Class Project and Useful Articles

- Project Introduction and Guidelines
- Project Hints
- Unit Testing code that touches the HW registers
- Use single periodic callback if possible
- CANTools
- Exploring DBC Autogenerated API

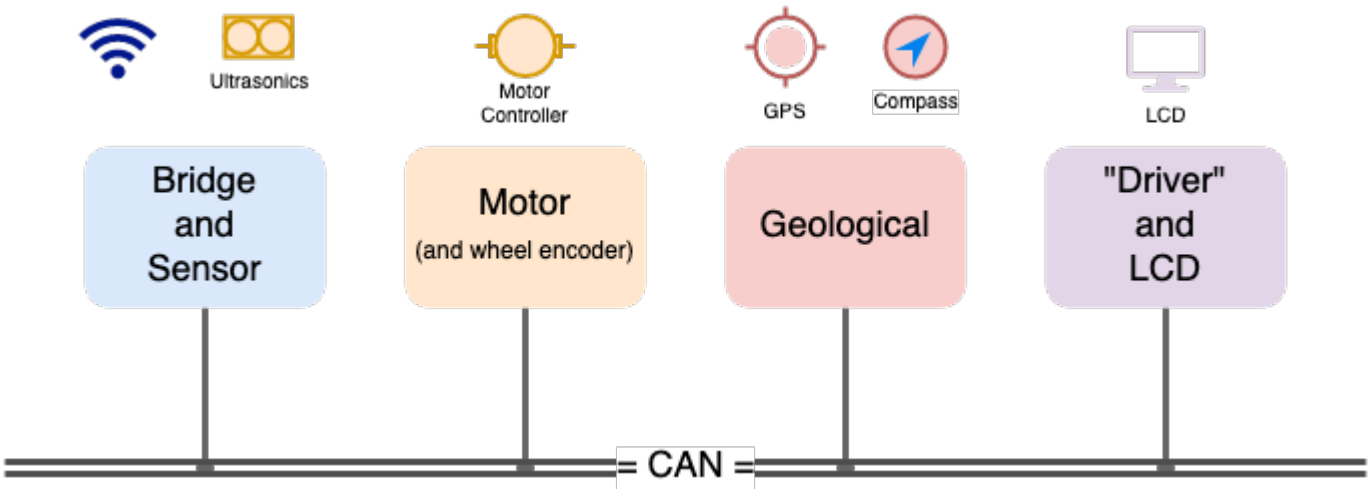
Project Introduction and Guidelines

Shopping List

1. Four SJtwo Development Kits + 4 CAN Transceivers
2. Bridge and Sensors Accessories - Bluetooth or Wifi Breakout Board and Ultrasonic Sensors
3. Motor Controller Accessories - RPM Sensors
4. GEO Controller Accessories: GPS Breakout Board, Compass, GPS Antenna
5. Extra LCD and LEDs for visual effects.
6. Mechanical - RC Car, RC CAR Battery charger, Lithium-Ion Battery
7. PCB parts and other Miscellaneous parts for Hardware.

Controllers

Various different controllers are used and each should have limited and exclusive responsibilities. Use this article as a reference to derive your project schedule and individual team member tasks.



An example of how GPS coordinates are sent from the Mobile phone:

1. Mobile phone sends data as ASCII text to the Bridge
2. The bridge controller receives information over UART
3. The bridge controller sends data using DBC/CAN to the Geological
4. No other controller should need to know the destination coordinates because the Geological controller will guide the Driver controller with its compass heading, and tell it to stop when necessary

Suggested Roles

- Mobile application development
- 3+ General developers (could be dedicated developer per controller or just collaborative effort)
- Testing and Integration person (2 people who can work together)

1: Bridge and Sensor Controller

Requirements

- Shall be interfaced to all of the distance sensors
  - Note that your board only has 3 ADC channels if you choose to use this interface
  - Note that ultrasonic sensors can collide in their sound waves and interfere with each other. It may be better to use different frequency based sensors (maybe different vendor or models) to minimize the interference.
- Shall output all sensor information with a minimum refresh rate of 20Hz
- Shall provide the interface (such as Bluetooth serial) to a mobile application running on a phone
  - The typical interface to a Bluetooth or Wifi is UART

