Commander 코드 분석

오픈 소스 PX4 를 지원하는 비행제어장치 설계 및 시스템 분석(EA20180971)

박동희

# Commander

PX4의 비즈니스 로직. 상태머신

Commander App은 비행 명령과 비행체 정보를 모아 수행할 명령을 결정하고, 다른 App에게 명령을 내린다.

관련 코드: <https://github.com/PX4/Firmware/tree/master/src/modules/commander>

## Commander App

Commander 앱에서 하는 일. 비행에 관련된 의사 결정을 한다. 기체 상태에 관한 토픽을 읽고, 명령을 내린다.

주요 상태 및 명령

* 비행모드 전환 및 수행지시 (상태머신으로 구현)
* 파라미터 업데이트 (parameter\_update)
* RC입력 읽기(manual\_control\_setpoint)
* 센서값(sensor\_combined) 읽기
* 시스템 파워 관리: 파워를 선택
* Safety 스위치(safety) 체크
* 기체상태 관리(vtol\_vehicle\_status)
* 기체 위치(vehicle\_global\_position)
* 기체 로컬 위치(vehicle\_local\_position)
* 기체 자세(vehicle\_attitude)
* 기체 착륙 여부(vehicle\_land\_detected)
* CPU로드, 배터리 상태 체크(cpuload, batterystatus)

## 관련 토픽

Commander 에서 출판(Publish)하고 구독(Subscribe)하는 토픽

### Commander 에서 Publish 토픽

armed\_pub  
commander\_state\_pub  
homePub  
led\_control\_pub  
mission\_pub  
command\_ack\_pub  
control\_mode\_pub  
roi\_pub  
status\_pub  
status\_pub  
vehicle\_status\_flags\_pub

### Commander 에서 구독하는 토픽

다음은 Commander App에서 구독하는 토픽이다.. PX4에서 사용되는 대부분의 토픽을 구독한다.

토픽에 대한 자세한 내용은 msg 디렉토리 참고. <https://github.com/PX4/Firmware/tree/master/msg>

\_VEHICLE\_ATTITUDE\_CONTROLS  
battery\_status  
cpuload  
differential\_pressure  
estimator\_status  
geofence\_result  
manual\_control\_setpoint  
mission\_result  
offboard\_control\_mode  
parameter\_update  
position\_setpoint\_triplet  
safety  
sensor\_accel  
sensor\_combined  
sensor\_correction  
sensor\_gyro  
sensor\_mag  
sensor\_preflight  
subsystem\_info  
system\_power  
telemetry\_status  
vehicle\_attitude  
vehicle\_command  
vehicle\_global\_position  
vehicle\_gps\_position  
vehicle\_land\_detected  
vehicle\_local\_position  
vehicle\_status  
vtol\_vehicle\_status

## State Mahcine

모드 변환 의사 결정 상태머신

set\_nav\_state()

main\_state\_transition()

arming\_state\_transition()

## 주요 함수

int commander\_main(int argc, char \*argv[])

Commander App 시작 함수. 데몬의 상태 출력하거나 기체에 명령(arm, disarm, preflight check, takeoff, land, calibration, 모드전환)을 내릴 수 있다.

void usage(const char \*reason);

nsh 터미널에서 commander 명령 사용법 출력

void control\_status\_leds(vehicle\_status\_s \*status\_local, const actuator\_armed\_s \*actuator\_armed, bool changed,  
 battery\_status\_s \*battery\_local, const cpuload\_s \*cpuload\_local);

기체의 상태에 따라 상태 LED를 제어.

void get\_circuit\_breaker\_params();

서킷 브레이커에 관한 파라미터 값을 읽어 온다. 서킷 브레이커를 사용하면, 센서나 입력신호에 대한 체크 하지 않고, 사전비행 체크(preflight check) 또는 비행 중 체크해야하는 내용을 체크하지 않고 통과할 수 있다.

void set\_control\_mode();

모드 전환. 선택한 제어모드에 따라서 관련 상태 플래그를 수정한다.

bool stabilization\_required();

현재 기체 상태를 읽어, stabilizationd이 요구되는지 체크한다.

void print\_reject\_mode(const char \*msg);

모드 전환시. reject(거절)하는 이유를 로그로 남기고, 소리를 낸다.

void print\_reject\_arm(const char \*msg);

arming 할때. reject(거절)하는 이유를 로그로 남기고, 소리를 낸다.

void print\_status();

기체의 타입, USB연결 여부, 파워 상태, 위도, 경도, 고도, 홈 위치, 데이터 링크 상태, 네베게이션 상태, 암상태 등을 알려준다.

transition\_result\_t arm\_disarm(bool arm, orb\_advert\_t \*mavlink\_log\_pub, const char \*armedBy);

arming 또는 disarm 명령을 내리고 그 결과를 리턴한다.

void \*commander\_low\_prio\_loop(void \*arg);

센서 캘리브레이션 또는 파라미터 로드와 같은 우선순위 낮은 테스크를 실행하는 루프

static void answer\_command(const vehicle\_command\_s &cmd, unsigned result, orb\_advert\_t &command\_ack\_pub);

기체에 명령을 내렸을때 수행 결과 여부를 알려준다.

### Commander 클래스

위치: [class Commander](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/modules/commander/Commander.hpp#L69)

Commander::main()

Commander 클래스의 메인 함수.

Commander::handle\_command()

기체에 내려진 명령(REPOSITION, SET MODE, ARM DISARM, FLIGHT TERMINATION, SET HOME, GUIDED ENABLE, RETURN TO LAND, TAKE OFF, LAND, PRECISION LAND, MISSION START )을 수행하고 그 결과를 리턴

Commander::run()

Commander 앱의 메인 쓰레드.

## 실행 순서대로 읽기

Commander 클래스는 ModuleBase 템플릿 클래스를 상속하여 구현한다. 그래서 Commander 클래스 실행 순서는 ModuleBase 템플릿 클래스를 참조하여 실행한다.

참고:

* ModuleBase 클래스 Firmware/src/platforms/px4\_module.h Firmware/src/templates/module/module.cpp
* Commander 클래스 Firmware/src/modules/commander/Commander.hpp Firmware/src/modules/commander/commander.cpp

1. [commander\_main()](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/modules/commander/commander.cpp#L266)함수
2. [Commander::main()](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/modules/commander/commander.cpp#L283) 실행, [ModuleBase::main()](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/platforms/px4_module.h#L117), [ModuleBase::start\_command\_base()](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/platforms/px4_module.h#L181)
3. [Commander::task\_spawn()](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/modules/commander/commander.cpp#L4001) 실행, [ModuleBase::run\_trampoline()](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/platforms/px4_module.h#L152) 실행
4. [Commander::run()](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/modules/commander/commander.cpp#L1157) 초기화

4-1. [Parameter setup](https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/modules/commander/commander.cpp#L1168) 4-2.

1. <https://github.com/PX4/Firmware/blob/3293fe47f9d33fa83fdac2061695f1d20a2e09fb/src/modules/commander/commander.cpp#L1482><Commander::run() while()> 내부 5-1 파라미터 업데이트 5-2 파워버튼 핸들링 5-3 RC 컨트롤러 토픽 업데이트 5-4 Offboard 토픽 업데이트 및 타임아웃 설정 5-5 텔레메트리 상태 업데이트 5-6 시스템 파워 상태 업데이트

### Commander::run()

void  
Commander::run()  
{  
 bool sensor\_fail\_tune\_played = false;  
 bool arm\_tune\_played = false;  
 bool was\_landed = true;  
 bool was\_falling = false;  
 bool was\_armed = false;  
  
 // 센서 초기화 여부 확인 플래그  
 status\_flags.condition\_system\_sensors\_initialized = true;  
  
 // 파라미터 값 로드  
 param\_t \_param\_sys\_type = param\_find("MAV\_TYPE");  
 param\_t \_param\_system\_id = param\_find("MAV\_SYS\_ID");  
 param\_t \_param\_component\_id = param\_find("MAV\_COMP\_ID");  
 param\_t \_param\_enable\_datalink\_loss = param\_find("NAV\_DLL\_ACT");  
 param\_t \_param\_offboard\_loss\_act = param\_find("COM\_OBL\_ACT");  
 param\_t \_param\_offboard\_loss\_rc\_act = param\_find("COM\_OBL\_RC\_ACT");  
 param\_t \_param\_enable\_rc\_loss = param\_find("NAV\_RCL\_ACT");  
 param\_t \_param\_datalink\_loss\_timeout = param\_find("COM\_DL\_LOSS\_T");  
 param\_t \_param\_highlatencydatalink\_loss\_timeout = param\_find("COM\_HLDL\_LOSS\_T");  
 param\_t \_param\_rc\_loss\_timeout = param\_find("COM\_RC\_LOSS\_T");  
 param\_t \_param\_datalink\_regain\_timeout = param\_find("COM\_DL\_REG\_T");  
 param\_t \_param\_highlatencydatalink\_regain\_timeout = param\_find("COM\_HLDL\_REG\_T");  
 param\_t \_param\_ef\_throttle\_thres = param\_find("COM\_EF\_THROT");  
 param\_t \_param\_ef\_current2throttle\_thres = param\_find("COM\_EF\_C2T");  
 param\_t \_param\_ef\_time\_thres = param\_find("COM\_EF\_TIME");  
 param\_t \_param\_rc\_in\_off = param\_find("COM\_RC\_IN\_MODE");  
 param\_t \_param\_rc\_arm\_hyst = param\_find("COM\_RC\_ARM\_HYST");  
 param\_t \_param\_min\_stick\_change = param\_find("COM\_RC\_STICK\_OV");  
 param\_t \_param\_geofence\_action = param\_find("GF\_ACTION");  
 param\_t \_param\_disarm\_land = param\_find("COM\_DISARM\_LAND");  
 param\_t \_param\_low\_bat\_act = param\_find("COM\_LOW\_BAT\_ACT");  
 param\_t \_param\_offboard\_loss\_timeout = param\_find("COM\_OF\_LOSS\_T");  
 param\_t \_param\_arm\_without\_gps = param\_find("COM\_ARM\_WO\_GPS");  
 param\_t \_param\_arm\_switch\_is\_button = param\_find("COM\_ARM\_SWISBTN");  
 param\_t \_param\_rc\_override = param\_find("COM\_RC\_OVERRIDE");  
 param\_t \_param\_arm\_mission\_required = param\_find("COM\_ARM\_MIS\_REQ");  
 param\_t \_param\_flight\_uuid = param\_find("COM\_FLIGHT\_UUID");  
 param\_t \_param\_takeoff\_finished\_action = param\_find("COM\_TAKEOFF\_ACT");  
  
 param\_t \_param\_fmode\_1 = param\_find("COM\_FLTMODE1");  
 param\_t \_param\_fmode\_2 = param\_find("COM\_FLTMODE2");  
 param\_t \_param\_fmode\_3 = param\_find("COM\_FLTMODE3");  
 param\_t \_param\_fmode\_4 = param\_find("COM\_FLTMODE4");  
 param\_t \_param\_fmode\_5 = param\_find("COM\_FLTMODE5");  
 param\_t \_param\_fmode\_6 = param\_find("COM\_FLTMODE6");  
  
