# CMOS

Cortex-M Operating System

# Table of Contents


# 1 General Concept

CMOS is an Operating System written in plain C++ aiming at simplifying programming of the Cortex-M Processors. It consists of several Packages being pre-compiled for the specific Cores. These Packages are:

- Kernel Library
- Graphics Library
- Filesystem Library
- ICD (**I**ntegrated **C**ircuit **D**river) Library

The Kernel Library is the Core of CMOS, being responsible for:

- Scheduling of Threads
- Dynamic Memory Management (Heap)
- Static Memory Management (Stack)
- Semaphore Management

The Graphics Library provides a simple-to-use Framework to create basic GUIs, while the Filesystem Library implements an abstracted Filesystem, being able to mount any Kind of Memory that complies with the Interface Requirement. Finally, the ICD Library provides the Implementation of Drivers for specific ICs to the User.

CMOS can be downloaded here:

https://github.com/leonuetzel/CMOS

To make the Operating System independent from the specific MCU it is embedded in, CMOS has some Requirements to be met in order to be executable.

# 2 Requirements

Using the corresponding Device Driver Library already implements the Requirements below.
This Library can be found here:

https://github.com/leonuetzel/DeviceDriver

## 2.1 Linker Script Symbols

The following Symbols need to be defined in the Linker Script:

| Symbol | Description |
|---|---|
| __cmos_heap_start__ | Heap Start Address |
| __cmos_heap_size__ | Heap Size in Bytes |
| __cmos_heap_blockSize__ | Heap Block Size in Bytes |
| __cmos_heap_blockNumber__ | Number of Blocks of Heap Memory (can be calculated from above Values) |
| __cmos_heap_blockOwner__ | Address of an empty Array of "__cmos_heap_blockNumber__ " Number of Bytes |
| __cmos_stack_start__ | Stack Start Address |
| __cmos_stack_size__ | Stack Size in Bytes |
| __cmos_stack_blockSize__ | Stack Block Size in Bytes |
| __cmos_stack_blockNumber__ | Number of Blocks of Stack Memory (can be calculated from above Values) |
| __cmos_stack_blockOwner__ | Address of an empty Array of "__cmos_stack_blockNumber__ " Number of Bytes |
| __cmos_mainStackSize__ | Stack Size of the *main* Function |
| __cmos_initial_stackpointer__ | Value that Stackpointer should be initialized with at Reset |
| __cmos_numberOfThreads__ | Maximum Number of Threads |

## 2.2 Exceptions

CMOS defines the Vector Table for the specific Cores. All Cortex-M cores have multiple Exceptions in common, as well as the Reset Exception from which the Processor starts to execute Code. The User has to write Code for this Exception and needs to initialize CMOS:

```
1. // Start OS
2. CMOS& cmos = CMOS::get();
3. cmos.run();
4.
5.
6. //  Reset if System ever returns
7. cmos.reset();
```

## 2.3 Other Preparations

The User needs to copy Code and non-zero initialized Data that is stored in/executed from the SRAM at Runtime from the Flash Memory to the SRAM at Startup. CMOS suggests placing these Code Segments in Linker Input Sections starting with ".code_ram". To make the Placement more readable, the Define "CODE_RAM" has been made in "defines.hpp". Usage:

```
1. CODE_RAM uint8 TestClass::testFunction()
2. {
3.
4. }
```

Also, global static Variables and Objects need to be initialized. Therefore, the Compiler provides a Linker Input Section called ".init_array" that is simply the Start Address of an Array of Function Pointers, that need to be called in ascending Order.

CMOS suggests defining the following Symbols to be used by the Startup Code in the Linker Script:

| Symbol | Description |
|---|---|
| __cmos_code_flash_start__ | Start Address in Flash Memory of Code, that is stored in Flash Memory, but will be executed from SRAM at Runtime |
| __cmos_code_flash_size__ | Size of Code in Bytes, that is stored in Flash Memory, but will be executed from SRAM at Runtime |
| __cmos_code_ram_start__ | Start Address in SRAM of Code, that is stored in Flash Memory, but will be executed from SRAM at Runtime |
| __cmos_data_flash_start__ | Start Address in Flash Memory of Data, that is stored in Flash Memory, but will be stored in SRAM at Runtime |
| __cmos_data_flash_size__ | Size of Data in Bytes, that is stored in Flash Memory, but will be stored in SRAM at Runtime |
| __cmos_data_ram_start__ | Start Address in SRAM of Data, that is stored in Flash Memory, but will be stored in SRAM at Runtime |
| __cmos_constructor_start__ | Start Address of Code, that is needed to initialized global static Variables and Objects |
| __cmos_constructor_size__ | Size of Code in Bytes, that is needed to initialized global static Variables and Objects |

# 3 Class & Function Reference

## 3.1 Kernel Library

### 3.1.1 Data Types

There are a few Data Types pre-defined:

| Data Type | Description |
|---|---|
| uint8 | 8 Bit unsigned Integer |
| uint16 | 16 Bit unsigned Integer |
| uint32 | 32 Bit unsigned Integer |
| uint64 | 64 Bit unsigned Integer |
| int8 | 8 Bit signed Integer |
| int16 | 16 Bit signed Integer |
| int32 | 32 Bit signed Integer |
| int64 | 64 Bit signed Integer |
| feedback | Defined as a uint32, but used as Return Type to indicate if the Function finished as intended or an Error occurred. This is supposed to be used together with the Constants "OK" and "FAIL". |

### 3.1.2 Constants

| Constant | Data Type | Value | Description |
|---|---|---|---|
| OK | feedback | 0 | Return Value for Functions with Return Type "feedback" |
| FAIL | feedback | 1 | Return Value for Functions with Return Type "feedback" |

### 3.1.3 Register Definitions

For every Core, the Processor Peripherals are defined with their respective Register Sets. The Registers are defined as Pointers in the Namespace "CORE". Writing to a Register can be done with

```
1. *CORE::NVIC::IABR = 0xDEADBEEF;
```

Reading a Register:

```
1. uint32 registerValue = *CORE::NVIC::IABR;
```

Directly writing to Processor Registers is not recommended, because the Operating Systems already handles everything related to the Processor Peripherals and should not disturbed.

## 3.1.4 Interfaces

The Operating System defines Interfaces for Standard Modules to be used by the Device Drivers and the User Application to simplify porting Code to other Microcontrollers.

### *3.1.4.1 I_CAN*

#### 3.1.4.1.1 e_error: uint8

This Enum is used to log CAN Bus Errors that occurred.

#### 3.1.4.1.2 e_state: uint8

This Enum is used to show the current CAN Bus State

#### 3.1.4.1.3 feedback tx(const CAN_Frame& canFrame)

Transmits a CAN Frame.

#### 3.1.4.1.4 feedback rx(CAN_Frame& canFrame)

Writes the next CAN Frame from the internal Rx Buffer to *canFrame* or sleeps until a valid CAN Frame is received.

#### 3.1.4.1.5 uint32 get_numberOfUnread() const

Returns the Number of unread CAN Frames in the internal Rx Buffer

#### 3.1.4.1.6 bool is_dataAvailable() const

Returns *true*, if there are unread CAN Frames in the internal Rx Buffer - otherwise *false* is returned.

#### 3.1.4.1.7 uint16 get_eventID()

Returns the Event ID of the CAN Rx Event, triggered when a valid CAN Frame is received.

#### 3.1.4.1.8 e_state get_state()

Returns the current State of the CAN Bus. See e_state: uint8.

#### 3.1.4.1.9 const UniqueArray<e_error>& get_errors() const

Returns a List of Errors that were observed on the CAN Bus. Every unique Error is only listed once.

#### 3.1.4.1.10 void clearErrors()

Clears the Error List.

#### 3.1.4.1.11 uint32 get_baudRate() const

Returns the Baudrate of the CAN Bus.

#### 3.1.4.1.12 feedback recoverFromBusOffState()

Recovers the CAN Bus Module from Bus Off State (possibly by resetting the Module).

### 3.1.4.2 I_CRC

#### 3.1.4.2.1 feedback init(uint8 initialValue, uint8 polynomial, bool reverseOutputData = false, bool reverseInputData = false)

Initializes the CRC Module for 8-Bit CRC Computing.

#### 3.1.4.2.2 feedback init(uint16 initialValue, uint16 polynomial, bool reverseOutputData = false, bool reverseInputData = false)

Initializes the CRC Module for 16-Bit CRC Computing.

#### 3.1.4.2.3 feedback init(uint32 initialValue, uint32 polynomial, bool reverseOutputData = false, bool reverseInputData = false)

Initializes the CRC Module for 32-Bit CRC Computing.

#### 3.1.4.2.4 I_CRC& operator<<(uint8 data)

This Operator is used to conveniently shift 8-Bit Data for CRC-Computation into the CRC Module.

#### 3.1.4.2.5 I_CRC& operator<<(uint16 data)

This Operator is used to conveniently shift 16-Bit Data for CRC-Computation into the CRC Module.

#### 3.1.4.2.6 I_CRC& operator<<(uint32 data)

This Operator is used to conveniently shift 32-Bit Data for CRC-Computation into the CRC Module.

#### 3.1.4.2.7 uint32 operator()()

This Operator is used to obtain the Result of the CRC-Computation, which should be aligned to the used Bit Width by the User.

### 3.1.4.3 I_DMA

#### 3.1.4.3.1 f_callback

This Function Type Definition is used to give the User the Possibility to provide a Callback Function to be executed after the Data Transfer has finished in asynchronous non-blocking Fashion.

#### 3.1.4.3.2 e_priority: uint8

This Enum is used to indicate the Priority of the Data Transfer.

#### 3.1.4.3.3 e_dataType: uint8

This Enum is used to indicate the Bit Width of the Data Transfer.

#### 3.1.4.3.4 e_direction: uint8

This Enum is used to indicate the Direction of the Data Transfer.