    - Bluetooth requires pairing and is usually stable after that
    - Wifi would require your wi-fi to act as an access point for your mobile phone to
    - Both interfaces should work, and you can pick one based on past semesters' reports
    - A recommendation is that a Mobile phone sends a line of command terminated by a newline, and then the Bridge controller can parse the information (such as using scanf)
- Required but low priority:
    - Provide battery voltage reading to Mobile Application and possibly to the DRIVER to output on the LCD

2: Motor Controller

Requirements

- Shall be interfaced with speed sensors or wheel encoders to provide speed information
    - This should be transmitted on the CAN bus as either kph or mph; usually, the units will be small and you could use DBC scale of `0.001` or `0.01`
- Shall be interfaced to motor controllers (or servo controller) to control steering and speed
    - Recommend hobby grade RC car (such as Traxxas), and not a $20 RC car from Amazon
- Provide self-test capability button
    - Self-Test pressed, so Driver controller should be commanded to a 5 seconds wheel test mode
    - This should include forward, backward motion, and steering test
- Shall process the speed command, and compensate for the grade of the ground (uphill or downhill)
    - For example, the DRIVER may command 0.5 kph, and the motor controller should process the wheel encoder and use a simple PID control loop to match the command regardless of the grade or battery power

---

3: Geological Controller

Requirements

- Shall be interfaced to a magnetometer
- Shall be interfaced with a GPS
- Shall provide raw GPS and compass readings
- Shall have the ability to "set destination" and then provide heading degree towards destination
    - The driver controller should not need to get involved in GPS data. It should simply receive a compass destination heading, and an actual compass heading. When there is no obstacle, it should simply try to take its current heading towards the destination, heading (which is just a

compass degree)
- Shall have the waypoints algorithm

4: Driver and LCD Controller

Requirements

- Shall be interfaced to an LCD display to output meaningful and diagnostic information
  - Sensor values commanded motor values etc.
  - A Recommendation is to use a simple UART or I2C based LCD
  - Something like this
- Shall receive all relevant sensor messages and process them for obstacle avoidance
- Shall receive compass actual degree, and heading degree (towards the destination)
- Shall have the obstacle avoidance algorithm
  - In the absence of any obstacles, it shall use the compass to follow the GPS destination
- Shall send output drive commands to the Motor Controller

5: Mobile Application

Requirements

- Minimize the buttons (and hence the code)
  - If there is no Bluetooth connection, automatically display a list of Bluetooth devices
  - Once connected, automatically show the Google Maps page
- Shall have Google Maps or similar for a user to pick the destination
- Shall transmit the destination to the Bridge Controller as necessary
- The Optional requirement to display car data, such as speed, compass, and anything else useful to you during debugging. Definitely, some extra credit opportunity, here based on how well you accomplish your Mobile Application.

The definite article More (Definitions, Synonyms, Translation)

Focus

There are different domains in your project that we need to ensure are working very robustly.

1. Distance Sensors

One person should focus solely on ensuring that VERY reliable values are being output from the sensor controller at least at 20Hz. Consider the following test case scenarios:

- Smoke tests: Test values while stationary
- Interference tests: With a known test scenario (such as blocking one side), are the sensor values stable? For example, if we are blocking only the right side, is the left sensor value still stable?
- Test on a moving car: Are the sensor mounts stable? Are the sensors pivoted downwards such that the floor is appearing to be an obstacle?
- There are many more test scenarios but the overall objective is to ensure that sensors are very reliable.

2. Compass

- Test in stationary vehicle
- Test while vehicle is moving
  - Does the tilt of the vehicle make a big difference?
- Test while moving the front motor
- Test while moving the rear motor
- Test while moving both motors
  - Is there magnetic interference?

3. Motor and wheel encoder

- Test with varying battery voltage
- Test with different speed command and ensure RPM consistency

The rest is mainly more and more testing, but there should be a systematic approach to this.

Project Hints

Git strategy to share common DBC file

A common issue for everyone is how to have separate projects in your Git repo.

1. One option is to create a different folder, one for each project
   - Maybe have a top-level DBC file, and manually copy to all other folders upon update
   - Maybe infrastructure code can find the DBC file at the root level directory itself? This might be a simple modification in the Python-based scons script

2. Another option is to use different Git branches
   - Single folder for your project, such as lpc40xx_freertos but different "virtual" master branches
   - `git checkout master_driver`
   - `git checkout master_sensor`
   - The "master" branch is where the DBC is at, so when people want to get the latest, what they do is:
     - `git checkout master_driver`
     - `git checkout master`
     - `git pull origin master`
     - `git checkout master_driver`
     - `git rebase master`
     - `git push origin head`

How can you nest an external repository's DBC file in your project repository.

