*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\**

*\* Copyright (c) 2015 PX4 Development Team. All rights reserved.*

*\**

*\* Redistribution and use in source and binary forms, with or without*

*\* modification, are permitted provided that the following conditions*

*\* are met:*

*\**

*\* 1. Redistributions of source code must retain the above copyright*

*\* notice, this list of conditions and the following disclaimer.*

*\* 2. Redistributions in binary form must reproduce the above copyright*

*\* notice, this list of conditions and the following disclaimer in*

*\* the documentation and/or other materials provided with the*

*\* distribution.*

*\* 3. Neither the name PX4 nor the names of its contributors may be*

*\* used to endorse or promote products derived from this software*

*\* without specific prior written permission.*

*\**

*\* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS*

*\* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT*

*\* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS*

*\* FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE*

*\* COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,*

*\* INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,*

*\* BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS*

*\* OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED*

*\* AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT*

*\* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN*

*\* ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE*

*\* POSSIBILITY OF SUCH DAMAGE.*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*/\**

*\* @file attitude\_estimator\_q\_main.cpp*

*\**

*\* Attitude estimator (quaternion based)*

*\**

*\* @author Anton Babushkin <anton.babushkin@me.com>*

*\*/*

#include <drivers/drv\_hrt.h>

#include <lib/ecl/geo/geo.h>

#include <lib/ecl/geo\_lookup/geo\_mag\_declination.h>

#include <mathlib/math/filter/LowPassFilter2p.hpp>

#include <mathlib/mathlib.h>

#include <px4\_config.h>

#include <px4\_posix.h>

#include <px4\_tasks.h>

#include <systemlib/err.h>

#include <parameters/param.h>

#include <perf/perf\_counter.h>

#include <uORB/topics/att\_pos\_mocap.h>

#include <uORB/topics/parameter\_update.h>

#include <uORB/topics/sensor\_combined.h>

#include <uORB/topics/vehicle\_attitude.h>

#include <uORB/topics/vehicle\_global\_position.h>

#include <uORB/topics/vehicle\_magnetometer.h>

**extern** "C" \_\_EXPORT int attitude\_estimator\_q\_main(int argc, char \*argv[]);

**using** math::Vector;

**using** math::Matrix;

**using** math::Quaternion;

**class** **AttitudeEstimatorQ**;

**namespace** attitude\_estimator\_q

{

AttitudeEstimatorQ \*instance;

} *// namespace attitude\_estimator\_q*

**class** **AttitudeEstimatorQ**

{

public:

*/\*\**

*\* Constructor*

*\*/*

AttitudeEstimatorQ();

*/\*\**

*\* Destructor, also kills task.*

*\*/*

~AttitudeEstimatorQ();

*/\*\**

*\* Start task.*

*\**

*\* @return OK on success.*

*\*/*

int start();

**static** void task\_main\_trampoline(int argc, char \*argv[]);

void task\_main();

private:

**const** float \_dt\_min = 0.00001f;

**const** float \_dt\_max = 0.02f;

bool \_task\_should\_exit = false; */\*\*< if true, task should exit \*/*

int \_control\_task = -1; */\*\*< task handle for task \*/*

int \_params\_sub = -1;

int \_sensors\_sub = -1;

int \_global\_pos\_sub = -1;

int \_vision\_sub = -1;

int \_mocap\_sub = -1;

int \_magnetometer\_sub = -1;

orb\_advert\_t \_att\_pub = nullptr;

**struct** {

param\_t w\_acc;

param\_t w\_mag;

param\_t w\_ext\_hdg;

param\_t w\_gyro\_bias;

param\_t mag\_decl;

param\_t mag\_decl\_auto;

param\_t acc\_comp;

param\_t bias\_max;

param\_t ext\_hdg\_mode;

} \_params\_handles{}; */\*\*< handles for interesting parameters \*/*

float \_w\_accel = 0.0f;

float \_w\_mag = 0.0f;

float \_w\_ext\_hdg = 0.0f;

float \_w\_gyro\_bias = 0.0f;

float \_mag\_decl = 0.0f;

bool \_mag\_decl\_auto = false;

bool \_acc\_comp = false;

float \_bias\_max = 0.0f;

int32\_t \_ext\_hdg\_mode = 0;

