_sensors_health |= MAV_SYS_STATUS_SENSOR_ABSOLUTE_PRESSURE;
    }
    if (g.compass_enabled && compass.healthy() && ahrs.use_compass()) {
        control_sensors_health |= MAV_SYS_STATUS_SENSOR_3D_MAG;
    }
    if (gps.status() > AP_GPS::NO_GPS) {
        control_sensors_health |= MAV_SYS_STATUS_SENSOR_GPS;
    }
#if OPTFLOW == ENABLED
    if (optflow.healthy()) {
        control_sensors_health |= MAV_SYS_STATUS_SENSOR_OPTICAL_FLOW;
    }
#endif
    if (ap.rc_receiver_present && !failsafe.radio) {
        control_sensors_health |= MAV_SYS_STATUS_SENSOR_RC_RECEIVER;
    }
    if (!ins.get_gyro_health_all() || !ins.gyro_calibrated_ok_all()) {
        control_sensors_health &= ~MAV_SYS_STATUS_SENSOR_3D_GYRO;
    }
    if (!ins.get_accel_health_all()) {
        control_sensors_health &= ~MAV_SYS_STATUS_SENSOR_3D_ACCEL;
    }

    if (ahrs.initialised() && !ahrs.healthy()) {
        // AHRS subsystem is unhealthy
        control_sensors_health &= ~MAV_SYS_STATUS_AHRS;
    }

    int16_t battery_current = -1;
    int8_t battery_remaining = -1;

    if (battery.has_current() && battery.healthy()) {
        battery_remaining = battery.capacity_remaining_pct();

        battery_current = battery.current_amps() * 100;
```

```
    }

#if AP_TERRAIN_AVAILABLE
    switch (terrain.status()) {
    case AP_Terrain::TerrainStatusDisabled:
        break;
    case AP_Terrain::TerrainStatusUnhealthy:
        // To-Do: restore unhealthy terrain status reporting once terrain is used in copter
        //control_sensors_present |= MAV_SYS_STATUS_TERRAIN;
        //control_sensors_enabled |= MAV_SYS_STATUS_TERRAIN;
        //break;
    case AP_Terrain::TerrainStatusOK:
        control_sensors_present |= MAV_SYS_STATUS_TERRAIN;
        control_sensors_enabled |= MAV_SYS_STATUS_TERRAIN;
        control_sensors_health  |= MAV_SYS_STATUS_TERRAIN;
        break;
    }
#endif

#if CONFIG_SONAR == ENABLED
    if (sonar.num_sensors() > 0) {
        control_sensors_present |= MAV_SYS_STATUS_SENSOR_LASER_POSITION;
        control_sensors_enabled |= MAV_SYS_STATUS_SENSOR_LASER_POSITION;
        if (sonar.has_data()) {
            control_sensors_health |= MAV_SYS_STATUS_SENSOR_LASER_POSITION;
        }
    }
#endif

    if (!ap.initialised || ins.calibrating()) {
        // while initialising the gyros and accels are not enabled
        control_sensors_enabled &= ~(MAV_SYS_STATUS_SENSOR_3D_GYRO | MAV_SYS_STATUS_SENSOR_3D_ACCEL);
        control_sensors_health &= ~(MAV_SYS_STATUS_SENSOR_3D_GYRO | MAV_SYS_STATUS_SENSOR_3D_ACCEL);
    }

    mavlink_msg_sys_status_send(
        chan,
        control_sensors_present,
        control_sensors_enabled,
        control_sensors_health,
        (uint16_t)(scheduler.load_average(MAIN_LOOP_MICROS) * 1000),
        battery.voltage() * 1000, // mV
        battery_current,        // in 10mA units
        battery_remaining,      // in %
        0, // comm drops %,
        0, // comm drops in pkts,
        0, 0, 0, 0);

}

void NOINLINE Copter::send_location(mavlink_channel_t chan)
{
    uint32_t fix_time;
    // if we have a GPS fix, take the time as the last fix time. That
    // allows us to correctly calculate velocities and extrapolate
    // positions.
    // If we don't have a GPS fix then we are dead reckoning, and will
    // use the current boot time as the fix time.
    if (gps.status() >= AP_GPS::GPS_OK_FIX_2D) {
        fix_time = gps.last_fix_time_ms();
    } else {
        fix_time = millis();
    }
    const Vector3f &vel = inertial_nav.get_velocity();
    mavlink_msg_global_position_int_send(
        chan,
        fix_time,
        current_loc.lat,                // in 1E7 degrees
        current_loc.lng,                // in 1E7 degrees
        (ahrs.get_home().alt + current_loc.alt) * 10UL,      // millimeters above sea level
        current_loc.alt * 10,           // millimeters above ground
        vel.x,                          // X speed cm/s (+ve North)
        vel.y,                          // Y speed cm/s (+ve East)
        vel.z,                          // Z speed cm/s (+ve up)
        ahrs.yaw_sensor);               // compass heading in 1/100 degree
}

void NOINLINE Copter::send_nav_controller_output(mavlink_channel_t chan)

{
    const Vector3f &targets = attitude_control.angle_ef_targets();
```

```
    mavlink_msg_nav_controller_output_send(
        chan,
        targets.x / 1.0e2f,
        targets.y / 1.0e2f,
        targets.z / 1.0e2f,
        wp_bearing / 1.0e2f,
        wp_distance / 1.0e2f,
        pos_control.get_alt_error() / 1.0e2f,
        0,
        0);
}

// report simulator state
void NOINLINE Copter::send_simstate(mavlink_channel_t chan)
{
#if CONFIG_HAL_BOARD == HAL_BOARD_SITL
    sitl.simstate_send(chan);
#endif
}

void NOINLINE Copter::send_hwstatus(mavlink_channel_t chan)
{
    mavlink_msg_hwstatus_send(
        chan,
        hal.analogin->board_voltage()*1000,
        hal.i2c->lockup_count());
}

void NOINLINE Copter::send_servo_out(mavlink_channel_t chan)
{
#if HIL_MODE != HIL_MODE_DISABLED
    // normalized values scaled to -10000 to 10000
    // This is used for HIL.  Do not change without discussing with HIL maintainers

#if FRAME_CONFIG == HELI_FRAME
    mavlink_msg_rc_channels_scaled_send(
        chan,
        millis(),
        0, // port 0
        g.rc_1.servo_out,
        g.rc_2.servo_out,
        g.rc_3.radio_out,
        g.rc_4.servo_out,
        0,
        0,
        0,
        0,
        receiver_rssi);
#else
    mavlink_msg_rc_channels_scaled_send(
        chan,
        millis(),
        0,         // port 0
        g.rc_1.servo_out,
        g.rc_2.servo_out,
        g.rc_3.radio_out,
        g.rc_4.servo_out,
        10000 * g.rc_1.norm_output(),
        10000 * g.rc_2.norm_output(),
        10000 * g.rc_3.norm_output(),
        10000 * g.rc_4.norm_output(),
        receiver_rssi);
#endif
#endif // HIL_MODE
}

void NOINLINE Copter::send_radio_out(mavlink_channel_t chan)
{
    mavlink_msg_servo_output_raw_send(
        chan,
        micros(),
        0,     // port
        hal.rcout->read(0),
        hal.rcout->read(1),
        hal.rcout->read(2),
        hal.rcout->read(3),
        hal.rcout->read(4),
        hal.rcout->read(5),
        hal.rcout->read(6),
        hal.rcout->read(7));
```

