PIC32MX PIO Library Documentation

Revision 1.0

Luka Gacnik

November 2022

Table of contents

Table of figures

Acronyms

PIO	Programmable Input Output
I/O	Input/Output
Z	Impedance
CPU	Central Processing Unit
MCU	Microcontroller Unit
CLR	Clear
SET	Set
INV	Invert
CN	Change Notice
PPS	Peripheral Pin Select
GPIO	General Purpose Input Output
ISR	Interrupt Service Routine
SFR	Special Function Register

1 Library description

The library consists of basic functions that control any Programmable Input/Output module of an PIC32MX series of the Microchip microcontrollers.

1.1 About the Programmable Input Output

A Programmable Input/Output (PIO) module can consist of many different registers which provide many possibilities for interfacing an external analog or digital device. Such registers usually provide basic abilities like:

- Generating high or low state at a specific pin configured as a digital output
- Reading state of a specific pin when configured as a high-Z input
- Modifying pin's data direction (I/O), type (digital/analog), driver (push-pull/open-drain), pull type (up, down, up-down), slew rate (fast, slow)
- Assigning one pin to either of multiple other peripheral modules
- Generating an interrupt on a specific event (e.g. negative/positive edge)

In a microcontroller a PIO functionality could also be referred to as the Parallel Input/Output since PIO registers for reading/writing data from/to pins are accessed in parallel manner (e.g. loading 32-bit data from CPU to a latch register would generate a user-defined state at all 32 pins in 2-3 clock cycles).

The library is based on the PIC32MX MCU series. Some of the key features of its PIO modules are:

- Individual output pin open-drain enable/disable
- Individual input pin pull-up enable/disable

- Monitor select inputs and generate interrupt on a mismatch condition
- Operate during Sleep and Idle modes
- Fast bit manipulation using CLR, SET, and INV registers

