



## STR71x firmware library

---

### Introduction

#### About this manual

This document is the STR71x firmware library user manual. It describes the STR71x peripheral firmware library: a collection of routines, data structures and macros that cover the features of each peripheral.

The firmware library user manual is structured as follows:

- Definitions, document conventions and firmware library rules
- Overview of the firmware library (package content, library structure), installation guidelines, and examples of how to use the library.
- Detailed description the firmware library: configuration structure and function descriptions for each peripheral in detail.

#### About STR71x firmware library

The STR71x firmware library is a firmware package consisting of device drivers for all standard STR71x peripherals. You can use any STR71x device in applications without in-depth study of each peripheral specification. As a result, using this library can save you a lot of the time that you would otherwise spend in coding and also the cost of developing and integrating your application.

Each device driver consists of a set of functions covering the functionality of the peripheral. Since all the STR71x peripherals and their corresponding registers are memory-mapped, a peripheral can be easily controlled using 'C' code. The source code, developed in 'C', is fully documented. A basic knowledge of 'C' programming is required.

The library contains a complete firmware in 'C' and it is independent from any software tool-chains. Only the start-up files are tool-chain dependent.

Since the library is generic and covers all of the capabilities of each peripheral, it may not be as efficient as it could be from a code size and/or execution speed point of view. For many applications the library may be used as is. However, for applications with strong constraints in terms of code size and/or execution speed, the library drivers should be used as a reference, to show you how to configure the peripheral and which you can then tailor to your specific application requirements.

# Contents

- 1 Document and library rules ..... 6**
  - 1.1 Abbreviations ..... 6
  - 1.2 Styles and symbols ..... 6
  - 1.3 Naming conventions ..... 7
  - 1.4 C coding rules ..... 7
  
- 2 Firmware library ..... 11**
  - 2.1 Package description ..... 11
    - 2.1.1 Examples ..... 11
    - 2.1.2 Library ..... 12
    - 2.1.3 Project ..... 12
  - 2.2 File description ..... 12
  - 2.3 How to use the STR71x library ..... 14
  
- 3 Peripheral firmware overview ..... 17**
  - 3.1 Power control unit (PCU) ..... 17
    - 3.1.1 Data structures ..... 17
    - 3.1.2 Firmware library functions ..... 20
  - 3.2 Reset and clock control unit (RCCU) ..... 29
    - 3.2.1 Data structures ..... 29
    - 3.2.2 Firmware library functions ..... 36
  - 3.3 Advanced peripheral bus bridges (APB) ..... 47
    - 3.3.1 Data structures ..... 47
    - 3.3.2 Common parameter values ..... 50
    - 3.3.3 Firmware library functions ..... 50
  - 3.4 Enhanced interrupt controller (EIC) ..... 52
    - 3.4.1 Data structures ..... 53
    - 3.4.2 Firmware library functions ..... 56
  - 3.5 General purpose input output (GPIO) ..... 65
    - 3.5.1 Data structures ..... 65
    - 3.5.2 Common parameter values ..... 68
    - 3.5.3 Firmware library functions ..... 69
  - 3.6 External Interrupts (XTI) ..... 75

---

|          |  |            |
|----------|--|------------|
| 3.6.1    | Data structures                                    | 75         |
| 3.6.2    | Common parameter values                            | 77         |
| 3.6.3    | Firmware library functions                         | 78         |
| 3.7      | Real time clock (RTC)                              | 84         |
| 3.7.1    | Data structures                                    | 84         |
| 3.7.2    | Firmware library functions                         | 86         |
| 3.8      | Watchdog timer (WDG)                               | 94         |
| 3.8.1    | Data structures                                    | 94         |
| 3.8.2    | Firmware library functions                         | 96         |
| 3.9      | Timer (TIM)  | 101        |
| 3.9.1    | Data structures                                    | 101        |
| 3.9.2    | Firmware library functions                         | 107        |
| 3.10     | Buffered serial peripheral interface (BSPI)        | 120        |
| 3.10.1   | Data structures                                    | 121        |
| 3.10.2   | Firmware library functions                         | 124        |
| 3.11     | Universal asynchronous receiver transmitter (UART) | 140        |
| 3.11.1   | Data structures                                    | 140        |
| 3.11.2   | Firmware library functions                         | 145        |
| 3.12     | Inter-integrated circuit (I2C)                     | 165        |
| 3.12.1   | Data structures                                    | 165        |
| 3.12.2   | Firmware library functions                         | 170        |
| 3.13     | Controller area network (CAN)                      | 184        |
| 3.13.1   | Data structures                                    | 184        |
| 3.13.2   | Firmware library functions                         | 192        |
| 3.14     | 12-bit analog-to-digital converter (ADC12)         | 213        |
| 3.14.1   | Data structures                                    | 213        |
| 3.14.2   | Firmware library functions                         | 216        |
| 3.15     | External memory interface (EMI)                    | 222        |
| 3.15.1   | Data structures                                    | 222        |
| 3.15.2   | Firmware library functions                         | 224        |
| <b>4</b> | <b>Revision history</b>                            | <b>226</b> |

## List of tables

|           |  |     |
|-----------|--|-----|
| Table 1.  | List of acronyms . . . . .                   | 6   |
| Table 2.  | Source/header file list . . . . .            | 12  |
| Table 3.  | Peripheral firmware functions . . . . .      | 17  |
| Table 4.  | PCU structure fields . . . . .               | 18  |
| Table 5.  | PCU voltage regulator status . . . . .       | 19  |
| Table 6.  | PCU voltage regulator descriptions . . . . . | 19  |
| Table 7.  | WFI clocks . . . . .                         | 19  |
| Table 8.  | PCU flags . . . . .                          | 20  |
| Table 9.  | PCU library functions . . . . .              | 20  |
| Table 10. | RCCU registers. . . . .                      | 30  |
| Table 11. | Clock divider parameters . . . . .           | 31  |
| Table 12. | RCLK clock sources . . . . .                 | 31  |
| Table 13. | PLL1 multiplication factors . . . . .        | 32  |
| Table 14. | PLL2 multiplication factors . . . . .        | 32  |
| Table 15. | PLL division factors. . . . .                | 33  |
| Table 16. | USB clock sources . . . . .                  | 33  |
| Table 17. | RCCU internal clocks . . . . .               | 34  |
| Table 18. | RCCU interrupts . . . . .                    | 34  |
| Table 19. | RCCU flags. . . . .                          | 35  |
| Table 20. | RCCU reset sources. . . . .                  | 35  |
| Table 21. | PLL1 free running modes . . . . .            | 36  |
| Table 22. | APB registers . . . . .                      | 47  |
| Table 23. | Bridge peripheral codes . . . . .            | 49  |
| Table 24. | APBx values . . . . .                        | 50  |
| Table 25. | APB library functions . . . . .              | 50  |
| Table 26. | EIC peripheral registers . . . . .           | 53  |
| Table 27. | IRQChannel values. . . . .                   | 55  |
| Table 28. | FIQChannel values . . . . .                  | 56  |
| Table 29. | GPIO registers . . . . .                     | 66  |
| Table 30. | GPIO pin modes . . . . .                     | 67  |
| Table 31. | Port_Pins parameter value . . . . .          | 68  |
| Table 32. | GPIOx values . . . . .                       | 68  |
| Table 33. | GPIO library functions. . . . .              | 69  |
| Table 34. | XTI registers . . . . .                      | 76  |
| Table 35. | XTI modes . . . . .                          | 77  |
| Table 36. | Trigger edge polarity values . . . . .       | 77  |
| Table 37. | Lines values . . . . .                       | 77  |
| Table 38. | XTI library functions . . . . .              | 78  |
| Table 39. | RTC structure fields . . . . .               | 84  |
| Table 40. | RTC flags . . . . .                          | 86  |
| Table 41. | RTC interrupts . . . . .                     | 86  |
| Table 42. | RTC library functions . . . . .              | 86  |
| Table 43. | WDG structure fields. . . . .                | 95  |
| Table 44. | WDG library functions. . . . .               | 96  |
| Table 45. | TIM structure fields . . . . .               | 101 |
| Table 46. | TIMx values . . . . .                        | 103 |
| Table 47. | TIM flags . . . . .                          | 103 |
| Table 48. | TIM clock sources. . . . .                   | 104 |

|           |   |     |
|-----------|---|-----|
| Table 49. | TIM clock edges . . . . .                               | 104 |
| Table 50. | TIM channels . . . . .                                  | 104 |
| Table 51. | Output compare modes . . . . .                          | 105 |
| Table 52. | Logic levels . . . . .                                  | 105 |
| Table 53. | Counter operations . . . . .                            | 106 |
| Table 54. | PWM input parameters . . . . .                          | 106 |
| Table 55. | TIM interrupts . . . . .                                | 106 |
| Table 56. | TIM library functions . . . . .                         | 107 |
| Table 57. | BSPI registers . . . . .                                | 121 |
| Table 58. | BSPI flags . . . . .                                    | 123 |
| Table 59. | BSPI interrupts . . . . .                               | 123 |
| Table 60. | BSPI transmit interrupt sources . . . . .               | 124 |
| Table 61. | BSPI receive interrupt sources . . . . .                | 124 |
| Table 62. | BSPI library functions . . . . .                        | 124 |
| Table 63. | UART structure fields . . . . .                         | 140 |
| Table 64. | UART FIFOs . . . . .                                    | 142 |
| Table 65. | UART stop bits . . . . .                                | 143 |
| Table 66. | UART parity modes . . . . .                             | 143 |
| Table 67. | UART modes . . . . .                                    | 144 |
| Table 68. | Interrupt and status flags . . . . .                    | 144 |
| Table 69. | UARTx values . . . . .                                  | 145 |
| Table 70. | UART library functions . . . . .                        | 145 |
| Table 71. | I <sup>2</sup> C registers . . . . .                    | 165 |
| Table 72. | I <sup>2</sup> C addressing modes . . . . .             | 167 |
| Table 73. | I <sup>2</sup> C transfer direction . . . . .           | 167 |
| Table 74. | I <sup>2</sup> C flags . . . . .                        | 168 |
| Table 75. | I <sup>2</sup> C transmission status messages . . . . . | 169 |
| Table 76. | I <sup>2</sup> C reception status messages . . . . .    | 170 |
| Table 77. | I <sup>2</sup> C library functions . . . . .            | 170 |
| Table 78. | CAN structure fields . . . . .                          | 185 |
| Table 79. | CAN message interface structure fields . . . . .        | 186 |
| Table 80. | Standard CAN bitrates . . . . .                         | 187 |
| Table 81. | CAN control register bits . . . . .                     | 188 |
| Table 82. | CAN status register bits . . . . .                      | 188 |
| Table 83. | CAN test register bits . . . . .                        | 189 |
| Table 84. | CAN message interface register bits . . . . .           | 190 |
| Table 85. | CAN wake-up modes . . . . .                             | 191 |
| Table 86. | CAN message structure parameters . . . . .              | 191 |
| Table 87. | CAN message identifier types . . . . .                  | 192 |
| Table 88. | CAN identifier limit values . . . . .                   | 192 |
| Table 89. | CAN library functions . . . . .                         | 192 |
| Table 90. | ADC12 registers . . . . .                               | 214 |
| Table 91. | ADC12 conversion modes . . . . .                        | 215 |
| Table 92. | ADC12 channels . . . . .                                | 215 |
| Table 93. | ADC12 flags . . . . .                                   | 216 |
| Table 94. | ADC12 library functions . . . . .                       | 216 |
| Table 95. | EMI modes . . . . .                                     | 223 |
| Table 96. | EMI library functions . . . . .                         | 224 |
| Table 97. | Document revision history . . . . .                     | 226 |

# 1 Document and library rules

The user manual and the firmware library use the conventions described in the sections below.

## 1.1 Abbreviations

[Table 1](#) describes the different abbreviations used in this document.

**Table 1. List of acronyms**

| Acronym | Peripheral / Unit                           |
|---------|---|
| ADC12   | 12-bit Analog to Digital Converter          |
| APB     | Advanced Peripheral Bus bridges             |
| BSPI    | Buffered Serial Peripheral Interface        |
| CAN     | Controller Area Network                     |
| TIM     | Timer                                       |
| EIC     | Enhanced Interrupt Controller               |
| XTI     | External Interrupts                         |
| EMI     | External Memory Interface                   |
| FLASH   | Embedded Flash                              |
| GPIO    | General Purpose Input Output                |
| I2C     | Inter-Integrated Circuit                    |
| PCU     | Power Control Unit                          |
| RCCU    | Reset & Clock Control Unit                  |
| RTC     | Real Time Clock                             |
| UART    | Universal Asynchronous Receiver Transmitter |
| USB     | Universal Serial Bus                        |
| WDG     | Watchdog Timer                              |

## 1.2 Styles and symbols

The firmware library uses the following style conventions:

- Program listings, program examples, structure definitions and function prototypes are shown in a special typeface. This example is a function prototype shown in *courier* typeface.  

```
int PPP_Config (int * pPointer);
```
- Program portions, variables and function names quoted in a paragraph are in *italic typeface*. For example

*bVariable* is used as local variable in the function *PPP\_Config*.

- A comment in a program listing is shown in *italic typeface*. For example:  

```
int PPP_Config (int * pPointer); /*this is a comment */
```

## 1.3 Naming conventions

The Firmware Library uses the following naming conventions:

- **PPP** is used to reference to any peripheral acronym, e.g. **ADC**. See the section above for more information on peripheral acronyms.
- System and source/header file names must be preceded by '71x\_', e.g. *71x\_conf.h*, except driver source code and header files for **PPP** peripheral, e.g. *71x\_tim.h*. Only lower case letters, digits, underscore and at most one suffix are allowed in filenames.
- Constants used in one file are defined at the head of that file. A constant used in more than one file should be defined in a header file. All constants use upper case characters.
- Registers are considered as constants. Their names are in upper case letters and have in most case the same acronyms as in the STR71x reference manual document.
- Variable names are preceded with a letter which indicates the type/size of the variable (b:8 bits, w: 16 bits, d: 32 bits, p: pointer). The variables are in lower case and each word begins with an upper case, e.g. *bVariable*, *pPointer*.
- Peripheral function names are preceded with the corresponding peripheral acronym in upper case followed by an underscore. The first letter in each word is upper case, e.g. **EIC\_CurrentIRQChannelValue**. Only one underscore is allowed in a function name to separate the peripheral acronym from the rest of the function name.
- Functions declared locally don't have any prefix.
- Functions used to initialize a peripheral are named **PPP\_Init**, e.g. **PCU\_Init**.
- Functions for checking whether the specified PPP flag is set or not end with *status*, e.g. **TIM\_FlagStatus**.
- Functions used to configure a peripheral functionality end with *config*, e.g. **WDG\_PrescalerConfig**. Functions that configure a peripheral interrupt are called **PPP\_ITConfig**.
- Functions performing a specific action have a name that ends with a verb describing that action, e.g. activate, clear, send, write...
- Functions clearing specific peripheral flags are called **PPP\_FlagClear**.

## 1.4 C coding rules

The following rules are used in the Firmware Library.

- 12 generic types are defined for variables whose type and size are fixed, and are typically used for register access. These types are defined in the file *71x\_type.h*:
 

```
typedef unsigned long    u32;
typedef unsigned short  u16;
typedef unsigned char   u8;

typedef signed long     s32;
typedef signed short    s16;
typedef signed char     s8;

typedef volatile unsigned long  vu32;
typedef volatile unsigned short vu16;
typedef volatile unsigned char  vu8;
```

```
typedef volatile signed long   vs32;
typedef volatile signed short vs16;
typedef volatile signed char   vs8;
```

- **bool** type is defined in the file *71x\_type.h* as:

```
typedef enum
{
    FALSE = 0,
    TRUE  = !FALSE
} bool;
```

- **FlagStatus** type is defined in the file *71x\_type.h*. Two values can be assigned to this variable: **SET** or **RESET**.

```
typedef enum
{
    RESET = 0,
    SET   = !RESET
} FlagStatus;
```

- **FunctionalState** type is defined in the file *71x\_type.h*. Two values can be assigned to this variable: **ENABLE** or **DISABLE**.

```
typedef enum
{
    DISABLE = 0,
    ENABLE  = !DISABLE
} FunctionalState;
```

- Pointers to peripherals are used to access the peripheral control registers. Peripheral pointers point to data structures that represent the mapping of the peripheral control registers. A structure should be defined for each peripheral in *71x\_map.h* file. The example below illustrates the **GPIO** register structure declaration:

```
typedef volatile struct
{
    vu16 PC0;
    vu16 EMPTY1;
    vu16 PC1;
    vu16 EMPTY2;
    vu16 PC2;
    vu16 EMPTY3;
    vu16 PD;
} GPIO_TypeDef;
```

Register names are the register acronyms written in upper case for each peripheral. *EMPTY<sub>i</sub>* (*i* is an integer that indexes the reserved field) replaces a reserved field and is never used.

Sometimes an enumeration is needed for a peripheral. This enumeration is called *PPP\_Registers* and is declared in the file *71x\_ppp.h*, e.g.:

```
typedef enum
{
    GPIO_PC0 = 0x00,
    GPIO_PC1 = 0x04,
    GPIO_PC2 = 0x08,
    GPIO_PD  = 0x0C,
} GPIO_Registers;
```



Peripherals are declared in **71x\_map.h** file. This example shows the declaration of the *GPIO* peripheral:

```
#ifndef EXT
    #define EXT extern
#endif
...

/* APB2 Base Address definition */
#define APB2_BASE 0xE0000000

#define GPIO0_BASE      (APB2_BASE + 0x3000)
#define GPIO1_BASE      (APB2_BASE + 0x4000)

/* GPIO0 and GPIO1 peripherals declaration */
#ifndef DEBUG

...
#ifdef _GPIO0
    #define GPIO0                ((GPIO_TypeDef *)GPIO0_BASE)
#endif /*GPIO0*/

#ifdef _GPIO1
    #define GPIO1                ((GPIO_TypeDef *)GPIO1_BASE)
#endif /*GPIO1*/
...
#else /* DEBUG */

#ifdef _GPIO0
    EXT GPIO_TypeDef             *GPIO0;
#endif /*GPIO0*/

#ifdef _GPIO1
    EXT GPIO_TypeDef             *GPIO1;
#endif /*GPIO1*/

...
#endif /* DEBUG */
```

To enter debug mode you have to define the label *DEBUG* in **71X\_conf.h** file. Debug mode allows you to watch the peripheral registers in a debugger window, but it uses up some additional memory space and makes the code less efficient. In both cases *GPIO0* is a pointer to the first address of *GPIO0* port.

*EXT* variable is defined in the file **71x\_lib.c** as follows:

```
#define EXT
```

*DEBUG* mode is initialized as follows in the **71x\_lib.c** file:

```
#define DEBUG
#ifdef DEBUG
    void debug(void)
    {
#ifdef _GPIO0
        GPIO0 = (GPIO_TypeDef *)GPIO0_BASE;
#endif
    }
#endif
```

```

#ifdef _GPIO1
    GPIO1 = (GPIO_TypeDef *)GPIO1_BASE;
#endif

#ifdef _GPIO2
    GPIO2 = (GPIO_TypeDef *)GPIO2_BASE;
#endif

    ...
}
#endif /* DEBUG */
    
```

*Note:* Debug mode increases the code size and reduces the code performance. For this reason, it is recommended to use it only when debugging the application and to remove it from the final application code.

Each peripheral has several dedicated registers which contain different flags. Registers are defined within a dedicated structure for each peripheral. Flag definition is adapted to each peripheral case (defined in **71x\_ppp.h** file). Flags are defined as acronyms written in upper case and prefixed by 'PPP\_' prefix, PPP is the acronym of the corresponding peripheral. The flags of each peripheral are included in an enumeration named *PPP\_Flags*. Some peripherals adopt this naming convention.

Each flag label is defined as a byte divided into two parts:

- The 3 MSB bits are allocated to identify the register to which the flag belongs (it is possible to number registers from 0 to 7).
- The 5 LSB bits are allocated to identify the flag position in the register to which it belongs (it is possible to number flags from 0 to 31).

| Bit 7    | Bit 6 | Bit 5 | Bit 4        | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|----------|-------|-------|--------------|-------|-------|-------|-------|
| Register |       |       | Bit position |       |       |       |       |

The following gives an example of flag definitions:

```

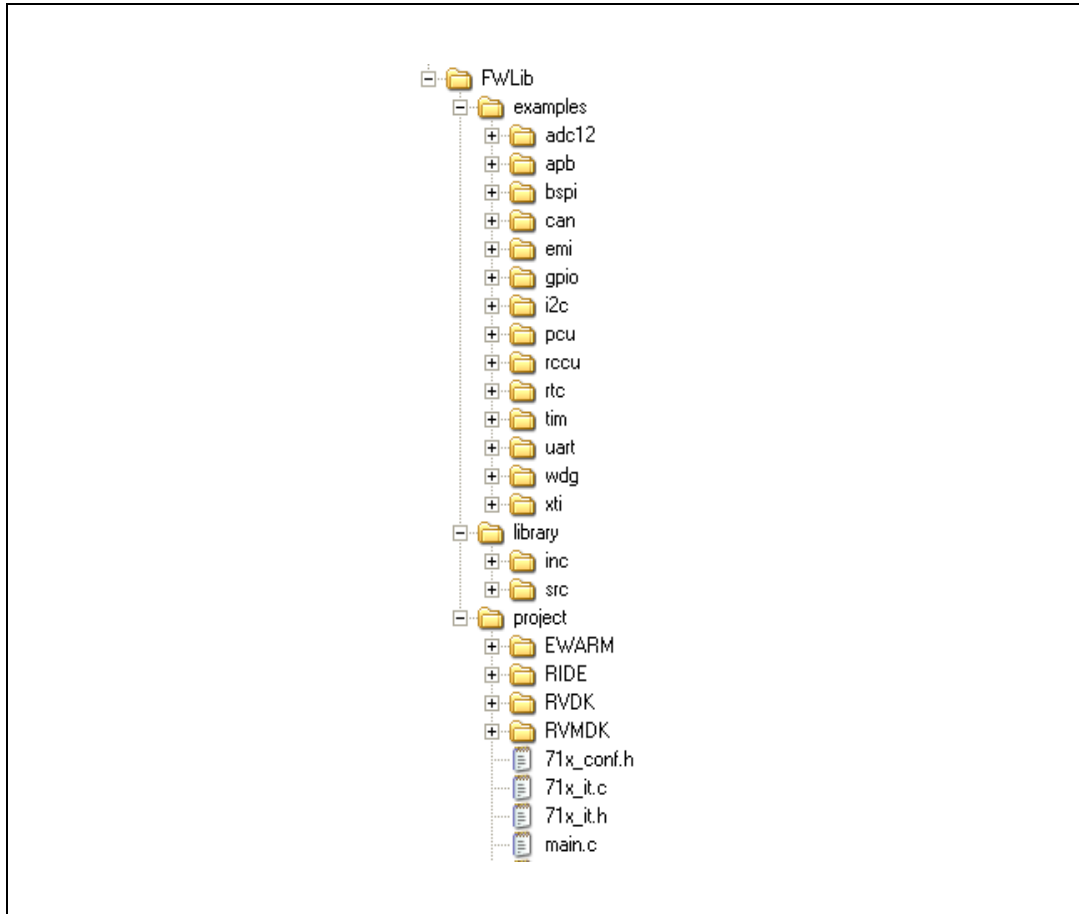
typedef enum
{
    APB_APBT = 0x26,
    APB_OUTM = 0x25,
    APB_ABORT = 0x21,
    APB_ABTEN = 0x49,
    APB_OMnRW = 0x68,
    APB_TOnRW = 0x88
} APB_Flags;
    
```

## 2 Firmware library

### 2.1 Package description

The firmware library is supplied in one single zip package. The extraction of the zip file will give the one folder "**STR71xFWLib\FWLib**" containing the following sub-directories:

**Figure 1. Firmware library directory structure**



#### 2.1.1 Examples

This directory contains for each peripheral sub-directory, the minimum set of files needed to run a typical example on how to use a peripheral:

- **Readme.txt:** a brief text file describing the example and how to make it work
- **71x\_conf.h:** the header file to configure used peripherals and miscellaneous defines
- **71x\_it.c:** the source file containing the interrupt handlers (the function bodies may be empty if not used)
- **71x\_it.h:** header file including all interrupt handler prototypes
- **main.c:** the example program

*Note: All examples are independent from any software toolchain and running on STMicroelectronics STR71x-EVAL board and can be easily tailored to any other hardware.*

### 2.1.2 Library

This directory contains all the subdirectories and files that form the core of the library:

- **inc** sub-directory contains the firmware library header files that do not need to be modified by the user:
  - **71x\_type.h**: Contains the common data types and enumeration used in all other files
  - **71x\_map.h**: Contains the peripherals memory mapping and registers data structures
  - **71x\_lib.h**: Main header file including all other headers
  - **71x\_ppp.h** (one header file per peripheral): contains the function prototypes, data structures and enumeration
- **src** sub-directory contains the firmware library source files that do not need to be modified by the user:
  - **71x\_ppp.c** (one source file per peripheral): contains the function bodies of each peripheral

*Note:* All library files are coded in Strict ANSI-C and are independent from any firmware toolchain.

### 2.1.3 Project

This directory contains a standard template project program that compiles all library files and also all the user modifiable files needed to create a new project:

- **71x\_conf.h**: The configuration header file with all peripherals defined by default
- **71x\_it.c**: The source file containing the interrupt handlers (the function bodies are empty in this template)
- **71x\_it.h**: header file including all interrupt handlers prototypes
- **main.c**: The main program body
- **EWARM, RVDK, RVMDK, RIDE**: For each toolchain usage, refer to Readme.txt file available in the same sub-directory

## 2.2 File description

Several files are used in the firmware library. [Table 2](#) enumerates and describes the different files used in the firmware library.

**Table 2. Source/header file list**

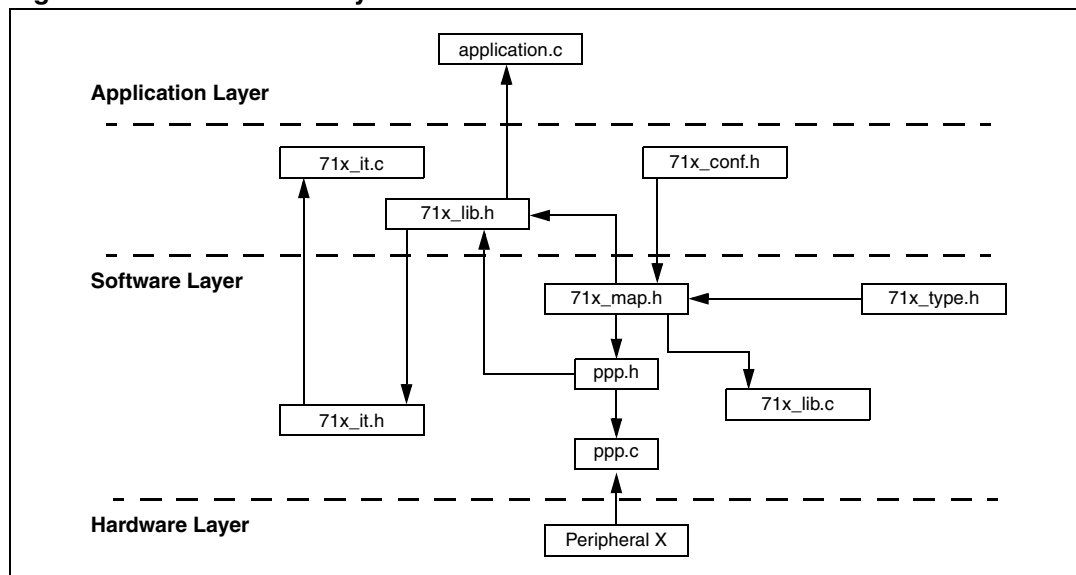
| Filename   | Description  |
|------------|--|
| 71x_conf.h | Parameter configuration file. It should be modified by the user to specify several parameters to interface with the library before running any application.<br>You can enable or disable peripherals if you use the template and you can also change the value of your external crystal oscillator value.<br>Using this file you can select to compile the library in DEBUG or RELEASE mode. |
| main.c     | The main example program body.   |

**Table 2. Source/header file list (continued)**

| Filename   | Description   |
|------------|---|
| 71x_it.c   | Peripheral interrupt functions file. It is modified by you by including the interrupt functions code used in your application. In case of multiple interrupt requests mapped on the same interrupt vector, the function performs a polling on interrupt flags contained inside the peripheral in order to establish the exact source of the interrupt. The names of these functions are already provided in the firmware library. |
| 71x_it.h   | Header file including the definition of interrupt functions.  |
| 71x_lib.c  | Debug mode initialization file. It includes the definition of variable pointers pointing each one to the first address of the dedicated peripheral and the definition of one function called when you choose to enter in debug mode. This function initializes the defined pointers.  |
| 71x_lib.h  | Header file including all the peripherals' header files. It is the unique file to be included in the user application to interface with the library.  |
| 71x_map.h  | This file implements memory mapping and physical registers address definition for both development and debug modes. This file is supplied with all peripherals.   |
| 71x_type.h | Common declarations file. It includes common types and constants which are used by all peripheral drivers.  |
| 71x_ppp.c  | Driver source code file of the PPP peripheral written in C language.  |
| 71x_ppp.h  | Header file of the PPP peripheral. It includes the functions prototypes, data structures and enumerations used by the driver and also available for the user.   |

Figure 2 shows the firmware library architecture and file inclusion relationships.

**Figure 2. Firmware library file structure**



For each peripheral a source code file *71x\_ppp.c* and a header file *71x\_ppp.h* are provided.

- the *71x\_ppp.c* file contains all the firmware functions required to use the peripheral
- the *71x\_ppp.h* file contains all the functions prototypes, data structures, data types and enumeration needed. It contains also the inline functions code, if any.

One memory mapping file *71x\_map.h* is supplied for all peripherals. This file contains all the register declarations for both development and debug modes.

The header file *71x\_lib.h* includes all the peripheral header files. This is the only file that has to be included in the user application to interface with the library.

You can modify the *71x\_conf.h* to define which peripherals need to be included in the application to make it run.

Other files are also provided: *71x\_it.h* and *71x\_it.c*. These files contain all the firmware functions required to handle the STR71x interrupts.

If you want to handle an interrupt routine, you should modify the appropriate function in the file *71x\_it.c* (the body of each function is empty by default).

*Note:* Only the *71x\_conf.h* and *71x\_it.c* files should be modified to link the application with the library.

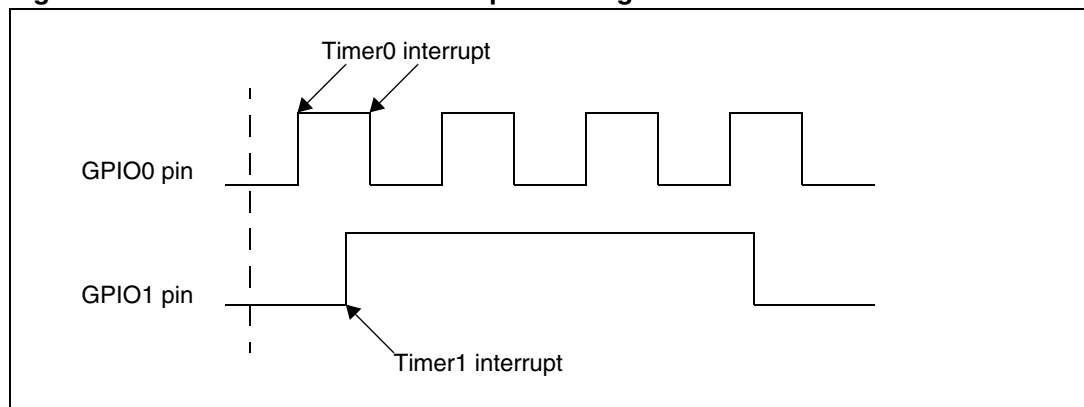
## 2.3 How to use the STR71x library

This section describes how to implement an application using the STR71x firmware library.

Example: The application initializes and starts two timer overflow interrupts. When a timer expires, an interrupt, handled in *71x\_it.c*, is generated. Timer 0 generates an overflow interrupt every 130 ms and toggles the *GPIO0* pin level on/off. Timer 1 is configured to generate an interrupt every 2.1s and to set *GPIO1* pin to high level for 1.05 s. The chip is clocked by an external oscillator at a frequency of 16 MHz.

Timer 0 interrupt priority is set to be higher than Timer 1 interrupt priority to allow the Timer 0 *T0TIM1\_IRQHandler* interrupt routine to be served during the execution of the Timer 1 interrupt routine. The following figure shows the behavior of the *GPIO0* and *GPIO1* pins and Timer 0 and Timer 1 interrupts.

**Figure 3. Timer0 and Timer1 interrupt handling**



The following steps explain how to use the STR71x library to build the user application

- Edit the *71x\_conf.h* file and uncomment the `_GPIO` and `_TIM` labels, this allows you to include the firmware libraries of the *GPIO* and the *TIM* peripherals in your firmware. Uncomment the `_GPIO0`, `_GPIO1`, `_TIM0` and `_TIM1` labels to be able to access the peripheral registers. Uncomment the `DEBUG` label, this allows you to see the peripheral registers content in the watch window.

```

#ifndef __71x_CONF_H
#define __71x_CONF_H
#define DEBUG
...
#define _TIM /*to include the TIM library*/
#define _TIM0 /*to access the TIM0 registers*/
#define _TIM1 /*to access the TIM1 registers*/
...
#define _GPIO /*to include the GPIO library*/
#define _GPIO0 /*to access the GPIO0 registers*/
#define _GPIO1 /*to access the GPIO1 registers*/
...
#endif /* __71x_CONF_H */

```

- Edit the file *71x\_it.c* and add the following code to the Timer 0 interrupt handler routine *T0TIMI\_IRQHandler*:

```

void T0TIMI_IRQHandler (void)
{
    TIM_FlagClear(TIM0, TIM_TOF); /* to clear the TIM0 interrupt
    flag */
    GPIO_WordWrite(GPIO0, ~GPIO_WordRead(GPIO0)); /* invert the
    GPIO0 output level */
}

```

- Add the following code to the Timer 1 interrupt handler routine *T1TIMI\_IRQHandler*:

```

void T1TIMI_IRQHandler (void)
{
    u32 i;
    TIM_FlagClear(TIM1, TIM_TOF); /*to clear the TIM1 interrupt
    flag */
    GPIO_WordWrite(GPIO1, ~GPIO_WordRead(GPIO1));
    for (i=0; i<0x000FFFFFF; i++); /* delay */
    GPIO_WordWrite(GPIO1, ~GPIO_WordRead(GPIO1));
}

```

- The initialization of the *GPIO0*, *GPIO1*, *TIM0* and *TIM1* peripherals is done in the application source file, *main.c* in our case. The configuration of the *EIC* and the *IRQ* channels priority is also done in this file.
- Edit the *main.c* file and add the following code:

```

#include "71x_lib.h"
int main(void)
{

    #ifdef DEBUG
        debug();
    #endif

    /*EIC configuration*/
    /*Set Timer 0 interrupt channel priority level to 2*/
    EIC_IRQChannelPriorityConfig(T0TIMI_IRQChannel,2);
    /*Enable Timer 0 IRQ channel interrupts*/
    EIC_IRQChannelConfig(T0TIMI_IRQChannel,ENABLE);

    /* Set Timer 1 interrupt channel priority level to 1*/

```

```
EIC_IRQChannelPriorityConfig(T1TIMI_IRQChannel,1);
/* Enable Timer 1 IRQ channel interrupts*/
EIC_IRQChannelConfig(T1TIMI_IRQChannel,ENABLE);

/*configure GPIO0 pins as push-pull output mode*/
GPIO_Config(GPIO0, 0xFFFF, GPIO_OUT_PP);
/* Set GPIO0 pins to low level*/
GPIO_WordWrite(GPIO0, 0x0000);

/* configure GPIO1 pins as push-pull output mode*/
GPIO_Config(GPIO1, 0xFFFF, GPIO_OUT_PP);
/* Set GPIO1 pins to low level*/
GPIO_WordWrite(GPIO1, 0x0000);

/*Timer 0 Configuration*/
/*Initialize the Timer*/
TIM_Init (TIM0);
/*Configure Timer 0 Prescaler*/
TIM_PrescalerConfig (TIM0,0x0F);
/* Enable Timer 0 Overflow Interrupt*/
TIM_ITConfig(TIM0,TIM_TO_IT,ENABLE);

/* Timer 1 Configuration*/
/* Initialize the Timer*/
TIM_Init (TIM1);
/* Configure Timer 1 Prescaler*/
TIM_PrescalerConfig (TIM1,0xFF);
/* Enable Timer 1 Overflow Interrupt*/
TIM_ITConfig(TIM1,TIM_TO_IT,ENABLE);

/* Enable IRQ interrupts*/
EIC_IRQConfig(ENABLE);
/* Start Timer 0*/
TIM_CounterConfig (TIM0,TIM_START);
/* Start Timer 1*/
TIM_CounterConfig (TIM1,TIM_START);
/*Infinite loop*/
while(1);
}
```



## 3 Peripheral firmware overview

This chapter describes in detail each peripheral firmware library. The related functions are fully documented. An example of use of the function is given and some important considerations to take into account to avoid failure are also provided.

This chapter describes each peripheral firmware library in detail. The related functions are fully documented. An example of use of the function is given and some important considerations to take into account to avoid failure are also provided.

[Table 3](#) describes peripheral firmware functions.

**Table 3. Peripheral firmware functions**

| Function name          | Description   |
|------------------------|---|
| Function prototype     | Prototype declaration                               |
| Behavior Description   | Brief explanation of how the functions are executed |
| Input Parameter {x}    | Description of the input parameters                 |
| Output parameter {x}   | Description of the output parameters                |
| Return Value           | Value returned by the function                      |
| Required Preconditions | Conditions required before calling the function     |
| Called Functions       | Other library functions called by the function      |
| See also               | Related functions for reference                     |

### 3.1 Power control unit (PCU)

The PCU driver may be used for a variety of purposes, including power management configuration and low power mode selection.

The first section describes the data structures used in the PCU firmware library. The second section presents the PCU firmware library functions.

#### 3.1.1 Data structures

##### PCU register structure

The PCU register structure *PCU\_TypeDef* is defined in the *71x\_map.h* file as follows:

```
typedef volatile struct
{
    vu16 MDIVR;
    u16  EMPTY1;
    vu16 PDIVR;
    u16  EMPTY2;
    vu16 RSTR;
    u16  EMPTY3;
    vu16 PLL2CR;
    u16  EMPTY4;
    vu16 BOOTCR;
    u16  EMPTY5;
```

```

    vu16 PWRCR;
    u16  EMPTY6;
} PCU_TypeDef;

```

[Table 4](#) describes the *PCU* structure fields:

**Table 4. PCU structure fields**

| Register | Description                       |
|----------|-----------------------------------|
| MDIVR    | CPU frequency division factor     |
| PDIVR    | APB frequency division factor     |
| RSTR     | Peripheral Reset Control Register |
| PLL2CR   | PLL2 Control Register             |
| BOOTCR   | Boot Configuration Register       |
| PWRCR    | Power Control Register            |

The *PCU* peripheral is declared in the same file:

```

...
#define PCU_BASE    0xA0000040
...
#ifndef DEBUG
...
#ifdef _PCU
    #define PCU                ((PCU_TypeDef *) PCU_BASE)
#endif /*PCU*/
...
#else /* DEBUG */
...
#ifdef _PCU
    EXT PCU_TypeDef          *PCU;
#endif /*PCU*/

...
#endif

```

When debug mode is used, *PCU* pointer is initialized in the file *71x\_lib.c*:

```

#ifdef _PCU
    #define PCU                ((PCU_TypeDef *) PCU_BASE)
#endif /*PCU*/

```

In debug mode, *\_PCU* must be defined, in the file *71x\_conf.h*, to access the peripheral registers as follows:

```

#define _PCU

```

Some *PCU* functions use the *RCCU* and the *XTI* registers and functions, *\_RCCU* and *\_XTI* must be defined in *71x\_conf.h* file to make the *RCCU* and the *XTI* functions accessible:

```

#define _RCCU
#define _XTI

```

## Voltage regulator status

The following enumeration defines the states of the voltage regulator. The *VR\_Status* enumeration is defined in the file *71x\_pcu.h*:

```
typedef enum
{
    PCU_STABLE,
    PCU_UNSTABLE
} PCU_VR_Status;
```

[Table 5](#) describes the status of *PCU* voltage regulator.

**Table 5. PCU voltage regulator status**

| Regulator State | Description   |
|-----------------|---|
| PCU_STABLE      | The Main Voltage Regulator is stable and can be used to power the device. |
| PCU_UNSTABLE    | The Main Voltage Regulator is not yet stable                              |

## PCU voltage regulators

The following enumeration defines the power control unit voltage regulators. The *PCU\_VR* enumeration is declared in the file *71x\_pcu.h*:

```
typedef enum
{
    PCU_MVR = 0x0008,
    PCU_LPR = 0x0010
} PCU_VR;
```

[Table 6](#) describes the *PCU* voltage regulators.

**Table 6. PCU voltage regulator descriptions**

| Voltage Regulators | Description                     |
|--------------------|---------------------------------|
| PCU_MVR            | The Main Voltage Regulator      |
| PCU_LPR            | The Low Power Voltage Regulator |

## WFI clocks

The following enumeration defines the *WFI* mode clocks. *WFI\_CLOCKS* enumeration is declared in the file *71x\_pcu.h*:

```
typedef enum
{
    WFI_CLOCK2_16,
    WFI_Ck_AF
} WFI_CLOCKS;
```

[Table 7](#) lists the different *WFI* Clocks.

**Table 7. WFI clocks**

| WFI Clocks    | Description                              |
|---------------|--|
| WFI_CLOCK2_16 | CLOCK2/16 clock is selected during LPWFI |
| WFI_Ck_AF     | The RTC clock is selected during LPWFI   |

### PCU flags

The following enumeration defines the *PCU* flags. *PCU\_Flags* enumeration is defined in the file *71x\_pcu.h*:

```
typedef enum
{
    PCU_WREN = 0x8000,
    PCU_VROK = 0x1000
} PCU_Flags;
```

[Table 8](#) lists the different *PCU* flags.

**Table 8. PCU flags**

| PCU Flags | Description                                   |
|-----------|---|
| PCU_WREN  | Power Control Unit register Write Enable flag |
| PCU_VROK  | Main Voltage Regulator OK flag                |

### 3.1.2 Firmware library functions

[Table 9](#) enumerates the different functions of the *PCU* library.