```
384 }
385
386 void NOINLINE Copter::send_vfr_hud(mavlink_channel_t chan)
387 {
388     mavlink_msg_vfr_hud_send(
389         chan,
390         gps.ground_speed(),
391         gps.ground_speed(),
392         (ahrs.yaw_sensor / 100) % 360,
393         (int16_t)(motors.get_throttle())/10,
394         current_loc.alt / 100.0f,
395         climb_rate / 100.0f);
396 }
397
398 void NOINLINE Copter::send_current_waypoint(mavlink_channel_t chan)
399 {
400     mavlink_msg_mission_current_send(chan, mission.get_current_nav_index());
401 }
402
403 #if CONFIG_SONAR == ENABLED
404 void NOINLINE Copter::send_rangefinder(mavlink_channel_t chan)
405 {
406     // exit immediately if sonar is disabled
407     if (!sonar.has_data()) {
408         return;
409     }
410     mavlink_msg_rangefinder_send(
411             chan,
412             sonar.distance_cm() * 0.01f,
413             sonar.voltage_mv() * 0.001f);
414 }
415 #endif
416
417
418 /*
419   send PID tuning message
420  */
421 void Copter::send_pid_tuning(mavlink_channel_t chan)
422 {
423     const Vector3f &gyro = ahrs.get_gyro();
424     if (g.gcs_pid_mask & 1) {
425         const DataFlash_Class::PID_Info &pid_info = g.pid_rate_roll.get_pid_info();
426         mavlink_msg_pid_tuning_send(chan, PID_TUNING_ROLL,
427                                     pid_info.desired*0.01f,
428                                     degrees(gyro.x),
429                                     pid_info.FF*0.01,
430                                     pid_info.P*0.01,
431                                     pid_info.I*0.01,
432                                     pid_info.D*0.01);
433         if (!HAVE_PAYLOAD_SPACE(chan, PID_TUNING)) {
434             return;
435         }
436     }
437     if (g.gcs_pid_mask & 2) {
438         const DataFlash_Class::PID_Info &pid_info = g.pid_rate_pitch.get_pid_info();
439         mavlink_msg_pid_tuning_send(chan, PID_TUNING_PITCH,
440                                     pid_info.desired*0.01f,
441                                     degrees(gyro.y),
442                                     pid_info.FF*0.01f,
443                                     pid_info.P*0.01f,
444                                     pid_info.I*0.01f,
445                                     pid_info.D*0.01f);
446         if (!HAVE_PAYLOAD_SPACE(chan, PID_TUNING)) {
447             return;
448         }
449     }
450     if (g.gcs_pid_mask & 4) {
451         const DataFlash_Class::PID_Info &pid_info = g.pid_rate_yaw.get_pid_info();
452         mavlink_msg_pid_tuning_send(chan, PID_TUNING_YAW,
453                                     pid_info.desired*0.01f,
454                                     degrees(gyro.z),
455                                     pid_info.FF*0.01f,
456                                     pid_info.P*0.01f,
457                                     pid_info.I*0.01f,
458                                     pid_info.D*0.01f);
459         if (!HAVE_PAYLOAD_SPACE(chan, PID_TUNING)) {
460             return;
461         }
462     }
463     if (g.gcs_pid_mask & 8) {
```

```
464         const DataFlash_Class::PID_Info &pid_info = g.pid_accel_z.get_pid_info();
465         mavlink_msg_pid_tuning_send(chan, PID_TUNING_ACCZ,
466                                     pid_info.desired*0.01f,
467                                     -(ahrs.get_accel_ef_blended().z + GRAVITY_MSS),
468                                     pid_info.FF*0.01f,
469                                     pid_info.P*0.01f,
470                                     pid_info.I*0.01f,
471                                     pid_info.D*0.01f);
472         if (!HAVE_PAYLOAD_SPACE(chan, PID_TUNING)) {
473             return;
474         }
475     }
476 }
477
478
479 void NOINLINE Copter::send_statustext(mavlink_channel_t chan)
480 {
481     mavlink_statustext_t *s = &gcs[chan-MAVLINK_COMM_0].pending_status;
482     mavlink_msg_statustext_send(
483         chan,
484         s->severity,
485         s->text);
486 }
487
488 // are we still delaying telemetry to try to avoid Xbee bricking?
489 bool Copter::telemetry_delayed(mavlink_channel_t chan)
490 {
491     uint32_t tnow = millis() >> 10;
492     if (tnow > (uint32_t)g.telem_delay) {
493         return false;
494     }
495     if (chan == MAVLINK_COMM_0 && hal.gpio->usb_connected()) {
496         // this is USB telemetry, so won't be an Xbee
497         return false;
498     }
499     // we're either on the 2nd UART, or no USB cable is connected
500     // we need to delay telemetry by the TELEM_DELAY time
501     return true;
502 }
503
504
505 // try to send a message, return false if it won't fit in the serial tx buffer
506 bool GCS_MAVLINK::try_send_message(enum ap_message id)
507 {
508     uint16_t txspace = comm_get_txspace(chan);
509
510     if (copter.telemetry_delayed(chan)) {
511         return false;
512     }
513
514 #if HIL_MODE != HIL_MODE_SENSORS
515     // if we don't have at least 250 micros remaining before the main loop
516     // wants to fire then don't send a mavlink message. We want to
517     // prioritise the main flight control loop over communications
518     if (copter.scheduler.time_available_usec() < 250 && copter.motors.armed()) {
519         copter.gcs_out_of_time = true;
520         return false;
521     }
522 #endif
523
524     switch(id) {
525     case MSG_HEARTBEAT:
526         CHECK_PAYLOAD_SIZE(HEARTBEAT);
527         copter.gcs[chan-MAVLINK_COMM_0].last_heartbeat_time = hal.scheduler->millis();
528         copter.send_heartbeat(chan);
529         break;
530
531     case MSG_EXTENDED_STATUS1:
532         // send extended status only once vehicle has been initialised
533         // to avoid unnecessary errors being reported to user
534         if (copter.ap.initialised) {
535             CHECK_PAYLOAD_SIZE(SYS_STATUS);
536             copter.send_extended_status1(chan);
537             CHECK_PAYLOAD_SIZE(POWER_STATUS);
538             copter.gcs[chan-MAVLINK_COMM_0].send_power_status();
539         }
540         break;
541
542     case MSG_EXTENDED_STATUS2:
543         CHECK_PAYLOAD_SIZE(MEMINFO);
```

```
544         copter.gcs[chan-MAVLINK_COMM_0].send_meminfo();
545         break;
546
547     case MSG_ATTITUDE:
548         CHECK_PAYLOAD_SIZE(ATTITUDE);
549         copter.send_attitude(chan);
550         break;
551
552     case MSG_LOCATION:
553         CHECK_PAYLOAD_SIZE(GLOBAL_POSITION_INT);
554         copter.send_location(chan);
555         break;
556
557     case MSG_LOCAL_POSITION:
558         CHECK_PAYLOAD_SIZE(LOCAL_POSITION_NED);
559         send_local_position(copter.ahrs);
560         break;
561
562     case MSG_NAV_CONTROLLER_OUTPUT:
563         CHECK_PAYLOAD_SIZE(NAV_CONTROLLER_OUTPUT);
564         copter.send_nav_controller_output(chan);
565         break;
566
567     case MSG_GPS_RAW:
568         return copter.gcs[chan-MAVLINK_COMM_0].send_gps_raw(copter.gps);
569
570     case MSG_SYSTEM_TIME:
571         CHECK_PAYLOAD_SIZE(SYSTEM_TIME);
572         copter.gcs[chan-MAVLINK_COMM_0].send_system_time(copter.gps);
573         break;
574
575     case MSG_SERVO_OUT:
576         CHECK_PAYLOAD_SIZE(RC_CHANNELS_SCALED);
577         copter.send_servo_out(chan);
578         break;
579
580     case MSG_RADIO_IN:
581         CHECK_PAYLOAD_SIZE(RC_CHANNELS_RAW);
582         copter.gcs[chan-MAVLINK_COMM_0].send_radio_in(copter.receiver_rssi);
583         break;
584
585     case MSG_RADIO_OUT:
586         CHECK_PAYLOAD_SIZE(SERVO_OUTPUT_RAW);
587         copter.send_radio_out(chan);
588         break;
589
590     case MSG_VFR_HUD:
591         CHECK_PAYLOAD_SIZE(VFR_HUD);
592         copter.send_vfr_hud(chan);
593         break;
594
595     case MSG_RAW_IMU1:
596         CHECK_PAYLOAD_SIZE(RAW_IMU);
597         copter.gcs[chan-MAVLINK_COMM_0].send_raw_imu(copter.ins, copter.compass);
598         break;
599
600     case MSG_RAW_IMU2:
601         CHECK_PAYLOAD_SIZE(SCALED_PRESSURE);
602         copter.gcs[chan-MAVLINK_COMM_0].send_scaled_pressure(copter.barometer);
603         break;
604
605     case MSG_RAW_IMU3:
606         CHECK_PAYLOAD_SIZE(SENSOR_OFFSETS);
607         copter.gcs[chan-MAVLINK_COMM_0].send_sensor_offsets(copter.ins, copter.compass, copter.barometer);
608         break;
609
610     case MSG_CURRENT_WAYPOINT:
611         CHECK_PAYLOAD_SIZE(MISSION_CURRENT);
612         copter.send_current_waypoint(chan);
613         break;
614
615     case MSG_NEXT_PARAM:
616         CHECK_PAYLOAD_SIZE(PARAM_VALUE);
617         copter.gcs[chan-MAVLINK_COMM_0].queued_param_send();
618         break;
619
620     case MSG_NEXT_WAYPOINT:
621         CHECK_PAYLOAD_SIZE(MISSION_REQUEST);
622         copter.gcs[chan-MAVLINK_COMM_0].queued_waypoint_send();
623         break;
```