#### 3.1.4.3.5 feedback reset()

Resets the DMA Module.

#### 3.1.4.3.6 feedback transfer(const void* source, const void* destination, e_dataType sourceType, e_dataType destinationType, bool source_inc, bool dest_inc, e_direction direction, bool circularMode, uint16 numberOfItems, e_priority priority = e_priority::LOW, bool waitForTransferEnd = true, f_callback callback = nullptr)

This Function starts a Data Transfer with the given Parameters. It is possible to use this Function as a blocking Call by setting the Parameter *waitForTransferEnd* to *false*. This makes the Function executing the Transfer and then return. Otherwise, the Function returns immediately after initializing the DMA Module and executing the Callback Function *callback* when the Transfer is finished and *callback* is not set to *nullptr*.

### *3.1.4.4 I_I2C*

#### 3.1.4.4.1 e_mode: uint8

This Enum is used to indicate the I2C Bus Speed.

#### 3.1.4.4.2 feedback start(uint8 slaveAddress, bool write, uint8 numberOfBytes = 0, uint32 timeout_ms = 100)

Transmits an I2C START Condition with the Slave Address. This can be used to query for Devices. It returns *OK* if an ACK Condition is received, *FAIL* if it's a NACK Condition. The Parameter *numberOfBytes* indicates the Number of Data Bytes to be transmitted after the START Condition to automatically send a STOP Condition then.

#### 3.1.4.4.3 void stop()

Sends a STOP Condition on the I2C Bus. This is normally automatically sent after the specified Number of Bytes, so this Function doesn't need to be called.

#### 3.1.4.4.4 feedback tx(uint8 data, uint32 timeout_ms = 100)

Transmits a single Byte on the Bus. Before calling this Function the first Time, a START Condition should be sent using the start Function.

#### 3.1.4.4.5 uint8 rx(uint32 timeout_ms = 100)

Receives a single Byte from the Bus. This is a blocking Function, waiting a maximum *timeout_ms* Milliseconds until returning. In Case of a Timeout, the Value 0 is expected to be returned.

### 3.1.4.5 I_MDIO

#### 3.1.4.5.1 uint16 readRegister(uint16 address)

Reads the Register on the specified Address.

#### 3.1.4.5.2 feedback writeRegister(uint16 address, uint16 data)

Writes the Data to the specified Address, but doesn't read it back for Confirmation.

### 3.1.4.6 I_NVM

#### 3.1.4.6.1 feedback write(uint32 address, <dataType> data)

Writes the Data to the specified Address, but doesn't read it back for Confirmation.
<dataType> is to be substituted with the corresponding Data Type.

#### 3.1.4.6.2 <dataType> read_<dataType>(uint32 address)

Reads Data from the specified Address.
<dataType> is to be substituted with the corresponding Data Type.

### 3.1.4.7 I_Ringbuffer

This is a Template Structure.
Configurable Data Types:

- dataType

#### 3.1.4.7.1 feedback write(const dataType& data)

Appends the Data to the Buffer. Returns *FAIL*, if the Buffer is full – otherwise *OK*.

#### 3.1.4.7.2 dataType read()

Reads the oldest Data from the Buffer.

#### 3.1.4.7.3 void clear()

Clears all Data in the Ringbuffer.

#### 3.1.4.7.4 void reset()

Resets the Ringbuffer and clears all Data.

#### 3.1.4.7.5 uint32 get_numberOfUnread() const

Returns the Number of unread Data in the Ringbuffer

##### 3.1.4.7.6 bool is_empty() const

Returns *true*, if there is no unread Data in the Ringbuffer, *false* otherwise.

##### 3.1.4.7.7 bool is_full() const

Returns *true*, if the Ringbuffer is full, *false* otherwise.

##### 3.1.4.7.8 uint32 get_size() const

Returns the maximum Size of the Ringbuffer.

##### 3.1.4.7.9 bool is_valid() const

Returns *true*, if all Parameters of the Ringbuffer are valid, *false* otherwise. Invalid Parameters can occur, if the Memory of the Ringbuffer is not allocated or the Pointers not initialized.

##### 3.1.4.7.10 bool contains(const dataType& data) const

Returns *true*, if any of the unread Data is equal to *data*, *false* otherwise.

### *3.1.4.8 I_Semaphore*

#### 3.1.4.8.1 feedback lock()

Locks the Semaphore. If the Semaphore is already locked, it sleeps until the Semaphore is unlocked again.
Returns *OK*, if the Lock was successful, *FAIL* otherwise.
Failing to lock a Semaphore can occur, if the Semaphore doesn't exist.

#### 3.1.4.8.2 feedback unlock()

Unlocks the Semaphore. Returns *OK*, if the Unlock was successful, *FAIL* otherwise.
Failing to unlock a Semaphore can occur, if the Permissions are insufficient or the Semaphore doesn't exist. Insufficient Permission can be caused, if the Semaphore was locked by another Thread, User, etc… and not by the current one.

#### 3.1.4.8.3 feedback force_unlock()

Forces the Semaphore to be unlocked. This Function is only to be used in system-critical Conditions, because it disturbs the intended Software Mechanism.
Returns *OK*, if the Unlock was successful, *FAIL* otherwise.
Failing to unlock a Semaphore can occur, if the Semaphore doesn't exist.

### *3.1.4.9 I_Serial*

This is a Template Structure.
Configurable Data Types:

- dataType

This Class implements the common serial Interface with basic Functions common to most Implementations. The Class itself is derived from the external Ringbuffer Class, which is used to store the received Data (= Rx Buffer).

#### 3.1.4.9.1 constexpr inline I_Serial()

Constructs the Serial Class. Intended to be used by derived Classes only.

#### 3.1.4.9.2 inline ~I_Serial()

Destructs the Serial Class. Intended to be used by derived Classes only.

#### 3.1.4.9.3 virtual dataType rx() = 0

Reads Data from the internal Rx Ringbuffer. If no Data is available in the Rx Buffer, it waits until Data is received.

#### 3.1.4.9.4 virtual feedback tx() = 0

Transmits the internal Tx Buffer over the serial Interface.
Returns *OK*, if the Transmission was successful, *FAIL* otherwise. Also returns *FAIL*, if the Semaphore Handling went wrong.

#### 3.1.4.9.5 virtual void clear() = 0

Clears the internal Rx Buffer of the Interface.

#### 3.1.4.9.6 virtual uint32 get_numberOfUnread() const = 0

Returns the number of unread Elements in the Rx Buffer.

#### 3.1.4.9.7 virtual bool is_empty() const = 0

Returns *true*, if there are no unread Messages in the Rx Buffer.

#### 3.1.4.9.8 virtual bool is_full() const = 0

Returns *true*, if the Rx Buffer is full and no new Messages cant be stored.

#### 3.1.4.9.9 virtual bool contains(const dataType& data) const = 0

Returns *true*, if any of the unread Data is equal to *data*, *false* otherwise.

#### 3.1.4.9.10 virtual I_Serial& operator<<(char data) = 0

Writes 1 Character to the Tx Buffer.

#### 3.1.4.9.11 virtual I_Serial& operator<<(uint8 data) = 0

Writes 1 Byte to the Tx Buffer.

#### 3.1.4.9.12 virtual I_Serial& operator<<(uint16 data) = 0

Writes 2 Bytes to the Tx Buffer.

#### 3.1.4.9.13 virtual I_Serial& operator<<(uint32 data) = 0

Writes 4 Bytes to the Tx Buffer.

#### 3.1.4.9.14 virtual I_Serial& operator<<(int8 data) = 0

Writes 1 Byte to the Tx Buffer.

#### 3.1.4.9.15 virtual I_Serial& operator<<(int16 data) = 0

Writes 2 Bytes to the Tx Buffer.

#### 3.1.4.9.16 virtual I_Serial& operator<<(int32 data) = 0

Writes 4 Bytes to the Tx Buffer.

#### 3.1.4.9.17 virtual I_Serial& operator<<(float data) = 0

Writes a float to the Tx Buffer.

#### 3.1.4.9.18 virtual I_Serial& operator<<(double data) = 0

Writes a double to the Tx Buffer.

#### 3.1.4.9.19 virtual I_Serial& operator<<(const Array<dataType>& data) = 0

Writes an Array to the Tx Buffer.

#### 3.1.4.9.20 virtual I_Serial& operator<<(const String& data) = 0

Writes a String to the Tx Buffer.

#### 3.1.4.9.21 virtual I_Serial& operator>>(char& data) = 0

Reads a Character from the Rx Buffer.

#### 3.1.4.9.22 virtual I_Serial& operator>>(uint8& data) = 0

Reads 1 Byte from the Rx Buffer.

#### 3.1.4.9.23 virtual I_Serial& operator>>(uint16& data) = 0

Reads 2 Bytes from the Rx Buffer.

#### 3.1.4.9.24 virtual I_Serial& operator>>(uint32& data) = 0

Reads 4 Bytes from the Rx Buffer.

#### 3.1.4.9.25 virtual I_Serial& operator>>(int8& data) = 0

Reads 1 Byte from the Rx Buffer.

#### 3.1.4.9.26 virtual I_Serial& operator>>(int16& data) = 0

Reads 2 Bytes from the Rx Buffer.

#### 3.1.4.9.27 virtual I_Serial& operator>>(int32& data) = 0

Reads 4 Bytes from the Rx Buffer.

#### 3.1.4.9.28 virtual I_Serial& operator>>(float& data) = 0

Reads a float from the Rx Buffer.

#### 3.1.4.9.29 virtual I_Serial& operator>>(double& data) = 0

Reads a double from the Rx Buffer.

#### 3.1.4.9.30 virtual I_Serial& operator>>(String& data) = 0

Reads a String from the Rx Buffer.


### *3.1.4.10 I_SPI*

This Class implements the common SPI Interface.

