Index

Core Functionality Layer

HMI

Digital Control

Communications

High-Level Hardware Abstraction Layer

This layer of software is the only direct interface that the program has with the hardware and so it is meant to facilitate development in the layers that go above it and make it easier to port the code into a physical platform, since these are ideally the only places where it would need to be changed.

Global

Enumerations

> **`Ports`**

Hardware port enumeration. This exists mainly for use by the GPIO abstraction modules - Sensor and Actuator.

```
PORT_A, PORT_B, PORT_C, PORT_D, PORT_E, PORT_F
```

> **`Pins`**

Hardware pin enumeration.

```
PIN_0, PIN_1, PIN_2, PIN_3, PIN_4, PIN_5, PIN_6, PIN_7,
PIN_8, PIN_9, PIN_10, PIN_11, PIN_12, PIN_13, PIN_14, PIN_15
```

> **`UpdateMode`**

Conversion update mode, primarily for use by the GPIO abstraction modules - Sensor and Actuator.

```
UPD_MODE_CONTINUOUS, UPD_MODE_SINGLE_SHOT
```

`UPD_MODE_CONTINUOUS`: Signal for conversion is given once and all future conversions happen repeatedly, without user intervention, separated by a previously established sampling period.
`UPD_MODE_SINGLE_SHOT`: Only one conversion happens at the given signal.

For each mode the user should be able to prompt the module to keep the converted values in a list and generate events at the end of each conversion to be able to react promptly upon it.

Communications Redundancy Engine

Sensor

Enumerations

> **PulseCountingPolarity**

```
PCP_CNT_RISING, PCP_CNT_FALLING_, PCP_CNT_RISING_AND_FALLING
```

- PCP_CNT_RISING: Counts rising edge pulses;
- PCP_CNT_FALLING: Counts falling edge pulses;
- PCP_CNT_RISING_AND_FALLING: Counts rising and falling edge pulses.

> **SensorErrors**

Error values to be set when the user requests an action that cannot be performed.

```
E_OK, E_MUTED, E_NOT_DELETED, E_NOT_CONFIGURED, E_RUNNING, E_NOT_RUNNING,
E_WRONG_MODE, E_CNV_NOT_AVAILABLE, E_INVALID_INTERFACE_CONFIG,
E_BAD_PARAMETER
```

- E_OK: Operation was successful
- E_MUTED: The entity cannot tend to the user's request because the module is mutex-locking a static attribute.
- E_NOT_DELETED: The entity has not been deleted yet;
- E_NOT_CONFIGURED: The entity is not yet configured to perform any task;
- E_RUNNING: The entity is already performing a specific task and it cannot be reconfigured until it stops;
- E_NOT_RUNNING: The entity is not executing any task at the moment;
- E_WRONG_MODE: The user called for functionality not destined to run in the chosen operation mode;
- E_CNV_NOT_AVAILABLE: A certain conversion has not yet been made and thus its value is not available for retrieval;
- E_INVALID_INTERFACE_CONFIG: An invalid pin configuration has been chosen.
- E_BAD_PARAMETER: A parameter with an invalid value has been passed onto a method;

> **SensorModes**

```
SNS_MODE_PULSE_COUNTING, SNS_MODE_ANALOG, SNS_MODE_DIGITAL
```

- SNS_MODE_PULSE_COUNTING: Pin used as an input for a timer used as a counter ;
- SNS_MODE_ANALOG: Pin used for analogue input;
- SNS_MODE_DIGITAL: Pin used for digital input.

Structures

> `SensorConfig`

Structure for declaring and keeping all aspects related to the configuration of a Sensor entity.

Attributes:

- `port: Ports`
  Selected port for this entity.
- `pin: Pins`
  Selected pin for this entity.
- `mode: SensorModes`
  Chosen operation mode for this entity.
- `sampling_period: uint32`
  Period elapsed between each conversion in Continuous Conversion Mode.
- `update_mode: UpdateModes`
  Chosen Conversion Mode.
- `conversion_complete_callback: (uint32) -> void` A function to be called when a conversion or reading is completed. Should be left NULL if it is to be ignored.
- `pulse_counting_polarity: PulseCountingPolarity`
  Relevant only for Pulse Counting sensor mode. Defines whether the sensor counts rising edge pulses, falling edge pulses or both.

> `SensorState`

- `created: boolean`
  Flag that signals whether or not the entity has been created.
- `configured: boolean`
  Flag that signals whether or not the entity has been configured.
- `running: boolean`
  Flag that signals whether or not the entity is executing its task.