 /\* failsafe response to loss of navigation accuracy \*/  
 param\_t \_param\_posctl\_nav\_loss\_act = param\_find("COM\_POSCTL\_NAVL");  
  
  
 // 파라미터 또는 캘리브레이션 관련 쓰레드.  
 pthread\_t commander\_low\_prio\_thread;  
  
 // 초기화  
 if (led\_init() != OK) {  
 PX4\_WARN("LED init failed");  
 }  
  
 if (buzzer\_init() != OK) {  
 PX4\_WARN("Buzzer init failed");  
 }  
  
 // 파워 버튼 상태에 관한 토픽 구독  
 int power\_button\_state\_sub = orb\_subscribe(ORB\_ID(power\_button\_state));  
 {  
 // we need to do an initial publication to make sure uORB allocates the buffer, which cannot happen  
 // in IRQ context.  
 power\_button\_state\_s button\_state;  
 button\_state.timestamp = 0;  
 button\_state.event = 0xff;  
 power\_button\_state\_pub = orb\_advertise(ORB\_ID(power\_button\_state), &button\_state);  
 orb\_copy(ORB\_ID(power\_button\_state), power\_button\_state\_sub, &button\_state);  
 }  
  
 // 파워상태를 알람하는 콜백 함수 등록  
 if (board\_register\_power\_state\_notification\_cb(power\_button\_state\_notification\_cb) != 0) {  
 PX4\_ERR("Failed to register power notification callback");  
 }  
  
 // RC 입력 플래그 초기.  
 // We want to accept RC inputs as default  
 status\_flags.rc\_input\_blocked = false;  
 status.rc\_input\_mode = vehicle\_status\_s::RC\_IN\_MODE\_DEFAULT;  
 internal\_state.main\_state = commander\_state\_s::MAIN\_STATE\_MANUAL;  
 internal\_state.timestamp = hrt\_absolute\_time();  
 status.nav\_state = vehicle\_status\_s::NAVIGATION\_STATE\_MANUAL;  
 status.arming\_state = vehicle\_status\_s::ARMING\_STATE\_INIT;  
  
 status.failsafe = false;  
  
 // Offboard 신호 한번 이상 받았는지, RC신호 한번 이상 받았는지 체크 플래그  
 /\* neither manual nor offboard control commands have been received \*/  
 status\_flags.offboard\_control\_signal\_found\_once = false;  
 status\_flags.rc\_signal\_found\_once = false;  
  
 // RC, Offboard, 데이터 텔레메트리 신호 잃은적 있는지 체크 플래그  
 /\* mark all signals lost as long as they haven't been found \*/  
 status.rc\_signal\_lost = true;  
 status\_flags.offboard\_control\_signal\_lost = true;  
 status.data\_link\_lost = true;  
 status\_flags.offboard\_control\_loss\_timeout = false;  
  
 status\_flags.condition\_system\_hotplug\_timeout = false;  
  
 status.timestamp = hrt\_absolute\_time();  
  
 status\_flags.condition\_power\_input\_valid = true;  
 status\_flags.usb\_connected = false;  
 status\_flags.rc\_calibration\_valid = true;  
  
 // 서킷 블레이커. 기본으로 모두 false! circuit breaker가 false가 되면 연결 여부를 체크한다.  
 // power check는 power 커넥터로 부터 전원 들어오는지 체크  
 // airspd check는 airspeed 센서 연결 여부 체크  
 // engine failure check 는 RC 엔진 연결 여부 체크  
 // gps failure check 는 gps 에러 여부 체크  
 status\_flags.circuit\_breaker\_engaged\_power\_check = false;  
 status\_flags.circuit\_breaker\_engaged\_airspd\_check = false;  
 status\_flags.circuit\_breaker\_engaged\_enginefailure\_check = false;  
 status\_flags.circuit\_breaker\_engaged\_gpsfailure\_check = false;  
 get\_circuit\_breaker\_params();  
  
 // 위치, 속도가 유효한지 체크 하는 플래그.  
 status\_flags.condition\_global\_position\_valid = false;  
 status\_flags.condition\_local\_position\_valid = false;  
 status\_flags.condition\_local\_velocity\_valid = false;  
 status\_flags.condition\_local\_altitude\_valid = false;  
  
 // 기체 상태 vehicle\_status 관한 토픽 초기화  
 status\_pub = orb\_advertise(ORB\_ID(vehicle\_status), &status);  
  
 if (status\_pub == nullptr) {  
 warnx("ERROR: orb\_advertise for topic vehicle\_status failed (uorb app running?).\n");  
 warnx("exiting.");  
 px4\_task\_exit(PX4\_ERROR);  
 }  
  
 // arming 관련된 토픽 초기화  
 memset(&armed, 0, sizeof(armed));  
 /\* armed topic \*/  
 orb\_advert\_t armed\_pub = orb\_advertise(ORB\_ID(actuator\_armed), &armed);  
 hrt\_abstime last\_disarmed\_timestamp = 0;  
  
 // 기체 컨트롤 관련된 토픽 초기화  
 memset(&control\_mode, 0, sizeof(control\_mode));  
 orb\_advert\_t control\_mode\_pub = orb\_advertise(ORB\_ID(vehicle\_control\_mode), &control\_mode);  
  
 // 홈 위치 관련된 토픽 초기화  
 orb\_advert\_t home\_pub = nullptr;  
 memset(&\_home, 0, sizeof(\_home));  
  
 // 명령 ACK 관련된 토픽 초기화  
 orb\_advert\_t command\_ack\_pub = nullptr;  
 orb\_advert\_t commander\_state\_pub = nullptr;  
 orb\_advert\_t vehicle\_status\_flags\_pub = nullptr;  
  
 // 미션 초기화  
 /\* init mission state, do it here to allow navigator to use stored mission even if mavlink failed to start \*/  
 mission\_init();  
  
 /\* Start monitoring loop \*/  
 unsigned counter = 0;  
 int stick\_off\_counter = 0;  
 int stick\_on\_counter = 0;  
  
 bool low\_battery\_voltage\_actions\_done = false;  
 bool critical\_battery\_voltage\_actions\_done = false;  
 bool emergency\_battery\_voltage\_actions\_done = false;  
 bool dangerous\_battery\_level\_requests\_poweroff = false;  
  
 bool status\_changed = true;  
 bool param\_init\_forced = true;  
  
 bool updated = false;  
  
 // safety 토픽 구독  
 int safety\_sub = orb\_subscribe(ORB\_ID(safety));  
 memset(&safety, 0, sizeof(safety));  
 safety.safety\_switch\_available = false;  
 safety.safety\_off = false;  
  
 // geofence 결과 토픽 구독  
 int geofence\_result\_sub = orb\_subscribe(ORB\_ID(geofence\_result));  
 struct geofence\_result\_s geofence\_result;  
 memset(&geofence\_result, 0, sizeof(geofence\_result));  
  
 // manual control setpoint 토픽 구독. RC입력에 관한 토픽.  
 int sp\_man\_sub = orb\_subscribe(ORB\_ID(manual\_control\_setpoint));  
 memset(&sp\_man, 0, sizeof(sp\_man));  
  
 // offboard control 토픽 구독.  
 int offboard\_control\_mode\_sub = orb\_subscribe(ORB\_ID(offboard\_control\_mode));  
 memset(&offboard\_control\_mode, 0, sizeof(offboard\_control\_mode));  
  
 // landing 여부 판단하는 토픽 구독  
 int land\_detector\_sub = orb\_subscribe(ORB\_ID(vehicle\_land\_detected));  
 land\_detector.landed = true;  
  
 // mavlink또는 rc 도는 navigator에 의해 내려진 명령을 읽을 수 있는 토픽 구독  
 /\* Subscribe to command topic \*/  
 int cmd\_sub = orb\_subscribe(ORB\_ID(vehicle\_command));  
  
 // 파라미터 변경 여부를 알려주는 토픽 구독  
 int param\_changed\_sub = orb\_subscribe(ORB\_ID(parameter\_update));  
  
 // 배터리 상태 토픽 구독  
 /\* Subscribe to battery topic \*/  
 int battery\_sub = orb\_subscribe(ORB\_ID(battery\_status));  
 memset(&battery, 0, sizeof(battery));  
  
 // GPS, RC, 텔레메트리 등의 시스템 상태에 관한 토픽 구독  
 /\* Subscribe to subsystem info topic \*/  
 int subsys\_sub = orb\_subscribe(ORB\_ID(subsystem\_info));  
 struct subsystem\_info\_s info;  
 memset(&info, 0, sizeof(info));  
  
 // 시스템 파워(USB, Battery, Servo Rail) 에 관한 토픽 구독  
 int system\_power\_sub = orb\_subscribe(ORB\_ID(system\_power));  
  
 // 모터 제어 정보에 관한 토픽 구독  
 int actuator\_controls\_sub = orb\_subscribe(ORB\_ID\_VEHICLE\_ATTITUDE\_CONTROLS);  
  
 /\* Subscribe to vtol vehicle status topic \*/  
 int vtol\_vehicle\_status\_sub = orb\_subscribe(ORB\_ID(vtol\_vehicle\_status));  
 //struct vtol\_vehicle\_status\_s vtol\_status;  
 memset(&vtol\_status, 0, sizeof(vtol\_status));  
 vtol\_status.vtol\_in\_rw\_mode = true; //default for vtol is rotary wing  
  
 // Estimator 상태 정보 구독  
 int estimator\_status\_sub = orb\_subscribe(ORB\_ID(estimator\_status));  
 struct estimator\_status\_s estimator\_status;  
  
 // 이륙 후 navigator failure 상태 체크  
 /\* class variables used to check for navigation failure after takeoff \*/  
 hrt\_abstime time\_at\_takeoff = 0; // last time we were on the ground  
 hrt\_abstime time\_last\_innov\_pass = 0; // last time velocity innovations passed  
 bool nav\_test\_passed = false; // true if the post takeoff navigation test has passed  
 bool nav\_test\_failed = false; // true if the post takeoff navigation test has failed  
  
 int cpuload\_sub = orb\_subscribe(ORB\_ID(cpuload));  
 memset(&cpuload, 0, sizeof(cpuload));  
  
 control\_status\_leds(&status, &armed, true, &battery, &cpuload);  
  
 thread\_running = true;  
  
 /\* update vehicle status to find out vehicle type (required for preflight checks) \*/  
 int32\_t system\_type;  
 param\_get(\_param\_sys\_type, &system\_type); // get system type  
 status.system\_type = (uint8\_t)system\_type;  
 status.is\_rotary\_wing = is\_rotary\_wing(&status) || is\_vtol(&status);  
 status.is\_vtol = is\_vtol(&status);  
  
 commander\_boot\_timestamp = hrt\_absolute\_time();  
  
 // initially set to failed  
 \_last\_lpos\_fail\_time\_us = commander\_boot\_timestamp;  
 \_last\_gpos\_fail\_time\_us = commander\_boot\_timestamp;  
 \_last\_lvel\_fail\_time\_us = commander\_boot\_timestamp;  
  