#### 3.1.4.10.1 virtual uint8 operator()(uint8 data) = 0

Writes and reads simultaneously 1 Byte to/from the Serial Interface.

#### 3.1.4.10.2 virtual Array<uint8> operator()(const Array<uint8>& data) = 0

Writes and reads simultaneously n Byte to/from the Serial Interface. n is determined by the given Array's Size.


### 3.1.5 File Formats

#### *3.1.5.1 CSV*

#### *3.1.5.2 JSON*

#### *3.1.5.3 XML*


### 3.1.6 Standalone Functions

## 3.1.7 Template Classes

### *3.1.7.1 Array*

The Array Class is a dynamic Implementation of the C++ Standard Array using the Heap to manage its Memory.

#### 3.1.7.1.1 constexpr inline Array()

Creates an empty Array with Size 0 and does not allocate any Memory.

#### 3.1.7.1.2 inline Array(const Array& array)

Creates an Array by copying the Contents of the given Array using the Copy-Constructor.

#### 3.1.7.1.3 Array(dataType defaultValue, uint32 numberOfElements = 1)

Creates an Array with the desired Size that is filled with all its Elements set to the given default Value.

#### 3.1.7.1.4 inline ~Array()

Destructs the Array by calling the Destructors of all its Elements. Memory dynamically allocated by an Element must be managed by the Element itself.

#### 3.1.7.1.5 static constexpr inline bool is_valid(uint32 index)

Validates the Index by comparing it against the constant Value *c_invalid*.

#### 3.1.7.1.6 constexpr inline uint32 get_size() const

Returns the Array's Size in Elements.

#### 3.1.7.1.7 constexpr inline const dataType* get_data() const

Returns the internal Pointer to the Start of the Array. This Function should be used with Care.

#### 3.1.7.1.8 dataType mediumValue() const

Returns the mathematical Medium Value of all Elements in the Array. To use this Function, the Operators += and /= must be available.

#### 3.1.7.1.9 uint32 count(const dataType& data) const

Counts, how often the given Value occurs in the Array.

#### 3.1.7.1.10 uint32 find(const dataType& data, uint32 start = 0, uint32 occurence = 1) const

Returns the Index where the $occurence_{th}$ Occurrence of *data* since the Index *start* is located.

#### 3.1.7.1.11 uint32 find(const Array<dataType>& data, uint32 start = 0, uint32 occurence = 1) const

Returns the Index where the $occurence_{th}$ Occurrence of *data* since the Index *start* is located.
This Function searches for a Series of Elements with the Length of the given Array.

#### 3.1.7.1.12 bool contains(const dataType& data) const

Checks, if the Array contains minimum 1 of the searched Elements.

#### 3.1.7.1.13 Array sub(uint32 index, uint32 size = c_invalid) const

Returns n Elements starting a the given Index.

#### 3.1.7.1.14 Array subWithDelimiter(dataType delimiter, uint32 subArrayNumber) const

Returns a Subarray that is delimited by the given Delimiter Element. The Subarray with the Elements from Index 0 until the first Delimiter Element has the Index 0. The Subarrays do not contain the Delimiter Element.

#### 3.1.7.1.15 Array<Array> explode(dataType delimiter) const

Splits the Array into Subarrays that are separated by the given Delimiter Element. The Delimiter Element itself is not included in the Subarrays.

#### 3.1.7.1.16 feedback set_size(uint32 newSize, bool fillWithDefaultValue = true, dataType defaultValue = dataType())

Sets the Array's Size to the desired Value. The newly created Elements are set to the given default Value if desired. If the newSize is shorter than the actual Size, all Elements that are not needed are simply cut off, but not destructed.

#### 3.1.7.1.17 inline void fill(const dataType& data)

Sets all Elements to the given Value.

#### 3.1.7.1.18 feedback insert(uint32 index, const dataType& data)

Inserts the given Value at the desired Index.

#### 3.1.7.1.19 feedback insert(uint32 index, const Array& array)

Insert the Array at the given Index.

#### 3.1.7.1.20 feedback erase(uint32 index, uint32 numberOfElementsToErase = 1)

Erases the desired Number of Elements starting from the given Index. The Destructor of the erased Elements is called.

#### 3.1.7.1.21 void erase()

Erases the whole Array by calling the Element's Destructors.

#### 3.1.7.1.22 inline feedback eraseToEnd(uint32 index)

Erases all Elements starting from the given Index until the End of the Array by calling the Function erase.

#### 3.1.7.1.23 inline feedback eraseFromEnd(uint32 numberOfElementsToErase)

Erases the given Number of Elements starting at the Array's End with descending Index by calling the Function erase.

#### 3.1.7.1.24 void eraseAllOccurences(dataType elementToErase, uint32 start = 0)

Erases all Elements where the Operator = returns true starting at the given Index. These Elements are erased using the Function erase.

#### 3.1.7.1.25 Array& operator=(const Array& array)

First, erases all Elements of the Array using erase, then inserts all Elements of the given Array using insert.

#### 3.1.7.1.26 Array operator+(const dataType& data) const

Returns a new Array where the given Element is appended to this.

#### 3.1.7.1.27 inline Array& operator+=(const dataType& data)

Appends the given Element to this Array.

#### 3.1.7.1.28 Array operator+(const Array& array) const

Returns a new Array where the given Array is appended to this.

#### 3.1.7.1.29 inline Array& operator+=(const Array& array)

Appends the given Array to this Array.

#### 3.1.7.1.30 constexpr inline dataType operator[](uint32 index) const

Returns the Element at the given Index by Value. Altering the Return Value does not alter the Array.
Does not check for out-of-boundary Access.

#### 3.1.7.1.31 constexpr inline dataType& operator[](uint32 index)

Returns the Element at the given Index by Reference. Altering the Return Value alters the Array.
Does not check for out-of-boundary Access.

#### 3.1.7.1.32 constexpr inline const dataType& operator()(uint32 index) const

Does the same as operator[];

#### 3.1.7.1.33 bool operator==(const Array<dataType> array) const

Compares two Arrays for Equalness using an efficient Approach by first comparing the Size of the Arrays and then comparing the single Elements.

#### 3.1.7.1.34 inline bool operator!=(const Array<dataType> array) const

Returns the inverse Result from calling operator==.

#### 3.1.7.1.35 constexpr inline dataType* begin() const

Returns the Pointer to the first Element of the Array. Not to be used by the User but needed by auto-Loops in C++.

#### 3.1.7.1.36 constexpr inline dataType* end() const

Returns the Pointer to the last Element of the Array. Not to be used by the User but needed by auto-Loops in C++.


ARM CORTEX
Cortex-M1

### *3.1.7.2 Pair*

This Class provides a simple Implementation of pairwise Data Management.

#### 3.1.7.2.1 constexpr inline Pair()

Creates a default constructed Pair.

#### 3.1.7.2.2 constexpr inline Pair(const dataType_1& first, const dataType_2& second)

Creates a Pair with the given Values.

#### 3.1.7.2.3 constexpr inline Pair(const Pair& pair)

Creates a Pair as a Copy from the given Pair. Uses the Copy-Constructor to copy the Contents.

#### 3.1.7.2.4 inline ~Pair()

Destructs the Pair by calling the Destructor of its Elements.

#### 3.1.7.2.5 constexpr inline dataType_1& first()

Returns a Reference to the first Element of the Pair.

#### 3.1.7.2.6 constexpr inline dataType_2& second()

Returns a Reference to the second Element of the Pair.

#### 3.1.7.2.7 constexpr inline dataType_1 first() const

Returns the first Value by Value.

#### 3.1.7.2.8 constexpr inline dataType_2 second() const

Returns the second Value by Value.

#### 3.1.7.2.9 inline Pair& operator=(const Pair<dataType_1, dataType_2>& pair)

Overwrites the Content of this Pair with the given Values.

#### 3.1.7.2.10 inline bool operator==(const Pair<dataType_1, dataType_2>& pair) const

Compares the Content of this Pair with the given Pair. Returns true, if it equals. Requires the == Operator to exist for both Datatypes.

### *3.1.7.3 Triplet*

This Class extends the Pair by one additional Element. All Functions of the Pair Class can be used.

#### 3.1.7.3.1 constexpr inline Triplet()

Creates a default constructed Triplet.

#### 3.1.7.3.2 constexpr inline Triplet(const dataType_1& first, const dataType_2& second, const dataType_3& third)

Creates a Triplet with the given Values.

#### 3.1.7.3.3 constexpr inline Triplet(const Triplet& array)

Creates a Triplet as a Copy from the given Triplet. Uses the Copy-Constructor to copy the Contents.

#### 3.1.7.3.4 inline ~Triplet()

Destructs the Pair by calling the Destructor of its Elements.

#### 3.1.7.3.5 constexpr inline dataType_3& third()

Returns a Reference to the third Element of the Triplet.

#### 3.1.7.3.6 constexpr inline dataType_3 third() const

Returns the third Value by Value.

#### 3.1.7.3.7 inline Triplet& operator=(const Triplet<dataType_1, dataType_2, dataType_3>& triplet)

Overwrites the Content of this Triplet with the given Values.

#### 3.1.7.3.8 inline bool operator==(const Triplet<dataType_1, dataType_2, dataType_3>& triplet) const

Compares the Content of this Triplet with the given Triplet. Returns true, if it equals. Requires the == Operator to exist for all three Datatypes.

### *3.1.7.4 Matrix*

The Matrix is a simple Table like an Excel Sheet. The Datatype is configurable as Template Parameter.

#### 3.1.7.4.1 void enableBoundaries(uint32 numberOfColumns, uint32 numberOfRows)

Reserves the Memory for the required Columns and Rows. Also sets the Matrix Size to these Values.

#### 3.1.7.4.2 Matrix()

Creates an empty Matrix with 0 Columns and Rows.