Vector<3> \_gyro;

Vector<3> \_accel;

Vector<3> \_mag;

Vector<3> \_vision\_hdg;

Vector<3> \_mocap\_hdg;

Quaternion \_q;

Vector<3> \_rates;

Vector<3> \_gyro\_bias;

Vector<3> \_vel\_prev;

hrt\_abstime \_vel\_prev\_t = 0;

Vector<3> \_pos\_acc;

bool \_inited = false;

bool \_data\_good = false;

bool \_ext\_hdg\_good = false;

void update\_parameters(bool force);

int update\_subscriptions();

bool init();

bool update(float dt);

*// Update magnetic declination (in rads) immediately changing yaw rotation*

void update\_mag\_declination(float new\_declination);

};

AttitudeEstimatorQ::AttitudeEstimatorQ()

{

\_params\_handles.w\_acc = param\_find("ATT\_W\_ACC");

\_params\_handles.w\_mag = param\_find("ATT\_W\_MAG");

\_params\_handles.w\_ext\_hdg = param\_find("ATT\_W\_EXT\_HDG");

\_params\_handles.w\_gyro\_bias = param\_find("ATT\_W\_GYRO\_BIAS");

\_params\_handles.mag\_decl = param\_find("ATT\_MAG\_DECL");

\_params\_handles.mag\_decl\_auto = param\_find("ATT\_MAG\_DECL\_A");

\_params\_handles.acc\_comp = param\_find("ATT\_ACC\_COMP");

\_params\_handles.bias\_max = param\_find("ATT\_BIAS\_MAX");

\_params\_handles.ext\_hdg\_mode = param\_find("ATT\_EXT\_HDG\_M");

\_vel\_prev.zero();

\_pos\_acc.zero();

\_gyro.zero();

\_accel.zero();

\_mag.zero();

\_vision\_hdg.zero();

\_mocap\_hdg.zero();

\_q.zero();

\_rates.zero();

\_gyro\_bias.zero();

\_vel\_prev.zero();

\_pos\_acc.zero();

}

*/\*\**

*\* Destructor, also kills task.*

*\*/*

AttitudeEstimatorQ::~AttitudeEstimatorQ()

{

**if** (\_control\_task != -1) {

*/\* task wakes up every 100ms or so at the longest \*/*

\_task\_should\_exit = true;

*/\* wait for a second for the task to quit at our request \*/*

unsigned i = 0;

**do** {

*/\* wait 20ms \*/*

usleep(20000);

*/\* if we have given up, kill it \*/*

**if** (++i > 50) {

px4\_task\_delete(\_control\_task);

**break**;

}

} **while** (\_control\_task != -1);

}

attitude\_estimator\_q::instance = nullptr;

}

int AttitudeEstimatorQ::start()

{

ASSERT(\_control\_task == -1);

*/\* start the task \*/*

\_control\_task = px4\_task\_spawn\_cmd("attitude\_estimator\_q",

SCHED\_DEFAULT,

SCHED\_PRIORITY\_ESTIMATOR,

2000,

(px4\_main\_t)&AttitudeEstimatorQ::task\_main\_trampoline,

nullptr);

**if** (\_control\_task < 0) {

warn("task start failed");

**return** -errno;

}

**return** OK;

}

void AttitudeEstimatorQ::task\_main\_trampoline(int argc, char \*argv[])

{

attitude\_estimator\_q::instance->task\_main();

}

void AttitudeEstimatorQ::task\_main()

{

#ifdef \_\_PX4\_POSIX

perf\_counter\_t \_perf\_accel(perf\_alloc\_once(PC\_ELAPSED, "sim\_accel\_delay"));

perf\_counter\_t \_perf\_mpu(perf\_alloc\_once(PC\_ELAPSED, "sim\_mpu\_delay"));

perf\_counter\_t \_perf\_mag(perf\_alloc\_once(PC\_ELAPSED, "sim\_mag\_delay"));

#endif

\_sensors\_sub = orb\_subscribe(ORB\_ID(sensor\_combined));

\_vision\_sub = orb\_subscribe(ORB\_ID(vehicle\_vision\_attitude));