**Table 9. PCU library functions**

| Function Name                     | Description   |
|-----------------------------------|---|
| <a href="#">PCU_MVRStatus</a>     | Reads and returns the <i>PCU</i> main voltage regulator status    |
| <a href="#">PCU_FlagStatus</a>    | Reads and returns the status of a specified <i>PCU</i> flag       |
| <a href="#">PCU_VRConfig</a>      | Configures (Enables / Disables) the <i>PCU</i> voltage regulators |
| <a href="#">PCU_VRStatus</a>      | Reads and returns the <i>PCU</i> voltage regulator status         |
| <a href="#">PCU_LVDDisable</a>    | Disables the Low Voltage Detector                                 |
| <a href="#">PCU_LVDStatus</a>     | Reads and returns the <i>LVD</i> status                           |
| <a href="#">PCU_LPModesConfig</a> | Configures the different STR71x low power mode options            |
| <a href="#">PCU_WFI</a>           | Used to enter WFI and LPWFI modes                                 |
| <a href="#">PCU_STOP</a>          | Used to enter STOP mode   |
| <a href="#">PCU_STANDBY</a>       | Used to enter STANDBY mode  |
| <a href="#">PCU_FlashBurstCmd</a> | This routine is used to set the FLASH to LP/BURST mode            |
| <a href="#">PCU_32OSCCmd</a>      | This routine is used to enable/disable the 32 kHz oscillator      |

## PCU\_MVRStatus

| Function Name          | PCU_MVRStatus  |
|------------------------|--|
| Function Prototype     | PCU_VR_Status PCU_MVRStatus (void);  |
| Behavior Description   | This routine checks and returns the main voltage regulator stability status.   |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | The stability status of the Main Voltage Regulator. Refer to <a href="#">Voltage regulator status on page 19</a> for more details on the allowed values of this parameter. |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to check the main voltage regulator status:

```

{
...
PCU_VR_Status MainVR_Status;
MainVR_Status = PCU_MVRStatus ();
...
}

```

## PCU\_FlagStatus

| Function Name          | PCU_FlagStatus  |
|------------------------|---|
| Function Prototype     | FlagStatus PCU_FlagStatus (PCU_Flags Xflag);  |
| Behavior Description   | This routine is used to check the status of the specified <i>PCU</i> flag.  |
| Input Parameter        | <i>Xflag</i> : specifies the <i>PCU</i> flag. Refer to <a href="#">PCU flags on page 20</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The specified flag status:<br>– <i>SET</i> : The <i>PCU</i> flag is set.<br>– <i>RESET</i> : The <i>PCU</i> flag is reset.                            |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to check the *PCU* flag status.

```

{
FlagStatus WREN_Status;
FlagStatus VROK_Status;
...
/* WREN_Status holds the WEN flag status */
WREN_Status = PCU_FlagStatus (PCU_WREN);
}

```

```

/* VROK_Status holds the Main VR_OK flag status */
VROK_Status = PCU_FlagStatus (PCU_VROK);
...
}

```

## PCU\_VRConfig

| Function Name          | PCU_VRConfig  |
|------------------------|---|
| Function Prototype     | void PCU_VRConfig(PCU_VR Xvr, FunctionalState NewState);  |
| Behavior Description   | This routine is used to configure (enable / bypass) the <i>PCU</i> voltage regulators.  |
| Input Parameter 1      | <i>Xvr</i> : Specifies the <i>PCU</i> voltage regulator.<br>Refer to <a href="#">PCU voltage regulators on page 19</a> for more details on the allowed values of this parameter.  |
| Input Parameter 2      | <i>NewState</i> : Specifies whatever the corresponding voltage regulator is enabled or bypassed.<br>– <i>ENABLE</i> : Enable the corresponding <i>PCU</i> voltage regulator<br>– <i>DISABLE</i> : Bypass the corresponding <i>PCU</i> voltage regulator |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to bypass the main voltage regulator.

```

{
...
PCU_VRConfig(PCU_MVR, DISABLE);
...
}

```

## PCU\_LVDDisable

| Function Name          | PCU_LVDDisable  |
|------------------------|---|
| Function Prototype     | void PCU_LVDDisable (void);                               |
| Behavior Description   | This routine is used to disable the Low Voltage Detector. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to disable the *LVD* feature.

```
{
...
  PCU_LVDDisable ();
...
}
```

**PCU\_VRStatus**

| Function Name          | PCU_VRStatus   |
|------------------------|--|
| Function Prototype     | FunctionalState PCU_VRStatus (PCU_VR xVR);   |
| Behavior Description   | This routine gets the <i>PCU</i> voltage regulator status.   |
| Input Parameter        | <i>xVR</i> : specifies the <i>PCU</i> voltage regulator.<br>Refer to <a href="#">PCU voltage regulators on page 19</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | The state of the specified voltage regulator.<br>– <i>ENABLE</i> : The voltage regulator is enabled.<br>– <i>DISABLE</i> : The voltage regulator is bypassed.                    |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to check the *PCU* voltage regulator status.

```
{
  FlagStatus MVR_Status;
  FlagStatus LPVR_Status;
...
  MVR_Status = PCU_VRStatus (PCU_MVR);
  LPVR_Status = PCU_VRStatus (PCU_LPR);
...
}
```

**PCU\_LVDStatus**

| Function Name          | PCU_LVDStatus   |
|------------------------|---|
| Function Prototype     | FunctionalState PCU_LVDStatus (void);   |
| Behavior Description   | This routine gets and returns the LVD status.   |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The <i>LVD</i> status.<br>– <i>ENABLE</i> : the <i>LVD</i> is enabled<br>– <i>DISABLE</i> : . |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to check the *LVD* feature status.

```
{  
...  
    FunctionalState LVD_Status;  
    LVD_Status = PCU_LVDStatus ();  
...  
}
```



## PCU\_LPModesConfig

| Function Name          | PCU_LPModesConfig  |
|------------------------|--|
| Function Prototype     | void PCU_LPModesConfig (FunctionalState PLL1_State, FunctionalState MVR_State, FunctionalState FLASH_State, FunctionalState LPWFI_State, LPWFI_Clock_TypeDef LPWFI_Clock);   |
| Behavior Description   | This routine configures the different STR71x low power mode options.   |
| Input Parameter 1      | <i>PLL1_State</i> : the state of PLL1 when CK_AF selected.<br>– <i>ENABLE</i> : PLL1 is not disabled when CK_AF is selected<br>– <i>DISABLE</i> : PLL1 is disabled automatically when CK_AF is selected as system clock  |
| Input Parameter 2      | <i>MVR_State</i> : the state of the Main Voltage Regulator when CK_AF is selected.<br>– <i>ENABLE</i> : MVREG is enabled during LPWFI and STOP modes<br>– <i>DISABLE</i> : MVREG is disabled during LPWFI and STOP modes |
| Input Parameter 3      | <i>FLASH_State</i> : the state of the flash during LPWFI and STOP modes.<br>– <i>ENABLE</i> : Flash Stand-by mode during LPWFI and STOP modes<br>– <i>DISABLE</i> : FLASH in powerdown during LPWFI and STOP modes       |
| Input Parameter 4      | <i>LPWFI_State</i> : the state of low power mode during LPWFI.<br>– <i>ENABLE</i> : Enable Low power mode during WFI mode<br>– <i>DISABLE</i> : Disable Low power mode disabled during WFI mode                          |
| Input Parameter 5      | <i>LPWFI_Clock</i> : the selected clock during LPWFI.<br>– <i>LPWFI_CLK2_16</i> : CLK2/16 is selected during LPWFI<br>– <i>LPWFI_CK_AF</i> : CK_AF is selected during LPWFI  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example disables the PLL1 automatically when CK\_AF is selected as source clock, puts the FLASH in PWD mode, disables the MVR during STOP and WFI mode, enables low power mode during WFI mode (LPWI) and selects CK\_AF as source clock during LPWFI mode.

```

{
/* User code */
/* Set the MVR disabled and the FLASH in PWD mode during STOP
mode */
PCU_LPModesConfig(DISABLE, DISABLE, DISABLE, ENABLE, LPWFI_CK_AF);
/*      |      |      |      |      |_CK_AF is selected during LPWFI
      |      |      |      |      |_Enable Low power mode during WFI mode
      |      |      |      |      |_FLASH in powerdown during LPWFI and STOP modes
      |      |      |      |      |_MVREG is disabled during LPWFI and STOP modes
      |      |      |      |      |_PLL1 is disabled automatically when CK_AF is selected */
/* User code */

```





**PCU\_STANDBY**

| Function Name          | PCU_STANDBY                                |
|------------------------|--|
| Function Prototype     | void PCU_STANDBY (void);                   |
| Behavior Description   | This routine is used to enter STANDBY mode |
| Input Parameter        | None                                       |
| Output Parameter       | None                                       |
| Return Value           | None                                       |
| Required preconditions | None                                       |
| Called Functions       | None                                       |

**Example:**

This examples configures the RTC alarm then goes into STANDBY mode

```

{
  /* Configure RTC alarm after RTC_ALARM_DELAY s */
  RTC_AlarmConfig(RTC_CounterValue() + RTC_ALARM_DELAY);

  /* Go into Standby */
  PCU_STANDBY();
}

```

**PCU\_FlashBurstCmd**

| Function Name          | PCU_FlashBurstCmd  |
|------------------------|--|
| Function Prototype     | PCU_FlashBurstCmd(FunctionalState NewState);   |
| Behavior Description   | This routine is used to set the FLASH to LP/BURST mode   |
| Input Parameter        | <i>NewStatus</i> : specifies whether the FLASH BURST mode is enabled or disabled.<br>– ENABLE: FLASH in BURST mode (default mode)<br>– DISABLE: FLASH in LP mode, the maximum allowed execution frequency = 33 MHz |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example shows how to enable/disable FLASH Burst mode.

```

{
  /* Set the FLASH in LP mode */
  PCU_FlashBurstCmd(DISABLE);

  /* Set the FLASH in BURST mode */
  PCU_FlashBurstCmd(ENABLE);
}

```

**PCU\_32OSCCmd**

| Function Name          | PCU_32OSCCmd  |
|------------------------|---|
| Function Prototype     | void PCU_32OSCCmd (FunctionalState NewState)  |
| Behavior Description   | Enables or disables the 32 kHz oscillator   |
| Input Parameter        | <i>NewState</i> : specifies whether the 32 kHz oscillator is enabled or disabled.<br>– <i>ENABLE</i> : enables the 32 kHz oscillator.<br>– <i>DISABLE</i> : disables the 32 kHz oscillator. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable the 32 kHz oscillator.

```
{ ...

    /* Enable the 32 kHz oscillator */
    PCU_32OSCCmd(ENABLE);
    ...
}
```

**3.2 Reset and clock control unit (RCCU)**

The *RCCU* driver may be used for a variety of purposes, including internal clock configuration and clock source selection.

The first section describes the data structure members used in the *RCCU* firmware library. The second section presents the *RCCU* firmware library functions.

**3.2.1 Data structures****RCCU register structure**

The *RCCU* peripheral register structure *RCCU\_TypeDef* is defined in the *71x\_map.h* file as follows:

```
typedef volatile struct
{
    vu32 CCR;
    u32 EMPTY1;
    vu32 CFR;
    u32 EMPTY2[3];
    vu32 PLL1CR;
    vu32 PER;
    vu32 SMR;
} RCCU_TypeDef;
```

Table 10 presents the *RCCU* registers.

**Table 10. RCCU registers**

| Register | Description                        |
|----------|------------------------------------|
| CCR      | Clock Control Register             |
| CFR      | Clock Flag Register                |
| PLL1CR   | <i>PLL1</i> configuration Register |
| PER      | Peripheral Clock Enable Register   |
| SMR      | System Mode Register               |

The *RCCU* peripheral is declared in the same file:

```

...
#define RCCU_BASE    0xA0000000
...
#ifndef DEBUG
...
#ifdef _RCCU
    #define RCCU                ((RCCU_TypeDef *)RCCU_BASE)
#endif /*RCCU*/
...
#else /* DEBUG */
...
#ifdef _RCCU
    EXT RCCU_TypeDef            *RCCU;
#endif /*RCCU*/
...
#endif

```

When debug mode is used, *RCCU* pointer is initialized in *71x\_lib.c* file:

```

#ifdef _RCCU
#ifdef _RCCU
    RCCU = (RCCU_TypeDef *)RCCU_BASE;
#endif

```

In debug mode the following macro is defined in *71x\_conf.h* file:

- *\_RCCU* is defined to access the corresponding peripheral registers.  
#define \_RCCU

### Clock divider parameters

The following enumeration defines the clock divider parameters. *Clock\_Div* enumeration is defined in the file *71x\_rccu.h*:

```

typedef enum
{
    RCCU_DEFAULT = 0x00,
    RCCU_RCLK_2 = 0x01,
    RCCU_RCLK_4 = 0x02,
    RCCU_RCLK_8 = 0x03
} RCCU_Clock_Div;

```

[Table 11](#) lists the different values of the clock divider parameter.

**Table 11. Clock divider parameters**

| Clock Divider | Description                                     |
|---------------|---|
| RCCU_DEFAULT  | The default divider (reset value) division by 1 |
| RCCU_RCLK_2   | Divide the <i>RCLK</i> by 2                     |
| RCCU_RCLK_4   | Divide the <i>RCLK</i> by 4                     |
| RCCU_RCLK_8   | Divide the <i>RCLK</i> by 8                     |

### RCLK clock source

The following enumeration defines the *RCLK* clock sources. *RCLK\_Clocks* enumeration is defined in the file *71x\_rccu.h*:

```
typedef enum
{
    RCCU_PLL1_Output,
    RCCU_CLOCK2_16,
    RCCU_CLOCK2,
    RCCU_CK_AF
} RCCU_RCLK_Clocks;
```

[Table 12](#) describes the *RCLK* clock sources.

**Table 12. RCLK clock sources**

| RCLK Clock Source | Description   |
|-------------------|---|
| RCCU_PLL1_Output  | Select the <i>PLL1</i> output as <i>RCLK</i> clock source |
| RCCU_CLOCK2_16    | Select the <i>RCLK2/16</i> as <i>RCLK</i> clock source    |
| RCCU_CLOCK2       | Select the <i>Clock2</i> as <i>RCLK</i> clock source      |
| RCCU_CK_AF        | Select the <i>RTC</i> as <i>RCLK</i> clock source         |

### PLL1 multiplication factors

The following enumeration defines the *PLL1* multiplication factors. *RCCU\_PLL1\_Mul* enumeration is declared in the file *71x\_rccu.h*:

```
typedef enum
{
    RCCU_PLL1_Mul_12 = 0x01,
    RCCU_PLL1_Mul_16 = 0x03,
    RCCU_PLL1_Mul_20 = 0x00,
    RCCU_PLL1_Mul_24 = 0x02
} RCCU_PLL1_Mul;
```

[Table 13](#) lists the different values of the *PLL1* multiplication factor.

**Table 13. PLL1 multiplication factors**

| PLL Multiplication factor | Description                       |
|---------------------------|-----------------------------------|
| RCCU_PLL1_Mul_12          | Multiplication factor equal to 12 |
| RCCU_PLL1_Mul_16          | Multiplication factor equal to 16 |
| RCCU_PLL1_Mul_20          | Multiplication factor equal to 20 |
| RCCU_PLL1_Mul_24          | Multiplication factor equal to 24 |

### PLL2 multiplication factors

The following enumeration defines the *PLL2* multiplication factors. *RCCU\_PLL2\_Mul* enumeration is declared in the file *71x\_rccu.h*:

```
typedef enum
{
    RCCU_PLL2_Mul_12 = 0x01,
    RCCU_PLL2_Mul_16 = 0x03,
    RCCU_PLL2_Mul_20 = 0x00,
    RCCU_PLL2_Mul_28 = 0x02
} RCCU_PLL2_Mul;
```

[Table 14](#) lists the different values of the *PLL2* multiplication factor.

**Table 14. PLL2 multiplication factors**

| PLL Multiplication factor | Description                       |
|---------------------------|-----------------------------------|
| RCCU_PLL2_Mul_12          | Multiplication factor equal to 12 |
| RCCU_PLL2_Mul_16          | Multiplication factor equal to 16 |
| RCCU_PLL2_Mul_20          | Multiplication factor equal to 20 |
| RCCU_PLL2_Mul_28          | Multiplication factor equal to 28 |

### PLL division factors

The following enumeration defines the *PLL* division factors. *RCCU\_PLL\_Div* enumeration is declared in the file *71x\_rccu.h*:

```
typedef enum
{RCCU_Div_1 = 0x00,
    RCCU_Div_2 = 0x01,
    RCCU_Div_3 = 0x02,
    RCCU_Div_4 = 0x03,
    RCCU_Div_5 = 0x04,
    RCCU_Div_6 = 0x05,
    RCCU_Div_7 = 0x06
} RCCU_PLL_Div;
```



[Table 15](#) lists the different values of the *PLL* division factor.

**Table 15. PLL division factors**

| PLL Division factor | Description                |
|---------------------|----------------------------|
| RCCU_Div_1          | Division factor equal to 1 |
| RCCU_Div_2          | Division factor equal to 2 |
| RCCU_Div_3          | Division factor equal to 3 |
| RCCU_Div_4          | Division factor equal to 4 |
| RCCU_Div_5          | Division factor equal to 5 |
| RCCU_Div_6          | Division factor equal to 6 |
| RCCU_Div_7          | Division factor equal to 7 |

### USB clock source

The following enumeration defines the *USB* clock sources. *USB\_Clocks* enumeration is defined in the file *71x\_rccu.h*:

```
typedef enum
{
  RCCU_PLL2_Output = 0x01,
  RCCU_USBCK = 0x00
} RCCU_USB_Clocks;
```

[Table 16](#) describes the *USB* clock sources.

**Table 16. USB clock sources**

| USB Clock Source | Description  |
|------------------|--|
| RCCU_PLL2_Output | Select the <i>PLL2</i> output as <i>USB</i> clock source |
| RCCU_USBCK       | Select the <i>USBCK</i> as <i>USB</i> clock source       |

### RCCU internal clocks

The following enumeration defines the *RCCU* internal clocks. *RCCU\_Clocks* enumeration is defined in the file *71x\_rccu.h*:

```
typedef enum
{
  RCCU_CLK2,
  RCCU_RCLK,
  RCCU_MCLK,
  RCCU_PCLK2,
  RCCU_PCLK1
} RCCU_Clocks;
```

[Table 17](#) describes the RCCU internal clocks.

**Table 17. RCCU internal clocks**

| RCCU Clock | Description   |
|------------|---|
| RCCU_CLK2  | The reference input clock to the programmable Phase Locked Loop frequency multiplier. |
| RCCU_RCLK  | Clock Output from <i>RCCU</i>   |
| RCCU_MCLK  | Main system clock, to ARM & memory  |
| RCCU_PCLK1 | APB1 peripheral clock.  |
| RCCU_PCLK2 | APB2 peripheral clock.  |

### RCCU interrupts

The following enumeration defines the *RCCU* interrupt source. *RCCU\_Interrupts* enumeration is defined in the file *71x\_rccu.h*:

```
typedef enum
{
    RCCU_PLL1_LOCK_IT,
    RCCU_CKAF_IT,
    RCCU_CK2_16_IT,
    RCCU_STOP_IT
} RCCU_Interrupts;
```

[Table 18](#) describes the *RCCU* interrupts.

**Table 18. RCCU interrupts**

| RCCU Interrupt    | Description                                  |
|-------------------|--|
| RCCU_PLL1_LOCK_IT | <i>PLL1</i> lock interrupt                   |
| RCCU_CKAF_IT      | Clock alternate function switching interrupt |
| RCCU_CK2_16_IT    | Clock2/16 switching interrupt                |
| RCCU_STOP_IT      | Stop interrupt                               |

### RCCU flags

The following enumeration defines the *RCCU* flags. *RCCU\_Flags* enumeration is defined in the file *71x\_rccu.h*:

```
typedef enum
{
    RCCU_PLL1_LOCK = 0x0002,
    RCCU_CKAF_ST = 0x0004,
    RCCU_PLL1_LOCK_I = 0x0800,
    RCCU_CKAF_I = 0x1000,
    RCCU_CK2_16_I = 0x2000,
    RCCU_STOP_I = 0x4000
} RCCU_Flags;
```

Table 19 describes the *RCCU* flags.

**Table 19. RCCU flags**

| RCCU flag        | Description   |
|------------------|---|
| RCCU_PLL1_LOCK   | PLL1 lock interrupt pending flag                          |
| RCCU_CKAF_ST     | CK_AF Status  |
| RCCU_PLL1_LOCK_I | PLL1 Lock Interrupt pending bit                           |
| RCCU_CKAF_I      | Clock alternate function switching interrupt pending flag |
| RCCU_CK2_16_I    | Clock2/16 switching interrupt pending flag                |
| RCCU_STOP_I      | Stop Interrupt pending bit                                |

### Reset sources

The following enumeration defines the *RCCU* reset sources. *RCCU\_ResetSources* enumeration is defined in the file *71x\_rccu.h*:

```
typedef enum {
    RCCU_ExternalReset = 0x00000000,
    RCCU_SoftwareReset = 0x00000020,
    RCCU_WDGReset = 0x00000040,
    RCCU_RTCAAlarmReset = 0x00000080,
    RCCU_LVDReset = 0x00000200,
    RCCU_WKPRReset = 0x00000400
}RCCU_ResetSources;
```

Table 20 describes the *RCCU* reset sources.

**Table 20. RCCU reset sources**

| RCCU reset sources  | Description   |
|---------------------|---|
| RCCU_ExternalReset  | Reset caused by external pin  |
| RCCU_SoftwareReset  | Reset caused by software  |
| RCCU_WDGReset       | Reset caused by Watchdog  |
| RCCU_RTCAAlarmReset | Reset caused by <i>RTC</i> to wake-up the system from StandBy mode  |
| RCCU_LVDReset       | Reset caused by <i>LVD</i>  |
| RCCU_WKPRReset      | Reset caused by wake-up pin to wake-up the system from StandBy mode |

### PLL1 free running modes

The following enumeration defines the PLL1 free running modes.

*RCCU\_PLL1FreeRunningModes* enumeration is declared in the file *71x\_rccu.h*:

```
typedef enum {
    RCCU_PLL1FreeRunning125,
    RCCU_PLL1FreeRunning250,
    RCCU_PLL1FreeRunning500
}RCCU_PLL1FreeRunningModes;
```

[Table 21](#) describes the PLL1 free running modes.

**Table 21. PLL1 free running modes**

| PLL1 free running modes | Description           |
|-------------------------|-----------------------|
| RCCU_PLL1FreeRunning125 | PLL1 provides 125 kHz |
| RCCU_PLL1FreeRunning250 | PLL1 provides 250 kHz |
| RCCU_PLL1FreeRunning500 | PLL1 provides 500 kHz |

### 3.2.2 Firmware library functions

The following table enumerates the different functions of the *RCCU* library.

| Function Name                                  | Description  |
|--|--|
| <a href="#">RCCU_Div2Config</a>                | Enables/Disables the programmable clock division by two          |
| <a href="#">RCCU_Div2Status</a>                | Checks if the programmable clock division by 2 is enabled or not |
| <a href="#">RCCU_MCLKConfig</a>                | Configures the <i>MCLK</i> clock divider coefficient             |
| <a href="#">RCCU_PCLK1Config</a>               | Configures the <i>PCLK1</i> clock divider coefficient            |
| <a href="#">RCCU_PCLK2Config</a>               | Configures the <i>PCLK2</i> clock divider coefficient            |
| <a href="#">RCCU_PLL1Config</a>                | Configures the <i>PLL1</i> multiplication and divider parameters |
| <a href="#">RCCU_PLL2Config</a>                | Configures the <i>PLL2</i> multiplication and divider parameters |
| <a href="#">RCCU_RCLKSourceConfig</a>          | Configures the <i>RCLK</i> clock source                          |
| <a href="#">RCCU_RCLKClockSource</a>           | Reads and returns the <i>RCLK</i> clock source                   |
| <a href="#">RCCU_USBCLKConfig</a>              | Configures the <i>USB</i> clock source                           |
| <a href="#">RCCU_USBClockSource</a>            | Reads and returns the <i>USB</i> clock source                    |
| <a href="#">RCCU_FrequencyValue</a>            | Computes any internal <i>RCCU</i> clock frequency                |
| <a href="#">RCCU_ITConfig</a>                  | Configures the <i>RCCU</i> interrupts                            |
| <a href="#">RCCU_FlagStatus</a>                | Checks the <i>RCCU</i> interrupt status                          |
| <a href="#">RCCU_FlagClear</a>                 | Clears an <i>RCCU</i> flag                                       |
| <a href="#">RCCU_ResetSource</a>               | Returns the source of the system reset                           |
| <a href="#">RCCU_PLL1FreeRunningModeConfig</a> | This routine is used to configure the PLL1 in free running mode  |
| <a href="#">RCCU_PLL1Disable</a>               | This routine is used to switch off the PLL1                      |
| <a href="#">RCCU_PLL2Disable</a>               | This routine is used to switch off the PLL2                      |
| <a href="#">RCCU_GenerateSWReset</a>           | This routine generates software reset.                           |

### RCCU\_Div2Config

| Function Name          | RCCU_Div2Config  |
|------------------------|--|
| Function Prototype     | <code>void RCCU_Div2Config(FunctionalState NewState);</code>   |
| Behavior Description   | This routine is used to enable or disable the programmable clock division of the <i>CLOCK1</i> input clock signal by 2. It sets or clears the <i>Div2</i> flag in the <i>CLK_FLAG</i> register   |
| Input Parameter        | <i>NewState</i> : specifies whether the programmable divider can divide the <i>CLOCK1</i> input clock signal by two or not<br>– <i>ENABLE</i> : enables the clock division by two of of CLK signal<br>– <i>DISABLE</i> : disables the clock division by two of of CLK signal |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to enable the clock division by two:

```
RCCU_Div2Config (ENABLE);
/* Sets the Div2 flag in the CLK_FLAG register. */
```

### RCCU\_Div2Status

| Function Name          | RCCU_Div2Status  |
|------------------------|--|
| Function Prototype     | <code>FlagStatus RCCU_Div2Status (void);</code>  |
| Behavior Description   | This routine gets the current status of the programmable clock division by two. It checks the status of the <i>Div2</i> flag in the <i>CLK_FLAG</i> register |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | The <i>Div2</i> Flag status.<br>– <i>SET</i> : The division by 2 is enabled<br>– <i>RESET</i> : The division by 2 is disabled                                |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to enable the programmable clock divider if it is not enabled.

```
FlagStatus Current_Div2Flag;
if (Current_Div2Flag == RESET)
{
    RCCU_Div2Config (ENABLE);
    /*.....User Code.....*/
}
```

**RCCU\_MCLKConfig**

| Function Name          | RCCU_MCLKConfig  |
|------------------------|--|
| Function Prototype     | <code>void RCCU_MCLKConfig(Clock_Div New_Clock);</code>  |
| Behavior Description   | This routine is used to configure the <i>MCLK</i> clock divider.   |
| Input Parameter        | <i>New_Clock</i> : Specifies the <i>MCLK</i> clock divider value.<br>Refer to <a href="#">Clock divider parameters on page 30</a> for more details on the allowed values of this parameter |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Note:** *The MCLK frequency must be greater or equal to the frequency of APB1 and APB2 bridges.*

**Example:**

This example illustrates how to set the *MCLK* clock to *RCLK / 1*.

```
{
...
RCCU_MCLKConfig (RCCU_DEFAULT);
...
}
```

**RCCU\_PCLK1Config**

| Function Name          | RCCU_PCLK1Config   |
|------------------------|--|
| Function Prototype     | <code>void RCCU_PCLK1Config(Clock_Div New_Clock);</code>   |
| Behavior Description   | This routine is used to select the division factor for <i>RCLK</i> to obtain the <i>PCLK1</i> clock for the APB1 fast peripherals ( <i>PCLK1</i> ).  |
| Input Parameter        | <i>New_Clock</i> : Specifies the <i>PCLK1</i> clock divider value.<br>Refer to <a href="#">Clock divider parameters on page 30</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to set the *PCLK1* clock to *RCLK / 8*.

```
{
..
RCCU_PCLK1Config (RCCU_RCLK_8);
...
}
```

## RCCU\_PCLK2Config

| Function Name          | RCCU_PCLK2Config  |
|------------------------|---|
| Function Prototype     | <code>void RCCU_PCLK2Config(Clock_Div New_Clock);</code>  |
| Behavior Description   | This routine is used to select the division factor for RCLK to obtain the PCLK2 clock for the APB2 peripherals (PCLK2).   |
| Input Parameter        | <i>New_Clock</i> : specifies the <i>PCLK2</i> clock divider value. Refer to <a href="#">Clock divider parameters on page 30</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to set the *PCLK2* clock to *RCLK / 2*.

```
{
...
RCCU_PCLK2Config (RCCU_RCLK_2);
...
}
```

## RCCU\_PLL1Config

| Function Name          | RCCU_PLL1Config   |
|------------------------|---|
| Function Prototype     | <code>void RCCU_PLL1Config (RCCU_PLL1_Mul New_Mul, RCCU_PLL_Div New_Div );</code>   |
| Behavior Description   | This routine configures the PLL1 division and multiplication factors.   |
| Input Parameter 1      | <i>New_Mul</i> : the <i>PLL1</i> clock multiplication factor. Refer to <a href="#">PLL1 multiplication factors on page 31</a> for more details on the allowed values of this parameter. |
| Input Parameter 2      | <i>New_Div</i> : the <i>PLL1</i> clock division factor. Refer to <a href="#">PLL division factors on page 32</a> for more details on the allowed values of this parameter.              |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to set the *PLL1* multiplication factor to 20 and the divider factor to 5.

```
{
...
RCCU_PLL1Config (RCCU_PLL1_Mul_20, RCCU_Div_5);
...
}
```

**RCCU\_PLL2Config**

| Function Name          | RCCU_PLL2Config   |
|------------------------|---|
| Function Prototype     | <code>void RCCU_PLL2Config (RCCU_PLL2_Mul New_Mul, RCCU_PLL_Div New_Div, u32 HCLK_Clock);</code>  |
| Behavior Description   | Configures the PLL2 division and multiplication factors.  |
| Input Parameter 1      | <i>New_Mul</i> : the <i>PLL2</i> clock multiplication factor. Refer to <a href="#">PLL2 multiplication factors on page 32</a> for more details on the allowed values of this parameter. |
| Input Parameter 2      | <i>New_Div</i> : the <i>PLL2</i> clock division factor. Refer to <a href="#">PLL division factors on page 32</a> for more details on the allowed values of this parameter.              |
| Input Parameter 3      | <i>HCLK_Clock</i> : the clock value present on HCLK pin (in Hz).  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to set the *PLL2* multiplication factor to 28 and the divider factor to 7.

```
{
...
RCCU_PLL2Config (RCCU_PLL2_Mul_28, RCCU_Div_7, 4000000);
...
}
```

**RCCU\_RCLKSourceConfig**

| Function Name          | RCCU_RCLKSourceConfig  |
|------------------------|--|
| Function Prototype     | <code>void RCCU_RCLKSourceConfig(RCLK_Clocks New_Clock);</code>  |
| Behavior Description   | This routine selects the clock source for the <i>RCLK</i> .  |
| Input Parameter        | <i>New_Clock</i> : the <i>RCLK</i> clock source. Refer to <a href="#">RCLK clock source on page 31</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |



**Example:**

This example illustrates how to set the *PLL1* multiplication factor to 20 and the divider factor to 4 then select *PLL1* output as clock source of *RCLK* clock.

```

{
...
RCCU_PLL1Config (RCCU_Mul_20, RCCU_Div_4);
/* Wait PLL to lock */
while(RCCU_FlagStatus (RCCU_PLL1_LOCK) == RESET);
RCCU_RCLKSourceConfig (RCCU_PLL1_Output);
...
}

```

**RCCU\_RCLKClockSource**

| Function Name          | RCCU_RCLKClockSource   |
|------------------------|--|
| Function Prototype     | RCLK_Clocks RCCU_RCLKClockSource (void);   |
| Behavior Description   | This routine gets and returns the current clock source of the <i>RCLK</i> .  |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | The current <i>RCLK</i> clock source.<br>Refer to <a href="#">RCLK clock source on page 31</a> for more details on the allowed values of this parameter. |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to get the *RCLK* clock source.

```

{
...
RCLK_Clocks RCLK_CLOCK_SOURCE;
RCLK_CLOCK_SOURCE = RCCU_RCLKClockSource ();
...
}

```

**RCCU\_USBCLKConfig**

| Function Name          | RCCU_USBCLKConfig   |
|------------------------|---|
| Function Prototype     | <code>void RCCU_USBCLKConfig (USB_Clocks New_Clock);</code>   |
| Behavior Description   | This routine is used to select the clock source of the <i>USB</i> peripheral.   |
| Input Parameter        | <i>New_Clock</i> : the USB clock source. Refer to <a href="#">USB clock source on page 33</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the *USB* peripheral source clock as *PLL2* output.

```
{
...
RCCU_USBCLKConfig (RCCU_PLL2_Output);
...
}
```

**RCCU\_USBClockSource**

| Function Name          | RCCU_USBClockSource   |
|------------------------|---|
| Function Prototype     | <code>USB_Clocks RCCU_USBClockSource (void);</code>   |
| Behavior Description   | This routine gets the clock source of the <i>USB</i> peripheral.  |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The <i>USB</i> clock source. Refer to <a href="#">USB clock source on page 33</a> for more details on the allowed values of this parameter. |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to get the *USB* clock source:

```
{
RCCU_USB_Clocks Current_USBClock;
...
Current_USBClock = RCCU_USBClockSource ();
...
}
```

## RCCU\_FrequencyValue

| Function Name          | RCCU_FrequencyValue   |
|------------------------|---|
| Function Prototype     | <code>u32 RCCU_FrequencyValue(RCCU_Clocks Internal_Clk);</code>   |
| Behavior Description   | This routine computes any internal <i>RCCU</i> clock frequency value passed in parameters.  |
| Input Parameter        | <i>Internal_Clk</i> : the <i>RCCU</i> internal clock to compute the frequency. Refer to <a href="#">RCCU internal clocks on page 33</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The frequency value of the specified clock in Hz of the internal clock passed in parameter.   |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how get the *MCLK* clock frequency value.

```
{
    u32 MCLK_Freq;
    ...
    MCLK_Freq = RCCU_FrequencyValue (RCCU_MCLK);
    ...
}
```

## RCCU\_ITConfig

| Function Name          | RCCU_ITConfig  |
|------------------------|--|
| Function Prototype     | <code>void RCCU_ITConfig (RCCU_Interrupts RCCU_IT, FunctionalState NewState);</code>   |
| Behavior Description   | This routine configures the specified <i>RCCU</i> interrupts.  |
| Input Parameter 1      | <i>RCCU_IT</i> : specifies the <i>RCCU</i> interrupt to configure. Refer to <a href="#">RCCU interrupts on page 34</a> for more details on the allowed values of this parameter.         |
| Input Parameter 2      | <i>NewState</i> : Indicates the new status of the <i>RCCU</i> interrupt.<br>– <i>ENABLE</i> : to enable the specified interrupt<br>– <i>DISABLE</i> : to disable the specified interrupt |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to configure the *PLL1* lock interrupt.

```
{
    ...
    RCCU_ITConfig (RCCU_PLL1_LOCK_IT, ENABLE);
    ...
}
```

**RCCU\_FlagStatus**

| Function Name          | RCCU_FlagStatus   |
|------------------------|---|
| Function Prototype     | FlagStatus RCCU_FlagStatus(RCCU_Flags RCCU_flag);   |
| Behavior Description   | Checks whether the specified interrupt is enabled or disabled.  |
| Input Parameter        | <i>RCCU_flag</i> : the flag to see its status. Refer to <a href="#">RCCU flags on page 34</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The specified interrupt status.<br>– <i>SET</i> : The specified flag is set.<br>– <i>RESET</i> : The specified flag is reset                            |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to check if the *PLL1* is locked.

```

{
...
    while (RCCU_FlagStatus (RCCU_PLL1_LOCK_IT) == RESET);
...
}

```

**RCCU\_FlagClear**

| Function Name          | RCCU_FlagClear   |
|------------------------|--|
| Function Prototype     | void RCCU_FlagClear (RCCU_Flags RCCU_flag);  |
| Behavior Description   | This routine is used to clear the specified interrupt flag in the RCCU registers passed in parameter.  |
| Input Parameter        | <i>RCCU_flag</i> : Specifies the flag to clear. Refer to <a href="#">RCCU flags on page 34</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to clear *PLL1* lock pending interrupt flag.

```

{
...
    RCCU_FlagClear (RCCU_PLL1_LOCK_I);
...
}

```

## RCCU\_ResetSource

| Function Name          | RCCU_ResetSource  |
|------------------------|---|
| Function Prototype     | RCCU_ResetSources RCCU_ResetSource (void);              |
| Behavior Description   | This routine is used to return the system reset source. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The reset source  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to define the system reset source.

```

{
RCCU_ResetSources ResetSource;
ResetSource=RCCU_ResetSource();
switch (ResetSource)
{
case RCCU_RTCArmReset: GPIO_BitWrite(GPIO0,0,1);break;
case RCCU_WKPRReset: GPIO_BitWrite(GPIO0,1,1);break;
default :
GPIO_BitWrite(GPIO0,3,1);break;
}
}

```

## RCCU\_PLL1FreeRunningModeConfig

| Function Name          | RCCU_PLL1FreeRunningModeConfig   |
|------------------------|--|
| Function Prototype     | RCCU_PLL1FreeRunningModeConfig(RCCU_PLL1FreeRunningModes NewPll1FreeRunningMode);                                    |
| Behavior Description   | This routine is used to configure the PLL1 in free running mode  |
| Input Parameter        | RCCU_PLL1FreeRunningModes: PLL1 in free running mode.<br>Refer to <a href="#">PLL1 free running modes on page 35</a> |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example shows how to clock the system with the PLL1 configured in free running mode.

```

{
/*Configure the PLL in free running mode at 125kHz */
RCCU_PLL1FreeRunningModeConfig(RCCU_PLL1FreeRunning125);

/* Set the RCLK to the PLL1 output */
RCCU_RCLKSourceConfig(RCCU_PLL1_Output);
}

```

**RCCU\_PLL1Disable**

| Function Name          | RCCU_PLL1Disable                            |
|------------------------|---|
| Function Prototype     | void RCCU_PLL1Disable (void);               |
| Behavior Description   | This routine is used to switch off the PLL1 |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example shows how to switch off the PLL1.

```

{
    /* switch off the PLL1 */
    RCCU_PLL1Disable ();
}

```

**RCCU\_PLL2Disable**

| Function Name          | RCCU_PLL2Disable                            |
|------------------------|---|
| Function Prototype     | void RCCU_PLL2Disable (void);               |
| Behavior Description   | This routine is used to switch off the PLL2 |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example shows how to switch off the PLL2.

```

{
    /* switch off the PLL2 */
    RCCU_PLL2Disable ();
}

```

## RCCU\_GenerateSWReset

| Function Name          | RCCU_GenerateSWReset                               |
|------------------------|--|
| Function Prototype     | void RCCU_GenerateSWReset (void);                  |
| Behavior Description   | This routine is used to generate a software reset. |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example shows how to generate the software reset.

```

{
    /* Generate Software Reset */
    RCCU_GenerateSWReset ();
}

```

## 3.3 Advanced peripheral bus bridges (APB)

The *APB* driver may be used for a variety of purposes, including *APB* bridges configuration and error status reporting.

The first section describes the data structure members used in the *APB* firmware library. The second section presents the *APB* firmware library functions.

### 3.3.1 Data structures

#### APB register structure

The *APB* peripheral register structure *APB\_TypeDef* is defined in the *71x\_map.h* file as follows:

```

typedef struct
{
    vu32 CKDIS;
    vu32 SWRES;
} APB_TypeDef;

```

[Table 22](#) presents the *APB* registers.

**Table 22. APB registers**

| Register | Description                        |
|----------|------------------------------------|
| CKDIS    | <i>APB</i> Clock Disable Register  |
| SWRES    | <i>APB</i> Software Reset Register |

The *APB* bridges are declared in the same file:

```

...
#define APB1_BASE    0xC0000000
#define APB2_BASE    0xE0000000
....
#ifndef DEBUG
...
#ifdef _APB1
    #define APB1                ((APB_TypeDef *) (APB1_BASE+0x10))
#endif /*APB1*/

#ifdef _APB2
    #define APB2                ((APB_TypeDef *) (APB2_BASE+0x10))
#endif /*APB2*/
...
#else /* DEBUG */
...
#ifdef _APB1
    EXT APB_TypeDef            *APB1;
#endif /*APB1*/

#ifdef _APB2
    EXT APB_TypeDef            *APB2;
#endif /*APB2*/
...
#endif

```

When debug mode is used, *APB* pointer is initialized in *71x\_lib.c* file:

```

#ifdef _APB1
#ifdef _APB1
    APB1 = (APB_TypeDef *)    (APB1_BASE + 0x10);
#endif
#endif /* _APB1 */

#ifdef _APB2
    APB2 = (APB_TypeDef *)    (APB2_BASE + 0x10);
#endif /* _APB2 */

```

In debug mode the following macros are defined in *71x\_conf.h* file:

- *\_APB* is defined to include the *APB* library
- *\_APB1* and *\_APB2* are defined to access the corresponding peripheral registers.

```

#define _APB
#define _APB1
#define _APB2

```



## Bridge peripheral codes

[Table 23](#) enumerates the various peripherals and their defined codes depending on which bridge they belong.

**Table 23. Bridge peripheral codes**

| Peripheral      | Bridge      | Code definition |        |
|-----------------|-------------|-----------------|--------|
| I2C0_Periph     | APB1 Bridge | 0x0001          |        |
| I2C1_Periph     |             | 0x0002          |        |
| UART0_Periph    |             | 0x0008          |        |
| UART1_Periph    |             | 0x0010          |        |
| UART2_Periph    |             | 0x0020          |        |
| UART3_Periph    |             | 0x0040          |        |
| USB_Periph      |             | 0x0080          |        |
| CAN_Periph      |             | 0x0100          |        |
| BSPI0_Periph    |             | 0x0200          |        |
| BSPI1_Periph    |             | 0x0400          |        |
| HDLC_Periph     |             | 0x2000          |        |
| APB1_ALL_Periph |             | 0x27FB          |        |
| XTI_Periph      |             | APB2 Bridge     | 0x0001 |
| GPIO0_Periph    |             |                 | 0x0004 |
| GPIO1_Periph    | 0x0008      |                 |        |
| GPIO2_Periph    | 0x0010      |                 |        |
| ADC12_Periph    | 0x0040      |                 |        |
| CKOUT_Periph    | 0x0080      |                 |        |
| TIM0_Periph     | 0x0400      |                 |        |
| TIM1_Periph     | 0x0200      |                 |        |
| TIM2_Periph     | 0x0400      |                 |        |
| TIM3_Periph     | 0x0800      |                 |        |
| RTC_Periph      | 0x1000      |                 |        |
| EIC_Periph      | 0x8000      |                 |        |
| APB2_ALL_Periph | 0x5FDD      |                 |        |

Each peripheral code is defined in the *71x\_apb.h* file. The example below illustrates the definition of *I2C0* peripheral code:

```
#define I2C0_Periph 0x0001
```

### 3.3.2 Common parameter values

#### APBx values

[Table 24](#) shows the allowed values of the APBx variable.

**Table 24. APBx values**

| APBx | Description                |
|------|----------------------------|
| APB1 | Selects <i>APB1</i> bridge |
| APB2 | Selects <i>APB2</i> bridge |

### 3.3.3 Firmware library functions

[Table 25](#) enumerates the different functions of the *APB* library.

**Table 25. APB library functions**

| Function Name                     | Description   |
|-----------------------------------|---|
| <a href="#">APB_ClockConfig</a>   | Enables/Disables the peripheral clock gating on the specified <i>APB</i> bridge.        |
| <a href="#">APB_SwResetConfig</a> | Enables/Disables the software reset for peripherals on the specified <i>APB</i> bridge. |

#### APB\_ClockConfig

| Function Name          | APB_ClockConfig   |
|------------------------|---|
| Function Prototype     | <code>void APB_ClockConfig (APB_TypeDef *APBx, FunctionalState NewState, u16 Periph);</code>  |
| Behavior Description   | Enables/Disables the peripheral clock gating on the specified <i>APB</i> bridge.  |
| Input Parameter 1      | APBx: selects the <i>APB</i> bridge (APB1 or APB2)  |
| Input Parameter 2      | <i>NewStatus</i> : specifies if the peripheral clock is running or stopped.<br>– <i>ENABLE</i> : The peripheral clock is running<br>– <i>DISABLE</i> : The peripheral clock is stopped. |
| Input Parameter 3      | <i>Periph</i> : specifies the <i>APB</i> bridge peripheral:<br>Refer to <a href="#">Bridge peripheral codes on page 49</a> for more details on the allowed values of this parameter.    |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

- Note:
- 1 One or more peripheral clocks can be enabled on the same *APB* bridge with a single *APB\_ClockConfig* function call. One or more peripheral clocks can be disabled with another *APB\_ClockConfig* function call.
  - 2 It is not possible to simultaneously enable and disable several peripheral clocks in a single *APB\_ClockConfig* function call.

**Example:**

This example enables clock gating for *I2C0* and *UART1* peripherals on the *APB1* bridge and disables the clock for *GPIO2*, *ADC12* peripherals on the *APB2* bridge.

```
{
  /* Enables the clock signal to the I2C0 peripheral. */
  APB_ClockConfig (APB1, ENABLE, I2C0_Periph);

  /* Enables the clock signal to the UART1 peripheral.*/
  APB_ClockConfig (APB1, ENABLE, UART1_Periph);

  /* Disables the clock signal to the GPIO2 peripheral. */
  APB_ClockConfig (APB2, DISABLE, GPIO2_Periph);

  /* Disables the clock signal to the ADC12 peripheral. */
  APB_ClockConfig (APB2, DISABLE, ADC12_Periph);
}
```

The *APB\_ClockConfig* parameter 3 can accept several peripherals using the logical operator OR.

```
{
  /* Enables the clock signal to the I2C0 & UART1 peripherals */
  APB_ClockConfig (APB1, ENABLE, I2C0_Periph | UART1_Periph);

  /* Disables the clock signal to the GPIO2 & ADC12 peripherals */
  APB_ClockConfig (APB2, DISABLE, GPIO2_Periph | ADC12_Periph);
}
```

## APB\_SwResetConfig

| Function Name          | APB_SwResetConfig   |
|------------------------|---|
| Function Prototype     | void APB_SwResetConfig (APB_TypeDef *APBx, FunctionalState New_Status, ul6 Periph);   |
| Behavior Description   | This function enables/disables the software reset for the specified peripheral on the specified APB Bridge. It updates the bits[14:0] of the <i>SWRES</i> register with the value of the input parameter. |
| Input Parameter 1      | APBx: selects the APB bridge (APB1 or APB2).  |
| Input Parameter 2      | <i>New_Status</i> : the peripheral software reset new status to be set.<br>– <i>ENABLE</i> : The peripheral is kept under reset<br>– <i>DISABLE</i> : The peripheral is reset by the system-wide reset    |
| Input Parameter 3      | <i>Periph</i> : specifies the APB bridge peripheral:<br>Refer to <a href="#">Bridge peripheral codes on page 49</a> for more details on the allowed values of this parameter.                             |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

- Note:**
- 1 One or more peripheral software resets can be enabled on the same APB bridge with a single *APB\_SwResetConfig* function call. One or more peripheral software resets can be disabled with another *APB\_SwResetConfig* function call.
  - 2 It is not possible to simultaneously enable and disable several peripheral software resets in a single *APB\_SwResetConfig* function call.

