# Figure table

DIAGRAMS

[Diagram 1: Sensor Data Struct 3](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140160)

[Diagram 2: Carbon Handler Design Class Diagram 3](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140161)

[Diagram 3: LoraWAN Helper Design Class Diagram 4](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140162)

CODE EXAMPLES

[Code Example 1: Data Repository Initialization 5](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140167)

[Code Example 2: Fluctuating delay 6](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140168)

[Code Example 3: Movement handler initialization 6](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140169)

[Code Example 4: Movement task function 6](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140170)

[Code Example 5: Temperature and Humidity sensor warm up 7](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140171)

[Code Example 6: CO2 callback function 7](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140172)

[Code Example 7: Light callback function 8](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140173)

[Code Example 8: LoraWAN helper initialization function 8](file:///C:\Users\PC\Documents\Embedded_C_Design.docx#_Toc8140174)

# Design

## Embedded Design:

During the design phase of the project, various design-oriented classes were created based on the conceptual classes as well as system sequence diagrams that were defined throughout the course of the analysis phase. Due to the low-level nature of the system, it was difficult to decide on a single architecture for the entire system, in the end it was decided to utilize a microservice-based architecture, the decision was made due to the fact that the non-functional requirements could be separated into three different services:

1. Data-Repository service;
   1. *The data repository service was later further separated into four different data-handlers to make the system more maintainable and to avoid having classes that employ the god-object anti-pattern, the handlers that were created were as follows:*
      1. CO2 Handler,
      2. Temperature and Humidity handler,
      3. Light handler,
      4. Movement handler.
2. LoraWAN Helper service;
   1. *The main responsibility of this service is to handle data send-off as well retrieval-of the data that is sent to the device through loraWAN. Further details of the implementation will be discussed in the Implementation part of this document.*
3. Event Reactor service;
   1. *The purpose of this service is to handle such tasks as enabling the lights when movement is detected based on the preferences of the user which are handled by the loraWAN helper service.*

Additionally, it was decided to utilize FreeRTOS to handle these services to improve the efficiency of the system, this decision allowed scheduling of various parts of the system, to save battery-life the data-repository service performs the measurements every minute while the LoraWAN would send out the average every 5 minutes.

Furthermore, due to the requirements the device code was written in C programming language, while the bridge-app that communicates with the device was written in Java programming language.

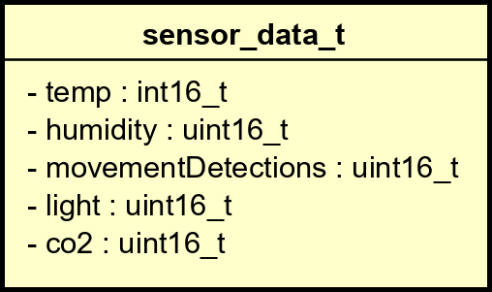
Next, while designing the data repository service it was decided to utilize dependency injection by passing a pointer to the struct which contains most of the necessary information in uint16 format, it was decided to use this format to save memory space as the readings would never reach a higher value than 65535 and lower value than 0, however due to the fact that temperature is measured in Celsius and not Kelvin the temperature reading could reach a negative value, thus a signed version of uint16 was used instead, the reason why int16 was used instead of int8 for temperature, humidity, light and movement even though the sensors are not able to return a value larger than the max value of int8 is because the system calculates the average, thus adding 5 readings together produces a value outside of the limit of int8, however LoraWAN helper service sends out an average instead of the sum thus the information that is sent out is in int8 format due to the limitation of the driver. In addition to the pointer to the struct, a pointer to a semaphore is also passed to data repository service initializer to ensure data safety.

Diagram 1: Sensor Data Struct

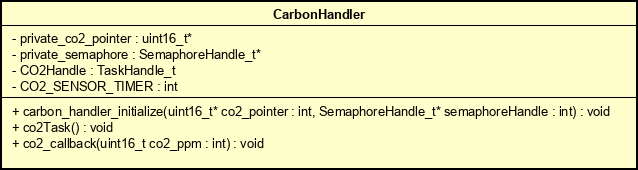
As mentioned previously, the data repository service was separated into five handlers that are initialized by the data repository, this improves the separation of concerns of the system as each of the handlers deal with one specific type of data. All handlers follow the same general structure:

Diagram 2: Carbon Handler Design Class Diagram

1. An initialization function:
   1. *This function receives a pointer to a specific value of the sensor data as well as a pointer to a semaphore which ensures data safety, these pointers are saved in a static variable that is only accessible by the class itself. The reason why it is static is the fact that there is only a single instance of each handler during runtime.*
2. A task function:
   1. *This function is continuously executed by a FreeRTOS task.*
3. Optional: A callback function:
   1. *This function is called by the driver.*

In addition to this structure, due to the use of FreeRTOS, each handler contains a TaskHandle, pointer to a semaphore as well as a specific timer which determines the interval at which the task should be executed. During this phase it was decided that the interval at which the tasks would execute would be 1 minute for every sensor apart from movement, as it would not make much sense to gather data every minute, the task that collects movement data executes every 1 second instead. To make sure that other sensors can perform readings as well, movement task has the lowest priority, while the LoraWAN helper has the highest priority, this ensures that even if something goes wrong within one of the tasks and the semaphore is not given away, the LoraWAN helper can still perform its task every 5 minutes.

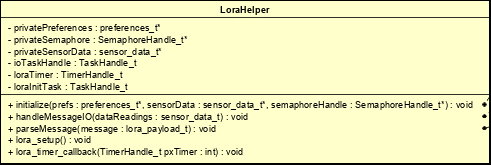
 Furthermore, the LoraWAN helper retrieves a pointer to the preference, semaphore as well as the aforementioned struct, the reason why it retrieves a pointer to the preference is so that when a message is retrieved it can immediately modify the preferences. The preferences….. . The same pointer to the semaphore allows FreeRTOS to easily control the flow of the system thanks to the task priorities. In addition, it was decided to include a timer which would delay the LoraWAN Helper at startup so that the sensors would have enough time to collect data for the initial send-off. For more information on how exactly do the functions work please refer to the Implementation:Embedded C section of this document.

Diagram 3: LoraWAN Helper Design Class Diagram

In order to connect the bridge-app and the actual device, a very simple 6-byte long data protocol was designed, the first byte of this protocol contains the low byte of the CO2 reading, this is due to limitation in the driver which only allows 8-bit values and the CO2 reading being above the 8-bit threshold, while the second byte contains the high byte of the CO2 reading, the following 4 bytes contain temperature, humidity, light and movement respectively which are all 8-bit long, as security was not a concern at this part of the entire project, no further steps were taken to encrypt the data. With design choices finalized, the project was able proceed with the implementation phase.

# Embedded C: Implementation

During the implementation phase the diagrams, which were created during the design phase, were materialized(?). Due to the size of the project, in this section only the main concepts will be described, the interested reader can refer to <https://www.freertos.org/> to read more about FreeRTOS as well as <http://havns.dk/IoT/> to learn more about how exactly does each sensor work.

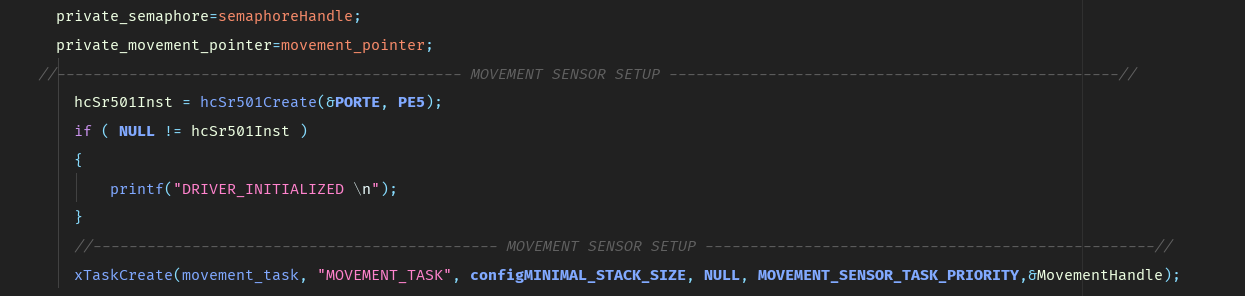
One thing to note about the execution of each of the tasks is that they are delayed using vTaskDelayUntil function, this allows the system to be executed in specific time intervals ignoring what actions does the task perform itself, unlike vTaskDelay function which adds a specific amount of delay on top of the amount of time taken to perform the task’s actions.