\_mocap\_sub = orb\_subscribe(ORB\_ID(att\_pos\_mocap));

\_params\_sub = orb\_subscribe(ORB\_ID(parameter\_update));

\_global\_pos\_sub = orb\_subscribe(ORB\_ID(vehicle\_global\_position));

\_magnetometer\_sub = orb\_subscribe(ORB\_ID(vehicle\_magnetometer));

update\_parameters(true);

hrt\_abstime last\_time = 0;

px4\_pollfd\_struct\_t fds[1] = {};

fds[0].fd = \_sensors\_sub;

fds[0].events = POLLIN;

**while** (!\_task\_should\_exit) {

int ret = px4\_poll(fds, 1, 1000);

**if** (ret < 0) {

*// Poll error, sleep and try again*

usleep(10000);

PX4\_WARN("POLL ERROR");

**continue**;

} **else** **if** (ret == 0) {

*// Poll timeout, do nothing*

PX4\_WARN("POLL TIMEOUT");

**continue**;

}

update\_parameters(false);

*// Update sensors*

sensor\_combined\_s sensors;

**if** (orb\_copy(ORB\_ID(sensor\_combined), \_sensors\_sub, &sensors) == PX4\_OK) {

*// Feed validator with recent sensor data*

**if** (sensors.timestamp > 0) {

\_gyro(0) = sensors.gyro\_rad[0];

\_gyro(1) = sensors.gyro\_rad[1];

\_gyro(2) = sensors.gyro\_rad[2];

}

**if** (sensors.accelerometer\_timestamp\_relative != sensor\_combined\_s::RELATIVE\_TIMESTAMP\_INVALID) {

\_accel(0) = sensors.accelerometer\_m\_s2[0];

\_accel(1) = sensors.accelerometer\_m\_s2[1];

\_accel(2) = sensors.accelerometer\_m\_s2[2];

**if** (\_accel.length() < 0.01f) {

PX4\_ERR("WARNING: degenerate accel!");

**continue**;

}

}

\_data\_good = true;

}

*// Update magnetometer*

bool magnetometer\_updated = false;

orb\_check(\_magnetometer\_sub, &magnetometer\_updated);

**if** (magnetometer\_updated) {

vehicle\_magnetometer\_s magnetometer = {};

**if** (orb\_copy(ORB\_ID(vehicle\_magnetometer), \_magnetometer\_sub, &magnetometer) == PX4\_OK) {

\_mag(0) = magnetometer.magnetometer\_ga[0];

\_mag(1) = magnetometer.magnetometer\_ga[1];

\_mag(2) = magnetometer.magnetometer\_ga[2];

**if** (\_mag.length() < 0.01f) {

PX4\_ERR("WARNING: degenerate mag!");

**continue**;

}

}

}

*// Update vision and motion capture heading*

bool vision\_updated = false;

orb\_check(\_vision\_sub, &vision\_updated);

**if** (vision\_updated) {

vehicle\_attitude\_s vision;

**if** (orb\_copy(ORB\_ID(vehicle\_vision\_attitude), \_vision\_sub, &vision) == PX4\_OK) {

math::Quaternion q(vision.q);

math::Matrix<3, 3> Rvis = q.to\_dcm();

math::Vector<3> v(1.0f, 0.0f, 0.4f);

*// Rvis is Rwr (robot respect to world) while v is respect to world.*

*// Hence Rvis must be transposed having (Rwr)' \* Vw*

*// Rrw \* Vw = vn. This way we have consistency*

\_vision\_hdg = Rvis.transposed() \* v;

*// vision external heading usage (ATT\_EXT\_HDG\_M 1)*

**if** (\_ext\_hdg\_mode == 1) {

*// Check for timeouts on data*

\_ext\_hdg\_good = vision.timestamp > 0 && (hrt\_elapsed\_time(&vision.timestamp) < 500000);

}

}

}

bool mocap\_updated = false;

orb\_check(\_mocap\_sub, &mocap\_updated);

**if** (mocap\_updated) {

att\_pos\_mocap\_s mocap;

**if** (orb\_copy(ORB\_ID(att\_pos\_mocap), \_mocap\_sub, &mocap) == PX4\_OK) {

math::Quaternion q(mocap.q);

math::Matrix<3, 3> Rmoc = q.to\_dcm();

math::Vector<3> v(1.0f, 0.0f, 0.4f);

