MC1322x Software Driver

Reference Manual

Document Number: 22XDRVRRM
Rev. 1.5
05/2011

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Contents

About This Book

Chapter 1
MC1322x Software Driver Overview

Chapter 2
Clock and Reset Module (CRM) Driver

Chapter 3
GPIO Driver

Chapter 4
Interrupt Controller (ITC) Driver

Chapter 5
Non-Volatile Memory (NVM) Driver

Chapter 6
UART Driver

Chapter 7
Timer (TMR) Driver

Chapter 8
Inter-integrated Circuit (I2C) Driver

Chapter 9
Synchronous Serial Interface (SSI) Driver

Chapter 10
Analog to Digital Converter (ADC) Driver

Chapter 11
Serial Peripheral Interface (SPI) Driver

About This Book

This user's guide provides a detailed description of the MC1322x software drivers, their interfaces, usage and examples of how to perform key activities.

Audience

This guide is intended for application designers and users of the MC1322x device.

Organization

This document is organized into 11 chapters.

Chapter 1 MC1322x Software Driver Overview - Briefly describes the MC1322x and provides an overview of the MC1322x software drivers.

Chapter 2 Clock and Reset Module (CRM) Driver - Describes the CRM driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 3 GPIO Driver - Describes the GPIO driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 4 Interrupt Controller (ITC) Driver - Describes the ITC driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 5 Non-Volatile Memory (NVM) Driver - Describes the NVM driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 6 UART Driver - Describes the UART driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 7 Timer (TMR) Driver - Describes the TMR driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 8 Inter-integrated Circuit (I2C) Driver - Describes the I2C driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 9 Synchronous Serial Interface (SSI) Driver - Describes the SSI driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 10 Analog to Digital Converter (ADC) Driver - Describes the ADC driver, its functionality, hardware compatibility and all included files and functionality.

Chapter 11 Serial Peripheral Interface (SPI) Driver - Describes the SPI driver, its functionality, hardware compatibility and all included files and functionality.

Revision History

The following table summarizes revisions to this document since the previous release (Rev 1.4).

Revision History

Location	Revision
Entire document.	Updated every chapter.

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

API	Application Programming Interface
CE	Consumer Electronics
LQI	Link Quality Indicator
NW Layer	Network Layer
PAN	Personal Area Network
NV	Non volatile
NVM	Non volatile memory

References

The following sources were referenced to produce this book:

1. IEEE 802.15.4 Standard
2. Freescale MC1322x Data Sheet
3. Freescale MC1321x Data Sheet
4. Freescale MC1322x Reference Manual
5. Standard for Part 15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low rate wireless personal area networks (WPAN). IEEE Std 802.15.4-2006, IEEE, New York, NY, 2006.

Chapter 1
MC1322x Software Driver Overview

The MC1322x family is Freescale's third-generation ZigBee platform which incorporates a complete, low power, 2.4 GHz radio frequency transceiver, 32-bit ARM7TDMI-S core based MCU, hardware acceleration for both the IEEE 802.15.4 MAC and AES security, and a full set of MCU peripherals into a 99-pin LGA Platform-in-Package (PiP). A full 32-bit ARM7TDMI-S core operates up to 26 MHz. A 128 Kbyte FLASH memory is mirrored into a 96 Kbyte RAM for upper stack and applications software. In addition, an 80 Kbyte ROM is available for boot software, standardized IEEE 802.15.4 MAC and communications stack software. A full set of peripherals and Direct Memory Access (DMA) capability for transceiver packet data complement the processor core. A MC1322x device contains the following hardware modules:

- IEEE 802.15.4 MAC accelerator (MACA) (sequencer and DMA interface)
- Advanced encryption/decryption hardware engine (AES 128-bit)
- Dedicated NVM SPI interface for managing 128 Kbyte serial FLASH memory
- Clock and Reset Module (CRM) to control clock and power management resources including the KBI interface
- Interrupt Controller (ITC) Module
- Dedicated 802.15.4 modem/radio interface module (RIF)
- Two dedicated UART modules (UARTx) capable of 2Mbps with CTS/RTS support
- SPI port (SPI) with programmable master and slave operation
- Two 12-bit analog-to-digital converters (ADCs) share 8 input channels
- Timer Module (TMR) with Four independent 16-bit timers with PWM capability and cascade capability
- Inter-integrated circuit (I2C) interface
- Synchronous Serial Interface (SSI) with I2S and SPI capability and FIFO data buffering
- Up to 64 programmable I/O shared by peripherals and GPIO

To assist applications development on the MC1322x platform, Freescale provides a number of software drivers written in C code. These drivers are described in this document.

Chapter 2
Clock and Reset Module (CRM) Driver

2.1 Overview

The MC1322x primary mode control is through the Clock and Reset Module (CRM) which is accessed with the CRM driver.

2.1.1 CRM Hardware Module

The CRM is responsible for the management of the MC1322x including reset, power management, wake-up from low power, clock control, and KBI interface. The CRM is a dedicated module to handle top level all clock, reset and power functions and also controls the onboard voltage regulators for power management. The CRM also contains a small block called the Sleep Module that runs when the rest of the device is entirely powered down and it keeps time during sleep and wake modes. For detailed information on the CRM module, see the *MC1322x Reference Manual* (MC1322xRM).

The CRM module features include:

- Manages system reset and available software initiated system reset
- Controls clock gating for power savings
- Power management of internal regulated voltage sources including buck regulator
- Sleep mode management
  - 2 types of sleep: Hibernation or Doze
  - Controlled power down
  - Programmable degree of power-down
  - Critical programmed values are retained during sleep
- Wake-up mode management
  - Chip is gracefully powered up.
  - Clocks are automatically turned on.
  - Wake-up available by programmable timers.
  - Wake-up available by external interrupts from KBI pads.
- Watchdog (COP) surveillance timer
- Real Time Clock (RTC)
- Bus stealing mode to keep ARM quiet during idle periods.
- Management of ring oscillator and optional 32.768 KHz oscillators
- Management of reference oscillator
- Management of MCU core and peripheral clocks

2.1.2 Driver Functionality

The driver allows the developer to use the CRM functions easily. It allows the developer to:

- Place the IC in doze mode
- Place the IC in hibernate mode
- Control the wake up sources
- Enable/disable the wake up timer time-out interrupt
- Enable/disable the interrupts for the external wake up sources
- Manage wake ups
- Read the peripheral device status
- Initialize the COP counter
- Service the COP counter in order to prevent a COP counter time-out
- Force the COP counter time-out
- Set the time-out value for the RTC
- Enable/disable the RTC time-out interrupt
- Enable/disable bus cycle stealing from the MCU by the radio
- Halt the MCU until an interrupt occurs
- Control the reference crystal
- Protect the reference crystal configuration
- Enable the 32KHZ oscillator
- Disable the 2KHZ ring oscillator
- Enable the 2KHZ ring oscillator calibration
- Abort the 2KHZ ring oscillator calibration
- Control the voltage regulators
- Program the trim values for the various voltage regulators
- Set the output voltage for digital outputs to either 1.8V or 3.3V
- Set the output voltage for nonvolatile memory interface to either 1.8V or 3.3V
- Enable/disable the JTAG and Nexus module
- Initiate a software reset
- Register an interrupt software routine for any CRM interrupt source

2.2 Include Files

Table 2-1 shows the CRM driver files that must be included in the application C-files to have access to the driver function calls.

Table 2-1. CRM Driver Include Files

Include File Name	Description
CRM.h	Global header file that defines the public data types and specifies the public functions prototypes

2.3 CRM Driver API Functions

Table 2-2 shows the available API functions for the CRM Driver.

Table 2-2. CRM Driver API Function List

CRM Driver API Function Name	Description
CRM_GoToSleep ()	This function places the device in sleep mode (either Doze or Hibernate) and configures the sleep mode parameters.
CRM_WuCntl ()	The function is called to configure the Wake up sources.
CRM_ModuleEnStatus ()	The function simply reports which modules are enabled and which modules are disabled.
CRM_CopCntl ()	This function is called to configure the COP counter : enable/disable the COP counter, configure the COP to either generate an interrupt or reset the MCU in the event of a COP counter time-out , set the COP counter time-out period and write protect the content of the COP control register.
CRM_ForceCopTimeout ()	The function forces the COP counter time-out.
CRM_BusStealingCntl ()	This function configures the bus cycle stealing from the ARM by the Radio.
CRM_Wait4Irq ()	This function halts the CPU until an interrupt occurs.
CRM_RefXtalControl ()	The function controls the reference crystal oscillator.
CRM_2kToXtal32Switch()	The function starts the 32khz oscillator and after that stops the 2KHz ring oscillator.
CRM_RingOscCal ()	The function starts the calibration process for the 2KHz ring oscillator.
CRM_RingOscAbortCal ()	The function is called to abort the Ring oscillator calibration process.
CRM_SetPowerSource ()	The function sets the power configuration for the MCU.
CRM_VRegCntl ()	The function is called to configure one of the following three voltage regulators: Buck voltage regulator, 1.5V voltage regulator or 1.8V voltage regulator.
CRM_VRegTrimm ()	The function is called to adjust the behavior of the specified device.
CRM_SetDigOutDriveStrength()	The function is called to set the output voltage level for the digital pads.
crm_SetSPIFDriveStrength()	The function is called to set the voltage level of the SPI interface with the internal flash.
CRM_RegisterISR ()	The function is called to register the function that pfISR points to as the interrupt software routine for the interrupt source specified by crmIS parameter.
CRM_Isr()	The function will be set as interrupt routine for all the Crm interrupt sources.

2.4 Driver Exported Constants

2.4.1 pfCallback_t

Constant Structure

```
typedef void (*pfCallback_t)(void);
```

Description

Used as a function pointer type within the following functions: CRM_2kToXtal32Switch(), CRM_Xtal32EnCallback(), CRM_RingOscCal(), CRM_RegisterISR(), CRM_Isr().

2.4.2 crmErr_t

Constant Structure

```
typedef enum
{
    gCrmErrNoError_c,
    gCrmErrCopCntlWP_c,
    gCrmErrXtalCntlWP_c,
    gCrmErrInvalidParameters_c,
    gCrmErrCalInProgress_c,
    gCrmErrNoCalInProgress_c,
    gCrmErrRingOscOff_c,
    gCrmErrIgnoredInActualPowerMode_c,
    gCrmErrInvalidPowerSource_c,
    gCrmErrBuckNotEnabledNorBypassed_c
}crmErr_t;
```

Description

Specifies the possible return values for the CRM driver API functions.

Constant Values

The constant values are self explanatory.

2.4.3 crmPowerSource_t

Constant Structure

```
typedef enum
{
    gCrmPwS3V3Battery_c,
    gCrmPwSBuckRegulation_c,
    gCrmPwS1V8Battery_c,
    gCrmPwSNotValid_c
}crmPowerSource_t;
```

Description

Specifies the possible power sources for the MC1322x.

Constant Values

- gCrmPwS3V3Battery_c: a 3.3V battery powers the MC1322x.Buck regulator output is tied to 3.3V battery
- gCrmPwSBuckRegulation_c: a 3.3V battery powers the MC1322x.Buck regulator output is powering the 1.8V and 1.5V regulators
- gCrmPwS1V8Battery_c: a 1.8V battery powers the MC1322x.The buck regulator output is tied to the 1.8V battery

2.4.4 crmPadsDriveStrength_t

Constant Structure

```
typedef enum
{
    gPadStandardDrive_c,
    gPadHighDrive_c
} crmPadsDriveStrength_t;
```

Description

Specifies the possible drive strength for digital pads.

Constant Values

- gPadStandardDrive_c: standard drive strength for all digital pads
- gPadHighDrive_c: high drive strength for all digital pads

2.4.5 crmAnalogVRegEnable_t

Constant Structure

```
typedef enum
{
    gARegDisable_c,
    gRxTxRegEnable_c,
    gARegNotValid_c,
    gRxTxandPLLRegEnable_c = 3
}crmAnalogVRegEnable_t;
```

Description

Specifies the possible power settings for the analog section.

Constant Values

- gARegDisable_c: Analog voltage regulator (1.5V voltage regulator) is disabled
- gRxTxRegEnable_c: Rx/Tx voltage regulator is on
- gRxTxandPLLRegEnable_c: Rx/Tx plus PLL voltage regulators are on.

2.4.6 crmAnalogVRegCurrentSel_t

Constant Structure

```
typedef enum
{
     gARegCurent4mA_c,
     gARegCurent20mA_c,
     gARegCurentNotValid_c,
     gARegCurent40mA_c = 3
}crmAnalogVRegCurrentSel_t;
```

Description

Specifies the possible values for analog voltage regulator current selector.

Constant Values

- gARegCurent4mA_c: sourced current is 4mA (pre-transceiver state)
- gARegCurent20mA_c: sourced current is 20mA (normal transceiver state)
- gARegCurent40mA_c: sourced current is 40mA (6dB transceive state).

2.4.7 crmVReg_t

Constant Structure

```
typedef enum
{
     gBuckVReg_c,
     g1P5VReg_c,
     g1P8VReg_c,
     gMaxVReg_c
}crmVReg_t;
```

Description

Specifies the possible voltage regulators that this driver can handle.

Constant Values

The constant values are self explanatory.

2.4.8 crmInterruptSource_t

Constant Structure

```
typedef enum
{
     gCrmTimerWuEvent_c,
     gCrmRTCWuEvent_c,
     gCrmKB4WuEvent_c,
     gCrmKB5WuEvent_c,
     gCrmKB6WuEvent_c,
     gCrmKB7WuEvent_c,
     gCrmCalDoneEvent_c,
     gCrmXtal32ReadyEvent_c,
     gCrmCOPTimeoutEvent_c
}crmInterruptSource_t;
```

Description

Specifies the possible interrupts generated by Crm hardware.

Constant Values

The constant values are self explanatory.

2.4.9 crmTrimmedDevice_t

Constant Structure

```
typedef enum
{
     gBgapBatt_c,
     gBgap1P8V_c,
     gVReg0P9VTrim_c,
     gVReg0P9ITrim_c,
     gVRegXtalTrim_c,
     gVRegOscTrim_c,
     gTrimMax_c
}crmTrimmedDevice_t;
```

Description

Specifies the possible devices that can be trimmed using this driver.

Constant Values

The constant values are self explanatory.

2.4.10 crmXtalStartupDelay_t

Constant Structure

```
typedef enum
```

```
{
    gXSD512_c,
    gXSD1024_c,
    gXSD2048_c,
    gXSD4096_c,
    gXSD8192_c,
    gXSD16384_c,
    gXSD32768_c,
    gXSD65536_c
}crmXtalStartupDelay_t;
```

Description

Specifies the possible number of reference crystal cycles that will be counted before crystal starts.

Constant Values

The constant values are self explanatory.

2.4.11 crmBSCntl_t

Constant Structure

```
typedef enum
{
     gAutomaticBs_c,
     gManualBs_c,
     gDisableBs_c
}crmBSCntl_t;
```

Description

Specifies the possible values that can be chosen for bus stealing control.

Constant Values

The constant values are self explanatory.

2.4.12 crmWuSource_t

Constant Structure

```
typedef enum
{
     gExtWu_c,
     gTimerWu_c,
     gRtcWu_c
} crmWuSource_t;
```

Description

Specifies the possible wake up sources.

Constant Values

The constant values are self explanatory.

2.4.13 crmRamRet_t

Constant Structure

```
typedef enum
{
     gRamRet8k_c,
     gRamRet32k_c,
     gRamRet64k_c,
     gRamRet96k_c
} crmRamRet_t;
```

Description

Specifies the possible amount of ram that can be retained in low power modes.

Constant Values

The constant values are self explanatory.

2.4.14 crmMcuRet_t

Constant Structure

```
typedef enum
{
     gNoMcuRet_c,
     gMcuRet_c,
     gMcuAndDigPadRet_c=3
} crmMcuRet_t;
```

Description

Specifies the way that the MCU is retained in low power modes.

Constant Values

The constant values are self explanatory.

2.4.15 crmBuckClkDiv_t

Constant Structure

```
typedef enum
{
     gBuckClkDiv_16_c,
     gBuckClkDiv_8_c = 8,
     gBuckClkDiv_9_c,
     gBuckClkDiv_10_c,
     gBuckClkDiv_11_c,
     gBuckClkDiv_12_c,
     gBuckClkDiv_13_c,
     gBuckClkDiv_14_c,
     gBuckClkDiv_15_c,
}crmBuckClkDiv_t;
```

Description

Specifies the possible values of the divider for buck regulator clock.

Constant Values

The constant values are self explanatory.

2.4.16 crmExtWuEvent_t

Constant Structure

```
typedef enum
{
     gExtWuEventLevel_c,
     gExtWuEventEdge_c
}crmExtWuEvent_t;
```

Description

Specifies the possible external wake up events: level or edge detection.

Constant Values

- gExtWuEventLevel_c means that the wake up event will be the logical level selected by one of the ExtWuPol_t enumeration values
- gExtWuEventEdge_c means that the wake up event will be the edge selected by one of the ExtWuPol_t enumeration values

2.4.17 crmExtWuPol_t

Constant Structure

```
typedef enum
{
     gExtWuPolLLevel_or_NEdge_c,
     gExtWuPolHLevel_or_PEdge_c
}crmExtWuPol_t;
```

Description

Specifies the possible values of the external wakeup polarity.

Constant Values

- gExtWuPolLLevel_or_NEdge_c: in the case that the external event was set to gExtWuEventLevel_c this value means that the wake up is triggered by the low level; in the case that the external event was set to gExtWuEventEdge_c this value means that the wake up is triggered by the negative edge
- gExtWuPolHLevel_or_PEdge_c: in the case that the external event was set to gExtWuEventLevel_c this value means that the wake up is triggered by the high level; in the case that the external event was set to gExtWuEventEdge_c this value means that the wake up is triggered by the positive edge

2.4.18 crmSleep_t

Constant Structure

```
typedef struct
{
     gHibernate_c,
     gDoze_c
} crmSleep_t;
```

Description

Specifies the possible sleep modes: hibernate or doze.

Constant Values

The constant values are self explanatory.

2.5 Driver Exported Structures

2.5.1 crmWuCtrl_t

Structure

```
typedef struct
{
    crmWuSource_t wuSource;
    union
    {
        crmExtWuCtrl_t   ext;
        crmTimerWuCtrl_t timer;
        crmRtcWuCtrl_t   rtc;
    }ctrl;
}crmWuCtrl_t;
```

Description

Retains the parameters used to control the wake up.

2.5.1.1 Structure Elements

wuSource

Contains the wake up source to configure.

Ctrl

A union of three other structures that configure the wake up source defined by the wuSource.

2.5.2 crmExtWuCtrl_t

Structure

```
typedef union
{
    uint32_t word;
    struct
    {
        uint32_t  kbi4WuEn:1;
        uint32_t  kbi5WuEn:1;
        uint32_t  kbi6WuEn:1;
        uint32_t  kbi7WuEn:1;
        uint32_t  kbi4WuEvent:1;
        uint32_t  kbi5WuEvent:1;
        uint32_t  kbi6WuEvent:1;
        uint32_t  kbi7WuEvent:1;
        uint32_t  kbi4WuPol:1;
        uint32_t  kbi5WuPol:1;
        uint32_t  kbi6WuPol:1;
        uint32_t  kbi7WuPol:1;
```

```
            uint32_t  kbi4IntEn:1;
            uint32_t  kbi5IntEn:1;
            uint32_t  kbi6IntEn:1;
            uint32_t  kbi7IntEn:1;
        }bit;
        struct
        {
            uint32_t  kbiWuEn:4;
            uint32_t  kbiWuEvent:4;
            uint32_t  kbiWuPol:4;
            uint32_t  kbiIntEn:4;
        }nibble;
    }crmExtWuCtrl_t;
```

Description

Describes the configuration of the external wake up source.