 // 사전 비행 체크  
 int32\_t rc\_in\_off = 0;  
  
 param\_get(\_param\_rc\_in\_off, &rc\_in\_off);  
  
 int32\_t arm\_switch\_is\_button = 0;  
 param\_get(\_param\_arm\_switch\_is\_button, &arm\_switch\_is\_button);  
  
 int32\_t arm\_without\_gps\_param = 0;  
 param\_get(\_param\_arm\_without\_gps, &arm\_without\_gps\_param);  
 arm\_requirements = (arm\_without\_gps\_param == 1) ? ARM\_REQ\_NONE : ARM\_REQ\_GPS\_BIT;  
  
 int32\_t arm\_mission\_required\_param = 0;  
 param\_get(\_param\_arm\_mission\_required, &arm\_mission\_required\_param);  
 arm\_requirements |= (arm\_mission\_required\_param & (ARM\_REQ\_MISSION\_BIT | ARM\_REQ\_ARM\_AUTH\_BIT));  
  
 status.rc\_input\_mode = rc\_in\_off;  
  
 // user adjustable duration required to assert arm/disarm via throttle/rudder stick  
 int32\_t rc\_arm\_hyst = 100;  
 param\_get(\_param\_rc\_arm\_hyst, &rc\_arm\_hyst);  
 rc\_arm\_hyst \*= COMMANDER\_MONITORING\_LOOPSPERMSEC;  
  
 int32\_t datalink\_loss\_act = 0;  
 int32\_t rc\_loss\_act = 0;  
 int32\_t datalink\_loss\_timeout = 10;  
 int32\_t highlatencydatalink\_loss\_timeout = 120;  
 float rc\_loss\_timeout = 0.5;  
 int32\_t datalink\_regain\_timeout = 0;  
 int32\_t highlatencydatalink\_regain\_timeout = 0;  
 float offboard\_loss\_timeout = 0.0f;  
 int32\_t offboard\_loss\_act = 0;  
 int32\_t offboard\_loss\_rc\_act = 0;  
 int32\_t posctl\_nav\_loss\_act = 0;  
  
 int32\_t geofence\_action = 0;  
  
 int32\_t flight\_uuid = 0;  
  
 // AUTO MODE(MISSION, HOLD 모드)에서 RC 스틱이 들어왔을때 이전 모드(position mode)로 돌아가는지 결정  
 // rc\_override 가 0 이면, AUTO MODE에서 RC 스틱 입력이 들어와도 계속 AUTO MODE를 유지한다.  
 /\* RC override auto modes \*/  
 int32\_t rc\_override = 0;  
  
 int32\_t takeoff\_complete\_act = 0;  
  
 /\* Thresholds for engine failure detection \*/  
 float ef\_throttle\_thres = 1.0f;  
 float ef\_current2throttle\_thres = 0.0f;  
 float ef\_time\_thres = 1000.0f;  
 uint64\_t timestamp\_engine\_healthy = 0; /\*\*< absolute time when engine was healty \*/  
  
 int32\_t disarm\_when\_landed = 0;  
 int32\_t low\_bat\_action = 0;  
  
 // commander의 상태머신 변경여부 플래그  
 /\* check which state machines for changes, clear "changed" flag \*/  
 bool main\_state\_changed = false;  
 bool failsafe\_old = false;  
  
 bool have\_taken\_off\_since\_arming = false;  
  
 // 사전 비행 체크, 캘리브레이션 관련된 테스크 초기화  
 pthread\_attr\_t commander\_low\_prio\_attr;  
 pthread\_attr\_init(&commander\_low\_prio\_attr);  
 pthread\_attr\_setstacksize(&commander\_low\_prio\_attr, PX4\_STACK\_ADJUSTED(3000));  
  
#ifndef \_\_PX4\_QURT  
 // This is not supported by QURT (yet).  
 struct sched\_param param;  
 (void)pthread\_attr\_getschedparam(&commander\_low\_prio\_attr, &param);  
  
 /\* low priority \*/  
 param.sched\_priority = SCHED\_PRIORITY\_DEFAULT - 50;  
 (void)pthread\_attr\_setschedparam(&commander\_low\_prio\_attr, &param);  
#endif  
  
 pthread\_create(&commander\_low\_prio\_thread, &commander\_low\_prio\_attr, commander\_low\_prio\_loop, nullptr);  
 pthread\_attr\_destroy(&commander\_low\_prio\_attr);  
  
 arm\_auth\_init(&mavlink\_log\_pub, &status.system\_id);  
  
 while (!should\_exit()) {  
  
 transition\_result\_t arming\_ret = TRANSITION\_NOT\_CHANGED;  
  
 // 파라미터 업데이트  
 bool params\_updated = false;  
 orb\_check(param\_changed\_sub, &params\_updated);  
  
 if (params\_updated || param\_init\_forced) {  
  
 // 파라미터 변경 여부 체크  
 struct parameter\_update\_s param\_changed;  
 orb\_copy(ORB\_ID(parameter\_update), param\_changed\_sub, &param\_changed);  
  
 updateParams();  
  
 // 파라미터 업데이트  
 if (!armed.armed) {  
 if (param\_get(\_param\_sys\_type, &system\_type) != OK) {  
 PX4\_ERR("failed getting new system type");  
  
 } else {  
 status.system\_type = (uint8\_t)system\_type;  
 }  
  
 /\* disable manual override for all systems that rely on electronic stabilization \*/  
 if (is\_rotary\_wing(&status) || (is\_vtol(&status) && vtol\_status.vtol\_in\_rw\_mode)) {  
 status.is\_rotary\_wing = true;  
  
 } else {  
 status.is\_rotary\_wing = false;  
 }  
  
 /\* set vehicle\_status.is\_vtol flag \*/  
 status.is\_vtol = is\_vtol(&status);  
  
 /\* check and update system / component ID \*/  
 int32\_t sys\_id = 0;  
 param\_get(\_param\_system\_id, &sys\_id);  
 status.system\_id = sys\_id;  
  
 int32\_t comp\_id = 0;  
 param\_get(\_param\_component\_id, &comp\_id);  
 status.component\_id = comp\_id;  
  
 get\_circuit\_breaker\_params();  
  
 status\_changed = true;  
 }  
  
 // 위험(safety) 관련 파라미터 값 로드  
 param\_get(\_param\_enable\_datalink\_loss, &datalink\_loss\_act);  
 param\_get(\_param\_enable\_rc\_loss, &rc\_loss\_act);  
 param\_get(\_param\_datalink\_loss\_timeout, &datalink\_loss\_timeout);  
 param\_get(\_param\_highlatencydatalink\_loss\_timeout, &highlatencydatalink\_loss\_timeout);  
 param\_get(\_param\_rc\_loss\_timeout, &rc\_loss\_timeout);  
 param\_get(\_param\_rc\_in\_off, &rc\_in\_off);  
 status.rc\_input\_mode = rc\_in\_off;  
 param\_get(\_param\_rc\_arm\_hyst, &rc\_arm\_hyst);  
 param\_get(\_param\_min\_stick\_change, &min\_stick\_change);  
 param\_get(\_param\_rc\_override, &rc\_override);  
 // percentage (\* 0.01) needs to be doubled because RC total interval is 2, not 1  
 min\_stick\_change \*= 0.02f;  
 rc\_arm\_hyst \*= COMMANDER\_MONITORING\_LOOPSPERMSEC;  
 param\_get(\_param\_datalink\_regain\_timeout, &datalink\_regain\_timeout);  
 param\_get(\_param\_highlatencydatalink\_regain\_timeout, &highlatencydatalink\_regain\_timeout);  
 param\_get(\_param\_ef\_throttle\_thres, &ef\_throttle\_thres);  
 param\_get(\_param\_ef\_current2throttle\_thres, &ef\_current2throttle\_thres);  
 param\_get(\_param\_ef\_time\_thres, &ef\_time\_thres);  
 param\_get(\_param\_geofence\_action, &geofence\_action);  
 param\_get(\_param\_disarm\_land, &disarm\_when\_landed);  
 param\_get(\_param\_flight\_uuid, &flight\_uuid);  
  
 // If we update parameters the first time  
 // make sure the hysteresis time gets set.  
 // After that it will be set in the main state  
 // machine based on the arming state.  
 if (param\_init\_forced) {  
 auto\_disarm\_hysteresis.set\_hysteresis\_time\_from(false, disarm\_when\_landed \* 1\_s);  
 }  
  
 param\_get(\_param\_low\_bat\_act, &low\_bat\_action);  
 param\_get(\_param\_offboard\_loss\_timeout, &offboard\_loss\_timeout);  
 param\_get(\_param\_offboard\_loss\_act, &offboard\_loss\_act);  
 param\_get(\_param\_offboard\_loss\_rc\_act, &offboard\_loss\_rc\_act);  
 param\_get(\_param\_arm\_switch\_is\_button, &arm\_switch\_is\_button);  
  
 param\_get(\_param\_arm\_without\_gps, &arm\_without\_gps\_param);  
 arm\_requirements = (arm\_without\_gps\_param == 1) ? ARM\_REQ\_NONE : ARM\_REQ\_GPS\_BIT;  
 param\_get(\_param\_arm\_mission\_required, &arm\_mission\_required\_param);  
 arm\_requirements |= (arm\_mission\_required\_param & (ARM\_REQ\_MISSION\_BIT | ARM\_REQ\_ARM\_AUTH\_BIT));  
  
 /\* flight mode slots \*/  
 param\_get(\_param\_fmode\_1, &\_flight\_mode\_slots[0]);  
 param\_get(\_param\_fmode\_2, &\_flight\_mode\_slots[1]);  
 param\_get(\_param\_fmode\_3, &\_flight\_mode\_slots[2]);  
 param\_get(\_param\_fmode\_4, &\_flight\_mode\_slots[3]);  
 param\_get(\_param\_fmode\_5, &\_flight\_mode\_slots[4]);  
 param\_get(\_param\_fmode\_6, &\_flight\_mode\_slots[5]);  
  
 /\* failsafe response to loss of navigation accuracy \*/  
 param\_get(\_param\_posctl\_nav\_loss\_act, &posctl\_nav\_loss\_act);  
  
 param\_get(\_param\_takeoff\_finished\_action, &takeoff\_complete\_act);  
  
 param\_init\_forced = false;  
 }  
  
 // 파워 버튼 토픽 변경 여부 체크  
 orb\_check(power\_button\_state\_sub, &updated);  
  
 if (updated) {  
 power\_button\_state\_s button\_state;  
 orb\_copy(ORB\_ID(power\_button\_state), power\_button\_state\_sub, &button\_state);  
  
 if (button\_state.event == power\_button\_state\_s::PWR\_BUTTON\_STATE\_REQUEST\_SHUTDOWN) {  
 px4\_shutdown\_request(false, false);  
 }  
 }  
  
