```
/****************************************************************************
 *  Copyright (c) 2025 Stanislas Le Person
 *
 *  Description: Dynamic Soaring PX4 module
 *  - Computes wind gradient and energy rate
 *  - Publishes energy rate to debug_value
 ****************************************************************************/

#include <float.h>
#include <fw_performance_model/PerformanceModel.hpp>
#include <drivers/drv_hrt.h>
#include <lib/geo/geo.h>
#include <lib/atmosphere/atmosphere.h>
#include <lib/npfg/CourseToAirspeedRefMapper.hpp>
#include <lib/npfg/AirspeedDirectionController.hpp>
#include <lib/mathlib/mathlib.h>
#include <lib/perf/perf_counter.h>
#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/defines.h>
#include <px4_platform_common/module.h>
#include <px4_platform_common/module_params.h>
#include <px4_platform_common/posix.h>
#include <px4_platform_common/px4_work_queue/WorkItem.hpp>
#include <slew_rate/SlewRate.hpp>
#include <uORB/uORB.h>
#include <uORB/topics/debug_value.h>
#include <px4_platform_common/log.h>
#include <uORB/Publication.hpp>
#include <uORB/PublicationMulti.hpp>
#include <uORB/Subscription.hpp>
#include <uORB/SubscriptionCallback.hpp>
```

```
#include <uORB/topics/airspeed_validated.h>

#include <uORB/topics/vehicle_local_position.h>
#include <uORB/topics/wind.h>

#include <px4_platform_common/events.h>
#include <iostream>

#include <px4_platform_common/getopt.h>

#include <uORB/topics/sensor_combined.h>

#include <uORB/SubscriptionInterval.hpp>

using namespace time_literals;

class FwDynamicSoaring : public ModuleBase<FwDynamicSoaring>, public ModuleParams
{
public:
  FwDynamicSoaring(): ModuleParams(nullptr)
  {
    //_last_run = hrt_absolute_time();
    _last_run = 0;
    _have_prev_energy = false;
    PX4_INFO("FwDynamicSoaring constructor called !");
  }

  ~FwDynamicSoaring() override = default;

  static FwDynamicSoaring *instantiate(int argc, char *argv[]){
    PX4_INFO("FwDynamicSoaring instantiated");
```

```
    return new FwDynamicSoaring();
}

static int task_spawn(int argc, char *argv[]){
    _task_id = px4_task_spawn_cmd("fw_dynamic_soaring",
                                  SCHED_DEFAULT,
                                  SCHED_PRIORITY_DEFAULT,
                                  4096,
                                  (px4_main_t)&run_trampoline,
                                  (char *const *)argv);

    if (_task_id < 0) {
        _task_id = -1;
        return -errno;
    }

    return 0;
}

static int print_usage(const char *reason = nullptr){
    if (reason) {
        PX4_WARN("%s\n", reason);
    }

    PRINT_MODULE_DESCRIPTION(
        R"DESCR_STR(
### Description
Dynamic Soaring module estimates wind gradient and energy rate.
### Usage
$ fw_dynamic_soaring start
)DESCR_STR");
```

```
    PRINT_MODULE_USAGE_NAME("fw_dynamic_soaring", "soaring");
    PRINT_MODULE_USAGE_COMMAND("start");
    PRINT_MODULE_USAGE_DEFAULT_COMMANDS();

    return 0;
}

static int custom_command(int argc, char *argv[]){
    return print_usage("unrecognized command");
}

void run() override
{
    PX4_INFO("Dynamic Soaring: run() started");

    // Local subscriptions and publications used in the loop
    uORB::Publication<debug_value_s> dbg_pub{ORB_ID(debug_value)};

    // Loop timing
    constexpr uint32_t loop_delay_us = 100_ms;

    const float TURN_DURATION_S = 3.5f; // seconds approximate per turn (tunable)
    const float CLIMB_DURATION_S = 2.5f;
    const float DESCEND_DURATION_S = 2.5f;

    hrt_abstime phase_timer_start = hrt_absolute_time();

    while (!should_exit()) {

        // -------------------------
```

```
// Read sensors
// -------------------------
vehicle_local_position_s lpos{};
airspeed_validated_s asp{};
wind_s wind{};
//manual_control_input_s manual{};

bool have_lpos = false;
bool have_asp  = false;
bool have_wind = false;
//bool have_manual = false;

if (_local_pos_sub.updated()) {
   _local_pos_sub.copy(&lpos);
   have_lpos = true;
}

if (_airspeed_sub.updated()) {
   _airspeed_sub.copy(&asp);
   have_asp = true;
}

if (_wind_sub.updated()) {
   _wind_sub.copy(&wind);
   have_wind = true;
}

// -------------------------
// Compute energy & wind gradient (phase 1 logic)
// -------------------------
if (have_lpos && have_asp) {
```

```
        const float alt = -lpos.z; // NED -> up