2.5.2.1 Structure Elements

word

Allows this data to be used as an unsigned int.

bit

Allows this data to be used by bit.

nibble

Allows this data to be used by nibble.

- kbi4WuEn, kbi5WuEn, kbi6WuEn, kbi7WuEn: these bits control which of the four external are acted on to wake up the chip from the sleep states (kbijWuEn = 1 external wake up enabled, kbijWuEn = 0 external wake up disabled)
- kbi4WuEvent, kbi5WuEvent, kbi6WuEvent, kbi7WuEvent: these bits selects whether external wake up events will be edge, if 1, or level detected, if 0
- kbi4WuPol, kbi5WuPol, kbi6WuPol, kbi7WuPol: these polarity bits control the detection state of the external wake up inputs (kbijWuPol = 1 detect high level or positive edge, kbijWuPol = 0 detect low level or negative edge)
- kbi4IntEn, kbi5IntEn, kbi6IntEn, kbi7IntEn: these bits enable any of the four external wake up interrupts

2.5.3 crmTimerWuCtrl_t

Structure

```
typedef struct
{
    uint32_t timerWuEn:1;
    uint32_t timerWuIntEn:1;
```

```
    uint32_t timeOut;
}crmTimerWuCtrl_t;
```

Description

Describes the configuration of the sleep timer.

2.5.3.1 Structure Elements

timeOut	This value sets the sleep duration.
timerWuIntEn	This bit enables the sleep timer compare interrupt.
timerWuEn	This bit enables the sleep timer compare.

2.5.4 crmRtcWuCtrl_t

Structure

```
typedef struct
{
    uint32_t rtcWuEn:1;
    uint32_t rtcWuIntEn:1;
    uint32_t timeOut;
}crmRtcWuCtrl_t;
```

Description

Describes the configuration of the RTC timer.

2.5.4.1 Structure Elements

timeOut

This value sets the time that an RTC interrupt will be generated periodically.

rtcWuIntEn

This bit enables the RTC timer compare interrupts.

rtcWuEn

This bit enables the RTC timer compare.

2.5.5 crmSleepCtrl_t

Structure

```
typedef   struct
{
    uint8_t sleepType:1;
    uint8_t ramRet:2;
```

```
    uint8_t mcuRet:1;
    uint8_t digPadRet:1;
    pfCallback_t  pfToDoBeforeSleep;
}crmSleepCtrl_t;
```

Description

Configures the sleep mode.

2.5.5.1 Structure Elements

sleepType	This bit selects which sleep mode will be entered SleepType = 0 means hibernate SleepType = 1 means doze
ramRet	These bits pick the amount of RAM that will be retained during hibernate or doze. RamRet= 0 8 Kbytes RamRet= 1 32 Kbytes RamRet= 2 64 Kbytes RamRet= 3 96 Kbytes
mcuRet	Setting this bit will allow all the states of the MCU, Modem, and Analog Control to be saved during any sleep mode.
digPadRet	Setting this bit will allow power to the digital pad ring (all pads associated with GPIO) in sleep mode.
pToDoBeforeSleep	This is a pointer of pfCallback_t type (typedef void (*pfCallback_t)(void)).The function that it points to will be called just before entering sleep mode. Set this pointer to NULL and no function will be called before entering sleep mode. It is important to set This parameter to a function or to NULL because otherwise a call to the CrmGoToSleep function will have an unpredictable results.

2.5.6 crmBuckCntl_t

Structure

```
typedef   struct
{
    uint32_t buckEn:1;
    uint32_t buckSyncRecEn:1;
    uint32_t buckBypassEn:1;
    uint32_t buckClkDiv:4;
}crmBuckCntl_t;
```

Description

Configures the buck regulator.

2.5.6.1 Structure Elements

buckEn — When set this bit starts the buck regulator.The 1.5V and 1.8V regulators input are tied to the buck regulator output. After setting this bit the software must wait until buck regulator is ready prior to enable 1.5V or 1.8V regulators.

buckSyncRecEn — This bit enables the synchronous rectifier of the buck regulator.It should always be set when the buck regulator is enabled.

buckBypassEn — When 1 this bit causes the system to bypass the buck voltage regulator. This will need to be done when battery falls below 2,5V.Software must wait until the buck regulator bypass is ready prior to enable 1.5V or 1.8V regulators.

buckClkDiv — These bits select the integer clock divider between decimal 2 and 16 inclusively from the reference crystal clock.The code word needs to be properly selected so that the resultant buck clock will be as close to 1.6MHZ as possible.

2.5.7 crmVReg1P5VCntl_t

Structure

```
typedef   struct
{
    uint32_t vReg1P5VEn:2;
    uint32_t vReg1P5VISel:2;
}crmVReg1P5VCntl_t;
```

Description

Configures the 1.5V voltage regulator.

2.5.7.1 Structure Elements

vReg1P5VEn — These bits turn on and off the 1.5V voltage regulators that supply the analog functions. The possible selections can be seen in crmAnalogVRegEnable_t enumeration in CRM.h;

vReg1P5VISel — These bits control the amount of current sourced out of the analog voltage regulator if it is enabled by the previous field. The possible selections are defined by the crmAnalogVRegCurrentSel_t enumeration in CRM.h;

2.5.8 crmVRegCntl_t

Structure

```
typedef   struct
{
    crmVReg_t vReg;
    union
    {
        crmBuckCntl_t buckCntl;
        crmVReg1P5VCntl_t vReg1P5VCntl;
```

```
            bool_t vReg1P8VEn;
        }cntl;
    }crmVRegCntl_t;
```

Description

Configures one of the three voltage regulators:

- Buck regulator
- 1.5V voltage regulator
- 1.8V voltage regulator.

2.5.8.1 Structure Elements

vReg	This member establishes the voltage regulator configured by the structure.
cntl	This member is a union of configuration structures.
buckCntl	This structure was discussed above.
vReg1P5VCntl	This structure was discussed above.
vReg1P8VEn	A boolean value that establishes if 1.8V regulator is going to be enabled or disabled.

2.5.9 crmRefXtalCntl_t

Structure

```
typedef union
{
    uint32_t word;
    struct
    {
        uint32_t wp:1;
        uint32_t smallSig:1;
        uint32_t startupDelayBypass:1;
        uint32_t startupDelaySel:3;
        uint32_t refXtalDiv:6;
    }bit;
}crmRefXtalCntl_t;
```

Description

Configures the reference crystal oscillator.

2.5.9.1 Structure Elements

wp	Setting this bit to 1 will write protect the reference crystal configuration. Once set only a hard or soft reset can release the protection.
smallSig	This bit has to be set to 1 when the input clock source to the crystal circuit is a low voltage level. When the input voltage is greater then 1.0Vpp with a direct inject clock signal the software can set this bit to 0.This will save power.

startupDelayBypass	In normal startup, a timer in crystal will count a certain number of crystal clock cycles (equivalent to about 680 us).When this bit is set, this timer is bypassed and no wait will occur.
startupDelaySel	This field can take one of the values defined in crmXtalStartupDelay_t.
refXtalDiv	This value divides the rate of reference crystal to the entire MCU.It will not affect the rate of the Modem clock. The divisor value is refXtalDiv + 1.

2.5.10 crmModuleEnableStatus_t

Structure

```
typedef union
{
    uint32_t word;
    struct
    {
        uint32_t armEn:1;
        uint32_t macaEn:1;
        uint32_t asmEn:1;
        uint32_t spiEn:1;
        uint32_t gpioEn:1;
        uint32_t uart1En:1;
        uint32_t uart2En:1;
        uint32_t tmrEn:1;
        uint32_t rifEn:1;
        uint32_t i2cEn:1;
        uint32_t ssiEn:1;
        uint32_t spiFEn:1;
        uint32_t adcEn:1;
        uint32_t ahbEn:1;
        uint32_t jtagEn:1;
        uint32_t nexEn:1;
        uint32_t tmxEn:1;
        uint32_t aimEn:1;
    }bit;
}crmModuleEnableStatus_t;
```

Description

This data was defined as a union to be both word and bit accessible. It is used to report which module is enabled and which module is disabled in the MCU.

2.5.10.1 Structure Elements

Structure elements name are self explanatory.

2.5.11 crmCopCntl_t

Structure

```
typedef  union
{
    uint32_t word;
    struct
    {
        uint32_t copEn:1;
        uint32_t copOut:1;
        uint32_t copWP:1;
        uint32_t copTimeOut:7;
    } bit;
}crmCopCntl_t;
```

Description

This data was defined as a union to be both word and bit accessible. It is used to configure the COP counter.

2.5.11.1 Structure Elements

copEn	This bit enables/disable the cop counter.
copOut	The state of this bit determines the effect of the COP timeout state being detected and selects either the interrupt output or the reset output.
copWP	This bit controls the write protection feature of the COP control register.
copTimeOut	This value determines the timeout period of the COP counter.The COP counter counts Reference crystal clocks.

2.6 Exported Macros

2.6.1 CRM_WuTimerInterruptEnable ()

Enables the wake up timer interrupt.

2.6.2 CRM_WuTimerInterruptDisable ()

Disables the wake up timer interrupt.

2.6.3 CRM_CopReset ()

Resets the COP counter.

2.6.4 CRM_RTCInterruptEnable ()

Enables the RTC timer interrupt.

2.6.5 CRM_RTCInterruptDisable ()

Disables the RTC timer interrupt.

2.6.6 CRM_RefXtalConfigProtection ()

Write protects the reference crystal settings.

2.6.7 CRM_RingOscillatorDisable ()

Disables the 2 KHz ring oscillator.

2.6.8 CRM_RingOscillatorEnable ()

Enables the 2 KHz ring oscillator.

2.6.9 CRM_RTCSetTimeout (timeOut)

Sets the timeout value for the RTC timer.

2.6.10 CRM_Xtal32Enable ()

Starts the 32 KHz oscillator.

2.6.11 CRM_JtagNexusEnable ()

Enables the JTAG/NEXUS debugger.

2.6.12 CRM_JtagNexusDisable ()

Disables the JTAG/NEXUS debugger.

2.6.13 CRM_SoftReset ()

Resets the MCU.

2.6.14 CRM_VRegIsReady (vRegMask)

It can be used to test whether a specific regulator has started and is ready to be used. *vRegMask* can take one of the following values:

```
V_REG_MASK_BUCK
V_REG_MASK_1P5V
V_REG_MASK_1P8V
```

For other values a compiler error will be issued.

2.7 CRM Driver API Function Descriptions

2.7.1 CRM_GoToSleep ()

Prototype

```
void CRM_GoToSleep
(
        crmSleepCtrl_t *pSleepCtrl
);
```

Description

This function places the IC in sleep mode (either doze or hibernate) and configures the sleep mode parameters:

- If the sleep mode is doze or hibernate
- How much RAM is to be retained while in sleep mode
- If the MCU status is to be retained while in sleep mode
- If the digital pad ring is to be powered while in sleep mode
- A pointer to a function that will be called just before MCU enters sleep mode.If this pointer is NULL no function is called

CAUTION

Prior to call this function the wake up source has to be configured by using CRM_WuCntl function.

2.7.1.1 Function Parameters

Name

pSleepCtrl

In/Out

input

Description

A pointer to a crmSleepCtrl_t structure.This structure is presented in Exported structures section.

2.7.2 CRM_WuCntl ()

Prototype

```
crmErr_t CRM_WuCntl
(
        crmWuCtrl_t *pWuCtrl
);
```

Description

The function is called to configure the wake up sources.

- The *wuSource* field of the *crmWuCtrl* structure that *pWuCtrl* points to can take one of the following values: gExtWu_c, gTimerWu_c or gRtcWu_c. For any other value the function returns gCrmErrInvalidParameters_c
- For gExtWu_c the function configures the external wake up sources.Any of the KBI4, KBI5, KBI6 or KBI7 pins can be enabled/disabled to wake up the MCU, set to be active on an specific edge or level , set to produce an interrupt on wake up or not

- For gTimerWu_c the function configures the wake up timer.The wake up timer can be enabled/disabled to wake up the MCU after a specific timeout and to produce interrupt on wake up or not
- For gRtcWu_c the function configures the RTC timer.The RTC timer can be enabled/disabled to wake up the MCU after a specific timeout and to produce interrupt on wake up or not

CAUTION

This function shall be called prior to any call of the CRM_GoToSleep function.

2.7.2.1 Function Parameters

Name

pWuCtrl

In/Out

input

Description

A pointer to a crmWuCtrl_t structure.This structure is presented in Exported structures section.

2.7.2.2 Returns

Type

crmErr_t

Description

The status of the operation.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidParameters_c

2.7.3 CRM_ModuleEnStatus ()

Prototype

```
crmModuleEnableStatus_t CRM_ModuleEnStatus(void);
```

Description

The function reports which module is enabled and which module is disabled.

2.7.3.1 Returns

Type

crmModuleEnableStatus_t

Description

This structure is presented in Exported structures section.

2.7.4 CRM_CopCntl ()

Prototype

```
crmErr_t CRM_CopCntl
(
        crmCopCntl_t copCntl
);
```

Description

This function is called to configure the COP counter: enable/disable the COP counter, configure the COP to either generate an interrupt or reset the MCU in the event of a COP counter timeout , set the COP counter timeout period and write protect the content of the COP control register.If the COP control register is already protected when this function is called it returns gCrmErrCopCntlWP_c.

2.7.4.1 Function Parameters

Name

copCntl

In/Out

input

Description

This structure is presented in Exported structures section.

2.7.4.2 Returns

Type

crmErr_t

Description

The status of the operation.

Possible Values

- gCrmErrNoError_c
- gCrmErrCopCntlWP_c

2.7.5 CRM_ForceCopTimeout ()

Prototype

```
crmErr_t CRM_ForceCopTimeout (void);
```

Description

The function forces COP counter timeout.This function may be used to debug the COP counter interrupt handling portion of a user application. If the COP control register is write protected when this function is called it returns gCrmErrCopCntlWP_c.

2.7.5.1 Returns

Type

crmErr_t

Description

The status of the operation.

Possible Values

- gCrmErrNoError_c
- gCrmErrCopCntlWP_c

2.7.6 CRM_BusStealingCntl ()

Prototype

```
crmErr_t CRM_BusStealingCntl
(
        crmBSCntl_t bsCntl,
        bool_t armClkGate,
        uint8_t cyclesToSteal
);
```

Description

This function configures the bus cycle stealing from the ARM by the Radio.This function allow the user to specify whether they want to enable automatic or manual bus stealing, whether or not the ARM's clock should be turned off when it does not have control of the bus and what is the maximum number of consecutive bus cycles that can be stolen from the ARM.

2.7.6.1 Function Parameters

Name

bsCntl

In/Out

input

Description

This structure is presented in Exported structures section.

Name

armClkGate

In/Out

input

Description

This parameter selects whether the ARM's clock will be gated off when it does not have the control of the bus in order to save power.

Name

cyclesToSteal

In/Out

input

Description

This value sets the number of cycles to steal away from the ARM.The maximum number of cycles to steal is 64.

2.7.6.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidParameters_c

2.7.7 CRM_Wait4Irq ()

Prototype

```
void CRM_Wait4Irq (void);
```

Description

This function halts the ARM until an interrupt occurs.

2.7.8 CRM_RefXtalControl ()

Prototype

```
crmErr_t CRM_RefXtalControl
(
        crmRefXtalCntl_t xtalCntl
);
```

Description

The function controls the reference crystal.This function allows the user to set crystal input voltage level, crystal startup count, crystal startup count bypass ,crystal divider value and to write protect this values (except the crystal divider). If these values are already write protected the function returns gCrmErrXtalCntlWP_c.

2.7.8.1 Function Parameters

Name

xtalCntl

In/Out

input

Description

This structure is presented in Exported structures section.

2.7.8.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrXtalCntlWP_c

2.7.9 CRM_2kToXtal32Switch ()

Prototype

```
crmErr_t CRM_2kToXtal32Switch
(
        bool_t armHalted,
        pfCallback_t pfToCallback
);
```

Description

The function starts the 32 KHz oscillator. This function allows the user to specify whether the ARM will be halted or not until the oscillator is ready. If no halting is selected the user can provide a pointer to a callback function that will be called once the 32 KHz oscillator is up and running. If this pointer is NULL no function will be called.

CAUTION

The CRM interrupt must be enabled prior to call this function.

2.7.9.1 Function Parameters

Name

armHalted

In/Out

input

Description

Specifies whether the ARM will be halted or not until the 32KHz oscillator starts.

Name

pfCallback_t

In/Out

input

Description

A pointer to a function that will be called when the 32Khz is ready.

2.7.10 CRM_RingOscCal ()

Prototype

```
crmErr_t CRM_RingOscCal
(
        uint16_t calLength,
        bool_t haltMcu,
        void * pTR
);
```

Description

The function starts the calibration process for the 2 KHz ring oscillator.This function allow the user to specify how long the calibration process is to last and if the ARM will be halted or not until the calibration process is complete.If “no halting” is selected the user is allowed to provide a pointer to a callback function that will be called once the calibration process finishes.If “halting” is selected the function provides the value of the calibration register.If at the moment the function is called the ring oscillator is disabled the function returns gCrmErrRingOscOff_c.If a calibration process is already in progress the function returns gCrmErrCalInProgress_c.If the CalLength parameter is 0 the function returns gCrmErrInvalidParameters_c.

CAUTION

The CRM interrupt must be enabled prior to call this function.

2.7.10.1 Function Parameters

Name

calLength

In/Out

input

Description

Specifies how many ring oscillator cycles calibration will last

Name

haltMcu

In/Out

input

Description

Specifies whether or not the ARM will be halted until the calibration finishes.

Name

pTR

In/Out

input

Description

If the ARM is not halted pTR represents the address of the callback function that will be called when calibration finishes. If the ARM is halted until the calibration finishes the pRT represents the address of an uint32_t data where the function will deliver a value representing the number of reference crystal cycles that were necessary for calibration.

2.7.10.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrRingOscOff_c
- gCrmErrCalInProgress_c
- gCrmErrInvalidParameters_c

2.7.11 CRM_RingOscAbortCal ()

Prototype

```
crmErr_t CRM_RingOscAbortCal (void);
```

Description

The function is called to abort the Ring oscillator calibration process.If no calibration is in progress the function returns gCrmErrNoCalInProgress_c.

2.7.11.1 Returns

Type

crmErr_t

Description

The status of the operation.

Possible Values

- gCrmErrNoError_c
- gCrmErrNoCalInProgress_c

2.7.12 CRM_SetPowerSource ()

Prototype

```
crmErr_t CRM_SetPowerSource
(
        crmPowerSource_t pwSource
);
```

Description

The function sets the power configuration for the MCU. This settings are retained during sleep modes. There are three possible setting: gCrmPwS3V3Battery_c, gCrmPwSBuckRegulation_c, gCrmPwS1V8Battery_c. For any other values the function returns gCrmErrInvalidParameters_c.

2.7.12.1 Function Parameters

Name

pwSource

In/Out

input

Description

This enumeration is presented in Exported structures section.

2.7.12.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidParameters_c

2.7.13 CRM_VRegCntl ()

Prototype

```
crmErr_t CRM_VRegCntl
(
        crmVRegCntl_t* pVRegCntl
);
```

Description

The function is called to configure one of the following three voltage regulators:

- buck voltage regulator
- 1.5V voltage regulator
- 1.8V voltage regulator

For this, the vReg field of the crmVRegCntl structure has to be set to one of following values: gBuckVReg_c

- g1P5VReg_c
- g1P8VReg_c

If not, the function returns gCrmErrInvalidParameters_c.