#### 3.1.7.4.3 Matrix(uint32 numberOfColumns, uint32 numberOfRows)

Creates a Matrix with the desired Size. The Cells are filled with default Values.

#### 3.1.7.4.4 Matrix(uint32 numberOfColumns, uint32 numberOfRows, dataType defaultValue)

Creates a Matrix with the desired Size. The Cells are filled with the given defaultValue.

#### 3.1.7.4.5 ~Matrix()

Frees the Memory used by the Matrix and also calls the Destructors of its Elements. If Elements have been allocated using dynamic Memory, the User is responsible for freeing the respective Memory.

#### 3.1.7.4.6 uint32 get_numberOfRows() const

Returns the actual Number of Rows of the Matrix.

#### 3.1.7.4.7 uint32 get_numberOfColumns() const

Returns the actual Number of Columns of the Matrix.

#### 3.1.7.4.8 dataType get_cell(uint32 columnNumber, uint32 rowNumber) const

Returns the Cell Content by Value. Altering the Return Value doesn't alter the Matrix.

#### 3.1.7.4.9 dataType& cell(uint32 columnNumber, uint32 rowNumber)

Returns the Cell Content by Reference. Altering the Return Value also alters the Matrix.

#### 3.1.7.4.10 Array<dataType> get_row(uint32 rowNumber) const

Returns the desired Row by Value. Altering the Return Value doesn't alter the Matrix.

#### 3.1.7.4.11 Array<dataType>& get_row(uint32 rowNumber)

Returns the desired Row by Reference. Altering the Return Value also alters the Matrix.

#### 3.1.7.4.12 Array<dataType> get_column(uint32 columnNumber) const

Returns the desired Column by Value. Altering the Return Value doesn't alter the Matrix.

#### 3.1.7.4.13 void insert_row(uint32 rowNumber, Array<dataType>& row)

Inserts the Row at the given Row Number.

#### 3.1.7.4.14 void insert_column(uint32 columnNumber, Array<dataType>& column)

Inserts the Column at the given Column Number.

#### 3.1.7.4.15 void append_row(Array<dataType>& row)

Appends the Row at the End of the Matrix.

#### 3.1.7.4.16 void append_column(Array<dataType>& column)

Appends the Column at the End of the Matrix.

#### 3.1.7.4.17 void erase()

Erases the Content of the Matrix by calling the Destructors of its Elements. Elements which have Memory dynamically allocated have to free this Memory themselves. Sets the Matrix's Size to 0.

#### 3.1.7.4.18 feedback delete_row(uint32 rowNumber)

Deletes the Row by calling the Destructors of the Row's Elements. Elements which have Memory dynamically allocated have to free this Memory themselves.

#### 3.1.7.4.19 feedback delete_column(uint32 columnNumber)

Deletes the Column by calling the Destructors of the Column's Elements. Elements which have Memory dynamically allocated have to free this Memory themselves.

#### 3.1.7.4.20 void fill(dataType fillValue)

Overwrites all Elements of the Matrix to the desired Value.

#### 3.1.7.4.21 void set_row(uint32 rowNumber, Array<dataType> row)

Sets the Content of the respective Row to the desired Value. If the given Row is shorter than the Matrix's Row Size, it is lengthened by appending it with default Values. If it is longer, the Matrix' Row Size is lengthened by appending default Values to all Rows.

#### 3.1.7.4.22 void set_column(uint32 columnNumber, Array<dataType> column)

Sets the Content of the respective Column to the desired Value. If the given Column is shorter than the Matrix's Column Size, it is lengthened by appending it with default Values. If it is longer, the Matrix' Column Size is lengthened by appending default Values to all Columns.

#### 3.1.7.4.23 void set_cell(uint32 columnNumber, uint32 rowNumber, dataType value)

Sets the Content of the respective Cell to the given Value.

#### 3.1.7.4.24 Matrix& operator+=(const Matrix& matrix)

Appends the given Matrix by repeatedly using the Function append_row.

#### 3.1.7.4.25 Matrix operator+(const Matrix& matrix) const

Returns a new Matrix by creating an empty Matrix and using the Operator += with this and the given Matrix.

### *3.1.7.5 Ringbuffers*

All Ringbuffers implement the Interface Class I_Ringbuffer with a different Way of managing the underlying Memory.

#### 3.1.7.5.1 I_Ringbuffer

See I_Ringbuffer.

#### 3.1.7.5.2 RingbufferDynamic

The dynamic Ringbuffer stores its Elements on the dynamic Memory, making it possible to re-size the Buffer during Runtime.

##### 3.1.7.5.2.1 inline RingbufferDynamic(uint32 size = 0, dataType defaultValue = dataType())

Creates a Ringbuffer with the desired Size and set all Elements to the given default Value.

##### 3.1.7.5.2.2 inline ~RingbufferDynamic() override

Destructs the Ringbuffer and calls the Destructors of all its Elements. Memory dynamically allocated by an Element must be managed by the Element itself.

##### 3.1.7.5.2.3 feedback set_size(uint32 size)

Re-sizes the Ringbuffer. If the new Size is smaller than the actual Size and unread Elements would be lost by decreasing the Buffer's Size, the Operation is not carried out and returns *FAIL*.

##### 3.1.7.5.2.4 feedback write(const dataType& data) override

Writes a new Element to the Buffer. Returns *FAIL* if the Buffer is full already.

##### 3.1.7.5.2.5 dataType read() override

Reads an Element from the Buffer. If the Buffer is empty, returns a default constructed Element.

##### 3.1.7.5.2.6 inline void clear() override

Clears the Buffer so that it's empty.

#### 3.1.7.5.2.7 inline void reset() override

Resets the Buffer's Head and Tail to 0, effectively clearing it's Content.

#### 3.1.7.5.2.8 uint32 get_numberOfUnread() const override

Returns the Number of unread Elements.

#### 3.1.7.5.2.9 inline bool is_empty() const override

Returns *true*, if the Buffer has no unread Elements.

#### 3.1.7.5.2.10 inline bool is_full() const override

Returns *true*, if the Buffer is full.

#### 3.1.7.5.2.11 inline uint32 get_size() const override

Returns the total Buffer Size, regardless of if the Elements are read or not.

#### 3.1.7.5.2.12 inline bool is_valid() const override

Returns *true*, if the Buffer itself is initialized with a valid Memory Pointer and its Size is greater than 0.

#### 3.1.7.5.2.13 bool contains(const dataType& data) const override

Returns *true*, if the given Value occurs in the unread Elements.

### 3.1.7.5.3 RingbufferExternal

The external Ringbuffer stores its Data in a given Memory Area, that is provided by the User on Construction. The Memory's Size and Location is therefore fixed and cant be changed during Runtime.

#### 3.1.7.5.3.1 constexpr inline RingbufferExternal(dataType* buffer, uint32 size)

Creates the Ringbuffer by providing a Pointer to the Memory Area and the Memory's Size. The Pointer must not be null.

#### 3.1.7.5.3.2 inline ~RingbufferExternal() override

Destructs the Ringbuffer. This Function does not call any Element's Destructor.

#### 3.1.7.5.3.3 feedback write(const dataType& data) override

Writes a new Element to the Buffer. Returns *FAIL* if the Buffer is full already.

#### 3.1.7.5.3.4 dataType read() override

Reads an Element from the Buffer. If the Buffer is empty, returns a default constructed Element.

#### 3.1.7.5.3.5 inline void clear() override

Clears the Buffer so that it's empty.

#### 3.1.7.5.3.6 inline void reset() override

Resets the Buffer's Head and Tail to 0, effectively clearing it's Content.

#### 3.1.7.5.3.7 uint32 get_numberOfUnread() const override

Returns the Number of unread Elements.

#### 3.1.7.5.3.8 inline bool is_empty() const override

Returns *true*, if the Buffer has no unread Elements.

#### 3.1.7.5.3.9 inline bool is_full() const override

Returns *true*, if the Buffer is full.

#### 3.1.7.5.3.10 inline uint32 get_size() const override

Returns the total Buffer Size set in the Constructor, regardless of if the Elements are read or not.

#### 3.1.7.5.3.11 inline bool is_valid() const override

Returns *true*, if the Buffer itself is initialized with a valid Memory Pointer and its Size is greater than 0.

#### 3.1.7.5.3.12 bool contains(const dataType& data) const override

Returns *true*, if the given Value occurs in the unread Elements.

#### 3.1.7.5.3.13 constexpr inline dataType* get_data() const

Returns the Pointer to the Memory Area set by the Constructor.

### 3.1.7.5.4 RingbufferStatic

The static Ringbuffer stores its Elements as a Member Variable, locating them on the same Memory as the Class itself. This is intended to be used to place the Buffer on the Stack or the global Memory. The maximum Number of Elements that can be stored in this Buffer is set by the Template Parameter *N*.

#### 3.1.7.5.4.1 constexpr inline RingbufferStatic()

Creates the empty Ringbuffer.

#### 3.1.7.5.4.2 inline ~RingbufferStatic() override

Destructs the Ringbuffer. All Element's Destructors are called as they are Member Variables.

#### 3.1.7.5.4.3 feedback write(const dataType& data) override

Writes a new Element to the Buffer. Returns *FAIL* if the Buffer is full already.

#### 3.1.7.5.4.4 dataType read() override

Reads an Element from the Buffer. If the Buffer is empty, returns a default constructed Element.

#### 3.1.7.5.4.5 inline void clear() override

Clears the Buffer so that it's empty.

#### 3.1.7.5.4.6 inline void reset() override

Resets the Buffer's Head and Tail to 0, effectively clearing it's Content.

#### 3.1.7.5.4.7 uint32 get_numberOfUnread() const override

Returns the Number of unread Elements.

#### 3.1.7.5.4.8 inline bool is_empty() const override

Returns *true*, if the Buffer has no unread Elements.