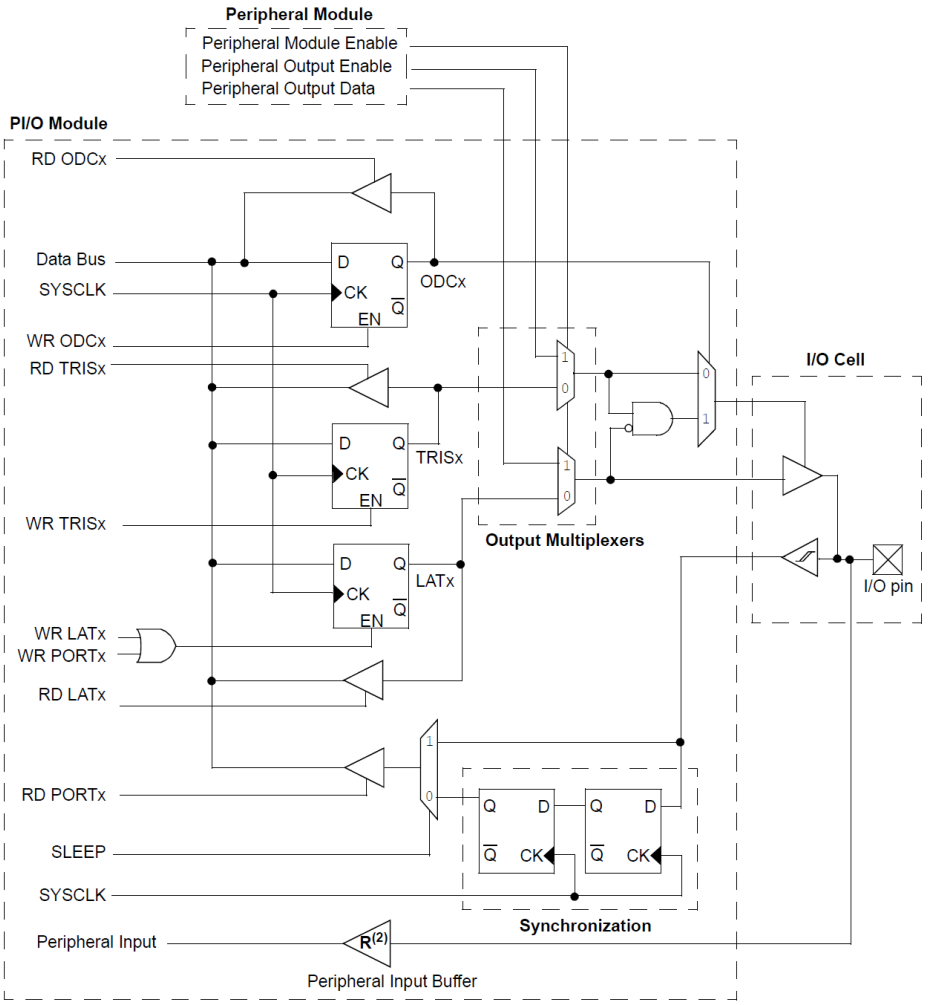

Figure 1.1: PIC32MX typical PIO module block diagram

In addition to basic functions PIC32MX devices come with Input Change Notification (CN) registers which can generate interrupt requests in response to input change-of-state. And a Peripheral Pin Select (PPS) registers that provide control bits assigning pins to available peripheral modules.

1.2 Library features

Currently, the PIO driver is capable of performing the following operations:

- Configuring a GPIO pin with a selectable data pin direction, pin type, driver type, and pull type
- Configuring a peripheral pin via PPS registers
- Configuring CN registers for external event triggered ISR
- Reading input pin state and generating output pin state

2 Dependencies

The library consists of the following source and header files:

- `Pio.c`
- `Pio.h`
- `Pio_sfr.h`

All the external user-accessible function prototypes are provided in the file `Pio.h`. Their definitions and other local functions like ISR handlers are contained in the `Pio.c` file. The file `Pio_sfr.h` contains all the macro definitions for register bit-fields and SFR memory layout type.

The library depends on the following other libraries:

- `Cfg.h`
- `Ic.h`

The library `Cfg.h` provides the functions for unlocking a specific set of registers required by the PPS registers. Interrupts are configured using the `Ic.h` library.

Additionally, there needs to be one of the standard Microchip libraries included (since the compiler XC32 is the made by the Microchip), that is the `sys\attrib.h` library. It is used for its ISR macro definition that is needed for ISR function attribute definition.

Lastly, its important to provide the fact that this library was meant to be used with a specific MCU - the PIC32MX170Bxxx series made by the Microchip. As previously mentioned the XC32 compiler was used to compile the library files, meaning there are some specifics in the library which are unique to the XC32 compiler (such as ISR

function attribute). In case any other MCU of the PIC32 family would be used, some peripheral libraries (used by this driver) might need modifications (like different register sets, etc.).

3 Notes and warnings

Some code related facts must be known prior to using the PIO library;

a) **Single interrupt CN vector**

The Input Change (CN) register set provides an option for user-defined functions to be executed at specific external event. Each PIO module has its CN set of SFRs, however all share the same interrupt vector. This means that a single vector priorty can be used for all CN register sets (two in case of PIC32MX device). However when a user-defined function is provided for CN ISR handler, one such function will correspond to all pins of the same CN set of registers. The PIC32MX device has only one CN interrupt vector but it has two separate interrupt flags, each for one CN register set.

b) **Interrupt priority of an Input Change**

Prior to using of the `PIO_ConfigInputChange()` function the (sub)priority level of all the PIO modules interrupt vector must be defined. This is done via providing a constant value using macros `CN_ISR_IPL`, `CN_ICX_IPL`, and `CN_ICX_ISL`.

4 Future ideas

None.

5 Custom types

The library comes with a set of custom made structure or enumeration types. For the sake of convenience these types are explained throughout this chapter. Note that individual enumeration types are not covered here because they (and their set of values) follow self-explanatory naming conventions.

5.1 PinInfo_t

This structure-based type contains the properties for storing information about any of provided pin codes which are defined in the `Pio_sfr.h` file. The type consists of the following properties:

- **pinMod** (*PioSfr_t* ∗) - a pointer to a PIO module base address of a certain pin
- **pinDir** (*PioPinDirect_t*) - a data direction of a certain pin
- **pinPos** (*PioPinPos_t*) - a position of a certain pin

This type or types of individual members are used are used as return types of PIO related functions which decode usable information out of given pin code. In context of PIO library a pin code is a 32-bit constant macro that stores information about basic pin's properties. The following list explains meaning of individual codes:

- Bits 0-7: **PPS register offset** - a register offset for the PPS set of registers relative to the base register
- Bits 8-15: **PPS peripheral code** - a code used written to a specific PPS register which determines a peripheral that controls this pin
- Bits 16-23: **pin position** - a position of this pin

- Bits 24-27: **pin direction** - a data direction of this pin
- Bits 28-31: **PIO module** - an offset for a PIO module of this pin relative to the first PIO module

There are two types of pins - GPIO and peripheral-controlled pins.