        const float airspeed = PX4_ISFINITE(asp.true_airspeed_m_s) ? asp.true_airspeed_m_s :
asp.calibrated_airspeed_m_s;
        compute_energy(alt, airspeed);
    }

    if (have_lpos && have_wind) {
        const float alt = -lpos.z;
        const float wind_speed = std::sqrt(wind.windspeed_north * wind.windspeed_north +
wind.windspeed_east * wind.windspeed_east);
        compute_wind_gradient(alt, wind_speed);
    }

    const float elapsed_s = (hrt_absolute_time() - phase_timer_start) / 1e6f;

    // Simple scripted transition timing based on elapsed_s and current phase
    switch (_phase) {
    case DSPhase::IDLE:
      // If wind gradient positive and we have energy gain, start a CLIMB automatically
      if (_wind_gradient > 0.2f && _energy_rate > 0.0f) {
        _phase = DSPhase::CLIMB;
        PX4_INFO("DS auto -> CLIMB (wind grad %.2f, E_rate %.3f)", (double)_wind_gradient,
(double)_energy_rate);
        phase_timer_start = hrt_absolute_time();
      }
      break;

    case DSPhase::CLIMB:
      // After climb duration -> TURN
      if (elapsed_s > CLIMB_DURATION_S) {
        _phase = DSPhase::TURN;
        PX4_INFO("DS auto -> TURN");
```

```
            phase_timer_start = hrt_absolute_time();
        }
        break;

    case DSPhase::TURN:
        // After turn duration -> DESCEND
        if (elapsed_s > TURN_DURATION_S) {
            _phase = DSPhase::DESCEND;
            PX4_INFO("DS auto -> DESCEND");
            phase_timer_start = hrt_absolute_time();
        }
        break;

    case DSPhase::DESCEND:
        // After descend duration -> CLIMB (complete cycle)
        if (elapsed_s > DESCEND_DURATION_S) {
            _phase = DSPhase::CLIMB;
            PX4_INFO("DS auto -> CLIMB (cycle)");
            phase_timer_start = hrt_absolute_time();
        }
        break;
    }


// Publish debug values for analysis (always safe)
{
    debug_value_s dbg{};
    dbg.timestamp = hrt_absolute_time();
    dbg.ind = 0; // energy rate
    dbg.value = _energy_rate;
    dbg_pub.publish(dbg);
```

```
        }
        {
            debug_value_s dbg{};
            dbg.timestamp = hrt_absolute_time();
            dbg.ind = 1; // wind gradient
            dbg.value = _wind_gradient;
            dbg_pub.publish(dbg);
        }
        {
            debug_value_s dbg{};
            dbg.timestamp = hrt_absolute_time();
            dbg.ind = 2; // DS phase (as float)
            dbg.value = static_cast<float>(_phase);
            dbg_pub.publish(dbg);
        }

        // -------------------------
        // Loop delay
        // -------------------------
        px4_usleep(loop_delay_us);
    } // end while

    PX4_INFO("Dynamic Soaring: run() exiting");
  }
private:
  uORB::Subscription _local_pos_sub{ORB_ID(vehicle_local_position)};
  uORB::Subscription _airspeed_sub{ORB_ID(airspeed_validated)};
  uORB::Subscription _wind_sub{ORB_ID(wind)};

  hrt_abstime _last_run{0};
  bool _have_prev_energy{false};
```

```
float _energy_total_prev{0.0f};
float _energy_rate{0.0f};
float _wind_speed_prev{0.0f};
float _wind_alt_prev{0.0f};
float _wind_gradient{0.0f};

enum class DSPhase : uint8_t { IDLE, CLIMB, TURN, DESCEND };
DSPhase _phase{DSPhase::IDLE};

void compute_energy(float alt, float airspeed){
   const float g = 9.81f;
   const hrt_abstime now = hrt_absolute_time();

   float potential = g * alt;
   float kinetic = 0.5f * airspeed * airspeed;
   float total = potential + kinetic;

   if (!_have_prev_energy || _last_run == 0) {
      // initialize previous energy and timestamp
      _energy_total_prev = total;
      _energy_rate = 0.0f;
      _last_run = now;
      _have_prev_energy = true;
      return;
   }

   const float dt = (now - _last_run) / 1e6f; // seconds

   if (dt <= 0.0001f) {
      // too small dt, avoid division by zero / noise
      _energy_rate = 0.0f;
```

```
    } else {
      _energy_rate = (total - _energy_total_prev) / dt;
    }

    _energy_total_prev = total;
    _last_run = now;
  }

  void compute_wind_gradient(float alt, float wind_speed){
    if (fabsf(alt - _wind_alt_prev) > 0.5f) { // 0.5m resolution
      _wind_gradient = (wind_speed - _wind_speed_prev) / (alt - _wind_alt_prev);
      _wind_alt_prev = alt;
      _wind_speed_prev = wind_speed;
    }
  }
};

extern "C" __EXPORT int fw_dynamic_soaring_main(int argc, char *argv[])
{
  //PX4_INFO("Hello_DS");
  return FwDynamicSoaring::main(argc, argv);
}
```