 // RC(manual control setpoint) 변경 여부 체크  
 orb\_check(sp\_man\_sub, &updated);  
  
 if (updated) {  
 orb\_copy(ORB\_ID(manual\_control\_setpoint), sp\_man\_sub, &sp\_man);  
 }  
 // Offboard 제어 모드 인지 체크  
 orb\_check(offboard\_control\_mode\_sub, &updated);  
  
 if (updated) {  
 orb\_copy(ORB\_ID(offboard\_control\_mode), offboard\_control\_mode\_sub, &offboard\_control\_mode);  
 }  
  
 if (offboard\_control\_mode.timestamp != 0 &&  
 offboard\_control\_mode.timestamp + OFFBOARD\_TIMEOUT > hrt\_absolute\_time()) {  
 if (status\_flags.offboard\_control\_signal\_lost) {  
 status\_flags.offboard\_control\_signal\_lost = false;  
 status\_flags.offboard\_control\_loss\_timeout = false;  
 status\_changed = true;  
 }  
  
 } else {  
 if (!status\_flags.offboard\_control\_signal\_lost) {  
 status\_flags.offboard\_control\_signal\_lost = true;  
 status\_changed = true;  
 }  
  
 /\* check timer if offboard was there but now lost \*/  
 if (!status\_flags.offboard\_control\_loss\_timeout && offboard\_control\_mode.timestamp != 0) {  
 if (offboard\_loss\_timeout < FLT\_EPSILON) {  
 /\* execute loss action immediately \*/  
 status\_flags.offboard\_control\_loss\_timeout = true;  
  
 } else {  
 /\* wait for timeout if set \*/  
 status\_flags.offboard\_control\_loss\_timeout = offboard\_control\_mode.timestamp +  
 OFFBOARD\_TIMEOUT + offboard\_loss\_timeout \* 1e6f < hrt\_absolute\_time();  
 }  
  
 if (status\_flags.offboard\_control\_loss\_timeout) {  
 status\_changed = true;  
 }  
 }  
 }  
  
 // 텔레메트리 상태 루프.  
 poll\_telemetry\_status();  
  
 orb\_check(system\_power\_sub, &updated);  
  
 if (updated) {  
 system\_power\_s system\_power = {};  
 orb\_copy(ORB\_ID(system\_power), system\_power\_sub, &system\_power);  
  
 if (hrt\_elapsed\_time(&system\_power.timestamp) < 200\_ms) {  
 if (system\_power.servo\_valid &&  
 !system\_power.brick\_valid &&  
 !system\_power.usb\_connected) {  
 /\* flying only on servo rail, this is unsafe \*/  
 status\_flags.condition\_power\_input\_valid = false;  
  
 } else {  
 status\_flags.condition\_power\_input\_valid = true;  
 }  
  
 /\* if the USB hardware connection went away, reboot \*/  
 if (status\_flags.usb\_connected && !system\_power.usb\_connected) {  
 /\*  
 \* apparently the USB cable went away but we are still powered,  
 \* so lets reset to a classic non-usb state.  
 \*/  
 mavlink\_log\_critical(&mavlink\_log\_pub, "USB disconnected, rebooting.")  
 usleep(400000);  
 px4\_shutdown\_request(true, false);  
 }  
 }  
 }  
  
 // safety 토픽 변경 여부 체크  
 orb\_check(safety\_sub, &updated);  
  
 if (updated) {  
 bool previous\_safety\_off = safety.safety\_off;  
  
 if (orb\_copy(ORB\_ID(safety), safety\_sub, &safety) == PX4\_OK) {  
  
 /\* disarm if safety is now on and still armed \*/  
 if (armed.armed && (status.hil\_state == vehicle\_status\_s::HIL\_STATE\_OFF)  
 && safety.safety\_switch\_available && !safety.safety\_off) {  
  
 if (TRANSITION\_CHANGED == arming\_state\_transition(&status, battery, safety, vehicle\_status\_s::ARMING\_STATE\_STANDBY,  
 &armed, true /\* fRunPreArmChecks \*/, &mavlink\_log\_pub,  
 &status\_flags, arm\_requirements, hrt\_elapsed\_time(&commander\_boot\_timestamp))  
 ) {  
 status\_changed = true;  
 }  
 }  
  
 // Notify the user if the status of the safety switch changes  
 if (safety.safety\_switch\_available && previous\_safety\_off != safety.safety\_off) {  
  
 if (safety.safety\_off) {  
 set\_tune(TONE\_NOTIFY\_POSITIVE\_TUNE);  
  
 } else {  
 tune\_neutral(true);  
 }  
  
 status\_changed = true;  
 }  
 }  
 }  
  
 // VTOL 기체 상태 변경 있는지 체크  
 orb\_check(vtol\_vehicle\_status\_sub, &updated);  
  
 if (updated) {  
 /\* vtol status changed \*/  
 orb\_copy(ORB\_ID(vtol\_vehicle\_status), vtol\_vehicle\_status\_sub, &vtol\_status);  
 status.vtol\_fw\_permanent\_stab = vtol\_status.fw\_permanent\_stab;  
  
 /\* Make sure that this is only adjusted if vehicle really is of type vtol \*/  
 if (is\_vtol(&status)) {  
  
 // Check if there has been any change while updating the flags  
 if (status.is\_rotary\_wing != vtol\_status.vtol\_in\_rw\_mode) {  
 status.is\_rotary\_wing = vtol\_status.vtol\_in\_rw\_mode;  
 status\_changed = true;  
 }  
  
 if (status.in\_transition\_mode != vtol\_status.vtol\_in\_trans\_mode) {  
 status.in\_transition\_mode = vtol\_status.vtol\_in\_trans\_mode;  
 status\_changed = true;  
 }  
  
 if (status.in\_transition\_to\_fw != vtol\_status.in\_transition\_to\_fw) {  
 status.in\_transition\_to\_fw = vtol\_status.in\_transition\_to\_fw;  
 status\_changed = true;  
 }  
  
 if (status\_flags.vtol\_transition\_failure != vtol\_status.vtol\_transition\_failsafe) {  
 status\_flags.vtol\_transition\_failure = vtol\_status.vtol\_transition\_failsafe;  
 status\_changed = true;  
 }  
  
 if (armed.soft\_stop != !status.is\_rotary\_wing) {  
 armed.soft\_stop = !status.is\_rotary\_wing;  
 status\_changed = true;  
 }  
 }  
 }  
  
 // 로컬 위치, 글로벌 위치(GPS) 변경 여부 체크하고, 변경되어 있으면 업데이트  
 \_local\_position\_sub.update();  
 \_global\_position\_sub.update();  
  
 // Set the allowable positon uncertainty based on combination of flight and estimator state  
 // When we are in a operator demanded position control mode and are solely reliant on optical flow, do not check position error becasue it will gradually increase throughout flight and the operator will compensate for the drift  
 bool reliant\_on\_opt\_flow = ((estimator\_status.control\_mode\_flags & (1 << estimator\_status\_s::CS\_OPT\_FLOW))  
 && !(estimator\_status.control\_mode\_flags & (1 << estimator\_status\_s::CS\_GPS))  
 && !(estimator\_status.control\_mode\_flags & (1 << estimator\_status\_s::CS\_EV\_POS)));  
 bool operator\_controlled\_position = (internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_POSCTL);  
 \_skip\_pos\_accuracy\_check = reliant\_on\_opt\_flow && operator\_controlled\_position;  
 if (\_skip\_pos\_accuracy\_check) {  
 \_eph\_threshold\_adj = INFINITY;  
 } else {  
 \_eph\_threshold\_adj = \_eph\_threshold.get();  
 }  
  
 // Check if quality checking of position accuracy and consistency is to be performed  
 const bool run\_quality\_checks = !status\_flags.circuit\_breaker\_engaged\_posfailure\_check;  
  
 /\* Check estimator status for signs of bad yaw induced post takeoff navigation failure  
 \* for a short time interval after takeoff. Fixed wing vehicles can recover using GPS heading,  
 \* but rotary wing vehicles cannot so the position and velocity validity needs to be latched  
 \* to false after failure to prevent flyaway crashes \*/  
 if (run\_quality\_checks && status.is\_rotary\_wing) {  
 bool estimator\_status\_updated = false;  
 orb\_check(estimator\_status\_sub, &estimator\_status\_updated);  
  
 if (estimator\_status\_updated) {  
 orb\_copy(ORB\_ID(estimator\_status), estimator\_status\_sub, &estimator\_status);  
  
 if (status.arming\_state == vehicle\_status\_s::ARMING\_STATE\_STANDBY) {  
 // reset flags and timer  
 time\_at\_takeoff = hrt\_absolute\_time();  
 nav\_test\_failed = false;  
 nav\_test\_passed = false;  
  
 } else if (land\_detector.landed) {  
 // record time of takeoff  
 time\_at\_takeoff = hrt\_absolute\_time();  
  
 } else {  
 // if nav status is unconfirmed, confirm yaw angle as passed after 30 seconds or achieving 5 m/s of speed  
 const bool sufficient\_time = (hrt\_elapsed\_time(&time\_at\_takeoff) > 30\_s);  
  
 const vehicle\_local\_position\_s &lpos = \_local\_position\_sub.get();  
 const bool sufficient\_speed = (lpos.vx \* lpos.vx + lpos.vy \* lpos.vy > 25.0f);  
  
 bool innovation\_pass = estimator\_status.vel\_test\_ratio < 1.0f && estimator\_status.pos\_test\_ratio < 1.0f;  
  
 if (!nav\_test\_failed) {  
 if (!nav\_test\_passed) {  
 // pass if sufficient time or speed  
 if (sufficient\_time || sufficient\_speed) {  
 nav\_test\_passed = true;  
 }  
  
 // record the last time the innovation check passed  
 if (innovation\_pass) {  
 time\_last\_innov\_pass = hrt\_absolute\_time();  
 }  
  
 // if the innovation test has failed continuously, declare the nav as failed  
 if (hrt\_elapsed\_time(&time\_last\_innov\_pass) > 1\_s) {  
 nav\_test\_failed = true;  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "CRITICAL NAVIGATION FAILURE - CHECK SENSOR CALIBRATION");  
 }  
 }  
 }  
 }  
 }  
 }  
  
 // 글로벌 위치 정확도 체크  
 /\* run global position accuracy checks \*/  
 // Check if quality checking of position accuracy and consistency is to be performed  
 if (run\_quality\_checks) {  
 if (nav\_test\_failed) {  
 status\_flags.condition\_global\_position\_valid = false;  
 status\_flags.condition\_local\_position\_valid = false;  
 status\_flags.condition\_local\_velocity\_valid = false;  
  
 } else {  
 if (!\_skip\_pos\_accuracy\_check) {  
 // use global position message to determine validity  
 const vehicle\_global\_position\_s&global\_position = \_global\_position\_sub.get();  
 check\_posvel\_validity(true, global\_position.eph, \_eph\_threshold\_adj, global\_position.timestamp, &\_last\_gpos\_fail\_time\_us, &\_gpos\_probation\_time\_us, &status\_flags.condition\_global\_position\_valid, &status\_changed);  
 }  
  