Any peripheral-controlled pin has all bit field (provided by upper list) defined. Such a pin is controlled by a certain peripheral module (e.g. SPI pins MOSI, MISO, SCK and SS). To properly configure them as operational pins one needs to write a specific PPS code into a specific PPS register. This is defined by first two bullets in the upper list. Additionally, peripheral modules do not have control over tri-state registers of a specific PIO module which makes additional requirement for a data direction to be manually configured for a peripheral-controlled pin. The position of a pin is needed to modify specific bit-field of PIO SFRs. Lastly, offset of PIO module base address is used to identify the pin's PIO module.

The GPIO pin codes have first 16 bits undefined or rather set to `0xFFFF` which identifies them as GPIO pins. These aren't controlled by any peripheral therefore PPS register set aren't modified but only PIO registers are written to. Since its not known what might be the data direction of such a pin, the bits 24-27 are unused (set to `0xF`). User must manually provide data direction of a pin. Therefore, only the pin position and module codes are used to identify basic properties of any GPIO pin on a PIC32MX device.

The discussed pin codes are by default used by this library's functions unless stated otherwise. If a modification of a certain pin's properties is desired to be done not by using pin codes then the user must resort to directly modifying PIO and PPS SFRs.

5.2 PioSfr_t

This structure-based type contains the properties for modification of PIO related SFRs. Throughout this library it is mostly used as a pointer to a base address of any PIO module. MCU memory contents starting at such a specific base address are modified when changing the value of any member that is contained by this structured type. The

way of each member is positioned in the structure is strictly defined and shall not be modified by the user. Member layout defines the exact position of PIO SFRs, relative to any of PIO base address.

Additionally, the type *PpsSfr_t* is provided by the file `Pio_sfr.h` intended for accessing PPS SFRs. However, this type isn't used with any of externally seen functions but rather throughout the PIO library in the `Pio.c` file.

6 Function description

The library includes the following functions:

- `PIO_ConfigPpsSfr()`
- `PIO_ReleasePpsSfr()`
- `PIO_ConfigInputChange()`
- `PIO_SetIsrHandler()`
- `PIO_ConfigPpsPin()`
- `PIO_ConfigGpioPin()`
- `PIO_ConfigGpioPinDir()`
- `PIO_ConfigGpioPinType()`
- `PIO_ConfigGpioPinPull()`
- `PIO_ConfigGpioPinDriver()`
- `PIO_ReadPinCode()`
- `PIO_ReadPinModule()`
- `PIO_ReadPinDirection()`
- `PIO_ReadPinPosition()`
- `PIO_ClearPin()`
- `PIO_SetPin()`
- `PIO_TogglePin()`
- `PIO_ReadPin()`

The following sub-sections describe the use and operationality of these functions.

6.1 PIO_ConfigPpsSfr()

PROTOTYPE

`PIO_ConfigPpsSfr(pinCode)`

DESCRIPTION

Configures a pin, specified by a pin code, as a peripheral-controlled pin.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

Returns `false` if the passed pin code is recognized as a GPIO pin, otherwise it returns `true`.

OPERATION

The PPS register offset and register peripheral-related code are decoded from a pin code. Then all PPS registers are unlocked for writing with the function `CFG_UnlockPpsAccess()`, PPS register is modified, and registers locked using the `CFG_LockPpsAccess()` function.

NOTES

Interrupts are temporarily disabled.

LIMITATIONS

None.

6.2 PIO_ReleasePpsSfr()

PROTOTYPE

`PIO_ReleasePpsSfr(pinCode)`

DESCRIPTION

Releases PPS control over an output peripheral-controlled pin.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

Returns `false` if the passed pin code is recognized as a GPIO pin or pin is an input, otherwise it returns `true`.

OPERATION

The PPS register offset peripheral-related code is decoded from a pin code. Then PPS registers are unlocked for writing with the function `CFG_UnlockPpsAccess()`, the code `0x00` is written to a specific PPS register, and registers locked using the `CFG_LockPpsAccess()` function.

NOTES

Interrupts are temporarily disabled.

LIMITATIONS

None.

6.3 PIO_ConfigInputChange()

PROTOTYPE

```
PIO_ConfigInputChange(pinCode, pullType)
```

DESCRIPTION

Configures Change Notice (CN) SFRs for the corresponding pin. External state change on a specific pin will trigger an ISR.

PARAMETERS

- **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin
- **`pullType`** (*`PioPullType_t`*) - an internal resistor pull type relative to a configured pin

RETURNS

Returns `false` if the passed pin code has false module offset code, otherwise it returns `true`.