Attributes: Static

> `sensor_objects: Sensor*[]`

List of pointers to all currently existing objects of this type.

Objects are added to this list when created and removed when deleted. This exists to facilitate resource management between said objects.

> `sensor_static_mutex: OSMutex`

OS-specific mutex object that is meant to lock read or write access to sensitive module-wide static attributes.

Attributes: Instance

> `config: SensorConfig`

Structure holding the current configuration values for its entity.

> `state: SensorState`

Structure holding the current state values for its entity.

> `input_buffer: CircularBuffer`

Circular buffer that allows for storage and retrieval of past conversion values.
It is meant to be written to by the Sensor module exclusively and automatically upon finishing a conversion The retrieval of data from it should also be intermediated by this module.
It is most useful in Continuous Conversion Mode, but allowed for use in Single Shot Conversion Mode as well.

> `last_error: SensorErrors`

Error code for the last occurring error when accessing the entity's methods.

Methods

> **`create(`***`input_buffer: CircularBuffer*`***`) -> boolean`**

- ***brief***

  Initialization of `input_buffer` and other critical parameters to the "reset" state and adds the current object to the `sensor_objects` list.

- ***return***

  - `TRUE` and `last_error = E_OK` if the creation was successful;
  - `FALSE` and
    - `last_error = E_MUTED` if the module is mutex-locking a static attribute that needs to be accessed or modified during configuration. It doesn't signal an error so to speak, but a brief restriction;
    - `last_error = E_RUNNING` if the entity is already performing a specific task. The solution is to call `stop()` on this entity to stop its execution, `delete()` to delete it and call `create()` again to try creating again;
    - `last_error = E_NOT_DELETED` hasn't yet been deleted. The solution is to call `delete()` on this entity to delete it and call `create()` again to try creating again;
    - `last_error = E_BAD_PARAMETER` if one or more of the provided buffer's attributes is invalid (e.g. its size os too large or its data pointer is null).

> **`configure(`***`config: SensorConfig*`***`) -> boolean`**

- ***brief***

  Entity configuration method. Every aspect of the configuration should be handled at once, from here.

- ***parameters***
    - `config`

      SensorConfig structure that contains instructions on how the entity should be configured to operate.
- ***return***
    - `TRUE` and `last_error = E_OK` if the configuration was successful;
    - `FALSE` and
        - `last_error = E_INVALID_INTERFACE_CONFIG` if the chosen port and pin combination is completely unavailable or cannot be assigned that specific task;
        - `last_error = E_MUTED` if the module is mutex-locking a static attribute that needs to be accessed or modified during configuration. It doesn't signal an error so to speak, but a brief restriction;
        - `last_error = E_RUNNING` if the entity is already performing a specific task. The solution is to call `stop()` on this entity to stop its execution and call `configure()` again to try configuring again;
        - `last_error = E_BAD_PARAMETER` if an otherwise invalid parameter has been passed onto the function.

> **`start() -> boolean`**

- ***brief***

  Start executing the configured task.
- ***return***
    - `TRUE` and `last_error = E_OK` if the configuration was successful;
    - `FALSE` and `last_error = E_NOT_CONFIGURED` if the entity is not yet configured to perform any task;

> **`fetch_value(`*`index: uint32`*`) -> uint32`**

- ***brief***

  Fetch the conversion value `index` values behind the last one.
- ***return***
    - The conversion value `index` values behind the last one.

> **`stop() -> boolean`**

- ***brief***

  Stop executing the running task.
- ***return***
    - `TRUE` and `last_error = E_OK` if the configuration was successful;

    - `FALSE` and `last_error = E_NOT_RUNNING` if the entity was not executing any task in the first place. This isn't an error as much as it is a warning.

> **`delete() -> boolean`**

- ***brief***

  Remove object from `sensor_objects`.

- ***return***

    - `TRUE` and `last_error = E_OK` if the deletion was successful;
    - `FALSE` and
        - `last_error = E_RUNNING` if the entity is performing a specific task. The solution is to call `stop()` on this entity to stop its execution and calling `delete()` again to try configuring again;
        - `last_error = E_MUTED` if the module is mutex-locking a static attribute that needs to be accessed or modified during deletion. It doesn't signal an error so to speak, but a brief restriction.

