EE\_VCU Framework

EVD

**VCU Framework interfaces description**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Notes** | **Date** | **Author** |
| 01 | First draft | 09/05/2019 | Nicolò Bosio |

**Table of Contents**

1. Introduction 3

1.1 Purpose 3

1.2 Scope 3

2. Framework description 4

DATA TYPES 4

IDE/STANDARD 4

VERSION 4

SCHEDULING 4

POWER SUPPLY 6

Init: 6

Use: 6

ANALOG INPUTS 7

Init: 7

Use: 7

DIGITAL INPUTS 8

Init: 8

Configuration: 8

Use: 12

DIGITAL OUTPUTS 13

Init: 13

Use: 13

PWM OUTPUTS 15

Init: 15

Use: 15

MOTOR 17

Init: 17

Use: 17

IMU 19

Init: 19

Use: 19

CAPUTRE INPUTS 20

Init: 20

Use: 20

EEPROM 21

Init: 21

Use: 21

CAN 22

Init: 22

Use: 24

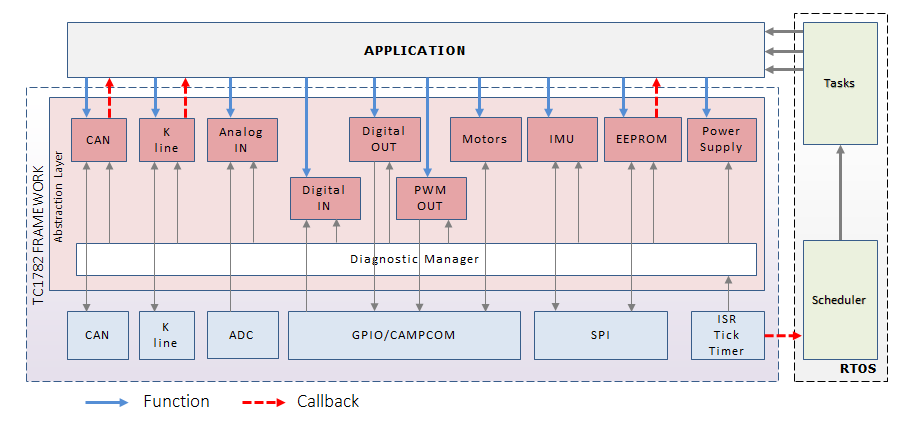
# 1. Introduction

## 1.1 Purpose

The goal of this document is to present an understandable overview over the VCU\_FMW developed on the TC1782 INFINEON microcontroller, to allow a software developer to call its function on a high level application without worrying about the low level implementation.

## 1.2 Scope

Below is a representation of the logical interconnection of the entire application.



The subject of this document is the part indicated as TC1782 framework and it constitutes the interface between the application and the hardware.

The framework doesn’t have constraints related to the use or not of a RTOS, but will make a 1msec interrupt available to the application.

This can be used by the application to manage a simple internal multirate scheduler or a real realtime operating system (RTOS).

The framework works as a bridge between an high level application and the "Micro Abstraction Layer".

This to allows the reuse of the high level application on new hardware in case of replacement of the microcontroller.

This is valid in the event that the features remain the same (number and type of IO). Should the number and type of IOs vary, the application will inevitably also have to be revised.

# 2. Framework description

## DATA TYPES

The framework redefines main data types as follows:

*boolean types 🡪 bool\_t*

*single type reals 🡪 float32\_t*

*8 bits signed integers 🡪 int8\_t*

*8 bits unsigned integers 🡪 uint8\_t*

*16 bits signed integers 🡪 int16\_t*

*16 bits unsigned integers 🡪 uint16\_t*

*32 bits signed integers 🡪 int32\_t*

*32 bits unsigned integers 🡪 uint32\_t*

These types are defined in the ee\_types.h header in the lib\ee\_types folder.

All interfaces to and from the high level application use the types described above.

## IDE/STANDARD

The framework is provided as a project for TASKING v6.3r1 compiler. Compatibility with other versions and or compilers is not guaranteed.

The framework is written in accordance with the ANSI / ISO C99 standard.

Indications of the MISRA C standard are respected where possible.

Prototypes of all framework functions available for an high level application are in the header file “VcuFmw.h”.

## VERSION

The framework version is defined in the header VcuFmw\_Private.h with

***#define VCU\_FRAMEWORK\_VER 0xAABBCCCC***

Where AA is the major version, BB the minor version and CCCC the build version.