#### 3.1.7.5.4.9 inline bool is_full() const override

Returns *true*, if the Buffer is full.

#### 3.1.7.5.4.10 inline uint32 get_size() const override

Returns the total Buffer Size set by the Template Parameter *N*.

#### 3.1.7.5.4.11 inline bool is_valid() const override

Returns *true*, if the Buffer itself is initialized with a valid Memory Pointer.

#### 3.1.7.5.4.12 bool contains(const dataType& data) const override

Returns *true*, if the given Value occurs in the unread Elements.

### 3.1.7.6 I_Serial

See I_Serial.

### 3.1.7.7 UniqueArray

This Class implements an Array that contains every Value only once. Two Elements with the same Value are impossible.

3.1.7.7.1 constexpr inline UniqueArray()

Creates an empty Array.

3.1.7.7.2 inline UniqueArray(const UniqueArray& uniqueArray)

Creates this Array by copying the Elements from the given Array.

3.1.7.7.3 inline ~UniqueArray()

Destroys the Array. The Destructors of all Elements are called.

3.1.7.7.4 constexpr inline uint32 get_size() const

Returns the Number of Elements in this Array.

3.1.7.7.5 inline bool contains(const dataType& data) const

Returns *true*, if the Array contains an Element with this Value.

3.1.7.7.6 inline void erase()

Erases all Elements in this Array. Calls the Destructor of every erased Element.

3.1.7.7.7 inline feedback erase(const dataType& data)

Erases the Element with the given Value. Returns *OK*, it this Element has been erased. Returns *FAIL*, if this Element could not be erased (maybe because it doesn't exist).

3.1.7.7.8 inline UniqueArray& operator=(const UniqueArray& uniqueArray)

Copies the Content from the given Array and overwrites the Content of this Array with it.

3.1.7.7.9 UniqueArray operator+(const dataType& data) const

Creates a new Array from this Array and appends the given Element to it.

3.1.7.7.10 UniqueArray& operator+=(const dataType& data)

Appends the given Element to this Array only if the Element's Value doesn't already exist in this Array.

3.1.7.7.11 UniqueArray operator+(const UniqueArray& uniqueArray) const

Creates a new Array from this Array and appends the given Array. Duplicate Values are not appended.

#### 3.1.7.7.12 UniqueArray& operator+=(const UniqueArray& uniqueArray)

Appends the given Array to this Array. Duplicate Values are not appended.

#### 3.1.7.7.13 constexpr inline dataType* begin() const

Returns the Pointer to the first Element of the Array. Not to be used by the User but needed by auto-Loops in C++.

#### 3.1.7.7.14 constexpr inline dataType* end() const

Returns the Pointer to the last Element of the Array. Not to be used by the User but needed by auto-Loops in C++.

### *3.1.7.8 UniquePairArray*

This Class implements a Key-Value Array with unique Keys.

3.1.7.8.1 constexpr inline UniquePairArray()

Creates an empty Array.

3.1.7.8.2 inline UniquePairArray(const UniquePairArray& uniquePairArray)

Creates this Array by copying the Elements from the given Array.

3.1.7.8.3 inline ~UniquePairArray()

Destroys the Array. The Destructors of all Elements are called.

3.1.7.8.4 constexpr inline uint32 get_size() const

Returns the Number of Elements in this Array.

3.1.7.8.5 bool contains(const dataType_1& first) const

Returns *true*, if the Key-Value exists in this Array.

3.1.7.8.6 inline void erase()

Erases all Elements in this Array. Calls the Destructor of every erased Element.

3.1.7.8.7 feedback erase(const dataType_1& first)

Erases the Element with the given Key-Value. Returns *OK*, it this Element has been erased. Returns *FAIL*, if this Element could not be erased (maybe because it doesn't exist).

3.1.7.8.8 inline UniquePairArray& operator=(const UniquePairArray& uniquePairArray)

Copies the Content from the given Array and overwrites the Content of this Array with it.

3.1.7.8.9 UniquePairArray operator+(const Pair<dataType_1, dataType_2>& pair) const

Creates a new Array from this Array and appends the given Element to it.

3.1.7.8.10 UniquePairArray& operator+=(const Pair<dataType_1, dataType_2>& pair)

Appends the given Element to this Array only if the Element's Value doesn't already exist in this Array.

3.1.7.8.11 UniquePairArray operator+(const UniquePairArray& uniquePairArray) const

Creates a new Array from this Array and appends the given Array. Duplicate Values are not appended.

3.1.7.8.12 UniquePairArray& operator+=(const UniquePairArray& uniquePairArray)

Appends the given Array to this Array. Duplicate Values are not appended.

#### 3.1.7.8.13 dataType_2 operator[](const dataType_1& first) const

Returns the Value by Value corresponding to the given Key-Value. If the Key-Value doesn't exist in this Array, a default constructed Pair is returned, which is NOT an Element of this Array.

#### 3.1.7.8.14 dataType_2& operator[](const dataType_1& first)

Returns the Value by Reference corresponding to the given Key-Value. If the Key-Value doesn't exist in this Array, a default constructed Pair is appended to this Array and the Reference to the Value is returned.

#### 3.1.7.8.15 constexpr inline Pair<dataType_1, dataType_2>* begin() const

Returns the Pointer to the first Element of the Array. Not to be used by the User but needed by auto-Loops in C++.

#### 3.1.7.8.16 constexpr inline Pair<dataType_1, dataType_2>* end() const

Returns the Pointer to the last Element of the Array. Not to be used by the User but needed by auto-Loops in C++.

### *3.1.7.9 Vector2*

This Class represents a 2-dimensional Vector (or Point – which is a Vector starting at 0/0) with a flexible Datatype used for the Coordinates.

#### 3.1.7.9.1 constexpr inline Vector2()

Creates the default Vector with x = 0 and y = 0.

#### 3.1.7.9.2 constexpr inline Vector2(dataType x, dataType y)

Creates a Vector with the given Values.

#### 3.1.7.9.3 constexpr inline Vector2(const Vector2& start, const Vector2& end)

Creates a Vector that points from *start* to *end*.

#### 3.1.7.9.4 double absolute() const

Calculates the geometric Length of the Vector using Pythagorean.

#### 3.1.7.9.5 constexpr inline Vector2 positiveVector() const

Returns the Vector with positive Coordinates only.

#### 3.1.7.9.6 constexpr inline Vector2 sign() const

Returns a Vector with the Signs of this Vector. If the respective Coordinate is negative, the output Coordinate is "-1", otherwise "1".

#### 3.1.7.9.7 constexpr inline Vector2<double> unitVector() const

Returns this Vector but scaled to the total Length 1.

#### 3.1.7.9.8 double angle(Vector2 v) const

Returns the Angle between this Vector and the Vector *v*.

#### 3.1.7.9.9 double angleToX() const

Calculates the Angle between this Vector and the x-Coordinate Axis.

#### 3.1.7.9.10 double angleToY() const

Calculates the Angle between this Vector and the y-Coordinate Axis.

#### 3.1.7.9.11 constexpr inline Vector2& min_x(Vector2& v) const

Returns the smaller x-Coordinate of both Vectors.

#### 3.1.7.9.12 constexpr inline Vector2& max_x(Vector2& v) const

Returns the bigger x-Coordinate of both Vectors.

#### 3.1.7.9.13 constexpr inline Vector2& min_y(Vector2& v) const

Returns the smaller y-Coordinate of both Vectors.

#### 3.1.7.9.14 constexpr inline Vector2& max_y(Vector2& v) const

Returns the bigger y-Coordinate of both Vectors.

#### 3.1.7.9.15 constexpr inline dataType min() const

Returns the smaller Coordinate of x and y.

#### 3.1.7.9.16 constexpr inline dataType max() const

Returns the bigger Coordinate of x and y.

#### 3.1.7.9.17 constexpr inline bool operator==(const Vector2& v) const

Returns *true*, if this Vector and *v* are equal.

#### 3.1.7.9.18 constexpr inline bool operator!=(const Vector2& v) const

Returns *false*, if this Vector and *v* are equal.

#### 3.1.7.9.19 constexpr inline Vector2& operator=(const Vector2& v)

Copies the Contents of *v* to this Vector.

#### 3.1.7.9.20 constexpr inline Vector2 operator-(const Vector2& v) const

Returns the Result of the mathematical Operation: this – *v*

#### 3.1.7.9.21 constexpr inline Vector2 operator+(const Vector2& v) const

Returns the Result of the mathematical Operation: this + *v*

#### 3.1.7.9.22 constexpr inline Vector2 operator*(dataType2 scalar) const

Returns this Vector scaled by the Factor *scalar*.

#### 3.1.7.9.23 constexpr inline double operator*(Vector2 v) const

Returns the Dot Product of this Vector and *v*.

#### 3.1.7.9.24 constexpr inline Vector2 operator/(dataType2 scalar) const

Returns this Vector scaled by the Factor: 1 / *scalar*

#### 3.1.7.9.25 constexpr inline Vector2& operator+=(const Vector2& v)

Mathematically appends *v* to this Vector.

#### 3.1.7.9.26 constexpr inline Vector2& operator-=(const Vector2& v)

Mathematically subtracts *v* from this Vector.

#### 3.1.7.9.27 constexpr inline Vector2& operator*=(dataType2 scalar)

Scales this Vector by the Factor *scalar*.

#### 3.1.7.9.28 constexpr inline Vector2& operator*=(Vector2 v)

To be honest: I have no Idea what this Function does mathematically. Can I multiply two Vectors together and yield a Vector again?

#### 3.1.7.9.29 constexpr inline Vector2& operator/=(dataType2 scalar)

Scales this Vector by the Factor: 1 / *scalar*

### *3.1.7.10 Vector3*

This Class represents a 3-dimensional Vector (or Point – which is a Vector starting at 0/0/0) with a flexible Datatype used for the Coordinates.