- If Buck regulator is to be enabled and the power source was not set prior to gCrmBuckRegulation_c the function returns gCrmErrIgnoredInActualPowerMode_c.
- If buckBypassEn and buckEn are found both set in the buckCntl structure the function returns gCrmErrInvalidParameters_c.
- If vReg1P5VEn field in vReg1P5VCntl structure has other value other than gARegDisable_c, gRxTxRegEnable_c or gRxTxandPLLRegEnable_c the function returns gCrmErrInvalidParameters_c.
- If vReg1P5VISel field in vReg1P5VCntl structure has other value then gARegCurent4mA_c, gARegCurent20mA_c or gARegCurent40mA_c the function returns gCrmErrInvalidParameters_c.
- If PWR_SOURCE field in SYS_CNTL register is 2’b11 the function returns gCrmErrInvalidPowerSource_c.
- If PWR_SOURCE field in SYS_CNTL is gCrmPwSBuckRegulation_c, BUCK_EN and BUCK_BYPASS_EN bits in VREG_CNTL register are both 0 and the user tries to start the 1.8V or 1.5V regulator the function returns gCrmErrBuckNotEnabledNorBypassed_c.

2.7.13.1 Function Parameters

Name

pVRegCntl

In/Out

input

Description

This structure is presented in Exported structures section.

2.7.13.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidPowerSource_c
- gCrmErrBuckNotEnabledNorBypassed_c
- gCrmErrIgnoredInActualPowerMode_c
- gCrmErrInvalidParameters_c

2.7.14 CRM_VRegTrimm ()

Prototype

```
crmErr_t CRM_VRegTrimm
(
        crmTrimmedDevice_t trimmedDevice,
        uint8_t trimmValue
);
```

Description

The function is called to adjust the behavior of the specified device. If the trimmedDevice parameter has other value then the values specified in the crmTrimmedDevice_t enumeration or if the trimmValue is bigger than 15 the function returns gCrmErrInvalidParameters_c .

2.7.14.1 Function Parameters

Name

trimmedDevice

In/Out

input

Description

Specifies the device that is to be trimmed.

Name

trimmValue

In/Out

input

Description

This is the value for the parameter that will be adjusted.

2.7.14.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidParameters_c

2.7.15 CRM_SetDigOutDriveStrength ()

Prototype

```
crmErr_t CRM_SetDigOutDriveStrength
(
        crmPadsDriveStrength_t padsDriveStrength
);
```

Description

The function is called to set the drive strength for the digital pads. If padsDriveStrength parameter has other values then the values specified in the crmPadsDriveStrength_t enumeration the function returns gCrmErrInvalidParameters_c. If PWR_SOURCE field in SYS_CNTL register is 2'b11 the function returns gCrmErrInvalidPowerSource_c.

2.7.15.1 Function Parameters

Name

padsDriveStrength

In/Out

input

Description

Specifies whether the drive strength for the digital pads is to be standard drive or high drive.

2.7.15.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidParameters_c

2.7.16 CRM_SetSPIFDriveStrength ()

Prototype

```
crmErr_t CRM_SetSPIFDriveStrength
(
        crmPadsDriveStrength_t spiFDriveStrength
);
```

Description

The function is called to set the drive strength of the SPI interface to the internal flash. If spiFDriveStrength parameter has other values then the values specified in the crmPadsDriveStrength_t enumeration the

function returns gCrmErrInvalidParameters_c. If PWR_SOURCE field in SYS_CNTL register is 2'b11 the function returns gCrmErrInvalidPowerSource_c.

2.7.16.1 Function Parameters

Name

spiFDriveStrength

In/Out

input

Description

Specifies the drive strength of the SPI interface to the internal flash.

2.7.16.2 Returns

Type

CrmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidParameters_c

2.7.17 CRM_RegisterISR ()

Prototype

```
crmErr_t CRM_RegisterISR
(
        crmInterruptSource_t crmIS, pfCallback_t pfISR
);
```

Description

The function is called to register the function that pfISR points to as the interrupt software routine for the interrupt source specified by crmIS parameter.If the crmIS has other value then the values specified by the CrmInterruptSource_t enumeration, the function returns gCrmErrInvalidParameters_c. To deregister a callback call this function with pfISR = NULL.

2.7.17.1 Function Parameters

Name

crmIS

In/Out

input

Description

Specifies the interrupt source that will be associated with the function pointed by pfISR (next parameter).

Name

pfISR

In/Out

input

Description

This is a pointer to a function that will be called when the associated interrupt occurs.

2.7.17.2 Returns

Type

crmErr_t

Description

Function execution status.

Possible Values

- gCrmErrNoError_c
- gCrmErrInvalidParameters_c

2.7.18 CRM_Isr ()

Prototype

```
void CRM_Isr (void);
```

Description

The function will be set as interrupt routine for all the Crm interrupt sources. This will be done using the interface provided by the ITC driver.

Chapter 3
GPIO Driver

3.1 Overview

The MC1322x has a possible 64 GPIOs (general purpose digital inputs/outputs). Many of these pins are shared with on-chip peripherals such as the timer, external interrupts, or communication channels. When these other modules or functions are not using these pins, they can be configured as general-purpose I/O control. Each I/O port can be configured as an input or output, has programmable pullup or pulldown resistors, and has complete read and write capability. The GPIO Driver simplifies programming of the peripheral interfaces and use of the GPIO.

3.1.1 GPIO Hardware

The MC1322x device GPIO is controlled by a module that determines their use. The module has configuration registers that must be programmed to set the usage. For detailed information on the GPIO module, see the *MC1322x Reference Manual* (MC1322xRM).

The GPIO have the following features:

- 64 General-purpose IO pins
- Default to GPIO controlled as inputs with pulldown or pullup resistors enabled (except for JTAG/NEXUS and some analog pins)
- Programmable input hysteresis
- Software controlled pullups or pulldowns on all pins
- Software controlled state keepers on all pins
- Shared functionality with MCU peripherals or special functions
- The GPIO module/control generates no interrupts
- Eight “stay-alive” KBI signal for use during low power modes

3.1.2 Driver Functionality

As described in the previous section there are 64 possible GPIO. The MC1322x hardware does not organize the GPIO pins as dedicated ports as commonly done on other MCUs. The GPIO are mostly grouped by peripheral function as an alternative.

For driver efficiency and for consistency with GPIO control registers, the 64 GPIO have been defined as two 32-bit GPIO software ports designated as:

- GpioPort0 – which contains GPIO0 to GPIO31 with GPIO0 being the least significant bit position

- GpioPort1 – which contains GPIO32 to GPIO63 with GPIO32 being the least significant bit position

These definitions are used in the following sections.

The driver allows the developer to use the GPIO functions easily. It allows the developer to:

- Initialize the GPIO ports for functionality (GPIO or alternate function)
- Set the data direction for each GPIO pin
- Configure the pullup/pulldown settings for each GPIO pin
- Enable/clear/configure interrupts for the keyboard interrupt capable GPIO pins
- Set, clear or toggle a GPIO output
- Read a value from the specified GPIO pin(s)

3.2 Include Files

Table 3-1 shows the GPIO driver files that must be included in the application C-files in order to have access to the driver function calls.

Table 3-1. GPIO Driver Include Files

Include File Name	Description
GPIO_Interface.h	Global header file that defines the public data types and specifies the public functions prototypes

3.3 GPIO Driver API Functions

Table 3-2 shows the available API functions for the GPIO Driver.

Table 3-2. GPIO Driver API Function List

GPIO Driver API Function Name	Description
Gpio_InitPort ()	The function is called to set the initial port settings of the port *gpioPort* specified.
Gpio_WrPortSetting()	The function is called to modify the attribute *portAttr* of the port *gpioPort* specified.
Gpio_RdPortSetting()	The function is called to read the value of attribute *portAttr* of the *gpioPort* port specified
Gpio_SetPortDir ()	The function is called to set the direction value of the *gpioPort* port specified
Gpio_GetPortDir ()	The function is called to read the direction value of the *gpioPort* port specified.
Gpio_SetPinDir()	The function is called to set the direction value of the pin *gpioPin* specified.
Gpio_GetPinDir()	The function is called to read the direction value of the pin *gpioPin* specified.
Gpio_SetPortData()	The function is called to set the port data value of the *gpioPort* port specified
Gpio_GetPortData()	The function is called to read data value of the *gpioPort* port specified.
Gpio_SetPinData()	The function is called to set the data value of the pin *gpioPin* specified.
Gpio_GetPinData()	The function is called to read the direction value of the pin *gpioPin* specified.

Table 3-2. GPIO Driver API Function List (continued)

Gpio_TogglePin()	The function is called to toggle the state of the pin *gpioPin* specified.
Gpio_SetPinReadSource()	The function is called to set the read source of the pin *gpioPin* specified.
Gpio_GetPinReadSource()	The function is called to read the read source of the pin *gpioPin* specified.
Gpio_EnPinPullup()	The function is called to enable pull-up of the pin *gpioPin* specified.
Gpio_IsPinPullupEn()	The function is called to get the pull-up enable state of the pin *gpioPin* specified.
Gpio_SelectPinPullup()	The function is called to select pull-up/pull-down of the pin *gpioPin* specified.
Gpio_GetPinPullupSel ()	The function is called to get the pull-up selection state of the pin *gpioPin* specified.
Gpio_EnPinPuKeeper()	The function is called to enable pull-up keeper of the pin *gpioPin* specified.
Gpio_IsPinPuKeeperEn()	The function is called to get the pull-up keeper enable state of the pin *gpioPin* specified.
Gpio_EnPinHyst()	The function is called to enable hysteresis of the pin *gpioPin* specified.
Gpio_IsPinHystEn()	The function is called to get the hysteresis enable state of the pin *gpioPin* specified.
Gpio_SetPortFunction()	This function shall allow the user to set the functionality (GPIO or alternate function) of the GPIO port *gpioPort* specified.
Gpio_SetPinFunction()	The function is called to set function mode of the pin *gpioPin* specified.
Gpio_GetPinFunction()	The function is called to get function mode of the pin *gpioPin* specified.

3.4 Driver Exported Constants

3.4.1 GpioErr_t

Constant Structure

```
typedef enum
{
    gGpioErrNoError_c = 0,
    gGpioErrInvalidParameter_c
} GpioErr_t;
```

Description

Specifies the possible return values for the GPIO driver API functions.

Constant Values

The constant values are self explanatory.

3.4.2 GpioPort_t

Constant Structure

```
typedef enum
{
    gGpioPort0_c = 0,
    gGpioPort1_c,
    gGpioPortMax_c
} GpioPort_t;
```

Description

Specifies the possible values for the GPIO ports.

Constant Values

The constant values are self explanatory.

3.4.3 GpioDirection_t

Constant Structure

```
typedef enum
{
    gGpioDirIn_c = 0,
    gGpioDirOut_c,
    gGpioDirMax_c
} GpioDirection_t;
```

Description

Specifies the possible values for the GPIO pad direction.

Constant Values

The constant values are self explanatory.

3.4.4 GpioPinState_t

Constant Structure

```
typedef enum
{
    gGpioPinStateLow_c = 0,
    gGpioPinStateHigh_c,
    gGpioPinStateMax_c,
} GpioPinState_t;
```

Description

Specifies the possible values for the GPIO pad state.

Constant Values

The constant values are self explanatory.

3.4.5 GpioPinReadSel_t

Constant Structure

```
typedef enum
{
    gGpioPinReadPad_c = 0,
    gGpioPinReadReg_c,
    gGpioPinReadMax_c
} GpioPinReadSel_t;
```

Description

Specifies the possible values for the GPIO read pad source.

Constant Values

The constant values are self explanatory.

3.4.6 GpioPinPullupSel_t

Constant Structure

```
typedef enum
{
    gGpioPinPulldown_c = 0,
    gGpioPinPullup_c,
    gGpioPinPullupMax_c
} GpioPinPullupSel_t;
```

Description

Specifies the possible values for the GPIO pad pullup selection.

Constant Values

The constant values are self explanatory.

3.4.7 GpioFunctionMode_t

Constant Structure

```
typedef enum
{
    gGpioNormalMode_c = 0,
    gGpioAlternate1Mode_c,
    gGpioAlternate2Mode_c,
    gGpioAlternate3Mode_c,
    gGpioFunctionModeMax_c
```

```
} GpioFunctionMode_t;
```

Description

Specifies the GPIO functionality mode: GPIO or alternate mode.

Constant Values

The constant values are self explanatory.

3.4.8 GpioPin_t

Constant Structure

```
typedef enum
{
    gGpioPin0_c,   gGpioPin1_c,  gGpioPin2_c,  gGpioPin3_c,  gGpioPin4_c,  gGpioPin5_c,
gGpioPin6_c, gGpioPin7_c,
    gGpioPin8_c,   gGpioPin9_c, gGpioPin10_c, gGpioPin11_c, gGpioPin12_c, gGpioPin13_c,
gGpioPin14_c, gGpioPin15_c,
    gGpioPin16_c, gGpioPin17_c, gGpioPin18_c, gGpioPin19_c, gGpioPin20_c, gGpioPin21_c,
gGpioPin22_c, gGpioPin23_c,
    gGpioPin24_c, gGpioPin25_c, gGpioPin26_c, gGpioPin27_c, gGpioPin28_c, gGpioPin29_c,
gGpioPin30_c, gGpioPin31_c,
    gGpioPin32_c, gGpioPin33_c, gGpioPin34_c, gGpioPin35_c, gGpioPin36_c, gGpioPin37_c,
gGpioPin38_c, gGpioPin39_c,
    gGpioPin40_c, gGpioPin41_c, gGpioPin42_c, gGpioPin43_c, gGpioPin44_c, gGpioPin45_c,
gGpioPin46_c, gGpioPin47_c,
    gGpioPin48_c, gGpioPin49_c, gGpioPin50_c, gGpioPin51_c, gGpioPin52_c, gGpioPin53_c,
gGpioPin54_c, gGpioPin55_c,
    gGpioPin56_c, gGpioPin57_c, gGpioPin58_c, gGpioPin59_c, gGpioPin60_c, gGpioPin61_c,
gGpioPin62_c, gGpioPin63_c,
    gGpioPinMax_c
} GpioPin_t;
```

Description

Specifies the possible values for GPIO pins.

Constant Values

The constant values are self explanatory.

3.4.9 GpioPortAttr_t

Constant Structure

```
typedef enum
{
    gGpioDirAttr_c = 0,
    gGpioDataAttr_c ,
    gGpioInputDataSelAttr_c,
    gGpioPullUpEnAttr_c ,
```

```
    gGpioPullUpSelAttr_c,
    gGpioHystEnAttr_c,
    gGpioPullUpKeepAttr_c,
    gGpioMaxAttr_c
}GpioPortAttr_t;
```

Description

Specifies the possible values for attributes of a port.

Constant Values

The constant values are self explanatory.

3.5 Exported Structures and Data Types

3.5.1 GpioPortInit_t

Structure

```
typedef struct
{
    uint32_t portDir;
    uint32_t portData;
    uint32_t portInputDataSel;
    uint32_t portPuEn;
    uint32_t portPuSel;
    uint32_t portHystEn;
    uint32_t portPuKeepEn;
}GpioPortInit_t;
```

Description

This structure writes initial settings for a GPIO port.

3.5.1.1 Structure Elements

portDir	Specifies the direction value for pins of a port.
portData	Specifies the data value for pins of a port.
portInputDataSel	Specifies the read source value for pins of a port.
portPuEn	Specifies the pull-up enable value for pins of a port.
portPuSel	Specifies the pull-up selection value for pins of a port.
portHystEn	Specifies the hysteresis enable value for pins of a port.
portPuKeepEn	Specifies the pull-up keeper enable value for pins of a port.

3.6 GPIO Driver API Function Descriptions

3.6.1 Gpio_InitPort ()

Prototype

```
GpioErr_t Gpio_InitPort
(
        GpioPort_t gpioPort,
        GpioPortInit_t *gpioPortInit
);
```

Description

The function is called to set the initial port settings of the port *gpioPort* specified as parameter. Initial settings are passed to the function through *gpioPortInit* structure pointer. The user can set initial port direction, data, read source, pull-up enable, pull-up selection, pull-up keeper and hysteresis enable and the function can be called usually at program start-up.

There are some tests performed before setting the initial values. If the port *gpioPort* specified as parameter is grater or equal than *gGpioPortMax_c* or *gpioPortInit* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function sets the initial port settings and returns the *gAdcErrNoError_c* value.

3.6.1.1 Function Parameters

Name

gpioPort

In/Out

input

Description

Port which will be affected by the function,

Name

gpioPortInit

In/Out

input

Description

Pointer to a structure that hold the initial port settings.

3.6.1.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.2 Gpio_WrPortSetting ()

Prototype

```
GpioErr_t Gpio_WrPortSetting
(
        GpioPort_t gpioPort,
        GpioPortAttr_t portAttr,
        uint32_t portAttrValue,
        uint32_t mask
);
```

Description

The function is called to modify the attribute *portAttr* of the port *gpioPort* specified as parameter. The *mask* mask specifies which pins will be affected by the modification.

There are some tests performed before writing the new settings. If the port *gpioPort* specified as parameter is grater or equal than *gGpioPortMax_c* or *portAttr* is grater or equal than *gMaxAttr_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function modifies port's attribute and returns the *gAdcErrNoError_c* value.

3.6.2.1 Function Parameters

Name

gpioPort

In/Out[1]

input

Description

The port which will be affected by the function.

Name

portAttr

In/Out

input

Description

The port attribute that should be modified.

Name

portAttrValue

In/Out

input

Description

The new port attribute value.

Name

mask

In/Out

input

Description

Specifies which pins of the port *gpioPort* should be affected by the function.

3.6.2.2 Returns

Type

GpioErr_t

1.

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.3 Gpio_RdPortSetting ()

Prototype

```
GpioErr_t Gpio_RdPortSetting
(
        GpioPort_t gpioPort,
        GpioPortAttr_t portAttr,
        uint32_t *portAttrValue
);
```

Description

The function is called to read the value of attribute *portAttr* of the *gpioPort* port specified as parameter.

There are some tests performed before reading the *portAttr* settings. If the port *gpioPort* specified as parameter is grater or equal than *gGpioPortMax_c* or *portAttr* is grater or equal than *gMaxAttr_c* or *portAttrValue* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function reads the port attribute and returns the *gAdcErrNoError_c* value.

3.6.3.1 Function Parameters

Name

gpioPort

In/Out

input

Description

The port which will be affected by the function.

Name

portAttr

In/Out

input

Description

The port attribute that should be modified.

Name

portAttrValue

In/Out

output

Description

The pointer to the location where attribute value will be placed.

3.6.3.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.4 Gpio_SetPortDir ()

Prototype

```
GpioErr_t Gpio_SetPortDir
(
        GpioPort_t gpioPort,
        uint32_t portDir,
        uint32_t mask
);
```

Description

The function is called to set the direction value of the *gpioPort* port specified as parameter. The direction is modified according with *portDir* and *mask* values. For example, if the GPIO pin 31 is input pin and

GPIO pin 15 is output pin, the value of *gpioPort* should be gGpioPort0_c, the value of *portDir* should be 0x00008000 and the value of *mask* should be 0x80008000.

There are some tests performed before setting the port direction. If the port *gpioPort* specified as parameter is grater or equal than *gGpioPortMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function writes the port direction value and returns the *gAdcErrNoError_c* value.