The current version is 01010001.

## SCHEDULING

In order to provide an accurate TIC TIMER, the framework invokes a callback function at a rate of 1 ms.

This callback must be implemented in the application with the following prototype:

***void APP\_Tick\_callback(void);***

This callback is called inside a system interrupt thus it must be very fast (shorter than 20us).

## POWER SUPPLY

### Init:

The following function is needed to initialize all function and data correlated to the power supply on the board:

***void POWER\_Init(void);***

### Use:

The framework defines some callback functions that have to be implemented according to the following prototype:

***void POWER\_xxx\_callback(void);***

In particular at KEY\_IN switch on the following function will be called:

***void POWER\_SwitchON\_callback(void);***

and at KEY\_IN switch off the following function will be called:

***void POWER\_SwitchOFF\_callback(void);***

This function shall be used to trigger all functionalities needed at the end of the application (i.e. error codes saving into EEPROM). Particular care shall be put in the implementation because this function is called inside an interrupt.

Subsequently the switch-off consent function must be invoked to end the POWER LATCH phase and turn off the VCU.

This is done through the following function:

***void POWER\_SwitchOFF(void);***

If the user turns KEY\_IN back on before the application has called the <*POWER\_SwitchOFF*> function the framework will invoke again the <*POWER\_SwitchON\_callback*>. This should be managed properly by the application.

## ANALOG INPUTS

### Init:

It is necessary to call the following initialization function before using the functions for reading the analog channels:

***void ANALOG\_Init(void);***

### Use:

To read an analog channel the following function has to be called:

***float32\_t ANALOG\_Init(uint16\_t channel);***

The function returns the channel voltage in [V] except for the pcb temperature which is measured in [°C].

The argument passed to this function shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| AN\_1 | Analog channel 1 |
| AN\_2 | Analog channel 2 |
| AN\_3 | Analog channel 3 |
| AN\_4 | Analog channel 4 |
| AN\_5 | Analog channel 5 |
| AN\_6 | Analog channel 6 |
| AN\_7 | Analog channel 7 |
| SAFE\_SW\_01 | Analog safe switch 1 |
| SAFE\_SW\_02 | Analog safe switch 2 |
| SAFE\_SW\_03 | Analog safe switch 3 |
| VREF\_5\_SEN1 | 5V power supply 1 (for sensors) |
| VREF\_5\_SEN2 | 5V power supply 2 (for sensors) |
| VREF\_5\_AUX | Auxiliary 5V power supply |
| VREF\_12 | 12V reference |
| VPOWER\_CAN | Can bus power supply |
| VBB\_LP | Low power 12V power supply |
| VBB\_HP | High power 12V power supply |
| PCB\_TEMP | PCB temperature |

## DIGITAL INPUTS

### Init:

It is necessary to call the following initialization function before using the functions for reading the digital channels:

***void DIGIN\_Init(void);***

### Configuration:

After the initialization of all framework modules the digital input module can be configured using the following function:

***bool\_t DIGIN\_Setup(TIC12400\_SETUP\_Type\* pSetup);***

The <*TIC12400\_SETUP\_Type*> type is defined as follows:

typedef struct

{

uint8\_t SwitchMode[10]; // TIC12400\_MODE\_xxx

uint8\_t PollingPeriod; // TIC12400\_POLLING\_TIME\_xxx

uint8\_t Debounce; // TIC12400\_DEBOUNCE\_xxx

uint8\_t Threshold\_0\_3; // TIC12400\_THRESHOLD\_xxx

uint8\_t Threshold\_4\_7; // "

uint8\_t Threshold\_8\_11; // "

uint8\_t WettingCurrent\_0\_1; // TIC12400\_WETTING\_CURR\_xxx

uint8\_t WettingCurrent\_2\_3; // "

uint8\_t WettingCurrent\_4; // "

uint8\_t WettingCurrent\_5; // "

uint8\_t WettingCurrent\_6\_7; // "

uint8\_t WettingCurrent\_8\_9; // "

uint8\_t WettingTime; // TIC12400\_WETTING\_TIME\_xxx

uint8\_t CleanTime; // TIC12400\_CLEAN\_TIME\_xxx

uint8\_t CleanCurrent\_0\_5; // TIC12400\_CLEAN\_CURR\_xxx

uint8\_t CleanCurrent\_6\_10; // "

uint16\_t ClearCurrentEnable; // BitMask for Clean Current Enable

}

TIC12400\_SETUP\_Type;

<*SwitchMode*> is a vector of 10 elements used to configure the switching mode for each digital input channel. Element 0 of the vector corresponds to the digital input 1 and element 9 to the digital input 10.