 // use local position message to determine validity  
 const vehicle\_local\_position\_s &local\_position = \_local\_position\_sub.get();  
 check\_posvel\_validity(local\_position.xy\_valid, local\_position.eph, \_eph\_threshold\_adj, local\_position.timestamp, &\_last\_lpos\_fail\_time\_us, &\_lpos\_probation\_time\_us, &status\_flags.condition\_local\_position\_valid, &status\_changed);  
 check\_posvel\_validity(local\_position.v\_xy\_valid, local\_position.evh, \_evh\_threshold.get(), local\_position.timestamp, &\_last\_lvel\_fail\_time\_us, &\_lvel\_probation\_time\_us, &status\_flags.condition\_local\_velocity\_valid, &status\_changed);  
 }  
 }  
  
 if((\_last\_condition\_global\_position\_valid != status\_flags.condition\_global\_position\_valid) && status\_flags.condition\_global\_position\_valid) {  
 // If global position state changed and is now valid, set respective health flags to true. For now also assume GPS is OK if global pos is OK, but not vice versa.  
 set\_health\_flags\_healthy(subsystem\_info\_s::SUBSYSTEM\_TYPE\_AHRS, true, status);  
 set\_health\_flags\_present\_healthy(subsystem\_info\_s::SUBSYSTEM\_TYPE\_GPS, true, true, status);  
 }  
  
 check\_valid(\_local\_position\_sub.get().timestamp, \_failsafe\_pos\_delay.get() \* 1\_s, \_local\_position\_sub.get().z\_valid, &(status\_flags.condition\_local\_altitude\_valid), &status\_changed);  
  
 // 착륙 여부 판단 토픽 변경여부 체크  
 orb\_check(land\_detector\_sub, &updated);  
  
 if (updated) {  
 orb\_copy(ORB\_ID(vehicle\_land\_detected), land\_detector\_sub, &land\_detector);  
  
 // Only take actions if armed  
 if (armed.armed) {  
 if (was\_landed != land\_detector.landed) {  
 if (land\_detector.landed) {  
 mavlink\_and\_console\_log\_info(&mavlink\_log\_pub, "Landing detected");  
  
 } else {  
 mavlink\_and\_console\_log\_info(&mavlink\_log\_pub, "Takeoff detected");  
 have\_taken\_off\_since\_arming = true;  
  
 // Set all position and velocity test probation durations to takeoff value  
 // This is a larger value to give the vehicle time to complete a failsafe landing  
 // if faulty sensors cause loss of navigation shortly after takeoff.  
 \_gpos\_probation\_time\_us = \_failsafe\_pos\_probation.get() \* 1\_s;  
 \_lpos\_probation\_time\_us = \_failsafe\_pos\_probation.get() \* 1\_s;  
 \_lvel\_probation\_time\_us = \_failsafe\_pos\_probation.get() \* 1\_s;  
 }  
 }  
  
 if (was\_falling != land\_detector.freefall) {  
 if (land\_detector.freefall) {  
 mavlink\_and\_console\_log\_info(&mavlink\_log\_pub, "Freefall detected");  
 }  
 }  
 }  
  
 was\_landed = land\_detector.landed;  
 was\_falling = land\_detector.freefall;  
 }  
  
 /\* Update hysteresis time. Use a time of factor 5 longer if we have not taken off yet. \*/  
 hrt\_abstime timeout\_time = disarm\_when\_landed \* 1\_s;  
  
 if (!have\_taken\_off\_since\_arming) {  
 timeout\_time \*= 5;  
 }  
  
 auto\_disarm\_hysteresis.set\_hysteresis\_time\_from(false, timeout\_time);  
  
 // Check for auto-disarm  
 if (armed.armed && land\_detector.landed && disarm\_when\_landed > 0) {  
 auto\_disarm\_hysteresis.set\_state\_and\_update(true);  
  
 } else {  
 auto\_disarm\_hysteresis.set\_state\_and\_update(false);  
 }  
  
 if (auto\_disarm\_hysteresis.get\_state()) {  
 arm\_disarm(false, &mavlink\_log\_pub, "auto disarm on land");  
 }  
  
 if (!warning\_action\_on) {  
 // store the last good main\_state when not in an navigation  
 // hold state  
 main\_state\_before\_rtl = internal\_state.main\_state;  
  
 } else if (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_AUTO\_RTL  
 && internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_AUTO\_LOITER  
 && internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_AUTO\_LAND) {  
 // reset flag again when we switched out of it  
 warning\_action\_on = false;  
 }  
  
 orb\_check(cpuload\_sub, &updated);  
  
 if (updated) {  
 orb\_copy(ORB\_ID(cpuload), cpuload\_sub, &cpuload);  
 }  
  
 // 배터리 상태 토픽 변경 여부 체크  
 orb\_check(battery\_sub, &updated);  
  
 if (updated) {  
 orb\_copy(ORB\_ID(battery\_status), battery\_sub, &battery);  
  
 /\* only consider battery voltage if system has been running 6s (usb most likely detected) and battery voltage is valid \*/  
 if ((hrt\_elapsed\_time(&commander\_boot\_timestamp) > 6\_s)  
 && battery.voltage\_filtered\_v > 2.0f \* FLT\_EPSILON) {  
  
 /\* if battery voltage is getting lower, warn using buzzer, etc. \*/  
 if (battery.warning == battery\_status\_s::BATTERY\_WARNING\_LOW &&  
 !low\_battery\_voltage\_actions\_done) {  
  
 low\_battery\_voltage\_actions\_done = true;  
  
 if (armed.armed) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "LOW BATTERY, RETURN TO LAND ADVISED");  
  
 } else {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "LOW BATTERY, TAKEOFF DISCOURAGED");  
 }  
  
 status\_changed = true;  
  
 } else if (battery.warning == battery\_status\_s::BATTERY\_WARNING\_CRITICAL &&  
 !critical\_battery\_voltage\_actions\_done) {  
  
 critical\_battery\_voltage\_actions\_done = true;  
  
 if (!armed.armed) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "CRITICAL BATTERY, SHUT SYSTEM DOWN");  
  
 } else {  
 if (low\_bat\_action == 1 || low\_bat\_action == 3) {  
 // let us send the critical message even if already in RTL  
 if (TRANSITION\_DENIED != main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_RTL, status\_flags, &internal\_state)) {  
 warning\_action\_on = true;  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "CRITICAL BATTERY, RETURNING TO LAND");  
  
 } else {  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "CRITICAL BATTERY, RTL FAILED");  
 }  
  
 } else if (low\_bat\_action == 2) {  
 if (TRANSITION\_DENIED != main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_LAND, status\_flags, &internal\_state)) {  
 warning\_action\_on = true;  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "CRITICAL BATTERY, LANDING AT CURRENT POSITION");  
  
 } else {  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "CRITICAL BATTERY, LANDING FAILED");  
 }  
  
 } else {  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "CRITICAL BATTERY, RETURN TO LAUNCH ADVISED!");  
 }  
 }  
  
 status\_changed = true;  
  
 } else if (battery.warning == battery\_status\_s::BATTERY\_WARNING\_EMERGENCY &&  
 !emergency\_battery\_voltage\_actions\_done) {  
  
 emergency\_battery\_voltage\_actions\_done = true;  
  
 if (!armed.armed) {  
 // Request shutdown at the end of the cycle. This allows  
 // the vehicle state to be published after emergency landing  
 dangerous\_battery\_level\_requests\_poweroff = true;  
 } else {  
 if (low\_bat\_action == 2 || low\_bat\_action == 3) {  
 if (TRANSITION\_CHANGED == main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_LAND, status\_flags, &internal\_state)) {  
 warning\_action\_on = true;  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "DANGEROUS BATTERY LEVEL, LANDING IMMEDIATELY");  
  
 } else {  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "DANGEROUS BATTERY LEVEL, LANDING FAILED");  
 }  
  
 } else {  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "DANGEROUS BATTERY LEVEL, LANDING ADVISED!");  
 }  
 }  
  
 status\_changed = true;  
 }  
  
 /\* End battery voltage check \*/  
 }  
 }  
  
 /\* update subsystem info which arrives from outside of commander\*/  
 do {  
 orb\_check(subsys\_sub, &updated);  
 if (updated) {  
 orb\_copy(ORB\_ID(subsystem\_info), subsys\_sub, &info);  
 set\_health\_flags(info.subsystem\_type, info.present, info.enabled, info.ok, status);  
 status\_changed = true;  
 }  
 } while(updated);  
  
 /\* If in INIT state, try to proceed to STANDBY state \*/  
 if (!status\_flags.condition\_calibration\_enabled && status.arming\_state == vehicle\_status\_s::ARMING\_STATE\_INIT) {  
  
 arming\_ret = arming\_state\_transition(&status, battery, safety, vehicle\_status\_s::ARMING\_STATE\_STANDBY, &armed,  
 true /\* fRunPreArmChecks \*/, &mavlink\_log\_pub, &status\_flags,  
 arm\_requirements, hrt\_elapsed\_time(&commander\_boot\_timestamp));  
  
 if (arming\_ret == TRANSITION\_DENIED) {  
 /\* do not complain if not allowed into standby \*/  
 arming\_ret = TRANSITION\_NOT\_CHANGED;  
 }  
 }  
  
 // 미션 결과 체크  
 const auto prev\_mission\_instance\_count = \_mission\_result\_sub.get().instance\_count;  
  
 if (\_mission\_result\_sub.update()) {  
 const mission\_result\_s &mission\_result = \_mission\_result\_sub.get();  
  
 // if mission\_result is valid for the current mission  
 const bool mission\_result\_ok = (mission\_result.timestamp > commander\_boot\_timestamp) && (mission\_result.instance\_count > 0);  
  
 status\_flags.condition\_auto\_mission\_available = mission\_result\_ok && mission\_result.valid;  
  
 if (mission\_result\_ok) {  
  
 if (status.mission\_failure != mission\_result.failure) {  
 status.mission\_failure = mission\_result.failure;  
 status\_changed = true;  
  
 if (status.mission\_failure) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "Mission cannot be completed");  
 }  
 }  
  
 /\* Only evaluate mission state if home is set \*/  
 if (status\_flags.condition\_home\_position\_valid &&  
 (prev\_mission\_instance\_count != mission\_result.instance\_count)) {  
  
 if (!status\_flags.condition\_auto\_mission\_available) {  
 /\* the mission is invalid \*/  
 tune\_mission\_fail(true);  
  
 } else if (mission\_result.warning) {  
 /\* the mission has a warning \*/  
 tune\_mission\_fail(true);  
  
 } else {  
 /\* the mission is valid \*/  
 tune\_mission\_ok(true);  
 }  
 }  
 }  
 }  
  
 // 지오펜스 결과 여부 체크  
 orb\_check(geofence\_result\_sub, &updated);  
  
 if (updated) {  
 orb\_copy(ORB\_ID(geofence\_result), geofence\_result\_sub, &geofence\_result);  
 }  
  
 // 지오펜스가 있으면 지오펜스 벗어났는지 체크하고, 비행 명령 수행  
 // Geofence actions  
 if (armed.armed && (geofence\_result.geofence\_action != geofence\_result\_s::GF\_ACTION\_NONE)) {  
  
 static bool geofence\_loiter\_on = false;  
 static bool geofence\_rtl\_on = false;  
  