3.6.4.1 Function Parameters

Name

gpioPort

In/Out

input

Description

The GPIO port to set direction.

Name

portDir

In/Out

input

Description

The GPIO port direction value.

Name

mask

In/Out

input

Description

Specifies which pins of the port *gpioPort* should be affected by the function.

3.6.4.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.5 Gpio_GetPortDir ()

Prototype

```
GpioErr_t Gpio_GetPortDir
(
        GpioPort_t gpioPort,
        uint32_t *portDir
);
```

Description

The function is called to read the direction value of the *gpioPort* port specified as parameter.

There are some tests performed before reading the port direction. If the port *gpioPort* specified as parameter is grater or equal than *gGpioPortMax_c* or *portDir* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function reads the port direction value and returns the *gAdcErrNoError_c* value.

3.6.5.1 Function Parameters

Name

gpioPort

In/Out

input

Description

The GPIO port to get direction.

Name

portDir

In/Out

output

Description

Pointer to memory location where GPIO port direction value will be placed.

3.6.5.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.6 Gpio_SetPinDir ()

Prototype

```
GpioErr_t Gpio_SetPinDir
(
        GpioPin_t gpioPin,
        GpioDirection_t gpioPinDir
);
```

Description

The function is called to set the direction value of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin direction. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinDir* is grater or equal than *gGpioDirMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function writes the pin direction value and returns the *gAdcErrNoError_c* value.

3.6.6.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to set direction.

Name

gpioPinDir

In/Out

input

Description

The pin direction value.

3.6.6.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.7 Gpio_GetPinDir ()

Prototype

```
GpioErr_t Gpio_GetPinDir
(
        GpioPin_t gpioPin,
        GpioDirection_t *gpioPinDir
);
```

Description

The function is called to read the direction value of the pin *gpioPin* specified as parameter.

There are some tests performed before reading the pin direction. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinDir* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function reads the pin direction value and returns the *gAdcErrNoError_c* value.

3.6.7.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to get direction.

Name

gpioPinDir

In/Out

output

Description

The pin direction value.

3.6.7.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.8 Gpio_SetPortData ()

Prototype

```
GpioErr_t Gpio_SetPortData
(
        GpioPort_t gpioPort,
        uint32_t  portData,
        uint32_t mask
);
```

Description

The function is called to set the port data value of the *gpioPort* port specified as parameter. The data is modified according with *portData* and *mask* values. For example, if the GPIO pin 31 is set in LOW state and GPIO pin 15 is set in HIGH state, the value of *gpioPort* should be gGpioPort0_c, the value of *portData* should be 0x00008000 and the value of *mask* should be 0x80008000.

There are some tests performed before setting the port data. If the port *gpioPort* specified as parameter is grater or equal than *gGpioPortMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function writes the port data value and returns the *gAdcErrNoError_c* value. Only the pins unmasked by the mask *mask* are affected by the function.

3.6.8.1 Function Parameters

Name

gpioPort

In/Out

input

Description

The GPIO port to set data.

Name

portData

In/Out

input

Description

GPIO port data value

Name

mask

In/Out

input

Description

Specifies which pins of the port *gpioPort* should be affected by the function.

3.6.8.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.9 Gpio_GetPortData ()

Prototype

```
GpioErr_t Gpio_GetPortData
(
        GpioPort_t gpioPort,
        uint32_t *portData
```

```
);
```

Description

The function is called to read data value of the *gpioPort* port specified as parameter.

There are some tests performed before reading the port data. If the port *gpioPort* specified as parameter is grater or equal than *gGpioPortMax_c* or *portData* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function reads the port data value and returns the *gAdcErrNoError_c* value.

3.6.9.1 Function Parameters

Name

gpioPort

In/Out

input

Description

The GPIO port to get data.

Name

portData

In/Out

output

Description

Pointer to memory location where GPIO port data value will be placed.

3.6.9.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.10 Gpio_SetPinData ()

Prototype

```
GpioErr_t Gpio_SetPinData
(
        GpioPin_t gpioPin,
        GpioPinState_t gpioPinState
);
```

Description

The function is called to set the data value of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinState* is grater or equal than *gGpioPinStateMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function writes the pin data value and returns the *gAdcErrNoError_c* value.

3.6.10.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to set data.

Name

gpioPinState

In/Out

input

Description

Pin data value (gGpioPinStateLow_c for LOW level, gGpioPinStateHigh_c for HIGH level)

3.6.10.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.11 Gpio_GetPinData ()

Prototype

```
GpioErr_t Gpio_GetPinData
(
        GpioPin_t gpioPin,
        GpioPinState_t *gpioPinState
);
```

Description

The function is called to read the data value of the pin *gpioPin* specified as parameter.

There are some tests performed before reading the pin state value. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinState* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function reads the pin state value and returns the *gGpioErrNoError_c* value.

3.6.11.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to get data.

Name

gpioPinState

In/Out

output

Description

Pin state value (gGpioPinStateLow_c for LOW level, gGpioPinStateHigh_c for HIGH level).

3.6.11.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.12 Gpio_TogglePin ()

Prototype

```
GpioErr_t Gpio_TogglePin
(
        GpioPin_t gpioPin
);
```

Description

The function is called to toggle the state of the pin *gpioPin* specified as parameter.

There are some tests performed before toggling the pin state. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function toggles pin state and returns the *gAdcErrNoError_c* value.

3.6.12.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to toggle state.

3.6.12.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.13 Gpio_SetPinReadSource ()

Prototype

```
GpioErr_t Gpio_SetPinReadSource
(
        GpioPin_t gpioPin,
        GpioPinReadSel_t gpioReadSource
);
```

Description

The function is called to set the read source of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioReadSource* is grater or equal than *gGpioPinReadMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function writes the pin read source and returns the *gAdcErrNoError_c* value.

3.6.13.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to set read source.

Name

gpioReadSource

In/Out

input

Description

Pin read source (gGpioPinReadPad_c for reading data from pin, gGpioPinReadReg_c for reading data from latch).

3.6.13.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.14 Gpio_GetPinReadSource ()

Prototype

```
GpioErr_t Gpio_GetPinReadSource
(
        GpioPin_t gpioPin,
        GpioPinReadSel_t *gpioReadSource
```

```
);
```

Description

The function is called to get the read source of the pin *gpioPin* specified as parameter.

There are some tests performed before reading the pin direction. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioReadSource* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function gets the pin read source and returns the *gAdcErrNoError_c* value.

3.6.14.1 Function Parameters

Name

gpioPin

In/Out

input

Description

GPIO pin to get data

Name

gpioReadSource

In/Out

output

Description

Pin read source value (gGpioPinReadPad_c for reading data from pin, gGpioPinReadReg_c for reading data from latch)

3.6.14.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.15 Gpio_EnPinPullup()

Prototype

```
GpioErr_t Gpio_EnPinPullup
(
        GpioPin_t gpioPin,
        bool_t gpioEnPinPu
);
```

Description

The function is called to enable pull-up of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function enables the pin pull-up and returns the *gAdcErrNoError_c* value.

3.6.15.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to enable pull-up.

Name

gpioEnPinPu

In/Out

input

Description

The pin pull-up enable (TRUE for enabling pull-up, FALSE for disabling pull-up).

3.6.15.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.16 Gpio_IsPinPullupEn ()

Prototype

```
GpioErr_t Gpio_IsPinPullupEn
(
        GpioPin_t gpioPin,
        bool_t *gpioEnPinPu
);
```

Description

The function is called to get the pull-up enable state of the pin *gpioPin* specified as parameter.

There are some tests performed before reading the pin direction. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioEnPinPu* is NULL the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function gets the pull-up enable state and returns the *gAdcErrNoError_c* value.

3.6.16.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to get pull-up enable state (enabled/disabled).

Name

gpioEnPinPu

In/Out

output

Description

The pin pull-up enable state (TRUE for pull-up enabled, FALSE for pull-up disabled).

3.6.16.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.17 Gpio_SelectPinPullup ()

Prototype

```
GpioErr_t Gpio_SelectPinPullup
(
        GpioPin_t gpioPin,
        GpioPinPullupSel_t gpioPinPuSel
);
```

Description

The function is called to select pull-up/pull-down of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinPuSel* is grater or equal than *gGpioPinPullupMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function set the pin pull-up selection and returns the *gAdcErrNoError_c* value.

3.6.17.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to select pull-up/pull-down.

Name

gpioPinPuSel

In/Out

input

Description

The pin pull-up selection (gGpioPinPulldown_c for pull-down selection, gGpioPinPullup_c for pull-up selection).

3.6.17.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.18 Gpio_GetPinPullupSel ()

Prototype

```
GpioErr_t Gpio_GetPinPullupSel
(
        GpioPin_t gpioPin,
        GpioPinPullupSel_t *gpioPinPuSel
```

```
);
```

Description

The function is called to get the pull-up selection state of the pin *gpioPin* specified as parameter.

There are some tests performed before reading the pin direction. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinPuSel* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function gets the pull-up selection and returns the *gAdcErrNoError_c* value.

3.6.18.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to get pull-up selection state (pull-up/pull-down).

Name

gpioPinPuSel

In/Out

output

Description

The pin pull-up selection state (gGpioPinPulldown_c for pull-down selection, gGpioPinPullup_c for pull-up selection).

3.6.18.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.19 Gpio_EnPinPuKeeper ()

Prototype

```
GpioErr_t Gpio_EnPinPuKeeper
(
        GpioPin_t gpioPin,
        bool_t gpioEnPinPuKeep
);
```

Description

The function is called to enable pull-up keeper of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function enables the pin pull-up keeper and returns the *gAdcErrNoError_c* value.

3.6.19.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to enable pull-up keeper.

Name

gpioEnPinPuKeep

In/Out

input

Description

The pin pull-up keeper enable (TRUE for enabling pull-up keeper, FALSE for disabling pull-up keeper).

3.6.19.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.20 Gpio_IsPinPuKeeperEn ()

Prototype

```
GpioErr_t Gpio_IsPinPuKeeperEn
(
        GpioPin_t gpioPin,
        bool_t *gpioEnPinPuKeep
);
```

Description

The function is called to get the pull-up keeper enable state of the pin *gpioPin* specified as parameter.

There are some tests performed before reading the pin direction. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinEnPuKeep* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function gets the pull-up keeper enable state and returns the *gAdcErrNoError_c* value.

3.6.20.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to get pull-up keeper enable state (enabled/disabled).

Name

gpioEnPinPuKeep

In/Out

output

Description

The pin pull-up keeper enable state (TRUE for pull-up keeper enabled, FALSE for pull-up keeper disabled).

3.6.20.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.21 Gpio_EnPinHyst ()

Prototype

```
GpioErr_t Gpio_EnPinHyst
(
        GpioPin_t gpioPin,
        bool_t gpioEnPinHyst
);
```

Description

The function is called to enable hysteresis of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function enables the hysteresis and returns the *gAdcErrNoError_c* value.

3.6.21.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to enable hysteresis.

Name

gpioEnPinHyst

In/Out

input

Description

The pin hysteresis enable (TRUE for enabling hysteresis, FALSE for disabling hysteresis).

3.6.21.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.22 Gpio_IsPinHystEn ()

Prototype

```
GpioErr_t Gpio_IsPinHystEn
(
        GpioPin_t gpioPin,
        bool_t *gpioEnPinHyst
);
```

Description

The function is called to get the hysteresis enable state of the pin *gpioPin* specified as parameter.

There are some tests performed before reading the pin direction. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioEnPinHyst* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function gets the hysteresis enable state and returns the *gAdcErrNoError_c* value.

3.6.22.1 Function Parameters

Name

gpioPin

In/Out

input

Description

The GPIO pin to get hysteresis enable state (enabled/disabled).

Name

gpioEnPinHyst

In/Out

output

Description

The pin hysteresis enable state (TRUE for hysteresis enabled, FALSE for hysteresis disabled).

3.6.22.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c

- gGpioErrInvalidParamater_c

3.6.23 Gpio_SetPortFunction ()

Prototype

```
GpioErr_t Gpio_SetPortFunction
(
        GpioPort_t gpioPort,
        GpioFunctionMode_t gpioPortFunction,
        uint32_t mask
);
```

Description

This function shall allow the user to set the functionality (GPIO or alternate function) of the GPIO port *gpioPort* specified as parameter. Only masked pins will be set with function mode *gpioPortFunction* specified as parameter.

There are some tests performed before writing port function. If the *gpioPort* port specified as parameter is grater or equal than *gGpioPortMax_c* or *gpioPortFunction* is grater or equal than *gGpioFunctionModeMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function writes the function mode and returns the *gAdcErrNoError_c* value.

3.6.23.1 Function Parameters

Name

gpioPort

In/Out

input

Description

The GPIO port to set function mode.

Name

gpioPortFunction

In/Out

input

Description

The port function mode.

Name

mask

In/Out

input

Description

The mask used for setting function mode.

3.6.23.2 Returns

Type

GpioErr_t

Description

Function execution status.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.24 Gpio_SetPinFunction ()

Prototype

```
GpioErr_t Gpio_SetPinFunction
(
        GpioPin_t gpioPinNo,
        GpioFunctionMode_t gpioPinFunction
);
```

Description

The function is called to set function mode of the pin *gpioPin* specified as parameter.

There are some tests performed before setting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinFunction* is grater or equal than *gGpioFunctionModeMax_c* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function sets the function mode and returns the *gAdcErrNoError_c* value.

3.6.24.1 Function Parameters

Name

gpioPinNo

In/Out

input

Description

The GPIO pin to set function mode.

Name

gpioPinFunction

In/Out

input

Description

This parameter specifies the pin function mode.

3.6.24.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c
- gGpioErrInvalidParamater_c

3.6.25 Gpio_GetPinFunction ()

Prototype

```
GpioErr_t Gpio_GetPinFunction
(
        GpioPin_t gpioPinNo,
        GpioFunctionMode_t* gpioPinFunction
);
```

Description

The function is called to get function mode of the pin *gpioPin* specified as parameter.

There are some tests performed before getting the pin data. If the *gpioPin* pin specified as parameter is grater or equal than *gGpioPinMax_c* or *gpioPinFunction* is *NULL* the function exits with *gGpioErrInvalidParamater_c* value.

If all of these tests have been passed, the function gets the function mode and returns the *gAdcErrNoError_c* value.

3.6.25.1 Function Parameters

Name

gpioPinNo

In/Out

input

Description

The GPIO pin to get function mode.

Name

gpioPinFunction

In/Out

input

Description

This parameter specifies the function mode.

3.6.25.2 Returns

Type

GpioErr_t

Description

The status of the operation.

Possible Values

- gGpioErrNoError_c

- gGpioErrInvalidParamater_c

Chapter 4
Interrupt Controller (ITC) Driver

4.1 Overview

The MC1322x MCU interrupt structure is based fundamentally on the ARM7TDMI-S processor. The interrupt control is distributed across the peripheral devices, the Interrupt Controller (ITC), and the CPU control. The ITC Driver simplifies programming of the ITC module.

4.1.1 MC1322x Interrupt Request Service and the ITC Module

The MCU interrupt request service can be viewed as comprised of three basic functions as illustrated in Figure 4-1.

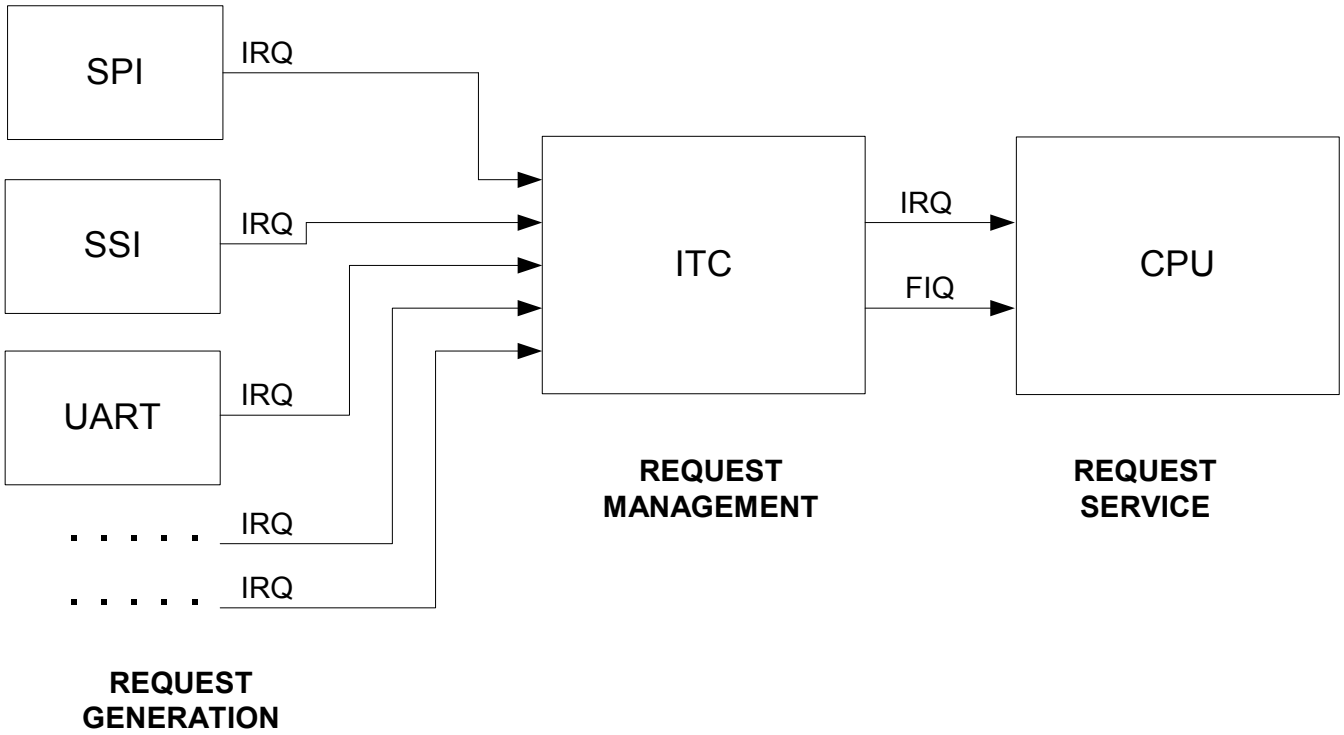

Figure 4-1. Interrupt Request Service Functions

- Request Generation - An interrupt request typically gets generated at a peripheral device. The peripheral control determines the source(s) of the interrupt request. There is a single request per peripheral

- Request Management - The Interrupt Controller (ITC) accepts the interrupt request from each module source. These multiple requests are prioritized and mapped to one of two basic requests to the CPU, i.e., the fast interrupt request (FIQ) and the normal interrupt request (IRQ)
- Request Service - The CPU has the final responsibility of servicing the interrupt requests that have been mapped into the two simple signals of FIQ and IRQ

The interrupt service is controlled as well as disabled or enabled across these three functions.

4.1.2 Request Service

The ARM7TDMI-S processor has seven operating modes and two of these are used to service interrupts:

- Fast interrupt (FIQ) mode supports a data transfer or channel process - The Fast Interrupt Request (FIQ) exception supports data transfers or channel processes and is caused by a request on the FIQ internal signal
- Interrupt (IRQ) mode is used for general-purpose interrupt handling - The Interrupt Request (IRQ) exception is a normal interrupt caused by a request on the IRQ internal signal. IRQ has a lower priority than FIQ, and is masked on entry to an FIQ sequence. IRQ can be disabled at any time, by setting the I bit in the CPSR from a privileged mode

The interrupt modes are entered as exception processes. Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example to service an interrupt from a peripheral. There are seven exception conditions and when multiple exceptions arise at the same time, a fixed priority system determines the order in which they are handled.