### Example:

This example illustrates how to configure the *BSPi0* and the *BSPi1* peripherals to be kept always under reset:

```
{
  APB_SwResetConfig (APB1, ENABLE, BSPi0_Periph | BSPi1_Periph);
}
```

## 3.4 Enhanced interrupt controller (EIC)

The driver may be used for several purposes, such as enabling and disabling interrupts (*IRQ*) and fast interrupts (*FIQ*), enabling and disabling individual *IRQ* channels, changing *IRQ* channel priorities, saving and restoring context and installing *IRQ* handlers.

The first and the second sections describe the data structure and the common parameter values. The second one presents the *EIC* firmware library functions.

### 3.4.1 Data structures

#### EIC register structure

The *EIC* peripheral register structure *EIC\_TypeDef* is defined in the file *71x\_map.h* as follows:

```
typedef volatile struct
{
    vu32 ICR;
    vu32 CICR;
    vu32 CIPR;
    u32 EMPTY1[3];
    vu32 IVR;
    vu32 FIR;
    vu32 IER;
    u32 EMPTY2[7];
    vu32 IPR;
    u32 EMPTY3[7];
    u32 SIR[32];
} EIC_TypeDef;
```

[Table 26](#) presents the *EIC* peripheral registers.

**Table 26. EIC peripheral registers**

| Register | Description                         |
|----------|-------------------------------------|
| ICR      | Interrupt Control Register          |
| CICR     | Current Interrupt Channel Register  |
| CIPR     | Current Interrupt Priority Register |
| IVR      | Interrupt Vector Register           |
| FIR      | Fast Interrupt Register             |
| IER      | Interrupt Enable Register           |
| IPR      | Interrupt Pending Register          |
| SIR      | Source Interrupt Registers          |

*EIC* peripheral is declared in the *71x\_map.h* file as follows:

```
...
/* EIC Base Address Definition */
#define EIC_BASE    0xFFFFF800
...
#ifndef DEBUG
...
#ifdef _EIC
#define EIC          ((EIC_TypeDef *)EIC_BASE)
#endif /*EIC*/
...
#else /* DEBUG */
...
#ifdef _EIC
    EXT EIC_TypeDef    *EIC;
#endif /*EIC*/
...
#endif
```

When debug mode is used, *EIC* pointer is initialized in *71x\_lib.c* file as follows:

```
#ifdef _EIC
    EIC = (EIC_TypeDef *)EIC_BASE;
#endif
```

In debug mode *\_EIC* variable must be defined in *71x\_conf.h* file to include the *EIC* library:

```
#define _EIC
```

### IRQ channel enumeration

The following enumeration defines the different values of the *IRQ* channels.

*IRQChannel\_TypeDef* enumeration is defined in *71x\_eic.h* file:

```
typedef enum
{
    TOTIMI_IRQChannel      = 0,
    FLASH_IRQChannel      = 1,
    RCCU_IRQChannel       = 2,
    RTC_IRQChannel        = 3,
    WDG_IRQChannel        = 4,
    XTI_IRQChannel        = 5,
    USBHP_IRQChannel      = 6,
    I2C0ITERR_IRQChannel  = 7,
    I2C1ITERR_IRQChannel  = 8,
    UART0_IRQChannel      = 9,
    UART1_IRQChannel      = 10,
    UART2_IRQChannel      = 11,
    UART3_IRQChannel      = 12,
    SPI0_IRQChannel       = 13,
    SPI1_IRQChannel       = 14,
    I2C0_IRQChannel       = 15,
    I2C1_IRQChannel       = 16,
    CAN_IRQChannel        = 17,
    ADC_IRQChannel        = 18,
    T1TIMI_IRQChannel     = 19,
    T2TIMI_IRQChannel     = 20,
    T3TIMI_IRQChannel     = 21,
    HDLC_IRQChannel       = 25,
    USBLP_IRQChannel      = 26,
    TOTOI_IRQChannel      = 29,
    TOOC1_IRQChannel      = 30,
    TOOC2_IRQChannel      = 31
} IRQChannel_TypeDef;
```

[Table 27](#) describes the different values of IRQChannel variable.

**Table 27. IRQChannel values**

| IRQChannel           | Peripheral Interrupt                | Value |
|----------------------|-------------------------------------|-------|
| T0TIMI_IRQChannel    | Timer 0 global interrupt            | 0     |
| FLASH_IRQChannel     | Embedded Flash global interrupt     | 1     |
| RCCU_IRQChannel      | RCCU global interrupt               | 2     |
| RTC_IRQChannel       | Real Time Clock global interrupts   | 3     |
| WDG_IRQChannel       | Watchdog timer interrupts           | 4     |
| XTI_IRQChannel       | WIU Wake-up event interrupt         | 5     |
| USBHP_IRQChannel     | USB high priority event interrupt   | 6     |
| I2C0ITERR_IRQChannel | I <sup>2</sup> C 0 error interrupt  | 7     |
| I2C1ITERR_IRQChannel | I <sup>2</sup> C 1 error interrupt  | 8     |
| UART0_IRQChannel     | UART 0 global interrupt             | 9     |
| UART1_IRQChannel     | UART 1 global interrupt             | 10    |
| UART2_IRQChannel     | UART 2 global interrupt             | 11    |
| UART3_IRQChannel     | UART 3 global interrupt             | 12    |
| SPI0_IRQChannel      | BSPI 0 global interrupt             | 13    |
| SPI1_IRQChannel      | BSPI 1 global interrupt             | 14    |
| I2C0_IRQChannel      | I <sup>2</sup> C 0 global interrupt | 15    |
| I2C1_IRQChannel      | I <sup>2</sup> C 1 global interrupt | 16    |
| CAN_IRQChannel       | CAN module global interrupt         | 17    |
| ADC_IRQChannel       | ADC sample ready interrupt          | 18    |
| T1TIMI_IRQChannel    | Timer 1 global interrupt            | 19    |
| T2TIMI_IRQChannel    | Timer 2 global interrupt            | 20    |
| T3TIMI_IRQChannel    | Timer 3 global interrupt            | 21    |
| HDLC_IRQChannel      | HDLC global interrupt               | 25    |
| USBLP_IRQChannel     | USB low priority event interrupt    | 26    |
| T0TOI_IRQChannel     | Timer 0 overflow interrupt          | 29    |
| T0OC1_IRQChannel     | Timer 0 output compare 1 interrupt  | 30    |
| T0OC2_IRQChannel     | Timer 0 output compare 2 interrupt  | 31    |

### FIQ channel enumeration

The following enumeration defines the different values of the *FIQ* channel.

*FIQChannel\_TypeDef* enumeration is defined in the file *etc.h*:

```
typedef enum
{
    T0TIMI_FIQChannel    = 0x00000001,
    WDG_FIQChannel      = 0x00000002,
    WDG_T0TIMI_FIQChannels = 0x00000003
} FIQChannel_TypeDef;
```

[Table 28](#) describes the different values of FIQChannel.

**Table 28. FIQChannel values**

| FIQChannel           | Interrupt Source  | Value |
|----------------------|---|-------|
| T0TIMI_FIQChannel    | Timer 0 global interrupt  | 1     |
| WDG_FIQChannel       | Watchdog timer interrupt  | 2     |
| WDGT0TIMI_FIQChannel | Both timer 0 global interrupt and watchdog interrupt. The two FIQs take place at the same time. | 3     |

### 3.4.2 Firmware library functions

The table below enumerates the different functions of the *EIC* library.

| Function Name                                  | Function Description   |
|--|--|
| <a href="#">EIC_Init</a>                       | Initializes the Enhanced Interrupt Controller                  |
| <a href="#">EIC_IRQConfig</a>                  | Enables or Disables IRQ interrupts in the EIC                  |
| <a href="#">EIC_FIQConfig</a>                  | Enables or Disables FIQ interrupts in the EIC                  |
| <a href="#">EIC_IRQChannelConfig</a>           | Enables or Disables the selected IRQ channel interrupts        |
| <a href="#">EIC_FIQChannelConfig</a>           | Enables or Disables the selected FIQ channel interrupts        |
| <a href="#">EIC_IRQChannelPriorityConfig</a>   | Sets the IRQ channel priority level                            |
| <a href="#">EIC_CurrentPriorityLevelValue</a>  | Returns the current priority level                             |
| <a href="#">EIC_CurrentPriorityLevelConfig</a> | Changes the current priority level                             |
| <a href="#">EIC_CurrentIRQChannelValue</a>     | Returns the current served IRQ channel number                  |
| <a href="#">EIC_CurrentFIQChannelValue</a>     | Returns the FIQ channel number causing the FIQ interrupt       |
| <a href="#">EIC_FIQPendingBitClear</a>         | Clears FIQ Pending bit according to the input passed parameter |



**EIC\_Init**

| Function Name          | EIC_Init   |
|------------------------|--|
| Function Prototype     | <code>void EIC_Init(void);</code>  |
| Behavior Description   | Initializes the Interrupt Controller as following: <ul style="list-style-type: none"> <li>– FIQ and IRQ are disabled,</li> <li>– All channel interrupts are disabled,</li> <li>– All pending bits are cleared.</li> <li>– The current priority level set to zero,</li> <li>– IVR register initialized with the upper half of the load PC instruction op-code,</li> <li>– All SIR registers are initialized with the offset to the corresponding IRQ routine vector.</li> </ul> |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Note:** *The EIC is initialized in the 71x\_init.s before entering the C code, so you don't need to initialize the EIC after a system reset. This function is required if a software reset is performed.*

**Example:**

This example shows how to initialize the interrupt controller.

```

{
...
/* Initialize the interrupt controller */
EIC_Init();
...
}

```

**EIC\_IRQConfig**

| Function Name          | EIC_IRQConfig   |
|------------------------|---|
| Function Prototype     | <code>void EIC_IRQConfig(FunctionalState NewState);</code>  |
| Behavior Description   | Enables or Disables IRQ interrupts.   |
| Input Parameter        | <i>NewState</i> : specifies whether the <i>IRQ</i> is enabled or disabled.<br>– <i>ENABLE</i> : enables the <i>IRQ</i> interrupt<br>– <i>DISABLE</i> : disables the <i>IRQ</i> interrupt. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">EIC_FIQConfig on page 58</a>  |

**Example:**

This example shows how to enable the *IRQ* interrupts.

```
{
...
/*Enable IRQ interrupts */
EIC_IRQConfig(ENABLE);
...
}
```

**EIC\_FIQConfig**

| Function Name          | EIC_FIQConfig   |
|------------------------|---|
| Function Prototype     | <code>void EIC_FIQConfig(FunctionalState NewState);</code>  |
| Behavior Description   | Sets or clears the <i>FIQ_EN</i> bit located in <i>EIC.CIR</i> register.  |
| Input Parameter        | <i>NewState</i> : the <i>FIQ</i> interrupt new status to be set.<br>– <i>ENABLE</i> : enables the <i>FIQ</i> interrupt<br>– <i>DISABLE</i> : disables the <i>FIQ</i> interrupt. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">EIC_IRQConfig on page 58</a>  |

**Example:**

This example shows how to enable the *FIQ* interrupts.

```
{
/*Enable FIQ interrupts */
EIC_FIQConfig(ENABLE);
...
}
```

## EIC\_IRQChannelConfig

| Function Name          | EIC_IRQChannelConfig  |
|------------------------|---|
| Function Prototype     | void EIC_IRQChannelConfig(IRQChannel_TypeDef IRQChannel, FunctionalState NewState);   |
| Behavior Description   | Configures the specified IRQ Channel.   |
| Input Parameter 1      | <i>IRQChannel</i> : the channel to be configured.<br>Refer to <a href="#">IRQ channel enumeration on page 54</a> for details on allowed values of <i>IRQChannel</i> parameter.                            |
| Input Parameter 2      | <i>NewState</i> : specifies whether the <i>IRQ</i> channel interrupt will be enabled or disabled.<br>– <i>ENABLE</i> : enables the channel interrupt<br>– <i>DISABLE</i> : disables the channel interrupt |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">EIC_IRQConfig on page 58</a> ,<br><a href="#">EIC_IRQChannelPriorityConfig on page 61</a>   |

### Example:

This example shows how to enable the Timer 0 global interrupts and disable the *UART0* interrupts.

```

{
...
/* Enable Timer 0 global interrupts */
EIC_IRQChannelConfig(T0TIMI_IRQChannel, ENABLE);
...
/* Disable UART 0 global interrupts */
EIC_IRQChannelConfig(UART0_IRQChannel, DISABLE);
...
}

```

**EIC\_FIQChannelConfig**

| Function Name          | EIC_FIQChannelConfig   |
|------------------------|--|
| Function Prototype     | <code>void EIC_FIQChannelConfig(FIQChannel_TypeDef FIQChannel, FunctionalState NewState);</code>   |
| Behavior Description   | Sets or clears the corresponding channel bit located in <i>EIC.FIR</i> register.   |
| Input Parameter 1      | <i>FIQChannel</i> : the FIQ channel to be configured.<br>Refer to <a href="#">FIQ channel enumeration on page 55</a> for details on allowed values of <i>FIQChannel</i> parameter.                                       |
| Input Parameter 2      | <i>NewState</i> : specifies whether the FIQ channel interrupt will be enabled or disabled.<br>– <i>ENABLE</i> : enables the <i>FIQ</i> channel interrupt<br>– <i>DISABLE</i> : disables the <i>FIQ</i> channel interrupt |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |
| See also               | <a href="#">EIC_FIQConfig on page 58</a>   |

**Example:**

This example how to disable the *WDG FIQ* interrupts and enable *Timer 0 FIQ* interrupts.

```

{
...
/* Disable WDG FIQ interrupts */
EIC_FIQChannelConfig(WDG_FIQChannel, DISABLE);
...
/* Enable Timer 0 FIQ interrupts */
EIC_FIQChannelConfig(TOTIMI_FIQChannel, ENABLE);
...
}

```

## EIC\_IRQChannelPriorityConfig

| Function Name          | EIC_IRQChannelPriorityConfig   |
|------------------------|--|
| Function Prototype     | <code>void EIC_IRQChannelPriorityConfig(<br/>IRQChannel_TypeDef IRQChannel, u8 Priority);</code>   |
| Behavior Description   | Configures the selected IRQ channel priority.  |
| Input Parameter 1      | <i>IRQChannel</i> : the channel to be configured.<br>Refer to <a href="#">IRQ channel enumeration on page 54</a> for details on allowed values of <i>IRQChannel</i> parameter. |
| Input Parameter 2      | <i>Priority</i> : specifies the priority of the channel. Priority variable must take an integer value in the range [0..15]   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |
| See also               | <a href="#">EIC_IRQConfig on page 58</a> ,<br><a href="#">EIC_IRQChannelConfig on page 59</a>  |

**Caution:** If you select a priority level equal to 0, this channel will not be served by the EIC.

### Example:

This example show how to enable timer 0 global interrupts with a priority level equal to 5.

```
{
...
/* Set the timer 0 global interrupts priority level to 5 */
EIC_IRQChannelPriorityConfig(TOTIMI_IRQChannel, 0x05);
...
/* Enable interrupts not the timer 0 global interrupts channel
*/
EIC_IRQChannelConfig(TOTIMI_IRQChannel, ENABLE);
...
/* Enable IRQ interrupts */
EIC_IRQConfig(ENABLE);
...
}
```

## EIC\_CurrentPriorityLevelValue

| Function Name          | EIC_CurrentPriorityLevelValue   |
|------------------------|---|
| Function Prototype     | u8 EIC_CurrentPriorityLevelValue(void);                               |
| Behavior Description   | Returns the current priority level of the current served IRQ routine. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The current priority level  |
| Required Preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">EIC_CurrentPriorityLevelConfig on page 62</a>             |

### Example:

This example shows how to get the priority level of the current interrupt service routine:

```

{
    vu8 bCurrentPriorityLevelValue;
    /* Get the current served IRQ channel priority level.*/
    bCurrentPriorityLevelValue = EIC_CurrentPriorityLevelValue();
}

```

## EIC\_CurrentPriorityLevelConfig

| Function Name          | EIC_CurrentPriorityLevelConfig  |
|------------------------|---|
| Function Prototype     | void EIC_CurrentPriorityLevelConfig(u8 NewPriorityLevel);   |
| Behavior Description   | This function changes the current priority level of the served IRQ routine.                             |
| Input Parameter        | <i>NewPriorityLevel</i> : specifies the new priority level. It must be an integer in the range [0..15]. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">EIC_CurrentPriorityLevelValue on page 62</a>  |

- Note:**
- 1 This function changes the current priority level, but doesn't change the channel priority level.
  - 2 If this function is used in an interrupt service routine, the new priority level must be higher than the current priority level.
  - 3 To decrease the current priority level you must ensure that all pending bits are cleared.
  - 4 If the `EIC_Init()` function is used after the `EIC_CurrentPriorityLevelConfig()` function the current priority level is reset to zero.

**Caution:** If this function is used in a non interrupt service routine, all IRQ channels with a priority level lower or equal to the new priority level will be disabled.

**Examples:**

The current priority level is increased by 1.

```
void an_IRQ_Routine(void)
{
    ...

    EIC_CurrentPriorityLevelConfig(EIC_CurrentPriorityLevelValue()+
    1);
    ...
}
```

This example shows how to disable all *IRQ* channel with a priority level lower or equal to 3.

```
int main(void)
{
    ...
    EIC_CurrentPriorityLevelConfig(3);
    ...
}
```

**EIC\_CurrentIRQChannelValue**

| Function Name          | EIC_CurrentIRQChannelValue  |
|------------------------|---|
| Function Prototype     | <code>IRQChannel_TypeDef EIC_CurrentIRQChannelValue(void);</code>   |
| Behavior Description   | Returns the current served IRQ channel number.  |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The current served IRQ channel number.<br>Refer to <a href="#">IRQ channel enumeration on page 54</a> for details on allowed values of <i>IRQChannel</i> parameter. |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example shows how to get the current *IRQ* channel level value.

```
{
    IRQChannel_TypeDef bCurrentIRQChannelValue;
    bCurrentIRQChannelValue = EIC_CurrentIRQChannelValue();
}
```

### EIC\_CurrentFIQChannelValue

| Function Name          | EIC_FIQChannelValue  |
|------------------------|--|
| Function Prototype     | FIQChannel_TypeDef EIC_CurrentFIQChannelValue(void);   |
| Behavior Description   | This function gets and return the current served FIQ channel number  |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | The current served FIQ channel number.<br>Refer to <a href="#">FIQ channel enumeration on page 55</a> for details on allowed values of FIQChannel parameter. |
| Required Preconditions | None   |
| Called Functions       | None   |
| See also               | <a href="#">EIC_FIQPendingBitClear on page 65</a>  |

**Note:** When you use both Timer 0 and Watchdog Timer FIQ interrupts in your application, this function is required in the FIQ interrupts service routine for software prioritization and to define which FIQ pending bit you have to clear.

**Example:**

This example shows how to use the *EIC\_FIQChannelValue* function in the FIQ interrupts service routine to define the source of FIQ interrupts.

```
void FIQ_Handler(void)
{
    switch (EIC_CurrentFIQChannelValue())
    {
        case TOTIMI_FIQChannel:
            TIM0->SR &= ~0x2000; /* Clear the Timer 0 interrupt flag */
            /* Clear the corresponding FIQ pending bit */
            EIC_FIQPendingBitClear(TOTIMI_FIQChannel);
            /*.....User code.....*/
            break;
        case WDG_FIQChannel:
            WDG->SR = 0x0000; /* Clear the WDG interrupt flag */
            /* Clear the corresponding FIQ pending bit */
            EIC_FIQPendingBitClear(WDG_FIQChannel);
            /*.....User code.....*/
            break;
        case WDGTOTIMI_FIQChannels:
            WDG->SR = 0x000; /* Clear the WDG interrupt flag */
            TIM0->SR &= ~0x2000; /* Clear the Timer 0 interrupt flag */
            /* Clear the FIQ pending bits */
            EIC_FIQPendingBitClear(WDGTOTIMI_FIQChannels);
            /*.....User code.....*/
            break;
    }
}
```



## EIC\_FIQPendingBitClear

| Function Name          | EIC_FIQPendingBitClear  |
|------------------------|---|
| Function Prototype     | <code>void EIC_FIQPendingBitClear(FIQChannel_TypeDef FIQChannel);</code>  |
| Behavior Description   | Clears the selected <i>FIQ</i> Pending bit located in <i>FIR</i> register by writing 1 to the corresponding bit.  |
| Input Parameter        | <i>FIQChannel</i> : specifies the <i>FIQ</i> channel to be configured.<br>Refer to <a href="#">FIQ channel enumeration on page 55</a> for details on allowed values of <i>FIQChannel</i> parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">EIC_CurrentFIQChannelValue on page 64</a>   |

### Example:

This example shows how to use of the *EIC\_FIQPendingBitClear* function in the *FIQ* interrupts service routine to define the source of *FIQ* interrupts.

```
void FIQ_Handler(void)
{
    TIM0->SR &= ~0x2000; /* Clear the Timer 0 interrupt flag */
    /* Clear the corresponding FIQ pending bit */
    EIC_FIQPendingBitClear(T0TIMI_FIQChannel);
}
```

## 3.5 General purpose input output (GPIO)

The GPIO driver may be used for several purposes, including pin configuration, reading a port pin and writing data into the port pin.

The first section describes the data structure used in the GPIO firmware library. The second section presents the GPIO firmware library functions.

### 3.5.1 Data structures

#### GPIO register structure

The GPIO peripheral register structure *GPIO\_TypeDef* is defined in the file *71x\_map.h* as follows:

```
typedef struct
{
    vu16 PC0;
    vu16 EMPTY1;
    vu16 PC1;
    vu16 EMPTY2;
    vu16 PC2;
    vu16 EMPTY3;
```

```

    vu16 PD;
    vu16 EMPTY4;
} GPIO_TypeDef;

```

[Table 29](#) describes the *GPIO* registers.

**Table 29. GPIO registers**

| Register | Description                   |
|----------|-------------------------------|
| PC0      | Port Configuration Register 0 |
| PC1      | Port Configuration Register 1 |
| PC2      | Port Configuration Register 2 |
| PD       | Data Register                 |

*GPIO* peripheral is declared in the *71x\_map.h* file as follows:

```

/* APB2 Base Address definition*/
#define APB2_BASE 0xE0000000

/* GPIO0 and GPIO1 Base Address definitions*/
#define GPIO0_BASE (APB2_BASE + 0x3000)
#define GPIO1_BASE (APB2_BASE + 0x4000)
#define GPIO2_BASE (APB2_BASE + 0x5000)

/* GPIO0 and GPIO1 peripherals declaration*/
#ifndef DEBUG
#ifdef _GPIO0
#define GPIO0 ((GPIO_TypeDef *)GPIO0_BASE)
#endif /*_GPIO0*/

#ifdef _GPIO1
#define GPIO1 ((GPIO_TypeDef *)GPIO1_BASE)
#endif /*_GPIO1*/

#ifdef _GPIO2
#define GPIO2 ((GPIO_TypeDef *)GPIO2_BASE)
#endif /*_GPIO2*/
...
#else /* DEBUG */
#ifdef _GPIO0
EXT GPIO_TypeDef *GPIO0;
#endif /*_GPIO0*/

#ifdef _GPIO1
EXT GPIO_TypeDef *GPIO1;
#endif /*_GPIO1*/

#ifdef _GPIO2
EXT GPIO_TypeDef *GPIO2;
#endif /*_GPIO2*/
...
#endif

```

When debug mode is used, *GPIO* pointer is initialized in *71x\_lib.c* file:

```
#ifdef _GPIO0
    GPIO0 = (GPIO_TypeDef *)GPIO0_BASE;
#endif

#ifdef _GPIO1
    GPIO1 = (GPIO_TypeDef *)GPIO1_BASE;
#endif

#ifdef _GPIO2
    GPIO2 = (GPIO_TypeDef *)GPIO2_BASE;
#endif
```

In debug mode the following variables are defined in *71x\_conf.h* file:

- `_GPIO` is defined to include the *GPIO* library
- `_GPIO0`, `_GPIO1` and `_GPIO2` are defined to access the peripheral registers.

```
#define TQFP144
#define _GPIO
#define _GPIO0
#define _GPIO1

#ifdef TQFP144
#define _GPIO2
#endif /*TQFP144*/
```

### GPIO pin mode enumeration

The following enumeration defines the different possible pin modes for *GPIO* peripheral. *GpioPinMode\_TypeDef* enumeration is defined in *71x\_gpio.h* file:

```
typedef enum
{
    GPIO_HI_AIN_TRI,
    GPIO_IN_TRI_TTL,
    GPIO_IN_TRI_CMOS,
    GPIO_IPUPD_WP,
    GPIO_OUT_OD,
    GPIO_OUT_PP,
    GPIO_AF_OD,
    GPIO_AF_PP
} GpioPinMode_TypeDef;
```

[Table 30](#) describes the pin modes.

**Table 30. GPIO pin modes**

| Pin mode         | Description                           |
|------------------|---------------------------------------|
| GPIO_HI_AIN_TRI  | High impedance Analog Input Tristate  |
| GPIO_IN_TRI_TTL  | Input Tristate TTL                    |
| GPIO_IN_TRI_CMOS | INPUT Tristate CMOS                   |
| GPIO_IPUPD_WP    | Input Pull-Up/Pull-Down Configuration |
| GPIO_OUT_OD      | Open Drain Output                     |

**Table 30. GPIO pin modes (continued)**

| Pin mode    | Description                          |
|-------------|--------------------------------------|
| GPIO_OUT_PP | Push-Pull Output                     |
| GPIO_AF_OD  | Open Drain Output Alternate-Function |
| GPIO_AF_PP  | Push-Pull Output Alternate-Function  |

### 3.5.2 Common parameter values

#### Port pin values

[Table 31](#) shows the *Port\_Pins* parameter value for each pin.

**Table 31. Port\_Pins parameter value**

| Port_Pin value | Corresponding pin  |
|----------------|--------------------|
| 0x0001         | Port pin number 0  |
| 0x0002         | Port pin number 1  |
| 0x0004         | Port pin number 2  |
| 0x0008         | Port pin number 3  |
| 0x0010         | Port pin number 4  |
| 0x0020         | Port pin number 5  |
| 0x0040         | Port pin number 6  |
| 0x0080         | Port pin number 7  |
| 0x0100         | Port pin number 8  |
| 0x0200         | Port pin number 9  |
| 0x0400         | Port pin number 10 |
| 0x0800         | Port pin number 11 |
| 0x1000         | Port pin number 12 |
| 0x2000         | Port pin number 13 |
| 0x4000         | Port pin number 14 |
| 0x8000         | Port pin number 15 |

#### GPIOx values

[Table 32](#) shows the allowed values of *GPIOx* variable.

**Table 32. GPIOx values**

| GPIOx | Description             |
|-------|-------------------------|
| GPIO0 | To select <i>GPIO 0</i> |
| GPIO1 | To select <i>GPIO 1</i> |
| GPIO2 | To select <i>GPIO 2</i> |

### 3.5.3 Firmware library functions

[Table 33](#) enumerates the different functions of the *GPIO* library.

**Table 33. GPIO library functions**

| Function Name                  | Description  |
|--------------------------------|--|
| <a href="#">GPIO_Config</a>    | Configures the selected GPIO I/O pins according to the input passed in parameter     |
| <a href="#">GPIO_BitRead</a>   | Returns a port pin value depending on the input passed parameters                    |
| <a href="#">GPIO_ByteRead</a>  | Reads the specified data port byte (upper or lower eight bits) and returns its value |
| <a href="#">GPIO_WordRead</a>  | Reads the specified data port and returns its value                                  |
| <a href="#">GPIO_BitWrite</a>  | Sets or clears the selected data port bit  |
| <a href="#">GPIO_ByteWrite</a> | Writes a byte value into the selected port register byte (upper or lower eight bits) |
| <a href="#">GPIO_WordWrite</a> | Writes a word value into the selected port register                                  |

#### GPIO\_Config

| Function Name          | GPIO_Config   |
|------------------------|---|
| Function Prototype     | <code>void GPIO_Config (GPIO_TypeDef *GPIOx, u16 Port_Pins, GpioPinMode_TypeDef GPIO_Mode);</code>  |
| Behavior Description   | Configures the selected GPIO I/O pins according to the input passed parameters. Refer to the datasheet for more details on input modes, output modes and port names.  |
| Input Parameter 1      | <i>GPIOx</i> : selects the port to be configured. <i>x</i> can be 0, 1 or 2. Refer to <a href="#">GPIOx values on page 68</a> for details on the allowed values of this parameter.                                  |
| Input Parameter 2      | <i>Port_Pins</i> : this parameter specifies the port pin placement. You can select more than one pin. Refer to <a href="#">Port pin values on page 68</a> for more details on the allowed values of this parameter. |
| Input Parameter 3      | <i>GPIO_Mode</i> : specifies the pin mode. Refer to <a href="#">GPIO pin mode enumeration on page 67</a> for more information on the allowed values of this parameter.  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

- Note:
- 1 When you change a port pin mode, the other port pins modes stay unchanged.
  - 2 You can configure more than one pin of the same GPIO peripheral. It is not possible to configure pins of more than one GPIO peripheral at the same call of *GPIO\_Config* function. Moreover it is not possible to configure more than one pin of the same GPIO peripheral in more than one mode by calling once the *GPIO\_Config* function.

**Example:**

This example illustrates how to configure *GPIO0*, *GPIO1* and *GPIO2* port pins.

```

{
  #define GPIO_PIN4 0x04
  #define GPIO_PIN5 0x05
  /*Configure the GPIO0 pin 0 as Open Drain Output */
  GPIO_Config (GPIO0, 0x0001, GPIO_OUT_OD);

  /*Configure the GPIO0 pins 4 and 5 as a Push-Pull Output */
  GPIO_Config (GPIO0, (0x0001<<GPIO_PIN4) | (0x0001<<GPIO_PIN5),
  GPIO_OUT_PP);

  /*Configure the GPIO0 pin 15 as standard TTL input */
  GPIO_Config (GPIO0, 0x8000, GPIO_IN_TRI_TTL);

  /*Configure the GPIO1 pins 0, 1, 2 and 3 as High impedance
  Analog Input */
  GPIO_Config (GPIO1, 0x000F, GPIO_HI_AIN_TRI);

  /* Configure all the GPIO2 pins as Push-Pull Output */
  GPIO_Config (GPIO2, 0xFFFF, GPIO_OUT_PP);
}

```

**GPIO\_BitRead**

| Function Name          | GPIO_BitRead  |
|------------------------|---|
| Function Prototype     | u8 GPIO_BitRead (GPIO_TypeDef *GPIOx, u8 Port_Pin);   |
| Behavior Description   | Reads the specified data port bit and returns its value.  |
| Input Parameter 1      | <i>GPIOx</i> : selects the port to be read. <i>x</i> can be 0, 1 or 2.<br>Refer to <a href="#">GPIOx values on page 68</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | Port_Pin: Specifies the pin to see the value.<br>Refer to <a href="#">Port pin values on page 68</a> for more details on this parameter.  |
| Output Parameter       | None  |
| Return Value           | The selected port pin value.<br>It may be 0x00 if the port pin is reset or 0x01 if the port pin is set.   |
| Required preconditions | The port pin must be configured as <i>GPIO_IN_TRI_TTL</i> or as <i>GPIO_IN_TRI_CMOS</i> mode.   |
| Called Functions       | None  |
| See also               | <a href="#">GPIO_ByteRead on page 71</a> ,<br><a href="#">GPIO_WordRead on page 72</a>  |

**Example:**

In This example, the values of *GPIO0* port pin 0, *GPIO1* port pin 2 and *GPIO2* port pin 15 are read.

```

{
  vu8 bValue;

  /* Get the P0.0 bit value */
  bValue = GPIO_BitRead (GPIO0,0);

  /* Get the P1.2 bit value */
  bValue = GPIO_BitRead (GPIO1, 2);

  /* Get the P2.15 bit value */
  bValue = GPIO_BitRead (GPIO2, 15);
}

```

**GPIO\_ByteRead**

| Function Name          | GPIO_ByteRead   |
|------------------------|---|
| Function Prototype     | u8 GPIO_ByteRead (GPIO_TypeDef *GPIOx, u8 Port_Byte);   |
| Behavior Description   | Reads the specified data port byte and returns its value.   |
| Input Parameter 1      | <i>GPIOx</i> : selects the port to be configured. x can be 0, 1 or 2.<br>Refer to <a href="#">GPIOx values on page 68</a> for details on the allowed values of this parameter.  |
| Input Parameter 2      | <i>Port_Byte</i> : specifies which byte to be read (upper or lower eight bits) . It must be one of the following: <ul style="list-style-type: none"> <li>– <i>GPIO_MSB</i> corresponds to the upper byte</li> <li>– <i>GPIO_LSB</i> corresponds to the lower byte.</li> </ul> |
| Output Parameter       | None  |
| Return Value           | The selected port byte value  |
| Required preconditions | The port pin must be configured as <i>GPIO_IN_TRI_TTL</i> or as <i>GPIO_IN_TRI_CMOS</i> mode.   |
| Called Functions       | None  |
| See also               | <a href="#">GPIO_BitRead on page 70</a> ,<br><a href="#">GPIO_WordRead on page 72</a>   |

**Example:**

In This example *GPIO0* port lower byte and *GPIO1* port upper byte values are read.

```

{
  vu8 bValue;

  /* Get P0.[0:7] bits value */
  bValue = GPIO_ByteRead (GPIO0, GPIO_LSB);

  /* Get P1.[8:15] bits value */
  bValue = GPIO_ByteRead (GPIO1, GPIO_MSB);
}

```

**GPIO\_WordRead**

| Function Name          | GPIO_WordRead   |
|------------------------|---|
| Function Prototype     | <code>u16 GPIO_WordRead (GPIO_TypeDef *GPIOx);</code>   |
| Behavior Description   | Reads the value of the specified <i>GPIOx PD</i> data port register and returns its value.  |
| Input Parameter        | <i>GPIOx</i> : selects the port to be read. <i>x</i> can be 0, 1 or 2.<br>Refer to <a href="#">GPIOx values on page 68</a> for details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The specified port data value   |
| Required preconditions | The port pin must be configured as <i>IN_TRI_TTL</i> or as <i>IN_TRI_CMOS</i> mode.   |
| Called Functions       | None  |
| See also               | <a href="#">GPIO_BitRead on page 70</a> ,<br><a href="#">GPIO_ByteRead on page 71</a>   |

**Example:**

This example illustrates how to read the *GPIO0* port pin values.

```

{
    u16 wValue;

    /* Get all P0 pin value */
    wValue = GPIO_WordRead (GPIO0);
}

```



**GPIO\_BitWrite**

| Function Name          | GPIO_BitWrite   |
|------------------------|---|
| Function Prototype     | <code>void GPIO_BitWrite (GPIO_TypeDef *GPIOx,<br/>u8 Port_Pin, u8 Bit_val);</code>   |
| Behavior Description   | Sets or clears the selected data port bit   |
| Input Parameter 1      | <i>GPIOx</i> : selects the port. <i>x</i> can be 0, 1 or 2. Refer to <a href="#">GPIOx values on page 68</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Port_Pin</i> : Specifies the pin to set or clear the value.<br>Refer to <a href="#">Port pin values on page 68</a> for more details on this parameter.         |
| Input Parameter 3      | <i>Bit_Val</i> : the value to be written<br>0: clear the port pin<br>1: set the port pin  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | The port pin must be configured as <i>OUT_OD</i> or as <i>OUT_PP</i> .  |
| Called Functions       | None  |
| See also               | <a href="#">GPIO_ByteWrite on page 74</a> ,<br><a href="#">GPIO_WordWrite on page 75</a>  |

- Note:**
- 1 When a port pin value is changed, the other port pins values stay unchanged.
  - 2 The Input Pull Up/Pull Down Configuration may change according to the related input pin value.

**Example:**

This example shows how to clear the *GPIO* port pin 0 and sets the *GPIO1* port pin 15.

```
{
  /* Clear the P0.0 bits */
  GPIO_BitWrite(GPIO0, 0, 0);

  /* Set the P1.15 bits */
  GPIO_BitWrite(GPIO1, 15, 1);
}
```

**GPIO\_ByteWrite**

| Function Name          | GPIO_ByteWrite   |
|------------------------|--|
| Function Prototype     | <code>void GPIO_ByteWrite (GPIO_TypeDef *GPIOx,<br/>u8 Port_Byte, u8 Port_Val);</code>   |
| Behavior Description   | Write byte value to the selected GPIOx PD register.  |
| Input Parameter 1      | <i>GPIOx</i> : selects the port. x can be 0, 1 or 2.<br>Refer to <a href="#">GPIOx values on page 68</a> for details on the allowed values of this parameter.                                  |
| Input Parameter 2      | <i>Port_Byte</i> : specifies which byte to be written. It must be one of the following:<br>– <i>GPIO_MSB</i> corresponds to the upper byte<br>– <i>GPIO_LSB</i> corresponds to the lower byte. |
| Input Parameter 3      | <i>Port_Val</i> : the value of the byte to be written.   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The port pin must be configured as <i>OUT_OD</i> or as <i>OUT_PP</i> .   |
| Called Functions       | None   |
| See also               | <a href="#">GPIO_BitWrite on page 73</a> ,<br><a href="#">GPIO_WordWrite on page 75</a>  |

- Note:**
- 1 When a port byte value is changed, the other port byte value stay unchanged.
  - 2 The Input Pull Up/Pull Down Configuration may change according to the related input pin value.

**Example:**

This example shows how to set all the *GPIO0[0:7]* port pins and clear all the *GPIO1[8:15]* port pins. the *GPIO0[8:15]* and *GPIO1[0:7]* remain unchanged.

```
{
  /*Set all the GPIO0[0:7] port pins. */
  GPIO_ByteWrite(GPIO0, GPIO_LSB, 0xFF);

  /* Clear all the GPI1[8:15] port pins */
  GPIO_ByteWrite(GPIO1, GPIO_MSB, 0x00);
}
```

## GPIO\_WordWrite

| Function Name          | GPIO_WordWrite   |
|------------------------|--|
| Function Prototype     | <code>void GPIO_WordWrite (GPIO_TypeDef *GPIOx, u16 Port_Val);</code>  |
| Behavior Description   | Writes a value in to the selected data port register.  |
| Input Parameter 1      | <i>GPIOx</i> : selects the port. <i>x</i> can be 0, 1 or 2.<br>Refer to <a href="#">GPIOx values on page 68</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Port_Val</i> : holds the word value to be written to the selected data port.  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The port pin must be configured as <i>GPIO_OUT_OD</i> or as <i>GPIO_OUT_PP</i> .   |
| Called Functions       | None   |
| See also               | <a href="#">GPIO_BitWrite on page 73</a> ,<br><a href="#">GPIO_ByteWrite on page 74</a>  |

### Example:

This example sets all the *GPIO0* port pins.

```
{
  /*Set all the GPIO0 port pins */
  GPIO_WordWrite(GPIO0, 0xFFFF);
}
```

## 3.6 External Interrupts (XTI)

The *XTI* driver may be used for several purposes, such as firmware IRQ generation, enabling and disabling interrupt lines, selecting the edge sensitivity, interrupt or Wake-up mode.

The first section describes the data structure used in the *XTI* firmware library. The second section presents the *XTI* firmware library functions.

### 3.6.1 Data structures

#### XTI register structure

The *XTI* peripheral register structure *XTI\_TypeDef* is defined in the file *71x\_map.h* as follows:

```
typedef volatile struct
{
  vu8  SR;
  u8   EMPTY1[7];
  vu8  CTRL;
  u8   EMPTY2[3];
  vu8  MRH;
```

```

u8  EMPTY3 [3] ;
vu8 MRL;
u8  EMPTY4 [3] ;
vu8 TRH;
u8  EMPTY5 [3] ;
vu8 TRL;
u8  EMPTY6 [3] ;
vu8 PRH;
u8  EMPTY7 [3] ;
vu8 PRL;
} XTI_TypeDef;

```

[Table 34](#) describes the *XTI* registers.

**Table 34. XTI registers**

| Register | Description                             |
|----------|---|
| SR       | Wake-Up Software Interrupt Register Low |
| CTRL     | Wake-Up Control Register                |
| MRH      | Wake-Up Mask Register High              |
| MRL      | Wake-Up Mask Register Low               |
| TRH      | Wake-Up Trigger Register High           |
| TRL      | Wake-Up Trigger Register Low            |
| PRH      | Wake-Up Pending Register High           |
| PRL      | Wake-Up Pending Register Low            |

*XTI* peripheral is declared in the *71x\_map.h* file as follows

```

/* APB2 Base Address definition */
#define APB2_BASE 0xE0000000

/* XTI Base Address definition */
#define XTI_BASE      (APB2_BASE + 0x101C)

/* XTI peripheral pointer declaration*/
#ifndef DEBUG
    EXT XTI_TypeDef *XTI;
    ...
#else
    #define XTI ((XTI_TypeDef *) XTI_BASE)
    ...
#endif

```

When debug mode is used, *XTI* pointer is initialized in *71x\_lib.c* file:

```

#ifdef _XTI
    XTI = (XTI_TypeDef *) XTI_BASE;
#endif /* _XTI */

```

In debug mode, *\_XTI* must be defined, in *71x\_conf.h* file, to include the *XTI* library

```

#define _XTI

```

### Wake-up line mode

The following enumeration defines the input wake-up line modes. *XTIMode\_TypeDef* enumeration is defined in *71x\_xti.h* file:

```
typedef enum
{
    XTI_WakeUp = 1,
    XTI_Interrupt,
    XTI_WakeUpInterrupt
} XTIMode_TypeDef;
```

[Table 35](#) describes the *XTI* modes.

**Table 35. XTI modes**

| Pin mode            | Description  |
|---------------------|--|
| XTI_WakeUp          | The interrupt lines are used to Wake Up the system from the STOP mode.   |
| XTI_Interrupt       | The interrupt lines are used to generate an IRQ interrupt on the XTI channel.  |
| XTI_WakeUpInterrupt | Interrupt lines are used to generate an IRQ interrupt on the XTI channel and to wake up the system from the STOP mode. |

### Trigger edge polarity

The following enumeration defines the values of the trigger edge polarity. *XTITriggerEdge\_TypeDef* enumeration is defined in *71x\_xti.h* file:

```
typedef enum
{
    XTI_FallingEdge,
    XTI_RisingEdge
} XTITriggerEdge_TypeDef;
```

[Table 36](#) lists the trigger edge polarity values.

**Table 36. Trigger edge polarity values**

| Trigger Edge Polarity | Description   |
|-----------------------|---|
| XTI_FallingEdge       | The Wake-Up interrupt will be set on the falling edge |
| XTI_RisingEdge        | The Wake-Up interrupt will be set on the rising edge  |

## 3.6.2 Common parameter values

### Lines

[Table 37](#) shows the *Lines* parameter value for each interrupt line.

**Table 37. Lines values**

| Lines value | Corresponding Line               |
|-------------|----------------------------------|
| XTI_Line0   | Line 0: Software interrupt       |
| XTI_Line1   | Line 1: <i>USB</i> wake-up event |
| XTI_Line2   | Line 2: External interrupt       |

**Table 37. Lines values (continued)**

| Lines value | Corresponding Line   |
|-------------|--|
| XTI_Line3   | Line 3: External interrupt                                 |
| XTI_Line4   | Line 4: External interrupt                                 |
| XTI_Line5   | Line 5: External interrupt                                 |
| XTI_Line6   | Line 6: CAN module receive pin                             |
| XTI_Line7   | Line 7: HDLC clock or I2C0 clock                           |
| XTI_Line8   | Line 8: HDLC receive pin or I2C0 Data                      |
| XTI_Line9   | Line 9: BSPI0 slave input data or UART3 receive data input |
| XTI_Line10  | Line 10: BSPI0 slave input serial clock or I2C1 clock      |
| XTI_Line11  | Line 11: BSPI1 Slave input serial clock                    |
| XTI_Line12  | Line 12: UART0 receive data input                          |
| XTI_Line13  | Line 13: UART1 receive data input                          |
| XTI_Line14  | Line 14: UART2 receive data input                          |
| XTI_Line15  | Line 15: WakeUp pin or RTC ALARM                           |

### 3.6.3 Firmware library functions

[Table 38](#) enumerates the different functions of the XTI library.