 // check for geofence violation  
 if (geofence\_result.geofence\_violated) {  
 static hrt\_abstime last\_geofence\_violation = 0;  
 const hrt\_abstime geofence\_violation\_action\_interval = 10\_s;  
  
 if (hrt\_elapsed\_time(&last\_geofence\_violation) > geofence\_violation\_action\_interval) {  
  
 last\_geofence\_violation = hrt\_absolute\_time();  
  
 switch (geofence\_result.geofence\_action) {  
 case (geofence\_result\_s::GF\_ACTION\_NONE) : {  
 // do nothing  
 break;  
 }  
 case (geofence\_result\_s::GF\_ACTION\_WARN) : {  
 // do nothing, mavlink critical messages are sent by navigator  
 break;  
 }  
 case (geofence\_result\_s::GF\_ACTION\_LOITER) : {  
 if (TRANSITION\_CHANGED == main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_LOITER, status\_flags, &internal\_state)) {  
 geofence\_loiter\_on = true;  
 }  
  
 break;  
 }  
 case (geofence\_result\_s::GF\_ACTION\_RTL) : {  
 if (TRANSITION\_CHANGED == main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_RTL, status\_flags, &internal\_state)) {  
 geofence\_rtl\_on = true;  
 }  
  
 break;  
 }  
 case (geofence\_result\_s::GF\_ACTION\_TERMINATE) : {  
 warnx("Flight termination because of geofence");  
 mavlink\_log\_critical(&mavlink\_log\_pub, "Geofence violation: flight termination");  
 armed.force\_failsafe = true;  
 status\_changed = true;  
 break;  
 }  
 }  
 }  
 }  
  
 // reset if no longer in LOITER or if manually switched to LOITER  
 geofence\_loiter\_on = geofence\_loiter\_on  
 && (internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_AUTO\_LOITER)  
 && (sp\_man.loiter\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_OFF  
 || sp\_man.loiter\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_NONE);  
  
 // reset if no longer in RTL or if manually switched to RTL  
 geofence\_rtl\_on = geofence\_rtl\_on  
 && (internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_AUTO\_RTL)  
 && (sp\_man.return\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_OFF  
 || sp\_man.return\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_NONE);  
  
 warning\_action\_on = warning\_action\_on || (geofence\_loiter\_on || geofence\_rtl\_on);  
 }  
  
 // revert geofence failsafe transition if sticks are moved and we were previously in a manual mode  
 // but only if not in a low battery handling action  
 if (rc\_override != 0 && !critical\_battery\_voltage\_actions\_done && (warning\_action\_on &&  
 (main\_state\_before\_rtl == commander\_state\_s::MAIN\_STATE\_MANUAL ||  
 main\_state\_before\_rtl == commander\_state\_s::MAIN\_STATE\_ALTCTL ||  
 main\_state\_before\_rtl == commander\_state\_s::MAIN\_STATE\_POSCTL ||  
 main\_state\_before\_rtl == commander\_state\_s::MAIN\_STATE\_ACRO ||  
 main\_state\_before\_rtl == commander\_state\_s::MAIN\_STATE\_RATTITUDE ||  
 main\_state\_before\_rtl == commander\_state\_s::MAIN\_STATE\_STAB))) {  
  
 // transition to previous state if sticks are touched  
 if ((\_last\_sp\_man.timestamp != sp\_man.timestamp) &&  
 ((fabsf(sp\_man.x - \_last\_sp\_man.x) > min\_stick\_change) ||  
 (fabsf(sp\_man.y - \_last\_sp\_man.y) > min\_stick\_change) ||  
 (fabsf(sp\_man.z - \_last\_sp\_man.z) > min\_stick\_change) ||  
 (fabsf(sp\_man.r - \_last\_sp\_man.r) > min\_stick\_change))) {  
  
 // revert to position control in any case  
 main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_POSCTL, status\_flags, &internal\_state);  
 mavlink\_log\_critical(&mavlink\_log\_pub, "Autopilot off, returned control to pilot");  
 }  
 }  
  
 // abort landing or auto or loiter if sticks are moved significantly  
 // but only if not in a low battery handling action  
 if (rc\_override != 0 && !critical\_battery\_voltage\_actions\_done &&  
 (internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_AUTO\_LAND ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_AUTO\_MISSION ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_AUTO\_LOITER)) {  
 // transition to previous state if sticks are touched  
  
 if ((\_last\_sp\_man.timestamp != sp\_man.timestamp) &&  
 ((fabsf(sp\_man.x - \_last\_sp\_man.x) > min\_stick\_change) ||  
 (fabsf(sp\_man.y - \_last\_sp\_man.y) > min\_stick\_change) ||  
 (fabsf(sp\_man.z - \_last\_sp\_man.z) > min\_stick\_change) ||  
 (fabsf(sp\_man.r - \_last\_sp\_man.r) > min\_stick\_change))) {  
  
 // revert to position control in any case  
 main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_POSCTL, status\_flags, &internal\_state);  
 mavlink\_log\_critical(&mavlink\_log\_pub, "Autopilot off, returned control to pilot");  
 }  
 }  
  
 // 미션 비행 끝  
 /\* Check for mission flight termination \*/  
 if (armed.armed && \_mission\_result\_sub.get().flight\_termination &&  
 !status\_flags.circuit\_breaker\_flight\_termination\_disabled) {  
  
 armed.force\_failsafe = true;  
 status\_changed = true;  
 static bool flight\_termination\_printed = false;  
  
 if (!flight\_termination\_printed) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "Geofence violation: flight termination");  
 flight\_termination\_printed = true;  
 }  
  
 if (counter % (1000000 / COMMANDER\_MONITORING\_INTERVAL) == 0) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "Flight termination active");  
 }  
 }  
  
 // RC 입력 신호 체크  
 if (!status\_flags.rc\_input\_blocked && sp\_man.timestamp != 0 &&  
 (hrt\_elapsed\_time(&sp\_man.timestamp) < (rc\_loss\_timeout \* 1\_s))) {  
  
 /\* handle the case where RC signal was regained \*/  
 if (!status\_flags.rc\_signal\_found\_once) {  
 status\_flags.rc\_signal\_found\_once = true;  
 set\_health\_flags(subsystem\_info\_s::SUBSYSTEM\_TYPE\_RCRECEIVER, true, true, true && status\_flags.rc\_calibration\_valid, status);  
 status\_changed = true;  
  
 } else {  
 if (status.rc\_signal\_lost) {  
 mavlink\_log\_info(&mavlink\_log\_pub, "MANUAL CONTROL REGAINED after %llums", hrt\_elapsed\_time(&rc\_signal\_lost\_timestamp) / 1000);  
 set\_health\_flags(subsystem\_info\_s::SUBSYSTEM\_TYPE\_RCRECEIVER, true, true, true && status\_flags.rc\_calibration\_valid, status);  
 status\_changed = true;  
 }  
 }  
  
 status.rc\_signal\_lost = false;  
  
 const bool in\_armed\_state = (status.arming\_state == vehicle\_status\_s::ARMING\_STATE\_ARMED);  
 const bool arm\_switch\_or\_button\_mapped = sp\_man.arm\_switch != manual\_control\_setpoint\_s::SWITCH\_POS\_NONE;  
 const bool arm\_button\_pressed = arm\_switch\_is\_button == 1  
 && sp\_man.arm\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_ON;  
  
 /\* DISARM  
 \* check if left stick is in lower left position or arm button is pushed or arm switch has transition from arm to disarm  
 \* and we are in MANUAL, Rattitude, or AUTO\_READY mode or (ASSIST mode and landed)  
 \* do it only for rotary wings in manual mode or fixed wing if landed.  
 \* Disable stick-disarming if arming switch or button is mapped \*/  
 const bool stick\_in\_lower\_left = sp\_man.r < -STICK\_ON\_OFF\_LIMIT && sp\_man.z < 0.1f  
 && !arm\_switch\_or\_button\_mapped;  
 const bool arm\_switch\_to\_disarm\_transition = arm\_switch\_is\_button == 0 &&  
 \_last\_sp\_man\_arm\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_ON &&  
 sp\_man.arm\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_OFF;  
  
 if (in\_armed\_state &&  
 status.rc\_input\_mode != vehicle\_status\_s::RC\_IN\_MODE\_OFF &&  
 (status.is\_rotary\_wing || (!status.is\_rotary\_wing && land\_detector.landed)) &&  
 (stick\_in\_lower\_left || arm\_button\_pressed || arm\_switch\_to\_disarm\_transition)) {  
  
 if (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_MANUAL &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_ACRO &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_STAB &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_RATTITUDE &&  
 !land\_detector.landed) {  
 print\_reject\_arm("NOT DISARMING: Not in manual mode or landed yet.");  
  
 } else if ((stick\_off\_counter == rc\_arm\_hyst && stick\_on\_counter < rc\_arm\_hyst) || arm\_switch\_to\_disarm\_transition) {  
 arming\_ret = arming\_state\_transition(&status, battery, safety, vehicle\_status\_s::ARMING\_STATE\_STANDBY, &armed,  
 true /\* fRunPreArmChecks \*/,  
 &mavlink\_log\_pub, &status\_flags, arm\_requirements, hrt\_elapsed\_time(&commander\_boot\_timestamp));  
 }  
  
 stick\_off\_counter++;  
  
 } else if (!(arm\_switch\_is\_button == 1 && sp\_man.arm\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_ON)) {  
 /\* do not reset the counter when holding the arm button longer than needed \*/  
 stick\_off\_counter = 0;  
 }  
  
 /\* ARM  
 \* check if left stick is in lower right position or arm button is pushed or arm switch has transition from disarm to arm  
 \* and we're in MANUAL mode.  
 \* Disable stick-arming if arming switch or button is mapped \*/  
 const bool stick\_in\_lower\_right = sp\_man.r > STICK\_ON\_OFF\_LIMIT && sp\_man.z < 0.1f  
 && !arm\_switch\_or\_button\_mapped;  
 /\* allow a grace period for re-arming: preflight checks don't need to pass during that time,  
 \* for example for accidential in-air disarming \*/  
 const bool in\_arming\_grace\_period = last\_disarmed\_timestamp != 0 && hrt\_elapsed\_time(&last\_disarmed\_timestamp) < 5\_s;  
 const bool arm\_switch\_to\_arm\_transition = arm\_switch\_is\_button == 0 &&  
 \_last\_sp\_man\_arm\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_OFF &&  
 sp\_man.arm\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_ON &&  
 (sp\_man.z < 0.1f || in\_arming\_grace\_period);  
  
 if (!in\_armed\_state &&  
 status.rc\_input\_mode != vehicle\_status\_s::RC\_IN\_MODE\_OFF &&  
 (stick\_in\_lower\_right || arm\_button\_pressed || arm\_switch\_to\_arm\_transition)) {  
 if ((stick\_on\_counter == rc\_arm\_hyst && stick\_off\_counter < rc\_arm\_hyst) || arm\_switch\_to\_arm\_transition) {  
  
 /\* we check outside of the transition function here because the requirement  
 \* for being in manual mode only applies to manual arming actions.  
 \* the system can be armed in auto if armed via the GCS.  
 \*/  
  
 if ((internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_MANUAL)  
 && (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_ACRO)  
 && (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_STAB)  
 && (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_ALTCTL)  
 && (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_POSCTL)  
 && (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_RATTITUDE)  
 ) {  
 print\_reject\_arm("NOT ARMING: Switch to a manual mode first.");  
  