4.1.3 Interrupt Request Generation

Interrupt requests are generated within a peripheral function. Each function has all its internal interrupt request sources channeled to its single IRQ signal. The ITC then priorities these requests and maps them to either the IRQ or FIRQ inputs to the CPU.

4.1.4 ITC Block

The ITC is a peripheral function that resides on the CPU data bus which collects interrupt requests from the peripheral sources and provides an interface to the CPU core. The ITC consists of a set of control registers and associated logic to perform interrupt masking of normal and fast interrupts. The priority levels can be controlled by software by simply masking interrupts.

The ITC provide the following features:

- Supports all internal interrupt sources
- Maps sources to normal or fast interrupt requests
- Indicates pending interrupt sources via a register for normal and fast interrupts
- Indicates highest priority interrupt number via register
- Independently enables/disables any interrupt source
- Provides a mechanism for software to force an interrupt

The MC1322x ITC has configuration registers that must be programmed to set the usage. For detailed information on the ITC module, see the *MC1322x Reference Manual* (MC1322xRM).

4.1.5 Driver Functionality

The driver allows the developer to use the ITC functions easily. It allows the developer to:

- Initialize the ITC
- Assign/remove a handler function for every interrupt source connected to the ITC
- Set priority for every interrupt source connected to ITC
- Read the priority for every interrupt source connected to ITC
- Test every interrupt source
- Set the minimum priority needed at a moment to trigger an interrupt
- Enable/disable every interrupt source
- Read status of every interrupt source, if it is enabled, disabled or pending
- Install/remove exception handlers for ARM (undefined instruction, SWI, prefetch and data abort)
- Enable/disable IRQ/FIQ interrupt handler functions
- Read current state of IRQ and FIQ ARM interrupt enabling flags

4.2 Include Files

Table 4-1 shows the GPIO driver files that must be included in the application C-files in order to have access to the driver function calls.

Table 4-1. GPIO Driver Include Files

Include File Name	Description
ITC_Interface.h	Global header file that defines the public data types and enumerates the public functions prototypes
Interrupt.h	Header file that defines the private data types and enumerates the private function prototypes

4.3 ITC Driver API Functions

Table 4-2 shows the available API functions for the ITC Driver.

Table 4-2. ITC Driver API Function List

ITC Driver API Function Name	Description
ITC_Init()	This function initializes the ARM Interrupt Control module hardware.
IntAssignHandler ()	This function assigns an interrupt handler to an IRQ number.
IntGetHandler()	This function returns the handler for an IRQ number.

Table 4-2. ITC Driver API Function List (continued)

ITC_SetPriority()	This function sets the priority to an IRQ number.
ITC_EnableInterrupt()	This function enables the interrupt corresponding to an IRQ number.
ITC_DisableInterrupt()	This function disables the interrupt corresponding to an IRQ number.
ITC_SetIrqMinimumLevel()	This function sets the normal interrupt level mask register – NIMASK.

4.4 Driver Exported Constants and Data Types

4.4.1 ItcErr_t

Constant Structure

```
typedef enum
{
    gItcErr_OK_c,
    gItcErr_InvalidNumber_c,
    gItcErr_InvalidPriority_c
} ItcErr_t;
```

Description

Specifies the possible return values for most of the ITC driver API functions.

Constant Values

- gItcErr_OK_c — no error
- gItcErr_InvalidNumber_c — wrong IRQ or exception number
- gItcErr_InvalidPriority_c — wrong priority

4.4.2 ItcPriority_t

Constant Structure

```
typedef enum
{
    gItcNormalPriority_c,
    gItcFastPriority_c
} ItcPriority_t;
```

Description

Specifies the two possible values of the priority: normal and fast.

Constant Values

- gItcNormalPriority_c — The corresponding peripheral will produce an IRQ interrupt
- gItcFastPriority_c — The corresponding peripheral will produce an FIQ interrupt

4.4.3 ItcNumber_t

Constant Structure

```
typedef enum
{
    gAsmInt_c,
    gUart1Int_c,
    gUart2Int_c,
    gCrmInt_c,
    gI2cInt_c,
    gTmrInt_c,
    gFlashInt_c,
    gMacaInt_c,
    gSsiInt_c,
    gAdcInt_c,
    gSpiInt_c,
    gMaxInt_c,
    gUndefinedException_c,
    gSupervisorException_c,
    gPrefetchAbortException_c,
    gDataAbortException_c,
    gMaxException_c
} ItcNumber_t;
```

Description

Specifies the possible values for the interrupt sources.

Constant Values

- gAsmInt_c — interrupt source is the ASM
- gUart1Int_c — interrupt source is the UART1 module
- gUart2Int_c — interrupt source is the UART2 module
- gCrmInt_c — interrupt source is the CRM
- gI2cInt_c — interrupt source is the I2C Module
- gTmrInt_c — interrupt source is the Timer Module
- gFlashInt_c — interrupt source is the internal SPI Flash Module
- gMacaInt_c — interrupt source is the MACA
- gSsiInt_c — interrupt source is the SSI Module
- gAdcInt_c — interrupt source is the ADC Module
- gSpiInt_c — interrupt source is the SPI Module
- gMaxInt_c — values equal or greater than this one are not allowed for ITC
- gUndefinedException_c — undefined instruction exception
- gSupervisorException_c — SWI exception
- gPrefetchAbortException_c — prefetch abort exception

- gDataAbortException_c — data abort exception
- gMaxException_c — values equal or greater than this one are not allowed

4.4.4 IntHandlerFunc_t

Constant Structure

```
typedef void (*IntHandlerFunc_t)(void);
```

Description

Pointer to a user provided function designed to treat one of the ARM exceptions or peripheral interrupts.

4.5 Exported Macros

4.5.1 gNoneInt_c

Structure

```
#define gNoneInt_c    gMaxInt_c
```

Description

Value used as parameter when calling ItcSetIrqMinimumLevel () function, to allow all interrupts. For details about gMaxInt_c, please refer to ItcNumber_t enumeration.

Constant Values

`gMaxInt_c`

4.5.2 ITC_TestSet ()

Structure

```
#define ITC_TestSet(Number)          ITC.IntFrc |= (1 << (Number))
```

Description

This macro triggers the interrupt associated with a specified peripheral or interrupt source. This is done by setting the corresponding bit in INTFRC register.

4.5.2.1 Macro Arguments

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.5.3 ITC_TestReset ()

Structure

```
#define ITC_TestReset(Number)        ITC.IntFrc &= ~(1 << (Number))
```

Description

This macro clears the test flag associated with a specified peripheral or interrupt source. This is done by clearing the corresponding bit in INTFRC register.

4.5.3.1 Macro Arguments

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.5.4 ITC_GetIntEnable ()

Structure

```
#define ITC_GetIntEnable(Number)     ((ITC.IntEnable >> (Number)) & 1)
```

Description

This macro reads the status of the enabling flag associated with the specified peripheral or interrupt source.

4.5.4.1 Macro Arguments

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.5.4.2 Returns

Description

Status of the specified bit in INTENABLE register.

Possible Values

1, 0

4.5.5 ITC_GetIntSrc ()

Structure

```
#define ITC_GetIntSrc(Number)        ((ITC.IntSrc >> (Number)) & 1)
```

Description

This macro reads the status of the flag associated with the specified peripheral or interrupt source.

4.5.5.1 Macro Arguments

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.5.5.2 Returns

Description

Status of the specified bit in INTSRC register

Possible Values

1, 0

4.5.6 ITC_GetFastPending ()

Structure

```
#define ITC_GetFastPending(Number)    ((ITC.FiPend >> (Number)) & 1)
```

Description

This macro reads the status of the flag associated with the specified peripheral or interrupt source, but only for fast interrupts.

4.5.6.1 Macro Arguments

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.5.6.2 Returns

Description

status of the specified bit in FIPEND register

Possible Values

1, 0

4.5.7 ITC_GetNormalPending ()

Structure

```
#define ITC_GetNormalPending(Number) ((ITC.NiPend >> (Number)) & 1)
```

Description

The macro reads the status of the flag associated with the specified peripheral or interrupt source, but only for normal interrupts.

4.5.7.1 Function Parameters

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.5.7.2 Returns

Description

Status of the specified bit in NIPEND register.

Possible Values

1, 0

4.5.8 IntEnableFIQ ()

Structure

```
#define IntEnableFIQ()          IntRestoreFIQ(0)
```

Description

The macro enables fast interrupts by clearing F bit in CPSR register. This macro is located in “Interrupt.h” header file.

4.5.9 IntEnableIRQ ()

Structure

```
#define IntEnableIRQ()          IntRestoreIRQ(0)
```

Description

The macro is used to enable normal interrupts by clearing I bit in CPSR register. This macro is located in “Interrupt.h” header file.

4.5.10 IntEnableAll ()

Structure

```
#define IntEnableAll()          IntRestoreAll(0)
```

Description

This macro enables interrupts by clearing I and F bits in CPSR register. This macro is located in "Interrupt.h" header file.

4.5.11 IntRemoveHandler ()

Structure

```
#define IntRemoveHandler(x)  IntAssignHandler((x), NULL)
```

Description

This macro removes an IRQ handler.

4.6 ITC Driver API Function Descriptions

4.6.1 ITC_Init ()

Prototype

```
void ITC_Init (void);
```

Description

The function is called to initialize the ITC registers, remove any handler from internal table. It can be called at anytime to reset the driver and ITC hardware.

After execution of this function, all peripheral interrupts will be disabled, FIQ and IRQ interrupts will be disabled; all interrupts will be set as normal type, none of them will be fast and test register INTFRC will be cleared.

4.6.2 IntAssignHandler ()

Prototype

```
ItcErr_t IntAssignHandler
(
    ItcNumber_t Number,
    IntHandlerFunc_t pfIrqHandler
);
```

Description

The function is called to assign a handler to a specified peripheral or interrupt source. There are some tests performed before modifying internal variables of the driver. If the *Number* is equal with *gMaxInt_c* or equal or greater than *gMaxException_c* the function will exit with *gItcErr_InvalidNumber_c* value.

If the *pfIrqHandler* is equal to *NULL*, the function will set its internal variable to point to a ‘dummy’ function, removing the handler that eventually was installed and returns *gItcErr_OK_c*. If all of these tests have been passed, the function copies the pfIrqHandler into handler table and exits with *gItcErr_OK_c* value.

4.6.2.1 Function Parameters

Name

Number

In/Out

input

Description

The peripheral interrupt source. or ARM exception associated with the handler.

Name

pfIrqHandler

In/Out

input

Description

Pointer to a user provided function.

4.6.2.2 Returns

Type

ItcErr_t

Description

The status of the operation.

Possible Values

- gItcErr_OK_c
- gItcErr_InvalidNumber_c

4.6.3 ITC_SetPriority ()

Prototype

```
ItcErr_t ITC_SetPriority
(
    ItcNumber_t Number,
    ItcPriority_t Priority
);
```

Description

The function is called to set the priority for the interrupt source specified by parameter *Number*. There are some tests performed before modifying the value in register INTTYPE. If the *Number* is equal or greater than *gMaxInt_c* the function will exit with *gItcErr_InvalidNumber_c* value.

If the *Priority* is different than *gItcFastPriority_c* or *gItcNormalPriority_c*, the function will exit with *gItcErr_InvalidPriority_c* value. If all of these tests have been passed, the function sets or clears the corresponding bit in INTTYPE register and exits with *gItcErr_OK_c* value.

4.6.3.1 Function Parameters

Name

Number

In/Out

input

Description

The peripheral interrupt source.

Name

Priority

In/Out

input

Description

The priority (normal or fast).

4.6.3.2 Returns

Type

ItcErr_t

Description

The status of the operation.

Possible Values

- gItcErr_OK_c
- gItcErr_InvalidNumber_c
- gItcErr_InvalidPriority_c

4.6.4 IntGetHandler()

Prototype

```
IntHandlerFunc_t IntGetHandler
(
    ItcNumber_t Number
);
```

Description

This function returns the handler for an IRQ number. If there is none assigned, it will return NULL.

4.6.4.1 Function Parameters

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.6.4.2 Returns

Type

IntHandlerFunc_t

Description

The pointer to the handler function or NULL if none assigned.

4.6.5 ITC_EnableInterrupt ()

Prototype

```
ItcErr_t ITC_EnableInterrupt
(
    ItcNumber_t Number
);
```

Description

The function is called to enable interrupts associated with a specified peripheral or interrupt source.

There is one test performed before writing into register INTENNUM. If the *Number* is equal or greater than *gMaxInt_c* the function will exit with *gItcErr_InvalidNumber_c* value.

If this test has been passed, the function writes into INTENNUM register the value specified by the parameter *Number*, and exits with *gItcErr_OK_c* value.

4.6.5.1 Function Parameters

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.6.5.2 Returns

Type

ItcErr_t

Description

The status of the operation.

Possible Values

- gItcErr_OK_c
- gItcErr_InvalidNumber_c

4.6.6 ITC_DisableInterrupt ()

Prototype

```
ItcErr_t ITC_DisableInterrupt
(
    ItcNumber_t Number
);
```

Description

The function is called to disable interrupts associated with a specified peripheral or interrupt source.

There is one test performed before writing into register INTDISNUM. If the *Number* is equal or greater than *gMaxInt_c* the function will exit with *gItcErr_InvalidNumber_c* value.

If this test has been passed, the function writes into INTDISNUM register the value specified by the parameter *Number*, and exits with *gItcErr_OK_c* value.

4.6.6.1 Function Parameters

Name

Number

In/Out

input

Description

The peripheral interrupt source.

4.6.6.2 Returns

Type

ItcErr_t

Description

The status of the operation.

Possible Values

- gItcErr_OK_c
- gItcErr_InvalidNumber_c

4.6.7 ITC_SetIrqMinimumLevel ()

Prototype

```
ItcErr_t ITC_SetNormalLevel
(
    ItcNumber_t Number
);
```

Description

The function is called to disable interrupts associated with a specified peripheral or interrupt source. There is one test performed before writing into register NIMASK. If the *Number* is greater than *gNoneInt_c* the function will exit with *gItcErr_InvalidNumber_c* value.

If this test has been passed, the function sets the corresponding bit in INTFRC register and exits with *gItcErr_OK_c* value.

4.6.7.1 Function Parameters

Name

Number

In/Out

input

Description

The minimum interrupt number needed to trigger the interrupt.

4.6.7.2 Returns

Type

ItcErr_t

Description

The status of the operation.

Possible Values

- gItcErr_OK_c
- gItcErr_InvalidNumber_c

4.6.8 InterruptInit ()

Prototype

```
void InterruptInit (void);
```

Description

The function is called to initialize the interrupt system. This does all the stuff in ItcInit, and further more, it resets the exception handlers to their default state (NULL). The function prototype is located in “Interrupt.h” header file.

This function should be called prior to any use of interrupts.

4.6.9 IntDisableFIQ ()

Prototype

```
unsigned int IntDisableFIQ (void);
```

Description

The function is called to disable fast interrupts by setting F bit in CPSR register. It returns the old value of F bit: ‘1’ if fast interrupts were disabled or ‘0’ otherwise. The function prototype is located in “Interrupt.h” header file.

4.6.9.1 Returns

Type

unsigned int

Description

The old value of the F bit.

Possible Values

1, 0

4.6.10 IntDisableIRQ ()

Prototype

```
unsigned int IntDisableIRQ (void);
```

Description

The function is called to disable normal interrupts by setting I bit in CPSR register. It returns the old value of I bit: ‘1’ if normal interrupts were disabled or ‘0’ otherwise. The function prototype is located in “Interrupt.h” header file.

4.6.10.1 Returns

Type

unsigned int

Description

The old value of the F bit.

Possible Values

1, 0

4.6.11 IntDisableAll ()

Prototype

```
unsigned int IntDisableAll (void);
```

Description

The function is called to disable interrupts by setting I and F bits in CPSR register. It returns the old values of both bits:

- 0: I = 0, F = 0 (both interrupts were enabled)
- 1: I = 0, F = 1 (fast interrupt was disabled and normal interrupt was enabled)
- 2: I = 1, F = 0 (fast interrupt was enabled and normal interrupt was disabled)
- 3: I = 1, F = 1 (both interrupts were disabled)

4.6.11.1 Returns

Type

unsigned int

Description

The old value of both the F and I bits.

Possible Values

0, 1, 2, 3

4.6.12 IntRestoreFIQ ()

Prototype

```
void IntRestoreFIQ (unsigned int f_bit)
```

Description

The function is called to restore fast interrupts by setting or clearing F bit in CPSR register, depending on passed parameter.

4.6.12.1 Function Parameters

Name

f_bit

In/Out

input

Description

Value for F bit, this value can be provided by function *IntDisableFIQ()*.

Possible Values

1, 0

4.6.13 IntRestoreIRQ ()

Prototype

```
void IntRestoreIRQ (unsigned int i_bit);
```

Description

The function is called to restore fast interrupts by setting or clearing I bit in CPSR register, depending on passed parameter.

4.6.13.1 Function Parameters

Name

i_bit

In/Out

input

Description

Value for F bit, this value can be provided by function *IntDisableIRQ()*.

Possible Values

1, 0

4.6.14 IntRestoreAll ()

Prototype

```
void IntRestoreAll (unsigned int if_bits);
```

Description

The function is called to restore interrupts by setting or clearing any of I and F bits in CPSR register. It accepts only these values as a parameter:

- 0: both interrupts to be enabled, I = 0 and F = 0
- 1: fast disabled and normal enabled, I = 0, F = 1
- 2: fast enabled and normal disabled, I = 1, F = 0
- 3: both interrupts will be disabled, I = 1 and F = 1

4.6.14.1 Function Parameters

Name

if_bits

In/Out

input

Description

Value for both F and I bits, this value can be provided by function *IntDisableAll()*.

Possible Values

0, 1, 2, 3

Chapter 5
Non-Volatile Memory (NVM) Driver

5.1 Overview

The MC1322x also contains a 128 Kbyte serial FLASH memory part of which is accessed via an internal dedicated SPI module (SPIF). The NVM memory driver is used to access the FLASH.

5.1.1 NVM Hardware Overview

The MC1322x non-volatile memory (NVM) is a 128 Kbyte serial FLASH. The serial FLASH is accessed via a dedicated SPI module designated as the SPIF. The FLASH erase, program, and read capability are all programmed through the SPIF port. The valid FLASH contents are accessed at boot time to load/initialize RAM for program execution. The FLASH is also accessed to re-initialize/restore RAM that had been un-powered during sleep. For detailed information on the CRM module, see the *MC1322x Reference Manual* (MC1322xRM).

With 96 kbytes of RAM and 128 kbytes of FLASH, there is 32 kbytes of excess capacity for other useful purposes than the main program code. The top 4 kbyte sector is reserved for factory use. There is sufficient capacity for non-volatile data store and optional application profiles if desired.

Some of the characteristics of the serial FLASH include:

- SPI serial interface with 13 MHz maximum data clock
- Accessed via a dedicated SPI module designated as the SPIF.
- Low power mode
- ID accessible under program control

5.1.2 NVM Driver Overview

The driver provides the following functions for the developer:

- Read from NVM
- Erase the NVM by sectors
- Write in NVM
- Verify data written in NVM
- Check NVM location for blank

5.2 Include Files

Table 5-1 shows the NVM driver file that must be included in the application C-files in order to have access to the driver function calls.

Table 5-1. NVM Driver Include File

Include File Name	Description
NVM.h	Global header file that defines the public data types and enumerates the public functions prototypes

5.3 NVM Driver API Functions

Table 5-2 shows the available API functions for the NVM Driver.