*// Rmoc is Rwr (robot respect to world) while v is respect to world.*

*// Hence Rmoc must be transposed having (Rwr)' \* Vw*

*// Rrw \* Vw = vn. This way we have consistency*

\_mocap\_hdg = Rmoc.transposed() \* v;

*// Motion Capture external heading usage (ATT\_EXT\_HDG\_M 2)*

**if** (\_ext\_hdg\_mode == 2) {

*// Check for timeouts on data*

\_ext\_hdg\_good = mocap.timestamp > 0 && (hrt\_elapsed\_time(&mocap.timestamp) < 500000);

}

}

}

bool gpos\_updated = false;

orb\_check(\_global\_pos\_sub, &gpos\_updated);

**if** (gpos\_updated) {

vehicle\_global\_position\_s gpos;

**if** (orb\_copy(ORB\_ID(vehicle\_global\_position), \_global\_pos\_sub, &gpos) == PX4\_OK) {

**if** (\_mag\_decl\_auto && gpos.eph < 20.0f && hrt\_elapsed\_time(&gpos.timestamp) < 1000000) {

*/\* set magnetic declination automatically \*/*

update\_mag\_declination(math::radians(get\_mag\_declination(gpos.lat, gpos.lon)));

}

**if** (\_acc\_comp && gpos.timestamp != 0 && hrt\_absolute\_time() < gpos.timestamp + 20000 && gpos.eph < 5.0f && \_inited) {

*/\* position data is actual \*/*

Vector<3> vel(gpos.vel\_n, gpos.vel\_e, gpos.vel\_d);

*/\* velocity updated \*/*

**if** (\_vel\_prev\_t != 0 && gpos.timestamp != \_vel\_prev\_t) {

float vel\_dt = (gpos.timestamp - \_vel\_prev\_t) / 1e6f;

*/\* calculate acceleration in body frame \*/*

\_pos\_acc = \_q.conjugate\_inversed((vel - \_vel\_prev) / vel\_dt);

}

\_vel\_prev\_t = gpos.timestamp;

\_vel\_prev = vel;

} **else** {

*/\* position data is outdated, reset acceleration \*/*

\_pos\_acc.zero();

\_vel\_prev.zero();

\_vel\_prev\_t = 0;

}

}

}

*/\* time from previous iteration \*/*

hrt\_abstime now = hrt\_absolute\_time();

**const** float dt = math::constrain((now - last\_time) / 1e6f, \_dt\_min, \_dt\_max);

last\_time = now;

**if** (update(dt)) {

vehicle\_attitude\_s att = {

.timestamp = sensors.timestamp,

.rollspeed = \_rates(0),

.pitchspeed = \_rates(1),

.yawspeed = \_rates(2),

.q = {\_q(0), \_q(1), \_q(2), \_q(3)},

.delta\_q\_reset = {},

.quat\_reset\_counter = 0,

};

*/\* the instance count is not used here \*/*

int att\_inst;

orb\_publish\_auto(ORB\_ID(vehicle\_attitude), &\_att\_pub, &att, &att\_inst, ORB\_PRIO\_HIGH);

}

}

#ifdef \_\_PX4\_POSIX

perf\_end(\_perf\_accel);

perf\_end(\_perf\_mpu);

perf\_end(\_perf\_mag);

#endif

orb\_unsubscribe(\_params\_sub);

orb\_unsubscribe(\_sensors\_sub);

orb\_unsubscribe(\_global\_pos\_sub);

orb\_unsubscribe(\_vision\_sub);

orb\_unsubscribe(\_mocap\_sub);

orb\_unsubscribe(\_magnetometer\_sub);

}

void AttitudeEstimatorQ::update\_parameters(bool force)

{

bool updated = force;

**if** (!updated) {

orb\_check(\_params\_sub, &updated);