 } else if (!status\_flags.condition\_home\_position\_valid &&  
 geofence\_action == geofence\_result\_s::GF\_ACTION\_RTL) {  
 print\_reject\_arm("NOT ARMING: Geofence RTL requires valid home");  
  
 } else if (status.arming\_state == vehicle\_status\_s::ARMING\_STATE\_STANDBY) {  
 arming\_ret = arming\_state\_transition(&status, battery, safety, vehicle\_status\_s::ARMING\_STATE\_ARMED, &armed,  
 !in\_arming\_grace\_period /\* fRunPreArmChecks \*/,  
 &mavlink\_log\_pub, &status\_flags, arm\_requirements, hrt\_elapsed\_time(&commander\_boot\_timestamp));  
  
 if (arming\_ret != TRANSITION\_CHANGED) {  
 usleep(100000);  
 print\_reject\_arm("NOT ARMING: Preflight checks failed");  
 }  
 }  
 }  
  
 stick\_on\_counter++;  
  
 } else if (!(arm\_switch\_is\_button == 1 && sp\_man.arm\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_ON)) {  
 /\* do not reset the counter when holding the arm button longer than needed \*/  
 stick\_on\_counter = 0;  
 }  
  
 \_last\_sp\_man\_arm\_switch = sp\_man.arm\_switch;  
  
 if (arming\_ret == TRANSITION\_DENIED) {  
 /\*  
 \* the arming transition can be denied to a number of reasons:  
 \* - pre-flight check failed (sensors not ok or not calibrated)  
 \* - safety not disabled  
 \* - system not in manual mode  
 \*/  
 tune\_negative(true);  
 }  
  
 /\* evaluate the main state machine according to mode switches \*/  
 bool first\_rc\_eval = (\_last\_sp\_man.timestamp == 0) && (sp\_man.timestamp > 0);  
 transition\_result\_t main\_res = set\_main\_state(status, &status\_changed);  
  
 /\* store last position lock state \*/  
 \_last\_condition\_global\_position\_valid = status\_flags.condition\_global\_position\_valid;  
  
 /\* play tune on mode change only if armed, blink LED always \*/  
 if (main\_res == TRANSITION\_CHANGED || first\_rc\_eval) {  
 tune\_positive(armed.armed);  
 main\_state\_changed = true;  
  
 } else if (main\_res == TRANSITION\_DENIED) {  
 /\* DENIED here indicates bug in the commander \*/  
 mavlink\_log\_critical(&mavlink\_log\_pub, "Switching to this mode is currently not possible");  
 }  
  
 /\* check throttle kill switch \*/  
 if (sp\_man.kill\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_ON) {  
 /\* set lockdown flag \*/  
 if (!armed.manual\_lockdown) {  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "MANUAL KILL SWITCH ENGAGED");  
 status\_changed = true;  
 armed.manual\_lockdown = true;  
 }  
  
 } else if (sp\_man.kill\_switch == manual\_control\_setpoint\_s::SWITCH\_POS\_OFF) {  
 if (armed.manual\_lockdown) {  
 mavlink\_log\_emergency(&mavlink\_log\_pub, "MANUAL KILL SWITCH OFF");  
 status\_changed = true;  
 armed.manual\_lockdown = false;  
 }  
 }  
  
 /\* no else case: do not change lockdown flag in unconfigured case \*/  
  
 } else {  
 if (!status\_flags.rc\_input\_blocked && !status.rc\_signal\_lost) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "MANUAL CONTROL LOST (at t=%llums)", hrt\_absolute\_time() / 1000);  
 status.rc\_signal\_lost = true;  
 rc\_signal\_lost\_timestamp = sp\_man.timestamp;  
 set\_health\_flags(subsystem\_info\_s::SUBSYSTEM\_TYPE\_RCRECEIVER, true, true, false, status);  
 status\_changed = true;  
 }  
 }  
  
 // 데이터 링크(텔레메트리) 체크  
 data\_link\_checks(highlatencydatalink\_loss\_timeout, highlatencydatalink\_regain\_timeout, datalink\_loss\_timeout, datalink\_regain\_timeout, &status\_changed);  
  
 // engine failure detection  
 // TODO: move out of commander  
 orb\_check(actuator\_controls\_sub, &updated);  
  
 if (updated) {  
 /\* Check engine failure  
 \* only for fixed wing for now  
 \*/  
 if (!status\_flags.circuit\_breaker\_engaged\_enginefailure\_check &&  
 !status.is\_rotary\_wing && !status.is\_vtol && armed.armed) {  
  
 actuator\_controls\_s actuator\_controls = {};  
 orb\_copy(ORB\_ID\_VEHICLE\_ATTITUDE\_CONTROLS, actuator\_controls\_sub, &actuator\_controls);  
  
 const float throttle = actuator\_controls.control[actuator\_controls\_s::INDEX\_THROTTLE];  
 const float current2throttle = battery.current\_a / throttle;  
  
 if (((throttle > ef\_throttle\_thres) && (current2throttle < ef\_current2throttle\_thres))  
 || status.engine\_failure) {  
  
 const float elapsed = hrt\_elapsed\_time(&timestamp\_engine\_healthy) / 1e6f;  
  
 /\* potential failure, measure time \*/  
 if ((timestamp\_engine\_healthy > 0) && (elapsed > ef\_time\_thres)  
 && !status.engine\_failure) {  
  
 status.engine\_failure = true;  
 status\_changed = true;  
  
 PX4\_ERR("Engine Failure");  
 set\_health\_flags(subsystem\_info\_s::SUBSYSTEM\_TYPE\_MOTORCONTROL, true, true, false, status);  
 }  
 }  
  
 } else {  
 /\* no failure reset flag \*/  
 timestamp\_engine\_healthy = hrt\_absolute\_time();  
  
 if (status.engine\_failure) {  
 status.engine\_failure = false;  
 status\_changed = true;  
 }  
 }  
 }  
  
 /\* Reset main state to loiter or auto-mission after takeoff is completed.  
 \* Sometimes, the mission result topic is outdated and the mission is still signaled  
 \* as finished even though we only just started with the takeoff. Therefore, we also  
 \* check the timestamp of the mission\_result topic. \*/  
 if (internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_AUTO\_TAKEOFF  
 && (\_mission\_result\_sub.get().timestamp > internal\_state.timestamp)  
 && \_mission\_result\_sub.get().finished) {  
  
 const bool mission\_available = (\_mission\_result\_sub.get().timestamp > commander\_boot\_timestamp)  
 && (\_mission\_result\_sub.get().instance\_count > 0) && \_mission\_result\_sub.get().valid;  
  
 if ((takeoff\_complete\_act == 1) && mission\_available) {  
 main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_MISSION, status\_flags, &internal\_state);  
  
 } else {  
 main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_LOITER, status\_flags, &internal\_state);  
 }  
 }  
  
 /\* check if we are disarmed and there is a better mode to wait in \*/  
 if (!armed.armed) {  
  
 /\* if there is no radio control but GPS lock the user might want to fly using  
 \* just a tablet. Since the RC will force its mode switch setting on connecting  
 \* we can as well just wait in a hold mode which enables tablet control.  
 \*/  
 if (status.rc\_signal\_lost && (internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_MANUAL)  
 && status\_flags.condition\_home\_position\_valid) {  
  
 main\_state\_transition(status, commander\_state\_s::MAIN\_STATE\_AUTO\_LOITER, status\_flags, &internal\_state);  
 }  
 }  
  
 /\* handle commands last, as the system needs to be updated to handle them \*/  
 orb\_check(cmd\_sub, &updated);  
  
 if (updated) {  
 struct vehicle\_command\_s cmd;  
  
 /\* got command \*/  
 orb\_copy(ORB\_ID(vehicle\_command), cmd\_sub, &cmd);  
  
 /\* handle it \*/  
 if (handle\_command(&status, cmd, &armed, &\_home, &home\_pub, &command\_ack\_pub, &status\_changed)) {  
 status\_changed = true;  
 }  
 }  
  
 /\* Check for failure combinations which lead to flight termination \*/  
 if (armed.armed &&  
 !status\_flags.circuit\_breaker\_flight\_termination\_disabled) {  
 /\* At this point the data link and the gps system have been checked  
 \* If we are not in a manual (RC stick controlled mode)  
 \* and both failed we want to terminate the flight \*/  
 if (internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_MANUAL &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_ACRO &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_RATTITUDE &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_STAB &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_ALTCTL &&  
 internal\_state.main\_state != commander\_state\_s::MAIN\_STATE\_POSCTL &&  
 status.data\_link\_lost) {  
  
 armed.force\_failsafe = true;  
 status\_changed = true;  
 static bool flight\_termination\_printed = false;  
  
 if (!flight\_termination\_printed) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "DL and GPS lost: flight termination");  
 flight\_termination\_printed = true;  
 }  
  
 if (counter % (1000000 / COMMANDER\_MONITORING\_INTERVAL) == 0) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "DL and GPS lost: flight termination");  
 }  
 }  
  
 /\* At this point the rc signal and the gps system have been checked  
 \* If we are in manual (controlled with RC):  
 \* if both failed we want to terminate the flight \*/  
 if ((internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_ACRO ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_RATTITUDE ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_MANUAL ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_STAB ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_ALTCTL ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_POSCTL) &&  
 status.rc\_signal\_lost) {  
  
 armed.force\_failsafe = true;  
 status\_changed = true;  
 static bool flight\_termination\_printed = false;  
  
 if (!flight\_termination\_printed) {  
 warnx("Flight termination because of RC signal loss and GPS failure");  
 flight\_termination\_printed = true;  
 }  
  
 if (counter % (1000000 / COMMANDER\_MONITORING\_INTERVAL) == 0) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "RC and GPS lost: flight termination");  
 }  
 }  
 }  
  
 /\* Get current timestamp \*/  
 const hrt\_abstime now = hrt\_absolute\_time();  
  
 // 홈 위치 자동 지정  
 // automatically set or update home position  
 if (!\_home.manual\_home) {  
 const vehicle\_local\_position\_s &local\_position = \_local\_position\_sub.get();  
  
 if (armed.armed) {  
 if ((!was\_armed || (was\_landed && !land\_detector.landed)) &&  
 (hrt\_elapsed\_time(&commander\_boot\_timestamp) > INAIR\_RESTART\_HOLDOFF\_INTERVAL)) {  
  
 /\* update home position on arming if at least 500 ms from commander start spent to avoid setting home on in-air restart \*/  
 set\_home\_position(home\_pub, \_home, false);  
 }  
  
 } else {  
 if (status\_flags.condition\_home\_position\_valid) {  
 if (land\_detector.landed && local\_position.xy\_valid && local\_position.z\_valid) {  
 /\* distance from home \*/  
 float home\_dist\_xy = -1.0f;  
 float home\_dist\_z = -1.0f;  
 mavlink\_wpm\_distance\_to\_point\_local(\_home.x, \_home.y, \_home.z,  
 local\_position.x, local\_position.y, local\_position.z,  
 &home\_dist\_xy, &home\_dist\_z);  
  
 if ((home\_dist\_xy > local\_position.eph \* 2) || (home\_dist\_z > local\_position.epv \* 2)) {  
  