The values to be used to set this vector shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_MODE\_disabled | The channel is disabled |
| TIC12400\_MODE\_SWTOGND | The smart chip inside the VCU acts as a current source for the channel |
| TIC12400\_MODE\_SWTOSUPPLY | The smart chip inside the VCU acts as a current sink for the channel |

<*PollingPeriod*> is a variable used to set the period to acquire the digital inputs.

The values to be used to set this vector shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_POLLING\_disabled | Continuous mode |
| TIC12400\_POLLING\_TIME\_2\_ms | Sampling every 2 ms |
| TIC12400\_POLLING\_TIME\_4\_ms | Sampling every 4 ms |
| TIC12400\_POLLING\_TIME\_8\_ms | Sampling every 8 ms |
| TIC12400\_POLLING\_TIME\_16\_ms | Sampling every 16 ms |
| TIC12400\_POLLING\_TIME\_32\_ms | Sampling every 32 ms |
| TIC12400\_POLLING\_TIME\_48\_ms | Sampling every 48 ms |
| TIC12400\_POLLING\_TIME\_64\_ms | Sampling every 64 ms |
| TIC12400\_POLLING\_TIME\_128\_ms | Sampling every 128 ms |
| TIC12400\_POLLING\_TIME\_256\_ms | Sampling every 256 ms |
| TIC12400\_POLLING\_TIME\_512\_ms | Sampling every 512 ms |
| TIC12400\_POLLING\_TIME\_1024\_ms | Sampling every 1024 ms |
| TIC12400\_POLLING\_TIME\_2048\_ms | Sampling every 2048 ms |
| TIC12400\_POLLING\_TIME\_4096\_ms | Sampling every 4096 ms |

<*Debounce*> is a variable used to set the number of consequent samples that have to be read before considering the switch valid.

The values to be used to set this vector shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_DEBOUNCE\_disable | No debounce |
| TIC12400\_DEBOUNCE\_2\_cycle | 2 consequent samples |
| TIC12400\_DEBOUNCE\_3\_cycle | 3 consequent samples |
| TIC12400\_DEBOUNCE\_4\_cycle | 4 consequent samples |

<*Threshold\_m\_n*> is a variable used to set the threshold to detect the switch of an input. One variable sets inputs from m to n. Three groups are present: 0 to 3, 4 to 7 and 8 to 11 (input with index 10 and 11 are manageable from the smart chip but are not connected on the VCU).

The values to be used to set these variables shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_THRESHOLD\_2\_volt | 2 V threshold |
| TIC12400\_THRESHOLD\_2\_7\_volt | 2,7 V threshold |
| TIC12400\_THRESHOLD\_3\_volt | 3 V threshold |
| TIC12400\_THRESHOLD\_4\_volt | 4 V threshold |

<*WettingCurrent\_m\_n*> is a variable used to set the wetting current to be injected/drawn on/from an input. One variable sets channels from m to n. Three groups are present: 0 to 1, 2 to 3, 4 (alone), 5 (alone), 6 to 7 and 8 to 9 (input with index 10 and 11 are manageable from the smart chip but are not connected on the VCU). The values to be used to set this variable shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_WETTING\_CURR\_disabled | No wetting current |
| TIC12400\_WETTING\_CURR\_1\_ma | 1 mA wetting current |
| TIC12400\_WETTING\_CURR\_2\_ma | 2 mA wetting current |
| TIC12400\_WETTING\_CURR\_5\_ma | 5 mA wetting current |
| TIC12400\_WETTING\_CURR\_10\_ma | 10 mA wetting current |
| TIC12400\_WETTING\_CURR\_15\_ma | 15 mA wetting current |

<*WettingTime*> is a variable used to set the time for which on each period the wetting current is applied.