}

**if** (updated) {

parameter\_update\_s param\_update;

orb\_copy(ORB\_ID(parameter\_update), \_params\_sub, &param\_update);

param\_get(\_params\_handles.w\_acc, &\_w\_accel);

param\_get(\_params\_handles.w\_mag, &\_w\_mag);

param\_get(\_params\_handles.w\_ext\_hdg, &\_w\_ext\_hdg);

param\_get(\_params\_handles.w\_gyro\_bias, &\_w\_gyro\_bias);

float mag\_decl\_deg = 0.0f;

param\_get(\_params\_handles.mag\_decl, &mag\_decl\_deg);

update\_mag\_declination(math::radians(mag\_decl\_deg));

int32\_t mag\_decl\_auto\_int;

param\_get(\_params\_handles.mag\_decl\_auto, &mag\_decl\_auto\_int);

\_mag\_decl\_auto = (mag\_decl\_auto\_int != 0);

int32\_t acc\_comp\_int;

param\_get(\_params\_handles.acc\_comp, &acc\_comp\_int);

\_acc\_comp = (acc\_comp\_int != 0);

param\_get(\_params\_handles.bias\_max, &\_bias\_max);

param\_get(\_params\_handles.ext\_hdg\_mode, &\_ext\_hdg\_mode);

}

}

bool AttitudeEstimatorQ::init()

{

*// Rotation matrix can be easily constructed from acceleration and mag field vectors*

*// 'k' is Earth Z axis (Down) unit vector in body frame*

Vector<3> k = -\_accel;

k.normalize();

*// 'i' is Earth X axis (North) unit vector in body frame, orthogonal with 'k'*

Vector<3> i = (\_mag - k \* (\_mag \* k));

i.normalize();

*// 'j' is Earth Y axis (East) unit vector in body frame, orthogonal with 'k' and 'i'*

Vector<3> j = k % i;

*// Fill rotation matrix*

Matrix<3, 3> R;

R.set\_row(0, i);

R.set\_row(1, j);

R.set\_row(2, k);

*// Convert to quaternion*

\_q.from\_dcm(R);

*// Compensate for magnetic declination*

Quaternion decl\_rotation;

decl\_rotation.from\_yaw(\_mag\_decl);

\_q = decl\_rotation \* \_q;

\_q.normalize();

**if** (PX4\_ISFINITE(\_q(0)) && PX4\_ISFINITE(\_q(1)) &&

PX4\_ISFINITE(\_q(2)) && PX4\_ISFINITE(\_q(3)) &&

\_q.length() > 0.95f && \_q.length() < 1.05f) {

\_inited = true;

} **else** {

\_inited = false;

}

**return** \_inited;

}

bool AttitudeEstimatorQ::update(float dt)

{

**if** (!\_inited) {

**if** (!\_data\_good) {

**return** false;

}

**return** init();

}

Quaternion q\_last = \_q;

*// Angular rate of correction*

Vector<3> corr;

float spinRate = \_gyro.length();

**if** (\_ext\_hdg\_mode > 0 && \_ext\_hdg\_good) {

**if** (\_ext\_hdg\_mode == 1) {

*// Vision heading correction*

*// Project heading to global frame and extract XY component*

Vector<3> vision\_hdg\_earth = \_q.conjugate(\_vision\_hdg);

float vision\_hdg\_err = \_wrap\_pi(atan2f(vision\_hdg\_earth(1), vision\_hdg\_earth(0)));

*// Project correction to body frame*

corr += \_q.conjugate\_inversed(Vector<3>(0.0f, 0.0f, -vision\_hdg\_err)) \* \_w\_ext\_hdg;

}

**if** (\_ext\_hdg\_mode == 2) {

*// Mocap heading correction*

*// Project heading to global frame and extract XY component*

Vector<3> mocap\_hdg\_earth = \_q.conjugate(\_mocap\_hdg);

float mocap\_hdg\_err = \_wrap\_pi(atan2f(mocap\_hdg\_earth(1), mocap\_hdg\_earth(0)));

*// Project correction to body frame*

corr += \_q.conjugate\_inversed(Vector<3>(0.0f, 0.0f, -mocap\_hdg\_err)) \* \_w\_ext\_hdg;

}

}

**if** (\_ext\_hdg\_mode == 0 || !\_ext\_hdg\_good) {

*// Magnetometer correction*

*// Project mag field vector to global frame and extract XY component*

Vector<3> mag\_earth = \_q.conjugate(\_mag);

float mag\_err = \_wrap\_pi(atan2f(mag\_earth(1), mag\_earth(0)) - \_mag\_decl);

float gainMult = 1.0f;