OPERATION

The CN and PIO SFRs are modified. Then appropriate interrupts SFRs have their priority and source enable registers configured.

NOTES

Interrupts are temporarily disabled.

LIMITATIONS

None.

6.4 PIO_SetIsrHandler()

PROTOTYPE

```
PIO_SetIsrHandler(pinCode, isrHandler)
```

DESCRIPTION

Configures Change Notice (CN) SFRs for the corresponding pin. External state change on a specific pin will trigger an ISR.

PARAMETERS

- **pinCode** (*uint32_t*) - a pin code that contains basic properties of a certain pin
- **isrHandler** (*void (*f) (void)*) - a pointer to the user-defined function

RETURNS

Returns `false` if the passed pin code has false module offset code, otherwise it returns `true`.

OPERATION

Decodes a PIO module base address of the pin code. Based on it either of two ISR handler pointers (two handlers for two PIO modules) is set.

NOTES

One user-defined function may be provided which will correspond to any pin of the same PIO module.

LIMITATIONS

None.

6.5 PIO_ConfigPpsPin()

PROTOTYPE

`PIO_ConfigPpsPin(pinCode, pinType)`

DESCRIPTION

Configures a peripheral-controlled pin as either analog or digital pin.

PARAMETERS

- **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin
- **`pinType`** (*`PioPinType_t`*) - a type (digital/analog) of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Decodes a PIO module base address of the peripheral-controlled pin code. Then it modifies tri-state (`TRIS`) and analog select (`ANSEL`) registers based on pin type, data direction and pin position.

NOTES

- This function shouldn't be confused with the `PIO_ConfigPpsSfr()` which configures PPS registers of a certain pin while the function `PIO_ConfigPpsPin()` configures PIO registers for a certain peripheral-controlled pin
- This function is always inlined since function is short and frequent usage is expected

LIMITATIONS

None.

6.6 PIO_ConfigGpioPin()

PROTOTYPE

`PIO_ConfigGpioPin(pinCode, pinType, pinDir)`

DESCRIPTION

Configures a GPIO pin with a selected type and data direction.

PARAMETERS

- **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin
- **`pinType`** (*`PioPinType_t`*) - a type (digital/analog) of a certain pin
- **`pinDir`** (*`PioPinDirect_t`*) - a data direction of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Decodes a PIO module base address of the GPIO pin code. Then it modifies tri-state (`TRIS`) and analog select (`ANSEL`) registers based on pin type, data direction and pin position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.7 PIO_ConfigGpioPinDir()

PROTOTYPE

```
PIO_ConfigGpioPinDir(pinCode, pinDir)
```

DESCRIPTION

Configures a GPIO pin with a selected data direction.

PARAMETERS

- **pinCode** (*uint32_t*) - a pin code that contains basic properties of a certain pin
- **pinDir** (*PioPinDirect_t*) - a data direction of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Decodes a PIO module base address of the GPIO pin code. Then it modifies tri-state (TRIS) register based on data direction and pin position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.8 PIO_ConfigGpioPinType()

PROTOTYPE

```
PIO_ConfigGpioPinType(pinCode, pinType)
```

DESCRIPTION

Configures a GPIO pin with a selected type.

PARAMETERS

- **pinCode** (*uint32_t*) - a pin code that contains basic properties of a certain pin
- **pinType** (*PioPinType_t*) - a type (digital/analog) of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Decodes a PIO module base address of the GPIO pin code. Then it modifies an analog select (ANSEL) registers based on pin type and pin position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.9 PIO_ConfigGpioPinPull()

PROTOTYPE

`PIO_ConfigGpioPinPull(pinCode, pullType)`

DESCRIPTION

Configures a GPIO input pin with an internal pull resistor.

PARAMETERS

- **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin
- **`pullType`** (*`PioPullType_t`*) - a resistor pull type of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Decodes a PIO module base address of the GPIO pin code and pin position. If pin is input then appropriate pull-up (`CNPU`) and/or pull-down (`CNPD`) register is modified, otherwise function is terminated.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.10 PIO_ConfigPinDriver()

PROTOTYPE

```
PIO_ConfigPinDriver(pinCode, pinDriver)
```

DESCRIPTION

Configures a GPIO or peripheral-controlled pin's type of driver.

PARAMETERS

- **pinCode** (*uint32_t*) - a pin code that contains basic properties of a certain pin
- **pinDriver** (*PioPinDriver_t*) - a driver type of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Decodes a PIO module base address of the pin code. Appropriate driver register (ODC) is modified based on pin position and driver type.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.11 PIO_ReadPinCode()

PROTOTYPE

`PIO_ReadPinCode(pinCode)`

DESCRIPTION

Decodes information about pin module, data direction, and position.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

Returns structure of the type *`PinInfo_t`* which stores information about pin module, data direction, and position.