**Table 38. XTI library functions**

| Function Name                          | Description  |
|--|--|
| <a href="#">XTI_Init</a>               | Initializes the XTI to the default configuration             |
| <a href="#">XTI_ModeConfig</a>         | Enables or disables the Interrupt and Wake-Up modes          |
| <a href="#">XTI_LineModeConfig</a>     | Selects the line trigger edge polarity                       |
| <a href="#">XTI_LineConfig</a>         | Enables or disables the selected line interrupt              |
| <a href="#">XTI_InterruptLineValue</a> | Returns the wake-up line number that generates the interrupt |
| <a href="#">XTI_PendingBitClear</a>    | Clears the XTI pending bit                                   |
| <a href="#">XTI_SWIRQGenerate</a>      | Generates a software IRQ interrupt on the XTI channel        |

## XTI\_Init

| Function Name          | XTI_Init  |
|------------------------|---|
| Function Prototype     | <code>void XTI_Init(void);</code>   |
| Behavior Description   | This routine is used to initialize the XTI cell: <ul style="list-style-type: none"><li>– All Wake-Up Lines are disabled</li><li>– Interrupt disabled</li><li>– Walk-up mode disabled</li><li>– Set on the falling edge of the input wake-up line</li><li>– All pending bits are cleared</li></ul> |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to initialize the *XTI*.

```
{  
...  
/* Initialize the XTI */  
XTI_Init();  
...  
}
```

**XTI\_ModeConfig**

| Function Name          | XTI_ModeConfig  |
|------------------------|---|
| Function Prototype     | void XTI_ModeConfig(XTIMode_TypeDef Mode, FunctionalState NewState);  |
| Behavior Description   | This function is used to enable or disable the interrupt and the wake-up mode of the specified input line.  |
| Input Parameter 1      | <i>Mode</i> : this parameter specifies the input line.<br>Refer to <a href="#">Wake-up line mode on page 77</a> for more details on the allowed values of this parameter. |
| Input Parameter 2      | <i>NewState</i> : Used to enable or disable the selected mode.<br>– <i>ENABLE</i> : enables the selected mode<br>– <i>DISABLE</i> : disables the selected mode.           |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable and disable Interrupt and Wake-up modes.

```

{
  /* Disable the Interrupt and WakeUp modes */
  XTI_ModeConfig(XTI_WakeUpInterrupt, DISABLE)
  /* Enable Interrupt mode */
  XTI_ModeConfig(XTI_Interrupt, ENABLE)
}

```



## XTI\_LineModeConfig

| Function Name          | XTI_LineModeConfig  |
|------------------------|---|
| Function Prototype     | <code>void XTI_LineModeConfig(u16 Lines, XTITriggerEdge_TypeDef TriggerEdge);</code>  |
| Behavior Description   | This routine is used to configure the trigger edge.   |
| Input Parameter 1      | <i>Lines</i> : this parameter specifies the input wake-up line. You can select more than one line, by logically OR'ing them.<br>Refer to <a href="#">Section 3.6.2: Common parameter values on page 77</a> for details on the allowed values of <i>Lines</i> parameter. |
| Input Parameter 2      | <i>TriggerEdge</i> : specifies the trigger edge polarity of the specified wake-up lines. Refer to <a href="#">Trigger edge polarity on page 77</a> for more details on the allowed values of this parameter.  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example shows how to set on the rising edge of the input wake-up line 3 and set on the falling edge of the input wake-up lines 1 and 2.

```
{
  /* Configure the line 3 edge */
  XTI_LineModeConfig(XTI_Line3, XTI_RisingEdge);
  /* Configure the lines 2 and 1 edge */
  XTI_LineModeConfig(XTI_Line2|XTI_Line1, XTI_FallingEdge);
}
```

## XTI\_LineConfig

| Function Name          | XTI_LineConfig   |
|------------------------|--|
| Function Prototype     | <code>void XTI_LineConfig(u16 Lines, FunctionalState NewState);</code>   |
| Behavior Description   | This routine is used to enable and disable the interrupts lines.   |
| Input Parameter 1      | <i>Lines</i> : specifies the lines to be configured. You can select more than one line, by logically OR'ing them.<br>Refer to <a href="#">Section 3.6.2: Common parameter values on page 77</a> for details on the allowed values of <i>Lines</i> parameter. |
| Input Parameter 2      | <i>NewStatus</i> : the input line interrupt new status to be set.<br>– <i>ENABLE</i> : to enable the input line interrupt<br>– <i>DISABLE</i> : to disable the input line interrupt  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example shows how to enable the line 2 and 1 Wake-Up interrupts and disable the line 3 Wake-Up interrupts.

```
{
  XTI_LineConfig(XTI_Line3, DISABLE);
  XTI_LineConfig(XTI_Line2 | XTI_Line1, ENABLE);
}
```

## XTI\_InterruptLineValue

| Function Name          | XTI_InterruptLineValue  |
|------------------------|---|
| Function Prototype     | <code>u16 XTI_InterruptLineValue(void);</code>                      |
| Behavior Description   | Gets and returns the input line number that generates an Interrupt. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The line number that generates the interrupt.                       |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example shows how to get the line number that generates the Wake-Up interrupt.

```
void XTI_IRQHandler(void)
{
  u16 wInterruptLine;
  // Read the line number that generates the interrupt.
  wInterruptLine = XTI_InterruptLineValue();
}
```

## XTI\_PendingBitClear

| Function Name          | XTI_PendingBitClear   |
|------------------------|---|
| Function Prototype     | <code>void XTI_PendingBitClear(u16 Lines);</code>   |
| Behavior Description   | This routine is used to clear the XTI interrupt pending bits.   |
| Input Parameter        | <i>Lines</i> : specifies the input wake-up lines interrupt the input wake-up lines interrupt. You can select more than one line by logically OR'ing them. Refer to <a href="#">Section 3.6.2: Common parameter values on page 77</a> for details on the allowed values of <i>Lines</i> parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to clear the pending bits in the XTI.

```
void XTI_IRQHandler(void)
{
    /* Read the line number that caused the interrupt, then
       clear the pending bit. */
    XTI_PendingBitClear(XTI_InterruptLineValue());
}
```

## XTI\_SWIRQGenerate

| Function Name          | XTI_SWIRQGenerate  |
|------------------------|--|
| Function Prototype     | <code>void XTI_SWIRQGenerate(void);</code>                 |
| Behavior Description   | This routine is used to generate a software IRQ interrupt. |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example generates an IRQ interrupt in the XTI channel.

```
int main(void)
{
    /*Enable interrupts in the XTI */
    XTI_ModeConfig(Interrupt, ENABLE);
    /*Configure the XTI channel priority level */
    EIC_IRQChannelPriorityConfig(XTI_IRQChannel, 1);
    /* Enable XTI interrupts */
    EIC_IRQChannelConfig(XTI_IRQChannel, ENABLE);
    /* Enable IRQ interrupts */
}
```

```

EIC_IRQConfig(ENABLE);
/* Generate a software IRQ interrupt */
XTI_SWIRQGenerate();
}

```

## 3.7 Real time clock (RTC)

The *RTC* driver may be used for a variety of purposes, including real time clock management and precise timing operations.

The first section describes the data structures used in the *RTC* firmware library. The second section presents the *RTC* firmware library functions.

### 3.7.1 Data structures

#### RTC register structure

The *RTC* register structure *RTC\_TypeDef* is defined in the *71x\_map.h* file as follows:

```

typedef volatile struct
{
    vu16 CRH;
    u16 EMPTY1;
    vu16 CRL;
    u16 EMPTY2;
    vu16 PRLH;
    u16 EMPTY3;
    vu16 PRL;
    u16 EMPTY4;
    vu16 DIVH;
    u16 EMPTY5;
    vu16 DIVL;
    u16 EMPTY6;
    vu16 CNTH;
    u16 EMPTY7;
    vu16 CNTL;
    u16 EMPTY8;
    vu16 ALRH;
    u16 EMPTY9;
    vu16 ALRL;
    u16 EMPTY10;
} RTC_TypeDef;

```

[Table 39](#) describes the *RTC* structure fields.

**Table 39. RTC structure fields**

| Register | Description                        |
|----------|------------------------------------|
| CRH      | Control High register              |
| CRL      | Control Low register               |
| PRLH     | Prescaler load value High register |
| PRL      | Prescaler load value Low register  |

**Table 39. RTC structure fields (continued)**

| Register | Description                           |
|----------|---------------------------------------|
| DIVH     | Prescaler Divider Value High register |
| DIVL     | Prescaler Divider Value Low register  |
| CNTH     | Counter High register                 |
| CNTL     | Counter Low register                  |
| ALRH     | Alarm High register                   |
| ALRL     | Alarm Low register                    |

The *RTC* peripheral is declared in the same file:

```

...
#define RTC_BASE          0xE000D000
...
#ifndef DEBUG
...
#ifdef _RTC
#define RTC                ((RTC_TypeDef *)RTC_BASE)
#endif /*RTC*/
...
#else /* DEBUG */
...
#ifdef _RTC
EXT RTC_TypeDef          *RTC;
#endif /*RTC*/
...
#endif

```

When debug mode is used, *RTC* pointer is initialized in the file *71x\_lib.c*:

```

#ifdef _RTC
RTC = (RTC_TypeDef *)RTC_BASE;
#endif

```

In debug mode, *\_RTC* must be defined, in the file *71x\_conf.h*, to access the peripheral registers as follows:

```

#define _RTC

```

### RTC flags

The following enumeration defines the different *RTC* flags. The *RTC\_FLAGS* enumeration is defined in the file *71x\_rtc.h*:

```

typedef enum
{
    RTC_GIR    = 0x0008,
    RTC_OWIR   = 0x0004,
    RTC_AIR    = 0x0002,
    RTC_SIR    = 0x0001
} RTC_FLAGS;

```

[Table 40](#) describes the states of *RTC* flags.

**Table 40. RTC flags**

| RTC Flags | Description                       |
|-----------|-----------------------------------|
| RTC_GIR   | RTC Global Interrupt Request flag |
| RTC_OWIR  | RTC overflow interrupt request    |
| RTC_AIR   | RTC Alarm interrupt request       |
| RTC_SIR   | RTC Second interrupt request      |

### RTC interrupts

The following enumeration defines the real time clock interrupts. The *RTC\_IT* enumeration is declared in the file *71x\_rtc.h*:

```
typedef enum
{
    RTC_GIT = 0x0008,
    RTC_OWIT = 0x0004,
    RTC_AIT = 0x0002,
    RTC_SIT = 0x0001,
} RTC_IT;
```

[Table 41](#) describes the *RTC* interrupts.

**Table 41. RTC interrupts**

| Constant Name | Description                |
|---------------|----------------------------|
| RTC_GIT       | The RTC Global Interrupt   |
| RTC_OWIT      | The RTC overflow interrupt |
| RTC_AIT       | The RTC Alarm interrupt    |
| RTC_SIT       | The RTC Second interrupt   |

## 3.7.2 Firmware library functions

[Table 42](#) enumerates the different functions of the *RTC* library.

**Table 42. RTC library functions**

| Function Name                       | Description   |
|-------------------------------------|---|
| <a href="#">RTC_PrescalerValue</a>  | Reads and returns the prescaler value                         |
| <a href="#">RTC_PrescalerConfig</a> | Configures the prescaler                                      |
| <a href="#">RTC_CounterValue</a>    | Reads and returns the counter value                           |
| <a href="#">RTC_CounterConfig</a>   | Update the RTC counter value                                  |
| <a href="#">RTC_AlarmValue</a>      | Reads and returns the Alarm time value                        |
| <a href="#">RTC_AlarmConfig</a>     | Configures the alarm time value                               |
| <a href="#">RTC_FlagStatus</a>      | Reads and returns the status of the specified <i>RTC</i> flag |
| <a href="#">RTC_FlagClear</a>       | Clears the specified <i>RTC</i> flag passed in parameter      |
| <a href="#">RTC_ITConfig</a>        | Enables / Disables the specified <i>RTC</i> interrupt         |

**Table 42. RTC library functions (continued)**

| Function Name                       | Description   |
|-------------------------------------|---|
| <a href="#">RTC_ITStatus</a>        | Checks the status of the specified <i>RTC</i> interrupt |
| <a href="#">RTC_ITClear</a>         | Clears the specified <i>RTC</i> interrupt               |
| <a href="#">RTC_EnterCfgMode</a>    | Enters the <i>RTC</i> configuration mode                |
| <a href="#">RTC_ExitCfgMode</a>     | Exit from the configuration mode                        |
| <a href="#">RTC_WaitForLastTask</a> | Waits for last task to be finished                      |

**RTC\_PrescalerValue**

| Function Name          | RTC_PrescalerValue                                |
|------------------------|---|
| Function Prototype     | <code>u32 RTC_PrescalerValue (void)</code>        |
| Behavior Description   | This routine gets the <i>RTC</i> prescaler value. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The current <i>RTC</i> prescaler value            |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to get the current *RTC* prescaler value:

```

{
    u32 wCurrent_Prescaler;
    ...
    wCurrent_Prescaler = RTC_PrescalerValue ();
    ...
}

```

**RTC\_PrescalerConfig**

| Function Name          | RTC_PrescalerConfig  |
|------------------------|--|
| Function Prototype     | <code>void RTC_PrescalerConfig (u32 Xprescaler);</code>  |
| Behavior Description   | This routine is used to configure the prescaler.   |
| Input Parameter        | <i>Xprescaler</i> : specifies the new <i>RTC</i> prescaler value.  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | <a href="#">RTC_WaitForLastTask</a> , <a href="#">RTC_EnterCfgMode</a> and <a href="#">RTC_ExitCfgMode</a> |

*Note:* The prescaler is a 20 bit value, the *RTC\_PrescalerConfig* truncates any value, passed in parameter, that exceeds 20 bits.

**Example:**

This example illustrates how to configure the *RTC* prescaler value.

```

{
    u32 wNew_Prescaler;
    ...
    /* Reads the previous prescaler value */
    wNew_Prescaler = RTC_PrescalerValue ();
    /* Add 0xFF */
    wNew_Prescaler += 0xFF;
    /* Configures the new prescaler value */
    RTC_PrescalerConfig (wNew_Prescaler);
    ...
}

```

**RTC\_CounterValue**

| Function Name          | RTC_CounterValue  |
|------------------------|---|
| Function Prototype     | u32 RTC_CounterValue (void);                            |
| Behavior Description   | This routine gets the current <i>RTC</i> counter value. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The current <i>RTC</i> counter value                    |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to get the *RTC* counter value.

```

{
    u32 wCounter;
    ...
    wCounter = RTC_CounterValue();
    ...
}

```



## RTC\_CounterConfig

| Function Name          | RTC_CounterConfig   |
|------------------------|---|
| Function Prototype     | <code>void RTC_CounterConfig (void);</code>                             |
| Behavior Description   | This routine is used to update the <i>RTC</i> counter with a new value. |
| Input Parameter        | The new <i>RTC</i> counter value  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | RTC_EnterCfgMode, RTC_WaitForLastTask and RTC_ExitCfgMode               |

### Example:

This example illustrates how to update the *RTC* counter with 0x23F value.

```
{
  ...
  RTC_CounterConfig(0x23F);
  ...
}
```

## RTC\_AlarmValue

| Function Name          | RTC_AlarmValue  |
|------------------------|---|
| Function Prototype     | <code>u32 RTC_AlarmValue (void);</code>               |
| Behavior Description   | This routine gets the current <i>RTC</i> alarm value. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | The current <i>RTC</i> alarm value (u32 format)       |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to read the current *RTC* alarm value.

```
{
  u32 wCurrent_Alarm_Time;
  ...
  wCurrent_Alarm_Time = RTC_AlarmValue ();
  ...
}
```

## RTC\_AlarmConfig

| Function Name          | RTC_AlarmConfig   |
|------------------------|---|
| Function Prototype     | <code>void RTC_AlarmConfig (u32 Xalarm);</code>   |
| Behavior Description   | This routine sets the alarm time value. It updates <i>AHI</i> & <i>ALO</i> register values.         |
| Input Parameter        | <i>Xalarm</i> : specifies the new <i>RTC</i> alarm time value.                                      |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | <code>RTC_WaitForLastTask</code> , <code>RTC_EnterCfgMode</code> and <code>RTC_ExitCfgMode</code> . |

### Example:

This example illustrates how to configure a new alarm time.

```

{
    u32 wAlarm_Time;
    ...
    wAlarm_Time = 0x007FFFFFFF;
    ...
    RTC_AlarmConfig (wAlarm_Time);
    ...
}

```

## RTC\_FlagStatus

| Function Name          | RTC_FlagStatus  |
|------------------------|---|
| Function Prototype     | <code>FlagStatus RTC_FlagStatus (RTC_FLAGS Xflag);</code>   |
| Behavior Description   | This routine is used to check if the specified <i>RTC</i> flag is set or not.   |
| Input Parameter        | <i>Xflag</i> : Designates an <i>RTC</i> flag. Refer to <a href="#">RTC flags on page 85</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The specified flag status:<br>– <i>SET</i> : The corresponding flag is SET<br>– <i>RESET</i> : The corresponding flag is RESET                        |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to check the *RTC* Second interrupt flag.

```

{
    ...
    /* Test if the Second interrupt request is set */
    if (RTC_FlagStatus (RTC_SIR))
    {
        ...
    }
}

```

## RTC\_FlagClear

| Function Name          | RTC_FlagClear  |
|------------------------|--|
| Function Prototype     | <code>void RTC_FlagClear (RTC_FLAGS Xflag);</code>   |
| Behavior Description   | This routine is used to clear the specified <i>RTC</i> flag.   |
| Input Parameter        | <i>Xflag</i> : Designates an <i>RTC</i> flag.<br>Refer to <a href="#">RTC flags on page 85</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | RTC_WaitForLastTask, RTC_EnterCfgMode and RTC_ExitCfgMode  |

### Example:

This example illustrates how to clear an *RTC* pending flag.

```
{
  /* Clear the Alarm request pending flag */
  RTC_FlagClear (RTC_AIR);
}
```

## RTC\_ITConfig

| Function Name          | RTC_ITConfig   |
|------------------------|--|
| Function Prototype     | <code>void RTC_ITConfig (RTC_IT Xrtcit, FunctionalState NewState);</code>  |
| Behavior Description   | This routine is used to enable or disable the specified <i>RTC</i> interrupt.  |
| Input Parameter 1      | <i>Xrtcit</i> : Designates an <i>RTC</i> interrupts. Refer to <a href="#">RTC interrupts on page 86</a> for more details on the allowed values of this parameter.                |
| Input Parameter 2      | <i>NewState</i> : designates the new interrupt status.<br>– <i>ENABLE</i> : The corresponding interrupt is enabled<br>– <i>DISABLE</i> : The corresponding interrupt is disabled |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to enable the *RTC* second interrupt.

```
{
  /* Enable the global interrupt */
  RTC_ITConfig (RTC_GIT, ENABLE);
  /* Enable the Second interrupt */
  RTC_ITConfig (RTC_SIT, ENABLE);
}
```

**RTC\_ITStatus**

| Function Name          | RTC_ITStatus  |
|------------------------|---|
| Function Prototype     | FunctionalState RTC_ITStatus (RTC_IT Xrtcit);   |
| Behavior Description   | This routine checks whether the specified <i>RTC</i> interrupt is enabled or not. It reads and returns the <i>CRH</i> flags status.   |
| Input Parameter        | <i>Xrtcit</i> : Designates an <i>RTC</i> interrupts. Refer to <a href="#">RTC interrupts on page 86</a> for more details on the allowed values of this parameter.                                     |
| Output Parameter       | None  |
| Return Value           | The interrupt status:<br><ul style="list-style-type: none"> <li>– <i>ENABLE</i>: the corresponding interrupt is enabled</li> <li>– <i>DISABLE</i>: the corresponding interrupt is disabled</li> </ul> |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to check the alarm interrupt status.

```

{
  /* test if the Alarm interrupt is enabled */
  if RTC_ITStatus (RTC_AIT)
  {
    ...
  }
}

```

**RTC\_ITClear**

| Function Name          | RTC_ITClear   |
|------------------------|---|
| Function Prototype     | void RTC_ITClear (RTC_IT Xrtcit);   |
| Behavior Description   | This routine is used to clear the specified interrupt pending request.  |
| Input Parameter        | <i>Xrtcit</i> : designates an <i>RTC</i> interrupt.<br>Refer to <a href="#">RTC interrupts on page 86</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | RTC_WaitForLastTask, RTC_EnterCfgMode and RTC_ExitCfgMode.  |

**Example:**

This example illustrates how to clear the second interrupt request.

```

{
  ...
  RTC_ITClear (RTC_SIT);
  ...
}

```

**RTC\_EnterCfgMode**

| Function Name          | RTC_EnterCfgMode   |
|------------------------|--|
| Function Prototype     | <code>void RTC_EnterCfgMode(void);</code>                        |
| Behavior Description   | This routine is used to enter the <i>RTC</i> configuration mode. |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | RTC_WaitForLastTask  |

**Example:**

This example illustrates how to enter in configuration mode.

```

{
...
    RTC_EnterCfgMode ();
...
}

```

**RTC\_ExitCfgMode**

| Function Name          | RTC_ExitCfgMode  |
|------------------------|--|
| Function Prototype     | <code>void RTC_ExitCfgMode(void);</code>                             |
| Behavior Description   | This routine is used to exit from the <i>RTC</i> configuration mode. |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | RTC_WaitForLastTask  |

**Example:**

This example illustrates how to exit from configuration mode.

```

{
...
    RTC_ExitCfgMode ();
...
}

```

**RTC\_WaitForLastTask**

| Function Name          | RTC_WaitForLastTask  |
|------------------------|--|
| Function Prototype     | <code>void RTC_WaitForLastTask(void);</code>                                 |
| Behavior Description   | This routine waits for the task to be completed on the <i>RTC</i> registers. |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to use the *RTC\_WaitForLastTask* function.

```
{
...
RTC_WaitForLastTask();
...
}
```

**3.8 Watchdog timer (WDG)**

The *WDG* driver may be used for a variety of purposes, including timing operations and watchdog functions.

The first section describes the data structures used in the *WDG* firmware library. The second section presents the *WDG* firmware library functions.

**3.8.1 Data structures****WDG register structure**

The *WDG* register structure *WDG\_TypeDef* is defined in the *71x\_map.h* file as follows:

```
typedef volatile struct
{
    vu16 CR;
    u16 EMPTY1;
    vu16 PR;
    u16 EMPTY2;
    vu16 VR;
    u16 EMPTY3;
    vu16 CNT;
    u16 EMPTY4;
    vu16 SR;
    u16 EMPTY5;
    vu16 MR;
    u16 EMPTY6;
    vu16 KR;
    u16 EMPTY7
} WDG_TypeDef;
```

[Table 43](#) describes the *WDG* structure fields.

**Table 43. WDG structure fields**

| Register | Description             |
|----------|-------------------------|
| CR       | Control register        |
| PR       | Prescaler register      |
| VR       | Pre-load Value register |
| CNT      | Counter register        |
| SR       | Status Register         |
| MR       | Mask Register           |
| KR       | Key Register            |

The *WDG* peripherals are declared in the same file:

```

...
#define WDG_BASE          (APB2_BASE + 0xE000)
...
#ifndef DEBUG
...
#ifdef _WDG
    #define WDG              ((WDG_TypeDef *)WDG_BASE)
#endif /*WDG*/
...
#else /* DEBUG */
...
#ifdef _WDG
    EXT WDG_TypeDef          *WDG;
#endif /*WDG*/
...
#endif

```

When debug mode is used, *WDG* pointers are initialized in the file *71x\_lib.c*:

```

#ifdef _WDG
    WDG = (WDG_TypeDef *)WDG_BASE;
#endif

```

In debug mode, *\_WDG* must be defined, in the file *71x\_conf.h*, to access the peripheral registers as follows:

```
#define _WDG
```

Some *RCCU* functions are called, *\_RCCU* must be defined, in *71x\_conf.h* file, to make the *RCCU* functions accessible:

```
#define _RCCU
```

### 3.8.2 Firmware library functions

Table 44 enumerates the different functions of the WDG library.

**Table 44. WDG library functions**

| Function Name                         | Description  |
|---------------------------------------|--|
| <a href="#">WDG_Enable</a>            | Enable Watchdog Mode   |
| <a href="#">WDG_CntRefresh</a>        | Refresh and update the WDG counter to avoid a system reset                   |
| <a href="#">WDG_PrescalerConfig</a>   | Set the counter prescaler value. Divide the counter clock by (Prescaler + 1) |
| <a href="#">WDG_CntReloadUpdate</a>   | Update the counter pre-load value  |
| <a href="#">WDG_PeriodValueConfig</a> | Set the prescaler and counter reload value based on the time needed          |
| <a href="#">WDG_CntOnOffConfig</a>    | Start or stop the free auto-reload timer to countdown                        |
| <a href="#">WDG_ECITConfig</a>        | Enable or Disable the end of count interrupt                                 |
| <a href="#">WDG_ECFlagClear</a>       | Clear the end of count flag  |
| <a href="#">WDG_ECStatus</a>          | Return the end of count status.  |

#### WDG\_Enable

| Function Name          | WDG_Enable                                    |
|------------------------|---|
| Function Prototype     | <code>void WDG_Enable ( void );</code>        |
| Behavior Description   | This routine is used to enable Watchdog mode. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

#### Example:

This example illustrates how to enable Watchdog mode:

```

{
...
/* Enable the watchdog mode */
WDG_Enable ();
...
}

```



**WDG\_CntRefresh**

| Function Name          | WDG_CntRefresh  |
|------------------------|---|
| Function Prototype     | <code>void WDG_CntRefresh (void);</code>  |
| Behavior Description   | This routine is used to refresh the counter to prevent a reset being generated by the Watchdog. It writes 0xA55A then 0x5AA5 in the key register. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to refresh the watchdog counter.

```
{
...
  WDG_CntRefresh ();
...
}
```

**WDG\_PrescalerConfig**

| Function Name          | WDG_PrescalerConfig   |
|------------------------|---|
| Function Prototype     | <code>void WDG_PrescalerConfig (u8 Prescaler);</code>   |
| Behavior Description   | This routine is used to set the the counter prescaler value. The clock to Timer Counter is divided by (Prescaler + 1) |
| Input Parameter        | <i>Prescaler</i> : specifies the prescaler value (8 bit).   |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to set the prescaler value.

```
{
...
  WDG_PrescalerConfig (0x62);
...
}
```

**WDG\_CntReloadUpdate**

| Function Name          | WDG_CntReloadUpdate  |
|------------------------|--|
| Function Prototype     | <code>void WDG_CntReloadUpdate (u16 PreLoadValue);</code>  |
| Behavior Description   | This routine is used to update the counter pre-load value. |
| Input Parameter        | <i>PreLoadValue</i> specifies the pre-load value (16 bit). |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to configure the *WDG* pre-load value.

```
{
...
void WDG_CntReloadUpdate (0x1234);
...
}
```

**WDG\_PeriodValueConfig**

| Function Name          | WDG_PeriodValueConfig   |
|------------------------|---|
| Function Prototype     | <code>void WDG_PeriodValueConfig (u32 Time);</code>                 |
| Behavior Description   | This routine is used to set the prescaler and counter reload value. |
| Input Parameter        | <i>Time</i> : the amount of time needed, in microseconds.           |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | RCCU_FrequencyValue, FindFactors                                    |

**Example:**

This example illustrates how to set the needed time ( $\mu$ s).

```
{
...
WDG_PeriodValueConfig (2589);
...
}
```

**WDG\_CntOnOffConfig**

| Function Name          | WDG_CntOnOffConfig  |
|------------------------|---|
| Function Prototype     | <code>void WDG_CntOnOffConfig (FunctionalState NewState);</code>                |
| Behavior Description   | This routine is used to start or stop the free auto-reload timer counting down. |
| Input Parameter        | <i>NewState</i> : specifies how the timer has to be started or stopped.         |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the *WDG* in order to start or stop the timer.

```

{
...
/* stop the counter */
WDG_CntOnOffConfig (DISABLE);
/* Start the counter */
WDG_CntOnOffConfig (ENABLE);
...
}

```

**WDG\_ECITConfig**

| Function Name          | WDG_ECITConfig   |
|------------------------|--|
| Function Prototype     | <code>void WDG_ECITConfig (FunctionalState NewState)</code>  |
| Behavior Description   | Enable or Disable the <i>WDG</i> end of count interrupt.   |
| Input Parameter        | <i>NewState</i> : specifies whether the <i>WDG</i> end of count interrupt is enabled or disabled.<br>– ENABLE: enables the <i>WDG</i> IRQ interrupt<br>– DISABLE: disables the <i>WDG</i> IRQ interrupt. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to enable the *WDG* end of count interrupt.

```

{
...

WDG_ECITConfig (ENABLE);
...
}

```

**WDG\_ECFlagClear**

| Function Name          | WDG_ECFlagClear                                      |
|------------------------|--|
| Function Prototype     | void WDG_ECFlagClear (void);                         |
| Behavior Description   | This routine is used to clear the end of count flag. |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how clear the end of count pending bit.

```
{
...
  WDG_ECFlagClear ();
...
}
```

**WDG\_ECStatus**

| Function Name          | WDG_ECStatus              |
|------------------------|---------------------------|
| Function Prototype     | u16 WDG_ECStatus (void);  |
| Behavior Description   | Read the status register. |
| Input Parameter        | None                      |
| Output Parameter       | None                      |
| Return Value           | The end of count status   |
| Required preconditions | None                      |
| Called Functions       | None                      |

**Example:**

This example illustrates how to get the end of count status

```
{
...
  u16 hStatus;
  hStatus = WDG_ECStatus();
...
}
```

## 3.9 Timer (TIM)

The *TIM* driver may be used for a variety of purposes, including timing operations, external wave generation and measurement.

The first section describes the data structures used in the *TIM* firmware library. The second section presents the *TIM* firmware library functions.

### 3.9.1 Data structures

#### TIM register structure

The *TIM* register structure *TIM\_TypeDef* is defined in the *71x\_map.h* file as follows:

```
typedef volatile struct
{
    vu16 ICAR;
    u16  EMPTY1;
    vu16 ICBR;
    u16  EMPTY2;
    vu16 OCAR;
    u16  EMPTY3;
    vu16 OCBR;
    u16  EMPTY4;
    vu16 CNTR;
    u16  EMPTY5;
    vu16 CR1;
    u16  EMPTY6;
    vu16 CR2;
    u16  EMPTY7;
    vu16 SR;
    u16  EMPTY8;
} TIM_TypeDef;
```

[Table 45](#) describes the *TIM* structure fields.

**Table 45. TIM structure fields**

| Register | Description               |
|----------|---------------------------|
| ICAR     | Input capture A register  |
| ICBR     | Input capture B register  |
| OCAR     | Output Compare A register |
| OCBR     | Output Compare B register |
| CNTR     | Counter Register          |
| CR1      | Control Register 1        |
| CR2      | Control Register 2        |
| SR       | Status Register           |

The *TIM* peripherals are declared in the same file:

```
...
#define TIM0_BASE      (APB2_BASE + 0x9000)
#define TIM1_BASE      (APB2_BASE + 0xA000)
```

```

#define TIM2_BASE      (APB2_BASE + 0xB000)
#define TIM3_BASE      (APB2_BASE + 0xC000)
...
#ifndef DEBUG
...
#ifdef _TIM0
#define TIM0            ((TIM_TypeDef *)TIM0_BASE)
#endif /*TIM0*/

#ifdef _TIM1
#define TIM1            ((TIM_TypeDef *)TIM1_BASE)
#endif /*TIM1*/

#ifdef _TIM2
#define TIM2            ((TIM_TypeDef *)TIM2_BASE)
#endif /*TIM2*/

#ifdef _TIM3
#define TIM3            ((TIM_TypeDef *)TIM3_BASE)
#endif /*TIM3*/
...
#else /* DEBUG */
...
#ifdef _TIM0
EXT TIM_TypeDef        *TIM0;
#endif /*TIM0*/

#ifdef _TIM1
EXT TIM_TypeDef        *TIM1;
#endif /*TIM1*/

#ifdef _TIM2
EXT TIM_TypeDef        *TIM2;
#endif /*TIM2*/

#ifdef _TIM3
EXT TIM_TypeDef        *TIM3;
#endif /*TIM3*/
...
#endif

```

When debug mode is used, *TIM* pointers are initialized in the file *71x\_lib.c*:

```

#ifdef _TIM0
TIM0 = (TIM_TypeDef *)TIM0_BASE;
#endif

#ifdef _TIM1
TIM1 = (TIM_TypeDef *)TIM1_BASE;
#endif

#ifdef _TIM2
TIM2 = (TIM_TypeDef *)TIM2_BASE;
#endif

```

```

#ifdef _TIM3
    TIM3 = (TIM_TypeDef *)TIM3_BASE;
#endif
    TIM0 = (TIM_TypeDef *)TIM0_BASE;

```

In debug mode, `_TIMx` must be defined in the file `71x_conf.h` to access the peripheral registers as follows:

```
#define _TIMx (x=0..3)
```

## TIM values

[Table 46](#) enumerates the different values of the TIMx.

**Table 46. TIMx values**

| TIM Value | Description |
|-----------|-------------|
| TIM0      | Timer 0     |
| TIM1      | Timer 1     |
| TIM2      | Timer 2     |
| TIM3      | Timer 3     |

## TIM flags enumeration

The following enumeration defines the different *TIM* flags. The *TIM\_Flags* enumeration is defined in the file `71x_tim.h`:

```

typedef enum
{
    TIM_ICFA = 0x8000,
    TIM_OCFA = 0x4000,
    TIM_TOF  = 0x2000,
    TIM_ICFB = 0x1000,
    TIM_OCFB = 0x0800
} TIM_Flags;

```

[Table 47](#) describes the states of *TIM* flags.

**Table 47. TIM flags**

| TIM Flags | Description                              |
|-----------|--|
| TIM_ICFA  | <i>TIM</i> Input Capture channel A flag  |
| TIM_OCFA  | <i>TIM</i> Output Compare channel A flag |
| TIM_TOF   | <i>TIM</i> Timer Overflow flag           |
| TIM_ICFB  | <i>TIM</i> Input Capture channel B flag  |
| TIM_OCFB  | <i>TIM</i> Output Compare channel B flag |

### Clock sources enumeration

The following enumeration defines the different *TIM* clock sources. The *TIM\_Clocks* enumeration is defined in the file *71x\_tim.h*:

```
typedef enum
{
    TIM_EXTERNAL,
    TIM_INTERNAL
} TIM_Clocks;
```

[Table 48](#) enumerates the clock sources.

**Table 48. TIM clock sources**

| Clock Source | Description  |
|--------------|--|
| TIM_EXTERNAL | The <i>TIM</i> peripheral is clocked by the external clock |
| TIM_INTERNAL | The <i>TIM</i> peripheral is clocked by <i>APB</i> clock   |

### Clock edges enumeration

The following enumeration defines the different *TIM* clock edges. The *TIM\_Clock\_Edges* enumeration is defined in the file *71x\_tim.h*:

```
typedef enum
{
    TIM_RISING,
    TIM_FALLING
} TIM_Clock_Edges;
```

[Table 49](#) enumerates the clock edges.

**Table 49. TIM clock edges**

| clock Edge  | Description                         |
|-------------|-------------------------------------|
| TIM_RISING  | The active edge is the rising edge  |
| TIM_FALLING | The active edge is the falling edge |

### TIM channels enumeration

The following enumeration defines the different *TIM* channels. The *TIM\_Channels* enumeration is defined in the file *71x\_tim.h*:

```
typedef enum
{
    TIM_CHANNEL_A,
    TIM_CHANNEL_B
} TIM_Channels;
```

[Table 50](#) enumerates the *TIM* channels.

**Table 50. TIM channels**

| TIM Channel   | Description |
|---------------|-------------|
| TIM_CHANNEL_A | Channel A   |
| TIM_CHANNEL_B | Channel B   |



## Output compare modes enumeration

The following enumeration defines the Output Compare modes. The *OC\_Modes* enumeration is defined in the file *71x\_tim.h*:

```
typedef enum
{
    TIM_TIMING,
    TIM_WAVE
} TIM_OC_Modes;
```

[Table 51](#) enumerates the output compare modes.

**Table 51. Output compare modes**

| Output Compare Mode | Description  |
|---------------------|--|
| TIM_TIMING          | The output compare mode is used for internal operations                      |
| TIM_WAVE            | The output compare mode is only used for external wave generation operations |

## Logic levels enumeration

The following enumeration defines the different logic levels. The *Logic\_Levels* enumeration is defined in the file *71x\_tim.h*:

```
typedef enum
{
    TIM_HIGH,
    TIM_LOW
} TIM_Logic_Levels;
```

[Table 52](#) enumerates the logic levels.

**Table 52. Logic levels**

| Logic level | Description   |
|-------------|---|
| TIM_HIGH    | The output compare is forced to the high level after the match between the counter and the output compare register. |
| TIM_LOW     | The output compare is forced to the low level after the match between the counter and the output compare register.  |

## Counter operations enumeration

The following enumeration defines the different counter operations. The *CounterOperations* enumeration is defined in the file *71x\_tim.h*:

```
typedef enum
{
    TIM_START,
    TIM_STOP,
    TIM_CLEAR
} TIM_CounterOperations;
```

[Table 53](#) enumerates the counter operations.

**Table 53. Counter operations**

| Counter Operation | Description   |
|-------------------|---|
| TIM_START         | Enables or resumes the counter                          |
| TIM_STOP          | Stops the TIM counter                                   |
| TIM_CLEAR         | Clears the TIM counter value and set its value to FFFCh |

### PWM input parameters structure

The following structure defines the different PWM input parameters. The *PWMI\_parameters* structure is defined in the file *71x\_tim.h*:

```
typedef struct
{
    u16 Pulse,
    u16 Period
} PWMI_parameters;
```

[Table 54](#) defines the PWM input parameters.

**Table 54. PWM input parameters**

| Structure Field | Description                       |
|-----------------|-----------------------------------|
| Pulse           | The pulse of the external signal  |
| Period          | The period of the external signal |

### TIM interrupts

The following section lists the different *TIM* interrupts. Those interrupts are declared in the file *71x\_tim.h*:

```
#define TIM_ICA_IT    0x8000
#define TIM_OCA_IT    0x4000
#define TIM_TO_IT     0x2000
#define TIM_ICB_IT    0x1000
#define TIM_OCB_IT    0x0800
```

[Table 55](#) describes the *TIM* interrupts.

**Table 55. TIM interrupts**

| TIM Interrupts | Description                                   |
|----------------|---|
| TIM_ICA_IT     | <i>TIM</i> Input Capture Channel A interrupt  |
| TIM_OCA_IT     | <i>TIM</i> Output Compare channel A interrupt |
| TIM_TO_IT      | <i>TIM</i> Timer Overflow interrupt           |
| TIM_ICB_IT     | <i>TIM</i> Input Capture Channel A interrupt  |
| TIM_OCB_IT     | <i>TIM</i> Output Compare channel B interrupt |

### 3.9.2 Firmware library functions

Table 56 enumerates the different functions of the *TIM* library.

**Table 56. TIM library functions**

| Function Name                | Description  |
|------------------------------|--|
| <i>TIM_Init</i>              | Initializes the <i>TIM</i> peripheral  |
| <i>TIM_ClockSourceConfig</i> | Configures the <i>TIM</i> clock source   |
| <i>TIM_ClockSourceValue</i>  | Gets the <i>TIM</i> clock source   |
| <i>TIM_PrescalerConfig</i>   | Configures the <i>TIM</i> prescaler Value  |
| <i>TIM_PrescalerValue</i>    | Gets the <i>TIM</i> prescaler Value  |
| <i>TIM_ClockLevelConfig</i>  | Configures the active clock level when the <i>TIM</i> is clocked by an external source |
| <i>TIM_ClockLevelValue</i>   | Gets the active clock level when the <i>TIM</i> is clocked by an external source       |
| <i>TIM_ICAPModeConfig</i>    | Configures the Input capture mode  |
| <i>TIM_ICAPValue</i>         | Gets the Input capture values  |
| <i>TIM_OCMPModeConfig</i>    | Configures the Output compare mode   |
| <i>TIM_OPMModeConfig</i>     | Configures the One pulse mode  |
| <i>TIM_PWMOModeConfig</i>    | Configures the PWM output mode   |
| <i>TIM_PWMIModeConfig</i>    | Configures the PWM input mode  |
| <i>TIM_PWMIValue</i>         | Reads and returns the PWM input parameters   |
| <i>TIM_ITConfig</i>          | Configures the <i>TIM</i> interrupts   |
| <i>TIM_FlagStatus</i>        | Gets the <i>TIM</i> flag Status  |
| <i>TIM_FlagClear</i>         | Clears the <i>TIM</i> pending flag   |
| <i>TIM_CounterConfig</i>     | Configures the <i>TIM</i> counter  |
| <i>TIM_CounterValue</i>      | This routine returns the timer counter value.  |

**TIM\_Init**

| Function Name          | TIM_Init   |
|------------------------|--|
| Function Prototype     | <code>void TIM_Init (TIM_TypeDef *TIMx);</code>  |
| Behavior Description   | This routine is used to Initialize the TIM peripheral registers to their default values.   |
| Input Parameter        | <i>TIMx</i> specifies the <i>TIM</i> to be initialized.<br>Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to initialize all the *TIM* peripherals:

```

{
...
/* Initialize the TIM0 peripheral */
TIM_Init (TIM0);
/* Initialize the TIM1 peripheral */
TIM_Init (TIM1);
/* Initialize the TIM2 peripheral */
TIM_Init (TIM2);
/*Initialize the TIM3 peripheral */
TIM_Init (TIM3);
...
}

```

## TIM\_ClockSourceConfig

| Function Name          | TIM_ClockSourceConfig   |
|------------------------|---|
| Function Prototype     | <code>void TIM_ClockSourceConfig (TIM_TypeDef *TIMx, TIM_Clocks Xclock);</code>   |
| Behavior Description   | This routine configures the source clock of the <i>TIM</i> peripheral.  |
| Input Parameter 1      | <i>TIMx</i> : specifies the <i>TIM</i> to be configured. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.            |
| Input Parameter 2      | <i>Xclock</i> specifies the <i>TIM</i> source clock. Refer to <a href="#">Clock sources enumeration on page 104</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to configure the *TIM0* timer to be clocked by an external source.

```
{
...
TIM_ClockSourceConfig (TIM0, TIM_EXTERNAL);
...
}
```

## TIM\_ClockSourceValue

| Function Name          | TIM_ClockSourceValue  |
|------------------------|---|
| Function Prototype     | <code>TIM_Clocks TIM_ClockSourceValue (TIM_TypeDef *TIMx);</code>   |
| Behavior Description   | This routine gets the source clock of the <i>TIM</i> peripheral.  |
| Input Parameter        | <i>TIMx</i> : specifies the <i>TIM</i> to check its source clock. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The <i>TIM</i> source clock. Refer to <a href="#">Clock sources enumeration on page 104</a> for more details on the allowed values of this parameter.                       |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to get the *TIM0* timer source clock.

```
{
TIM_Clocks CurrentTIMClock;
...
CurrentTIMClock = TIM_ClockSourceValue (TIM0);
...
}
```

**TIM\_PrescalerConfig**

| Function Name          | TIM_PrescalerConfig  |
|------------------------|--|
| Function Prototype     | <code>void TIM_PrescalerConfig (TIM_TypeDef *TIMx, u8 Xprescaler);</code>  |
| Behavior Description   | This routine configures the <i>TIM</i> prescaler value to divide the internal clock.   |
| Input Parameter 1      | <i>TIMx</i> specifies the <i>TIM</i> to be configured. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Xprescaler</i> : specifies the <i>TIM</i> prescaler value (8bit).   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to configure the *TIM3* prescaler.

```
{
...
TIM_PrescalerConfig (TIM3, 0x5);
...
}
```

**TIM\_PrescalerValue**

| Function Name          | TIM_PrescalerValue  |
|------------------------|---|
| Function Prototype     | <code>u8 TIM_PrescalerValue (TIM_TypeDef *TIMx);</code>   |
| Behavior Description   | This routine gets the <i>TIM</i> prescaler value when the internal clock is used.   |
| Input Parameter        | <i>TIMx</i> : specifies the timer to get its prescaler value. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The Current <i>TIM</i> prescaler Value (8bit).  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to get the *TIM2* prescaler value.

```
{
vu8 bCurrentPrescaler;
...
bCurrentPrescaler = TIM_PrescalerValue (TIM2);
...
}
```

**TIM\_ClockLevelConfig**

| Function Name          | TIM_ClockLevelConfig  |
|------------------------|---|
| Function Prototype     | <code>void TIM_ClockLevelConfig (TIM_TypeDef *TIMx, Clock_Levels Xlevel);</code>  |
| Behavior Description   | This routine configures the active level when using an external clock source.   |
| Input Parameter 1      | <i>TIMx</i> : specifies the TIM to be configured. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.                             |
| Input Parameter 2      | <i>Xlevel</i> : specifies the active edge of the external clock. Refer to <a href="#">Clock edges enumeration on page 104</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the *TIM0* to be clocked by an external clock source and the active edge to falling edge.

```

{
...
  /* Configure the Active Clock edge */
  TIM_ClockLevelConfig (TIM0, TIM_FALLING);
  /* Configure the TIM0 source clock */
  TIM_ClockSourceConfig (TIM0, TIM_EXTERNAL)
...
}

```

**TIM\_ClockLevelValue**

| Function Name          | TIM_ClockLevelValue   |
|------------------------|---|
| Function Prototype     | <code>Clock_Levels TIM_ClockLevelValue (TIM_TypeDef *TIMx);</code>  |
| Behavior Description   | This routine gets and returns the clock active level when using an external clock source.   |
| Input Parameter        | <i>TIMx</i> : specifies the TIM to be configured. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.             |
| Output Parameter       | None  |
| Return Value           | The external clock level of the specified timer. Refer to <a href="#">Clock edges enumeration on page 104</a> for more details on the allowed values of this parameter. |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to get the current clock level of the *TIM0*.

```

{
...
TIM_Clock_Edges CurrentLevel;
CurrentLevel = TIM_ClockLevelValue (TIM0);
...
}