The values to be used to set this variable shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_WETTING\_TIME\_64\_us | Wetting current applied for 64 us |
| TIC12400\_WETTING\_TIME\_128\_us | Wetting current applied for 128 us |
| TIC12400\_WETTING\_TIME\_192\_us | Wetting current applied for 192 us |
| TIC12400\_WETTING\_TIME\_256\_us | Wetting current applied for 256 us |
| TIC12400\_WETTING\_TIME\_320\_us | Wetting current applied for 320 us |
| TIC12400\_WETTING\_TIME\_384\_us | Wetting current applied for 384 us |
| TIC12400\_WETTING\_TIME\_448\_us | Wetting current applied for 448 us |
| TIC12400\_WETTING\_TIME\_512\_us | Wetting current applied for 512 us |
| TIC12400\_WETTING\_TIME\_640\_us | Wetting current applied for 640 us |
| TIC12400\_WETTING\_TIME\_768\_us | Wetting current applied for 768 us |
| TIC12400\_WETTING\_TIME\_896\_us | Wetting current applied for 896 us |
| TIC12400\_WETTING\_TIME\_1024\_us | Wetting current applied for 1024 us |
| TIC12400\_WETTING\_TIME\_2048\_us | Wetting current applied for 2048 us |

<*CleanTime*> is a variable used to set the time for which on each period the clean current is applied.

The values to be used to set this variable shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_CLEAN\_TIME\_64\_us | Clean current applied for 64 us |
| TIC12400\_CLEAN\_TIME\_128\_us | Clean current applied for 128 us |
| TIC12400\_CLEAN\_TIME\_192\_us | Clean current applied for 192 us |
| TIC12400\_CLEAN\_TIME\_256\_us | Clean current applied for 256 us |
| TIC12400\_CLEAN\_TIME\_320\_us | Clean current applied for 320 us |
| TIC12400\_CLEAN\_TIME\_384\_us | Clean current applied for 384 us |
| TIC12400\_CLEAN\_TIME\_448\_us | Clean current applied for 448 us |
| TIC12400\_CLEAN\_TIME\_512\_us | Clean current applied for 512 us |

<*CleanCurrent\_m\_n*> is a variable used to set the clean current to be injected/drawn on/from an input. One variable sets channels from m to n. Two groups are present: 0 to 5 and 6 to 10 (input with index is manageable from the smart chip but is not connected on the VCU). The values to be used to set this variable shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_CLEAN\_CURR\_10\_ma | 10 mA clean current |
| TIC12400\_CLEAN\_CURR\_15\_ma | 15 mA clean current |

<*CleanCurrentEnable*> is a bitmask used to enable the clean current on any particular channel. The values to be used to set this variable shall be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| TIC12400\_CLEAN\_NONE | No clean current applied |
| TIC12400\_CLEAN\_ALL | Clean current applied to all channels |
| TIC12400\_CLEAN\_SW0 | Clean current applied to switch 0 |
| TIC12400\_CLEAN\_SW1 | Clean current applied to switch 1 |
| TIC12400\_CLEAN\_SW2 | Clean current applied to switch 2 |
| TIC12400\_CLEAN\_SW3 | Clean current applied to switch 3 |
| TIC12400\_CLEAN\_SW4 | Clean current applied to switch 4 |
| TIC12400\_CLEAN\_SW5 | Clean current applied to switch 5 |
| TIC12400\_CLEAN\_SW6 | Clean current applied to switch 6 |
| TIC12400\_CLEAN\_SW7 | Clean current applied to switch 7 |
| TIC12400\_CLEAN\_SW8 | Clean current applied to switch 8 |
| TIC12400\_CLEAN\_SW9 | Clean current applied to switch 9 |

For example, if you want to apply the clean current only to sw0 and 3 you’d have to set <*CleanCurrentEnable>* to TIC12400\_CLEAN\_SW0|TIC12400\_CLEAN\_SW3

### Use:

To read a digital channel the following function has to be called:

***bool\_t DIGIN\_Get(uint16\_t channel);***

The function returns the physical state of the input input switch connected to each channel where true means “active” and false means “inactive”. For switches configured as *SWTOSUPPLY* the active state is when the input is high, while for switches configured as *SWTOGND* the active state corresponds to a low level.

## DIGITAL OUTPUTS

### Init:

It is necessary to call the following initialization function before utilizing the digital outputs module functions:

***void DIGOUT\_Init(void);***

### Use:

The VCU has got 7 high power high side channels, 6 low power high side channels and one low side channel.