The main functionality of the system was the data collection, as mentioned in the design section, the C aspect of the project follows the microservice architecture and utilizes repository design pattern to initialize the data collector, it does this by calling a publicly available method initialize\_repository, this method takes in a pointer to a sensor data struct as well as a pointer to the semaphore which is passed along to the specific data handlers. To keep the system maintainable, each sensor has its own data handler which is initialized from the Data Repository class. In addition, each initializer method receives a pointer to a specific sensor-data location, this is done to maintain the separation of concerns and to prevent different data handlers from modifying data that they should not be able to modify, however before they are sent out, data repository ensures that the default values are at 0 by assigning all readings a value of 0, this is done due to the fact that these variables end up storing random values that were present in the memory when the space was allocated by malloc function.

Code Example 1: Data Repository Initialization

Further, due to power-efficiency concerns FreeRTOS tasks were used to toggle sensors between OFF and ON, however because of the decision to use it as the kernel of the device, timing and priorities had to be considered to ensure proper flow of the system, thus every handler have their own priority macro and another one that acts as the time period to wait until performing the task again, the timer is calculated by dividing 954 by the tick rate which is accessed from the portmacro.h header file and then multiplying it be the amount of seconds that the timer should be running for, without multiplying the calculation returns the numbers of ticks required for 1 second, the reason why 954 is divided instead of 1000 is due to the fluctuating crystal oscillator frequencies in the atmega2560 MCU, unfortunately even with modified calculations the amount of delay fluctuates by approximately 50 milliseconds in the absolute worst case, although the average is much lower at approx. 5ms (as tested on two different atmega2560 MCU boards).

Code Example 2: Fluctuating delay

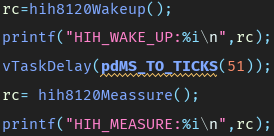
Next, as mentioned previously due to the nature of the sensor, the MovementHandler task had the lowest priority available to ensure that if movement is detected and some action is performed by the EventReactor, other sensors could still collect data. Further, the initialization method of the MovementHandler takes care of the driver initialization by calling the hcSr501Create function and passing along the port and the pin that the HC-SR501 sensor is connected to, moreover this function also creates the task that performs the actual reading by passing a reference to the function that the task should perform, name, stack configuration, function parameters, priority macro as well an address of the task handle, which is assigned to NULL by default. Once the task is created and the task scheduler started (which is started after initializing every service in the main function), the function movement\_task is then continuously executed every 1 second or every 35 seconds if movement is detected, the reason why there is a 35 second waiting period after a detection is to prevent false detections due to the fact that generally the sensor is checked in such a small period of time. Once the sensor is triggered, the function toggeLights is called twice with a delay in between which determines how long the light should stay turned on which, the light is toggled by moving the servo 90 degrees, waiting a second due to technical limitations of the servo and then moving it back to 0 degrees. … SERVO IMPLEMENTATION DISCUSSION GOES HERE …

Code Example 3: Movement handler initialization

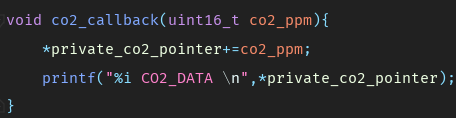
Code Example 4: Movement task function

In order to avoid repetition, every other initialization method will be excluded from this document as they follow the same structure:

1. Initialize private variables to store reading data,
2. Initialize the sensor driver,
3. Create the task.

Once the temperature and humidity handler is initialized, the function th\_task() continuously executes every 1 minute. In this function, the sensor is first woken up and due to technical limitations, the entire task is delayed by 51ms to allow the sensor to warm up before performing a measurement. Once the measurement is taken, functions hih8120GetTemperature\_x10() and hih8120GetHumidityPercent\_x10() retrieve the actual readings which are then divided by 10 and added to the humidity and temperature values of the sensor data struct, it was decided to ignore the decimal point of the readings because of the low impact that they provide.

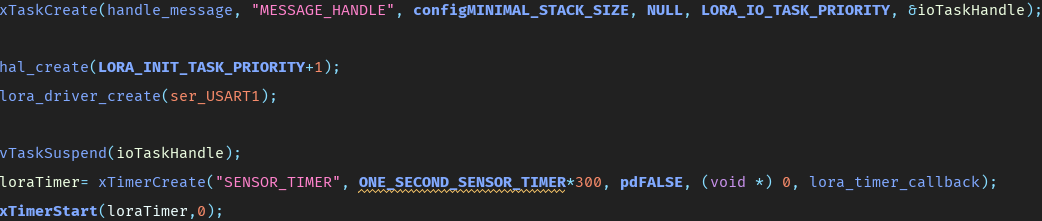
Code Example 5: Temperature and Humidity sensor warm up

 Further, the third data handler that is initialized is the CO2, the main difference between this initializer and the ones mentioned in previous paragraphs is the CO2 driver initialization, unlike most other drivers, this driver requires a reference to a function which is called as a callback whenever a measurement is performed by the sensor, in this case once the callback is executed the ppm is added to the value of co2\_ppm in the sensor data struct. once the sensor is initialized the CO2 task is executed continuously every minute.