```

**TIM\_ICAPModeConfig**

| Function Name          | TIM_ICAPModeConfig  |
|------------------------|---|
| Function Prototype     | <code>void TIM_ICAPModeConfig (TIM_TypeDef *TIMx, TIM_Channels Xchannel, Clock_Edges Xedge);</code>   |
| Behavior Description   | This routine is used to configure the Input capture mode.   |
| Input Parameter 1      | <i>TIMx</i> specifies the <i>TIM</i> to be initialized. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.                 |
| Input Parameter 2      | <i>Xchannel</i> : specifies the input capture channels. Refer to <a href="#">TIM channels enumeration on page 104</a> for more details on the allowed values of this parameter.   |
| Input Parameter 3      | <i>Xedge</i> : specifies the input capture sensitive edge. Refer to <a href="#">Clock edges enumeration on page 104</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |



**Example:**

This example illustrates how to configure the *TIM3* input capture channel A to detect Rising edge.

```
{
...
TIM_ICAPModeConfig (TIM3, TIM_CHANNEL_A, TIM_RISING);
...
}
```

**TIM\_ICAPValue**

| Function Name          | TIM_ICAPValue   |
|------------------------|---|
| Function Prototype     | u16 TIM_ICAPValue (TIM_TypeDef *TIMx, TIM_Channels Xchannel);   |
| Behavior Description   | This routine gets the input capture value.  |
| Input Parameter 1      | <i>TIMx</i> : specifies the TIM to check its Input Capture value.<br>Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.    |
| Input Parameter 2      | <i>Xchannel</i> : specifies the Input Capture channel.<br>Refer to <a href="#">TIM channels enumeration on page 104</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The input capture value of the specified timer and channel  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to get the input capture value of the *TIM3*.

```
{
...
u16 hICAPValue;
hICAPValue = TIM_ICAPValue (TIM3, TIM_CHANNEL_A);
...
}
```

**TIM\_OCMPModeConfig**

| Function Name          | TIM_OCMPModeConfig   |
|------------------------|--|
| Function Prototype     | <pre>void TIM_OCMPModeConfig ( TIM_TypeDef *TIMx,                           TIM_Channels Xchannel,                           u16 XpulseLength,                           TIM_OC_Modes Xmode,                           TIM_Logic_Levels Xlevel);</pre> |
| Behavior Description   | This routine is used to configure the output compare mode.   |
| Input Parameter 1      | <i>TIMx</i> specifies the <i>TIM</i> to be initialized.<br>Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.   |
| Input Parameter 2      | <i>Xchannel</i> : specifies the output compare channel.<br>Refer to <a href="#">TIM channels enumeration on page 104</a> for more details on the allowed values of this parameter.   |
| Input Parameter 3      | <i>XpulseLength</i> : specifies the pulse length.  |
| Input Parameter 4      | <i>Xmode</i> : specifies the output compare mode:<br>Refer to <a href="#">Output compare modes enumeration on page 105</a> for more details on the allowed values of this parameter.   |
| Input Parameter 5      | <i>Xlevel</i> : specifies the level of the external signal after the match occurs:<br>Refer to <a href="#">Logic levels enumeration on page 105</a> for more details on the allowed values of this parameter.  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to configure the *TIM3* output compare channel A to generate an external wave which toggle from low level to high level after 0x7FFF timer period.

```
{
...
TIM_OCMPModeConfig (TIM3, TIM_CHANNEL_A, 0x7FFF, TIM_WAVE,
TIM_HIGH)
...
}
```

**TIM\_OPModeConfig**

| Function Name          | TIM_OPModeConfig   |
|------------------------|--|
| Function Prototype     | <pre>void TIM_OPModeConfig (TIM_TypeDef *TIMx,                        ul6 XpulseLength,                        TIM_Loic_Levels XLevel1,                        TIM_Loic_Levels XLevel2,                        TIM_Clock_Edges Xedge);</pre>   |
| Behavior Description   | This routine is used to configure the one pulse mode.  |
| Input Parameter 1      | <i>TIMx</i> specifies the TIM to be configured. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.  |
| Input Parameter 2      | <i>XpulseLength</i> : specifies the pulse length.  |
| Input Parameter 3      | <i>XLevel1</i> : specifies the output level on the OCMPA pin during the pulse. Refer to <a href="#">Logic levels enumeration on page 105</a> for more details on the allowed values of this parameter.   |
| Input Parameter 4      | <i>XLevel2</i> : specifies the output level on the OCMPB pin after the pulse. Refer to <a href="#">Logic levels enumeration on page 105</a> for more details on the allowed values of this parameter.  |
| Input Parameter 5      | <p><i>Xedge</i>: specifies the edge to be detected by the input capture A pin. Refer to <a href="#">Clock edges enumeration on page 104</a> for more details on the allowed values of this parameter.</p> <ul style="list-style-type: none"> <li>– <i>TIM_RISING</i>: a rising edge will activate the one pulse mode</li> <li>– <i>TIM_FALLING</i>: a falling edge will activate the one pulse mode</li> </ul> |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to configure the *TIM3* one pulse mode to generate an external wave with a pulse length of 0xABC in terms of timer period. The pulse generation is activated by a rising edge on the input capture A pin.

```
{
...
TIM_OPModeConfig (TIM3, 0xABC, TIM_HIGH, TIM_LOW, TIM_RISING);
...
}
```

**TIM\_PWMOModeConfig**

| Function Name          | TIM_PWMOModeConfig   |
|------------------------|--|
| Function Prototype     | <code>void TIM_PWMOModeConfig (TIM_TypeDef *TIMx, u16 XDutyCycle, TIM_Logic_Levels XLevel1, u16 XFullperiod, TIM_Logic_Levels XLevel2);</code>   |
| Behavior Description   | This routine is used to configure the PWM output mode.   |
| Input Parameter 1      | <i>TIMx</i> specifies the <i>TIM</i> to be initialized. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.                              |
| Input Parameter 2      | <i>XDutyCycle</i> : specifies the PWM signal duty cycle.   |
| Input Parameter 3      | <i>XLevel1</i> : specifies the PWM signal level during the duty cycle. Refer to <a href="#">Logic levels enumeration on page 105</a> for more details on the allowed values of this parameter. |
| Input Parameter 4      | <i>XFullperiod</i> : specifies the PWM signal full period.   |
| Input Parameter 5      | <i>XLevel2</i> : specifies the PWM signal level out of the duty cycle. Refer to <a href="#">Logic levels enumeration on page 105</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to configure the *TIM3* to generate a PWM with a period of 0xFFFF timer period and a duty cycle of 50%.

```

{
...
TIM_PWMOModeConfig (TIM3, 0x7FF, TIM_HIGH, 0xFFF, TIM_LOW);
...
}

```

## TIM\_PWMIModeConfig

| Function Name          | TIM_PWMIModeConfig   |
|------------------------|--|
| Function Prototype     | <code>void TIM_PWMIModeConfig (TIM_TypeDef *TIMx, TIM_Clock_Edges Xedge);</code>   |
| Behavior Description   | This routine is used to configure the PWM input mode.  |
| Input Parameter 1      | <i>TIMx</i> specifies the <i>TIM</i> to be initialized. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.                          |
| Input Parameter 2      | <i>Xedge</i> : specifies the first edge of the external PWM signal. Refer to <a href="#">Clock edges enumeration on page 104</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to configure the *TIM3* to measure an external PWM signal.

```
{
...
TIM_PWMIModeConfig (TIM3, TIM_RISING);
...
}
```

## TIM\_PWMIValue

| Function Name          | TIM_PWMIValue   |
|------------------------|---|
| Function Prototype     | <code>PWMI_parameters TIM_PWMIValue (TIM_TypeDef *TIMx);</code>   |
| Behavior Description   | This routine is used to configure the PWM input mode.   |
| Input Parameter        | <i>TIMx</i> specifies the <i>TIM</i> to be initialized. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.         |
| Output Parameter       | None  |
| Return Value           | The PWM input parameters: pulse and period. Refer to <a href="#">PWM input parameters structure on page 106</a> for more details on the allowed values of this parameter. |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to get the PWM input signal parameters.

```
{
...
PWMI_parameters PWM_InputSignal;
PWM_InputSignal = TIM_PWMIValue (TIM3);
...
}
```

**TIM\_CounterConfig**

| Function Name          | TIM_CounterConfig   |
|------------------------|---|
| Function Prototype     | void TIM_CounterConfig (TIM_TypeDef *TIMx, CounterOperations Xoperation);   |
| Behavior Description   | This routine is used to start/stop and clear the selected timer counter   |
| Input Parameter 1      | <i>TIMx</i> specifies the <i>TIM</i> to be initialized. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.                         |
| Input Parameter 2      | <i>Xoperation</i> : specifies the operation of the counter. Refer to <a href="#">Counter operations enumeration on page 105</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to stop the *TIM3* counter.

```

{
...
TIM_CounterConfig (TIM3, TIM_STOP);
...
}

```

**TIM\_CounterValue**

| Function Name          | TIM_CounterValue   |
|------------------------|--|
| Function Prototype     | u16 TIM_CounterValue(TIM_TypeDef *TIMx)  |
| Behavior Description   | This routine returns the timer counter value.  |
| Input Parameter 1      | <i>TIMx</i> : specifies the <i>TIM</i> to get its counter value. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | The selected timer counter value   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to get the *TIM0* timer counter value.

```

{
u16 wTIMCounterValue;
...
wTIMCounterValue = TIM_CounterValue (TIM0);
...
}

```

## TIM\_ITConfig

| Function Name          | TIM_ITConfig  |
|------------------------|---|
| Function Prototype     | <code>void TIM_ITConfig (TIM_TypeDef *TIMx, u16 New_IT, FunctionalState NewState);</code>   |
| Behavior Description   | This routine is used to configure the <i>TIM</i> interrupt.   |
| Input Parameter 1      | <i>TIMx</i> : specifies the <i>TIM</i> to be configured. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.  |
| Input Parameter 2      | <i>New_IT</i> : specifies the <i>TIM</i> interrupt to be configured. You can specify one or more <i>TIM</i> interrupts to be configured using the logical operator 'OR'. Refer to <a href="#">TIM interrupts on page 106</a> for more details on the allowed values of this parameter.                                    |
| Input Parameter 3      | <i>NewState</i> : specifies the <i>TIM</i> interrupt state whether it would be enabled or disabled.<br><ul style="list-style-type: none"> <li>– <i>ENABLE</i>: the corresponding <i>TIM</i> interrupt(s) will be enabled</li> <li>– <i>DISABLE</i>: the corresponding <i>TIM</i> interrupt(s) will be disabled</li> </ul> |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to enable the *TIM3* overflow and input capture A interrupts.

```

{
...
    TIM_ITConfig (TIM3, TIM_TO_IT | TIM_ICA_IT, ENABLE);
...
}

```

## TIM\_FlagStatus

| Function Name          | TIM_FlagStatus  |
|------------------------|---|
| Function Prototype     | <code>FlagStatus TIM_FlagStatus (TIM_TypeDef *TIMx, TIM_Flags Xflag);</code>  |
| Behavior Description   | This routine is used to check a <i>TIM</i> flag status.   |
| Input Parameter 1      | <i>TIMx</i> : specifies the <i>TIM</i> to check a flag. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.               |
| Input Parameter 2      | <i>Xflag</i> : specifies the <i>TIM</i> flag to be tested. Refer to <a href="#">TIM flags enumeration on page 103</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The flag status passed in parameter.  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to check whether the Timer overflow interrupt of the *TIM3* is set.

```
{
...
FlagStatus TOF_Status;
TIM_FlagStatus = TIM_FlagStatus (TIM3, TIM_TOF);
...
}
```

**TIM\_FlagClear**

| Function Name          | TIM_FlagClear   |
|------------------------|---|
| Function Prototype     | void TIM_FlagClear (TIM_TypeDef *TIMx, TIM_Flags Xflag)   |
| Behavior Description   | This routine is used to clear the <i>TIM</i> flags.   |
| Input Parameter 1      | <i>TIMx</i> : specifies the TIM to clear a flag. Refer to <a href="#">TIM values on page 103</a> for more details on the allowed values of this parameter.        |
| Input Parameter 2      | Xflag: specifies the TIM flag to be cleared. Refer to <a href="#">TIM flags enumeration on page 103</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to clear the Timer overflow interrupt flag.

```
{
...
TIM_FlagClear (TIM3, TIM_TOF);
...
}
```

**3.10 Buffered serial peripheral interface (BSPI)**

The *BSPI* driver may be used for a variety of purposes including allowing several external peripherals to be linked through an *SPI* protocol bus.

The first section describes the data structures used in the *BSPI* firmware library. The second section presents the *BSPI* firmware library functions



### 3.10.1 Data structures

#### BSPI register structure

The *BSPI* peripheral register structure *BSPI\_TypeDef* is defined in the file *71x\_map.h* as follows:

```
typedef struct
{
    vu16 RXR;
    u16  EMPTY1;
    vu16 TXR;
    u16  EMPTY2;
    vu16 CSR1;
    u16  EMPTY3;
    vu16 CSR2;
    u16  EMPTY4;
    vu16 CLK;
    u16  EMPTY5;
} BSPI_TypeDef;
```

[Table 57](#) describes the *BSPI* registers.

**Table 57. BSPI registers**

| Register | Description                   |
|----------|-------------------------------|
| RXR      | Receive Register              |
| TXR      | Transmit Register             |
| CSR1     | Control/Status Register 1     |
| CSR2     | Control/Status Register 2     |
| CLK      | Master Clock Divider Register |

The two *BSPI* interfaces are declared in the same file *71x\_map.h*:

```
/* APB1 Base Address definition*/
#define APB1_BASE          0xC0000000

/* BSPI Base Address definition*/
#define BSPI0_BASE         (APB1_BASE + 0xA000)
#define BSPI1_BASE         (APB1_BASE + 0xB000)

/* BSPI peripheral pointer declaration*/
#ifndef DEBUG
...
#ifdef _BSPI0
#define BSPI0                ((BSPI_TypeDef *)BSPI0_BASE)
#endif /*BSPI0*/

#ifdef _BSPI1
#define BSPI1                ((BSPI_TypeDef *)BSPI1_BASE)
#endif /*BSPI1*/
...
#else /*DEBUG */
...

```

```

#ifdef _BSPIO
    EXT BSPI_TypeDef          *BSPIO;
#endif /*_BSPIO*/

#ifdef _BSPI1
    EXT BSPI_TypeDef          *BSPI1;
#endif /*_BSPI1*/...
#endif

```

When debug mode is used, *BSPI* pointer is initialized in *71x\_lib.c* file:

```

#ifdef _BSPIO
    BSPIO = (BSPI_TypeDef *)BSPIO_BASE;
#endif /*_BSPIO */
#ifdef _BSPI1
    BSPI1 = (BSPI_TypeDef *)BSPI1_BASE;
#endif /*_BSPI1 */

```

In debug mode, *\_BSPIO* and *\_BSPI1* must be defined, in *71x\_conf.h* file, to access the peripheral registers as follows:

```

#define _BSPI
#define _BSPIO
#define _BSPI1
...

```

### BSPI flags

The following enumeration defines the *BSPI* flags. *BSPI\_Flags* enumeration is defined in the file *71x\_bsp.h*:

```

typedef enum {
    BSPI_BERR = 0x04,
    BSPI_RFNE = 0x08,
    BSPI_RFF = 0x10,
    BSPI_ROFL = 0x20,
    BSPI_TFE = 0x40,
    BSPI_TUFL = 0x80,
    BSPI_TFF = 0x100,
    BSPI_TFNE = 0x200
} BSPI_Flags;

```

[Table 58](#) describes the BSPI flags.

**Table 58. BSPI flags**

| BSPI Register | Register Offset | Flag  | Flag Index | Flag Code |
|---------------|-----------------|---|------------|-----------|
| BSP_SR2       | 0x0C            | <i>BSPI_BERR</i> : Bus error flag               | 2          | 0x04      |
|               |                 | <i>BSPI_RFNE</i> : Receive FIFO not empty flag  | 3          | 0x08      |
|               |                 | <i>BSPI_RFF</i> : Receive FIFO full flag        | 4          | 0x10      |
|               |                 | <i>BSPI_ROFL</i> : Receive overflow flag        | 5          | 0x20      |
|               |                 | <i>BSPI_TFE</i> : Transmit FIFO empty flag      | 6          | 0x40      |
|               |                 | <i>BSPI_TUFL</i> : Transmit underflow flag      | 7          | 0x80      |
|               |                 | <i>BSPI_TFF</i> : Transmit FIFO full flag       | 8          | 0x100     |
|               |                 | <i>BSPI_TFNE</i> : Transmit FIFO not empty flag | 9          | 0x200     |

### BSPI interrupts

The following enumeration defines the *BSPI* interrupts. The *BSPI\_ITS\_ERR* is defined in the file *71x\_bspi.h* as follows:

```
typedef enum
{
    BSPI_BEIE = 0x80,
    BSPI_REIE = 0x10,
    BSPI_ALL_ERR = 0x90
} BSPI_IT_ERR;
```

[Table 59](#) enumerates the *BSPI* interrupts.

**Table 59. BSPI interrupts**

| BSPI Interrupt Flags | Description   |
|----------------------|---|
| BSPI_BEIE            | Bus Error Interrupt   |
| BSPI_REIE            | Receive Error Interrupt                                     |
| BSPI_ALL_ERR         | All Error interrupts: <i>BSPI_BEIE</i> and <i>BSPI_REIE</i> |

### Transmit interrupt sources

The following enumeration defines the *BSPI* transmit interrupt sources. The *TR\_IT\_SRCS* is defined in the file *71x\_bspi.h* as follows:

```
typedef enum
{
    BSPI_TR_FE,
    BSPI_TR_UFL,
    BSPI_TR_FF,
    BSPI_TR_DIS
} BSPI_IT_TR;
```

[Table 60](#) enumerates the *BSPI* transmit interrupt sources.

**Table 60. BSPI transmit interrupt sources**

| Transmit interrupt source | Description             |
|---------------------------|-------------------------|
| BSPI_TR_FE                | Transmit FIFO empty     |
| BSPI_TR_UFL               | Transmit underflow      |
| BSPI_TR_FF                | Transmit FIFO full      |
| BSPI_TR_DIS               | All interrupts disabled |

### Receive interrupt sources

The following enumeration defines the *BSPI* receive interrupt sources. The *RC\_IR\_SRCS* is defined in the file *71x\_bspi.h* as follows:

```
typedef enum
{
    BSPI_RC_FNE,
    BSPI_RC_FF,
    BSPI_RC_DIS
} BSPI_IT_RC;
```

[Table 61](#) enumerates the *BSPI* receive interrupt sources.

**Table 61. BSPI receive interrupt sources**

| Receive interrupt source | Description             |
|--------------------------|-------------------------|
| BSPI_RC_FNE              | Receive FIFO not empty  |
| BSPI_RC_FF               | Receive FIFO full       |
| BSPI_RC_DIS              | All interrupts disabled |

## 3.10.2 Firmware library functions

[Table 62](#) enumerates the different functions of the *BSPI* library.

**Table 62. BSPI library functions**

| Function Name                     | Description   |
|-----------------------------------|---|
| <a href="#">BSPI_BSPIOConf</a>    | Enable or disable <i>BSPI0</i> interface  |
| <a href="#">BSPI_Init</a>         | Initializes <i>BSPI</i> peripheral control and registers to their default reset values. |
| <a href="#">BSPI_Enable</a>       | Enables/disables the specified <i>BSPI</i> peripheral                                   |
| <a href="#">BSPI_MasterEnable</a> | Selects or deselects <i>BSPI</i> master mode  |
| <a href="#">BSPI_TrltSrc</a>      | Configures the transmit interrupt source  |
| <a href="#">BSPI_RcItSrc</a>      | Configures the receive interrupt source   |
| <a href="#">BSPI_TrFifoDepth</a>  | Configures the number of the words for the <i>BSPI</i> transmit <i>FIFO</i>             |

Table 62. BSPI library functions (continued)

| Function Name                  | Description   |
|--------------------------------|---|
| <i>BSPI_RcFifoDepth</i>        | Configures the number of the words for the <i>BSPI</i> receive <i>FIFO</i>                    |
| <i>BSPI_8bLEn</i>              | Sets the word length to 8 or 16 bits  |
| <i>BSPI_ClkFEdge</i>           | Enables capturing the first data sample on the first edge of <i>SCK</i> or on the second edge |
| <i>BSPI_ClkActiveHigh</i>      | Configures the clock to be active high or low   |
| <i>BSPI_FifoDisable</i>        | Disables <i>BSPI FIFO</i>   |
| <i>BSPI_ClockDividerConfig</i> | Configures <i>BSPI</i> clock divider  |
| <i>BSPI_FlagStatus</i>         | Checks whether the specified Flag is set or not   |
| <i>BSPI_WordSend</i>           | Transmits a single word of data   |
| <i>BSPI_WordReceive</i>        | Returns the recent received word  |
| <i>BSPI_ByteBufferSend</i>     | Transmits 8 bits data from a buffer   |
| <i>BSPI_ByteBufferReceive</i>  | Returns the recent 8 bits data received   |
| <i>BSPI_WordBufferSend</i>     | Transmits 16 bits of data from a buffer   |
| <i>BSPI_WordBufferReceive</i>  | Returns the recent 16 bits data received  |
| <i>BSPI_ErrItSrc</i>           | Enables the specified error interrupt   |

**BSPI\_BSPIOConf**

| Function Name          | BSPI_BSPIOConf  |
|------------------------|---|
| Function Prototype     | <code>void BSPI_BSPIOConf(FunctionalState NewState);</code>   |
| Description            | Configure <i>BSPI0</i> to be available on <i>GPIO0.0</i> to <i>GPIO0.3</i> pins.  |
| Input Parameter        | NewState: specified the status of the <i>BSPI0</i> .<br>– <i>ENABLE</i> : <i>BSPI0</i> interface will be enabled.<br>– <i>DISABLE</i> : <i>BSPI0</i> interface will be disabled |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable the *BSPI0* interface:

```

{
    ...
    BSPI_BSPIOConf (ENABLE);
    ...
}

```

**BSPI\_Init**

| Function Name          | BSPI_Init   |
|------------------------|---|
| Function Prototype     | <code>void BSPI_Init (BSPI_TypeDef *BSPIx);</code>  |
| Behavior Description   | Initializes <i>the BSPI</i> peripheral control and registers to their default reset values. |
| Input Parameter        | BSPIx: selects the BSPI peripheral to be initialized where x can be 0 or 1.                 |
| Output Parameters      | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to initialize the *BSPI0*:

```
{
...
  BSPI_Init (BSPI0);
...
}
```

**BSPI\_Enable**

| Function Name          | BSPI_Enable   |
|------------------------|---|
| Function Prototype     | <code>void BSPI_Enable (BSPI_TypeDef *BSPIx, FunctionalState NewState);</code>  |
| Behavior Description   | Enables/disables the specified BSPI peripheral.   |
| Input Parameter 1      | BSPIx: selects the BSPI peripheral to be enabled or disabled where x can be 0 or 1.   |
| Input Parameter 2      | NewState: specifies the status of the <i>BSPI</i> .<br>– <i>ENABLE</i> : the specified <i>BSPI</i> peripheral will be enabled.<br>– <i>DISABLE</i> : the specified <i>BSPI</i> peripheral will be disabled. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | I/O ports should be configured in their reset states and clock divider must be configured   |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable and disable the *BSPI0* interface:

```
{
...
  /*Configures BSPI clock divider */
  BSPI_ClockDividerConfig(BSPI0, 8);
...
}
```

```

...
/*Enable BSPI0 peripheral */
BSPI_Enable (BSPI0, ENABLE);
...
/*User code */
...
/* Disable BSPI0 peripheral */
BSPI_Enable (BSPI0, DISABLE);
...
}

```

## BSPI\_MasterEnable

| Function Name          | BSPI_MasterEnable  |
|------------------------|--|
| Function Prototype     | void BSPI_MasterEnable (BSPI_TypeDef *BSPIx, FunctionalState NewState);  |
| Behavior Description   | Configures the <i>BSPI</i> as a Master or a Slave.   |
| Input Parameter 1      | <i>BSPIx</i> : selects the BSPI peripheral to be configured as master or slave where x can be 0 or 1.  |
| Input Parameter 2      | <i>NewState</i> : specifies whether <i>BSPI</i> will be configured as master or slave.<br>– <i>ENABLE</i> : the specified <i>BSPI</i> will be configured as a master.<br>– <i>DISABLE</i> : the specified <i>BSPI</i> will be configured as a slave. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to configure the *BSPI0* as a master or a slave

```

{
...
/* Enable BSPI0 interface */
BSPI_BSPIOConf (ENABLE);
...

...
/*Configure BSPI0 as a master */
BSPI_MasterEnable (BSPI0, ENABLE);
...
/*Enable BSPI0 peripheral */
BSPI_Enable (BSPI0, ENABLE);
...
}

```

**BSPI\_TrItSrc**

| Function Name          | BSPI_TrItSrc  |
|------------------------|---|
| Function Prototype     | <code>void BSPI_TrItSrc(BSPI_TypeDef *BSPIx, TR_IT_SRCS TrItSrc);</code>  |
| Behavior Description   | Configures the transmit interrupt source.   |
| Input Parameter 1      | <i>BSPIx</i> : selects the BSPI peripheral to be configured where x can be 0 or 1.  |
| Input Parameter 2      | <i>TrItSrc</i> : specifies the transmit interrupt source. Refer to <a href="#">Transmit interrupt sources on page 123</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | <i>BSPI</i> system must be enabled  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the transmit interrupt source in the *BSPI0* peripheral:

```

{
    ...
    /*Enable BSPI0 peripheral */
    BSPI_Enable (BSPI0, ENABLE);
    ...
    /*Enable BSPI0 to generate an interrupt when a transmit
    underflow occurs */
    BSPI_TrItSrc(BSPI0, BSPI_TR_UFL);
    ...
}

```



**BSPI\_RcItSrc**

| Function Name          | BSPI_RcItSrc  |
|------------------------|---|
| Function Prototype     | <code>void BSPI_RcItSrc (BSPI_TypeDef *BSPIx,<br/>RC_IT_SRCS RcItSrc);</code>   |
| Behavior Description   | Configures the receive interrupt source.  |
| Input Parameter 1      | <i>BSPIx</i> : selects the BSPI peripheral to be configured where x can be 0 or 1.  |
| Input Parameter 2      | <i>RcItSrc</i> : specifies the source for the receive interrupt. Refer to <a href="#">Receive interrupt sources on page 124</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | <i>BSPI</i> system must be enabled  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the receive interrupt source in the *BSPI0* peripheral:

```

{
    ...
    /*Enable BSPI0 peripheral */
    BSPI_Enable (BSPI0, ENABLE);
    ...

    /*Enable BSPI0 to generate an interrupt when the FIFO is not
empty */
    BSPI_RcItSrc (BSPI0, BSPI_RC_FNE);
    ...
}

```

**BSPI\_TrFifoDepth**

| Function Name          | BSPI_TrFifoDepth   |
|------------------------|--|
| Function Prototype     | <code>void BSPI_TrFifoDepth(BSPI_TypeDef *BSPIx, u8 TDepth);</code>                |
| Behavior Description   | Configures the depth of the BSPI transmission FIFO.                                |
| Input Parameter 1      | <i>BSPIx</i> : selects the BSPI peripheral to be configured where x can be 0 or 1. |
| Input Parameter 2      | <i>TDepth</i> : specifies the depth of the transmit FIFO.                          |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to set the depth of the transmit *FIFO* to 8 words:

```
{
  ...
  BSPI_TrFifoDepth(BSPI0, 8);
  ...
}
```

**BSPI\_RcFifoDepth**

| Function Name          | BSPI_RcFifoDepth   |
|------------------------|--|
| Function Prototype     | <code>void BSPI_RcFifoDepth(BSPI_TypeDef *BSPIx, u8 RDepth);</code>            |
| Behavior Description   | Configures the depth of the BSPI reception FIFO.                               |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1. |
| Input Parameter 2      | <i>RDepth</i> : specifies the depth of the receive <i>FIFO</i> .               |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to set the depth of the receive *FIFO* to 5 words:

```
{
  ...
  BSPI_RcFifoDepth(BSPI0, 5);
  ...
}
```

**BSPI\_8bLEn**

| Function Name          | BSPI_8bLEn  |
|------------------------|---|
| Function Prototype     | <code>void BSPI_8bLEn(BSPI_TypeDef *BSPIx,<br/>FunctionalState NewState);</code>  |
| Behavior Description   | Sets the word length of the receive FIFO and transmit data registers to either 8 or 16 bits.  |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1.  |
| Input Parameter 2      | NewState: specifies if the word length is 8 or 16 bits.<br>– <i>ENABLE</i> : enables setting the word length to 8 bits<br>– <i>DISABLE</i> : disables setting the word length to 8 bits: the word length will be set to 16 bits |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to set word length to 16 bits for the *BSPI0* peripheral:

```
{
  ...
  BSPI_8bLEn(BSPI0, DISABLE);
  ...
}
```

**BSPI\_ClkFEdge**

| Function Name          | <b>BSPI_ClkFEdge</b>  |
|------------------------|---|
| Function Prototype     | <code>void BSPI_ClkFEdge (BSPI_TypeDef *BSPIx, FunctionalState NewState);</code>  |
| Behavior Description   | Enables capturing the first data sample on the first edge of <i>SCK</i> or on the second edge.  |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1.  |
| Input Parameter 2      | NewState: specifies whether capturing the first data sample on the first edge of <i>SCK</i> is enabled or disabled.<br><ul style="list-style-type: none"> <li>– <i>ENABLE</i>: enables capturing the first data sample on the first edge of <i>SCK</i></li> <li>– <i>DISABLE</i>: enables capturing the first data sample on the second edge of <i>SCK</i></li> </ul> |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure *BSPI0* to capture the first data sample on the first edge of *SCK*:

```

{
    ...
    BSPI_ClkFEdge (BSPI0, ENABLE);
    ...
}

```

**BSPI\_ClkActiveHigh**

| Function Name          | <b>BSPI_ClkActiveHigh</b>  |
|------------------------|--|
| Function Prototype     | <code>void BSPI_ClkActiveHigh (BSPI_TypeDef *BSPIx, FunctionalState NewState);</code>  |
| Behavior Description   | Configures the clock to be active high or low.   |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1.   |
| Input Parameter 2      | <i>Status</i> : specifies whether the clock is active high or low.<br>– <i>ENABLE</i> : configures the clock to be active high<br>– <i>DISABLE</i> : configures the clock to be active low |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to configure *BSPI0* clock to be active high:

```

{
    BSPI_ClkActiveHigh (BSPI0, ENABLE);
    ...
}

```

**BSPI\_FifoDisable**

| Function Name          | <b>BSPI_FifoDisable</b>  |
|------------------------|--|
| Function Prototype     | <code>void BSPI_FifoDisable (BSPI_TypeDef *BSPIx);</code>                      |
| Behavior Description   | Disables the FIFO of the specified BSPI.                                       |
| Input Parameter        | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to disable the *BSPI0 FIFO*:

```

{
    BSPI_FifoDisable (BSPI0);
    ...
}

```

**BSPI\_ClockDividerConfig**

| Function Name          | BSPI_ClockDividerConfig   |
|------------------------|---|
| Function Prototype     | <code>void BSPI_ClockDividerConfig (BSPI_TypeDef *BSPIx, u8 Div);</code>                                  |
| Behavior Description   | Configures <i>BSPI</i> clock divider.   |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1 to select the <i>BSPI</i> . |
| Input Parameter 2      | <i>Div</i> : holds the value of the clock divider.  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to set *BSPI0* clock divider to 8:

```

{
    BSPI_ClockDividerConfig (BSPI0, 8);
    ...
}

```

**BSPI\_FlagStatus**

| Function Name          | BSPI_FlagStatus  |
|------------------------|--|
| Function Prototype     | <code>FlagStatus BSPI_FlagStatus (BSPI_TypeDef *BSPIx, BSPI_Flags flag);</code>  |
| Description            | Checks whether the specified <i>BSPI</i> Flag is set or not.   |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1.   |
| Input Parameter 2      | <i>flag</i> : specifies the flag to see the status.<br>Refer to <a href="#">BSPI flags on page 122</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | The status of the specified flag:<br>– <i>SET</i> : if the tested <i>flag</i> is set<br>– <i>RESET</i> : if the tested <i>flag</i> is reset                      |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to test if the *BSPI0 TFE* (Transmit FIFO Empty) is set:

```

{
    FlagStatus bStatus1;
    ...
    /* Get the status of the BSPI transmit Fifo not empty flag */
    bStatus1 = BSPI_FlagStatus (BSPI0, BSPI_TFE);
    ...
}

```

**BSPI\_WordSend**

| Function Name          | BSPI_WordSend  |
|------------------------|--|
| Function Prototype     | void BSPI_WordSend (BSPI_TypeDef *BSPIx, u16 Data);                            |
| Description            | Transmits a single Word.   |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be configured where x can be 0 or 1. |
| Input Parameter 2      | Data: the word which will be transmitted.                                      |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to use the *BSPI\_WordSend* function to send a word 0x5D in the:

```

{
    ...
    /*Transmit a single Word */
    BSPI_WordSend (BSPI0, 0x5D);
    ...
}

```

**BSPI\_ByteBufferSend**

| Function Name          | BSPI_ByteBufferSend  |
|------------------------|--|
| Function Prototype     | <code>void BSPI_ByteBufferSend(BSPI_TypeDef *BSPIx, u8 *PtrToBuffer, u8 NbOfWords)</code>  |
| Behavior Description   | Transmits 8 bit data format from a buffer.   |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be used where x can be 0 or 1.                   |
| Input Parameter 2      | <i>PtrToBuffer</i> : is an 'u8' pointer to the first byte of the buffer to be transmitted. |
| Input Parameter 3      | <i>NbOfWords</i> : the number of bytes saved in the buffer to be sent.                     |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | BSPI_WordSend  |

**Example:**

This example illustrates how to send a buffer:

```

{
    vu8 pBuffer[] = "Hello";
    ...
    // Transmit 5 bytes data
    BSPI_ByteBufferSend (BSPI0, pBuffer, 5);
    ...
}

```



**BSPI\_WordBufferSend**

| Function Name          | BSPI_WordBufferSend   |
|------------------------|---|
| Function Prototype     | <code>void BSPI_WordBufferSend(BSPI_TypeDef *BSPIx, u16 *PtrToBuffer, u8 NbOfWords)</code>  |
| Description            | Transmits 16 bits data format from a buffer.  |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be used where x can be 0 or 1.                    |
| Input Parameter 2      | <i>PtrToBuffer</i> : is an 'u16' pointer to the first word of the buffer to be transmitted. |
| Input Parameter 3      | <i>NbOfWords</i> : indicates the number of words saved in the buffer to be sent.            |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | BSPI_WordSend   |

**Example:**

This example illustrates how to send a buffer:

```

{
    u16 pBuffer[] = {0x1234,0xAAAA,0x5555};
    ...
    /*Transmit 3 16-bit data format */
    BSPI_ByteBufferSend (BSPI0, pBuffer, 3);
    ...
}

```

**BSPI\_WordReceive**

| Function Name          | BSPI_BufferReceive   |
|------------------------|--|
| Function Prototype     | <code>u16 BSPI_WordReceive(BSPI_TypeDef *BSPIx);</code>                  |
| Description            | Returns the recent received word.  |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be used where x can be 0 or 1. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to receive a data from the BSPI0:

```

{
    u16 data;
    ...
    /*Receive a data from BSPI0 */
    data = BSPI_WordReceive (BSPI0);
    ...
}

```

**BSPi\_ByteBufferReceive**

| Function Name          | <b>BSPi_ByteBufferReceive</b>  |
|------------------------|--|
| Function Prototype     | void BSPi_ByteBufferReceive(BSPi_TypeDef *BSPiX, u8 *PtrToBuffer, u8 NbofWords)                    |
| Description            | Receives number of data words and stores them in user defined area.                                |
| Input Parameter 1      | <i>BSPiX</i> :selects BSPi peripheral to be used where x can be 0 or 1.                            |
| Input Parameter 2      | PtrToBuffer: is an 'u8' pointer to the first word of the defined area to save the received buffer. |
| Input Parameter 3      | NbofWords: indicates the number of bytes to be received in the buffer.                             |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | BSPi_WordReceive   |

**Example:**

This example illustrates how to receive 5 word from a buffer:

```

{
    u8 pBuffer[10];
    ...
    /*Receive 5 8-bit data and stores them in user-defined area */
    BSPi_ByteBufferReceive (BSPi0, pBuffer, 5);
    ...
}

```

**BSPi\_WordBufferReceive**

| Function Name          | <b>BSPi_ByteBufferReceive</b>   |
|------------------------|---|
| Function Prototype     | void BSPi_WordBufferReceive(BSPi_TypeDef *BSPiX, u16 *PtrToBuffer, u8 NbofWords)                    |
| Description            | Receives number of data words and stores them in user defined area.                                 |
| Input Parameter 1      | <i>BSPiX</i> : selects BSPi peripheral to be used where x can be 0 or 1.                            |
| Input Parameter 2      | PtrToBuffer: is an 'u16' pointer to the first word of the defined area to save the received buffer. |
| Input Parameter 3      | NbofWords: indicates the number of words to be received in the buffer.                              |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | BSPi_WordReceive  |

**Example:**

This example illustrates how to receive 5 word from a buffer:

```

{
    u16 pBuffer[10];
    ...
    /*Receive 5 16-bit data and stores them in user-defined area */
    BSPI_WordBufferReceive (BSPI0, pBuffer, 5);
    ...
}

```

**BSPI\_ErrItSrc**

| Function Name          | BSPI_ErrItSrc   |
|------------------------|---|
| Function Prototype     | void BSPI_ErrItSrc (BSPI_TypeDef *BSPIx, BSPI_IT_ERR BSPI_IE, FunctionalState NewState );   |
| Description            | Enables or disables the specified error interrupt.  |
| Input Parameter 1      | <i>BSPIx</i> : selects BSPI peripheral to be used where x can be 0 or 1.  |
| Input Parameter 2      | <i>BSPI_IE</i> : specifies the BSPI error interrupt to be enabled or disabled.<br>Refer to <a href="#">BSPI interrupts on page 123</a> for more details on the allowed values of this parameter |
| Input Parameter 3      | <i>NewState</i> : specified whether the <i>BSPI</i> error interrupt is enabled or disabled.<br>– <i>ENABLE</i> : to enable interrupt.<br>– <i>DISABLE</i> : to disable interrupt.               |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable the interrupt feature in the *BSPI0* peripheral:

```

{
    ...
    BSPI_ErrItSrc (BSPI0, BSPI_BEIE, ENABLE);
    ...
}

```

## 3.11 Universal asynchronous receiver transmitter (UART)

The *UART* driver may be used for a variety of purposes, including mode selection, Baud rate configuration and 8 and 9-bit byte reception and transmission.

The first section describes the data structures used in the *UART* firmware library. The second section presents the *UART* firmware library functions.

### 3.11.1 Data structures

#### UART register structure

The *UART* register structure *UART\_TypeDef* is defined in the *71x\_map.h* file as follows:

```
typedef struct
{
    vu16 BR;
    u16  EMPTY1;
    vu16 TxBUFR;
    u16  EMPTY2;
    vu16 RxBUFR;
    u16  EMPTY3;
    vu16 CR;
    u16  EMPTY4;
    vu16 IER;
    u16  EMPTY5;
    vu16 SR;
    u16  EMPTY6;
    vu16 GTR;
    u16  EMPTY7;
    vu16 TOR;
    u16  EMPTY8;
    vu16 TxRSTR;
    u16  EMPTY9;
    vu16 RxRSTR;
    u16  EMPTY10;
} UART_TypeDef;
```

[Table 63](#) describes the *UART* structure fields.

**Table 63. UART structure fields**

| Register | Description               |
|----------|---------------------------|
| BR       | Baud Rate Register        |
| TxBUFR   | Transmit Buffer Register  |
| RxBUFR   | Receive Buffer Register   |
| CR       | Control Register          |
| IER      | Interrupt Enable Register |
| SR       | Status Register           |
| GTR      | Guard Time Register       |
| TOR      | Time Out Register         |
| TxRSTR   | Tx FIFO Reset Register    |
| RxRSTR   | Rx FIFO Reset Register    |

The *UARTs* peripherals are declared in the same file:

```

...
#define APB1_BASE          0xC0000000
#define UART0_BASE        (APB1_BASE + 0x4000)
#define UART1_BASE        (APB1_BASE + 0x5000)
#define UART2_BASE        (APB1_BASE + 0x6000)
#define UART3_BASE        (APB1_BASE + 0x7000)
...
#ifndef DEBUG
...
#ifdef _UART0
#define UART0              ((UART_TypeDef *)UART0_BASE)
#endif /*_UART0*/

#ifdef _UART1
#define UART1              ((UART_TypeDef *)UART1_BASE)
#endif /*_UART1*/

#ifdef _UART2
#define UART2              ((UART_TypeDef *)UART2_BASE)
#endif /*_UART2*/

#ifdef _UART3
#define UART3              ((UART_TypeDef *)UART3_BASE)
#endif /*_UART3*/
...
#else /* DEBUG */
...
#ifdef _UART0
EXT UART_TypeDef          *UART0;
#endif /*_UART0*/

#ifdef _UART1
EXT UART_TypeDef          *UART1;
#endif /*_UART1*/

#ifdef _UART2
EXT UART_TypeDef          *UART2;
#endif /*_UART2*/

#ifdef _UART3
EXT UART_TypeDef          *UART3;
#endif /*_UART3*/
...
#endif

```

When debug mode is used, the *UART* pointers are initialized in the file *71x\_lib.c*:

```

#ifdef _UART0
    UART0 = (UART_TypeDef *)UART0_BASE;
#endif /* _UART0 */
#ifdef _UART1
    UART1 = (UART_TypeDef *)UART1_BASE;
#endif /* _UART1 */

```

```

#ifdef _UART2
    UART2 = (UART_TypeDef *)UART2_BASE;
#endif /* _UART2 */
#ifdef _UART3
    UART3 = (UART_TypeDef *)UART3_BASE;
#endif /* _UART3 */

```

In debug mode, `_UART0`, `_UART1`, `_UART2` and `_UART3` must be defined, in the file `71x_conf.h`, to access the peripheral registers as follows:

```

#define _UART
#define _UART0
#define _UART1
#define _UART2
#define _UART3

```

The UART library use the `RCCU` and `PCU` library. The `_RCCU` and `_PCU` must be defined, in the `71x_conf.h`, to access the RCCU and PCU registers and library functions as follows:

```

#define _RCCU
#define _PCU

```

## FIFOs

The following enumeration defines the `UART` FIFOs. The `UARTFIFO_TypeDef` enumeration is defined in the `71x_uart.h` file:

```

typedef enum
{
    UART_RxFIFO,
    UART_TxFIFO
} UARTFIFO_TypeDef;

```

[Table 64](#) describes the `UART` FIFOs.

**Table 64. UART FIFOs**

| Regulator State | Description   |
|-----------------|---------------|
| UART_RxFIFO     | Receive FIFO  |
| UART_TxFIFO     | Transmit FIFO |

## UART stop bits

The following enumeration defines the `UART` Stop Bits selection. The `UARTStopBits_TypeDef` enumeration is declared in the `71x_uart.h` file:

```

typedef enum
{
    UART_0_5_StopBits = 0x0000,
    UART_1_StopBits   = 0x0008,
    UART_1_5_StopBits = 0x0010,
    UART_2_StopBits   = 0x0018
} UARTStopBits_TypeDef;

```

[Table 65](#) describes the *UART* Stop Bits.

**Table 65. UART stop bits**

| Stop Bits         | Description   |
|-------------------|---------------|
| UART_0_5_StopBits | 0.5 stop bits |
| UART_1_StopBits   | 1 stop bit    |
| UART_1_5_StopBits | 1.5 stop bits |
| UART_2_StopBits   | 2 stop bits   |

### UART parity modes

The following enumeration defines the *UART* parity modes. The *UARTParity\_TypeDef* enumeration is declared in the *71x\_uart.h* file:

```
typedef enum
{
    UART_EVEN_PARITY = 0x0000,
    UART_ODD_PARITY = 0x0020,
    UART_NO_PARITY = 0x0000,
} UARTParity_TypeDef;
```

[Table 66](#) describes the *UART* parity modes.

**Table 66. UART parity modes**

| Parity Mode      | Description      |
|------------------|------------------|
| UART_NO_PARITY   | No parity mode   |
| UART_ODD_PARITY  | Odd parity mode  |
| UART_EVEN_PARITY | Even parity mode |

### UART modes

The following enumeration defines the *UART* modes. *WFI\_CLOCKS* enumeration is declared in the *71x\_uart.h* file:

```
typedef enum
{
    UARTM_8D = 0x0001,
    UARTM_7D_P = 0x0003,
    UARTM_9D = 0x0004,
    UARTM_8D_W = 0x0005,
    UARTM_8D_P = 0x0007
} UARTMode_TypeDef;
```

[Table 67](#) lists the different *UART* modes.