Table 5-2. NVM Driver API Function List

NVM Driver API Function Name	Description
NVM_Detect ()	This function executes a search for the known NVM types.
NVM_Read ()	This function reads bytes of data from the flash.
NVM_Erase ()	This function erases a set of sectors.
NVM_Write ()	This function writes bytes of data read from the memory to the address in flash.
NVM_Verify ()	This function verifies if bytes of data from the address in flash are equal with the data referred in RAM.
NVM_BlankCheck ()	This function verifies bytes of data from the address in NVM to be blank.

5.4 Driver Exported Constants

5.4.1 nvmType_t

Constant Structure

```
typedef enum
{
    gNvmType_NoNvm_c,
    gNvmType_SST_c,
    gNvmType_ST_c,
    gNvmType_ATM_c,
    gNvmType_Max_c
} nvmType_t;
```

Description

Specifies all possible NVM Types that this driver can manage.

5.4.2 nvmErr_t

Constant Structure

```
typedef enum
{
    gNvmErrNoError_c = 0,
    gNvmErrInvalidInterface_c,
    gNvmErrInvalidNvmType_c,
    gNvmErrInvalidPointer_c,
    gNvmErrWriteProtect_c,
    gNvmErrVerifyError_c,
    gNvmErrAddressSpaceOverflow_c,
    gNvmErrBlankCheckError_c,
    gNvmErrRestrictedArea_c,
    gNvmErrNvmRegOff_c,
    gNvmErrMaxError_c
} nvmErr_t;
```

Description

Specifies all possible error values returned by this driver functions.

5.4.3 nvmInterface_t

Constant Structure

```
typedef enum
{
    gNvmInternalInterface_c,
    gNvmExternalInterface_c,
    gNvmInterfaceMax_c
} nvmInterface_t;
```

Description

Specifies the two possible SPI interfaces that a NVM can be connected on.

- gNvmInternalInterface_c: internal SPI (SPIF).
- gNvmExternalInterface _c: external SPI (multiplexed with GPIO).

5.5 NVM Driver API Function Descriptions

5.5.1 NVM_Detect ()

Prototype

```
nvmErr_t NVM_Detect
(
    nvmInterface_t nvmInterface,
    nvmType_t* pNvmType
```

```
);
```

Description

This function executes a search for known flash types on the indicated SPI .

Usually this function is called first to determine the NVM type used as parameter for the rest of NVM driver functions.

CAUTION

If external SPI is used the user should configure the SPI pins as function 1 from GPIO (they are in GPIO mode default). If internal SPI is used the 1.8V regulator should be turn on first.

5.5.1.1 Function Parameters

Name

nvmInterface

In/Out

input

Description

This parameter specifies which SPI interface is used to drive the NVM.

Name

pNvmType

In/Out

output

Description

This parameter is used to return the NVM type found.

5.5.1.2 Returns

Type

`nvmErr_t`

Description

Function execution status.

Possible Values

- gNvmErrInvalidInterface_c — If nvmInterface is not one of the two valid values(gNvmInternalInterface_c or gNvmExternalInterface_c).
- gNvmErrNvmRegOff_c - If NVM buck regulator is off
- gNvmErrInvalidPointer_c — If pNvmType is NULL.
- gNvmErrNoError_c — In rest.

5.5.2 NVM_Read ()

Prototype

```
nvmErr_t NVM_Read
(
    nvmInterface_t nvmInterface,
    nvmType_t nvmType,
    void *pDest,
    uint32_t address,
    uint32_t numBytes
);
```

Description

This function reads *numBytes* bytes from *address* address in NVM and stores them at *pDest* address in RAM.

5.5.2.1 Function Parameters

Name

nvmInterface

In/Out

input

Description

This parameter specifies which SPI interface is used to drive the NVM.

Name

nvmType

In/Out

input

Description

This parameter specifies the NVM type connected on NvmInterface.

Name

pDest

In/Out

input

Description

This parameter specifies the destination address for read data.

Name

address

In/Out

input

Description

This parameter specifies the NVM address to read from.

Name

numBytes

In/Out

input

Description

This parameter specifies the number of bytes to read.

5.5.2.2 Returns

- gNvmErrInvalidInterface_c if nvmInterface is not one of the two valid values(gNvmInternalInterface_c or gNvmExternalInterface_c).
- gNvmErrInvalidNvmType_c if nvmType is not one of the three known values (gNvmType_SST_c , gNvmType_ST_c, gNvmType_ATM_c).
- gNvmErrAddressSpaceOverflow_c if (address +numBytes-1)> gNvmMaxAddress_c.
- gNvmErrInvalidPointer_c if pDest == NULL.
- gNvmErrRestrictedArea_c if nvmInterface = gNvmInternalInterface_c and the address space reaches last sector.

- gNvmErrNoError_c in rest.

CAUTION

If external SPI is used the user should configure the SPI pins as function 1 from GPIO (they are in GPIO mode default). If internal SPI is used the 1.8V regulator should be turn on first.

5.5.3 NVM_Erase ()

Prototype

```
nvmErr_t NVM_Erase (
    nvmInterface_t nvmInterface,
    nvmType_t nvmType,
    uint32_t sectorBitfield
);
```

Description

This function erases the sectors specified by *sectorBitfield*.

CAUTION

If external SPI is used the user should configure the SPI pins as function 1 from GPIO (they are in GPIO mode default). If internal SPI is used the 1.8V regulator should be turn on first.

5.5.3.1 Function Parameters

Name

nvmInterface

In/Out

input

Description

This parameter specifies which SPI interface is used to drive the NVM.

Name

nvmType

In/Out

input

Description

This parameter specifies the NVM type connected on nvmInterface.

Name

sectorBitfield

In/Out

input

Description

Sector mapping bitfield. The mean of the bits is depending of the memory type that is used, as follows:

- For non-volatile memories manufactured by SST (nvmType = gNvmType_SST_c NVM):
  - sector size = 4096 bytes
  - number of sectors = 32
  - each bit of the sectorBitfield corresponds to a sector, i.e. bit 0 represents sector 0 and bit 31 represents sector 31
- For non-volatile memories manufactured by ST or ATMEL (nvmType = gNvmType_ST_c or nvmType = gNvmType_ATM_c):
  - sector size = 32768 bytes
  - number of sectors = 4
  - only the least 4 significant bits represents a sector, i.e. bit 0 represents sector 0 and bit 3 represents sector 3

5.5.3.2 Returns

Type

nvmErr_t

Description

Function execution status.

Possible Values

- gNvmErrInvalidInterface_c — If nvmInterface is not one of the two valid values(gNvmInternalInterface_c or gNvmExternalInterface_c).
- gNvmErrInvalidNvmType_c — If nvmType is not one of the three known values(gNvmType_SST_c,gNvmType_ST_c,gNvmType_ATM_c).
- gNvmErrRestrictedArea_c — If the last sector is set to be erased.
- gNvmErrNoError_c — In rest.

5.5.4 NVM_Write ()

Prototype

```
nvmErr_t NVM_Write
(
    nvmInterface_t nvmInterface,
    nvmType_t nvmType,
    void *pSrc,
    uint32_t address,
    uint32_t numBytes
);
```

Description

This function writes *numBytes* bytes from *pSrc* address in RAM to *address* address in NVM. The user must ensure that the sectors to be written were previously erased using the NVM_Erase function. No error is returned if the user tries to write an un-erased sector.

CAUTION

If external SPI is used the user should configure the SPI pins as function 1 from GPIO (they are in GPIO mode default). If internal SPI is used the 1.8V regulator should be turn on first.

5.5.4.1 Function Parameters

Name

nvmInterface

In/Out

input

Description

This parameter specifies which SPI interface is used to drive the NVM.

Name

nvmType

In/Out

input

Description

This parameter specifies the NVM type connected on NvmInterface.

Name

pSrc

In/Out

input

Description

This parameter specifies the source address of the data to be written in NVM.

Name

address

In/Out

input

Description

This parameter specifies the NVM address to write at.

Name

numBytes

In/Out

input

Description

This parameter specifies the number of bytes to write.

5.5.4.2 Returns

Type

nvmErr_t

Description

Function execution status.

Possible Values

- gNvmErrInvalidInterface_c if nvmInterface is not one of the two valid values(gNvmInternalInterface_c or gNvmExternalInterface_c).

- gNvmErrInvalidNvmType_c if nvmType is not one of the three known values(gNvmType_SST_c,gNvmType_ST_c,gNvmType_ATM_c).
- gNvmErrAddressSpaceOverflow_c if (address +numBytes-1)> gNvmMaxAddress_c.
- gNvmErrRestrictedArea_c if nvmInterface = gNvmInternalInterface_c and the address space reaches last sector.
- gNvmErrWriteProtect_c if the NVM cannot be enabled to be write.
- gNvmErrVerifyError_c if after the write operation at least one location in NVM is different from its intended value.
- gNvmErrNoError_c in rest.

5.5.5 NVM_Verify ()

Prototype

```
nvmErr_t NVM_Verify
(
    nvmInterface_t nvmInterface,
    nvmType_t nvmType,
    void *pSrc,
    uint32_t address,
    uint32_t numBytes
);
```

Description

This function compares *numBytes* bytes from *pSrc* address in RAM with *numBytes* bytes from *address* address in NVM. If function returns gNvmErrNoError_c the data sets are identical.

CAUTION

If external SPI is used the user should configure the SPI pins as function 1 from GPIO (they are in GPIO mode default). If internal SPI is used the 1.8V regulator should be turn on first.

5.5.5.1 Function Parameters

Name

nvmInterface

In/Out

input

Description

This parameter specifies which SPI interface is used to drive the NVM.

Name

nvmType

In/Out

input

Description

This parameter specifies the NVM type connected on NvmInterface.

Name

pSrc

In/Out

input

Description

This parameter specifies the source address of the RAM data to be compared with the data from NVM.

Name

address

In/Out

input

Description

This parameter specifies the address of NVM data to be compared with the data from RAM.

Name

numBytes

In/Out

input

Description

This parameter specifies the number of bytes to compare.

5.5.5.2 Returns

Type

nvmErr_t

Description

Function execution status.

Possible Values

- gNvmErrInvalidInterface_c — If nvmInterface is not one of the two valid values(gNvmInternalInterface_c or gNvmExternalInterface_c).

- gNvmErrInvalidNvmType_c — If nvmType is not one of the three known values(gNvmType_SST_c,gNvmType_ST_c,gNvmType_ATM_c).
- gNvmErrAddressSpaceOverflow_c — If (address +numBytes-1)> gNvmMaxAddress_c.
- gNvmErrInvalidPointer_c if pSrc is NULL.
- gNvmErrVerifyError_c — If at least one location in NVM is different from its corresponding location in RAM.
- gNvmErrNoError_c — In rest.

5.5.6 NVM_BlankCheck ()

Prototype

```
nvmErr_t NVM_BlankCheck
(
    nvmInterface_t nvmInterface,
    nvmType_t nvmType,
    uint32_t address,
    uint32_t numBytes
);
```

Description

This function checks if *numBytes* bytes from *address* address in NVM are erased (0xff).

CAUTION

if external SPI is used the user should configure the SPI pins as function 1 from GPIO (they are in GPIO mode default). If internal SPI is used the 1.8V regulator should be turn on first.

5.5.6.1 Function Parameters

Name

nvmInterface

In/Out

input

Description

This parameter specifies which SPI interface is used to drive the NVM.

Name

nvmType

In/Out

input

Description

This parameter specifies the NVM type connected on NvmInterface.

Name

address

In/Out

input

Description

This parameter specifies the address of NVM data to be checked.

Name

numBytes

In/Out

input

Description

This parameter specifies the number of bytes to be checked.

5.5.6.2 Returns

Type

nvmErr_t

Description

Function execution status.

Possible Values

- gNvmErrInvalidInterface_c — If nvmInterface is not one of the two valid values(gNvmInternalInterface_c or gNvmExternalInterface_c).
- gNvmErrInvalidNvmType_c — If nvmType is not one of the three known values(gNvmType_SST_c,gNvmType_ST_c,gNvmType_ATM_c).
- gNvmErrAddressSpaceOverflow_c — If (address +numBytes-1)> gNvmMaxAddress_c.
- gNvmErrInvalidPointer_c — If pSrc is NULL.

- gNvmErrBlankCheckError_c — If at least one addressed location in NVM is not blank(0xff).
- gNvmErrNoError_c — In rest.

Chapter 6
UART Driver

6.1 Overview

The MC1322x supplies two Universal Asynchronous Receiver Transmitter (UART) modules; both of which are accessed with the UART driver. These are two independent modules designated UART1 and UART2 and can be used to connect to traditional RS232 serial ports or to communicate with other embedded controllers.

6.1.1 UART Hardware Module

Each UART has an independent fractional divider, baud rate generator that is clocked by the peripheral bus clock (generated from the reference oscillator divided by the CRM prescaler; typically 24 MHz max) which enables a broad range of baud rates up to 1,843.2 Kbaud. Transmit and receive use a common baud rate for each module. For detailed information on the CRM module, see the *MC1322x Reference Manual* (MC1322xRM).

The UART has many advanced features useful for reliable, high-throughput serial communication. The UART module features include:

- 8-bit only data
- Programmable one or two stop bits
- Programmable parity (even, odd, and none)
- Full duplex four-wire serial interface (RXD, TXD, RTS, and CTS)
- Programmable hardware flow control for RTS and CTS signals with programmable sense (high true/low true)
- Independent 32-byte receive FIFO and 32-byte transmit FIFO with level indicators
- Receiver detects framing errors, start bit error, break characters, parity errors, and overrun errors.
- Maskable interrupt request
- Time-out interrupt timer, which times out after eight non-present characters
- Baud rate generator to provide any multiple-of-2 baud rate between 1.2 Kbaud and 1,843.2 Kbaud

6.1.2 UART Driver Functionality

The driver allows the developer to use the UART functions easily. It allows the developer to:

- Open an UART peripheral instance
- Configure an UART peripheral instance
- Set the callback functions for an UART peripheral instance

- Set the receiver threshold for an UART peripheral instance
- Set the transmitter threshold for an UART peripheral instance
- Set the CTS threshold for an UART peripheral instance
- Launch the automatic baud rate detection for an UART peripheral instance
- Read data from an UART peripheral instance RxFIFO
- Cancel read data from an UART peripheral instance RxFIFO
- Write data in an UART peripheral instance TxFIFO
- Cancel write data in an UART peripheral instance TxFIFO
- Get the configuration of an UART peripheral instance
- Get the status of an UART peripheral instance
- Close an UART peripheral instance

6.2 Include Files

Table 6-1 shows the file that must be included in the application C-files in order to have access to the UART driver function calls.

Table 6-1. UART Driver Include File

Include File Name	Description
UartLowLevel.h	Header file that defines the public data types and enumerates the public function prototypes

6.3 UART Driver API Functions

Table 6-2 shows the available API functions for the UART Driver.

Table 6-2. UART Driver API Function List

UART Driver API Function Name	Description
UartOpen()	The function is called to initialize a UART module.
UartSetConfig()	The function is called to set the communication parameters (parity, stop bits, baud rate, flow control) for the named UART.
UartSetReceiverThreshold()	The function sets the threshold for the number of received characters (bytes) required in the Rx FIFO to trigger an interrupt from UART during an Rx
UartSetTransmitterThreshold()	The function sets the threshold for which an interrupt from the UART will be triggered during a Tx, if the number of available bytes in the Tx FIFO falls below this number.
UartSetCTSThreshold()	If flow control is enabled, if the number of characters in the Rx FIFO exceeds a second threshold, called CTS threshold, the UART will signal the host to stop transmission by asserting the RTS signal. The *UartSetCTSThreshold()* function sets the CTS threshold for the named UART.

Table 6-2. UART Driver API Function List (continued)

UartSetCallbackFunctions()	The function is called to set the callback functions for the read and write operations for the named UART
UartReadData()	The function is called to start a read data operation of the named UART.
UartGetByteFromRxBuffer()	The function is called to fetch one character from the UART receive FIFO
UartOpenCloseTransceiver()	The function is called is open or close the transceiver.
UartClearErrors()	The function is called to clear error status.
UartCancelReadData()	The function is called to cancel a read operation on the named UART.
UartWriteData()	The function is called to start a write data operation to the named UART.
UartCancelWriteData()	The function is called to cancel a write operation on the named UART.
UartGetStatus()	The function is called to get the status of the named UART.
UartGetConfig()	The function is called to get the communication parameters (parity, stop bits, baud rate, flow control) of the named UART.
UartGetUnreadBytesNumber()	The function is called to get the number of unread bytes still in UART.
UartClose()	The function is called to close the named UART.
UartSetHalfFlowControl()	This function sets the half flow control mode (the CTS line is driven by software).
UartIsr1()	The function will be set as interrupt routine for the first UART.
UartIsr2()	The function will be set as interrupt routine for the second UART

6.4 Driver Exported Constants

6.4.1 UartErr_t

Constant Structure

```
typedef enum
(
    gUartErrNoError_c = 0,
    gUartErrUartAlreadyOpen_c,
    gUartErrUartNotOpen_c,
    gUartErrNoCallbackDefined_c,
    gUartErrReadOngoing_c,
    gUartErrWriteOngoing_c,
    gUartErrInvalidClock_c,
    gUartErrNullPointer_c,
    gUartErrInvalidNrBytes_c,
    gUartErrInvalidBaudRate_c,
    gUartErrInvalidParity_c,
    gUartErrInvalidStop_c,
    gUartErrInvalidCTS_c,
    gUartErrInvalidThreshold_c,
    gUartErrWrongUartNumber_c,
    gUartErrErrMax_c
} UartErr_t;
```

Description

Specifies the possible error return values for the UART driver API functions.

Constant Values

The constant values are self explanatory.

6.4.2 UartReadStatus_t

Constant Structure

```
typedef enum
{
     gUartReadStatusComplete_c = 0,
     gUartReadStatusCanceled_c,
     gUartReadStatusError_c,
     gUartReadStatusMax_c
} UartReadStatus_t;
```

Description

Specifies the read status passed as a parameter to the read callback function.

Constant Values

- gUartReadStatusComplete_c - the read process is complete.
- gUartReadStatusCanceled_c - the read process was canceled.
- gUartReadStatusError_c - an error on the read process has occurred.

6.4.3 UartWriteStatus_t

Constant Structure

```
typedef enum
{
     gUartWriteStatusComplete_c = 0,
     gUartWriteStatusCanceled_c,
     gUartWriteStatusMax_c
} UartWriteStatus_t;
```

Description

Specifies the write status passed as a parameter to the write callback function.

Constant Values

- gUartWriteStatusComplete_c - the write process is complete.
- gUartWriteStatusCanceled_c - the write process was canceled.

6.4.4 UartParityMode_t

Constant Structure

```
typedef enum
{
     gUartParityNone_c = 0,
     gUartParityEven_c,
     gUartParityOdd_c,
     gUartParityMax_c
} UartParityMode_t;
```

Description

Specifies the possible values of the parity mode.

Constant Values

The constant values are self explanatory.

6.4.5 UartStopBits_t

Constant Structure

```
typedef enum
{
     gUartStopBits1_c = 0,
     gUartStopBits2_c,
     gUartStopBitsMax_c
} UartStopBits_t;
```

Description

Specifies the number of stop bits.

Constant Values

The constant values are self explanatory.

6.5 Exported Structures

6.5.1 UartConfig_t

Structure

```
typedef struct
{
    uint32_t         UartBaudrate;
    UartParityMode_t UartParity;
    UartStopBits_t   UartStopBits;
    bool_t           UartFlowControlEnabled;
    bool_t           UartRTSActiveHigh;
} UartConfig_t;
```

Description

Describes the configuration of the UART module: baudrate, parity bits, number of stop bits and flow control.