Code Example 6: CO2 callback function

The final data handler that is initialized by the data repository is the light data handler, this handler shares similar characteristics to the CO2 handler as the light sensor driver also requires a callback, however this callback is executed more often, while the CO2 callback is executed only on measurement, the light callback is executed on enable, fetching as well as disabling, more importantly due to shared TWI driver, light sensor has to be disabled while temperature and humidity measurements are being taken and vice-versa. The callback is quite simple, it consists of a variable to store the light reading (whose address is passed to the tsl2591GetVisibleRaw function) and a switch case to determine the return code, when the return code is TSL2591\_DATA\_READY the value of the reading is added to the value of raw light of the sensor data struct. As with the CO2 and Temperature tasks, the light task is executed every minute during which the sensor is enabled, data is fetched, and it is disabled once again.

Code Example 7: Light callback function

 In addition to the data repository service, another core service of the system is the LoraWAN helper service, this service takes care of the communication between the java bridge application. This service takes in a pointer to a semaphore in order to get access to the sensor data struct for send-off, the pointer to said struct as well as a pointer to the preference struct for the downlink message handling. During the initialization process the structure is quite similar to the handler initialization, however one key difference is that the lora helper service task is suspended during the initialization process, this is done to allow the sensor handlers to gather data before connecting and uploading data onto the LoraWAN server. The task is started again once a timer callback is called, the timer in question is also initialized and started in the initialization function of the loraWAN helper, this timer is set to execute once after 300 seconds (5 minutes). Once the task is started, function handle\_message is executed continuously every 5 minutes and 1 second (the 1 second is given for data handlers to finish their 5th readings). During the first execution of this function, lora\_setup is called to setup the connection. The function performs the following actions in the following order:

Code Example 8: LoraWAN helper initialization function

1. Resets the driver
   1. *It does this by calling lora\_driver\_reset\_rn2483 twice, first by passing the value of 1 and then by passing the value 0.*
2. Reboots the entire sensor and restores all parameters to their default values
   1. *This action is performed by calling lora\_driver\_rn2483\_factory\_reset.*
3. Assigns the frequency, data rate, channel, channel frequency, output power
   1. *This action is performed by calling lora\_driver\_configure\_to\_eu868. (for more information regarding the values please refer to:*<http://havns.dk/IoT/group__lora__basic__function.html#gad689bb72ab6e665ab420342c16912b82>)
4. Retrieves the HWEUI of the module
   1. *This action is performed by calling lora\_driver\_get\_rn2483\_hweui and passing an array of char which is used to store the hweui.*
5. Sets the OTAA identity
   1. *This action is performed by calling lora\_driver\_set\_otaa\_identity and passing the HWEUI of the device as well as appEUI and appKEY which are defined as macros in the LoraHelper.c file (40f0e6960ec746d8 and 234cc8845c2087ff6a56deb1f9c1d5b4 respectively)*
6. Joins the LoraWAN network
   1. *This action is performed by calling lora\_driver\_join function and passing LoRa\_OTAA enum, in addition this is put in a for loop and iterated through 4 times.*

In order to prevent battery waste, a switch-case statement is used to prevent pointless attempts to send data when loraWAN network is not joined, this switch case terminates the task if the LoraWAN network has no empty channel available, additionally the task is also terminated if the connection could not be established in 4 attempts. Once the loraWAN connection is established, the data is divided by the amount of minutes that the loraWAN service is delayed for (this is done to send out an average of the readings) and then loaded into a lora\_payload\_t struct (which consists of an array of bytes) according to the protocol that was designed during the design phase, due to the fact that this array only allows values of 1 byte per index, the CO2 value is separated into two before it is loaded into the array, this is done by using the AND bitwise operator to mask the lower 8 bits of the uint16 value and then right-shifting the higher 8 bits. Once the data is loaded into the struct, lora\_driver\_sent\_upload\_message is called in order to upload the struct to lorawan server, when the payload is uploaded to the server the driver also checks if anything was sent to the device, if so parseMessage function is executed which parses the downlink message. Once the uplink/downlink handling is completed, the sensor data struct is reset back to 0.