**Table 67. UART modes**

| UART Modes | Description                      |
|------------|----------------------------------|
| UARTM_8D   | 8-bit data (without parity) mode |
| UARTM_7D_P | 7-bit data mode with parity      |
| UARTM_9D   | 9-bit data mode (without parity) |
| UARTM_8D_W | 8-bit data mode with wake-up bit |
| UARTM_8D_P | 8-bit data mode with parity      |

### Interrupt and status flags

The following definitions define the *UART* Interrupt and status flags. The interrupt and status flags are defined in the *71x\_uart.h* file:

```
#define UART_TxFull           0x0200
#define UART_RxHalfFull      0x0100
#define UART_TimeOutIdle     0x0080
#define UART_TimeOutNotEmpty 0x0040
#define UART_OverrunError    0x0020
#define UART_FrameError      0x0010
#define UART_ParityError      0x0008
#define UART_TxHalfEmpty     0x0004
#define UART_TxEmpty         0x0002
#define UART_RxBufNotEmpty   0x0001
```

[Table 68](#) describes the Interrupt and status flags.

**Table 68. Interrupt and status flags**

| Interrupt and Status Flags | Description                                 |
|----------------------------|---|
| UART_TxFull                | Transmit Buffer Full (Status register only) |
| UART_RxHalfFull            | Receiver buffer Half Full                   |
| UART_TimeOutIdle           | Time-out Idle                               |
| UART_TimeOutNotEmpty       | Time-out Not Empty                          |
| UART_OverrunError          | Overrun Error                               |
| UART_FrameError            | Framing Error                               |
| UART_ParityError           | Parity Error                                |
| UART_TxHalfEmpty           | Transmit buffer Half Empty                  |
| UART_TxEmpty               | Transmit buffer Empty                       |
| UART_RxBufNotEmpty         | Receive Buffer Not Empty                    |



## UARTx values

Table 69 shows the allowed values of *UARTx* variable.

**Table 69. UARTx values**

| UARTx | Description            |
|-------|------------------------|
| UART0 | To select <i>UART0</i> |
| UART1 | To select <i>UART1</i> |
| UART2 | To select <i>UART2</i> |
| UART3 | To select <i>UART3</i> |

### 3.11.2 Firmware library functions

Table 70 enumerates the different functions of the *UART* library.

**Table 70. UART library functions**

| Function Name                   | Description  |
|---------------------------------|--|
| <i>UART_Init</i>                | This routine is used to initialize the selected <i>UART</i>  |
| <i>UART_ModeConfig</i>          | This routine configures the <i>UART</i> mode   |
| <i>UART_BaudRateConfig</i>      | This routine configures the baud rate of the selected <i>UART</i>  |
| <i>UART_ParityConfig</i>        | This routine configures the <i>UART</i> parity mode  |
| <i>UART_StopBitsConfig</i>      | This routine configures the number of stop bits of the selected <i>UART</i>  |
| <i>UART_Config</i>              | This routine configures the Baud rate, parity mode, the number of stop bits and the <i>UART</i> mode of the selected <i>UART</i> |
| <i>UART_ItConfig</i>            | This routine enables/disables the interrupts sources of the selected <i>UART</i>   |
| <i>UART_FifoConfig</i>          | This routine enables/disables the FIFOs of the selected <i>UART</i>  |
| <i>UART_FifoReset</i>           | This routine resets the selected FIFO of the selected <i>UART</i>  |
| <i>UART_LoopBackConfig</i>      | This routine enables/disables loop back in the selected <i>UART</i>  |
| <i>UART_TimeOutPeriodConfig</i> | This routine configures the time-out Period of the selected <i>UART</i>  |
| <i>UART_GuardTimeConfig</i>     | This routine configures the guard time of the selected <i>UART</i>   |
| <i>UART_RxConfig</i>            | This routine enables/disables reception on the selected <i>UART</i>  |
| <i>UART_OnOffConfig</i>         | This turns the selected <i>UART</i> On / Off   |
| <i>UART_ByteSend</i>            | This routine sends a 7-bit byte or an 8-bit byte   |
| <i>UART_9BitByteSend</i>        | This routine sends a 9-bit byte  |
| <i>UART_DataSend</i>            | This routine sends several 7-bit bytes or 8-bit bytes  |
| <i>UART_9BitDataSend</i>        | This routine sends several 9-bit bytes   |
| <i>UART_StringSend</i>          | This routine sends a 7-bit or 8-bit string   |
| <i>UART_ByteReceive</i>         | This routine gets a 7 or an 8-bit byte from the selected <i>UART</i>   |
| <i>UART_9BitByteReceive</i>     | This routine gets a 9-bit byte from the selected <i>UART</i> .   |
| <i>UART_DataReceive</i>         | This routine gets several 7 or 8-bit bytes from the selected <i>UART</i>   |

Table 70. UART library functions (continued)

| Function Name                               | Description   |
|---|---|
| <a href="#"><i>UART_9BitDataReceive</i></a> | This routine gets several 9-bit bytes from the selected <i>UART</i>       |
| <a href="#"><i>UART_StringReceive</i></a>   | This routine gets several 7- or 8-bit bytes from the selected <i>UART</i> |
| <a href="#"><i>UART_FlagStatus</i></a>      | This routine return the status of the selected <i>UART</i>                |
| <a href="#"><i>SendChar</i></a>             | This routine sends a character to the defined <i>UART</i>                 |

**UART\_Init**

| Function Name          | UART_Init  |
|------------------------|--|
| Function Prototype     | <code>void UART_Init (UART_TypeDef *UARTx) ;</code>  |
| Behavior Description   | This function initializes the selected UART registers to their reset values.   |
| Input Parameter        | <i>UARTx</i> : selects the UART to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to initialize the *UART0*:

```

{
    ...
    UART_Init (UART0) ;
    ...
}

```

## UART\_ModeConfig

| Function Name          | UART_ModeConfig   |
|------------------------|---|
| Function Prototype     | <code>void UART_ModeConfig(UART_TypeDef *UARTx, UARTMode_TypeDef UART_Mode);</code>   |
| Behavior Description   | This routine configures the <i>UART</i> mode.   |
| Input Parameter 1      | <i>UARTx</i> : selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>UART_Mode</i> : selects the <i>UART</i> mode. Refer to <a href="#">UART modes on page 143</a> for details on the allowed values of this parameter.   |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to configure the *UART0* mode:

```
{
  /*Configure the UART0 mode to 8 bit data without parity */
  UART_ModeConfig(UART0, UARTEM_8D);
  ...
}
```

## UART\_BaudRateConfig

| Function Name          | UART_BaudRateConfig   |
|------------------------|---|
| Function Prototype     | <code>void UART_BaudRateConfig(UART_TypeDef *UARTx, u32 BaudRate);</code>   |
| Behavior Description   | This routine configures the baud rate of the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>BaudRate</i> : The baud rate value in bps  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | <i>RCCU_FrequencyValue(RCCU_PCLK1)</i> : gets the <i>APB1</i> frequency.  |

### Example:

This example illustrates how to configure the *UART0* baud rate:

```
{
  /*Set the UART0 baud rate to 9600 bps */
  UART_BaudRateConfig(UART0, 9600);
  ...
}
```

## UART\_ParityConfig

| Function Name          | UART_ParityConfig   |
|------------------------|---|
| Function Prototype     | <code>void UART_ParityConfig(UART_TypeDef *UARTx, UARTParity_TypeDef Parity);</code>  |
| Behavior Description   | This function configures the data parity of the selected UART.  |
| Input Parameter 1      | <i>UARTx</i> : the selected UART. <i>x</i> can be 0, 1, 2 or 3.<br>Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Parity</i> : the parity type. Refer to <a href="#">UART parity modes on page 143</a> for details on the allowed values of this parameter.                              |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to configure the *UART0* parity mode to Odd parity:

```
{
  /* Configure the UART0 parity mode to odd parity */
  UART_ParityConfig(UART0, UART_ODD_PARITY);
  ...
}
```

## UART\_StopBitsConfig

| Function Name          | UART_StopBitsConfig   |
|------------------------|---|
| Function Prototype     | <code>void UART_StopBitsConfig(UART_TypeDef *UARTx, UARTStopBits_TypeDef StopBits);</code>  |
| Behavior Description   | This routine configures the number of stop bits of the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : Selects the UART to be configured. <i>x</i> can be 0, 1, 2 or 3.<br>Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>StopBits</i> : Selects the number of the stop bits. Refer to <a href="#">UART stop bits on page 142</a> for details on the allowed values of this parameter.                           |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the *UART0* stop bits number to 1 stop bit:

```
{
  /*Configure the UART0 stop bits to 1 stop bit */
  UART_StopBitsConfig(UART0, UART_1_StopBits);
  ...
}
```

**UART\_Config**

| Function Name          | UART_Config  |
|------------------------|--|
| Function Prototype     | <pre>void UART_Config( UART_TypeDef *UARTx,   u32 BaudRate,   UARTParity_TypeDef Parity,   UARTStopBits_TypeDef StopBits,   UARTMode_TypeDef Mode );</pre>   |
| Behavior Description   | This function configures the Baud rate, parity mode, the number of stop bits and the UART mode of the selected UART.   |
| Input Parameter 1      | <i>UARTx</i> : the selected UART. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter.                                     |
| Input Parameter 2      | <i>BaudRate</i> : the baudrate value in bps.   |
| Input Parameter 3      | <i>Parity</i> : selects the parity type. Refer to <a href="#">UART parity modes on page 143</a> for details on the allowed values of this parameter.   |
| Input Parameter 4      | <i>StopBits</i> : selects the number of the stop bits. Refer to <a href="#">UART stop bits on page 142</a> for details on the allowed values of this parameter.  |
| Input Parameter 5      | <i>Mode</i> : selects the UART mode. Refer to <a href="#">UART modes on page 143</a> for details on the allowed values of this parameter.  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | <a href="#">UART_ModeConfig on page 147</a> ,<br><a href="#">UART_BaudRateConfig on page 147</a> ,<br><a href="#">UART_ParityConfig on page 148</a> ,<br><a href="#">UART_StopBitsConfig on page 148</a> . |

**Example:**

This example illustrates how to configure the *UART0*:

```
{
  /* Configure the UART0 as following:
  - Baudrate = 9600 Bps
  - No parity
  - 8 data bits
  - 1 stop bit */
  UART_Config(UART0, 9600, UART_NO_PARITY, UART_1_StopBits,
  UARTM_8D);
  ...
}
```

## UART\_ItConfig

| Function Name          | UART_ItConfig   |
|------------------------|---|
| Function Prototype     | void UART_ItConfig(UART_TypeDef *UARTx,<br>u16 UART_Flag, FunctionalState NewState);  |
| Behavior Description   | This function enables or disables one or several interrupt sources of the selected UART.  |
| Input Parameter 1      | UARTx: the selected UART. x can be 0, 1, 2 or 3.<br>Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter.                        |
| Input Parameter 2      | UART_Flag: selects one or several UART interrupt sources.<br>Refer to <a href="#">Interrupt and status flags on page 144</a> for details on the allowed values of this parameter. |
| Input Parameter 3      | NewStatus: specifies whether the <i>interrupt source</i> is enabled or disabled.<br>– ENABLE: enables the <i>IRQ</i> interrupt<br>– DISABLE: disables the <i>IRQ</i> interrupt.   |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Note:** All the UART flags are listed in the `71x_uart.h` file can be selected as interrupt source except the `UART_TxFull` flag which is not affected by this function.

### Example:

This example illustrates how to enable and disable the Receiver Buffer Full Interrupt sources in the UART:

```

{
    /*Enable Receiver Buffer Full interrupt */
    UART_ItConfig(UART0, UART_RxBufFull, ENABLE);
    ...
    /*Disable Receiver Buffer Full interrupt */
    UART_ItConfig(UART0, UART_RxBufFull, DISABLE);
    ...
}

```

## UART\_FifoConfig

| Function Name          | UART_FifoConfig  |
|------------------------|--|
| Function Prototype     | <code>void UART_FifoConfig(UART_TypeDef *UARTx, FunctionalState NewState);</code>  |
| Behavior Description   | This routine Enables or Disables the Rx and Tx FIFOs of the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : the selected <i>UART</i> . <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>NewStatus</i> : specifies whether the <i>FIFOs</i> are enabled or disabled.<br>– <i>ENABLE</i> : enables <i>FIFOs</i><br>– <i>DISABLE</i> : disables <i>FIFOs</i> .         |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to enable the *UART0* FIFOs:

```
{
    /*Enable the UART 0 FIFOs */
    UART_FifoConfig(UART0, ENABLE);
    ...
}
```

## UART\_FifoReset

| Function Name          | UART_FifoReset  |
|------------------------|---|
| Function Prototype     | <code>void UART_FifoReset(UART_TypeDef *UARTx, UARTFIFO_TypeDef FIFO);</code>   |
| Behavior Description   | This function resets the Rx and the Tx FIFOs of the selected <i>UART</i> ..   |
| Input Parameter 1      | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>FIFO</i> : Selects the <i>FIFO</i> to reset. Refer to <a href="#">FIFOs on page 142</a> for details on the allowed values of this parameter.   |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to reset the *UART0* receive FIFO:

```
{
  /*Reset the UART0 receive FIFO */
  UART_FifoReset(UART0, UART_RxFIFO);
  ...
}
```

**UART\_LoopBackConfig**

| Function Name          | UART_LoopBackConfig   |
|------------------------|---|
| Function Prototype     | void UART_LoopBackConfig(UART_TypeDef *UARTx, FunctionalState NewState);  |
| Behavior Description   | This function enables or disables the loop back mode of the selected UART.  |
| Input Parameter 1      | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>NewState</i> : specifies whether loop back is enabled or disabled.<br>– <i>ENABLE</i> : enables Loop back<br>– <i>DISABLE</i> : disables Loop back.  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to loop back in the *UART0*:

```
{
  /* Enable loop back on the UART 0 */
  UART_LoopBackConfig(UART0, ENABLE);
  ...
}
```



## UART\_TimeOutPeriodConfig

| Function Name          | UART_TimeOutPeriodConfig   |
|------------------------|--|
| Function Prototype     | <code>void UART_TimeOutPeriodConfig(UART_TypeDef *UARTx, u16 TimeOutPeriod);</code>  |
| Behavior Description   | This routine configures the time-out period of the selected <i>UART</i> .  |
| Input Parameter 1      | <i>UARTx</i> : the selected UART. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>TimeOutPeriod</i> : the time-out period value.  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to configure the *UART0* time out period:

```

{
    /*Configure the UART0 time-out period */
    UART_TimeOutPeriodConfig(UART0, 0xFF);
    ...
}

```

## UART\_GuardTimeConfig

| Function Name          | UART_GuardTimeConfig  |
|------------------------|---|
| Function Prototype     | <code>void UART_GuardTimeConfig(UART_TypeDef *UARTx, u16 GuardTime);</code>   |
| Behavior Description   | This routine configures the guard time of the selected <i>UART</i> .  |
| Input Parameter 1      | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>GuardTime</i> : the guard time value   |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to configure the *UART0* guard time:

```

{
    /* Configure the guard time */
    UART_GuardTimeConfig(UART0, 0xFF);
    ...
}

```

**UART\_RxConfig**

| Function Name          | UART_RxConfig   |
|------------------------|---|
| Function Prototype     | <code>void UART_RxConfig(UART_TypeDef *UARTx, FunctionalState NewState);</code>   |
| Behavior Description   | This function enables or disables the selected UART data reception.   |
| Input Parameter 1      | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>NewState</i> : specifies whether the reception is enabled or disabled.<br>– <i>ENABLE</i> : enables reception<br>– <i>DISABLE</i> : disables reception.                                    |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable reception on the *UART0*:

```

{
    /* Enable reception */
    UART_RxConfig(UART0, ENABLE);
    ...
}

```

**UART\_OnOffConfig**

| Function Name          | UART_OnOffConfig  |
|------------------------|---|
| Function Prototype     | <code>void UART_OnOffConfig(UART_TypeDef *UARTx, FunctionalState NewState);</code>  |
| Behavior Description   | This routine turns On / Off the selected UART.  |
| Input Parameter 1      | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>NewState</i> : specifies whether the <i>UART</i> is enabled or disabled.<br>– <i>ENABLE</i> : enables the <i>UART</i><br>– <i>DISABLE</i> : disables the <i>UART</i> .                     |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to turn the UART0 On:

```
{
  /*Turn the UART On */
  UART_OnOffConfig(UART0, ENABLE);
  ...
}
```

**UART\_ByteSend**

| Function Name          | UART_ByteSend  |
|------------------------|--|
| Function Prototype     | void UART_ByteSend(UART_TypeDef *UARTx, u8 *Data);   |
| Behavior Description   | This function sends a 7-bit byte or an 8-bit byte using the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : Selects the UART to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : a pointer on the data byte to send.  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled. See <i>UART_Config</i> and <i>UART_OnOffConfig</i> functions.  |
| Called Functions       | None   |

**Example:**

This example illustrates how to send an 8-bit byte using the *UART0*:

```
{
  u8 Data = 0x55;
  UART_ByteSend(UART0, &Data);
}
```

## UART\_9BitByteSend

| Function Name          | UART_9BitByteSend  |
|------------------------|--|
| Function Prototype     | <code>void UART_9BitByteSend(UART_TypeDef *UARTx, u16 *Data);</code>   |
| Behavior Description   | This function sends a 9-bit data using the selected <i>UART</i> .  |
| Input Parameter 1      | <i>UARTx</i> : the selected UART. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : a pointer on the 9-bit data to send.   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.  |
| Called Functions       | None   |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>  |

### Example:

This example illustrates how to send a 9-bit byte using the *UART0*:

```
{
    u16 Data = 0x0155;
    UART_9BitByteSend(UART0, &Data);
}
```

## UART\_DataSend

| Function Name          | UART_DataSend  |
|------------------------|--|
| Function Prototype     | <code>void UART_DataSend(UART_TypeDef *UARTx, u8 *Data, u8 DataLength);</code>   |
| Behavior Description   | This routine sends several 7-bit bytes or 8-bit bytes using the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : the selected UART. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : The bytes start address.   |
| Input Parameter 3      | <i>DataLength</i> : The data bytes number.   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.  |
| Called Functions       | None   |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>  |

**Example:**

This example illustrates how to send several 8-bit bytes using the *UART0*:

```

{
  #define DATA_LENGTH 16
  u8 pDataToSend[DATA_LENGTH] = {'A', 'B', 'C', 'D', 'E', 'F',
  'G', 'H',
                                'I', 'J', 'K', 'L', 'M', 'N',
  'O', 'P'};
  UART_DataSend(UART0, pDataToSend, DATA_LENGTH);
  ...
}

```

**UART\_9BitDataSend**

| Function Name          | UART_9BitDataSend  |
|------------------------|--|
| Function Prototype     | void UART_9BitDataSend(UART_TypeDef *UARTx, u16 *Data, u8 DataLength);   |
| Behavior Description   | This function sends several 9-bit data using the selected <i>UART</i> .  |
| Input Parameter 1      | <i>UARTx</i> : the selected <i>UART</i> . <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : the bytes start address.   |
| Input Parameter 3      | <i>DataLength</i> : the data bytes number.   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.  |
| Called Functions       | None   |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>  |

**Example:**

This example illustrates how to send several 9-bit bytes using the *UART0*:

```

{
  #define DATA_LENGTH 16
  u16 pDataToSend[DATA_LENGTH] = {0x0101, 0x0002, 0x0103,
  0x0004,
                                0x0105, 0x0006, 0x0107, 0x0008,
                                0x0109, 0x000A, 0x010B, 0x000C,
                                0x010D, 0x000E, 0x010F, 0x000F};
  UART_9BitDataSend(UART0, pDataToSend, DATA_LENGTH);
  ...
}

```

## UART\_StringSend

| Function Name          | UART_StringSend   |
|------------------------|---|
| Function Prototype     | <code>void UART_StringSend(UART_TypeDef *UARTx, u8 *String);</code>   |
| Behavior Description   | This routine sends a 7-bit byte or 8-bit string using the selected <i>UART</i> .  |
| Input Parameter 1      | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>String</i> : The string start address.   |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.   |
| Called Functions       | UART_ByteSend   |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>   |

### Example:

This example illustrates how to send a string using the *UART0*:

```
{
    /* Send "Hello World" */
    UART_StringSend(UART0, (u8 *)&"Hello World");
    ...
}
```

## UART\_ByteReceive

| Function Name          | UART_ByteReceive   |
|------------------------|--|
| Function Prototype     | <code>u16 UART_ByteReceive(UART_TypeDef *UARTx, u8 *Data, u8 TimeOut);</code>  |
| Behavior Description   | This routine gets a 7 or an 8-bit byte from the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : the selected <i>UART</i> . <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : a pointer on the data where the data will be stored  |
| Input Parameter 3      | <i>TimeOut</i> : The Time-out value.   |
| Output Parameter       | The received 8-bit data.   |
| Return Value           | The <i>UARTx_SR</i> register content before reading the received data.   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.  |
| Called Functions       | None.  |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>  |

**Example:**

This example illustrates how to read the received data on the UART0:

```
{
  u8 bRxData;
  u16 wUART_Status;
  ...
  /*Read the received data */
  wUART_Status = UART_ByteReceive(UART0, &bRxData, 0xFF);
  /* Check if data received without error */
  if (wUART_Status & ( UART_ParityError |
                      UART_FrameError |
                      UART_OverrunError |
                      UART_TimeOutIdle ))
  {
    /* handles the error */
    ...
  }
  ...
  while(!(UART_FlagStatus(UART0) & UART_RxBufFull)); /* while
the receive                                     buffer is
empty */
  wUART_Status = UART_ByteReceive(UART0, &bRxData, 0xFF);
  ...
}
```

**UART\_9BitByteReceive**

| Function Name          | UART_9BitByteReceive   |
|------------------------|--|
| Function Prototype     | u16 UART_9BitByteReceive(UART_TypeDef *UARTx, u16 *Data, u8 TimeOut);  |
| Behavior Description   | This routine gets a 9-bit byte from the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : the selected UART. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : A pointer to the received data.  |
| Input Parameter 3      | <i>TimeOut</i> : The Time-out value.   |
| Output Parameter       | The received 9-bit data.   |
| Return Value           | the UARTx_SR register contents before reading the received data.   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.  |
| Called Functions       | None.  |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>  |

**Example:**

This example illustrates how to read the received data on the *UART0*:

```

{
    u16 wRxData;
    u16 wUART_Status;
    ...
    /*Read the received data */
    wUART_Status=UART_9BitByteReceive(UART0, &wRxData, 0xFF);
    /* Check if data received without error */
    if (wUART_Status & ( UART_ParityError |
                        UART_FrameError |
                        UART_OverrunError |
                        UART_TimeOutIdle ))
    {
        /* handles the error */
        ...
    }
    ...
    while(!(UART_FlagStatus(UART0) & UART_RxBufFull)); /* while
the receive                                     buffer is
empty */
    wUART_Status = UART_9BitByteReceive(UART0, &wRxData, 0xFF);
    ...
}

```



## UART\_DataReceive

| Function Name          | UART_DataReceive  |
|------------------------|---|
| Function Prototype     | u16 UART_DataReceive(UART_TypeDef *UARTx, u8 *Data, u8 DataLength, u8 TimeOut);   |
| Behavior Description   | Gets several 7 or 8-bit bytes from the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : the selected UART. x can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : A pointer to the received data.   |
| Input Parameter 3      | <i>DataLength</i> : The number of bytes to receive.   |
| Input Parameter 4      | <i>TimeOut</i> : The Time-out value.  |
| Output Parameter       | The received 8-bit data buffer  |
| Return Value           | The UARTx_SR register content before reading the received data.   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.   |
| Called Functions       | <a href="#">UART_ByteReceive</a>  |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>   |

### Example:

This example illustrates how to read several data from the *UART0*:

```

{
    #define DataLength 16
    u8 bRxData[DataLength];
    u16 wUART_Status;
    ...
    //Read the received data
    u16 UART_DataReceive(UART0, bRxData, DataLength, 0xff);
    /* Check if data received without error */
    if (wUART_Status & ( UART_ParityError |
                        UART_FrameError |
                        UART_OverrunError |
                        UART_TimeOutIdle ))
    {
        /* handles the error */
        ...
    }
}

```

**UART\_9BitDataReceive**

| Function Name          | UART_9BitDataReceive  |
|------------------------|---|
| Function Prototype     | u16 UART_9BitDataReceive(UART_TypeDef *UARTx, u16 *Data, u8 DataLength, u8 TimeOut);  |
| Behavior Description   | This routine gets several 9-bit bytes from the selected <i>UART</i> .   |
| Input Parameter 1      | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter 2      | <i>Data</i> : a pointer on the buffer where the data will be stored.  |
| Input Parameter 3      | <i>DataLength</i> : the number of bytes to receive.   |
| Input Parameter 4      | <i>TimeOut</i> : the time-out period value.   |
| Output Parameter       | The received 9-bit data buffer.   |
| Return Value           | The status register value before reading the received data.   |
| Required preconditions | The selected <i>UART</i> must be properly configured and enabled.   |
| Called Functions       | <a href="#">UART_9BitByteReceive</a>  |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>   |

**Example:**

This example illustrates how to read several 9-bit bytes from the *UART0*:

```

{
    #define DataLength 16
    u16 wRxData[DataLength];
    u16 wUART_Status;
    ...
    //Read the received data
    u16 UART_9BitDataReceive(UART0, wRxData, DataLength, 0xff);
    /*Check if data received without error */
    if (wUART_Status & ( UART_ParityError |
                        UART_FrameError |
                        UART_OverrunError |
                        UART_TimeOutIdle ))
    {
        /* handles the error */
        ...
    }
}

```

## UART\_StringReceive

| Function Name          | UART_StringReceive  |
|------------------------|---|
| Function Prototype     | <code>u16 UART_StringReceive (UART_TypeDef *UARTx,<br/>u8 *Data);</code>  |
| Behavior Description   | This function gets a string which ends with: End of string or Carriage return characters from the selected UART.  |
| Input Parameter1       | <i>UARTx</i> : Selects the <i>UART</i> to be configured. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Input Parameter2       | <i>Data</i> : a pointer on the buffer where the data will be stored.  |
| Output Parameter       | The received string.  |
| Return Value           | The <code>UARTx_SR</code> register contents before reading the received data.   |
| Required preconditions | The selected UART must be properly configured and enabled.  |
| Called Functions       | None.   |
| See also               | <a href="#">UART_Config on page 149</a> ,<br><a href="#">UART_OnOffConfig on page 154</a>   |

### Example:

This example illustrates how to get a string from the *UART0*:

```

{
    u8 bRxData[10];
    u16 wUART_Status;
    ...
    /* Read the received data */
    UART_StringReceive(UART0, &bRxData);
    /* Check if data received without error */
    if (wUART_Status & ( UART_ParityError |
                        UART_FrameError |
                        UART_OverrunError |
                        UART_TimeOutIdle ))
    {
        /* handles error */
        ...
    }
}

```

## UART\_FlagStatus

| Function Name          | UART_FlagStatus  |
|------------------------|--|
| Function Prototype     | <code>u16 UART_FlagStatus(UART_TypeDef *UARTx);</code>   |
| Behavior Description   | This routine returns the UARTx_SR register content of the selected UART.   |
| Input Parameter        | <i>UARTx</i> : the selected UART. <i>x</i> can be 0, 1, 2 or 3. Refer to <a href="#">UARTx values on page 145</a> for details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | The status register content of the selected <i>UART</i> .  |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to get the status of the *UART0*:

```
{
    u16 wUART_Status;
    wUART_Status = UART_FlagStatus(UART0);
}
```

## SendChar

| Function Name          | SendChar  |
|------------------------|---|
| Function Prototype     | <code>void SendChar(u8 *ch)</code>  |
| Behavior Description   | This function sends a character to the defined UART.  |
| Input Parameter        | <i>ch</i> : a pointer to the character to send.   |
| Output Parameter       | None  |
| Return Value           | None.   |
| Required preconditions | Define <code>USE_SERIAL_PORT</code> in <code>71x_conf.h</code> file:<br><code>#define USE_SERIAL_PORT</code><br>Then define the UART to be used: ex: <code>#define USE_UART0</code> |
| Called Functions       | <code>UART_ByteSend</code>  |

### Example:

This example illustrates how to send a char with the `SendChar` function:

```
in 71x_conf.h file define:
#define USE_SERIAL_PORT
#define USE_UART0

{
    char c;
    c='A';
    wUART_Status = SendChar((u8 *)c);
}
```

## 3.12 Inter-integrated circuit (I<sup>2</sup>C)

The I<sup>2</sup>C Bus interface module serves as interface between the microcontroller and the serial I<sup>2</sup>C bus. It provides both multi-master and slave functions, and controls all I<sup>2</sup>C bus-specific sequencing, protocol, arbitration and timing.

The I<sup>2</sup>C driver may be used to manage the I<sup>2</sup>C functionality in transmission and reception and it report the status of the action done.

The first section describes the data structures used in the I<sup>2</sup>C firmware library. The second one presents the firmware library functions.

### 3.12.1 Data structures

#### I<sup>2</sup>C register structure

The I<sup>2</sup>C registers structure *I2C\_TypeDef* is defined in the *71x\_map.h* file as follows:

```
typedef volatile struct
{
    vu8  CR;
    vu8  EMPTY1 [3];
    vu8  SR1;
    vu8  EMPTY2 [3];
    vu8  SR2;
    vu8  EMPTY3 [3];
    vu8  CCR;
    vu8  EMPTY4 [3];
    vu8  OAR1;
    vu8  EMPTY5 [3];
    vu8  OAR2;
    vu8  EMPTY6 [3];
    vu8  DR;
    vu8  EMPTY7 [3];
    vu8  ECCR;
    vu8  EMPTY8 [3];
} I2C_TypeDef;
```

[Table 71](#) presents the I<sup>2</sup>C registers.

**Table 71. I<sup>2</sup>C registers**

| Register | Description                                      |
|----------|--|
| CR       | I <sup>2</sup> C Control Register                |
| SR1      | I <sup>2</sup> C Status Register1                |
| SR2      | I <sup>2</sup> C Status Register2                |
| CCR      | I <sup>2</sup> C Clock Control Register          |
| ECCR     | I <sup>2</sup> C Extended Clock Control Register |
| OAR1     | I <sup>2</sup> C Own Address Register1           |
| OAR2     | I <sup>2</sup> C Own Address Register2           |
| DR       | I <sup>2</sup> C Data Register                   |

The two  $I^2C$  interfaces are declared in the same file:

```

...
#define APB1_BASE      0xC0000000
....
#define I2C0_BASE      (APB1_BASE + 0x1000)
#define I2C1_BASE      (APB1_BASE + 0x2000)
#ifndef DEBUG
...
#ifdef _I2C0
#define I2C0            ((I2C_TypeDef *)I2C0_BASE)
#endif /*I2C0*/

#ifdef _I2C1
#define I2C1            ((I2C_TypeDef *)I2C1_BASE)
#endif /*I2C1*/
...
#else /* DEBUG */
...
#ifdef _I2C0
EXT I2C_TypeDef        *I2C0;
#endif /*I2C0*/

#ifdef _I2C1
EXT I2C_TypeDef        *I2C1;
#endif /*I2C1*/
...
#endif

```

When debug mode is used,  $I^2C$  pointer is initialized in *71x\_lib.c* file:

```

#ifdef _I2C0
I2C0 = (I2C_TypeDef *)I2C0_BASE;
#endif /* _I2C0 */
#ifdef _I2C1
I2C1 = (I2C_TypeDef *)I2C1_BASE;
#endif /* _I2C1 */

```

`_I2C0` and `_I2C1` must be defined, in *71x\_conf.h* file, to access the peripheral registers as follows:

```

#define _I2C
#define _I2C0
#define _I2C1
...

```

When RCCU functions are called, `_RCCU` must be defined, in *71x\_conf.h* file, to make the *RCCU* functions accessible:

```

#define _RCCU

```

## $I^2C$ registers

The following enumeration defines the registers of the  $I^2C$  and their offset. *I2C\_Registers* enumeration is defined in the file *71x\_i2c.h* as follows:

```

typedef enum
{
    I2C_CR      = 0x00,

```

```

I2C_SR1    = 0x04,
I2C_SR2    = 0x08,
I2C_CCR    = 0x0C,
I2C_OAR1   = 0x10,
I2C_OAR2   = 0x14,
I2C_DR     = 0x18,
I2C_ECCR   = 0x1C
} I2C_Registers;

```

## I<sup>2</sup>C addressing modes

The following enumeration defines the I<sup>2</sup>C addressing mode. The *I2C\_Addressing* enumeration is defined in the file *71x\_i2c.h* as follows:

```

typedef enum
{
    I2C_Mode10,
    I2C_Mode7
} I2C_Addressing;

```

[Table 72](#) defines the I<sup>2</sup>C addressing modes.

**Table 72. I<sup>2</sup>C addressing modes**

| Addressing mode | Description            |
|-----------------|------------------------|
| I2C_Mode10      | 10-bit addressing mode |
| I2C_Mode7       | 7-bit addressing mode  |

## I<sup>2</sup>C transfer direction

The following enumeration defines the I<sup>2</sup>C transfer direction. The *I2C\_Direction* enumeration is defined in the file *71x\_i2c.h* as follows:

```

typedef enum
{
    I2C_RX,
    I2C_TX
} I2C_Direction;

```

[Table 73](#) defines the I<sup>2</sup>C transfer direction.

**Table 73. I<sup>2</sup>C transfer direction**

| I <sup>2</sup> C direction | Description  |
|----------------------------|--------------|
| I2C_TX                     | Transmission |
| I2C_RX                     | Reception    |

## I<sup>2</sup>C flags

The following definitions defines the I<sup>2</sup>C flags. *I2C flags* are defined in the file *71x\_i2c.h*:

```

#define I2C_SB                0x00001
#define I2C_M_SL              0x00002

```

```

#define I2C_ADSL                0x00004
#define I2C_BTF                 0x00008
#define I2C_BUSY                0x00010
#define I2C_TRA                 0x00020
#define I2C_ADD10              0x00040
#define I2C_EVF                 0x00080
#define I2C_GCAL                0x00100
#define I2C_BERR                0x00200
#define I2C_ARLO                0x00400
#define I2C_STOPF              0x00800
#define I2C_AF                  0x01000
#define I2C_ENDAD               0x02000
#define I2C_STOP                0x08000
#define I2C_ACK                 0x10000
#define I2C_START               0x20000

#define I2C_PESET_Mask          0x20
#define I2C_PERESET             0xDF
#define I2C_ENGC_Mask           0x10
#define I2C_START_Mask          0x08
#define I2C_STOP_Mask           0x02
#define I2C_ACK_Mask            0x04
#define I2C_ITE_Mask            0x01
#define I2C_Event_Mask          0x3FFF

```

[Table 74](#) describes I<sup>2</sup>C flags.

**Table 74. I<sup>2</sup>C flags**

| Flag   | Flag Code |
|--|-----------|
| I2C_SB: Start (Master mode) flag             | 0x00001   |
| I2C_M_SL: Master/Slave flag.                 | 0x00002   |
| I2C_ADSL: Address matched (Slave mode) flag. | 0x00004   |
| I2C_BTF: Byte transfer finished flag.        | 0x00008   |
| I2C_BUSY: Bus busy flag.                     | 0x00010   |
| I2C_TRA: Transmitter/Receiver flag.          | 0x00020   |
| I2C_ADD10: 10-Bit addressing in master mode  | 0x00040   |
| I2C_EVF: Event flag                          | 0x00080   |
| I2C_GCAL: General call (slave mode) flag     | 0x00100   |
| I2C_BERR: Bus error flag                     | 0x00200   |
| I2C_ARLO: Arbitration lost flag              | 0x00400   |
| I2C_STOPF: Stop detection (slave mode)       | 0x00800   |
| I2C_AF: Acknowledge failure flag             | 0x01000   |
| I2C_ENDAD: End of address transmission       | 0x02000   |
| I2C_STOP: Generation of a Stop condition     | 0x08000   |
| I2C_ACK: Acknowledge enable.                 | 0x10000   |
| I2C_START: Generation of a Start condition   | 0x20000   |



## I<sup>2</sup>C events

```
#define I2C_EVENT_SLAVE_ADDRESS_MATCHED
#define I2C_EVENT_SLAVE_BYTE_RECEIVED
#define I2C_EVENT_SLAVE_BYTE_TRANSMITTED
#define I2C_EVENT_MASTER_MODE_SELECT
#define I2C_EVENT_MASTER_MODE_SELECTED
#define I2C_EVENT_MASTER_BYTE_RECEIVED
#define I2C_EVENT_MASTER_BYTE_TRANSMITTED
#define I2C_EVENT_MASTER_MODE_ADDRESS10
#define I2C_EVENT_SLAVE_STOP_DETECTED
#define I2C_EVENT_SLAVE_ACK_FAILURE
#define I2C_BUS_ERROR_DETECTED
#define I2C_ARBITRATION_LOST
#define I2C_SLAVE_GENERAL_CALL
```

## I<sup>2</sup>C transmission status

The following enumeration defines the different types of message returned to define the status of transmission. The *I2C\_Tx\_Status* enumeration is declared in the file *71x\_i2c.h*

```
typedef enum {
    I2C_TX_NO,
    I2C_TX_SB,
    I2C_TX_AF,
    I2C_TX_ARLO,
    I2C_TX_BERR,
    I2C_TX_ADD_OK,
    I2C_TX_DATA_OK,
    I2C_TX_ONGOING
} I2C_Tx_Status;
```

[Table 75](#) describes I<sup>2</sup>C transmission status messages.

**Table 75. I<sup>2</sup>C transmission status messages**

| I2C_Tx_Status  | Description   |
|----------------|---|
| I2C_TX_NO      | No start condition is generated or detected so no transmission is occurring                       |
| I2C_TX_SB      | START condition is generated  |
| I2C_TX_AF      | Acknowledge Failure is detected   |
| I2C_TX_ARLO    | Arbitration Lost error is detected in multimaster mode  |
| I2C_TX_BERR    | Bus Error is detected   |
| I2C_TX_ADD_OK  | Slave address transmission is correctly completed in master mode or address matched in slave mode |
| I2C_TX_DATA_OK | Last transmitted data is correctly completed  |
| I2C_TX_ONGOING | The current transmission is occurring   |

## I<sup>2</sup>C reception status

The following enumeration defines the different types of message returned to define the status of the reception. The *I2C\_Rx\_Status* enumeration is declared in the file *71x\_i2c.h*

```
typedef enum {
    I2C_RX_NO,
    I2C_RX_SB,
    I2C_RX_AF,
    I2C_RX_ARLO,
    I2C_RX_BERR,
    I2C_RX_ADD_OK,
    I2C_RX_DATA_OK,
    I2C_RX_ONGOING
} I2C_Rx_Status;
```

[Table 76](#) describes I<sup>2</sup>C reception status messages.

**Table 76. I<sup>2</sup>C reception status messages**

| I2C_Rx_Status  | Description  |
|----------------|--|
| I2C_Rx_NO      | No start condition is generated or detected so no reception is occurring |
| I2C_Rx_SB      | START condition is generated   |
| I2C_Rx_AF      | Acknowledge Failure is detected  |
| I2C_Rx_ARLO    | Arbitration Lost error is detected in multimaster mode                   |
| I2C_Rx_BERR    | Bus Error is detected  |
| I2C_Rx_ADD_OK  | Slave address transmission is correctly completed                        |
| I2C_Rx_DATA_OK | Last Received data is correctly completed                                |
| I2C_Rx_ONGOING | The current reception is occurring                                       |

### 3.12.2 Firmware library functions

[Table 77](#) enumerates the different functions of the I<sup>2</sup>C library.