- Git submodule
  - Your DBC can live in a completely separate repo, maybe this repository is nothing but a single DBC file
  - You can nest this external git submodule as a folder inside your lpc40xx_freertos directory

- So if someone changes the DBC file at the dedicated dbc repo, then everyone needs to update it
  - git checkout master
  - cd dbc_directory (nested git submodule)
  - git pull origin master (of the external dbc repo, this will pull in the latest changes from there)
  - cd - (step outside of the nested git submodule)
  - git add dbc_directory (you update the githash that are pointing to the external repo's commit)
  - git commit -m "update dbc"
  - git push origin head

Motor Control

It is advised to have a state machine to move your car, such as forward, forward to reverse and reverse to forward.

```
void motor__run_state_machine_10hz(int call_counter) {
 switch (current_state) {
  case forward:
    if (desired_movement == reverse) {
      current_state = transient_state;
    }
    break;

  case transient_state:
    if (entry_to_this_state) {
      transient_state_entry_counter = call_counter;
    }
    if (call_counter >= (transient_state_entry_counter + 5) { // 500ms elapsed
      if (desired_movement == reverse) {
        current_state = reverse;
      } else {
        // ...
      }
    }
    break;
```

```
  case reverse:
    // ...
 }}
```

Simple PID

The motor controller should use a simple PID to control the motors.

```
// return percentage 0-100 which then may need to be translated to the Servo PWM
float motor_pwm_1hz(float actual_speed_kph, float target_speed_kph) {
  // Also handle the case of deceleration or a complete stop
  if (0 == target_speed_kph) {
    return 0;
  }

  // Handle other cases
}
```

Receive CAN in only one function

Students oftentimes tend to try to handle CAN frame reception in multiple functions. The problem is that this creates a non-deterministic operation as some frames may be dropped in one place in code, and may not be handled where you really mean to handle them.

Note the following properties:

- A while loop to empty out the CAN receive queue
  - Handling just one frame per function call will accumulate CAN frames leading to data loss
- After creating a message header, call all decode functions
  - Only one decode function will decode at most since the message header will match only once
  - This reduces your testing effort as you do not need manual switch/case statements

```
void can_handle_all_frames(void) {
```