#### 3.1.7.10.1 constexpr Vector3()

Creates the default Vector with x = 0, y = 0 and z = 0.

#### 3.1.7.10.2 constexpr Vector3(dataType x, dataType y, dataType z)

Creates a Vector with the given Values.

#### 3.1.7.10.3 constexpr Vector3(const Vector3& start, const Vector3& end)

Creates a Vector that points from *start* to *end*.

#### 3.1.7.10.4 double absolute() const

Calculates the geometric Length of the Vector using Pythagorean.

#### 3.1.7.10.5 constexpr inline Vector3 positiveVector() const

Returns the Vector with positive Coordinates only.

#### 3.1.7.10.6 constexpr inline Vector3 sign() const

Returns a Vector with the Signs of this Vector. If the respective Coordinate is negative, the output Coordinate is "-1", otherwise "1".

#### 3.1.7.10.7 constexpr inline Vector3<double> unitVector() const

Returns this Vector but scaled to the total Length 1.

#### 3.1.7.10.8 double angle(Vector3 v) const

Returns the Angle between this Vector and the Vector *v*.

#### 3.1.7.10.9 constexpr inline Vector3& min_x(Vector3& v) const

Returns the smaller x-Coordinate of both Vectors.

#### 3.1.7.10.10 constexpr inline Vector3& max_x(Vector3& v) const

Returns the bigger x-Coordinate of both Vectors.

#### 3.1.7.10.11 constexpr inline Vector3& min_y(Vector3& v) const

Returns the smaller y-Coordinate of both Vectors.

#### 3.1.7.10.12 constexpr inline Vector3& max_y(Vector3& v) const

Returns the bigger y-Coordinate of both Vectors.

#### 3.1.7.10.13 constexpr inline Vector3& min_z(Vector3& v) const

Returns the smaller z-Coordinate of both Vectors.

#### 3.1.7.10.14 constexpr inline Vector3& max_z(Vector3& v) const

Returns the bigger z-Coordinate of both Vectors.

#### 3.1.7.10.15 constexpr inline dataType min() const

Returns the smallest Coordinate of x, y and z.

#### 3.1.7.10.16 constexpr inline dataType max() const

Returns the biggest Coordinate of x, y and z.

#### 3.1.7.10.17 constexpr inline bool operator==(const Vector3& v) const

Returns *true*, if this Vector and *v* are equal.

#### 3.1.7.10.18 constexpr inline bool operator!=(const Vector3& v) const

Returns *false*, if this Vector and *v* are equal.

#### 3.1.7.10.19 constexpr inline Vector3& operator=(const Vector3& v)

Copies the Contents of *v* to this Vector.

#### 3.1.7.10.20 constexpr inline Vector3 operator-(const Vector3& v) const

Returns the Result of the mathematical Operation: this – *v*

#### 3.1.7.10.21 constexpr inline Vector3 operator+(const Vector3& v) const

Returns the Result of the mathematical Operation: this + *v*

#### 3.1.7.10.22 constexpr inline Vector3 operator*(dataType2 scalar) const

Returns this Vector scaled by the Factor *scalar*.

#### 3.1.7.10.23 constexpr inline double operator*(Vector3 v) const

Returns the Dot Product of this Vector and *v*.

#### 3.1.7.10.24 constexpr inline Vector3 operator/(dataType2 scalar) const

Returns this Vector scaled by the Factor: 1 / *scalar*.

#### 3.1.7.10.25 constexpr inline Vector3& operator+=(const Vector3& v)

Mathematically appends *v* to this Vector.

#### 3.1.7.10.26 constexpr inline Vector3& operator-=(const Vector3& v)

Mathematically subtracts *v* from this Vector.

#### 3.1.7.10.27 constexpr inline Vector3& operator*=(dataType2 scalar)

Scales this Vector by the Factor *scalar*.

#### 3.1.7.10.28 constexpr inline Vector3& operator*=(Vector3 v)

To be honest: I have no Idea what this Function does mathematically. Can I multiply two Vectors together and yield a Vector again?

#### 3.1.7.10.29 constexpr inline Vector3& operator/=(dataType2 scalar)

Scales this Vector by the Factor: 1 / *scalar*

### *3.1.7.11 Rectangle*

This Class implements a basic Rectangle with a Position and Size as Template Class.

3.1.7.11.1 constexpr inline Rectangle()

Creates a Rectangle with Position 0/0 and Size 0/0.

3.1.7.11.2 constexpr inline Rectangle(Vector2<dataType> position, Vector2<dataType> size)

Creates a Rectangle with the specified Values.

3.1.7.11.3 constexpr inline Rectangle(dataType position_x, dataType position_y, dataType size_x, dataType size_y)

Creates a Rectangle with the specified Values.

3.1.7.11.4 constexpr inline Vector2<dataType> get_topRightCorner() const

Returns the Top-Right Corner of the Rectangle. This Point is the yielded by the Addition of the Rectangle's Position and Size Vectors.

3.1.7.11.5 constexpr inline double get_surface() const

Returns the Surface of the Rectangle as the Product of its Size Coordinates.

3.1.7.11.6 constexpr inline bool contains(const Vector2<dataType> v) const

Returns *true*, if v lies within the Rectangle.

3.1.7.11.7 constexpr inline bool contains(const Rectangle& rectangle) const

Returns *true*, if the specified Rectangle fully lies within the Rectangle.

3.1.7.11.8 bool doesOverlap(const Rectangle& rectangle) const

Returns *true*, if the specified Rectangle overlaps with this Rectangle at any Point.

3.1.7.11.9 Rectangle get_overlap(const Rectangle& rectangle) const

Returns the Rectangle, that is common to both Rectangles.

3.1.7.11.10 inline double get_overlapSurface(const Rectangle& rectangle) const

Returns the Overlap Surface of this Rectangle with the specified Rectangle.

3.1.7.11.11 constexpr inline bool operator==(const Rectangle& rectangle) const

Returns *true*, if the specified Rectangle equals this Rectangle.

3.1.7.11.12 constexpr inline Rectangle& operator=(const Rectangle& rectangle)

Overwrites this Rectangle's Position and Size with the specified Rectangle's Values.

## 3.1.8 CMOS

This Class provides all Functions related to the basic Operating System Functionality to the User.

### *3.1.8.1 static CMOS& get()*

This Function is used to get the Reference to the Operating System. This is the only intended Way of using the Operating System since it is implemented as a Singleton.

### *3.1.8.2 void run()*

This Function must be called only once at System Startup. This Function does not return, but starts the Kernel which schedules the *main* Function next.

### *3.1.8.3 constexpr inline Time get_time() const*

Returns the current System Time, which starts at 01.01.2000 00:00:00 by calling the Function run.

### *3.1.8.4 void reset()*

Resets the Processor and triggers a system-wide Reset. The Reset Implementation of the System depends on the Vendor of the MCU.

### *3.1.8.5 uint8 thread_create(f_thread entry, String name, uint8 priority, uint32 stack_size = 0, uint8 mailBoxSize = Heap::get_blockSize() / sizeof(Thread::s_mail), void* object = nullptr)*

Creates and schedules a Thread for Execution. *entry* is the Function Pointer, that should be executed as Child Thread of the currently running Thread. *name* is the Name of the Thread that can later be used for querying the Operating System and does not have necessarily to be unique (even though it's recommended). priority is a Value from 0 to 255, with 255 being the highest Priority (= preempting every other Thread). If multiple Threads with identical Priority are scheduled, the Thread with the lower Thread ID is executed first. *stack_size* is the Stack Size of the newly created Thread and can only be assigned in Blocks (see the Linker Script Symbol __cmos_stack_blockSize__). The Value of *stack_size* will always be rounded up to the next Block and is set by default to 1 Block. *mailboxSize* is the Size of the Thread's Mail Box and is default set to as many Mails fit into one Heap Block (see __cmos_heap_blockSize__). The Parameter *object* is used only by the Template Function thread_create and should not be used by the User.

### *3.1.8.6 inline uint8 thread_create(void (Class::*entryFunction)(), Class* object, String name, uint8 priority, uint16 stack_size = 0, uint8 mailBoxSize = Heap::get_blockSize() / sizeof(Thread::s_mail))*

This Function does the same as thread_create, but allows the User to run Member Functions of C++ Classes as Threads. The Parameter *entryFunction* is the Pointer to the Member Function and *object* is the Pointer to the Object that should be used as Instance of the Class.
Usage:

```
1. thread_create(&ExampleClass::exampleMemberFunction, this, "Example Name", 100);
```

The other Parameters have the same Meaning as in thread_create.

#### 3.1.8.7 void thread_setPriority(uint8 newPriority)

Sets the Priority of the currently running Thread.

#### 3.1.8.8 feedback thread_joinChildThread(uint8 threadID, uint32 sleepTime_ms = 1)

This Function makes the currently running Thread sleep in *sleepTime_ms* Intervals until the specified Child Thread shuts down.

#### 3.1.8.9 feedback thread_detachChildThread(uint8 threadID)

Detaches the Child Threads Relation to the Parent Thread. The Child Thread won't be shut down anymore when the Parent Thread shuts down.

#### 3.1.8.10 feedback thread_shutdown(uint8 threadID, bool hardShutdown = false)

Shuts down the specified Thread. This is only allowed, if the currently running Thread has the Permission to do so by being the Parent Thread or if the specified Thread is detached from its Parent Thread. If *hardShutdown* is *true*, the specified Thread is shut down immediately without any Warning. If *false*, the specified Thread is notified about its Shutdown via a Mail with Type *EXIT* and this Function waits until the Thread shuts down itself by using the *return* Statement.

#### 3.1.8.11 constexpr inline bool thread_doesExist(const String& name) const

Returns *true*, if a Thread with the specified Name exists in the Kernel.