**Table 77. I<sup>2</sup>C library functions**

| Function Name                | Description  |
|------------------------------|--|
| <i>I2C_Init</i>              | Initializes I <sup>2</sup> C peripheral control and registers to their default reset values. |
| <i>I2C_OnOffConfig</i>       | Enables or disables I <sup>2</sup> C peripheral.   |
| <i>I2C_GeneralCallConfig</i> | Enables or disables I <sup>2</sup> C general call option.                                    |
| <i>I2C_STARTGenerate</i>     | Generates I <sup>2</sup> C communication START condition.                                    |
| <i>I2C_STOPGenerate</i>      | Generates I <sup>2</sup> C communication STOP condition.                                     |
| <i>I2C_AcknowledgeConfig</i> | Enables or disables I <sup>2</sup> C acknowledge feature.                                    |
| <i>I2C_ITConfig</i>          | Enables or disables I <sup>2</sup> C interrupt feature.                                      |
| <i>I2C_RegisterRead</i>      | Reads any I <sup>2</sup> C register and returns its value.                                   |
| <i>I2C_FlagStatus</i>        | Checks whether any I <sup>2</sup> C Flag is set or not.                                      |
| <i>I2C_FlagClear</i>         | Clears any I <sup>2</sup> C Flag.  |

**Table 77. I<sup>2</sup>C library functions (continued)**

| Function Name                 | Description   |
|-------------------------------|---|
| <i>I2C_SpeedConfig</i>        | Selects I <sup>2</sup> C clock speed and configures its corresponding mode.                                     |
| <i>I2C_AddressConfig</i>      | Defines the I <sup>2</sup> C bus address of the interface.  |
| <i>I2C_FCLKConfig</i>         | Configures frequency bits according to RCLK frequency.  |
| <i>I2C_AddressSend</i>        | Transmits the address byte to select the slave device.  |
| <i>I2C_ByteSend</i>           | Transmits single byte of data.  |
| <i>I2C_TransmissionStatus</i> | Checks for any error during transmission and returns the error status. It also checks for any pending requests. |
| <i>I2C_ByteReceive</i>        | Returns the most recently received byte.  |
| <i>I2C_ReceptionStatus</i>    | Checks the completion of reception and returns the reception status.  |
| <i>I2C_ErrorClear</i>         | Clears any error flags.   |
| <i>I2C_GetStatus</i>          | Reads the status registers of the specified I2C   |
| <i>I2C_GetLastEvent</i>       | Gets the last I2C event that has occurred.  |
| <i>I2C_CheckEvent</i>         | Checks whether the last I2C event is equal to the one passed as parameter.                                      |

**I2C\_Init**

| Function Name          | I2C_Init   |
|------------------------|--|
| Function Prototype     | <code>void I2C_Init (I2C_TypeDef *I2Cx);</code>  |
| Behavior Description   | Initializes I <sup>2</sup> C peripheral control and registers to their default reset values. |
| Input Parameter        | <i>I2Cx</i> : specifies the I2C to be initialized where x can be 0 or 1.                     |
| Output Parameters      | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to initialize the *I2C0* register to their reset value:

```

{
    ...
    I2C_Init (I2C0);
    ...
}

```

**I2C\_OnOffConfig**

| Function Name          | I2C_OnOffConfig  |
|------------------------|--|
| Function Prototype     | <code>void I2C_OnOffConfig (I2C_TypeDef *I2Cx, FunctionalState NewState);</code>   |
| Behavior Description   | Enables or disables the I2C peripheral.  |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to be enabled or disabled where x can be 0 or 1.   |
| Input Parameter 2      | <i>NewState</i> : specifies the new status to set<br>– <i>ENABLE</i> : enables the I <sup>2</sup> C peripheral.<br>– <i>DISABLE</i> : disables the I <sup>2</sup> C peripheral |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | I/O ports should be configured in their reset states.  |
| Called Functions       | None   |

**Example:**

This example illustrates how to enable and disable the *I2C0* interface:

```

{
    ...
    I2C_OnOffConfig (I2C0, ENABLE);
    ...
    /*user code */
    ...
    I2C_OnOffConfig (I2C0, DISABLE);
    ...
}

```

**I2C\_GeneralCallConfig**

| Function Name          | I2C_GeneralCallConfig   |
|------------------------|---|
| Function Prototype     | <code>void I2C_GeneralCallConfig (I2C_TypeDef *I2Cx, FunctionalState NewState);</code>  |
| Behavior Description   | Enables or disables I2C general call option.  |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to be configured where x can be 0 or 1.   |
| Input Parameter 2      | <i>NewState</i> : specifies the new status to set<br>– <i>ENABLE</i> : enables the I <sup>2</sup> C general call option<br>– <i>DISABLE</i> : disables the I <sup>2</sup> C general call option |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | I <sup>2</sup> Cx peripheral should be enabled.   |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable the General Call option:

```

{
  ...
  I2C_OnOffConfig (I2C0, ENABLE);
  ...
  I2C_GeneralCallConfig (I2C0, ENABLE);
  ...
}

```

**I2C\_STARTGenerate**

| Function Name          | I2C_STARTGenerate   |
|------------------------|---|
| Function Prototype     | void I2C_STARTGenerate (I2C_TypeDef *I2Cx, FunctionalState NewState);   |
| Behavior Description   | Enables or disables $I^2C$ start generation.  |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to be configured where x can be 0 or 1.   |
| Input Parameter 2      | <i>NewState</i> : specifies the new status to set<br>– <i>ENABLE</i> : enables the $I^2C$ start generation<br>– <i>DISABLE</i> : disables the $I^2C$ start generation |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | $I^2C_x$ peripheral should be enabled.  |
| Called Functions       | None  |

**Example:**

This example illustrates how to enable the start generation in the *I2C0* peripheral:

```

{
  ...
  I2C_OnOffConfig (I2C0, ENABLE);
  ...
  I2C_STARTGenerate (I2C0, ENABLE);
  ...
}

```

**I2C\_STOPGenerate**

| Function Name          | I2C_STOPGenerate   |
|------------------------|--|
| Function Prototype     | void I2C_STOPGenerate (I2C_TypeDef *I2Cx, FunctionalState NewState);   |
| Behavior Description   | Enables or disables I <sup>2</sup> C stop generation.  |
| Input Parameter 1      | I2Cx: specifies the I2C to be configured where x can be 0 or 1.  |
| Input Parameter 2      | NewState: specifies the new status to set:<br>– ENABLE: enables the I <sup>2</sup> C stop generation<br>– DISABLE: disables the I <sup>2</sup> C stop generation |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | I <sup>2</sup> Cx peripheral should be enabled.  |
| Called Functions       | None   |

**Example:**

This example illustrates how to enable the STOP Generation in the I2C0 peripheral:

```

{
    ...
    I2C_OnOffConfig (I2C0, ENABLE);
    ...
    I2C_STOPGenerate (I2C0, ENABLE);
    ...
}

```

**I2C\_AcknowledgeConfig**

| Function Name          | I2C_AcknowledgeConfig  |
|------------------------|--|
| Function Prototype     | void I2C_AcknowledgeConfig (I2C_TypeDef *I2Cx, FunctionalState NewState);  |
| Behavior Description   | Enables or disables I <sup>2</sup> C acknowledge feature.  |
| Input Parameter 1      | I2Cx: specifies the I2C to be configured where x can be 0 or 1.  |
| Input Parameter 2      | NewState: specifies the new status to set:<br>– ENABLE: enables the I <sup>2</sup> C acknowledge feature<br>– DISABLE: disables the I <sup>2</sup> C acknowledge feature |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | I <sup>2</sup> Cx peripheral should be enabled.  |
| Called Functions       | None   |

**Example:**

This example illustrates how to enable the acknowledge feature in the I2C0 peripheral:

```

{
    ...
    I2C_OnOffConfig (I2C0, ENABLE);
    ...
    I2C_AcknowledgeConfig (I2C0, ENABLE);
    ...
}

```

## I2C\_ITConfig

| Function Name          | I2C_ITConfig   |
|------------------------|--|
| Function Prototype     | <code>void I2C_ITConfig (I2C_TypeDef *I2Cx, FunctionalState NewState);</code>  |
| Behavior Description   | Enables or disables $I^2C$ interrupt feature.  |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to be configured where x can be 0 or 1.  |
| Input Parameter 2      | <i>NewState</i> : specifies the new status to set:<br>– <i>ENABLE</i> : enables the $I^2C$ interrupt feature<br>– <i>DISABLE</i> : disables the $I^2C$ interrupt feature |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | $I^2Cx$ peripheral should be enabled.  |
| Called Functions       | None   |

### Example:

This example illustrates how to enable the interrupt feature in the *I2C0* peripheral:

```

{
    ...
    I2C_OnOffConfig (I2C0, ENABLE);
    ...
    I2C_ITConfig (I2C0, ENABLE);
    ...
}

```

## I2C\_RegisterRead

| Function Name          | I2C_RegisterRead  |
|------------------------|---|
| Function Prototype     | <code>u8 I2C_RegisterRead (I2C_TypeDef *I2Cx, I2C_Registers reg);</code>  |
| Behavior Description   | Reads any I2C register and returns its value.   |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to read its register where x can be 0 or 1.   |
| Input Parameter 2      | <i>reg</i> : specifies the register to read. For more details on the allowed values of this parameter, refer to <a href="#">I2C registers on page 166</a> . |
| Output Parameter       | None  |
| Return Value           | The value of the register passed as parameter   |
| Required Preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to get a register value:

```

{
    vu8 Register_Value;
    ...
    Register_Value = I2C_RegisterRead (I2C0, I2C_CCR);
    ...
}

```

**I2C\_FlagStatus**

| Function Name          | I2C_FlagStatus   |
|------------------------|--|
| Function Prototype     | FlagStatus I2C_FlagStatus (I2C_TypeDef *I2Cx, u32 Flag);   |
| Behavior Description   | Checks whether the $I^2C$ flag is set or not.  |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to read its flags where <i>x</i> can be 0 or 1.  |
| Input Parameter 2      | <i>Flag</i> : specifies the flag to check. Refer to <a href="#">I2C flags on page 167</a> for more details on allowed values of <i>Flag</i> parameter. |
| Output Parameter       | None   |
| Return Value           | The specified flag status:<br>– <i>SET</i> : if the tested flag is set.<br>– <i>RESET</i> : if the tested flag is reset.                               |
| Required Preconditions | None   |
| Called Functions       | None.  |

**Example:**

This example illustrates how to get flag status:

```

{
  u32 I2CStatus;

  do
  {
    I2C_STARTGenerate( I2C0, ENABLE );
    while( I2C_FlagStatus( I2C0, I2C_SB) == RESET );
    I2C_AddressSend( I2C0, M24C08_Block3ADDRESS, I2C_Mode7,
I2C_TX );
    while(!((I2CStatus = I2C_GetStatus( I2C0 )) & I2C_EVF))
    while( I2C_FlagStatus( I2C0, I2C_ENDAD ) ==RESET );
    I2C_FlagClear( I2C0, I2C_ENDAD );

  }while( I2C_FlagStatus( I2C0, I2C_AF, I2CStatus ) == SET );
}

```



## I2C\_FlagClear

| Function Name          | I2C_FlagClear  |
|------------------------|--|
| Function Prototype     | <code>void I2C_FlagClear (I2C_TypeDef *I2Cx, u32 Flag, ...);</code>  |
| Behavior Description   | Clears the I2C Flag passed as a parameter.   |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to clear its flags where x can be 0 or 1.  |
| Input Parameter 2      | <i>Flag</i> : the flag to be read. Refer to <a href="#">I2C flags on page 167</a> for more details on allowed values of <i>Flag</i> parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | I2C_OnOffConfig  |

### Example:

This example illustrates how to clear the *I2C\_STOPF* flag:

```
{
  ...
  I2C_FlagClear (I2C0, I2C_STOPF)
  ...
}
```

## I2C\_SpeedConfig

| Function Name          | I2C_SpeedConfig   |
|------------------------|---|
| Function Prototype     | <code>void I2C_SpeedConfig (I2C_TypeDef *I2Cx, u32 Clock);</code>       |
| Behavior Description   | Configures the <i>I<sup>2</sup>C</i> clock speed.                       |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to be configured where x can be 0 or 1. |
| Input Parameter 2      | <i>Clock</i> : the <i>I<sup>2</sup>C</i> expected clock in Hertz.       |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | <a href="#">RCCU_FrequencyValue on page 43</a>                          |

### Example:

This example illustrates how to configure the clock speed:

```
{
  ...
  I2C_SpeedConfig (I2C0, 150000);
  ...
}
```

**I2C\_AddressConfig**

| Function Name          | I2C_AddressConfig   |
|------------------------|---|
| Function Prototype     | <code>void I2C_AddressConfig (I2C_TypeDef *I2Cx, u16 Address, I2C_Addressing Mode);</code>  |
| Behavior Description   | Configures the $I^2C$ bus address of the interface.   |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C to be configured where x can be 0 or 1.   |
| Input Parameter 2      | <i>Address</i> : the $I^2C$ bus address of the interface.   |
| Input Parameter 3      | <i>Mode</i> : specifies the addressing mode.<br>Refer to <a href="#">I2C addressing modes on page 167</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the address of the interface in the case of 7-bit addressing mode:

```

{
    ...
    I2C_AddressConfig (I2C0, 0xA0, I2C_Mode7);
    ...
}

```

**I2C\_FCLKConfig**

| Function Name          | I2C_FCLKConfig   |
|------------------------|--|
| Function Prototype     | <code>void I2C_FCLKConfig (I2C_TypeDef *I2Cx);</code>  |
| Behavior Description   | Configures $I^2C$ frequency bits according to <i>PCLK1</i> frequency. The selected I2C must be disabled. |
| Input Parameter        | <i>I2Cx</i> : specifies the I2C to be configured where x can be 0 or 1.                                  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | The $I^2C$ module is disabled ( <i>PE=0</i> )  |
| Called Functions       | <a href="#">RCCU_FrequencyValue</a>  |

**Example:**

This example illustrates how to configure the *I2C* frequency according to *PCLK1* frequency:

```

{
    ...
    /* the I2C module must be disabled (PE=0) */
    I2C_FCLKConfig (I2C0);
    ...
}

```

**I2C\_AddressSend**

| Function Name          | I2C_AddressSend  |
|------------------------|--|
| Function Prototype     | void I2C_AddressSend (I2C_TypeDef *I2Cx,<br>u16 Address, I2C_Addressing Mode,<br>I2C_Direction Direction);   |
| Behavior Description   | Sends the slave address with which the next communication will be performed.   |
| Input Parameter 1      | <i>I2Cx</i> : where <i>x</i> can be 0 or 1 to select the <i>I<sup>2</sup>C</i> peripheral which will send the slave address.   |
| Input Parameter 2      | <i>Address</i> : Indicates the slave address which will be transmitted.  |
| Input Parameter 3      | <i>Mode</i> : Refer to <a href="#">I2C addressing modes on page 167</a> for more details on the allowed values of this parameter   |
| Input Parameter 4      | <i>Direction</i> : specifies the communication direction.<br>Refer to <a href="#">I2C transfer direction on page 167</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | <a href="#">I2C_STARTGenerate</a>  |

**Example:**

This example illustrates how to send the address to slave in case of transmission direction and 7-bit addressing mode:

```

{
    ...
    I2C_AddressSend (I2C0, 0x43, I2C_Mode7, I2C_TX);
    ...
}

```

## I2C\_ByteSend

| Function Name          | I2C_ByteSend  |
|------------------------|---|
| Function Prototype     | <code>void I2C_ByteSend (I2C_TypeDef *I2Cx, u8 Data);</code>                                  |
| Behavior Description   | Transmits a single byte.  |
| Input Parameter 1      | <i>I2Cx</i> : specifies the I2C which will send the single byte where <i>x</i> can be 0 or 1. |
| Input Parameter 2      | <i>Data</i> : the byte to be transmitted.   |
| Output Parameter       | None  |
| Return Value           | None  |
| Required Preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to use the *I2C\_ByteSend* function to send a byte:

```
{
  ...
  I2C_ByteSend (I2C0, 0x5D);
  ...
}
```

## I2C\_TransmissionStatus

| Function Name          | I2C_TransmissionStatus  |
|------------------------|---|
| Function Prototype     | <code>I2C_Tx_Status<br/>I2C_TransmissionStatus (I2C_TypeDef *I2Cx);</code>  |
| Behavior Description   | Reports the current transmission status.  |
| Input Parameter        | <i>I2Cx</i> : specifies the I2C whose transmission status will be tested where <i>x</i> can be 0 or 1.  |
| Output Parameter       | None  |
| Return Value           | The transmission status. Refer to <a href="#">I2C transmission status on page 169</a> for more details on the allowed values of this parameter. |
| Required Preconditions | START generation.<br>Send slave address.  |
| Called Functions       | None  |

### Example:

This example illustrates how to report the status of transmission progress of *I2C0*:

```
{
  I2C_Tx_Status TransmissionStatus;
  ...
  TransmissionStatus = I2C_TransmissionStatus (I2C0);
  ...
}
```

## I2C\_ByteReceive

| Function Name          | I2C_ByteReceive   |
|------------------------|---|
| Function Prototype     | <code>u8 I2C_ByteReceive (I2C_TypeDef *I2Cx);</code>  |
| Behavior Description   | Returns the most recent received byte.  |
| Input Parameter        | <i>I2Cx</i> : specifies the I2C peripheral to get a received data where <i>x</i> can be 0 or 1. |
| Output Parameter       | None  |
| Return Value           | The value of the received byte.   |
| Required Preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to get the byte received:

```

{
    u8 received_byte;
    ...
    received_byte = I2C_ByteReceive (I2C0);
    ...
}

```

## I2C\_ReceptionStatus

| Function Name          | I2C_ReceptionStatus  |
|------------------------|--|
| Function Prototype     | <code>I2C_Rx_Status I2C_ReceptionStatus (I2C_TypeDef *I2Cx)</code>   |
| Behavior Description   | Reports the reception status.  |
| Input Parameter        | <i>I2Cx</i> : where <i>x</i> can be 0 or 1 to select the I <sup>2</sup> C peripheral which reception status will be tested.                  |
| Output Parameter       | None   |
| Return Value           | The status of reception. Refer to <a href="#">I2C reception status on page 170</a> for more details on the allowed values of this parameter. |
| Required Preconditions | START generation.<br>Send slave address.   |
| Called Functions       | None   |

**Example:** This example illustrates how to return the status of reception progress:

```

{
    I2C_Rx_Status ReceptionStatus;
    ...
    ReceptionStatus = I2C_ReceptionStatus(I2C0);
    ...
}

```

**I2C\_ErrorClear**

| Function Name          | I2C_ErrorClear   |
|------------------------|--|
| Function Prototype     | <code>void I2C_ErrorClear (I2C_TypeDef *I2Cx);</code>                            |
| Behavior Description   | Clears any error flag.   |
| Input Parameter        | I2Cx: specifies the I2C which error flags will be cleared where x can be 0 or 1. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to clear all error flags:

```

{
    ...
    I2C_ErrorClear (I2C0);
    ...
}

```

**I2C\_GetStatus**

| Function Name          | I2C_GetStatus  |
|------------------------|--|
| Function Prototype     | <code>u32 I2C_GetStatus (I2C_TypeDef *I2Cx);</code>                        |
| Behavior Description   | Reads the I2C status registers.  |
| Input Parameter        | I2Cx: specifies the I2C to get its status registers where x can be 0 or 1. |
| Output Parameter       | None   |
| Return Value           | The I2C status registers value.  |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to read all status registers:

```

{
    u32 I2C_Status_registers;
    I2C_Status_registers = I2C_GetStatus (I2C0);
    ...
}

```

**I2C\_GetLastEvent**

| Function Name          | I2C_GetLastEvent   |
|------------------------|--|
| Function Prototype     | u16 I2C_GetLastEvent (I2C_TypeDef *I2Cx);                                  |
| Behavior Description   | Gets the last I2Cx event that has occurred.                                |
| Input Parameter        | I2Cx: specifies the I2C to get its status registers where x can be 0 or 1. |
| Output Parameter       | None   |
| Return Value           | The last happened Event.   |
| Required Preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to get the last I2C0 event that has occurred:

```

{
    u16 I2C_Event;
    I2C_Event = I2C_GetLastEvent (I2C0);
    ...
}

```

**I2C\_CheckEvent**

| Function Name          | I2C_CheckEvent  |
|------------------------|---|
| Function Prototype     | FlagStatus I2C_CheckEvent (I2C_TypeDef *I2Cx, u16 I2C_EVENT);   |
| Behavior Description   | Checks whether the Last I2C Event is equal to the one passed as parameter.  |
| Input Parameter 1      | I2Cx: specifies the I2C to get its status registers where x can be 0 or 1.  |
| Input Parameter 2      | I2C_EVENT: specifies the event to be checked. Refer to <a href="#">I2C events on page 169</a> for more details on this parameter. |
| Output Parameter       | None  |
| Return Value           | The I2C status registers value.   |
| Required Preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to wait for EV8 event using I2C0:

```

{
    ...
    while (!I2C_CheckEvent (I2C0,
        I2C_EVENT_MASTER_BYTE_TRANSMITTED));
    ...
}

```

## 3.13 Controller area network (CAN)

The CAN driver may be used to exchange data between interconnected devices sharing a CAN bus compliant with the CAN 2.0 B standard. The application can use either polling mode or interrupt mode to handle the transmission and reception of frames.

The first section describes the data structures used in the CAN firmware library. The second section presents the CAN firmware library functions.

In this chapter, the terms “message” and “frame” will be alternatively used; they both designate a standard data packet of variable length.

### 3.13.1 Data structures

#### CAN registers' structure

The CAN registers' structure *CAN\_TypeDef* is defined in the *71x\_map.h* file as follows:

```
typedef volatile struct
{
    vu16 CR;
    vu16 EMPTY1;
    vu16 SR;
    vu16 EMPTY2;
    vu16 ERR;
    vu16 EMPTY3;
    vu16 BTR;
    vu16 EMPTY4;
    vu16 IDR;
    vu16 EMPTY5;
    vu16 TESTR;
    vu16 EMPTY6;
    vu16 BRPR;
    vu16 EMPTY7 [3];
    CAN_MsgObj_TypeDef sMsgObj [2];
    vu16 EMPTY8 [16];
    vu16 TR1R;
    vu16 EMPTY9;
    vu16 TR2R;
    vu16 EMPTY10 [13];
    vu16 ND1R;
    vu16 EMPTY11;
    vu16 ND2R;
    vu16 EMPTY12 [13];
    vu16 IP1R;
    vu16 EMPTY13;
    vu16 IP2R;
    vu16 EMPTY14 [13];
    vu16 MV1R;
    vu16 EMPTY15;
    vu16 MV2R;
    vu16 EMPTY16;
} CAN_TypeDef;
```



This structure embeds another structure named *CAN\_MsgObj\_TypeDef* to implement the message interface registers, defined as follows:

```
typedef volatile struct
{
    vu16 CRR;
    vu16 EMPTY1;
    vu16 CMR;
    vu16 EMPTY2;
    vu16 M1R;
    vu16 EMPTY3;
    vu16 M2R;
    vu16 EMPTY4;
    vu16 A1R;
    vu16 EMPTY5;
    vu16 A2R;
    vu16 EMPTY6;
    vu16 MCR;
    vu16 EMPTY7;
    vu16 DA1R;
    vu16 EMPTY8;
    vu16 DA2R;
    vu16 EMPTY9;
    vu16 DB1R;
    vu16 EMPTY10;
    vu16 DB2R;
    vu16 EMPTY11 [27];
} CAN_MsgObj_TypeDef;
```

[Table 78](#) describes the CAN structure fields.

**Table 78. CAN structure fields**

| Register | Description                   |
|----------|-------------------------------|
| CR       | Control Register              |
| SR       | Status Register               |
| ERR      | Error Counter                 |
| BTR      | Bit Timing Register           |
| IDR      | Interrupt Identifier Register |
| TESTR    | Test Register                 |
| BRPR     | BRP Extension Register        |
| TR1R     | Transmission Request 1        |
| TR2R     | Transmission Request 2        |
| ND1R     | New Data 1                    |
| ND2R     | New Data 2                    |
| IP1R     | Interrupt Pending 1 Register  |
| IP2R     | Interrupt Pending 2 Register  |

**Table 78. CAN structure fields (continued)**

| Register | Description              |
|----------|--------------------------|
| MV1R     | Message Valid 1 Register |
| MV2R     | Message Valid 2 Register |

[Table 79](#) describes the CAN message interface structure fields.

**Table 79. CAN message interface structure fields**

| Register | Description                    |
|----------|--------------------------------|
| CRR      | Command Request Register       |
| CMR      | Command Mask Register          |
| M1R      | Mask 1 Register                |
| M2R      | Mask 2 Register                |
| A1R      | Message Arbitration 1 Register |
| A2R      | Message Arbitration 2 Register |
| MCR      | Message Control Register       |
| DA1R     | Data A 1 Register              |
| DA2R     | Data A 2 Register              |
| DB1R     | Data B 1 Register              |
| DB2R     | Data B 2 Register              |

The CAN peripheral is also declared in the *71x\_map.h* file:

```

...
#define APB1_BASE 0xC0000000
...
#define CAN_BASE (APB1_BASE + 0x9000)
...
#ifndef DEBUG
...
#ifdef _CAN
#define CAN ((CAN_TypeDef *)CAN_BASE)
#endif
...
#else
...
#ifdef _CAN
EXT CAN_TypeDef *CAN;
#endif
...
#endif

```

When debug mode is used, CAN pointer is initialized in the file *71x\_lib.c*:

```

#ifdef _CAN
CAN = (CAN_TypeDef *)CAN_BASE;
#endif /* _CAN */

```

In order to access the peripheral registers, `_CAN` must be defined in the file `71x_conf.h` as follows:

```
#define _CAN
```

Some `XTI` and `PCU` functions are called, `_XTI` and `_PCU` must be defined, in `71x_conf.h` file, to make the `XTI` and `PCU` functions accessible:

```
#define _PCU
#define _XTI
```

### CAN standard bitrates

The following enumeration defines the different standard bitrates available. It is defined in the file `can.h`:

```
enum
{
    CAN_BITRATE_100K,
    CAN_BITRATE_125K,
    CAN_BITRATE_250K,
    CAN_BITRATE_500K,
    CAN_BITRATE_1M
};
```

[Table 80](#) describes the standard bitrates of the CAN with their timing parameters.

**Table 80. Standard CAN bitrates**

| Bitrate name     | Bitrate    | NTQ | TSEG1 | TSEG2 | SJW | BRP |
|------------------|------------|-----|-------|-------|-----|-----|
| CAN_BITRATE_100K | 100 kbit/s | 16  | 11    | 4     | 4   | 5   |
| CAN_BITRATE_125K | 125 kbit/s | 16  | 11    | 4     | 4   | 4   |
| CAN_BITRATE_250K | 250 kbit/s | 8   | 4     | 3     | 3   | 4   |
| CAN_BITRATE_500K | 500 kbit/s | 16  | 16    | 2     | 1   | 1   |
| CAN_BITRATE_1M   | 1 Mbit/s   | 8   | 4     | 3     | 1   | 1   |

Where;

NTQ: Number of Time Quantum

TSEG1: Time Segment before the sampling point

TSEG2: Time Segment after the sampling point

SJW: Synchronization Jump Width

BRP: Baud Rate Prescaler

### CAN control register bits

The following defines list the bit fields accessible in the CAN control register. They are declared in the file `can.h`:

```
#define CAN_CR_TEST    0x0080
#define CAN_CR_CCE    0x0040
#define CAN_CR_DAR    0x0020
#define CAN_CR_EIE    0x0008
#define CAN_CR_SIE    0x0004
#define CAN_CR_IE     0x0002
#define CAN_CR_INIT   0x0001
```

[Table 81](#) describes the CAN control register bits.

**Table 81. CAN control register bits**

| Bit mnemonic | Description                      |
|--------------|----------------------------------|
| CAN_CR_TEST  | Test mode                        |
| CAN_CR_CCE   | Configuration Change Enable      |
| CAN_CR_DAR   | Disable Automatic Retransmission |
| CAN_CR_EIE   | Error Interrupt Enable           |
| CAN_CR_SIE   | Status Interrupt Enable          |
| CAN_CR_IE    | Interrupt Enable                 |
| CAN_CR_INIT  | Initialization                   |

### CAN status register bits

The following defines list the bit fields accessible in the CAN status register. They are declared in the file *can.h*:

```
#define CAN_SR_LEC      0x0007
#define CAN_SR_TXOK    0x0008
#define CAN_SR_RXOK    0x0010
#define CAN_SR_EPASS   0x0020
#define CAN_SR_EWARN   0x0040
#define CAN_SR_BOFF    0x0080
```

[Table 82](#) describes the CAN status register bits.

**Table 82. CAN status register bits**

| Bit mnemonic | Description             |
|--------------|-------------------------|
| CAN_SR_LEC   | Last Error Code mask    |
| CAN_SR_TXOK  | Transmission successful |
| CAN_SR_RXOK  | Reception successful    |
| CAN_SR_EPASS | Error Passive State     |
| CAN_SR_EWARN | Warning State           |
| CAN_SR_BOFF  | Busoff State            |

### CAN test register bits

The following defines list the bit fields accessible in the CAN test register. They are declared in the file *can.h*:

```
#define CAN_TESTR_RX      0x0080
#define CAN_TESTR_TX1    0x0040
#define CAN_TESTR_TX0    0x0020
#define CAN_TESTR_LBACK  0x0010
#define CAN_TESTR_SILENT 0x0008
#define CAN_TESTR_BASIC  0x0004
```

Table 83 describes the CAN test register bits.

**Table 83. CAN test register bits**

| Bit mnemonic     | Description  |
|------------------|--|
| CAN_TESTR_RX     | CAN RX pin monitoring  |
| CAN_TESTR_TX1    | CAN TX pin monitoring  |
| CAN_TESTR_TX0    |  |
| CAN_TESTR_LBACK  | Loopback mode<br>In this mode, the CAN core has an internal loopback of TX to RX, and the CAN RX pin is logically disconnected from the bus. However, the CAN TX pin is still connected to the bus.  |
| CAN_TESTR_SILENT | Silent mode<br>In this mode, the CAN core has an internal loopback of TX to RX, and the CAN TX pin is logically disconnected from the bus and stuck to a recessive state. However, the CAN RX pin is still connected to the bus.                   |
| CAN_TESTR_BASIC  | Basic mode<br>This mode is used to minimize the resources needed to exchange data frames for a given node. The message RAM is not used anymore and there are only two dedicated message objects: the first one to transmit, the second to receive. |

### CAN message interface registers

The following defines list the bit fields accessible in the CAN Message Interface registers. They are declared in the file *can.h*:

```

/*IFx Command Request register */
#define CAN_CRR_BUSY      0x8000

/*IFn Command Mask register */
#define CAN_CMR_WRRD      0x0080
#define CAN_CMR_MASK      0x0040
#define CAN_CMR_ARB       0x0020
#define CAN_CMR_CONTROL   0x0010
#define CAN_CMR_CLRINTPND 0x0008
#define CAN_CMR_TXRQST    0x0004
#define CAN_CMR_DATAA     0x0002
#define CAN_CMR_DATAB     0x0001

/* IFn Mask 2 register */
#define CAN_M2R_MXTD      0x8000
#define CAN_M2R_MDIR      0x4000

/*IFn Arbitration 2 register */
#define CAN_A2R_MSGVAL    0x8000
#define CAN_A2R_XTD       0x4000
#define CAN_A2R_DIR       0x2000

/* IFn Message Control register */
#define CAN_MCR_NEWDAT    0x8000

```

```

#define CAN_MCR_MSGLST      0x4000
#define CAN_MCR_INTPND     0x2000
#define CAN_MCR_UMASK      0x1000
#define CAN_MCR_TXIE       0x0800
#define CAN_MCR_RXIE       0x0400
#define CAN_MCR_RMTEN      0x0200
#define CAN_MCR_TXRQST     0x0100
#define CAN_MCR_EOB        0x0080

```

Table 84 describes the CAN message interface register bits.

**Table 84. CAN message interface register bits**

| Bit mnemonic       | Description  |
|--------------------|--|
| CAN_CRR_BUSY       | Command Request register / Busy flag                                 |
| CAN_CMRR_WRRD      | Command Mask register / Write access                                 |
| CAN_CMRR_MASK      | Command Mask register / Mask access                                  |
| CAN_CMRR_ARB       | Command Mask register / Arbitration access                           |
| CAN_CMRR_CONTROL   | Command Mask register / Control access                               |
| CAN_CMRR_CLRINTPND | Command Mask register / Clear interrupt pending                      |
| CAN_CMRR_TXRQST    | Command Mask register / Transmit request release and New data access |
| CAN_CMRR_DATAA     | Command Mask register / Data bytes 0-3 access                        |
| CAN_CMRR_DATAB     | Command Mask register / Data bytes 4-7 access                        |
| CAN_M2R_MXTD       | Mask register / Mask extended identifier                             |
| CAN_M2R_MDIR       | Mask register / Mask message direction                               |
| CAN_A2R_MSGVAL     | Arbitration register / Message valid flag                            |
| CAN_A2R_XTD        | Arbitration register / Extended identifier                           |
| CAN_A2R_DIR        | Arbitration register / Message direction                             |
| CAN_MCR_NEWDAT     | Message Control register / New Data                                  |
| CAN_MCR_MSGLST     | Message Control register / Message Lost                              |
| CAN_MCR_INTPND     | Message Control register / Interrupt Pending                         |
| CAN_MCR_UMASK      | Message Control register / Use Acceptance Mask                       |
| CAN_MCR_TXIE       | Message Control register / Transmit Interrupt Enable                 |
| CAN_MCR_RXIE       | Message Control register / Receive Interrupt Enable                  |
| CAN_MCR_RMTEN      | Message Control register / Remote frame Enable                       |
| CAN_MCR_TXRQST     | Message Control register / Transmit Request                          |
| CAN_MCR_EOB        | Message Control register / End of Buffer                             |

## Wake-up modes

The following defines list wake-up modes available on the CAN cell. They are declared in the file *can.h*:

```
enum
{
    CAN_WAKEUP_ON_EXT,
    CAN_WAKEUP_ON_CAN
};
```

[Table 85](#) describes the CAN wake-up modes.

**Table 85. CAN wake-up modes**

| Wake-up mode      | Description  |
|-------------------|--|
| CAN_WAKEUP_ON_EXT | Wake-up on external event<br>The CAN cell is woken up by a falling edge on the I/O port 2.8.   |
| CAN_WAKEUP_ON_CAN | Wake-up on CAN bus activity<br>The CAN cell is woken up by a dominant state of the CAN bus (i.e. falling edge on the CAN RX pin = port 1.11) |

## CAN message structure

The structure that contains message data is defined as follows, and is declared in the file *can.h*:

```
typedef struct
{
    int IdType;
    u32 Id;
    u8 Dlc;
    u8 Data[8];
} canmsg;
```

[Table 86](#) describes the CAN message structure parameters.

**Table 86. CAN message structure parameters**

| CAN message parameter | Description                  |
|-----------------------|------------------------------|
| IdType                | Identifier type              |
| Id                    | Identifier value             |
| Dlc                   | Data Length Code             |
| Data[]                | Array containing data values |

### CAN message identifier types

The different CAN message identifier types are listed below, and are declared in the file *can.h*:

```
enum
{
    CAN_STD_ID,
    CAN_EXT_ID
};
```

[Table 87](#) describes the CAN message identifier types.

**Table 87. CAN message identifier types**

| ID type    | Description                  |
|------------|------------------------------|
| CAN_STD_ID | Standard identifier (11-bit) |
| CAN_EXT_ID | Extended identifier (29-bit) |

### CAN message identifier limits

The identifier limit values for each type are listed below, and are declared in the file *can.h*:

```
#define CAN_LAST_STD_ID    ((1<<11) - 1)
#define CAN_LAST_EXT_ID   ((1L<<29) - 1)
```

[Table 88](#) describes the CAN identifier limit values.

**Table 88. CAN identifier limit values**

| ID value        | Description                              |
|-----------------|--|
| CAN_LAST_STD_ID | Maximum standard identifier = 0x7FF      |
| CAN_LAST_EXT_ID | Maximum extended identifier = 0x1FFFFFFF |

## 3.13.2 Firmware library functions

[Table 89](#) enumerates the different functions of the CAN library.

**Table 89. CAN library functions**

| Function Name                       | Description   |
|-------------------------------------|---|
| <a href="#">CAN_EnterInitMode</a>   | Switches the CAN into initialization mode.                |
| <a href="#">CAN_LeaveInitMode</a>   | Leaves the initialization mode (switch into normal mode). |
| <a href="#">CAN_EnterTestMode</a>   | Switches the CAN into test mode.                          |
| <a href="#">CAN_LeaveTestMode</a>   | Leaves the current test mode (switch into normal mode).   |
| <a href="#">CAN_SetBitrate</a>      | Sets up a standard CAN bitrate.                           |
| <a href="#">CAN_SetTiming</a>       | Sets up the CAN timing with specific parameters.          |
| <a href="#">CAN_SleepRequest</a>    | Requests the CAN cell to enter sleep mode.                |
| <a href="#">CAN_SetUnusedMsgObj</a> | Configures the message object as unused.                  |
| <a href="#">CAN_SetTxMsgObj</a>     | Configures the message object as TX.                      |



**Table 89. CAN library functions (continued)**

| Function Name                          | Description  |
|--|--|
| <i>CAN_SetRxMsgObj</i>                 | Configures the message object as RX.   |
| <i>CAN_SetUnusedAllMsgObj</i>          | Configures all the message objects as unused.                                  |
| <i>CAN_Init</i>                        | Initializes the CAN cell and set the bitrate.                                  |
| <i>CAN_ReleaseMessage</i>              | Releases the message object.   |
| <i>CAN_ReleaseTxMessage</i>            | Releases the transmit message object.  |
| <i>CAN_ReleaseRxMessage</i>            | Releases the receive message object  |
| <i>CAN_UpdateMsgObj</i>                | Updates the message object.  |
| <i>CAN_TransmitRequest</i>             | Requests the transmission of a message object. A data or remote frame is sent. |
| <i>CAN_SendMessage</i>                 | Starts transmission of a message.  |
| <i>CAN_ReceiveMessage</i>              | Gets the message, if received.   |
| <i>CAN_WaitEndOfTx</i>                 | Waits until current transmission is finished.                                  |
| <i>CAN_BasicSendMessage</i>            | Starts transmission of a message in BASIC mode.                                |
| <i>CAN_BasicReceiveMessage</i>         | Gets the message in BASIC mode, if received.                                   |
| <i>CAN_GetMsgReceiveStatus</i>         | Tests the waiting status of a received message.                                |
| <i>CAN_GetMsgTransmitRequestStatus</i> | Tests the request status of a transmitted message.                             |
| <i>CAN_GetMsgInterruptStatus</i>       | Tests the interrupt status of a message object.                                |
| <i>CAN_GetMsgValidStatus</i>           | Tests the validity of a message object (ready to use).                         |
| <i>CAN_GetFlagStatus</i>               | Returns the state of the CAN flags: TxOK, RxOK, EPASS, EWARN and BOFF.         |
| <i>CAN_GetTransmitErrorCounter</i>     | Gets the CAN transmit Error counter.   |
| <i>CAN_GetReceiveErrorCounter</i>      | Gets the CAN receive Error counter.  |

**CAN\_EnterInitMode**

| Function Name          | CAN_EnterInitMode   |
|------------------------|---|
| Function Prototype     | <code>void CAN_EnterInitMode(u8 mask);</code>   |
| Behavior Description   | Switches the <i>CAN</i> into initialization mode.<br>This function must be used in conjunction with <a href="#">CAN_LeaveInitMode</a> . |
| Input Parameter        | <i>mask</i> : any binary value formed using the values defined in <a href="#">CAN control register bits on page 187</a> .               |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">CAN_Init on page 202</a> ,<br><a href="#">CAN_LeaveInitMode on page 194</a>   |

**Example:**

This example illustrates one method to initialize the *CAN*:

```

{
    ...
    /* Initialize the CAN and enable interrupts */
    CAN_EnterInitMode(CAN_CR_IE);
    ...
}

```

**CAN\_LeaveInitMode**

| Function Name          | CAN_LeaveInitMode   |
|------------------------|---|
| Function Prototype     | <code>void CAN_LeaveInitMode(void);</code>  |
| Behavior Description   | Leaves the initialization mode (switch into normal mode).<br>This function must be used in conjunction with <a href="#">CAN_EnterInitMode</a> . |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |
| See also               | <a href="#">CAN_EnterInitMode on page 194</a>   |

**Example:**

This example illustrates how to leave the initialization mode.

```

{
    ...
    CAN_LeaveInitMode();
    ...
}

```

## CAN\_EnterTestMode

| Function Name          | CAN_EnterTestMode  |
|------------------------|--|
| Function Prototype     | <code>void CAN_EnterTestMode(u8 mask);</code>  |
| Behavior Description   | Switches the CAN into test mode.   |
| Input Parameter        | <i>mask</i> : any binary value formed using the values defined in <a href="#">CAN test register bits on page 188</a> . |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

### Example:

This example illustrates how to enter any test mode.

```

{
    ...
    /* Switch the CAN into Loopback mode, i.e. RX is disconnected
       from the bus, and TX is internally linked to RX */
    CAN_EnterTestMode(CAN_TESTR_LBACK);
    ...
}

```

## CAN\_LeaveTestMode

| Function Name          | CAN_LeaveTestMode                                       |
|------------------------|---|
| Function Prototype     | <code>void CAN_LeaveTestMode(void);</code>              |
| Behavior Description   | Leaves the current test mode (switch into normal mode). |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to leave the current test mode.

```

{
    ...
    CAN_LeaveTestMode();
    ...
}

```

**CAN\_SetBitrate**

| Function Name          | CAN_SetBitrate   |
|------------------------|--|
| Function Prototype     | <code>void CAN_SetBitrate(int bitrate);</code>   |
| Behavior Description   | Sets up a standard CAN bitrate.  |
| Input Parameter        | <i>bitrate</i> : one of the <i>CAN_BITRATE_xxx</i> defines. Refer to <a href="#">CAN standard bitrates on page 187</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The CAN must be in initialization mode. PCLK1 must be at 8 MHz, otherwise, refer to <a href="#">CAN_SetTiming on page 196</a> .  |
| Called Functions       | None   |

**Example:**

This example illustrates how to set a standard bitrate for the CAN.

```

{
    ...
    // Enable the configuration change bit, to set the bitrate
    CAN_EnterInitMode(CAN_CR_CCE);
    CAN_SetBitrate(CAN_BITRATE_100K);
    CAN_LeaveInitMode();
    ...
}

```

**CAN\_SetTiming**

| Function Name          | CAN_SetTiming  |
|------------------------|--|
| Function Prototype     | <code>void CAN_SetTiming(u32 tseg1, u32 tseg2, u32 sjw, int brp);</code>               |
| Behavior Description   | Sets up the CAN timing with specific parameters.                                       |
| Input Parameter 1      | tseg1: Time Segment before the sample point position. It can take values from 1 to 16. |
| Input Parameter 2      | tseg2: Time Segment after the sample point position. It can take values from 1 to 8.   |
| Input Parameter 3      | sjw: Synchronisation Jump Width. It can take values from 1 to 4.                       |
| Input Parameter 4      | brp: Baud Rate Prescaler. It can take values from 1 to 1024.                           |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The CAN must be in initialization mode.  |
| Called Functions       | None   |

**Example:**

This example illustrates how to set specific timing parameters for the CAN.

```

{
  ...
  /* Enable the configuration change bit, to set the specific
     timing parameters: TSEG1=11, TSEG2=4, SJW=4, BRP=5 */
  CAN_EnterInitMode(CAN_CR_CCE);
  CAN_SetTiming(11, 4, 4, 5);
  CAN_LeaveInitMode();
  ...
}

```

**CAN\_SleepRequest**

| Function Name          | CAN_SleepRequest   |
|------------------------|--|
| Function Prototype     | void CAN_SleepRequest(u32 WakeupMode);   |
| Behavior Description   | Requests the CAN cell to enter sleep mode.   |
| Input Parameter        | WakeupMode: using one of the values defined in <a href="#">Wake-up modes on page 191</a> .   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | <a href="#">XTI_Init on page 79</a> ,<br><a href="#">XTI_LineModeConfig on page 81</a> ,<br><a href="#">XTI_LineConfig on page 82</a> ,<br><a href="#">XTI_ModeConfig on page 80</a> ,<br><a href="#">XTI_InterruptLineValue on page 82</a> ,<br><a href="#">XTI_PendingBitClear on page 83</a> ,<br><a href="#">PCU_STOP on page 27</a> |

**Example:**

This example illustrates how to activate the CAN cell sleep mode.

```

{
  ...
  CAN_SleepRequest(CAN_WAKEUP_ON_CAN);
  /* Note that the XTI interrupt handler is called as soon as the
     wake-up condition is reached */
  ...
}

```

**CAN\_SetUnusedMsgObj**

| Function Name          | CAN_SetUnusedMsgObj   |
|------------------------|---|
| Function Prototype     | u32 CAN_SetUnusedMsgObj (u32 msgobj) ;  |
| Behavior Description   | Configures the message object as unused.<br>The hardware will not process it until it is setup as RX or TX. |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.  |
| Output Parameter       | None  |
| Return Value           | 1: if interface is found<br>0: if no interface is available   |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure some message objects so that they will not be used.

```

{
    ...
    for (i=16; i<=31; i++)
        CAN_SetUnusedMsgObj (i) ;
    ...
}

```

**CAN\_SetTxMsgObj**

| Function Name          | CAN_SetTxMsgObj  |
|------------------------|--|
| Function Prototype     | u32 CAN_SetTxMsgObj (u32 msgobj, u32 idType, FunctionalState RemoteEN) ;   |
| Behavior Description   | Configures the message object as TX.   |
| Input Parameter 1      | <i>msgobj</i> : message object number, from 0 to 31.   |
| Input Parameter 2      | <i>idType</i> : message identifier type defined using the values described in <a href="#">CAN message identifier types on page 192</a> . |
| Input Parameter 3      | RemoteEN: If set to '1', the remote function is enabled. Otherwise, the remote function is disabled.                                     |
| Output Parameter       | None   |
| Return Value           | 1: if interface is found<br>0: if no interface is available  |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example describes how to configure a transmit message object.

```
{  
  ...  
  /* Defines the transmit message object 0 with standard  
    identifiers and disable the remote function */  
  CAN_SetTxMsgObj(0, CAN_STD_ID, DISABLE);  
  ...  
}
```

**Note:** *When defining which message object number to use for TX or RX, you must take into account the priority levels in the objects processing. The lower number (0) has the highest priority and the higher number (31) has the lowest priority, whatever their type.*

*It is also not recommended to have “holes” in the objects list for optimal performance. A typical usage of the message objects is, from low to high numbers:*

- transmit objects
- receive objects with filtering (fixed ID)
- receive objects with filtering (ID range)
- receive objects without filtering,
- unused objects

*independently of the FIFO depth.*

**CAN\_SetRxMsgObj**

| Function Name          | CAN_SetRxMsgObj  |
|------------------------|--|
| Function Prototype     | u32 CAN_SetRxMsgObj(u32 msgobj, u32 idType, u32 idLow, u32 idHigh, bool singleOrFifoLast);   |
| Behavior Description   | Configures the message object as RX.   |
| Input Parameter 1      | <i>msgobj</i> : message object number, from 0 to 31.   |
| Input Parameter 2      | <i>idType</i> : message identifier type, defined using the values described in <a href="#">CAN message identifier types on page 192</a> .  |
| Input Parameter 3      | <i>idLow</i> : low part of the identifier range used for acceptance filtering. It can take values from 0 to 0x7FF for standard ID, and values from 0 to 0x1FFFFFFF for extended ID.  |
| Input Parameter 4      | <i>idHigh</i> : high part of the identifier range used for acceptance filtering. It can take values from 0 to 0x7FF for standard ID, and values from 0 to 0x1FFFFFFF for extended ID.<br><i>idHigh</i> must be above <i>idLow</i> .<br>For more convenience, use one of the following values to set the maximum ID: <i>CAN_LAST_STD_ID</i> or <i>CAN_LAST_EXT_ID</i><br>Refer to <a href="#">CAN message identifier limits on page 192</a> for more details on the allowed values of this parameter. |
| Input Parameter 5      | <i>singleOrFifoLast</i> : End-of-buffer indicator, it can take the following values:<br>– <i>TRUE</i> for a single receive object or a FIFO receive object that is the last one of the FIFO<br>– <i>FALSE</i> for a FIFO receive object that is not the last one   |
| Output Parameter       | None   |
| Return Value           | 1: if interface is found<br>0: if no interface is available  |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example describes how to configure a receive message object.

```

{
    ...
    /* Define a receive FIFO of depth 2 (objects 0 and 1) for
       standard identifiers, in which IDs are filtered in the
       range 0x400-0x5FF */
    CAN_SetRxMsgObj(0, CAN_STD_ID, 0x400, 0x5FF, FALSE);
    CAN_SetRxMsgObj(1, CAN_STD_ID, 0x400, 0x5FF, TRUE);

    /* Define a single receive object for extended identifiers, in
       which all IDs are filtered in */
    CAN_SetRxMsgObj(2, CAN_EXT_ID, 0, CAN_LAST_EXT_ID, TRUE);
    ...
}

```



**Note:** Care must be taken when defining an ID range: all combinations of `idLow` and `idHigh` will not always produce the expected result, because of the way identifiers are filtered by the hardware.

The criteria applied to keep a received frame is as follows:

$(\text{received ID}) \text{ AND } (\text{ID mask}) = (\text{ID arbitration})$ , where AND is a bitwise operator.

Consequently, it is generally better to choose for `idLow` a value with some LSBs cleared, and for `idHigh` a value that “logically contains” `idLow` and with the same LSBs set.

**Example:** the range 0x100-0x3FF will work, but the range 0x100-0x2FF will not because 0x100 is not logically contained in 0x2FF (that is,  $0x100 \text{ AND } 0x2FF = 0$ ).