Events

> **`event_conversion_complete(value: uint32)`**

`[Empty]`

Typical Use Case

```
CircularBuffer buffer = ...
Sensor rpm_sensor;
// Configure for using pin 0 from port A, in pulse counting mode on the
// rising edge, with a sampling period of 1 ms and no conversion complete
// callback
SensorConfig rpm_sensor_config = {
    .port = PORT_A,
    .pin = PIN_0,
    .mode = SNS_MODE_PULSE_COUNTING,
    .pulse_counting_polarity = PCP_RISING_EDGE
    .sampling_period = 1000,
    .conversion_complete_callback = NULL,
}

// Create the sensor
sensor_create(&rpm_sensor, &buffer);
// Configure task parameters
sensor_configure(&rpm_sensor, &rpm_sensor_config);
// Set running
sensor_start(&rpm_sensor);

...

// fetch 5th value from the last one back
sensor_fetch(&rpm_sensor, 5);
```

```
// Configure the sensor again without stopping it. This is not permitted.
if (sensor_configure(&rpm_sensor, &rpm_sensor_config) != E_OK) {
    // Identify the error and treat it
    if(rpm_sensor.last_error == E_RUNNING) {
        ...
    }
    ...
}
// Stop the sensor
sensor_stop(&rpm_sensor);
// Start the sensor again. This is permitted.
sensor_start(&rpm_sensor);
// Stop the sensor again. This is permitted.
sensor_stop(&rpm_sensor);
// Delete the sensor
sensor_delete(&rpm_sensor);
```

Actuator

Structures

> **ActuatorConfig**

Structure for declaring and keeping all aspects related to the configuration of a Actuator entity.

Attributes:

[EMPTY]

> **ActuatorState**

- `configured:` **boolean**
  Flag that signals whether or not the entity has been configured.
- `running:` **boolean** Flag that signals whether or not the entity is executing its task.

Enumerations

> **ActuatorErrors**

[EMPTY]

> **ActuatorModes**

ACT_MODE_PWM, ACT_MODE_ANALOG, ACT_MODE_DIGITAL

[EMPTY]

Attributes: Static

> `actuator_objects: Actuator*[]`

List of pointers to all currently existing objects of this type.

Objects are added to this list when created and removed when deleted. This exists facilitate resource management between said objects.

Attributes: Instance

> `port: Ports`

> `pin: Pins`

> `mode: ActuatorModes`

> `output_buffer: CircularBuffer`

> `update_period: uint32`

> `update_mode: UpdateModes`

> `last_error: ActuatorErrors`

Methods

> **`[constructor] create() -> void`**

> **`configure_interface(`*`port: Ports, pin: Pins, mode: ActuatorModes`*`) -> boolean`**

> **`configure_mode(`*`update_mode: UpdateMode, update_period: uint32`*`) -> boolean`**

> **`start() -> boolean`**

> **`update(`*`value: uint32`*`) -> boolean`**

> **`pause() -> boolean`**

> **`[destructor] delete() -> void`**

Events

`event_conversion_complete(`*`value: uint32`***`)`**

Typical Use Case

Bluetooth

> **NOTE**: *The design of the Bluetooth module is being briefly postponed due to uncertainty on whether the program will be running on a simulation of a microcontroller or on a normal*

> *computer program, as this is relevant information for the case.*

Enumerations

Structures

Attributes: Static

Attributes: Instance

Methods

Events

Typical Use Case

Serial

Enumerations

> `SerialErrors`

> `Parity`

> `StopBits`

> `BufferTypes`

Structures

Attributes: Static

> `serial_objects: Serial*[]`

Attributes: Instance

> `config: SerialConfig`

> `state: SerialState`

> `output_buffer: CircularBuffer`

> `input_buffer: CircularBuffer`

> `last_error: SerialErrors`

Methods

> **`create() -> boolean**

> **`config_port`**`(port: char[], baudrate: uint32, stop_bits: StopBits, parity: Parity)` **`-> boolean`**

> **config_reception**(*terminating_sequence: char*, callback: (void) -> void*) -> **boolean**

> **config_sending**(*terminating_sequence: char*, callback: (void) -> void*) -> **boolean**

> **start**() -> **boolean**

> **pause**() -> **boolean**

> **send**(*buffer: char*, length: uint32*) -> **uint32**

> **flush**(*buffer_identifier: BufferTypes*) -> **boolean**

Events

> **event_msg_received**(*message: char[]*)

> **event_msg_sent**(*message: char[]*)

Typical Use Case