**const** float fifty\_dps = 0.873f;

**if** (spinRate > fifty\_dps) {

gainMult = math::min(spinRate / fifty\_dps, 10.0f);

}

*// Project magnetometer correction to body frame*

corr += \_q.conjugate\_inversed(Vector<3>(0.0f, 0.0f, -mag\_err)) \* \_w\_mag \* gainMult;

}

\_q.normalize();

*// Accelerometer correction*

*// Project 'k' unit vector of earth frame to body frame*

*// Vector<3> k = \_q.conjugate\_inversed(Vector<3>(0.0f, 0.0f, 1.0f));*

*// Optimized version with dropped zeros*

Vector<3> k(

2.0f \* (\_q(1) \* \_q(3) - \_q(0) \* \_q(2)),

2.0f \* (\_q(2) \* \_q(3) + \_q(0) \* \_q(1)),

(\_q(0) \* \_q(0) - \_q(1) \* \_q(1) - \_q(2) \* \_q(2) + \_q(3) \* \_q(3))

);

corr += (k % (\_accel - \_pos\_acc).normalized()) \* \_w\_accel;

*// Gyro bias estimation*

**if** (spinRate < 0.175f) {

\_gyro\_bias += corr \* (\_w\_gyro\_bias \* dt);

**for** (int i = 0; i < 3; i++) {

\_gyro\_bias(i) = math::constrain(\_gyro\_bias(i), -\_bias\_max, \_bias\_max);

}

}

\_rates = \_gyro + \_gyro\_bias;

*// Feed forward gyro*

corr += \_rates;

*// Apply correction to state*

\_q += \_q.derivative(corr) \* dt;

*// Normalize quaternion*

\_q.normalize();

**if** (!(PX4\_ISFINITE(\_q(0)) && PX4\_ISFINITE(\_q(1)) &&

PX4\_ISFINITE(\_q(2)) && PX4\_ISFINITE(\_q(3)))) {

*// Reset quaternion to last good state*

\_q = q\_last;

\_rates.zero();

\_gyro\_bias.zero();

**return** false;

}

**return** true;

}

void AttitudeEstimatorQ::update\_mag\_declination(float new\_declination)

{

*// Apply initial declination or trivial rotations without changing estimation*

**if** (!\_inited || fabsf(new\_declination - \_mag\_decl) < 0.0001f) {

\_mag\_decl = new\_declination;

} **else** {

*// Immediately rotate current estimation to avoid gyro bias growth*

Quaternion decl\_rotation;

decl\_rotation.from\_yaw(new\_declination - \_mag\_decl);

\_q = decl\_rotation \* \_q;

\_mag\_decl = new\_declination;

}

}

int attitude\_estimator\_q\_main(int argc, char \*argv[])

{

**if** (argc < 2) {

warnx("usage: attitude\_estimator\_q {start|stop|status}");

**return** 1;

}

**if** (!strcmp(argv[1], "start")) {

**if** (attitude\_estimator\_q::instance != nullptr) {

warnx("already running");

**return** 1;

}

attitude\_estimator\_q::instance = **new** AttitudeEstimatorQ;

**if** (attitude\_estimator\_q::instance == nullptr) {

warnx("alloc failed");

**return** 1;

}

**if** (OK != attitude\_estimator\_q::instance->start()) {

**delete** attitude\_estimator\_q::instance;

attitude\_estimator\_q::instance = nullptr;

warnx("start failed");

**return** 1;

}

**return** 0;

}

**if** (!strcmp(argv[1], "stop")) {

**if** (attitude\_estimator\_q::instance == nullptr) {

warnx("not running");

**return** 1;

}

**delete** attitude\_estimator\_q::instance;

attitude\_estimator\_q::instance = nullptr;

**return** 0;

}

**if** (!strcmp(argv[1], "status")) {

**if** (attitude\_estimator\_q::instance) {

warnx("running");

**return** 0;

} **else** {

warnx("not running");

**return** 1;

}

}

warnx("unrecognized command");

**return** 1;

}