 /\* update when disarmed, landed and moved away from current home position \*/  
 set\_home\_position(home\_pub, \_home, false);  
 }  
 }  
  
 } else {  
 /\* First time home position update - but only if disarmed \*/  
 set\_home\_position(home\_pub, \_home, false);  
 }  
 }  
  
 /\* Set home position altitude to EKF origin height if home is not set and the EKF has a global origin.  
 \* This allows home atitude to be used in the calculation of height above takeoff location when GPS  
 \* use has commenced after takeoff. \*/  
 if (!\_home.valid\_alt && local\_position.z\_global) {  
 set\_home\_position(home\_pub, \_home, true);  
  
 }  
 }  
  
 // check for arming state change  
 if (was\_armed != armed.armed) {  
 status\_changed = true;  
  
 if (!armed.armed) { // increase the flight uuid upon disarming  
 ++flight\_uuid;  
 // no need for param notification: the only user is mavlink which reads the param upon request  
 param\_set\_no\_notification(\_param\_flight\_uuid, &flight\_uuid);  
 last\_disarmed\_timestamp = hrt\_absolute\_time();  
 }  
 }  
  
 was\_armed = armed.armed;  
  
 // 기체 상태 또는 failsafe 상태에 따라 navigation 상태 바꾸기  
 /\* now set navigation state according to failsafe and main state \*/  
 bool nav\_state\_changed = set\_nav\_state(&status,  
 &armed,  
 &internal\_state,  
 &mavlink\_log\_pub,  
 (link\_loss\_actions\_t)datalink\_loss\_act,  
 \_mission\_result\_sub.get().finished,  
 \_mission\_result\_sub.get().stay\_in\_failsafe,  
 status\_flags,  
 land\_detector.landed,  
 (link\_loss\_actions\_t)rc\_loss\_act,  
 offboard\_loss\_act,  
 offboard\_loss\_rc\_act,  
 posctl\_nav\_loss\_act);  
  
 if (status.failsafe != failsafe\_old) {  
 status\_changed = true;  
  
 if (status.failsafe) {  
 mavlink\_log\_info(&mavlink\_log\_pub, "Failsafe mode enabled");  
  
 } else {  
 mavlink\_log\_info(&mavlink\_log\_pub, "Failsafe mode disabled");  
 }  
  
 failsafe\_old = status.failsafe;  
 }  
  
 // TODO handle mode changes by commands  
 if (main\_state\_changed || nav\_state\_changed) {  
 status\_changed = true;  
 main\_state\_changed = false;  
 }  
  
 // 기체 상태 출판(publish)  
 /\* publish states (armed, control\_mode, vehicle\_status, commander\_state, vehicle\_status\_flags) at 1 Hz or immediately when changed \*/  
 if (hrt\_elapsed\_time(&status.timestamp) >= 1\_s || status\_changed) {  
  
 set\_control\_mode();  
 control\_mode.timestamp = now;  
 orb\_publish(ORB\_ID(vehicle\_control\_mode), control\_mode\_pub, &control\_mode);  
  
 status.timestamp = now;  
 orb\_publish(ORB\_ID(vehicle\_status), status\_pub, &status);  
  
 armed.timestamp = now;  
  
 /\* set prearmed state if safety is off, or safety is not present and 5 seconds passed \*/  
 if (safety.safety\_switch\_available) {  
  
 /\* safety is off, go into prearmed \*/  
 armed.prearmed = safety.safety\_off;  
  
 } else {  
 /\* safety is not present, go into prearmed  
 \* (all output drivers should be started / unlocked last in the boot process  
 \* when the rest of the system is fully initialized)  
 \*/  
 armed.prearmed = (hrt\_elapsed\_time(&commander\_boot\_timestamp) > 5\_s);  
 }  
  
 orb\_publish(ORB\_ID(actuator\_armed), armed\_pub, &armed);  
  
 /\* publish internal state for logging purposes \*/  
 if (commander\_state\_pub != nullptr) {  
 orb\_publish(ORB\_ID(commander\_state), commander\_state\_pub, &internal\_state);  
  
 } else {  
 commander\_state\_pub = orb\_advertise(ORB\_ID(commander\_state), &internal\_state);  
 }  
  
 /\* publish vehicle\_status\_flags \*/  
 status\_flags.timestamp = hrt\_absolute\_time();  
  
 if (vehicle\_status\_flags\_pub != nullptr) {  
 orb\_publish(ORB\_ID(vehicle\_status\_flags), vehicle\_status\_flags\_pub, &status\_flags);  
  
 } else {  
 vehicle\_status\_flags\_pub = orb\_advertise(ORB\_ID(vehicle\_status\_flags), &status\_flags);  
 }  
 }  
  
 // arming 일어날떄 소리내고, 배터리 경고시 소리내기  
 /\* play arming and battery warning tunes \*/  
 if (!arm\_tune\_played && armed.armed && (!safety.safety\_switch\_available || (safety.safety\_switch\_available  
 && safety.safety\_off))) {  
 /\* play tune when armed \*/  
 set\_tune(TONE\_ARMING\_WARNING\_TUNE);  
 arm\_tune\_played = true;  
  
 } else if (!status\_flags.usb\_connected &&  
 (status.hil\_state != vehicle\_status\_s::HIL\_STATE\_ON) &&  
 (battery.warning == battery\_status\_s::BATTERY\_WARNING\_CRITICAL)) {  
 /\* play tune on battery critical \*/  
 set\_tune(TONE\_BATTERY\_WARNING\_FAST\_TUNE);  
  
 } else if ((status.hil\_state != vehicle\_status\_s::HIL\_STATE\_ON) &&  
 (battery.warning == battery\_status\_s::BATTERY\_WARNING\_LOW)) {  
 /\* play tune on battery warning \*/  
 set\_tune(TONE\_BATTERY\_WARNING\_SLOW\_TUNE);  
  
 } else if (status.failsafe) {  
 tune\_failsafe(true);  
  
 } else {  
 set\_tune(TONE\_STOP\_TUNE);  
 }  
  
 /\* reset arm\_tune\_played when disarmed \*/  
 if (!armed.armed || (safety.safety\_switch\_available && !safety.safety\_off)) {  
  
 //Notify the user that it is safe to approach the vehicle  
 if (arm\_tune\_played) {  
 tune\_neutral(true);  
 }  
  
 arm\_tune\_played = false;  
 }  
  
 /\* play sensor failure tunes if we already waited for hotplug sensors to come up and failed \*/  
 status\_flags.condition\_system\_hotplug\_timeout = (hrt\_elapsed\_time(&commander\_boot\_timestamp) > HOTPLUG\_SENS\_TIMEOUT);  
  
 if (!sensor\_fail\_tune\_played && (!status\_flags.condition\_system\_sensors\_initialized  
 && status\_flags.condition\_system\_hotplug\_timeout)) {  
 set\_tune\_override(TONE\_GPS\_WARNING\_TUNE);  
 sensor\_fail\_tune\_played = true;  
 status\_changed = true;  
 }  
  
 counter++;  
  
 int blink\_state = blink\_msg\_state();  
  
 if (blink\_state > 0) {  
 /\* blinking LED message, don't touch LEDs \*/  
 if (blink\_state == 2) {  
 /\* blinking LED message completed, restore normal state \*/  
 control\_status\_leds(&status, &armed, true, &battery, &cpuload);  
 }  
  
 } else {  
 /\* normal state \*/  
 control\_status\_leds(&status, &armed, status\_changed, &battery, &cpuload);  
 }  
  
 status\_changed = false;  
  
 if (!armed.armed) {  
 /\* Reset the flag if disarmed. \*/  
 have\_taken\_off\_since\_arming = false;  
 }  
  
 arm\_auth\_update(now, params\_updated || param\_init\_forced);  
  
 // Handle shutdown request from emergency battery action  
 if(!armed.armed && dangerous\_battery\_level\_requests\_poweroff){  
 mavlink\_log\_critical(&mavlink\_log\_pub, "DANGEROUSLY LOW BATTERY, SHUT SYSTEM DOWN");  
 usleep(200000);  
 int ret\_val = px4\_shutdown\_request(false, false);  
  
 if (ret\_val) {  
 mavlink\_log\_critical(&mavlink\_log\_pub, "SYSTEM DOES NOT SUPPORT SHUTDOWN");  
 dangerous\_battery\_level\_requests\_poweroff = false;  
  
 } else {  
 while (1) { usleep(1); }  
 }  
 }  
  
 usleep(COMMANDER\_MONITORING\_INTERVAL);  
 }  
  
 thread\_should\_exit = true;  
  
 /\* wait for threads to complete \*/  
 int ret = pthread\_join(commander\_low\_prio\_thread, nullptr);  
  
 if (ret) {  
 warn("join failed: %d", ret);  
 }  
  
 rgbled\_set\_color\_and\_mode(led\_control\_s::COLOR\_WHITE, led\_control\_s::MODE\_OFF);  
  
 /\* close fds \*/  
 led\_deinit();  
 buzzer\_deinit();  
 px4\_close(sp\_man\_sub);  
 px4\_close(offboard\_control\_mode\_sub);  
 px4\_close(safety\_sub);  
 px4\_close(cmd\_sub);  
 px4\_close(subsys\_sub);  
 px4\_close(param\_changed\_sub);  
 px4\_close(battery\_sub);  
 px4\_close(land\_detector\_sub);  
 px4\_close(estimator\_status\_sub);  
  
 thread\_running = false;  
}

# 한줄 읽기

## <2018-07-31 화>

왜 stick\_on\_counter < rc\_arm\_hyst ?

commit 94c8371ffe8b3118bb03d2874669bfbed93da747  
Author: Matthias Grob <maetugr@gmail.com>  
Date: Thu Dec 1 17:59:00 2016 +0100  
  
 Arm button fix: toggle arming state only once per arm button press  
  
diff --git a/src/modules/commander/commander.cpp b/src/modules/commander/commander.cpp  
--- a/src/modules/commander/commander.cpp  
+++ b/src/modules/commander/commander.cpp  
@@ -2565,10 +2565,10 @@  
 if ((status.is\_rotary\_wing || (!status.is\_rotary\_wing && land\_detector.landed)) && status.rc\_input\_mode != vehicle\_status\_s::RC\_IN\_MODE  
\_OFF &&  
 (status.arming\_state == vehicle\_status\_s::ARMING\_STATE\_ARMED || status.arming\_state == vehicle\_status\_s::ARMING\_STATE\_ARMED\_ERROR)  
&&  
 (internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_MANUAL ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_ACRO ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_STAB ||  
 internal\_state.main\_state == commander\_state\_s::MAIN\_STATE\_RATTITUDE ||  
 land\_detector.landed) &&  
 ((sp\_man.r < -STICK\_ON\_OFF\_LIMIT && sp\_man.z < 0.1f) || (arm\_switch\_is\_button == 1 && sp\_man.arm\_switch == manual\_control\_setpoint\_  
s::SWITCH\_POS\_ON)) ) {  
  
- if (stick\_off\_counter > rc\_arm\_hyst) {  
+ if (stick\_off\_counter == rc\_arm\_hyst && stick\_on\_counter < rc\_arm\_hyst) {