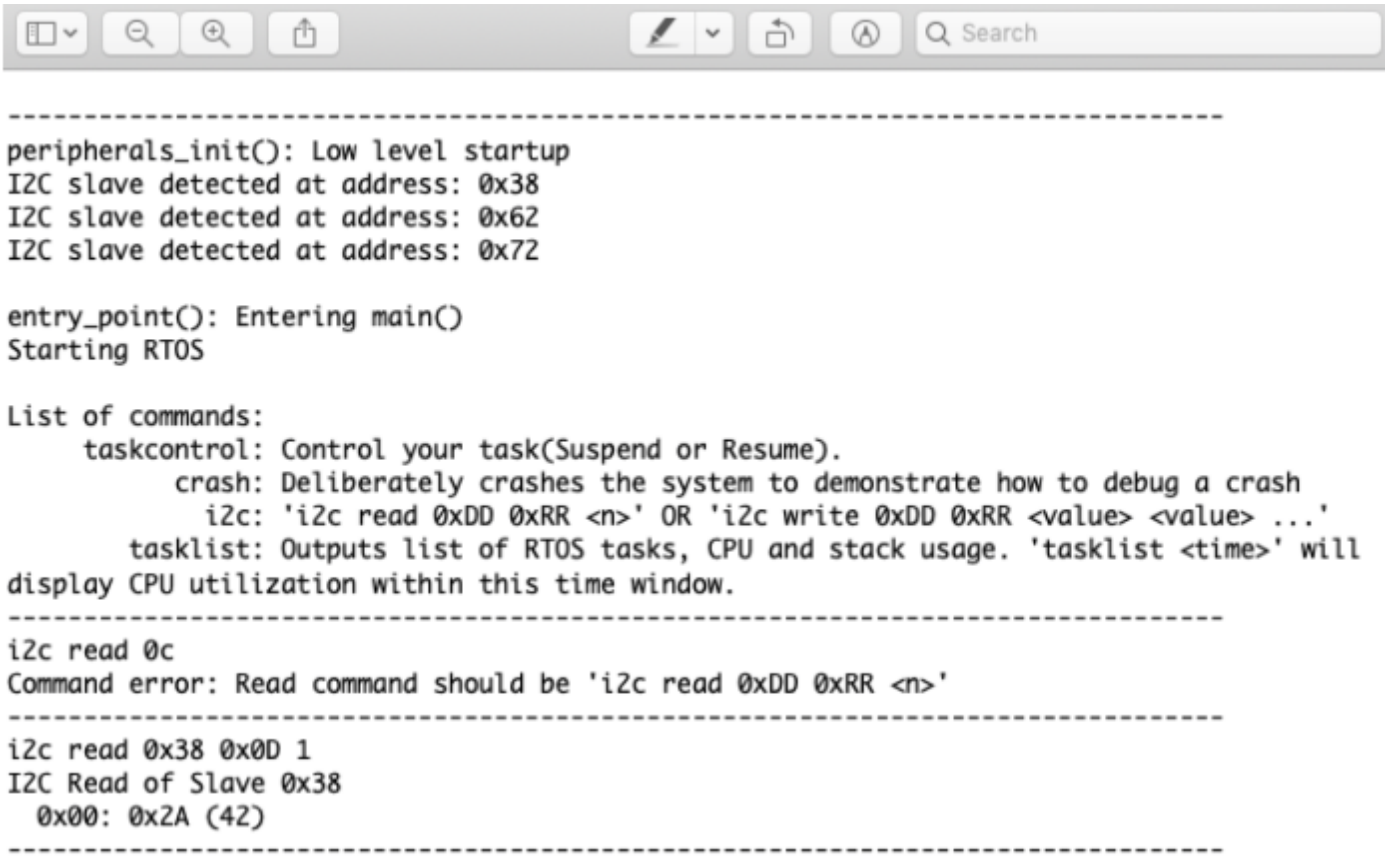
```
    while (can_rx(...)) {
      msg_hdr = create_msg_header();

      dbc_decode_...();
      dbc_decode_...();
    }}
```

Test I2C Sensor

If you have a sensor such as a compass that operates using I2C, then it is advised to use the built-in I2C CLI command on your SJ2 board to test out the sensor registers. Make sure you run the CLI task in your main.c and then simply type "help i2c" to explore the CLI command and trial the sensor data.

If the compass is interfaced on I2C and the slave address of the compass is 0x38. Then you can read a particular register using the CLI command `i2c read SLAVE_ADDRESS REGISTER_ADDRESS <n>`



Similarly, you can directly use the CLI to write the value to a particular register address of Compass by

using `i2c write SLAVE_ADDRESS REGISTER_ADDRESS VALUE`

```
List of commands:
     taskcontrol: Control your task(Suspend or Resume).
           crash: Deliberately crashes the system to demonstrate how to debug a crash
             i2c: 'i2c read 0xDD 0xRR <n>' OR 'i2c write 0xDD 0xRR <value> <value> ...'
       tasklist: Outputs list of RTOS tasks, CPU and stack usage. 'tasklist <time>' will
display CPU utilization within this time window.
--------------------------------------------------------------------------------
i2c write 0x38 0x2A 9
Wrote 1 bytes to slave 0x38
  [  0] = 0x09 (9)
--------------------------------------------------------------------------------
i2c read 0x38 0x2A 1
I2C Read of Slave 0x38
  0x00: 0x09 (9)
--------------------------------------------------------------------------------
```

Transmit GPS coordinates in between controllers

Use the following DBC design:

```
BO_ 201 GPS_DESTINATION: 4 BRIDGE_CONTROLLER
 SG_ GPS_DESTINATION_LONGITUDE : 0|32@1+ (0.000001,0) [0|0] "" GEO_CONTROLLER
```

Be sure to keep in mind:

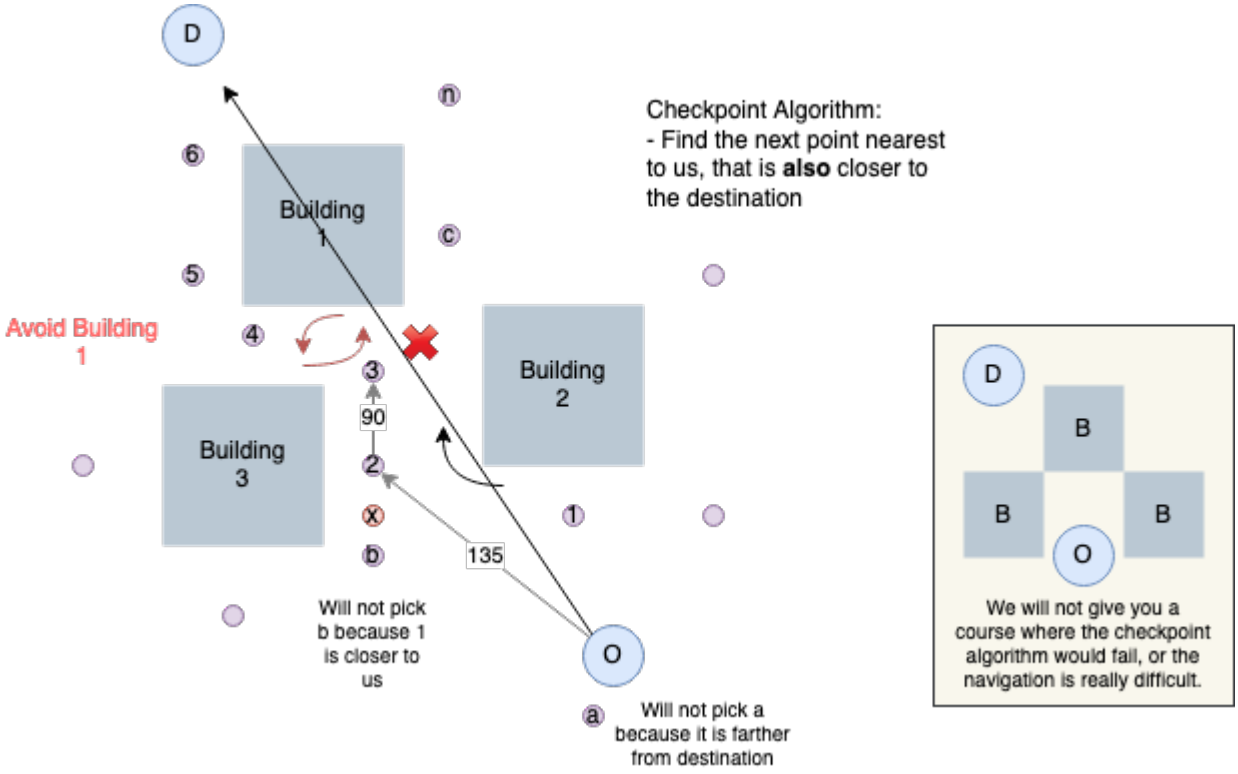
- A GPS coordinate is usually only sent with six decimal places maximum
- A `float`, is always a 32-bit float (IEEE standard)
- A `float` can only store up to 6 decimal points
  - However, if your number changes from `0.123456` to `456.123456`, then the precision is actually lost, and the number may actually be truncated to this: `456.123444`

More things to keep in mind:

- float is supported by HW on the ARM CM-4
- But a "double" uses software floating-point instructions

Checkpoints Algorithm

The algorithm should not be overcomplicated, and the checkpoints can just be static const array in your code.



Keep in mind:

- Mobile application should only send the destination, and not these checkpoints
- The Geo controller has the pre-mapped, constant points in the compiled ARM CPU code

```
// Probably already defined in your *.h file
typedef struct {
  float long;
  float lat;
} gps_coordinates_t;
// Define in your *.c file
static const gps_coordinates_t locations_we_can_travel[] = {
  {a, b},
  {c, d},
  {e, f},
```

```
  {g, h},
}
/**
 * Algorithm should iterate through all locations_we_can_travel[] and find:
 *  - Another point that is closest to origin
 *  - while also simultaneously closer to the destination at the same time
 *
 * Corner case: If next point is the destination itself, which is also possible
 *.             and in this case, you should flag that destination has been reached
 */gps_coordinates_t find_next_point(gps_coordinates_t origin, gps_coordinates_t destination);
```

Unit Testing code that touches the HW registers

This article guides you on how to unit-test code that reads or writes hardware registers of your SJ development board.

```
// Typical code
int get_ultrasonic_pulse_width(void) {
  // Send a pulse width
  LPC_GPIO1->CLR = (1 << 2);
  delay_us(10);
  LPC_GPIO1->SET = (1 << 2);

  const uint32_t previous = time_now();
  while (LPC_GPIO1->PIN & (1 << 3)) {
   ;
  }

  return time_delta(previous);}
```