#### 3.1.8.12 constexpr inline bool thread_doesExist(uint8 threadID) const

Returns *true*, if a Thread with the specified ID exists in the Kernel.

#### 3.1.8.13 constexpr inline bool isChildThread(uint8 threadID) const

Returns *true*, if the specified Thread is a Child Thread of the currently running Thread.

#### 3.1.8.14 constexpr inline uint8 get_runningThreadID() const

Returns the Thread ID of the currently running Thread.

#### 3.1.8.15 constexpr inline uint8 get_parentThreadID() const

Returns the Thread ID of the Parent Thread of the currently running Thread. If there is no Parent Thread (due to calling *detachChildThread* or the Thread being the *main* Function), the ID *threadID_invalid* is returned.

#### 3.1.8.16 inline String thread_getName(uint8 threadID) const

Returns the Name of the specified Thread.

#### 3.1.8.17 inline uint8 thread_getID(const String& name) const

Returns the Thread ID of the Thread with the specified Name. Since there can be multiple Threads with the same Name, the lowest Thread ID is returned. If there is no Thread with the specified Name, the ID *threadID_invalid* is returned.

#### *3.1.8.18 constexpr inline uint8 get_numberOfThreadsMaximum() const*

Returns the maximum possible Number of Threads in this Configuration. This Value is directly taken from the Symbol __cmos_numberOfThreads__.

#### *3.1.8.19 constexpr inline bool isValid_threadID(uint8 threadID) const*

Returns *true*, if the specified Thread ID is possibly valid. This does not necessarily mean, that there exists a Thread with that ID.

#### *3.1.8.20 static constexpr inline void invalidate(uint8& threadID)*

Invalides the Thread ID, so that the Function isValid_threadID returns *false*.

#### *3.1.8.21 inline void sleep()*

This Function makes the currently running Thread sleep until anything relevant happens.
Relevant Things:

- A Mail has been received
- An Event, that this Thread subscripted to, has been emitted

#### *3.1.8.22 uint16 sleep_untilEvent(uint32 timeout_ms = 0)*

This Function makes the currently running Thread sleep until any of the subscripted (and listened) Events is emitted. Returns the Event ID of the Event, that triggered the Wake Up.

#### *3.1.8.23 uint16 sleep_untilEvent(uint16 eventID, uint32 timeout_ms = 0)*

This Function makes the currently running Thread sleep until the specified Event is emitted. Returns the Event ID of the Event, that triggered the Wake Up.

#### *3.1.8.24 void sleep_untilMail(uint8 senderID)*

This Function makes the currently running Thread sleep until a Mail from the specified Sender is emitted. The Parameter *senderID* is the Thread ID of the Thread, that a Mail is expected from.

#### *3.1.8.25 inline void sleep_100us(uint32 units_of_100us)*

This Function is only available on the faster supported Processor Cores (at the Moment only the Cortex M7) and makes the currently running Thread sleep for the specified Amount of Time.

#### *3.1.8.26 inline void sleep_ms(uint32 ms)*

This Function makes the currently running Thread sleep for the specified Amount of Time.

#### *3.1.8.27 inline void sleep_sec(uint32 seconds)*

This Function makes the currently running Thread sleep for the specified Amount of Time.

#### *3.1.8.28 uint16 event_create()*

Creates an Event and returns its Event ID.

#### *3.1.8.29 feedback event_emit(uint16 eventID)*

Emits an Event for the specified Event ID to wake up all Subscribers.

#### *3.1.8.30 feedback event_listen(uint16 eventID)*

This Function only exists because of very fast Hardware. Sometimes an Event is emitted when a Hardware Module finishes its Operation (i.e. a DMA Controller), but the Hardware is so fast that the Thread setting the Hardware up, is not yet sleeping when the Event gets emitted. The Consequence is, that the Thread "misses" the Event because its not yet waiting for it and then goes sleeping (= waiting for an Event that already got emitted and possibly won't be emitted again). Therefore, the Thread will never wake up again.
The Solution to this Problem is to call this Function **before** setting up the Hardware Module. The Operating System will make a Note from this Call on, when the specified Event gets emitted. Calling sleep_untilEvent, sleep_untilEvent or sleep now will prevent the Thread from going to Sleep, but will directly return from that Function.
Only one Event ID can be listened to at the same Time.
**The Event ID that is listened to needs also to be subscripted to in Order for this Mechanism to work.**
Calling this Function multiple Times overwrites the last Event ID listened to with the new one.

#### *3.1.8.31 feedback event_unlisten()*

Calling this Function stops the Listening Mechanism for this Thread until event_listen is called again.

#### *3.1.8.32 feedback event_subscribe(uint16 eventID)*

Subscribes to an Event ID. Event IDs need to be subscripted to in Order to be woken up after calling sleep_untilEvent, sleep_untilEvent or sleep.
Multiple Event IDs can be subscripted to simultaneously.

#### *3.1.8.33 feedback event_unsubscribe()*

Removes all Event ID Subscriptions.

#### *3.1.8.34 feedback event_unsubscribe(uint16 eventID)*

Removes the Event ID Subscription of the specified Event ID.

#### *3.1.8.35 inline feedback send_mail(uint8 thread_ID, uint32 data)*

Sends a Mail to the specified Thread.

#### *3.1.8.36 inline bool is_mailAvailable() const*

Returns *true*, if there are any unread Mail in the Mail Inbox.

#### *3.1.8.37 inline Thread::s_mail read_mail()*

Returns the next unread Mail in the Mail Inbox. The returned Mail is marked as read and therefore removed from the Inbox.

#### 3.1.8.38 bool semaphore_doesExist(const void* semaphore) const

Returns true, if the specified Semaphore exists in the Operating System.

#### 3.1.8.39 uint8 semaphore_getOwner(const void* semaphore) const

Returns the Thread ID of the Thread currently owning the specified Semaphore. Returns *threadID_invalid*, if the specified Semaphore doesn't exist or the Semaphore is not locked.

#### 3.1.8.40 inline bool semaphore_isOwnedByRunningThread(const void* semaphore) const

Returns *true*, if the Semaphore has been locked by the currently running Thread.

#### 3.1.8.41 inline bool semaphore_isLocked(const void* semaphore) const

Returns *true*, if the Semaphore is currently locked.

#### 3.1.8.42 feedback semaphore_create(const void* semaphore)

Creates a new Semaphore in the Operating System. The intended Usage is to provide the Pointer to the Object that needs to be protected from simultaneous Access by multiple Threads (i.e. an Instance of a Serial Interface, CAN Bus, Memory Area, …) as Parameter *semaphore*. Returns *OK*, if the Semaphore has been successfully created and *FAIL* if the Value of semaphore is already being used for an existing Semaphore. This is because Semaphores need to be unique.

#### 3.1.8.43 feedback semaphore_erase(const void* semaphore)

Removes the specified Semaphore from the Operating System.

#### 3.1.8.44 feedback semaphore_lock(const void* semaphore)

Locks the specified Semaphore. If it is currently locked by another Thread, this Function sleeps until the Semaphore is available again and subsequently locks it. This Way it is guaranteed, that the Semaphore is locked, when this Function returns. Returns *OK*, if Semaphore is successfully locked, *FAIL* if the Semaphore does not exist.

#### 3.1.8.45 feedback semaphore_unlock(const void* semaphore)

Unlocks the specified Semaphore only if it was locked by the currently running Thread before. Returns *OK*, if the Unlock was successful, *FAIL* if the Semaphore was not locked by the currently running Thread or doesn't exist. On Thread Shutdown, all Semaphores that are currently locked by the Thread are automatically unlocked by the Operating System.

#### 3.1.8.46 void semaphore_transferAllOwnershipsToParentThread()

Transfers the Ownership of all Semaphores of the currently running Thread to the Parent Thread. This is particularly useful in Scenarios, where the Child Thread intends to shut down, but the Parent Thread needs to continue some Work on the Semaphore Objects.

#### 3.1.8.47 constexpr inline float thread_getLoadCpuPercent(uint8 thread_ID) const

This Function is only available on Cortex M3, Cortex M4 and Cortex M7.
Returns the CPU Usage of the specified Thread.

#### 3.1.8.48 constexpr inline float thread_getLoadStackPercent(uint8 thread_ID) const

This Function is only available on Cortex M3, Cortex M4 and Cortex M7.
Returns the Stack Usage of the specified Thread.

#### 3.1.8.49 constexpr inline float getLoadCpuPercent() const

This Function is only available on Cortex M3, Cortex M4 and Cortex M7.
Returns the total CPU Usage.

#### 3.1.8.50 constexpr inline float getLoadStackReservedAndUsedPercent() const

This Function is only available on Cortex M3, Cortex M4 and Cortex M7.
Returns the total Stack Usage, that is reserved by the Threads (via thread_create or thread_create) and is also really used by the Threads.

#### 3.1.8.51 constexpr inline float getLoadStackReservedAndUnusedPercent() const

This Function is only available on Cortex M3, Cortex M4 and Cortex M7.
Returns the total Stack Usage, that is reserved by the Threads (via thread_create or thread_create), but not used by the Threads.

#### 3.1.8.52 constexpr inline float getLoadStackPercent() const

This Function is only available on Cortex M3, Cortex M4 and Cortex M7.
Returns the total Stack Usage.

#### 3.1.8.53 constexpr inline float getLoadHeapPercent() const

This Function is only available on Cortex M3, Cortex M4 and Cortex M7.
Returns the total Heap Usage.

#### 3.1.8.54 inline feedback set_systemClock(uint32 clock)

Informs the Operating System about the Processor Clock to set up the System Timer correctly. This Function must be called everytime **before** changing the Processor Clock in order to provide correct Timing by the sleep Functions.