6.5.1.1 Structure Elements

- UartParityit - specifies the parity value of the character.
- UartStopBits - specifies the number of stop bits in the character.
- UartBaudrateit - specifies the value of the baud rate on transmission or reception.
- UartFlowControlEnabled - specifies if flow control mechanism is enabled
- UartRTSActiveHigh - specified the sense of the RTC/CTS flow control lines

6.5.2 UartReadErrorFlags_t

Structure

```
typedef struct
{
    uint32_t   UartReadOverrunError:1;
    uint32_t   UartParityError:1;
    uint32_t   UartFrameError:1;
    uint32_t   UartStartBitError:1;
    uint32_t   Reserved:4;
} UartReadErrorFlags_t;
```

Description

Lists the read error flags. Structure returned by the read callback function. On reset, state for structure's members will be set to FALSE.

6.5.2.1 Structure Elements

- UartReadOverrunError - read FIFO overrun error

- UartReadParityError - read parity error
- UartReadFrameError - read frame/stop bit error
- UartReadStartBitError - read start bit error

6.5.3 UartReadCallbackArgs_t

Structure

```
typedef struct
{
    UartReadStatus_t            UartStatus;
    uint16_t                    UartNumberBytesReceived;
    UartReadErrorFlags_t        UartReadError;
} UartReadCallbackArgs_t;
```

Description

Passed as a parameter to the read callback function.

6.5.3.1 Structure Elements

- UartStatus - the read status of the UART after the read operation is over.
- UartNumberBytes - number of bytes having been read
- UartReadError - read error flags corresponding to the read data.

6.5.4 UartWriteCallbackArgs_t

Structure

```
typedef struct
{
    UartWriteStatus_t       UartStatus;
    uint16_t                UartNumberBytesSent;
} UartWriteCallbackArgs_t;
```

Description

Passed as a parameter to the write callback function.

6.5.4.1 Structure Elements

- UartStatus - the write status of the UART after the write operation is over.
- UartNumberBytes - number of bytes being written.

6.5.5 UartCallbackFunctions_t

Structure

```
typedef struct {
    void (*pfUartReadCallback)(UartReadCallbackArgs_t* args);
    void (*pfUartWriteCallback)(UartWriteCallbackArgs_t* args);
} UartCallbackFunctions_t;
```

Description

Lists the callback functions for the UART driver.

6.5.5.1 Structure Elements

- pfUartReadCallback - a pointer to a function that will be set with the read callback function
- pfUartWriteCallback - a pointer to a function that will be set with the write callback function

6.6 Exported Macros

6.6.1 UART_NR_INSTANCES

It specifies the number of UART peripherals on the platform.

Function Parameters

None.

Returns

2

6.6.2 UART_1

It specifies the internal number of UART 1 instance.

Function Parameters

None

Returns

0

6.6.3 UART_2

It specifies the internal number of UART 2 instance.

Function Parameters

None

Returns

1

6.6.4 UartOpenReceiver(UartNumber)

Structure

```
#define UartOpenReceiver(UartNumber) UartOpenCloseTransceiver((UartNumber), BIT_RX_EN, TRUE)
```

Description

It opens the specified UART receiver.

6.6.4.1 Macro Arguments

Name

UartNumber

In/Out

input

Description

The UART whose receiver is desired to be opened.

6.6.5 UartCloseReceiver(UartNumber)

Structure

```
#define UartCloseReceiver(UartNumber) UartOpenCloseTransceiver((UartNumber), BIT_RX_EN, FALSE)
```

Description

It closes the specified UART receiver.

6.6.5.1 Macro Arguments

Name

UartNumber

In/Out

input

Description

The UART whose receiver is desired to be closed.

6.6.6 UartOpenTransmitter(UartNumber)

Structure

```
#define UartOpenTransmitter(UartNumber) UartOpenCloseTransceiver((UartNumber), BIT_TX_EN,
TRUE)
```

Description

It opens the specified UART transmitter.

6.6.6.1 Macro Arguments

Name

UartNumber

In/Out

input

Description

The UART whose transmitter is desired to be opened.

6.6.7 UartCloseTransmitter(UartNumber)

Structure

```
#define UartCloseTransmitter(UartNumber) UartOpenCloseTransceiver((UartNumber), BIT_TX_EN,
FALSE)
```

Description

It closes the specified UART transmitter.

6.6.7.1 Macro Arguments

Name

UartNumber

In/Out

input

Description

The UART whose transmitter is desired to be closed.

6.7 UART Driver API Functions

The function is called to initialize the RAM memory space for the UART driver (patch function table and global variable space). Actually this is not a interface function. It will be called only once, for all the UART driver instances. If this call is skipped, unpredictable behavior can appear when trying to access UART driver functions.

CAUTION

This function must be called prior to any other call to the UART driver interface functions.

6.7.1 UartOpen ()

Prototype

```
UartErr_t UartOpen
(
    uint8_t UartNumber,
    uint32_t PlatformClock
);
```

Description

The function is called to initialize the UART peripheral.

There are some tests performed before initializing the UART peripheral. If the UART instance number to open exceeds the UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. If *PlatformClock* parameter is equal to zero or greater then *PLATFORM_MAX_CLOCK*, the function exits with *gUartErrInvalidClock_c* value. If UART module is already initialized, the function exits with *gUartErrUartAlreadyOpen_c* value.

If all of these tests have been passed, the function initializes the internal driver variables, enables the hardware transmitter and receiver modules and exits with *gUartErrNoError_c* value.

CAUTION

This function must be called prior to any call to other functions, and it should be called only once. If it is called more than once, a *gUartErrUartAlreadyOpen_c* will be returned.

6.7.1.1 Function Parameters

Name

UartNumber

In/Out

input

Description

Number of the UART instance to open.

Name

PlatformClock

In/Out

input

Description

Platform clock (in Khz). Will be used for baudrate calculation.

6.7.1.2 Returns

Type

UartErr_t

Description

The status of the operation.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrInvalidClock_c
- gUartErrUartAlreadyOpen_c

6.7.2 UartSetConfig ()

Prototype

```
UartErr_t UartSetConfig
(
    uint8_t UartNumber,
    UartConfig_t* pConfig
);
```

Description

The function is called to set the communication parameters (parity, stop bits, baud rate, flow control) for the UART peripheral.

There are some tests performed before configuring the UART peripheral. If the UART peripheral instance number exceeds the UART_NR_INSTANCES, the function exits with *gUartErrWrongUartNumber_c*

value. If the pointer to the structure containing the configuration parameters is not initialized, the function exits with *gUartErrNullPointer_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value. If the value of the baud rate in the configuration structure is lower than 1200 bps or if it exceeds the maximum baud rate obtainable considering the frequency of the platform clock, the function exits with *gUartErrInvalidBaudrate_c* value. If the value of the parity is out of the permitted range, the function exits with *gUartErrInvalidParity_c* value. If the value of the stop bits is out of the permitted range, the function exits with *gUartErrInvalidStop_c* value. If a read operation is in place, the function exits with *gUartErrReadOngoing_c* value. If a write operation is currently running, the function exits with *gUartErrWriteOngoing_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gUartErrNoError_c* value .

CAUTION

The value of the baud rate at which the UART peripheral is communicating after a UartSetConfig call is set using an algorithm which makes calculations considering both the baud rate specified as parameter and the current platform clock. The algorithm is very precise for all the baud rates and platform clocks specified in the UART MC1322x Reference Design Manual. For values of the platform clock other than the ones in Reference Design Manual, the algorithm will set the specified baud rate very precisely only for values of platform clock (in KHz) which are integer multiple of 40. In the other cases, the baud rate will be a little bit different from the one received as parameter. The baud rate set by a UartSetConfig call can be checked by calling the UartGetConfig function.

If the UartSetConfig function enables the flow control mechanism, a check for the CTS threshold is performed. Thus, if the CTS threshold is lower than the RX threshold, the first one will be automatically updated to the value of the second.

6.7.2.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to configure.

Name

pConfig

In/Out

input

Description

Pointer to a structure containing the configuration parameters.

6.7.2.2 Returns

Type

UartErr_t

Description

The status of the operation.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrNullPointer_c
- gUartErrUartNotOpen_c
- gUartErrReadOngoing_c
- gUartErrWriteOngoing_c
- gUartErrInvalidBaudrate_c
- gUartErrInvalidParity_c
- gUartErrInvalidStop_c

6.7.3 UartSetCallbackFunctions ()

Prototype

```
UartErr_t UartSetCallbackFunctions
(
    uint8_t UartNumber,
    UartCallbackFunctions_t* pCallback
);
```

Description

The function is called to set the callback functions for the read and write operations for the UART peripheral.

There are some tests performed before setting the callback functions. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. If the pointer to the structure containing the

callback pointers is not initialized, the function exits with *gUartErrNullPointer_c* value. The function will also exit with this error if both the pointers to the callback functions are null. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value.

If all of these tests have been passed, the function is setting the internal driver callback pointers with the ones received in the structure and returns the *gUartErrNoError_c* value.

CAUTION

The function sets the callback functions only if the pointers received are not null. If both the pointers to the read and write callbacks are null, the functions exits with *gUartErrNullPointer_c value* and does not perform any action. In case that only one pointer to the callback functions is null, that callback function will not be set and after setting the other callback, the function will exit with *gUartErrNoError_c* value.

6.7.3.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to set the callback functions.

Name

pCallback

In/Out

input

Description

Pointer to a structure containing the pointers to the callback functions.

6.7.3.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrNullPointer_c
- gUartErrUartNotOpen_c

6.7.4 UartSetReceiverThreshold ()

Prototype

```
UartErr_t UartSetReceiverThreshold
(
    uint8_t UartNumber,
    uint8_t Threshold
);
```

Description

The UART peripheral receiver module has an internal 32 bytes FIFO buffer. During the receive operation, the characters received are placed in this buffer. An internal Rx interrupt is triggered when the number of characters in the FIFO exceeds a specified threshold. The *UartSetReceiverThreshold()* function sets the Rx threshold for a UART peripheral.

There are some tests performed before setting the threshold. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. The value of the threshold should be in the interval [2, 32]. If the value is out of this interval, the function exits with *gUartErrInvalidThreshold_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value. If a read operation is currently ongoing, the function exits with *gUartErrReadOngoing_c* value.

If all of these tests have been passed, the function sets the Rx threshold and exits with *gUartErrNoError_c* value.

CAUTION

If flow control is enabled and the new value of the Rx threshold is higher than the value of the CTS threshold, the last one will be automatically updated to the value of the first.

6.7.4.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to set the Rx FIFO threshold.

Name

Threshold

In/Out

input

Description

The value of the threshold.

6.7.4.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrInvalidThreshold_c
- gUartErrUartNotOpen_c
- gUartErrReadOngoing_c

6.7.5 UartSetTransmitterThreshold ()

Prototype

```
UartErr_t UartSetTransmitterThreshold
(
    uint8_t UartNumber,
    uint8_t Threshold
);
```

Description

The UART peripheral transmitter module has an internal 32 bytes FIFO buffer. During the transmit operation, the characters from FIFO are sent by shifting on the Tx pin. An internal Tx interrupt is triggered when the number of characters in the FIFO goes under a specified threshold. The *UartSetTransmitterThreshold()* function sets the threshold for the UART peripheral.

There are some tests performed before setting the threshold. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. The value of the threshold should be in the interval [1, 32]. If the value is out of this interval, the function exits with *gUartErrInvalidThreshold_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value. If a write operation is currently ongoing, the function exits with *gUartErrReadOngoing_c* value.

If all of these tests have been passed, the function sets the TX threshold and exits with *gUartErrNoError_c* value.

6.7.5.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to set the Tx FIFO threshold.

Name

Threshold

In/Out

input

Description

The value of the threshold.

6.7.5.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrInvalidThreshold_c
- gUartErrUartNotOpen_c
- gUartErrReadOngoing_c

6.7.6 UartSetCTSThreshold ()

Prototype

```
UartErr_t UartSetCTSThreshold
(
    uint8_t UartNumber,
    uint8_t Threshold
);
```

Description

The UART peripheral receiver module has an internal 32 bytes FIFO buffer. During the receive operation, the characters received are placed in this buffer. An internal Rx interrupt is triggered when the number of characters in the FIFO exceeds a specified threshold. If flow control is enabled, if the number of characters exceeds a second threshold, called CTS threshold, the UART will signal the host to stop transmission by asserting the RTS signal. The *UartSetCTSThreshold()* function sets the CTS threshold for a UART peripheral.

There are some tests performed before setting the threshold. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. The value of the threshold should be in the interval [2, 32] but always greater or equal to Rx threshold.

If the value received as parameter does not satisfy these conditions, the function exits with *gUartErrInvalidThreshold_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value. If a read operation is currently ongoing, the function exits with *gUartErrReadOngoing_c* value.

If all of these tests have been passed, the function sets the Rx threshold and exits with *gUartErrNoError_c* value.

6.7.6.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to set the CTS FIFO threshold.

Name

Threshold

In/Out

input

Description

The value of the threshold.

6.7.6.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrInvalidThreshold_c
- gUartErrUartNotOpen_c
- gUartErrReadOngoing_c

6.7.7 UartSetHalfFlowControl()

Prototype

```
void UartSetHalfFlowControl
(
    uint8_t uartNumber,
    bool_t bHalfFlowControl
);
```

Description

The function is called to configure the half flow control mode. If the half flow control mode is activated, the CTS line is driven by software, which means that the line is de asserted if the hardware FIFO of the UART is full (threshold was reached). The line is asserted when the number of bytes from FIFO is lower than the threshold value.

6.7.7.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to configure half flow control.

Name

bHalfFlowControl

In/Out

input

Description

The configuration option for half flow control mode.

6.7.7.2 Returns

Type

void

Description

None

6.7.8 UartReadData ()

Prototype

```
UartErr_t UartReadData
(
    uint8_t UartNumber,
```

```
        uint8_t* pBuf,
        uint16_t NumberBytes,
        bool_t UartDirectFifoMode
        );
```

Description

This function is called to initiate a read data operation on the Uart peripheral module.

The following tests are performed before the function starts the actual read:

- If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value.
- If the pointer to the destination buffer is not initialized, the function exits with the *gUartErrNullPointer_c* value.
- If the buffer length is 0, the function exits with *gUartErrInvalidNrBytes_c* value.
- If the UART peripheral instance was not previously opened, the function exists with *gUartErrUartNotOpen_c* value
- If a read operation is currently ongoing, the function exits with *gUartErrReadOngoing_c* value.
- If no callback was previously configured for the current UART instance, the function exits with *gUartErrNoCallbackDefined_c*.

If no error condition is detected, the function sets the internal driver variables, peripheral hardware registers and exists with gUartErrNoError_c

There are two functions covered by this function:

1. The function waits for the requested number of bytes to be received and exits.

This can be achieved by passing the *UartDirectFifoMode* parameter as 0 (FALSE).

After a UartReadData() function call, if no data is received on the UART, the driver remains in the ‘uart read ongoing’ state, until at least one character is received. The application can cancel at any time a read operation, by calling the UartCancelReadData() function. There is no timeout mechanism implemented in the UART driver.

The UART driver exits from a read operation in the following situations:

- The number of characters specified in the NumberBytes parameter has been received without errors. In this case, the read callback function is triggered with the parameters *UartStatus* set to *gUartReadStatusComplete_c* and *UartNbBytes* set to *NumberBytes*.
- After receiving at least one character, if no characters have been received for at least 8 character periods. In this case, the read callback function is triggered with the parameters *UartStatus* set to *gUartReadStatusComplete_c* and *UartNbBytes* set to number of characters received before the 8 non-present character period.
- During the receive procedure, if errors are detected. In this case, the read callback function is triggered with the parameters *UartStatus* set to *gUartReadStatusError_c* and *UartNbBytes* set to number of characters received before the error detection.

- During the receive procedure, if the *UartCancelReadData()* function is called. In this case, the read callback function is triggered with the parameters *UartStatus* set to *gUartReadStatusCanceled_c* and *UartNbBytes* set to number of characters received before the process cancellation.

The bytes received by the UartReadData() function are copied in the pBuf supplied by the user at the function call.

2. The function exposes directly the receive HW FIFO to the user.

This can be achieved by passing the UartDirectFifoMode as 1 (TRUE).

After a UartReadData() call, the driver remains in the ‘uart read ongoing’ state. The application can cancel at any time a read operation, by calling the *UartCancelReadData()* function. There is no timeout mechanism implemented in the UART driver.

The UART driver exits from a read operation in the following situations:

- During the receive procedure, if the *UartCancelReadData()* function is called. In this case, the read callback function is triggered with the parameters *UartStatus* set to gUartReadStatusCanceled_c and *UartNbBytes* set to number of characters received before the process cancellation.

In direct FIFO mode both *pBuf* and *NumberBytes* passed-in parameters are ignored. The contents of the HW fifo are not copied in *pBuf* as soon as characters are received, instead the *ReadCallback* function is called if:

- The Receive FIFO threshold is reached. The default receive threshold is set to 5 characters and can be changed by using the *UartSetReceiverThreshold* function.
- After receiving at least one character, no characters have been received for at least 8 character periods (uart data aging).

After the driver calls the receive callback in the above described situations, the user have to read directly the HW FIFO contents by calling the *UartGetByteFromRxBuffer* until it returns FALSE (FIFO emptied).

CAUTION

The Read callback is called in Interrupt Context. It is mandatory that the control is returned to the driver as fast as possible - usually the callback is implemented to have only a TS_SendEvent to the application task which actually handle the real copy.

6.7.8.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance.

Name

pBuf

In/Out

input

Description

Pointer to the destination buffer, where the received characters will be copied.

Name

NumberBytes

In/Out

input

Description

The number of bytes to receive.

Name

UseDirectFifoMode

In/Out

input

Description

Selects the read functionality. See the function description.

6.7.8.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrNullPointer_c
- gUartErrInvalidNrBytes_c
- gUartErrUartNotOpen_c
- gUartErrReadOngoing_c
- gUartErrNoCallbackDefined_c

6.7.9 UartCancelReadData ()

Prototype

```
UartErr_t UartCancelReadData
(
    uint8_t UartNumber
);
```

Description

The function is called to cancel a read operation on the UART peripheral.

There are some tests performed before canceling the operation. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value.

If all of these tests have been passed, the function is setting the internal driver variables, calls the read callback function and exits with *gUartErrNoError_c* value.

The read callback function is called with parameter *UartStatus* set to *gUartReadStatusCanceled_c* and parameter *UartNbBytes* set to number of characters received before the *UartCancelReadData()* call.

6.7.9.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance.

6.7.9.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrUartNotOpen_c

6.7.10 UartGetByteFromRxBuffer ()

Prototype

```
bool_t UartGetByteFromRxBuffer
(
    uint8_t UartNumber,
    uint8_t* pDst
);
```

Description

The function is called to copy one character from the Uart receive FIFO into the user-selected destination memory.

This function is intended to be used in the *DirectFifoMode* (see the *UartReadData* function) in which the driver exposes the receive HW FIFO directly to the user.

If at least one character is available in the HW receive FIFO, it will be extracted and updated in the memory location provided by the user at function call (*pDst*). The HW FIFO is emptied starting with the first received character (oldest in the FIFO). In this case the function returns TRUE.

If no character was received, or the FIFO was already emptied, this function returns FALSE and the *pDst* location is not updated.

6.7.10.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance.

Name

pDst

In/Out

input

Description

Pointer to the destination buffer, where the received character will be copied.

6.7.10.2 Returns

Description

Function execution status.