### CAN\_SetUnusedAllMsgObj

| Function Name          | CAN_SetUnusedAllMsgObj                                     |
|------------------------|--|
| Function Prototype     | <code>u32 CAN_InvalidateAllMsgObj(void);</code>            |
| Behavior Description   | Configures all the message objects as unused.              |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | 1, if all message objects are set to unused. Otherwise, 0. |
| Required preconditions | None   |
| Called Functions       | <a href="#">CAN_SetUnusedMsgObj on page 198</a>            |

#### Example:

This example describes how to invalidate all the message objects.

```
{
    ...
    CAN_SetUnusedAllMsgObj();
    /* now, some objects can be set up */
    ...
}
```

## CAN\_Init

| Function Name          | CAN_Init  |
|------------------------|---|
| Function Prototype     | <code>void CAN_Init(u8 mask, u32 bitrate);</code>   |
| Behavior Description   | Initializes the CAN cell and set the bitrate.   |
| Input Parameter 1      | <i>mask</i> : any binary value defined using the values described in <a href="#">CAN control register bits on page 187</a> .  |
| Input Parameter 2      | <i>bitrate</i> : one of the <code>CAN_BITRATE_xxx</code> defines. Refer to <a href="#">CAN standard bitrates on page 187</a> for more details on the allowed values of this parameter.              |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | PCLK1 must be 8 MHz. Otherwise refer to <a href="#">CAN_SetTiming on page 196</a> .   |
| Called Functions       | <a href="#">CAN_EnterInitMode on page 194</a> ,<br><a href="#">CAN_SetBitrate on page 196</a> ,<br><a href="#">CAN_LeaveInitMode on page 194</a> ,<br><a href="#">CAN_LeaveTestMode on page 195</a> |

### Example:

This example illustrates a typical CAN initialization.

```
{
  ...
  /* Initialize the CAN at 100 kbit/s and enable interrupts */
  CAN_Init(CAN_CR_IE, CAN_BITRATE_100K);
  ...
}
```

## CAN\_ReleaseMessage

| Function Name          | CAN_ReleaseMessage  |
|------------------------|---|
| Function Prototype     | <code>u32 CAN_ReleaseMessage(u32 msgobj);</code>            |
| Behavior Description   | Releases the message object.                                |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.        |
| Output Parameter       | None  |
| Return Value           | 1: if interface is found<br>0: if no interface is available |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to release a message object.

```
{
  ...
```

```

    /* Release the message object 0 */
    CAN_ReleaseMessage(0);
    ...
}

```

### CAN\_ReleaseTxMessage

| Function Name          | CAN_ReleaseTxMessage  |
|------------------------|---|
| Function Prototype     | void CAN_ReleaseTxMessage(u32 msgobj);                                  |
| Behavior Description   | Releases the transmit message object.                                   |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.                    |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | It is assumed that message interface 0 is always used for transmission. |
| Called Functions       | None  |

#### Example:

This example illustrates how to release a transmit message object.

```

{
    ...
    /* Release the transmit message object 0 */
    CAN_ReleaseTxMessage(0);
    ...
}

```

### CAN\_ReleaseRxMessage

| Function Name          | CAN_ReleaseRxMessage   |
|------------------------|--|
| Function Prototype     | void CAN_ReleaseRxMessage(u32 msgobj);                               |
| Behavior Description   | Releases the receive message object.                                 |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.                 |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | It is assumed that message interface 1 is always used for reception. |
| Called Functions       | None   |

#### Example:

This example illustrates how to release a receive message object.

```

{
    ...
    /* Release the receive message object 0 */
    CAN_ReleaseTxMessage(0);
    ...
}

```

## CAN\_UpdateMsgObj

| Function Name          | CAN_UpdateMsgObj  |
|------------------------|---|
| Function Prototype     | u32 CAN_UpdateMsgObj (u32 msgobj, canmsg pCanMsg);  |
| Behavior Description   | Updates the message object passed in parameter with the pCanMsg.  |
| Input Parameter 1      | <i>msgobj</i> : message object number, from 0 to 31.  |
| Input Parameter 2      | <i>pCanMsg</i> : pointer to the canmsg structure that contains the data to transmit: ID type, ID value, data length, data values.<br>Refer to <a href="#">CAN message structure on page 191</a> for more details. |
| Output Parameter       | None  |
| Return Value           | 1: if interface is found<br>0: if no interface is available   |
| Required preconditions | It is assumed that message interface 0 is used to update a can message object.  |
| Called Functions       | None  |

### Example:

This example illustrates how to update a message object already configured to transmit.

```

{
    ...
    canmsg CanMsg = { CAN_STD_ID, 0x111, 4, {0x10, 0x20, 0x40, 0x80}
};
    /* Update CAN Message 0 */
    CAN_UpdateMsgObj(0, &CanMsg);
    ...
}

```

## CAN\_TransmitRequest

| Function Name          | CAN_TransmitRequest   |
|------------------------|---|
| Prototype              | u32 CAN_TransmitRequest ( u32 msgobj);  |
| Behavior Description   | Starts the transmission of a message object. A data frame is transmitted if the message object is configured in transmission mode. A remote frame is transmitted if the message object is configured in reception mode. |
| Input Parameter        | <i>msgobj</i> : The message object number, from 0 to 31.  |
| Output Parameter       | None  |
| Return Value           | 1: if interface is found<br>0: if no interface is available   |
| Required preconditions | The message object must have been set up properly.  |

**Example:**

This example illustrates how to request the transmission of a data frame.

```

{
    ...
    /* Message object 0 is already configured for transmission*/
    /* Request the transmission of a data frame */
    CAN_TransmitRequest(0);
    ...
}

```

**CAN\_SendMessage**

| Function Name          | CAN_SendMessage   |
|------------------------|---|
| Function Prototype     | u32 CAN_SendMessage(u32 msgobj, canmsg* pCanMsg);   |
| Behavior Description   | Starts transmission of a message.   |
| Input Parameter 1      | <i>msgobj</i> : message object number, from 0 to 31.  |
| Input Parameter 2      | <i>pCanMsg</i> : pointer to the canmsg structure that contains the data to transmit: ID type, ID value, data length, data values.<br>Refer to <a href="#">CAN message structure on page 191</a> for more details. |
| Output Parameter       | None  |
| Return Value           | 1 if transmission was OK, else 0  |
| Required preconditions | The message object must have been set up properly.  |
| Called Functions       | CAN_UpdateMsgObj()<br>CAN_TransmitRequest()   |

**Example:**

This example illustrates how to send a single message.

```

{
    ...
    canmsg CanMsg = { CAN_STD_ID, 0x111, 4, {0x10, 0x20, 0x40,
0x80} };
    /* Send a standard ID data frame containing 4 data values */
    CAN_SendMessage(0, &CanMsg);
    ...
}

```

**CAN\_ReceiveMessage**

| Function Name          | CAN_ReceiveMessage   |
|------------------------|--|
| Function Prototype     | u32 CAN_ReceiveMessage(u32 msgobj, bool release, canmsg* pCanMsg);   |
| Behavior Description   | Gets the message, if received.   |
| Input Parameter 1      | <i>msgobj</i> : message object number, from 0 to 31.   |
| Input Parameter 2      | <i>release</i> : message release indicator, it can take the following values:<br>– <i>TRUE</i> : the message object is released at the same time as it is copied from message RAM, then it is free for next reception<br>– <i>FALSE</i> : the message object is not released, it is to the caller to do it |
| Input Parameter 3      | <i>pCanMsg</i> : pointer to the canmsg structure where the received message is copied. Refer to <a href="#">CAN message structure on page 191</a> for more details.  |
| Output Parameter       | None   |
| Return Value           | 1 if reception was OK, else 0 (no message pending)   |
| Required preconditions | The message object must have been set up properly.   |
| Called Functions       | <a href="#">CAN_GetMsgReceiveStatus on page 209</a>  |

**Example:**

This example illustrates how to receive a single message.

```

{
    ...
    canmsg CanMsg;
    /* Receive a message in the object 0 and ask for release */
    if (CAN_ReceiveMessage(0, TRUE, &CanMsg))
    {
        /* Check or copy the message contents */
    }
    else
    {
        /* Error handling */
    }
    ...
}

```

## CAN\_WaitEndOfTx

| Function Name          | CAN_WaitEndOfTx   |
|------------------------|---|
| Function Prototype     | <code>void CAN_WaitEndOfTx(void);</code>  |
| Behavior Description   | Waits until current transmission is finished.<br>This function should be called between two consecutive transmissions to ensure the latest frame has been completely emitted on the bus, and therefore cannot be aborted anymore. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | A message must have been sent before.   |
| Called Functions       | None  |

### Example:

This example illustrates how to wait for the end of transmission of the latest frame sent.

```

{
    ...
    /* Send consecutive data frames using message object 0 */
    for (i = 0; i < 10; i++)
    {
        CAN_SendMessage(0, CanMsgTable[i]);
        CAN_WaitEndOfTx();
    }
    ...
}

```

## CAN\_BasicSendMessage

| Function Name          | CAN_BasicSendMessage  |
|------------------------|---|
| Function Prototype     | <code>u32 CAN_BasicSendMessage(canmsg* pCanMsg);</code>   |
| Behavior Description   | Starts transmission of a message in BASIC mode. Refer to <a href="#">CAN test register bits on page 188</a> for more details on the Basic mode.   |
| Input Parameter        | <i>pCanMsg</i> : pointer to the canmsg structure that contains the data to transmit: ID type, ID value, data length, data values.<br>Refer to <a href="#">CAN message structure on page 191</a> for more details. |
| Output Parameter       | None  |
| Return Value           | 1 if transmission was OK, else 0  |
| Required preconditions | The CAN must have been switched into BASIC mode.  |
| Called Functions       | None  |

**Example:**

This example illustrates how to send a single message.

```

{
    ...
    canmsg CanMsg = { CAN_STD_ID, 0x111, 4, {0x10, 0x20, 0x40,
0x80} };
    CAN_BasicSendMessage(0, &CanMsg);
    ...
}

```

**CAN\_BasicReceiveMessage**

| Function Name          | CAN_BasicReceiveMessage   |
|------------------------|---|
| Function Prototype     | u32 CAN_BasicReceiveMessage(canmsg* pCanMsg);   |
| Behavior Description   | Gets the message in BASIC mode, if received. Refer to <a href="#">CAN test register bits on page 188</a> for more details on the Basic mode.                        |
| Input Parameter        | <i>pCanMsg</i> : pointer to the canmsg structure where the received message is copied. Refer to <a href="#">CAN message structure on page 191</a> for more details. |
| Output Parameter       | None  |
| Return Value           | 1 if reception was OK, else 0 (no message pending)  |
| Required preconditions | The CAN must have been switched into BASIC mode.  |
| Called Functions       | None  |

**Example:**

This example illustrates how to receive a single message in Basic mode.

```

{
    ...
    canmsg CanMsg;
    /* Receive a message */
    if (CAN_BasicReceiveMessage(&CanMsg))
    {
        /* Check or copy the message contents */
    }
    else
    {
        /* Error handling */
    }
    ...
}

```



## CAN\_GetMsgReceiveStatus

| Function Name          | CAN_GetMsgReceiveStatus  |
|------------------------|--|
| Function Prototype     | FlasgStatus CAN_GetMsgReceiveStatus(u32 msgobj);   |
| Behavior Description   | Tests the waiting status of a received message.  |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.                                       |
| Output Parameter       | None   |
| Return Value           | 1 if the corresponding message object has received a message waiting to be copied, else 0. |
| Required preconditions | The corresponding message object must have been set as RX.                                 |
| Called Functions       | None   |

### Example:

This example illustrates how to test the waiting status of a message.

```

{
    ...
    /* Test if a message is pending in the receive message
       object 0 */
    if (CAN_GetMsgReceiveStatus(0))
    {
        /* Receive the message from this message object (i.e. get its
           data from message RAM) */
    }
    ...
}

```

## CAN\_GetMsgTransmitRequestStatus

| Function Name          | CAN_GetMsgTransmitRequestStatus                                  |
|------------------------|--|
| Function Prototype     | FlasgStatus CAN_GetMsgTransmitRequestStatus(u32 msgobj);         |
| Behavior Description   | Tests the request status of a transmitted message.               |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.             |
| Output Parameter       | None   |
| Return Value           | 1 if the corresponding message is requested to transmit, else 0. |
| Required preconditions | A message must have been sent before.                            |
| Called Functions       | None   |

**Example:**

This example illustrates how to test transmit request status of a message.

```

{
  ...
  /* Send a message using object 0 */
  CAN_SendMessage(0, &CanMsg);
  /* Wait for the end of transmit request */
  while (CAN_GetMsgTransmitRequestStatus(0));
  /* Now, the message is being processed by the priority handler
    of the CAN cell, and ready to be emitted on the bus */
  ...
}

```

**CAN\_GetMsgInterruptStatus**

| Function Name          | CAN_GetMsgInterruptStatus  |
|------------------------|--|
| Function Prototype     | FlagStatus CAN_GetMsgInterruptStatus(u32 msgobj);                |
| Behavior Description   | Tests the interrupt status of a message object.                  |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.             |
| Output Parameter       | None   |
| Return Value           | 1 if the corresponding message has an interrupt pending, else 0. |
| Required preconditions | The interrupts must have been enabled.                           |
| Called Functions       | None   |

**Example:**

This example illustrates how to test the waiting status of a message.

```

{
  ...
  /* Send a message using object 0 */
  CAN_SendMessage(0, &CanMsg);
  /* Wait for the TX interrupt */
  while (!CAN_GetMsgInterruptStatus(0));
  ...
}

```

## CAN\_GetMsgValidStatus

| Function Name          | CAN_GetMsgValidStatus   |
|------------------------|---|
| Function Prototype     | FlasgStatus CAN_GetMsgValidStatus(u32 msgobj);  |
| Behavior Description   | Tests the validity of a message object (ready to use).<br>A valid object means that it has been set up either as TX or as RX, and so is used by the hardware. |
| Input Parameter        | <i>msgobj</i> : message object number, from 0 to 31.  |
| Output Parameter       | None  |
| Return Value           | 1 if the corresponding message object is valid, else 0.   |
| Required preconditions | None  |
| Called Functions       | None  |

### Example:

This example illustrates how to test the validity of a message object.

```

{
    ...
    if (CAN_GetMsgValidStatus(10))
    {
        /* Do something with message object 10 */
    }
}

```

## CAN\_GetFlagStatus

| Function Name          | CAN_GetFlagStatus  |
|------------------------|--|
| Prototype              | FlasgStatus CAN_GetFlagStatus( u32 CAN_Flag);  |
| Behavior Description   | Returns the state of the CAN flags: TxOK, RxOK, EPASS, EWARN and BOFF  |
| Input Parameter        | One of the following:<br>- CAN_SR_TXOK<br>- CAN_SR_RXOK<br>- CAN_SR_EPASS<br>- CAN_SR_EWARN<br>- CAN_SR_BOFF |
| Output Parameter       | None   |
| Return Value           | 1 if the flag passed in parameter is set, else 0.  |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example illustrates how to read the Ewarning status of the CAN cell.

```

{
  ...
  /* if CAN cell has reached the Ewarning status */
  if (CAN_GetFlagStatus(CAN_SR_EWARN))
  {
    /* Error handling*/
  }
  else
  {
    /*continue */
  }
  ...
}

```

**CAN\_GetTransmitErrorCounter**

| Function Name          | CAN_GetTransmitErrorCounter                |
|------------------------|--|
| Prototype              | u32 CAN_GetTransmitErrorCounter( void );   |
| Behavior Description   | Returns the transmit error counter content |
| Input Parameter        | None                                       |
| Output Parameter       | None                                       |
| Return Value           | Transmit Error Counter value               |
| Required preconditions | None                                       |
| Called Functions       | None                                       |

**Example:**

This example illustrates how to read the CAN transmit error counter.

```

{
  ...
  /* read the transmit error counter */
  if (CAN_GetTransmitErrorCounter() > 10)
  {
    /* Error handling */
  }
  ...
}

```

## CAN\_GetReceiveErrorCounter

| Function Name          | CAN_GetReceiveErrorCounter                |
|------------------------|---|
| Prototype              | u32 CAN_GetReceiveErrorCounter( void );   |
| Behavior Description   | Returns the receive error counter content |
| Input Parameter        | None                                      |
| Output Parameter       | None                                      |
| Return Value           | Receive Error Counter value               |
| Required preconditions | None                                      |
| Called Functions       | None                                      |

### Example:

This example illustrates how to read the CAN receive error counter.

```

{
    ...
    /* read the receive error counter */
    if (CAN_GetReceiveErrorCounter() > 20)
    {
        /* Error handling*/
    }
    ...
}

```

## 3.14 12-bit analog-to-digital converter (ADC12)

The *ADC12* driver may be used to manage the *ADC12* in its two modes of conversion: single or round-robin and to generate a maskable interrupt when a sample is ready.

The first section describes the data structures used in the *ADC12* firmware library. The second one presents the firmware library functions.

### 3.14.1 Data structures

#### ADC12 register structure

the *ADC12* peripheral registers' structure *ADC12\_TypeDef* is defined in the *71x\_map.h* file as follows:

```

typedef struct
{
    vu16 DATA0;
    u16  EMPTY1[3];
    vu16 DATA1;
    u16  EMPTY2[3];
    vu16 DATA2;
    u16  EMPTY3[3];
    vu16 DATA3;
    u16  EMPTY4[3];
    vu16 CSR;
}

```

```

u16 EMPTY5[7];
vu16 CPR;
u16 EMPTY6;
} ADC12_TypeDef;

```

[Table 90](#) presents the *ADC12* registers.

**Table 90. ADC12 registers**

| Register | Description                       |
|----------|-----------------------------------|
| DATA0    | ADC12 data register for Channel 0 |
| DATA1    | ADC12 data register for Channel 1 |
| DATA2    | ADC12 data register for Channel 2 |
| DATA3    | ADC12 data register for Channel 3 |
| CSR      | ADC12 control status register     |
| CPR      | ADC12 prescaler register          |

The *ADC12* peripheral is declared in the same file:

```

...
#define APB2_BASE 0xE0000000
...
#define ADC12_BASE      (APB2_BASE + 0x7000)
...
#ifndef DEBUG
...
#ifdef _ADC12
#define ADC12          ((ADC12_TypeDef *)ADC12_BASE)
#endif /*ADC12*/
...
#else /* DEBUG */
...
#ifdef _ADC12
EXT ADC12_TypeDef *ADC12;
#endif
...
#endif

```

When debug mode is used, *ADC12* pointer is initialized in *71x\_lib.c* file:

```

void debug(void)
{
...
#ifdef _ADC12
ADC12 = (ADC12_TypeDef *)ADC12_BASE;
#endif /* _ADC12 */
...
}

```

In debug mode, *\_ADC12* must be defined, in *71x\_conf.h* file, to access the peripheral registers as follows:

```

#define _ADC12

```

When *RCCU* functions are called, `_RCCU` must be defined, in *71x\_conf.h* file, to make the *RCCU* functions accessible:

```
#define _RCCU
```

The *ADC12* function accede to the *Bootconf* Register defined in the *PCU* structure, `_PCU` must be defined, in *71x\_conf.h* file, to make the *PCU* register accessible:

```
#define _PCU
```

### Modes of conversion

The following enumeration defines the modes of conversion. *ADC12\_Modes* enumeration is defined in the file *71x\_adc12.h*:

```
typedef enum
{
    ADC12_SINGLE,
    ADC12_ROUND
} ADC12_Modes;
```

[Table 91](#) presents the *ADC12* conversion modes.

**Table 91. ADC12 conversion modes**

| Type         | Description  |
|--------------|--|
| ADC12_SINGLE | Enables the single channel mode of conversion              |
| ADC12_ROUND  | Enables the round robin (multi-channel) mode of conversion |

### ADC12 channels

The following enumeration defines the *ADC12* channels. *ADC12\_Channels* enumeration is defined in the file *71x\_adc12.h* as follows:

```
typedef enum
{
    ADC12_CHANNEL0 = 0x00,
    ADC12_CHANNEL1 = 0x10,
    ADC12_CHANNEL2 = 0x20,
    ADC12_CHANNEL3 = 0x30
} ADC12_Channels;
```

[Table 92](#) defines the *ADC12* channels.

**Table 92. ADC12 channels**

| ADC12 Channels | Description |
|----------------|-------------|
| ADC12_CHANNEL0 | Channel 0.  |
| ADC12_CHANNEL1 | Channel 1.  |
| ADC12_CHANNEL2 | Channel 2.  |
| ADC12_CHANNEL3 | Channel 3.  |

### ADC12 flags

The following enumeration defines the *ADC12* flags. *ADC12\_Flags* enumeration is defined in the file *71x\_adc12.h*:

```
typedef enum
{
    ADC12_DA0 = 0x01,
    ADC12_DA1 = 0x02,
    ADC12_DA2 = 0x04,
    ADC12_DA3 = 0x08,
    ADC12_OR  = 0x2000
} ADC12_Flags;
```

[Table 93](#) presents the *ADC12* flags.

**Table 93. ADC12 flags**

| ADC12 Flags | Description  |
|-------------|--|
| ADC12_DA0   | Data available in Channel 0.   |
| ADC12_DA1   | Data available in Channel 1.   |
| ADC12_DA2   | Data available in Channel 2.   |
| ADC12_DA3   | Data available in Channel 3.   |
| ADC12_OR    | Overrun: specifies whether data on one of the channels has been overwritten before being read. |

### 3.14.2 Firmware library functions

[Table 94](#) enumerates the different functions of the *ADC12* library.

**Table 94. ADC12 library functions**

| Function Name                         | Description   |
|---------------------------------------|---|
| <a href="#">ADC12_Init</a>            | Initializes the <i>ADC12</i> .  |
| <a href="#">ADC12_ModeConfig</a>      | Configures the conversion mode (single channel or round robin operation). |
| <a href="#">ADC12_PrescalerConfig</a> | Configures the prescaler to define an oversampling frequency.             |
| <a href="#">ADC12_ChannelSelect</a>   | Selects the channel to be converted in single channel operation.          |
| <a href="#">ADC12_ConversionStart</a> | Starts the <i>ADC12</i> conversion in the selected mode.                  |
| <a href="#">ADC12_ConversionStop</a>  | Stops the <i>ADC12</i> conversion.  |
| <a href="#">ADC12_ConversionValue</a> | Gets the result of conversion from the selected data register.            |
| <a href="#">ADC12_FlagStatus</a>      | Returns the status of the specified flag.                                 |
| <a href="#">ADC12_ITConfig</a>        | Enables /Disables the <i>ADC12</i> interrupt.                             |



**ADC12\_Init**

| Function Name          | ADC12_Init  |
|------------------------|---|
| Function Prototype     | void ADC12_Init(void)   |
| Behavior Description   | This routine is used to initialize the ADC12 registers to their reset values. |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example shows how to initialize the *ADC12*:

```

{
  /* Initialize the converter registers */
  ADC12_Init();
  ...
}

```

**ADC12\_ModeConfig**

| Function Name          | ADC12_ModeConfig   |
|------------------------|--|
| Function Prototype     | void ADC12_ModeConfig (ADC12_Modes Mode);  |
| Behavior Description   | This routine is used to select the mode of conversion.   |
| Input Parameter        | <i>Mode</i> : specifies the conversion mode.<br>Refer to <a href="#">Modes of conversion on page 215</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | None   |
| Called Functions       | None   |

**Example:**

This example shows how to configure the mode of conversion:

```

{
  ...
  /* Enable single channel mode of conversion */
  ADC12_ModeConfig(ADC12_SINGLE);
  ...
}

```

**ADC12\_PrescalerConfig**

| Function Name          | ADC12_PrescalerConfig  |
|------------------------|--|
| Function Prototype     | <code>void ADC12_PrescalerConfig(u32 Adc12_clk);</code>  |
| Behavior Description   | This function is used to configure the ADC sampling frequency by setting the prescaler register. |
| Input Parameter        | <i>Adc12_clk</i> : specifies the ADC12 sampling frequency (Hz).                                  |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | Call of the <i>RCCU</i> library to get the <i>APB2</i> frequency                                 |
| Called Functions       | <a href="#">RCCU_FrequencyValue on page 43</a>   |

**Example:**

This example shows how to configure the prescaler.

```

{
    ...
    /* Configure the ADC sampling frequency to 500Hz*/
    ADC12_PrescalerConfig(500);
    ...
}

```

**ADC12\_ChannelSelect**

| Function Name          | ADC12_ChannelSelect   |
|------------------------|---|
| Function Prototype     | <code>void ADC12_ChannelSelect(ADC_Channels ADC12_Channel);</code>  |
| Behavior Description   | This function selects the channel to be converted in single channel mode of conversion.   |
| Input Parameter        | <i>ADC12_Channel</i> : specifies the channel to be converted. Refer to <a href="#">ADC12 channels on page 215</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | Called only if the mode of conversion selected is the single Channel mode   |
| Called Functions       | None  |
| See also               | <a href="#">ADC12_ModeConfig on page 217</a>  |

**Example:**

This example shows how and when to use the *ADC12\_ChannelSelect* function:

```

/* configure the single channel mode */
ADC12_ModeConfig(ADC12_SINGLE);
/* select the channel 3 to be converted */
ADC12_ChannelSelect(ADC12_CHANNEL3);

```

**ADC12\_ConversionStart**

| Function Name          | ADC12_ConversionStart                            |
|------------------------|--|
| Function Prototype     | void ADC12_ConversionStart (void);               |
| Behavior Description   | This routine is used to launch the conversion.   |
| Input Parameter        | None   |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | Mode of conversion and prescaler configured      |
| Called Functions       | None   |
| See also               | <a href="#">ADC12_ConversionStop on page 219</a> |

**Example:**

This example shows how to start the *ADC12* conversion.

```

{
    ...
    /* configure the prescaler */
    ADC12_PrescalerConfig();
    /* configure the mode */
    ADC12_ModeConfig(ADC12_SINGLE);
    /* select the channel 3 to be converted */
    ADC12_ChannelSelect(ADC12_CHANNEL3);
    /* Enable the ADC12 and Start the conversion */
    ADC12_ConversionStart();
    ...
}

```

**ADC12\_ConversionStop**

| Function Name          | ADC12_ConversionStop                              |
|------------------------|---|
| Function Prototype     | void ADC12_ConversionStop (void);                 |
| Behavior Description   | This routine is used to disable the ADC12 cell.   |
| Input Parameter        | None  |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | Conversion started                                |
| Called Functions       | None  |
| See also               | <a href="#">ADC12_ConversionStart on page 219</a> |

**Example:**

This example shows how to end the *ADC12* conversion.

```

{
    ...
    /* start the conversion */

```

```

    ADC12_ConversionStart();
    ...
    /* Stop the conversion */
    ADC12_ConversionStop();
    ...
}

```

## ADC12\_ConversionValue

| Function Name          | ADC12_ConversionValue   |
|------------------------|---|
| Function Prototype     | u16 ADC12_ConversionValue( ADC12_Channels ADC12_Channel);   |
| Behavior Description   | This functions gets the conversion result from the data register of the specified channel. It returns the value of the corresponding data register.   |
| Input Parameter        | <i>ADC12_Channel</i> : specifies the channel to get its result of conversion. Refer to <a href="#">ADC12 channels on page 215</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The result of conversion of the specified channel.  |
| Required preconditions | The conversion of the selected channel was finished and no overrun detected.  |
| Called Functions       | None  |
| See also               | <a href="#">ADC12_FlagStatus on page 221</a>  |

### Example:

This example shows how to get the result of conversion. It tests whether the end of conversion has been reached or not.

```

{
    u16 data;
    /* Test if the conversion of the channel 3 has finished */
    while ((ADC12_FlagStatus (ADC12_DA3)) == RESET);
    /* Test the overrun bit */
    if (ADC12_FlagStatus(ADC12_OR) == RESET)
        /* Get the conversion result */
        data = ADC12_ConversionValue(ADC12_CHANNEL3);
    ADC12_ConversionStop();
}

```

**ADC12\_FlagStatus**

| Function Name          | ADC12_FlagStatus  |
|------------------------|---|
| Function Prototype     | FlagStatus ADC12_FlagStatus(ADC12_Flags flag);  |
| Behavior Description   | This function is used to test the status of the specified flag.   |
| Input Parameter        | <i>flag</i> : specifies the flag to get the status.<br>Refer to <a href="#">ADC12 flags on page 216</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | The status of the specified flag. Its value can be:<br>– <i>SET</i> : The corresponding Flag is Set<br>– <i>RESET</i> : The corresponding Flag is cleared         |
| Required preconditions | Conversion started.   |
| Called Functions       | None  |
| See also               | <a href="#">ADC12_ConversionValue on page 220</a>   |

**Example:**

This example illustrates an example of use of the *ADC12\_flagStatus* function:

```

{
    u16 test;
    ...
    /* Test if the conversion on the Channel 3 has finished or not.
    */
    while ((ADC12_FlagStatus(ADC12_DA3)) == RESET);
    /* Test the overrun bit */
    if (ADC12_FlagStatus(ADC12_OR) == RESET)
        /* Get the conversion result */
        test = ADC12_ConversionValue(3);
}

```

**ADC12\_ITConfig**

| Function Name          | ADC12_ITConfig   |
|------------------------|--|
| Function Prototype     | void ADC12_ITConfig (FunctionalState NewState);  |
| Behavior Description   | This routine is used to enable or disable the interrupt routine.   |
| Input Parameter        | <i>NewState</i> : specifies whether the interrupt will be enabled or disabled.<br>– <i>ENABLE</i> : enable the interrupt routine.<br>– <i>DISABLE</i> : disable the interrupt routine. |
| Output Parameter       | None   |
| Return Value           | None   |
| Required preconditions | The modes of conversion and the channel to be converted must be configured.  |
| Called Functions       | None   |

*Note:* *ADC12\_ITConfig* enables or disables either one channel interrupt or all the channels. If the mode of conversion is the single mode, only the interrupt of the channel whose conversion has been activated can be enabled or disabled. In the case of the round-robin mode of conversion, all the channels' interrupts can be either enabled or disabled together.

**Example:**

```

{
    ...
    /* Enable interrupt */
    ADC12_ITConfig(ENABLE);
    ...
    ADC12_ITConfig(DISABLE);
    ...
}

```

**3.15 External memory interface (EMI)**

The *EMI* driver may be used to access any memory bank external to the chip, via the *EMI* interface.

The first section describes the data structure used in the *EMI* firmware library. The second section presents the *EMI* firmware library functions.

**3.15.1 Data structures**

*EMI* peripheral is declared in the *71x\_map.h* file as follows:

```

#define EXTMEM_BASE      0x60000000
#define EMI_BASE        (EXTMEM_BASE + 0x0C000000)
#ifndef DEBUG
...
#ifdef _EMI
#define EMI              ((EMI_TypeDef *)EMI_BASE)
#endif /*EMI*/

```

```

...
#else /* DEBUG */
...
#ifdef _EMI
    EXT EMI_TypeDef          *EMI;
#endif /*EMI*/
...
#endif

```

### EMI Banks

The following definitions are used to specify which *EMI* bank is *to be used*:

```

#define EMI_BANK0 0x00
#define EMI_BANK1 0x01
#define EMI_BANK2 0x02
#define EMI_BANK3 0x03

```

### EMI bus size

The following definitions defines the *EMI* data bus size.

```

#define EMI_SIZE_8      0x0000
#define EMI_SIZE_16    0x0001

```

[Table 95](#) describes the *EMI* modes.

**Table 95. EMI modes**

| Pin mode    | Description  |
|-------------|--|
| EMI_SIZE_8  | Configures the <i>EMI</i> to work in 8-bit data bus  |
| EMI_SIZE_16 | Configures the <i>EMI</i> to work in 16-bit data bus |

### EMI enable

The following definition is used to enable the *EMI*:

```

#define EMI_ENABLE      0x8000

```

### EMI number of wait states

The following definitions are used to configure the number of the *EMI wait states*:

```

#define EMI_0_WaitState      0x00
#define EMI_1_WaitState      0x01
#define EMI_2_WaitStates     0x02
#define EMI_3_WaitStates     0x03
#define EMI_4_WaitStates     0x04
#define EMI_5_WaitStates     0x05
#define EMI_6_WaitStates     0x06
#define EMI_7_WaitStates     0x07
#define EMI_8_WaitStates     0x08
#define EMI_9_WaitStates     0x09
#define EMI_10_WaitStates    0x0A
#define EMI_11_WaitStates    0x0B
#define EMI_12_WaitStates    0x0C
#define EMI_13_WaitStates    0x0D
#define EMI_14_WaitStates    0x0E
#define EMI_15_WaitStates    0x0F

```

### 3.15.2 Firmware library functions

[Table 96](#) enumerates the different functions of the *EMI* library.

**Table 96. EMI library functions**

| Function Name              | Description                                   |
|----------------------------|---|
| <a href="#">EMI_Config</a> | Configures the <i>EMI</i> peripheral          |
| <a href="#">EMI_Enable</a> | Enables or disables the <i>EMI</i> peripheral |

#### EMI\_Config

| Function Name          | EMI_Config  |
|------------------------|---|
| Function Prototype     | <code>void EMI_Config (u8 Bank_n, u16 B_SIZE, u16 C_LENGTH);</code>   |
| Behavior Description   | This routine is used, for each bank, to configure cycle length and bus size.  |
| Input Parameter 1      | <i>Bank_n</i> : selects the bank to be configured<br>Refer to <a href="#">Data structures on page 222</a> for more details on the allowed values of this parameter.   |
| Input Parameter 2      | <i>B_SIZE</i> : specifies the effective external bus size for an access to Bank n.<br>Refer to <a href="#">Data structures on page 222</a> for more details on the allowed values of this parameter.                                |
| Input Parameter 3      | <i>C_LENGTH</i> : specifies the number of wait states to be inserted in any read/write cycle performed in Bank n.<br>Refer to <a href="#">Data structures on page 222</a> for more details on the allowed values of this parameter. |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |



**EMI\_Enable**

| Function Name          | EMI_Enable  |
|------------------------|---|
| Function Prototype     | void EMI_Enable (u8 Bank_n, FunctionalState NewState);  |
| Behavior Description   | This routine is used, for each bank, to configure cycle length and bus size.  |
| Input Parameter 1      | <i>Bank_n</i> : selects the bank to be enabled or disabled.<br>Refer to <a href="#">Data structures on page 222</a> for more details on the allowed values of this parameter. |
| Input Parameter 2      | <i>NewState</i> : specifies whether the EMI bank will be enabled or disabled.<br>– ENABLE: enable the specified EMI bank.<br>– DISABLE: disable the specified EMI bank.       |
| Output Parameter       | None  |
| Return Value           | None  |
| Required preconditions | None  |
| Called Functions       | None  |

**Example:**

This example illustrates how to configure the *EMI*.

```

{
    u8 data;
    ...
    /* Configure the EMI bank 1: 7 wait states, 16-bit wide
    external memory device */
    EMI_Config(EMI_BANK1, EMI_SIZE_16 ,EMI_7_WaitStates)
    /* Enable EMI bank 1 */
    EMI_Enable (EMI_BANK1, ENABLE);
    /* Write data to the external memory */
    *(u32*)0x62000000 = 0x12345678; /* write 12345678 in bank 1 */
    /* Read data from the external memory */
    data = *(u8*)0x62000008; /* read from bank 1 */
    ...
}

```

## 4 Revision history

**Table 97. Document revision history**

| Date        | Revision | Changes   |
|-------------|----------|---|
| 24-Jan-2006 | 1.0      | Initial release.  |
| 31-Jan-04   | 1.1      | Changed file description in <a href="#">Section 2.1 on page 11</a> , below figure 1.<br>Changed RCCU_RCLK_Clocks enum type in <a href="#">RCLK clock source on page 31</a> .<br>Added RCCU_RTC_CLOCK row in <a href="#">RCLK clock source on page 31</a> .  |
| 27-Mar-04   | 1.2      | Added Register name differences <a href="#">Section 3</a>   |
| 10-Jul-04   | 1.3      | New types are defined in <a href="#">Section 1.4 on page 7</a><br>How to use the STR71x Library in <a href="#">Section 2.3 on page 14</a><br>PCU_EnterWFI function renamed to PCU_WFIEnter in <a href="#">Section 5.1.2.7</a><br>PCU_EnterLPM function renamed to PCU_LPMEEnter in <a href="#">Section 5.1.2.8</a><br>FLASH_AccessPrConfig function deleted in <a href="#">Section 3.16 on page 241</a><br>FLASH_EraseSector function renamed to FLASH_SectorErase in <a href="#">Section on page 248</a><br>FLASH_EraseBank function renamed to FLASH_BankErase in <a href="#">FLASH_BankErase on page 249</a><br>FLASH_EraseModule renamed to FLASH_ModuleErase in <a href="#">Section 5.16.2.8</a><br>New Function RCCU_ResetSources added in <a href="#">RCCU_ResetSource on page 45</a>  |
| 31-Mar-05   | 2.0      | Package Description <a href="#">Section 2.1 on page 11</a><br>How to use the STR71x Library <a href="#">Section 2.3 on page 14</a><br>STR71xDL7 renamed to 71xLibraryD <a href="#">Section 3.3.1 on page 18</a><br>STR71xnDL renamed to 71xLibraryR.lib <a href="#">Section 3.3.1 on page 18</a><br>newDL.prj & newnDL.prj renamed to debug.prj & release.prj <a href="#">Section 3.3.1 on page 18</a><br>New Enumerations PLL1 Multiplication factors and PLL2 Multiplication factors <a href="#">Section 3.2 on page 29</a><br>RCCU_PLL1Config & RCCU_PLL2Config prototype modified <a href="#">Section 3.2 on page 29</a><br>RTC_SetTime & RTC_SetAlarmTime functions deleted (see AN 1780) <a href="#">Section 3.7 on page 84</a><br>FLASH_BlockWrite: deleted <a href="#">Section 3.16 on page 241</a><br>FLASH_BlockRead: deleted <a href="#">Section 3.16 on page 241</a><br>TIM_CounterConfig function behaviour description modified <a href="#">TIM_CounterConfig on page 118</a><br>TIM_CounterValue function added <a href="#">TIM_CounterValue on page 118</a><br>ADC12_PrescalerConfig equation corrected <a href="#">ADC12_PrescalerConfig on page 218</a> |

Table 97. Document revision history (continued)

| Date      | Revision | Changes  |
|-----------|----------|--|
| 17-Sep-07 | 3        | <p>Changed document title to STR71x firmware library.<br/> Added <a href="#">List of tables</a>.<br/> Removed Flash and USB sections.<br/> Removed <a href="#">Section 4.16.1: Write Operation States</a>.<br/> Section 2: "Environment overview" removed<br/> <a href="#">Section 2.1: Package description on page 11</a> updated<br/> System file list updated in <a href="#">Section 2.2 on page 12</a><br/> Modified <a href="#">Section 3.1: Power control unit (PCU)</a><br/> Example code updated for <a href="#">WFI clocks on page 19</a><br/> Description table updated for <a href="#">WFI clocks on page 19</a><br/> Low power modes section removed.<br/> Renamed the following file names: ppp.c by 71x_ppp.c, ppp.h by 71x_ppp.h, adc12.h by 71x_adc12.h, pcu.h by 71x_pcu.h, rccu.h by 71x_rccu.h, emi.h by 71x_emi.h, apb.h by 71x_apb.h, bspi.h by 71x_bspi.h, can.h by 71x_can.h, gpio.h by 71x_gpio.h, i2c.h by 71x_i2c.h, rtc.h by 71x_rtc.h, tim.h by 71x_tim.h, uart.h by 71x_uart.h, wdg.h by 71x_wdg.h and xti.h by 71x_xti.h<br/> Removed functions <i>PCU_WFIEnter</i> and <i>PCU_LPMEnter</i>.<br/> Added functions <i>PCU_LPModesConfig</i>, <i>PCU_WFI</i>, <i>PCU_STOP</i>, <i>PCU_STANDBY</i>, <i>PCU_FlashBurstCmd</i>.<br/> In <a href="#">Section 3.2: Reset and clock control unit (RCCU)</a><br/> Example code updated for <a href="#">RCLK clock source on page 31</a><br/> New section added <a href="#">PLL1 free running modes on page 35</a><br/> <a href="#">RCCU_PLL1FreeRunningModeConfig</a>, <a href="#">RCCU_PLL1Disable</a>, <a href="#">RCCU_PLL2Disable</a>, <a href="#">PCU_32OSCCmd</a> added<br/> Renamed function <i>RCCU_PCLK</i> to <i>RCCU_PCLK2</i><br/> Renamed function <i>RCCU_FCLK</i> to <i>RCCU_PCLK1</i><br/> Renamed function <i>RCCU_RTC_CLOCK</i> to <i>RCCU_CK_AF</i><br/> Changed function <i>RCCU_PCLKConfig</i> to <i>RCCU_PCLK2Config</i><br/> Renamed function <i>RCCU_FCLKConfig</i> to <i>RCCU_PCLK1Config</i><br/> Added new function <i>RCCU_PLL1FreeRunningModeConfig</i><br/> Added new function <i>RCCU_PLL1Disable</i><br/> Added new function <i>RCCU_PLL2Disable</i><br/> Added new function <i>RCCU_GenerateSWReset</i><br/> Renamed <i>WFI_EXTERNAL</i> to <i>WFI_Ck_AF</i><br/> Modified <a href="#">Section 3.9: Timer (TIM)</a><br/> Description updated for <a href="#">TIM_ITConfig on page 119</a><br/> Input parameters corrected for <a href="#">TIM_OPModeConfig on page 115</a><br/> Description updated for <a href="#">TIM_ITConfig on page 119</a><br/> Renamed function <i>TIM_TOE_Mask</i> to <i>TIM_TOIE_Mask</i><br/> Renamed function <i>TIM_OCAIE_mask</i> to <i>TIM_OCAIE_Mask</i><br/> Renamed function <i>TIM_OCBIE_mask</i> to <i>TIM_OCBIE_Mask</i></p> |

Table 97. Document revision history (continued)

| Date      | Revision   | Changes  |
|-----------|------------|--|
| 17-Sep-07 | 3 (cont'd) | <p>In <a href="#">Section 3.10: Buffered serial peripheral interface (BSPI)</a></p> <ul style="list-style-type: none"> <li>Renamed the typedef <i>BSPI_ITS</i> to <i>BSPI_IT_ERR</i></li> <li>Renamed the interrupt error sources: <i>BSPI_BERIT</i> by <i>BSPI_BEIE</i>, <i>BSPI_RCIT</i> by <i>BSPI_REIE</i> and <i>BSPI_ALL</i> by <i>BSPI_ALL_ERR</i></li> <li>Changed typedef <i>BSPI_TR_IT_SRCS</i> to <i>BSPI_IT_TR</i></li> <li>Changed typedef <i>BSPI_RC_IR_SRCS</i> to <i>BSPI_IT_RC</i></li> <li>Changed <i>BSPI_ItEnable</i> function name to <i>BSPI_ErrItSrc</i></li> <li>Added new function <i>BSPI_WordBufferSend</i></li> <li>Added new function <i>BSPI_WordBufferReceive</i>,</li> <li>Changed <i>BSPI_BufferSend</i> function to <i>BSPI_ByteBufferSend</i></li> <li>Changed <i>BSPI_BufferReceive</i> function to <i>BSPI_ByteBufferReceive</i></li> </ul> <p>In <a href="#">Section 3.11: Universal asynchronous receiver transmitter (UART)</a></p> <ul style="list-style-type: none"> <li>Changed <i>UART_RxBufFull</i> to <i>UART_RxBufNotEmpty</i></li> <li>Changed <i>sendchar</i> function name to <i>SendChar</i></li> </ul> <p>In <a href="#">Section 3.12: Inter-integrated circuit (I2C)</a></p> <ul style="list-style-type: none"> <li>I<sup>2</sup>C Flags updated, <a href="#">I2C flags on page 167</a></li> <li>I2C_FlagStatus example modified, <a href="#">I2C_FlagStatus on page 176</a></li> <li>Note 2 added for <i>GPIO_BitWrite</i> and <i>GPIO_ByteWrite</i> tables</li> <li>Repeated I2C registers table deleted: <a href="#">Section 3.12.1 on page 165</a></li> <li><i>I2C_StringSend</i>, <i>I2C_BufferSend</i>, <i>I2C_BufferReceive</i> removed</li> <li>Added new functions <i>I2C_GetLastEvent</i> and <i>I2C_CheckEvent</i>.</li> <li>Added <a href="#">Section : I2C events on page 169</a>.</li> </ul> <p>In <a href="#">Section 3.13: Controller area network (CAN)</a></p> <ul style="list-style-type: none"> <li>Preconditions updated for <a href="#">CAN_SetBitrate on page 196</a></li> <li>Preconditions updated for <a href="#">CAN_Init on page 202</a></li> <li>Changed <i>CAN_InvalidateAllMsgObj</i> function name to <i>CAN_SetUnusedAllMsgObj</i></li> <li>Changed <i>CAN_IsMessageWaiting</i> function name to <i>CAN_GetMsgReceiveStatus</i></li> <li>Changed <i>CAN_IsTransmitRequested</i> function name to <i>CAN_GetMsgTransmitRequestStatus</i></li> <li>Changed <i>CAN_IsObjectValid</i> function name to <i>CAN_GetMsgValidStatus</i></li> <li>Changed <i>CAN_IsInterruptPending</i> function name to <i>CAN_GetMsgInterruptStatus</i></li> <li>Added new function <i>CAN_UpdateMsgObj</i></li> <li>Added new function <i>CAN_TransmitRequest</i></li> <li>Added new function <i>CAN_GetFlagStatus</i></li> <li>Added new function <i>CAN_GetTransmitErrorCounter</i></li> <li>Added new function <i>CAN_GetReceiveErrorCounter</i></li> </ul> <p>In <a href="#">Section 3.15: External memory interface (EMI)</a></p> <ul style="list-style-type: none"> <li>Modified <a href="#">EMI_Config on page 224</a></li> <li>Modified the define values in <a href="#">EMI Banks on page 223</a></li> <li>Added a new function <a href="#">EMI_Enable on page 225</a></li> </ul> |

**Please Read Carefully:**

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

**UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.**

**UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.**

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2007 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

[www.st.com](http://www.st.com)