#### 3.1.8.55 constexpr inline volatile uint64& get_ticks()

Returns the Number of Ticks since run has been called. 1 Second equals *c_clock_systick* Ticks. This Reference can be used to make more precise Timing Calculations than with the sleep Functions.

#### 3.1.8.56 constexpr inline NVIC& get_nvic()

Returns a Reference to the NVIC Controller.

#### 3.1.8.57 inline uint32 get_id_core() const

Returns the Processor ID.

## 3.1.9 Math

### *3.1.9.1 constexpr inline float nanFloat()*

Returns NaN (Not a Number).

### *3.1.9.2 constexpr inline double nanDouble()*

Returns NaN (Not a Number).

### *3.1.9.3 constexpr inline float infFloat()*

Returns inf (infinity).

### *3.1.9.4 constexpr inline double infDouble()*

Returns inf (infinity).

### *3.1.9.5 constexpr inline bool is_nan(float number)*

Returns *true*, if the Number equals NaN.

### *3.1.9.6 constexpr inline bool is_nan(double number)*

Returns *true*, if the Number equals NaN.

### *3.1.9.7 constexpr inline bool is_inf(float number)*

Returns *true*, if the Number equals inf.

### *3.1.9.8 constexpr inline bool is_inf(double number)*

Returns *true*, if the Number equals inf.

### *3.1.9.9 constexpr inline float round(float number, uint32 postDotLetters)*

Rounds the Number to postDotLetters Decimals after the Point using the "Rounding half up" Algorithm.

### *3.1.9.10 constexpr inline double round(double number, uint32 postDotLetters)*

Rounds the Number to postDotLetters Decimals after the Point using the "Rounding half up" Algorithm.

### *3.1.9.11 constexpr inline dataType radToDeg(dataType radiant)*

Returns the to-Degree-converted Value of *radiant*.

### *3.1.9.12 constexpr inline dataType degToRad(dataType degree)*

Returns the to-Radiant-converted Value of *degree*.

### *3.1.9.13 constexpr inline float log(uint32 base, float number)*

Returns $log_{base}(number)$.

#### 3.1.9.14 constexpr inline double log(uint32 base, double number)

Returns $log_{base}(number)$.

#### 3.1.9.15 constexpr inline float log2(float number)

Returns $log_2(number)$.

#### 3.1.9.16 constexpr inline double log2(double number)

Returns $log_2(number)$.

#### 3.1.9.17 constexpr inline uint32 log2(uint32 number)

Returns $log_2(number)$.

#### 3.1.9.18 constexpr inline dataType absolute(dataType value)

Returns the absolute Value of a Number.

#### 3.1.9.19 constexpr dataType_result power(dataType_base base, int32 exponent)

Returns $base^{exponent}$.

#### 3.1.9.20 constexpr uint32 numberOfDigits(dataType number)

Returns the Number of Digits in Decimal-System that the Number needs to be represented.

#### 3.1.9.21 constexpr inline int8 sign(dataType number)

Returns "-1" if the Number is below 0. Returns "1" otherwise.

#### 3.1.9.22 constexpr bool isAngleBetween(dataType angle, dataType angleStart, dataType angleEnd)

The Vectors from 0 with the Angles *angleStart* and *angleEnd* to the x-Coordinate form a Triangle in the Unit Circle. This Function returns *true*, if the Vector from 0 with the Angle *angle* to the x-Coordinate crosses the Triangle.

#### 3.1.9.23 constexpr uint8 digit(dataType number, int32 digitPosition)

Returns the Digit at Position *digitPosition* of *number*, when *number* is represented in Decimal Form.

#### 3.1.9.24 constexpr inline dataType min(dataType number1, dataType number2)

Returns the most negative Number.

#### 3.1.9.25 constexpr inline dataType min(dataType number1, dataType number2, dataType number3)

Returns the most negative Number.

#### 3.1.9.26 constexpr inline dataType min(dataType number1, dataType number2, dataType number3, dataType number4)

Returns the most negative Number.

#### *3.1.9.27 constexpr inline dataType min(dataType number1, dataType number2, dataType number3, dataType number4, dataType number5)*

Returns the most negative Number.

#### *3.1.9.28 constexpr inline dataType max(dataType number1, dataType number2)*

Returns the most positive Number.

#### *3.1.9.29 constexpr inline dataType max(dataType number1, dataType number2, dataType number3)*

Returns the most positive Number.

#### *3.1.9.30 constexpr inline dataType max(dataType number1, dataType number2, dataType number3, dataType number4)*

Returns the most positive Number.

#### *3.1.9.31 constexpr inline dataType max(dataType number1, dataType number2, dataType number3, dataType number4, dataType number5)*

Returns the most positive Number.

#### *3.1.9.32 dataType isPointRightOrLeft(Vector2<dataType> A, Vector2<dataType> B, Vector2<dataType> point)*

Creates a Line from Point A to Point B. The Direction of that Line is from A to B.
Returns a Value
< 0, if the Point *point* lies on the left Side
> 0, if the Point *point* lies on the right Side
of that Line when moving from A to B.
Returns 0, if *point* lies on that Line.

## 3.1.10 NVIC

### *3.1.10.1 inline feedback init()*

Initializes the NVIC.
Not to be used by the User.

### *3.1.10.2 inline feedback enable(uint16 interrupt)*

Enables the specified Interrupt.
Use together with the Namespace *Interrupt*.

### *3.1.10.3 inline feedback disable(uint16 interrupt)*

Disables the specified Interrupt.
Use together with the Namespace *Interrupt*.

### *3.1.10.4 inline feedback setPending(uint16 interrupt)*

Sets the specified Interrupt to Pending.
Use together with the Namespace *Interrupt*.

### *3.1.10.5 inline feedback setPriority(uint16 interrupt, uint8 priority)*

Sets the Priority of the specified Interrupt. The smaller the Priority, the higher the Priority.
Set priority between "1" and "*c_priority_max* – 1" to not interfere with the Operating System.
Use together with the Namespace *Interrupt*.

### *3.1.10.6 inline feedback clearPending(uint16 interrupt)*

Clears the Pending State of the specified Interrupt.
Use together with the Namespace *Interrupt*.

### *3.1.10.7 inline bool isEnabled(uint16 interrupt)*

Returns *true*, if the specified Interrupt is enabled – *false* otherwise.
Use together with the Namespace *Interrupt*.

### *3.1.10.8 inline bool isPending(uint16 interrupt)*

Returns *true*, if the specified Interrupt is pending – *false* otherwise.
Use together with the Namespace *Interrupt*.

### *3.1.10.9 inline bool isActive(uint16 interrupt)*

Returns *true*, if the specified Interrupt is active – *false* otherwise.
Use together with the Namespace *Interrupt*.
Available only on Cortex M3, Cortex M4 and Cortex M7.

### *3.1.10.10 inline uint32 set_basePriority(uint8 basePriority)*

Sets the Base Priority, below which (priority-wise, not value-wise) all Interrupts are inhibited.


#### *3.1.10.11 inline void enable_configurablePriorityExceptions()*

Enables all Exceptions with configurable Priority.

#### *3.1.10.12 inline void disable_configurablePriorityExceptions()*

Disables all Exceptions with configurable Priority.

#### *3.1.10.13 inline void enable_allExceptions()*

Enables all Exceptions.

#### *3.1.10.14 inline void disable_allExceptions()*

Disables all Exceptions.


### 3.1.11 String

## 3.1.12 Thread

### *3.1.12.1 s_mail*

Structure to represent Mails that can be sent between Threads.
Field *data*: 4 Byte Payload Data Field
Field *type*: Mail Type
Field *senderID*: Thread ID of sending Thread

### *3.1.12.2 e_mailType*

Type to indicate Purpose of Mail:

- *NORMAL*: Normal Mail between Threads
- *EXIT*: Shutdown Message for soft Shutdown (see thread_shutdown)

### *3.1.12.3 constexpr inline bool is_valid() const*

Returns *true*, if the Thread is configured properly to run in the Operating System.

## 3.1.13 Time

### *3.1.13.1 constexpr inline Time()*

Creates a new Time Object initialized at 01.01.2000 – 00:00:00.

### *3.1.13.2 constexpr inline Time(uint32 seconds)*

Creates a new Time Object without a Date.
Date is set to 00.00.0000.
Time is calculated by the Amount of Seconds.

### *3.1.13.3 constexpr inline Time(uint8 _second, uint8 _minute, uint32 _hour, uint8 _day, uint8 _month, uint16 _year)*

Creates a new Time Object with the specified Values.

### *3.1.13.4 e_weekday get_weekday() const*

Not implemented yet. Always returns *INVALID*.

### *3.1.13.5 constexpr inline Time operator++(int seconds)*

Increments the Time and Date by *seconds* Seconds, taking into Account Leap Years and the different Lengths of Months.


### 3.1.14 UART


## 3.2 Graphics Library

## 3.3 Filesystem Library

### 3.3.1 Interfaces

#### *3.3.1.1 I_Directory*

#### *3.3.1.2 I_Drive*

#### *3.3.1.3 I_Entry*

#### *3.3.1.4 I_File*

#### *3.3.1.5 I_Volume*

### 3.3.2 Filesystem

### 3.3.3 FAT32

#### *3.3.3.1 Bootsector_FAT32*

#### *3.3.3.2 Directory_FAT32*

#### *3.3.3.3 File_FAT32*

#### *3.3.3.4 MasterBootRecord*

#### *3.3.3.5 Volume_FAT32*


## 3.4 ICD Library

### 3.4.1 24LC02B

### 3.4.2 AD5175

### 3.4.3 BQ25887

### 3.4.4 DP83825I

### 3.4.5 MB85RC16

### 3.4.6 MCP3427

### 3.4.7 MCP3428

### 3.4.8 MCP23016

### 3.4.9 STC3100


# 4 Revision History

| Revision | Date | Changes |
|---|---|---|
| 0 | 10/2025 | Document created |