To set a digital output channel you have to use the following function:

***void DIGOUT\_Set(uint16\_t channel, bool\_t state);***

The function accepts two parameters. The first is channel on which to act, the second is the output switching level, where FALSE corresponds to OPEN output, and TRUE to closed output.

The channel values have to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| HSD\_LP\_01 | High side low power channel 1 |
| HSD\_LP\_02 | High side low power channel 2 |
| HSD\_LP\_03 | High side low power channel 3 |
| HSD\_LP\_04 | High side low power channel 4 |
| HSD\_LP\_05 | High side low power channel 5 |
| HSD\_LP\_06 | High side low power channel 6 |
| HSD\_HP\_01 | High side high power channel 1 |
| HSD\_HP\_02 | High side high power channel 2 |
| HSD\_HP\_02 | High side high power channel 2 |
| HSD\_HP\_03 | High side high power channel 3 |
| HSD\_HP\_04 | High side high power channel 4 |
| HSD\_HP\_05 | High side high power channel 5 |
| HSD\_HP\_06 | High side high power channel 6 |
| HSD\_HP\_07 | High side high power channel 7 |
| LSD\_LP\_01 | Low side low power channel 1 |

The framework automatically performs a diagnostic on the state of output channels. To read the results of this diagnostic the following function has to be called:

***uint16\_t DIGOUT\_Diag(uint16\_t channel);***

The return values for this function have to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| DIAG\_SCGND | Short circuit to GND |
| DIAC\_OC | Open circuit |
| DIAG\_UNSUPPORTED | Unsupported |

The diagnostic function is not present on the low side channel, so calling the <*DIGOUT\_Diag*> function on this channel will always return *DIAG\_UNSUPPORTED*.

## PWM OUTPUTS

### Init:

Before utilizing the pwm module it is necessary to initialize it with the following function:

***void PWM\_Init(uint16\_t frequency\_range);***

This function accepts one argument which is the frequency range shared by all outputs. It has to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| PWM\_FREQRANGE\_10\_625 | Frequency range from 10 [Hz] to 625 [Hz] |
| PWM\_FREQRANGE\_20\_1250 | Frequency range from 20 [Hz] to 1250 [Hz] |
| PWM\_FREQRANGE\_40\_2500 | Frequency range from 40 [Hz] to 2500 [Hz] |
| PWM\_FREQRANGE\_80\_5000 | Frequency range from 80 [Hz] to 5000 [Hz] |
| PWM\_FREQRANGE\_160\_10000 | Frequency range from 160 [Hz] to 10000 [Hz] |

### Use:

The VCU has got 2 high power pwm channels (shared with HSD\_HP\_01 and HSD\_HP\_02) and 2 low power pwm channels (shared with HSD\_LP\_01 and HSD\_LP\_02) and one low side channel.

It is possible to set the frequency of each channel independently with the following function:

***void PWM\_SetFrequency(uint16\_t channel, float32\_t value);***

It accepts two input parameters. The first is the channel which value has to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| HSD\_LP\_01 | High side low power channel 1 |
| HSD\_LP\_02 | High side low power channel 2 |
| HSD\_HP\_01 | High side high power channel 1 |
| HSD\_HP\_02 | High side high power channel 2 |

The second parameter is the frequency value for a specific channel expressed in [Hz].

The framework validates the frequency value passed by the application and eventually saturates it in the range set with the <*PWM\_Init*> function.

To set the duty cycle for a PWM channel the following function is needed:

***void PWM\_SetDutyCycle(uint16\_t channel, float32\_t value);***

The second parameter <*value*> is the duty cycle expressed in [%]. The framework saturates this parameter between 0% and 100%.

The framework automatically performs a diagnostic on the state of output PWM channels. To read the results of this diagnostic the following function has to be called:

***uint16\_t PWM\_Diag(uint16\_t channel);***

The return values for this function have to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| DIAG\_SCGND | Short circuit to GND |
| DIAC\_OC | Open circuit |
| DIAG\_UNSUPPORTED | Unsupported |

## MOTOR

### Init:

Before utilizing the functions of the motor module it is necessary to initialize it with the following function:

***void MOTOR\_Init(void);***

After initialization all motor channels are disabled and only diagnostic is automatically executed by the framework.