Before we solve the problem, let us write better code that is self expressive and does not require comments to understand its intent.

```
static void send_pulse_to_ultrasonic(void) {
  const uint32_t ultrasonic_pulse_pin = (1 << 2);
  LPC_GPIO1->CLR = ultrasonic_pulse_pin;
  delay_us(10);
  LPC_GPIO1->SET = ultrasonic_pulse_pin;
```

```
}
static void wait_for_ultrasonic_pulse_to_bounce_back() {
  while (LPC_GPIO1->PIN & (1 << 3)) {
   ;
  }
}
// Notice the clarity of this function compared to the previous code snippet
int get_ultrasonic_pulse_width(void) {
  send_pulse_to_ultrasonic();

  const uint32_t previous = time_now();
  wait_for_ultrasonic_pulse_to_bounce_back();
  return time_delta(previous);}
```

And the next level:

```
// Separate header file to abstract the hardware, such that we can mock out this API
// file: ultrasonic_pins.h
void ultrasonic_pins__set_pulse(bool true_for_logic_high);
bool ultrasonic_pins__get_input_pin_value(void);
```

```
#include "ultrasonic_pins.h"
static void send_pulse_to_ultrasonic(void) {
  // This can now move to ultrasonic_pins.c
  // const uint32_t ultrasonic_pulse_pin = (1 << 2);
  ultrasonic_pins__set_pulse(true);
  delay_us(10);
  ultrasonic_pins__set_pulse(false);
}
static void wait_for_ultrasonic_pulse_to_bounce_back() {
  while (ultrasonic_pins__get_input_pin_value()) {
   ;
  }
}
int get_ultrasonic_pulse_width(void) {
```

```
  send_pulse_to_ultrasonic();

  const uint32_t previous = time_now();
  wait_for_ultrasonic_pulse_to_bounce_back();
  return time_delta(previous);}
```

Use single periodic callback if possible

The problem with multiple callbacks is that the higher rate can interrupt a lower rate callback.

```
int number = 0;
void periodic_callback_10hz(uint32_t count) {
  number++;
}
// This can interrupt the 10hz in the middle of its operations
void periodic_callback_100hz(uint32_t count) {
  number++;
}
```

Use this instead:

```
int number = 0;
void periodic_callback_10hz(uint32_t count) {
  //number++;
}
// This can interrupt the 10hz in the middle of its operations
void periodic_callback_100hz(uint32_t count) {
  if (0 == (count % 10)) {
    number++;
  }
  number++;
}
```

CANTools

CANTools is a Python project that can read DBC files, and provide a lot of useful information.

https://pypi.org/project/cantools/

Exploring DBC Autogenerated API

DBC Encode API

```
void can_transmitter_option1(void) {
  dbc_DRIVER_HEARTBEAT_s heartbeat={};
  dbc_SENSOR_SONARS_s sensor_values={};
  dbc_message_header_t header;
  can__msg_t can_msg = {};
  header = dbc_encode_SENSOR_SONARS(can_msg.data.bytes, &sensor_values);
  can_msg.msg_id = header.message_id;
  can_msg.frame_fields.data_len = header.message_dlc;
  can__tx(can1, &can_msg, 0);
  header = dbc_encode_DRIVER_HEARTBEAT(can_msg.data.bytes, &heartbeat);
  can_msg.msg_id = header.message_id;
  can_msg.frame_fields.data_len = header.message_dlc;
  can__tx(can1, &can_msg, 0);}
```

DBC Encode and Send API

```
void can_transmitter_option2(void) {
  dbc_DRIVER_HEARTBEAT_s heartbeat={};
  dbc_SENSOR_SONARS_s sensor_values={};
  void *unused = NULL;
  dbc_encode_and_send_DRIVER_HEARTBEAT(unused, &sensor_values);
  dbc_encode_and_send_DRIVER_HEARTBEAT(unused, &heartbeat);
```

```
}
bool dbc_send_can_message(void * argument_from_dbc_encode_and_send, uint32_t message_id, const uint8_t
  can__msg_t can_msg = {};

  can_msg.msg_id = message_id;
  can_msg.frame_fields.data_len = dlc;
  memcpy(can_msg.data.bytes, bytes, sizeof(can_msg.data.bytes));

  can__tx(can1, &can_msg, 0);}
```