```
624
625     case MSG_RANGEFINDER:
626 #if CONFIG_SONAR == ENABLED
627         CHECK_PAYLOAD_SIZE(RANGEFINDER);
628         copter.send_rangefinder(chan);
629 #endif
630         break;
631
632     case MSG_TERRAIN:
633 #if AP_TERRAIN_AVAILABLE
634         CHECK_PAYLOAD_SIZE(TERRAIN_REQUEST);
635         copter.terrain.send_request(chan);
636 #endif
637         break;
638
639     case MSG_CAMERA_FEEDBACK:
640 #if CAMERA == ENABLED
641         CHECK_PAYLOAD_SIZE(CAMERA_FEEDBACK);
642         copter.camera.send_feedback(chan, copter.gps, copter.ahrs, copter.current_loc);
643 #endif
644         break;
645
646     case MSG_STATUSTEXT:
647         CHECK_PAYLOAD_SIZE(STATUSTEXT);
648         copter.send_statustext(chan);
649         break;
650
651     case MSG_LIMITS_STATUS:
652 #if AC_FENCE == ENABLED
653         CHECK_PAYLOAD_SIZE(LIMITS_STATUS);
654         copter.send_limits_status(chan);
655 #endif
656         break;
657
658     case MSG_AHRS:
659         CHECK_PAYLOAD_SIZE(AHRS);
660         copter.gcs[chan-MAVLINK_COMM_0].send_ahrs(copter.ahrs);
661         break;
662
663     case MSG_SIMSTATE:
664 #if CONFIG_HAL_BOARD == HAL_BOARD_SITL
665         CHECK_PAYLOAD_SIZE(SIMSTATE);
666         copter.send_simstate(chan);
667 #endif
668         CHECK_PAYLOAD_SIZE(AHRS2);
669         copter.gcs[chan-MAVLINK_COMM_0].send_ahrs2(copter.ahrs);
670         break;
671
672     case MSG_HWSTATUS:
673         CHECK_PAYLOAD_SIZE(HWSTATUS);
674         copter.send_hwstatus(chan);
675         break;
676
677     case MSG_MOUNT_STATUS:
678 #if MOUNT == ENABLED
679         CHECK_PAYLOAD_SIZE(MOUNT_STATUS);
680         copter.camera_mount.status_msg(chan);
681 #endif // MOUNT == ENABLED
682         break;
683
684     case MSG_BATTERY2:
685         CHECK_PAYLOAD_SIZE(BATTERY2);
686         copter.gcs[chan-MAVLINK_COMM_0].send_battery2(copter.battery);
687         break;
688
689     case MSG_OPTICAL_FLOW:
690 #if OPTFLOW == ENABLED
691         CHECK_PAYLOAD_SIZE(OPTICAL_FLOW);
692         copter.gcs[chan-MAVLINK_COMM_0].send_opticalflow(copter.ahrs, copter.optflow);
693 #endif
694         break;
695
696     case MSG_GIMBAL_REPORT:
697 #if MOUNT == ENABLED
698         CHECK_PAYLOAD_SIZE(GIMBAL_REPORT);
699         copter.camera_mount.send_gimbal_report(chan);
700 #endif
701         break;
702
703     case MSG_EKF_STATUS_REPORT:
```

```
        CHECK_PAYLOAD_SIZE(EKF_STATUS_REPORT);
        copter.ahrs.get_NavEKF().send_status_report(chan);
        break;

    case MSG_FENCE_STATUS:
    case MSG_WIND:
        // unused
        break;

    case MSG_PID_TUNING:
        CHECK_PAYLOAD_SIZE(PID_TUNING);
        copter.send_pid_tuning(chan);
        break;

    case MSG_VIBRATION:
        CHECK_PAYLOAD_SIZE(VIBRATION);
        send_vibration(copter.ins);
        break;

    case MSG_RETRY_DEFERRED:
        break; // just here to prevent a warning
    }

    return true;
}


const AP_Param::GroupInfo GCS_MAVLINK::var_info[] PROGMEM = {
    // @Param: RAW_SENS
    // @DisplayName: Raw sensor stream rate
    // @Description: Stream rate of RAW_IMU, SCALED_IMU2, SCALED_PRESSURE, and SENSOR_OFFSETS to ground station
    // @Units: Hz
    // @Range: 0 10
    // @Increment: 1
    // @User: Advanced
    AP_GROUPINFO("RAW_SENS", 0, GCS_MAVLINK, streamRates[0],  0),

    // @Param: EXT_STAT
    // @DisplayName: Extended status stream rate to ground station
    // @Description: Stream rate of SYS_STATUS, MEMINFO, MISSION_CURRENT, GPS_RAW_INT, NAV_CONTROLLER_OUTPUT, and LIMITS_STATUS to
    // @Units: Hz
    // @Range: 0 10
    // @Increment: 1
    // @User: Advanced
    AP_GROUPINFO("EXT_STAT", 1, GCS_MAVLINK, streamRates[1],  0),

    // @Param: RC_CHAN
    // @DisplayName: RC Channel stream rate to ground station
    // @Description: Stream rate of SERVO_OUTPUT_RAW and RC_CHANNELS_RAW to ground station
    // @Units: Hz
    // @Range: 0 10
    // @Increment: 1
    // @User: Advanced
    AP_GROUPINFO("RC_CHAN",  2, GCS_MAVLINK, streamRates[2],  0),

    // @Param: RAW_CTRL
    // @DisplayName: Raw Control stream rate to ground station
    // @Description: Stream rate of RC_CHANNELS_SCALED (HIL only) to ground station
    // @Units: Hz
    // @Range: 0 10
    // @Increment: 1
    // @User: Advanced
    AP_GROUPINFO("RAW_CTRL", 3, GCS_MAVLINK, streamRates[3],  0),

    // @Param: POSITION
    // @DisplayName: Position stream rate to ground station
    // @Description: Stream rate of GLOBAL_POSITION_INT to ground station
    // @Units: Hz
    // @Range: 0 10
    // @Increment: 1
    // @User: Advanced
    AP_GROUPINFO("POSITION", 4, GCS_MAVLINK, streamRates[4],  0),

    // @Param: EXTRA1
    // @DisplayName: Extra data type 1 stream rate to ground station
    // @Description: Stream rate of ATTITUDE and SIMSTATE (SITL only) to ground station
    // @Units: Hz
    // @Range: 0 10
    // @Increment: 1
    // @User: Advanced
    AP_GROUPINFO("EXTRA1",   5, GCS_MAVLINK, streamRates[5],  0),
```