### Use:

The following function sets the state of a motor channel:

***void MOTOR\_Enable(uint16\_t channel, bool\_t state);***

The first parameter is the channel to be activated/deactivated and its values are to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| MOTOR\_01 | Motor n 1 |
| MOTOR\_02 | Motor n 2 |

The second parameter is the state and when set to true means “activate”, when false “deactivate”.

The <*Motor\_Enable*> function works on both redunded lines ENABLE\_H and ENABLE\_L of the selected channel.

The following function has to be called to monitor the speed of a specific channel:

***float32\_t MOTOR\_GetSpeed(uint16\_t channel);***

The value returned by this function is expressed in [rad/s].

The following function has to be called to read the direction of a specific channel:

***float32\_t MOTOR\_GetDirection(uint16\_t channel);***

where the returned value is (+1) when the rotation is counterclockwise and negative (-1) when the rotation is clockwise.

The framework automatically performs a low level diagnostic with a period of 500 ms.

To read its results the following function has to be called:

***uint16\_t MOTOR\_Diag(uint16\_t channel);***

The return values for this function have to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| DIAG\_MOTOR\_SCGND\_L | Short to GND on ENABLE\_L |
| DIAG\_MOTOR\_SCBAT\_L | Short to VBAT on ENABLE\_L |
| DIAG\_MOTOR\_SCGND\_H | Short to GND on ENABLE\_H |
| DIAG\_MOTOR\_SCBAT\_H | Short to VBAT on ENABLE\_H |
| DIAG\_MOTOR\_OC | Open circuit |

## IMU

### Init:

Before utilizing the function that reads the IMU it is necessary to initialize the module with the following function:

***void IMU\_Init(void);***

### Use:

The following function is needed to read an IMU channel:

***float32\_t IMU\_Get(uint16\_t channel);***

The values returned by this function depends on the channel it was called upon. On the next table see value for the channel and unit for the unit of measurement of the returned value.

|  |  |  |
| --- | --- | --- |
| Value | Unit | Description |
| VCU\_ACC\_X | m/s2 | X axis acceleration referred to the VCU |
| VCU\_ACC\_Y | m/s2 | Y axis acceleration referred to the VCU |
| VCU\_ACC\_Z | m/s2 | Z axis acceleration referred to the VCU |
| VCU\_GYRO\_X | deg/s | Rotation speed on the X axis referred to the VCU |
| VCU\_GYRO\_Y | deg/s | Rotation speed on the Y axis referred to the VCU |
| VCU\_GYRO\_Z | deg/s | Rotation speed on the Z axis referred to the VCU |

## CAPTURE INPUTS

### Init:

Before utilizing the function to read the capture inputs it is necessary to initialize the module with the following function:

***void CAPIN\_Init(void);***

### Use:

The following function returns the frequency in [Hz] of a switching input:

***float32\_t CAPIN\_GetFrequency(uint16\_t channel);***

The duty cycle expressed in [%] of the input signal is returned by the following function:

***float32\_t CAPIN\_GetDutyCycle(uint16\_t channel);***

The argument of these functions is the channel to read and has to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| IC\_01 | Capture input 1 |
| IC\_02 | Capture input 2 |

The framework automatically performs a diagnostic on the capture inputs. To get the results of this diagnostic the following function has to be called:

***uint16\_t CAPIN\_Diag(uint16\_t channel);***

|  |  |
| --- | --- |
| Value | Description |
| DIAG\_SCGND | Short circuit to GND |
| DIAC\_SCVBAT | Short circuit to VBAT |
| DIAC\_OC | Open circuit |
| DIAG\_UNSUPPORTED | Unsupported |

## EEPROM

### Init:

Before using the functions to manage the external EEPROM memory the init function have to be called:

***void EEPROM\_Init(void);***

### Use:

The framework provides three functions to manage the EEPROM, respectively for reading, writing and deleting the memory:

***bool\_t EEPROM\_Read(uint32\_t addr, uint32\_t size, uint8\_t\* pBuffer);***

***bool\_t EEPROM\_Write(uint32\_t addr, uint32\_t size, uint8\_t\* pBuffer);***

***bool\_t EEPROM\_Erase(uint32\_t addr, uint32\_t size);***

The parameter <*addr*> is the start address to access the memory and <*size*> is the dimension in bytes to write/read on/from the memory. Both these parameters should be 128 bytes aligned.

The return value of these functions is true in case the operation succeded and false otherwise.