OPERATION

Directly writes pin information to the structure it later returns.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.12 PIO_ReadPinModule()

PROTOTYPE

`PIO_ReadPinModule(pinCode)`

DESCRIPTION

Decodes PIO module base address of a given pin code.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

Returns structure of the type *`PioSfr_t`* $*$ which stores information about pin module.

OPERATION

Direct return of a pin PIO module base address.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.13 PIO_ReadPinDirection()

PROTOTYPE

`PIO_ReadPinDirection(pinCode)`

DESCRIPTION

Decodes data direction of a given pin code.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

Returns structure of the type *`PioPinDirect_t`* which stores information about data direction of a pin.

OPERATION

Direct return of a pin data direction.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.14 PIO_ReadPinPosition()

PROTOTYPE

`PIO_ReadPinPosition(pinCode)`

DESCRIPTION

Decodes pin position of a given pin code.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

Returns structure of the type *`PioPinPos_t`* which stores information about pin position of a pin.

OPERATION

Direct return of a pin position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.15 PIO_ClearPin()

PROTOTYPE

`PIO_ClearPin(pinCode)`

DESCRIPTION

Clears output pin state for a given pin code.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Direct modification of a latch register (`LAT`) based on pin module and position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.16 PIO_SetPin()

PROTOTYPE

`PIO_SetPin(pinCode)`

DESCRIPTION

Sets output pin state for a given pin code.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Direct modification of a latch register (`LAT`) based on pin module and position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.17 PIO_TogglePin()

PROTOTYPE

`PIO_TogglePin(pinCode)`

DESCRIPTION

Toggles output pin state for a given pin code

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

No returns since this function is speed-optimized.

OPERATION

Direct modification of a latch register (`LAT`) based on pin module and position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

6.18 PIO_ReadPin()

PROTOTYPE

`PIO_ReadPin(pinCode)`

DESCRIPTION

Read input pin state for a given pin code.

PARAMETERS

• **`pinCode`** (*`uint32_t`*) - a pin code that contains basic properties of a certain pin

RETURNS

Returns an unsigned value (1 - high, 0 - low) of type *`uint32_t`*.

OPERATION

Direct read of port register (`PORT`) based on pin module and position.

NOTES

This function is always inlined since function is short and frequent usage is expected.

LIMITATIONS

None.

7 Examples

An examples is provided to demonstrate a simple operation of a PIO module. Examples provide only the main operation related code. Other processes such as the start-up of an MCU, programming of the device configuration register, setting up other peripheral modules, and similar is not provided here

7.1 Input Change example

This example provides the code for setting up a PIO module for the Input Change functionality by using Change Notice (CN) register set. The user-defined function `TestFunct()` is executed when the generated signal changes its state. For demonstration purpose a counter variable is used within the ISR handler. The pin labeled as `GPIO_RPB4` is configured to detect any changes on external signal which is provided by the pin labeled as `GPIO_RPA2`.

```
1  /** Custom libs **/
2  #include "Pio.h"
3
4  /** Test variable **/
5  static volatile uint32_t inputChangeCounter = 0;
6
7  /** Test prototype **/
8  static void TestFunct(void);
9
10
11 int main(int argc, char** argv)
12 {
13    /* Configure signal generator */
14    PIO_ConfigGpioPin(GPIO_RPA2, PIO_TYPE_DIGITAL, PIO_DIR_OUTPUT);
15
16    /* Set idle state of input signal */
17    PIO_ClearPin(GPIO_RPA2);
18
```

```
19      /* Configure input change feature of PIO module */
20      PIO_ConfigInputChange(GPIO_RPB4, PIO_CN_NONE);
21
22      /* Configure a function pointer */
23      PIO_SetIsrHandler(GPIO_RPB4, TestFunct);
24
25      /* Simulate input signal */
26      PIO_SetPin(GPIO_RPA2);
27      PIO_ClearPin(GPIO_RPA2);
28      PIO_SetPin(GPIO_RPA2);
29      PIO_ClearPin(GPIO_RPA2);
30
31      return 0;
32  }
33
34
35  /** Test function **/
36  static void TestFunct(void)
37  {
38      inputChangeCounter++;
39  }
```

A Revision history

Revision X.X (November 2022)

This is the initial released version of this document.