```
784
785
786     // @Param: EXTRA2
787     // @DisplayName: Extra data type 2 stream rate to ground station
788     // @Description: Stream rate of VFR_HUD to ground station
789     // @Units: Hz
790     // @Range: 0 10
791     // @Increment: 1
792     // @User: Advanced
793     AP_GROUPINFO("EXTRA2",   6, GCS_MAVLINK, streamRates[6],  0),
794
795     // @Param: EXTRA3
796     // @DisplayName: Extra data type 3 stream rate to ground station
797     // @Description: Stream rate of AHRS, HWSTATUS, and SYSTEM_TIME to ground station
798     // @Units: Hz
799     // @Range: 0 10
800     // @Increment: 1
801     // @User: Advanced
802     AP_GROUPINFO("EXTRA3",   7, GCS_MAVLINK, streamRates[7],  0),
803
804     // @Param: PARAMS
805     // @DisplayName: Parameter stream rate to ground station
806     // @Description: Stream rate of PARAM_VALUE to ground station
807     // @Units: Hz
808     // @Range: 0 10
809     // @Increment: 1
810     // @User: Advanced
811     AP_GROUPINFO("PARAMS",   8, GCS_MAVLINK, streamRates[8],  0),
812     AP_GROUPEND
813 };
814
815
816 // see if we should send a stream now. Called at 50Hz
817 bool GCS_MAVLINK::stream_trigger(enum streams stream_num)
818 {
819     if (stream_num >= NUM_STREAMS) {
820         return false;
821     }
822     float rate = (uint8_t)streamRates[stream_num].get();
823
824     // send at a much lower rate while handling waypoints and
825     // parameter sends
826     if ((stream_num != STREAM_PARAMS) &&
827         (waypoint_receiving || _queued_parameter != NULL)) {
828         rate *= 0.25f;
829     }
830
831     if (rate <= 0) {
832         return false;
833     }
834
835     if (stream_ticks[stream_num] == 0) {
836         // we're triggering now, setup the next trigger point
837         if (rate > 50) {
838             rate = 50;
839         }
840         stream_ticks[stream_num] = (50 / rate) - 1 + stream_slowdown;
841         return true;
842     }
843
844     // count down at 50Hz
845     stream_ticks[stream_num]--;
846     return false;
847 }
848
849 void
850 GCS_MAVLINK::data_stream_send(void)
851 {
852     if (waypoint_receiving) {
853         // don't interfere with mission transfer
854         return;
855     }
856
857     if (!copter.in_mavlink_delay && !copter.motors.armed()) {
858         handle_log_send(copter.DataFlash);
859     }
860
861     copter.gcs_out_of_time = false;
862
863     if (_queued_parameter != NULL) {
```

```
        if (streamRates[STREAM_PARAMS].get() <= 0) {
            streamRates[STREAM_PARAMS].set(10);
        }
        if (stream_trigger(STREAM_PARAMS)) {
            send_message(MSG_NEXT_PARAM);
        }
        // don't send anything else at the same time as parameters
        return;
    }

    if (copter.gcs_out_of_time) return;

    if (copter.in_mavlink_delay) {
        // don't send any other stream types while in the delay callback
        return;
    }

    if (stream_trigger(STREAM_RAW_SENSORS)) {
        send_message(MSG_RAW_IMU1);
        send_message(MSG_RAW_IMU2);
        send_message(MSG_RAW_IMU3);
    }

    if (copter.gcs_out_of_time) return;

    if (stream_trigger(STREAM_EXTENDED_STATUS)) {
        send_message(MSG_EXTENDED_STATUS1);
        send_message(MSG_EXTENDED_STATUS2);
        send_message(MSG_CURRENT_WAYPOINT);
        send_message(MSG_GPS_RAW);
        send_message(MSG_NAV_CONTROLLER_OUTPUT);
        send_message(MSG_LIMITS_STATUS);
    }

    if (copter.gcs_out_of_time) return;

    if (stream_trigger(STREAM_POSITION)) {
        send_message(MSG_LOCATION);
        send_message(MSG_LOCAL_POSITION);
    }

    if (copter.gcs_out_of_time) return;

    if (stream_trigger(STREAM_RAW_CONTROLLER)) {
        send_message(MSG_SERVO_OUT);
    }

    if (copter.gcs_out_of_time) return;

    if (stream_trigger(STREAM_RC_CHANNELS)) {
        send_message(MSG_RADIO_OUT);
        send_message(MSG_RADIO_IN);
    }

    if (copter.gcs_out_of_time) return;

    if (stream_trigger(STREAM_EXTRA1)) {
        send_message(MSG_ATTITUDE);
        send_message(MSG_SIMSTATE);
        send_message(MSG_PID_TUNING);
    }

    if (copter.gcs_out_of_time) return;

    if (stream_trigger(STREAM_EXTRA2)) {
        send_message(MSG_VFR_HUD);
    }

    if (copter.gcs_out_of_time) return;

    if (stream_trigger(STREAM_EXTRA3)) {
        send_message(MSG_AHRS);
        send_message(MSG_HWSTATUS);
        send_message(MSG_SYSTEM_TIME);

        send_message(MSG_RANGEFINDER);
#if AP_TERRAIN_AVAILABLE
        send_message(MSG_TERRAIN);
#endif
        send_message(MSG_BATTERY2);
        send_message(MSG_MOUNT_STATUS);
```

```
        send_message(MSG_OPTICAL_FLOW);
        send_message(MSG_GIMBAL_REPORT);
        send_message(MSG_EKF_STATUS_REPORT);
        send_message(MSG_VIBRATION);
    }
}


void GCS_MAVLINK::handle_guided_request(AP_Mission::Mission_Command &cmd)
{
    copter.do_guided(cmd);
}

void GCS_MAVLINK::handle_change_alt_request(AP_Mission::Mission_Command &cmd)
{
    // add home alt if needed
    if (cmd.content.location.flags.relative_alt) {
        cmd.content.location.alt += copter.ahrs.get_home().alt;
    }

    // To-Do: update target altitude for loiter or waypoint controller depending upon nav mode
}

void GCS_MAVLINK::handleMessage(mavlink_message_t* msg)
{
    uint8_t result = MAV_RESULT_FAILED;         // assume failure.  Each messages id is responsible for return ACK or NAK if requi

    switch (msg->msgid) {

    case MAVLINK_MSG_ID_HEARTBEAT:      // MAV ID: 0
    {
        // We keep track of the last time we received a heartbeat from our GCS for failsafe purposes
        if(msg->sysid != copter.g.sysid_my_gcs) break;
        copter.failsafe.last_heartbeat_ms = hal.scheduler->millis();
        copter.pmTest1++;
        break;
    }

    case MAVLINK_MSG_ID_SET_MODE:       // MAV ID: 11
    {
        handle_set_mode(msg, FUNCTOR_BIND(&copter, &Copter::set_mode, bool, uint8_t));
        break;
    }

    case MAVLINK_MSG_ID_PARAM_REQUEST_READ:         // MAV ID: 20
    {
        handle_param_request_read(msg);
        break;
    }

    case MAVLINK_MSG_ID_PARAM_REQUEST_LIST:         // MAV ID: 21
    {
        // mark the firmware version in the tlog
        send_text_P(SEVERITY_LOW, PSTR(FIRMWARE_STRING));

#if defined(PX4_GIT_VERSION) && defined(NUTTX_GIT_VERSION)
        send_text_P(SEVERITY_LOW, PSTR("PX4: " PX4_GIT_VERSION " NuttX: " NUTTX_GIT_VERSION));
#endif
        send_text_P(SEVERITY_LOW, PSTR("Frame: " FRAME_CONFIG_STRING));
        handle_param_request_list(msg);
        break;
    }

    case MAVLINK_MSG_ID_PARAM_SET:     // 23
    {
        handle_param_set(msg, &copter.DataFlash);
        break;
    }

    case MAVLINK_MSG_ID_MISSION_WRITE_PARTIAL_LIST: // MAV ID: 38
    {
        handle_mission_write_partial_list(copter.mission, msg);
        break;
    }


    // GCS has sent us a command from GCS, store to EEPROM
    case MAVLINK_MSG_ID_MISSION_ITEM:           // MAV ID: 39
    {
        if (handle_mission_item(msg, copter.mission)) {
            copter.Log_Write_EntireMission();
```

```
        }
        break;
    }

    // read an individual command from EEPROM and send it to the GCS
    case MAVLINK_MSG_ID_MISSION_REQUEST:     // MAV ID: 40
    {
        handle_mission_request(copter.mission, msg);
        break;
    }

    case MAVLINK_MSG_ID_MISSION_SET_CURRENT:    // MAV ID: 41
    {
        handle_mission_set_current(copter.mission, msg);
        break;
    }