## CAN

### Init:

Before using the functions to communicate on CAN bus the init function have to be called:

***void CAN\_Init(CAN\_CONFIG\_Type\* pConfig);***

The <CAN\_CONFIG\_Type> type is defined as follows:

typedef struct

{

struct

{

uint32\_t BitrateReg; // BitRate

}

Channel[\_CAN\_CHANNELS\_NUMBER];

CAN\_MESSAGE\_Type Message[\_CAN\_MESSAGES\_NUMBER]; // available slots

}

CAN\_CONFIG\_Type;

The <*Channel*> vector is used to configure the baudrate for the CAN channels.

|  |  |
| --- | --- |
| Value | Description |
| CAN\_BITRATE\_250KBIT | 250 Kbit/s |
| CAN\_BITRATE\_500KBIT | 500 Kbit/s |
| CAN\_BITRATE\_1MBIT | 1 Mbit/s |

The <CAN\_MESSAGE\_Type> type is defined as follows:

typedef struct

{

uint32\_t CanId;

union

{

uint16\_t RAW;

struct

{

uint16\_t Channel :2;

uint16\_t Extended :1;

uint16\_t Transmit :1;

uint16\_t Length :4;

uint16\_t dummy :8;

}

Flags;

}

Config;

CANRX\_CALLBACK\_Type pRxCallback;

}

CAN\_MESSAGE\_Type;

Where <*CanId*> is the address of a CAN message and <*Channel*> represents the can bus on which that particular CAN message is received/transmitted. <*Channel*> values have to be taken from the following table:

|  |  |
| --- | --- |
| Value | Description |
| CAN\_CHANNEL\_1 | Can channel n 1 |
| CAN\_CHANNEL\_2 | Can channel n 2 |
| CAN\_CHANNEL\_3 | Can channel n 3 |

The <*Exended*> field determines if the message uses the extended or standard CAN frame:

|  |  |
| --- | --- |
| Value | Description |
| CAN\_MSG\_STD11BIT | Use standard frame |
| CAN\_MSG\_STD29BIT | Use extended frame |

The <*Transmit*> field determines if the message has to be transmitted or received:

|  |  |
| --- | --- |
| Value | Description |
| CAN\_MSG\_RECEIVE | Receive message |
| CAN\_MSG\_TRANSMIT | Transmit message |

The <*Lenght*> field determines the length in bytes of the data field for that message:

|  |  |
| --- | --- |
| Value | Description |
| CAN\_MSG\_LEN0BYTE | 0 bytes data message |
| CAN\_MSG\_LEN1BYTE | 1 bytes data message |
| CAN\_MSG\_LEN2BYTE | 2 bytes data message |
| CAN\_MSG\_LEN3BYTE | 3 bytes data message |
| CAN\_MSG\_LEN4BYTE | 4 bytes data message |
| CAN\_MSG\_LEN5BYTE | 5 bytes data message |
| CAN\_MSG\_LEN6BYTE | 6 bytes data message |
| CAN\_MSG\_LEN7BYTE | 7 bytes data message |

The pointer to the callback function < *pRxCallback*> has to be used for receive message only, leave NULL otherwise.

The <*CANRX\_CALLBACK\_Type>* type is defined as follows:

typedef void (\*CANRX\_CALLBACK\_Type)( uint32\_t id , uint8\_t\* pData );

When a message is received, the associated callback function will have the corresponding id and received data on <*id*> and <*pData*> respectively.

Here’s an example for the setup of a receive message:

{ 0x80 ,

{ CAN\_CHANNEL\_3|CAN\_MSG\_STD11BIT|CAN\_MSG\_RECEIVE|CAN\_MSG\_LEN8BYTE } , &VcuTask\_Message\_Rx

}

In this case the framework calls the function <*VcuTask\_Message\_Rx*> when a message is received on CAN3 ID 0x80.

The framework allows to setup 128 different CAN messages.

### Use:

To transmit a CAN message it is necessary to call the following function:

***void CAN\_Tx(uint16\_t slot, uint8\_t\* pData, uint16\_t uSec)***

The parameter <*slot*> represent the index on the <*message*> vector corresponding to the setting of the message we want to transmit.

<*pData*> is a pointer to an 8 bit buffer that needs to remain valid for the time the transmit function is executed.

<*uSec*> is a timeout in [us] in which the function has to be executed.