    // GCS request the full list of commands, we return just the number and leave the GCS to then request each command individuall
    case MAVLINK_MSG_ID_MISSION_REQUEST_LIST:       // MAV ID: 43
    {
        handle_mission_request_list(copter.mission, msg);
        break;
    }

    // GCS provides the full number of commands it wishes to upload
    //  individual commands will then be sent from the GCS using the MAVLINK_MSG_ID_MISSION_ITEM message
    case MAVLINK_MSG_ID_MISSION_COUNT:          // MAV ID: 44
    {
        handle_mission_count(copter.mission, msg);
        break;
    }

    case MAVLINK_MSG_ID_MISSION_CLEAR_ALL:      // MAV ID: 45
    {
        handle_mission_clear_all(copter.mission, msg);
        break;
    }

    case MAVLINK_MSG_ID_REQUEST_DATA_STREAM:    // MAV ID: 66
    {
        handle_request_data_stream(msg, false);
        break;
    }

    case MAVLINK_MSG_ID_GIMBAL_REPORT:
    {
#if MOUNT == ENABLED
        handle_gimbal_report(copter.camera_mount, msg);
#endif
        break;
    }

    case MAVLINK_MSG_ID_RC_CHANNELS_OVERRIDE:       // MAV ID: 70
    {
        // allow override of RC channel values for HIL
        // or for complete GCS control of switch position
        // and RC PWM values.
        if(msg->sysid != copter.g.sysid_my_gcs) break;                          // Only accept control from our gcs
        mavlink_rc_channels_override_t packet;
        int16_t v[8];
        mavlink_msg_rc_channels_override_decode(msg, &packet);

        v[0] = packet.chan1_raw;
        v[1] = packet.chan2_raw;
        v[2] = packet.chan3_raw;
        v[3] = packet.chan4_raw;
        v[4] = packet.chan5_raw;
        v[5] = packet.chan6_raw;
        v[6] = packet.chan7_raw;
        v[7] = packet.chan8_raw;
        hal.rcin->set_overrides(v, 8);

        // record that rc are overwritten so we can trigger a failsafe if we lose contact with groundstation
        copter.failsafe.rc_override_active = true;

        // a RC override message is consiered to be a 'heartbeat' from the ground station for failsafe purposes
        copter.failsafe.last_heartbeat_ms = hal.scheduler->millis();
        break;
    }

    // Pre-Flight calibration requests
```

```
1104    case MAVLINK_MSG_ID_COMMAND_LONG:       // MAV ID: 76
1105    {
1106        // decode packet
1107        mavlink_command_long_t packet;
1108        mavlink_msg_command_long_decode(msg, &packet);
1109
1110        switch(packet.command) {
1111
1112        case MAV_CMD_START_RX_PAIR:
1113            // initiate bind procedure
1114            if (!hal.rcin->rc_bind(packet.param1)) {
1115                result = MAV_RESULT_FAILED;
1116            } else {
1117                result = MAV_RESULT_ACCEPTED;
1118            }
1119            break;
1120
1121        case MAV_CMD_NAV_TAKEOFF: {
1122            // param3 : horizontal navigation by pilot acceptable
1123            // param4 : yaw angle   (not supported)
1124            // param5 : latitude    (not supported)
1125            // param6 : longitude   (not supported)
1126            // param7 : altitude [metres]
1127
1128            float takeoff_alt = packet.param7 * 100;      // Convert m to cm
1129
1130            if(copter.do_user_takeoff(takeoff_alt, is_zero(packet.param3))) {
1131                result = MAV_RESULT_ACCEPTED;
1132            } else {
1133                result = MAV_RESULT_FAILED;
1134            }
1135            break;
1136        }
1137
1138
1139        case MAV_CMD_NAV_LOITER_UNLIM:
1140            if (copter.set_mode(LOITER)) {
1141                result = MAV_RESULT_ACCEPTED;
1142            }
1143            break;
1144
1145        case MAV_CMD_NAV_RETURN_TO_LAUNCH:
1146            if (copter.set_mode(RTL)) {
1147                result = MAV_RESULT_ACCEPTED;
1148            }
1149            break;
1150
1151        case MAV_CMD_NAV_LAND:
1152            if (copter.set_mode(LAND)) {
1153                result = MAV_RESULT_ACCEPTED;
1154            }
1155            break;
1156
1157        case MAV_CMD_CONDITION_YAW:
1158            // param1 : target angle [0-360]
1159            // param2 : speed during change [deg per second]
1160            // param3 : direction (-1:ccw, +1:cw)
1161            // param4 : relative offset (1) or absolute angle (0)
1162            if ((packet.param1 >= 0.0f)   &&
1163                (packet.param1 <= 360.0f) &&
1164                (is_zero(packet.param4) || is_equal(packet.param4,1.0f))) {
1165                copter.set_auto_yaw_look_at_heading(packet.param1, packet.param2, (int8_t)packet.param3, (uint8_t)packet.param4);
1166                result = MAV_RESULT_ACCEPTED;
1167            } else {
1168                result = MAV_RESULT_FAILED;
1169            }
1170            break;
1171
1172        case MAV_CMD_DO_CHANGE_SPEED:
1173            // param1 : unused
1174            // param2 : new speed in m/s
1175            // param3 : unused
1176            // param4 : unused
1177            if (packet.param2 > 0.0f) {
1178                copter.wp_nav.set_speed_xy(packet.param2 * 100.0f);
1179                result = MAV_RESULT_ACCEPTED;
1180            } else {
1181                result = MAV_RESULT_FAILED;
1182            }
1183            break;
```

```
1184
1185        case MAV_CMD_DO_SET_HOME:
1186            // param1 : use current (1=use current location, 0=use specified location)
1187            // param5 : latitude
1188            // param6 : longitude
1189            // param7 : altitude (absolute)
1190            result = MAV_RESULT_FAILED; // assume failure
1191            if(is_equal(packet.param1,1.0f) || (is_zero(packet.param5) && is_zero(packet.param6) && is_zero(packet.param7))) {
1192                if (copter.set_home_to_current_location_and_lock()) {
1193                    result = MAV_RESULT_ACCEPTED;
1194                }
1195            } else {
1196                Location new_home_loc;
1197                new_home_loc.lat = (int32_t)(packet.param5 * 1.0e7f);
1198                new_home_loc.lng = (int32_t)(packet.param6 * 1.0e7f);
1199                new_home_loc.alt = (int32_t)(packet.param7 * 100.0f);
1200                if (!copter.far_from_EKF_origin(new_home_loc)) {
1201                    if (copter.set_home_and_lock(new_home_loc)) {
1202                        result = MAV_RESULT_ACCEPTED;
1203                    }
1204                }
1205            }
1206            break;
1207
1208        case MAV_CMD_DO_SET_ROI:
1209            // param1 : regional of interest mode (not supported)
1210            // param2 : mission index/ target id (not supported)
1211            // param3 : ROI index (not supported)
1212            // param5 : x / lat
1213            // param6 : y / lon
1214            // param7 : z / alt
1215            Location roi_loc;
1216            roi_loc.lat = (int32_t)(packet.param5 * 1.0e7f);
1217            roi_loc.lng = (int32_t)(packet.param6 * 1.0e7f);
1218            roi_loc.alt = (int32_t)(packet.param7 * 100.0f);
1219            copter.set_auto_yaw_roi(roi_loc);
1220            result = MAV_RESULT_ACCEPTED;
1221            break;
1222
1223        case MAV_CMD_MISSION_START:
1224            if (copter.motors.armed() && copter.set_mode(AUTO)) {
1225                copter.set_auto_armed(true);
1226                result = MAV_RESULT_ACCEPTED;
1227            }
1228            break;
1229
1230        case MAV_CMD_PREFLIGHT_CALIBRATION:
1231            // exit immediately if armed
1232            if (copter.motors.armed()) {
1233                result = MAV_RESULT_FAILED;
1234                break;
1235            }
1236            if (is_equal(packet.param1,1.0f)) {
1237                // gyro offset calibration
1238                copter.ins.init_gyro();
1239                // reset ahrs gyro bias
1240                if (copter.ins.gyro_calibrated_ok_all()) {
1241                    copter.ahrs.reset_gyro_drift();
1242                    result = MAV_RESULT_ACCEPTED;
1243                } else {
1244                    result = MAV_RESULT_FAILED;
1245                }
1246            } else if (is_equal(packet.param3,1.0f)) {
1247                // fast barometer calibration
1248                copter.init_barometer(false);
1249                result = MAV_RESULT_ACCEPTED;
1250            } else if (is_equal(packet.param4,1.0f)) {
1251                result = MAV_RESULT_UNSUPPORTED;
1252            } else if (is_equal(packet.param5,1.0f)) {
1253                // 3d accel calibration
1254                float trim_roll, trim_pitch;
1255                // this blocks
1256                AP_InertialSensor_UserInteract_MAVLink interact(this);
1257                if(copter.ins.calibrate_accel(&interact, trim_roll, trim_pitch)) {
1258                    // reset ahrs's trim to suggested values from calibration routine
1259                    copter.ahrs.set_trim(Vector3f(trim_roll, trim_pitch, 0));
1260                    result = MAV_RESULT_ACCEPTED;
1261                } else {
1262                    result = MAV_RESULT_FAILED;
1263                }
```

```
            } else if (is_equal(packet.param5,2.0f)) {
                // accel trim
                float trim_roll, trim_pitch;
                if(copter.ins.calibrate_trim(trim_roll, trim_pitch)) {
                    // reset ahrs's trim to suggested values from calibration routine
                    copter.ahrs.set_trim(Vector3f(trim_roll, trim_pitch, 0));
                    result = MAV_RESULT_ACCEPTED;
                } else {
                    result = MAV_RESULT_FAILED;
                }
            } else if (is_equal(packet.param6,1.0f)) {
                // compassmot calibration
                result = copter.mavlink_compassmot(chan);
            }
            break;

        case MAV_CMD_PREFLIGHT_SET_SENSOR_OFFSETS:
            if (is_equal(packet.param1,2.0f)) {
                // save first compass's offsets
                copter.compass.set_and_save_offsets(0, packet.param2, packet.param3, packet.param4);
                result = MAV_RESULT_ACCEPTED;
            }
            if (is_equal(packet.param1,5.0f)) {
                // save secondary compass's offsets
                copter.compass.set_and_save_offsets(1, packet.param2, packet.param3, packet.param4);
                result = MAV_RESULT_ACCEPTED;
            }
            break;

        case MAV_CMD_COMPONENT_ARM_DISARM:
            if (is_equal(packet.param1,1.0f)) {
                // attempt to arm and return success or failure
                if (copter.init_arm_motors(true)) {
                    result = MAV_RESULT_ACCEPTED;
                }
            } else if (is_zero(packet.param1) && (copter.mode_has_manual_throttle(copter.control_mode) || copter.ap.land_complete
                copter.init_disarm_motors();
                result = MAV_RESULT_ACCEPTED;
            } else {
                result = MAV_RESULT_UNSUPPORTED;
            }
            break;

        case MAV_CMD_DO_SET_SERVO:
            if (copter.ServoRelayEvents.do_set_servo(packet.param1, packet.param2)) {
                result = MAV_RESULT_ACCEPTED;
            }
            break;

        case MAV_CMD_DO_REPEAT_SERVO:
            if (copter.ServoRelayEvents.do_repeat_servo(packet.param1, packet.param2, packet.param3, packet.param4*1000)) {
                result = MAV_RESULT_ACCEPTED;
            }
            break;

        case MAV_CMD_DO_SET_RELAY:
            if (copter.ServoRelayEvents.do_set_relay(packet.param1, packet.param2)) {
                result = MAV_RESULT_ACCEPTED;
            }
            break;

        case MAV_CMD_DO_REPEAT_RELAY:
            if (copter.ServoRelayEvents.do_repeat_relay(packet.param1, packet.param2, packet.param3*1000)) {
                result = MAV_RESULT_ACCEPTED;
            }
            break;

        case MAV_CMD_PREFLIGHT_REBOOT_SHUTDOWN:
            if (is_equal(packet.param1,1.0f) || is_equal(packet.param1,3.0f)) {
                AP_Notify::events.firmware_update = 1;
                copter.update_notify();
                hal.scheduler->delay(50);
                // when packet.param1 == 3 we reboot to hold in bootloader
                hal.scheduler->reboot(is_equal(packet.param1,3.0f));
                result = MAV_RESULT_ACCEPTED;
            }
            break;

        case MAV_CMD_DO_FENCE_ENABLE:
#if AC_FENCE == ENABLED
            result = MAV_RESULT_ACCEPTED;
```

```
1344
1345            switch ((uint16_t)packet.param1) {
1346                case 0:
1347                    copter.fence.enable(false);
1348                    break;
1349                case 1:
1350                    copter.fence.enable(true);
1351                    break;
1352                default:
1353                    result = MAV_RESULT_FAILED;
1354                    break;
1355            }
1356 #else
1357            // if fence code is not included return failure
1358            result = MAV_RESULT_FAILED;
1359 #endif
1360            break;
1361
1362 #if PARACHUTE == ENABLED
1363        case MAV_CMD_DO_PARACHUTE:
1364            // configure or release parachute
1365            result = MAV_RESULT_ACCEPTED;
1366            switch ((uint16_t)packet.param1) {
1367                case PARACHUTE_DISABLE:
1368                    copter.parachute.enabled(false);
1369                    copter.Log_Write_Event(DATA_PARACHUTE_DISABLED);
1370                    break;
1371                case PARACHUTE_ENABLE:
1372                    copter.parachute.enabled(true);
1373                    copter.Log_Write_Event(DATA_PARACHUTE_ENABLED);
1374                    break;
1375                case PARACHUTE_RELEASE:
1376                    // treat as a manual release which performs some additional check of altitude
1377                    copter.parachute_manual_release();
1378                    break;
1379                default:
1380                    result = MAV_RESULT_FAILED;
1381                    break;
1382            }
1383            break;
1384 #endif
1385
1386        case MAV_CMD_DO_MOTOR_TEST:
1387            // param1 : motor sequence number (a number from 1 to max number of motors on the vehicle)
1388            // param2 : throttle type (0=throttle percentage, 1=PWM, 2=pilot throttle channel pass-through. See MOTOR_TEST_THROTTL
1389            // param3 : throttle (range depends upon param2)
1390            // param4 : timeout (in seconds)
1391            result = copter.mavlink_motor_test_start(chan, (uint8_t)packet.param1, (uint8_t)packet.param2, (uint16_t)packet.param3
1392            break;
1393
1394 #if EPM_ENABLED == ENABLED
1395        case MAV_CMD_DO_GRIPPER:
1396            // param1 : gripper number (ignored)
1397            // param2 : action (0=release, 1=grab). See GRIPPER_ACTIONS enum.
1398            if(!copter.epm.enabled()) {
1399                result = MAV_RESULT_FAILED;
1400            } else {
1401                result = MAV_RESULT_ACCEPTED;
1402                switch ((uint8_t)packet.param2) {
1403                    case GRIPPER_ACTION_RELEASE:
1404                        copter.epm.release();
1405                        break;
1406                    case GRIPPER_ACTION_GRAB:
1407                        copter.epm.grab();
1408                        break;
1409                    default:
1410                        result = MAV_RESULT_FAILED;
1411                        break;
1412                }
1413            }
1414            break;
1415 #endif

1416
1417        case MAV_CMD_REQUEST_AUTOPILOT_CAPABILITIES: {
1418            if (is_equal(packet.param1,1.0f)) {
1419                copter.gcs[chan-MAVLINK_COMM_0].send_autopilot_version();
1420                result = MAV_RESULT_ACCEPTED;
1421            }
1422            break;
1423        }
```

```
1424
1425            default:
1426                result = MAV_RESULT_UNSUPPORTED;
1427                break;
1428            }
1429
1430            // send ACK or NAK
1431            mavlink_msg_command_ack_send_buf(msg, chan, packet.command, result);
1432
1433            break;
1434        }
1435
1436        case MAVLINK_MSG_ID_COMMAND_ACK:        // MAV ID: 77
1437        {
1438            copter.command_ack_counter++;
1439            break;
1440        }
1441
1442        case MAVLINK_MSG_ID_SET_POSITION_TARGET_LOCAL_NED:     // MAV ID: 84
1443        {
1444            // decode packet
1445            mavlink_set_position_target_local_ned_t packet;
1446            mavlink_msg_set_position_target_local_ned_decode(msg, &packet);
1447
1448            // exit if vehicle is not in Guided mode or Auto-Guided mode
1449            if ((copter.control_mode != GUIDED) && !(copter.control_mode == AUTO && copter.auto_mode == Auto_NavGuided)) {
1450                break;
1451            }
1452
1453            bool pos_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_POS_IGNORE;
1454            bool vel_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_VEL_IGNORE;
1455            bool acc_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_ACC_IGNORE;
1456
1457            /*
1458             * for future use:
1459             * bool force           = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_FORCE;
1460             * bool yaw_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_YAW_IGNORE;
1461             * bool yaw_rate_ignore = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_YAW_RATE_IGNORE;
1462             */
1463
1464            if (!pos_ignore && !vel_ignore && acc_ignore) {
1465                Vector3f pos_ned = Vector3f(packet.x * 100.0f, packet.y * 100.0f, -packet.z * 100.0f);
1466                pos_ned.z = copter.pv_alt_above_origin(pos_ned.z);
1467                copter.guided_set_destination_posvel(pos_ned, Vector3f(packet.vx * 100.0f, packet.vy * 100.0f, -packet.vz * 100.0f));
1468            } else if (pos_ignore && !vel_ignore && acc_ignore) {
1469                copter.guided_set_velocity(Vector3f(packet.vx * 100.0f, packet.vy * 100.0f, -packet.vz * 100.0f));
1470            } else if (!pos_ignore && vel_ignore && acc_ignore) {
1471                Vector3f pos_ned = Vector3f(packet.x * 100.0f, packet.y * 100.0f, -packet.z * 100.0f);
1472                pos_ned.z = copter.pv_alt_above_origin(pos_ned.z);
1473                copter.guided_set_destination(pos_ned);
1474            } else {
1475                result = MAV_RESULT_FAILED;
1476            }
1477
1478            break;
1479        }
1480
1481        case MAVLINK_MSG_ID_SET_POSITION_TARGET_GLOBAL_INT:    // MAV ID: 86
1482        {
1483            // decode packet
1484            mavlink_set_position_target_global_int_t packet;
1485            mavlink_msg_set_position_target_global_int_decode(msg, &packet);
1486
1487            // exit if vehicle is not in Guided mode or Auto-Guided mode
1488            if ((copter.control_mode != GUIDED) && !(copter.control_mode == AUTO && copter.auto_mode == Auto_NavGuided)) {
1489                break;
1490            }
1491
1492            bool pos_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_POS_IGNORE;
1493            bool vel_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_VEL_IGNORE;
1494            bool acc_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_ACC_IGNORE;
1495
1496            /*
1497             * for future use:
1498             * bool force           = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_FORCE;
1499             * bool yaw_ignore      = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_YAW_IGNORE;
1500             * bool yaw_rate_ignore = packet.type_mask & MAVLINK_SET_POS_TYPE_MASK_YAW_RATE_IGNORE;
1501             */
1502
1503            Vector3f pos_ned;
```

```
1504
1505            if(!pos_ignore) {
1506                Location loc;
1507                loc.lat = packet.lat_int;
1508                loc.lng = packet.lon_int;
1509                loc.alt = packet.alt*100;
1510                switch (packet.coordinate_frame) {
1511                    case MAV_FRAME_GLOBAL_RELATIVE_ALT:
1512                    case MAV_FRAME_GLOBAL_RELATIVE_ALT_INT:
1513                        loc.flags.relative_alt = true;
1514                        loc.flags.terrain_alt = false;
1515                        break;
1516                    case MAV_FRAME_GLOBAL_TERRAIN_ALT:
1517                    case MAV_FRAME_GLOBAL_TERRAIN_ALT_INT:
1518                        loc.flags.relative_alt = true;
1519                        loc.flags.terrain_alt = true;
1520                        break;
1521                    case MAV_FRAME_GLOBAL:
1522                    case MAV_FRAME_GLOBAL_INT:
1523                    default:
1524                        loc.flags.relative_alt = false;
1525                        loc.flags.terrain_alt = false;
1526                        break;
1527                }
1528                pos_ned = copter.pv_location_to_vector(loc);
1529            }
1530
1531            if (!pos_ignore && !vel_ignore && acc_ignore) {
1532                copter.guided_set_destination_posvel(pos_ned, Vector3f(packet.vx * 100.0f, packet.vy * 100.0f, -packet.vz * 100.0f));
1533            } else if (pos_ignore && !vel_ignore && acc_ignore) {
1534                copter.guided_set_velocity(Vector3f(packet.vx * 100.0f, packet.vy * 100.0f, -packet.vz * 100.0f));
1535            } else if (!pos_ignore && vel_ignore && acc_ignore) {
1536                copter.guided_set_destination(pos_ned);
1537            } else {
1538                result = MAV_RESULT_FAILED;
1539            }
1540
1541            break;
1542        }
1543
1544    #if HIL_MODE != HIL_MODE_DISABLED
1545        case MAVLINK_MSG_ID_HIL_STATE:          // MAV ID: 90
1546        {
1547            mavlink_hil_state_t packet;
1548            mavlink_msg_hil_state_decode(msg, &packet);
1549
1550            // set gps hil sensor
1551            Location loc;
1552            loc.lat = packet.lat;
1553            loc.lng = packet.lon;
1554            loc.alt = packet.alt/10;
1555            Vector3f vel(packet.vx, packet.vy, packet.vz);
1556            vel *= 0.01f;
1557
1558            gps.setHIL(0, AP_GPS::GPS_OK_FIX_3D,
1559                       packet.time_usec/1000,
1560                       loc, vel, 10, 0, true);
1561
1562            // rad/sec
1563            Vector3f gyros;
1564            gyros.x = packet.rollspeed;
1565            gyros.y = packet.pitchspeed;
1566            gyros.z = packet.yawspeed;
1567
1568            // m/s/s
1569            Vector3f accels;
1570            accels.x = packet.xacc * (GRAVITY_MSS/1000.0f);
1571            accels.y = packet.yacc * (GRAVITY_MSS/1000.0f);
1572            accels.z = packet.zacc * (GRAVITY_MSS/1000.0f);
1573
1574            ins.set_gyro(0, gyros);

1575
1576            ins.set_accel(0, accels);
1577
1578            copter.barometer.setHIL(packet.alt*0.001f);
1579            copter.compass.setHIL(0, packet.roll, packet.pitch, packet.yaw);
1580            copter.compass.setHIL(1, packet.roll, packet.pitch, packet.yaw);
1581
1582            break;
1583        }
```

```
#endif //  HIL_MODE != HIL_MODE_DISABLED

    case MAVLINK_MSG_ID_RADIO:
    case MAVLINK_MSG_ID_RADIO_STATUS:       // MAV ID: 109
    {
        handle_radio_status(msg, copter.DataFlash, copter.should_log(MASK_LOG_PM));
        break;
    }

    case MAVLINK_MSG_ID_LOG_REQUEST_DATA:
    case MAVLINK_MSG_ID_LOG_ERASE:
        copter.in_log_download = true;
        // fallthru
    case MAVLINK_MSG_ID_LOG_REQUEST_LIST:
        if (!copter.in_mavlink_delay && !copter.motors.armed()) {
            handle_log_message(msg, copter.DataFlash);
        }
        break;
    case MAVLINK_MSG_ID_LOG_REQUEST_END:
        copter.in_log_download = false;
        if (!copter.in_mavlink_delay && !copter.motors.armed()) {
            handle_log_message(msg, copter.DataFlash);
        }
        break;

#if HAL_CPU_CLASS > HAL_CPU_CLASS_16
    case MAVLINK_MSG_ID_SERIAL_CONTROL:
        handle_serial_control(msg, copter.gps);
        break;

    case MAVLINK_MSG_ID_GPS_INJECT_DATA:
        handle_gps_inject(msg, copter.gps);
        result = MAV_RESULT_ACCEPTED;
        break;

#endif

#if CAMERA == ENABLED
    case MAVLINK_MSG_ID_DIGICAM_CONFIGURE:      // MAV ID: 202
        break;

    case MAVLINK_MSG_ID_DIGICAM_CONTROL:
        copter.camera.control_msg(msg);
        copter.log_picture();
        break;
#endif // CAMERA == ENABLED

#if MOUNT == ENABLED
    case MAVLINK_MSG_ID_MOUNT_CONFIGURE:        // MAV ID: 204
        copter.camera_mount.configure_msg(msg);
        break;

    case MAVLINK_MSG_ID_MOUNT_CONTROL:
        copter.camera_mount.control_msg(msg);
        break;
#endif // MOUNT == ENABLED

    case MAVLINK_MSG_ID_TERRAIN_DATA:
    case MAVLINK_MSG_ID_TERRAIN_CHECK:
#if AP_TERRAIN_AVAILABLE
        copter.terrain.handle_data(chan, msg);
#endif
        break;

#if AC_RALLY == ENABLED
    // receive a rally point from GCS and store in EEPROM
    case MAVLINK_MSG_ID_RALLY_POINT: {
        mavlink_rally_point_t packet;
        mavlink_msg_rally_point_decode(msg, &packet);


        if (packet.idx >= copter.rally.get_rally_total() ||
            packet.idx >= copter.rally.get_rally_max()) {
            send_text_P(SEVERITY_LOW,PSTR("bad rally point message ID"));
            break;
        }

        if (packet.count != copter.rally.get_rally_total()) {
            send_text_P(SEVERITY_LOW,PSTR("bad rally point message count"));
            break;
        }
```

```
1664
1665            RallyLocation rally_point;
1666            rally_point.lat = packet.lat;
1667            rally_point.lng = packet.lng;
1668            rally_point.alt = packet.alt;
1669            rally_point.break_alt = packet.break_alt;
1670            rally_point.land_dir = packet.land_dir;
1671            rally_point.flags = packet.flags;
1672
1673            if (!copter.rally.set_rally_point_with_index(packet.idx, rally_point)) {
1674                send_text_P(SEVERITY_HIGH, PSTR("error setting rally point"));
1675            }
1676
1677            break;
1678        }
1679
1680        //send a rally point to the GCS
1681        case MAVLINK_MSG_ID_RALLY_FETCH_POINT: {
1682            //send_text_P(SEVERITY_HIGH, PSTR("## getting rally point in GCS_Mavlink.cpp 1")); // #### TEMP
1683
1684            mavlink_rally_fetch_point_t packet;
1685            mavlink_msg_rally_fetch_point_decode(msg, &packet);
1686
1687            //send_text_P(SEVERITY_HIGH, PSTR("## getting rally point in GCS_Mavlink.cpp 2")); // #### TEMP
1688
1689            if (packet.idx > copter.rally.get_rally_total()) {
1690                send_text_P(SEVERITY_LOW, PSTR("bad rally point index"));
1691                break;
1692            }
1693
1694            //send_text_P(SEVERITY_HIGH, PSTR("## getting rally point in GCS_Mavlink.cpp 3")); // #### TEMP
1695
1696            RallyLocation rally_point;
1697            if (!copter.rally.get_rally_point_with_index(packet.idx, rally_point)) {
1698               send_text_P(SEVERITY_LOW, PSTR("failed to set rally point"));
1699               break;
1700            }
1701
1702            //send_text_P(SEVERITY_HIGH, PSTR("## getting rally point in GCS_Mavlink.cpp 4")); // #### TEMP
1703
1704            mavlink_msg_rally_point_send_buf(msg,
1705                                             chan, msg->sysid, msg->compid, packet.idx,
1706                                             copter.rally.get_rally_total(), rally_point.lat, rally_point.lng,
1707                                             rally_point.alt, rally_point.break_alt, rally_point.land_dir,
1708                                             rally_point.flags);
1709
1710            //send_text_P(SEVERITY_HIGH, PSTR("## getting rally point in GCS_Mavlink.cpp 5")); // #### TEMP
1711
1712            break;
1713        }
1714    #endif // AC_RALLY == ENABLED
1715
1716        case MAVLINK_MSG_ID_AUTOPILOT_VERSION_REQUEST:
1717            copter.gcs[chan-MAVLINK_COMM_0].send_autopilot_version();
1718            break;
1719
1720        case MAVLINK_MSG_ID_LED_CONTROL:
1721            // send message to Notify
1722            AP_Notify::handle_led_control(msg);
1723            break;
1724
1725        }     // end switch
1726    } // end handle mavlink
1727
1728
1729    /*
1730     *  a delay() callback that processes MAVLink packets. We set this as the
1731     *  callback in long running library initialisation routines to allow
1732     *  MAVLink to process packets while waiting for the initialisation to
1733     *  complete
1734     */
1735    void Copter::mavlink_delay_cb()
1736    {
1737        static uint32_t last_1hz, last_50hz, last_5s;
1738        if (!gcs[0].initialised || in_mavlink_delay) return;
1739
1740        in_mavlink_delay = true;
1741
1742        uint32_t tnow = millis();
1743        if (tnow - last_1hz > 1000) {
```

```
        last_1hz = tnow;
        gcs_send_heartbeat();
        gcs_send_message(MSG_EXTENDED_STATUS1);
    }
    if (tnow - last_50hz > 20) {
        last_50hz = tnow;
        gcs_check_input();
        gcs_data_stream_send();
        gcs_send_deferred();
        notify.update();
    }
    if (tnow - last_5s > 5000) {
        last_5s = tnow;
        gcs_send_text_P(SEVERITY_LOW, PSTR("Initialising APM..."));
    }
    check_usb_mux();

    in_mavlink_delay = false;
}

/*
 *  send a message on both GCS links
 */
void Copter::gcs_send_message(enum ap_message id)
{
    for (uint8_t i=0; i<num_gcs; i++) {
        if (gcs[i].initialised) {
            gcs[i].send_message(id);
        }
    }
}

/*
 *  send data streams in the given rate range on both links
 */
void Copter::gcs_data_stream_send(void)
{
    for (uint8_t i=0; i<num_gcs; i++) {
        if (gcs[i].initialised) {
            gcs[i].data_stream_send();
        }
    }
}

/*
 *  look for incoming commands on the GCS links
 */
void Copter::gcs_check_input(void)
{
    for (uint8_t i=0; i<num_gcs; i++) {
        if (gcs[i].initialised) {
#if CLI_ENABLED == ENABLED
            gcs[i].update(g.cli_enabled==1?FUNCTOR_BIND_MEMBER(&Copter::run_cli, void, AP_HAL::UARTDriver *):NULL);
#else
            gcs[i].update(NULL);
#endif
        }
    }
}

void Copter::gcs_send_text_P(gcs_severity severity, const prog_char_t *str)
{
    for (uint8_t i=0; i<num_gcs; i++) {
        if (gcs[i].initialised) {
            gcs[i].send_text_P(severity, str);
        }
    }
}

/*
 *  send a low priority formatted message to the GCS
 *  only one fits in the queue, so if you send more than one before the
 *  last one gets into the serial buffer then the old one will be lost
 */
void Copter::gcs_send_text_fmt(const prog_char_t *fmt, ...)
{
    va_list arg_list;
    gcs[0].pending_status.severity = (uint8_t)SEVERITY_LOW;
    va_start(arg_list, fmt);
    hal.util->vsnprintf_P((char *)gcs[0].pending_status.text,
```

```
1824            sizeof(gcs[0].pending_status.text), fmt, arg_list);
1825    va_end(arg_list);
1826    gcs[0].send_message(MSG_STATUSTEXT);
1827    for (uint8_t i=1; i<num_gcs; i++) {
1828        if (gcs[i].initialised) {
1829            gcs[i].pending_status = gcs[0].pending_status;
1830            gcs[i].send_message(MSG_STATUSTEXT);
1831        }
1832    }
1833 }
```