Possible Values

- 1 (TRUE) = At least one character is available in the selected Uart RX FIFO. The character was extracted from the RX FIFO (first in) and copied in the *pDst* location.
- 0 (FALSE) = No character is available in the selected Uart RX FIFO. The *pDst* location was not updated

6.7.11 UartWriteData ()

Prototype

```
UartErr_t UartWriteData
(
    uint8_t UartNumber,
    uint8_t* pBuf,
    uint16_t NumberBytes
);
```

Description

The function is called to start a write data operation to the UART peripheral.

There are some tests performed before starting the operation. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. If the pointer to the buffer where the data will be sent from is not initialized, the function exits with *gUartErrNullPointer_c* value. If the number of characters to send is zero, the function exits with *gUartErrInvalidNrBytes_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value. If a write operation is in place, the function exits

with *gUartErrWriteOngoing_c* value. If there is no callback function defined for the write operation, the function exits with *gUartErrNoCallbackDefined_c* value.

If all of these tests have been passed, the function sets the internal driver variables, peripheral hardware registers and exits with *gUartErrNoError_c* value.

After a *UartWriteData()* call the driver remains in the ‘uart write ongoing’ state, until all the characters are send. The upper layer can cancel at any time a write operation, by calling the *UartCancelWriteData()* function.

The UART driver exits from a write operation in the following situations:

- the number of characters specified in the *NumberBytes* parameter has been sent. In this case, the write callback function is triggered with the parameters *UartStatus* set to *gUartWriteStatusComplete_c* and *UartNbBytes* set to *NumberBytes*.
- during the transmit procedure, the *UartCancelWriteData()* function is called. In this case, the write callback function is triggered with the parameters *UartStatus* set to *gUartWriteStatusCanceled_c* and *UartNbBytes* set to number of characters sent before the process cancellation.

6.7.11.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to write to.

Name

pBuf

In/Out

input

Description

The pointer to a buffer where the data will be sent from.

Name

NumberBytes

In/Out

input

Description

The number of characters to send.

6.7.11.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrNullPointer_c
- gUartErrInvalidNrBytes_c
- gUartErrUartNotOpen_c
- gUartErrWriteOngoing_c
- gUartErrNoCallbackDefined_c

6.7.12 UartCancelWriteData ()

Prototype

```
UartErr_t UartCancelWriteData
(
    uint8_t UartNumber
);
```

Description

The function is called to cancel a write operation on the UART peripheral.

There are some tests performed before canceling the operation. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value.

If all of these tests have been passed, the function is setting the internal driver variables and exits with *gUartErrNoError_c* value.

The write callback function will be called with parameter *UartStatus* set to *gUartWriteStatusCanceled_c* and parameter *UartNbBytes* set to number of characters transmitted before the transmission was interrupted.

CAUTION

The CancelWriteData call will not stop the transmission immediately. The number of characters present in the TX hardware FIFO will still be sent, and after that, the transmission will be stopped and the callback function will be called. This is done in order to prevent errors that can appear by interrupting the process during a character transmission.

6.7.12.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of UART instance to cancel the write operation.

6.7.12.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrUartNotOpen_c

6.7.13 UartGetConfig()

Prototype

```
UartErr_t UartGetConfig
(
    uint8_t UartNumber,
    UartConfig_t* pConfig
);
```

Description

The function is called to get the communication parameters (parity, stop bits, baud rate, flow control) of the UART peripheral.

There are some tests performed before getting the configuration. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gUartErrNullPointer_c* value. If UART module is not initialized, the function exits with the *gUartErrUartNotOpen_c* value.

If all of these tests have been passed, the function is filling the structure with the communication parameters according with the values in the peripheral hardware registers and returns the *gUartErrNoError_c* value.

6.7.13.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of the UART instance to get configuration from.

Name

pConfig

In/Out

output

Description

The pointer to a structure where the configuration parameters shall be placed.

6.7.13.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrNullPointer_c
- gUartErrUartNotOpen_c

6.7.14 UartGetStatus ()

Prototype

```
UartErr_t UartGetStatus (
    uint8_t UartNumber
);
```

Description

The function is called to get the status of the UART peripheral instance.

There is a test performed before getting the status of the UART. If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value.

If UART module is not initialized, the function returns the *gUartErrUartNotOpen_c* value. If a read operation is in place, the function returns the *gUartErrReadOngoing_c* value. If a write operation is in place, the function returns the *gUartErrWriteOngoing_c* value. If no callback functions are defined for at least one of the read and write operations, the function returns the *gUartErrNoCallbackDefined_c* value.If none of these conditions happens, the function returns the *gUartErrNoError_c* value indicating that UART is open, callback functions are defined but no operation is in place.

CAUTION

Even if more than one operation are in place, the function returns a value indicating the presence of only one of them. The priority is the one detailed in the function return value description. Ex: if both read and write operations are currently running, the function will return *gUartErrReadOngoing_c* value.

6.7.14.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of UART instance to get status from.

6.7.14.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrUartNotOpen_c
- gUartErrReadOngoing_c
- gUartErrWriteOngoing_c
- gUartErrNoCallbacksDefined_c

6.7.15 UartClose ()

Prototype

```
UartErr_t UartClose
(
    uint8_t UartNumber
);
```

Description

The function is called to close the UART peripheral.

There are some tests performed before closing . If the UART peripheral instance number exceeds UART_NR_INSTANCES (number of UART peripherals present on platform), the function exits with *gUartErrWrongUartNumber_c* value.. If a read operation is in place, the function exits with

gUartErrReadOngoing_c value. If a write operation is in place, the function exits with *gUartErrWriteOngoing_c* value.

If all of these tests have been passed, the function clears the internal driver variables, disables the peripheral hardware transmitter and receiver modules and exits with *gUartErrNoError_c* value.

6.7.15.1 Function Parameters

Name

UartNumber

In/Out

input

Description

The number of UART instance to close.

6.7.15.2 Returns

Type

UartErr_t

Description

Function execution status.

Possible Values

- gUartErrNoError_c
- gUartErrWrongUartNumber_c
- gUartErrReadOngoing_c
- gUartErrWriteOngoing_c

6.7.16 UartIsr1 ()

Prototype

```
void UartIsr1 (void);
```

Description

The function will be set as interrupt routine for the first UART peripheral. This will be done using the interface provided by the ITC driver.

6.7.17 UartIsr2 ()

Prototype

```
void UartIsr2 (void);
```

Description

The function will be set as interrupt routine for the second UART peripheral. This will be done using the interface provided by the ITC driver.

Chapter 7
Timer (TMR) Driver

7.1 Overview

Each Timer module (TMR) contains four identical counter/timer groups. Each 16 bit counter/timer group contains a prescaler, a counter, a load register, a hold register, a capture register, two compare registers, two status and control registers, and one control register. All of the registers except the prescaler are readable/writable.

7.1.1 Hardware Module

The module has configuration registers that must be programmed to set the usage. For detailed information on the TMR module, see the *MC1322x Reference Manual* (MC1322xRM).

The TMR module design includes these distinctive features:

- Four - 16 bit counters/timers.
- Count up/down.
- Counters are cascadable.
- Programmable count modulo.
- Max count rate equals peripheral clock/2 for external clocks.
- Max count rate equals peripheral clock for internal clocks.
- Count once or repeatedly.
- Counters are preloadable.
- Compare Registers are preloadable. (Available with Compare Load feature)
- Counters can share available input pins.
- Separate prescaler for each counter.
- Each counter has capture and compare capability.
- Programmable operation during debug mode.
- Input glitch filter (optional)
- Counting start can be synchronized across counters. (optional)

7.1.2 Driver Functionality

The driver allows the developer to use the TMR functions easily. It allows the developer to:

- Enable/disable each timer
- Set the operating mode for each timer

- Set the input source and input polarity for each timer
- Set the secondary input source for each timer
- Set the count frequency for each timer
- Set the count length for each timer
- Set count direction for each timer
- Set output mode and output polarity for each timer
- Enable/disable outputs for each timer
- Set/clear master mode for each timer
- Set/clear output on master option for each timer
- Set/clear co-channel initialization for each timer
- Set timer(s) capture mode
- Set compare value for each timer
- Enable/disable interrupts (compare overflow, input edge) for each timer
- Set the compare load control for each timer
- Set the value to the timer register for each timer
- Get the current value of the timer register for each timer
- Register callbacks for timer events

7.2 Include Files

Table 7-1 shows the TMR driver files that must be included in the application C-files in order to have access to the driver function calls.

Table 7-1. TMR Driver Include Files

Include File Name	Description
Timer.h	Global header file that defines the public data types and enumerates the public functions prototypes

7.3 TMR Driver API Functions

Table 7-2 shows the available API functions for the TMR Driver.

Table 7-2. TMR Driver API Function List

TMR Driver API Function Name	Description
TmrInit ()	The function is called to initialize the RAM memory space needed for the driver.
TmrEnable ()	The function is called to enable timer module.
TmrDisable ()	The function is called to disable timer module.
TmrSetMode ()	The function is called to set the working mode for the timer module.
TmrSetConfig ()	The function is called to set the parameters for the timer module.

Table 7-2. TMR Driver API Function List (continued)

TmrSetCallbackFunction ()	The function is called to set the callback function corresponding to a specific event of the timer module.
TmrSetStatusControl ()	The function is called to set the control parameters for the timer module.
TmrSetCompStatusControl ()	The function is called to set the comparator control parameters for the timer module.
TmrWriteValue ()	The function is called to write a new value to the timer register to be used as the base for its next count.
TmrGetMode ()	The function is called to get the working mode of the timer module.
TmrGetConfig ()	The function is called to get the parameters of the timer module.
TmrGetStatusControl ()	The function is called to get the status of the timer module.
TmrGetCompStatusControl ()	The function is called to get the comparator control parameters from the timer module.
TmrReadValue ()	The function is called to read the current count value from the timer module.
TmrIsr ()	Interrupt service routine for the timer module.

7.4 Driver Exported Constants

7.4.1 TmrErr_t

Constant Structure

```
typedef enum
{
    gTmrErrNoError_c = 0,
    gTmrErrTimerIsEn_c,
    gTmrErrTimerIsDis_c,
    gTmrErrNullPointer_c,
    gTmrErrInvalidParameter_c,
    gTmrErrTimerBusy_c,
    gTmrErrMax_c
} TmrErr_t;
```

Description

Specifies the possible return values for the Timer driver API functions.

Constant Values

The constant values are self explanatory.

7.4.2 TmrEvent_t

Constant structure

```
typedef  enum
{
    gTmrComp1Event_c = 0,
    gTmrComp2Event_c,
    gTmrCompEvent_c,
    gTmrOverEvent_c,
    gTmrEdgeEvent_c,
    gTmrMaxEvent
} TmrEvent_t;
```

Description

This data type enumerates the values for timer events.

Constant values

- gTmrComp1Event_c - Event generated when a successful compare occurs between a counter and its COMP1 register (Only if TCF1EN in TmrComparatorStatusCtrl_t is set)
- gTmrComp2Event_c - Event generated when a successful compare occurs between a counter and its COMP2 register (Only if TCF2EN in TmrComparatorStatusCtrl_t is set)
- gTmrCompEvent_c - Event generated when a successful compare occurs between a counter and one of its compare registers (Only if TCFIE in TmrStatusCtrl_t is set)
- gTmrOverEvent_c - Event generated when a counter rolls over its maximum value (Only if TOFIE in TmrStatusCtrl_t is set)
- gTmrEdgeEvent_c - Event generated by a transition of the input signal either positive or negative depending on IPS setting in TmrStatusCtrl_t (Only if IEFIE in TmrStatusCtrl_t is set)

7.4.3 TmrMode_t

Constant Structure

```
typedef enum
{
    gTmrNoOperation_c = 0,
    gTmrCntRiseEdgPriSrc_c ,
    gTmrCntRiseEdgFallEdgPriSrc_c,
    gTmrCntRiseEdgWhileSecInputHighActive_c,
    gTmrQuadCnt_c,
    gTmrCntPriSrcRiseEdgSecSrcSpecDir_c,
    gTmrEdgSecSrcTriggerPriCntTillComp_c,
    gTmrSyncCnt_c,
    gTmrModeMax_c
} TmrMode_t;
```

Description

Specifies the possible values of the count mode.

Constant Values

- gTmrNoOperation_c - no operation count mode.
- gTmrCntRiseEdgPriSrc_c - count rising edge of primary source.
- gTmrCntRiseEdgFallEdgPriSrc_c - count rising and falling edges of primary source.
- gTmrCntRiseEdgWhileSecInputHighActive_c - count rising edges of primary source while secondary input high active.
- gTmrQuadCnt_c - quadrature count mode, uses primary and secondary sources.
- gTmrCntPriSrcRiseEdgSecSrcSpecDir_c - count primary source rising edges, secondary source specify direction.
- gTmrEdgSecSrcTriggerPriCntTillComp_c - edge of secondary source trigger primary till compare
- gTmrSyncCnt_c - synchronous counter mode.

7.4.4 TmrPrimaryCntSrc_t

Constant Structure

```
typedef enum
{
    gTmrPrimaryCnt0Input_c = 0,
    gTmrPrimaryCnt1Input_c,
    gTmrPrimaryCnt2Input_c,
    gTmrPrimaryCnt3Input_c,
    gTmrPrimaryCnt0Output_c,
    gTmrPrimaryCnt1Output_c,
    gTmrPrimaryCnt2Output_c,
    gTmrPrimaryCnt3Output_c,
    gTmrPrimaryClkDiv1_c,
    gTmrPrimaryClkDiv2_c,
    gTmrPrimaryClkDiv4_c,
    gTmrPrimaryClkDiv8_c,
    gTmrPrimaryClkDiv16_c,
    gTmrPrimaryClkDiv32_c,
    gTmrPrimaryClkDiv64_c,
    gTmrPrimaryClkDiv128_c,
    gTmrPrimaryClkSrcMax_c,
} TmrPrimaryCntSrc_t;
```

Description

Specifies the possible values for primary count source.

Constant Values

- gTmrPrimaryCnt0Input_c - primary count source is counter 0 input pin.
- gTmrPrimaryCnt1Input_c - primary count source is counter 1 input pin.
- gTmrPrimaryCnt2Input_c - primary count source is counter 2 input pin.
- gTmrPrimaryCnt3Input_c - primary count source is counter 3 input pin.
- gTmrPrimaryCnt0Output_c - primary count source is counter 0 output pin.

- gTmrPrimaryCnt1Output_c - primary count source is counter 1 output pin.
- gTmrPrimaryCnt2Output_c - primary count source is counter 2 output pin.
- gTmrPrimaryCnt3Output_c - primary count source is counter 3 output pin.
- gTmrPrimaryClkDiv1_c - primary count source is IP Bus clock divide by 1 prescaler.
- gPrimaryTmrClkDiv2_c - primary count source is IP Bus clock divide by 2 prescaler.
- gPrimaryTmrClkDiv4_c - primary count source is IP Bus clock divide by 4 prescaler.
- gTmrPrimaryClkDiv8_c - primary count source is IP Bus clock divide by 8 prescaler.
- gTmrPrimaryClkDiv16_c - primary count source is IP Bus clock divide by 16 prescaler.
- gTmrPrimaryClkDiv32_c - primary count source is IP Bus clock divide by 32 prescaler.
- gTmrPrimaryClkDiv64_c - primary count source is IP Bus clock divide by 64 prescaler.
- gTmrPrimaryClkDiv128_c - primary count source is IP Bus clock divide by 128 prescaler.

7.4.5 TmrOutputMode_t

Constant Structure

```
typedef enum
{
    gTmrAssert_c = 0,
    gTmrClearOF_c,
    gTmrSetOF_c,
    gTmrToggleOF_c,
    gTmrToggleOUsingAlternateReg_c,
    gTmrSetOnCompClearOnSecInputEdg_c,
    gTmrSetOnCompClearOnRollover_c,
    gTmrEnGateClock_c,
    gTmrOutputModeMax_c
} TmrOutputMode_t;
```

Description

Specifies the possible values for output mode.

Constant Values

- gTmrAssert_c - output is asserted while counter is active.
- gTmrClearOF_c - clear OFLAG output on successful compare.
- gTmrSetOF_c - set OFLAG output on successful compare.
- gTmrToggleOF_c - toggle OFLAG output on successful compare.
- gTmrToggleUsingAlternateReg_c - toggle OFLAG output using alternating compare registers.
- gTmrSetOnCompClearOnSecInputEdg_c - output set on compare, cleared on secondary source input edge.
- gTmrSetOnCompClearOnRollover_c - output set on compare, cleared on counter rollover.
- gTmrEnGateClock_c - enable gated clock output while counter is active.

7.4.6 TmrSecondaryCntSrc_t

Constant Structure

```
typedef enum
{
     gTmrSecondaryCnt0Input_c = 0,
     gTmrSecondaryCnt1Input_c,
     gTmrSecondaryCnt2Input_c,
     gTmrSecondaryCnt3Input_c,
     gTmrSecondaryCntSrcMax_c
} TmrSecondaryCntSrc_t;
```

Description

Specifies the possible values for the secondary count source.

Constant Values

- gTmrSecondaryCnt0Input_c - secondary count source is counter 0 input pin.
- gTmrSecondaryCnt0Input_c - secondary count source is counter 1 input pin.
- gTmrSecondaryCnt0Input_c - secondary count source is counter 2 input pin.
- gTmrSecondaryCnt0Input_c - secondary count source is counter 3 input pin.

7.4.7 TmrNumber_t

Constant Structure

```
typedef struct
{
     gTmr0_c = 0,
     gTmr1_c,
     gTmr2_c,
     gTmr3_c,
     gTmrMax_c
} TmrNumber_t;
```

Description

Specifies the possible values for timer channels.

Constant Values

- gTmr0_c - timer channel 0.
- gTmr1_c - timer channel 1.
- gTmr2_c - timer channel 2.
- gTmr3_c - timer channel 3.

7.5 Driver Exported Data Types

7.5.1 TmrConfig_t

Structure

```
typedef union TmrConfig_tag
{
     TmrOutputMode_t tmrOutputMode;
     bool_t   tmrCoInit;
     bool_t   tmrCntDir;
     bool_t   tmrCntLen;
     bool_t            tmrCntOnce;
     TmrSecondarySrcCnt_ttmrSecondarySrcCnt;
     TmrPrimarySrcCnt_t tmrPrimarySrcCnt;
} TmrConfig_t;
```

Description

Describes the configuration of the Timer module (operating mode, primary input source, secondary input source, count frequency, count length, count direction, co-channel initialization, output mode).

7.5.1.1 Structure Elements

- tmrOutputMode - it specifies the count mode of the timer
- tmrCoInit - it enables another timer/counter within the module to force the re-initialization of this counter/timer when it has an active compare event.
- tmrCntDir - it selects either the normal direction up, or the reverse direction down.
- tmrCntLen - it determines if whether the counter counts to the compare value and then re-initializes itself to the value specified in the load register, or the counter continues counting past the compare value, to the binary roll over.
- tmrCntOnce - selects continuous or one shot counting mode.
- tmrSecondarySrcCnt - it specifies the secondary count source of the timer.
- tmrPrimarySrcCnt - it specifies the primary count source of the timer.

7.5.2 TmrStatusCtrl_t

Structure

```
typedef union TmrStatusCtrl_tag{
      struct{
            uint16_t OEN:1;
            uint16_t OPS:1;
            uint16_t FORCE:1;
            uint16_t VAL:1;
            uint16_t EEOF:1;
            uint16_t MSTR:1;
            uint16_t CAPMODE:2;
            uint16_t INPUT:1;
            uint16_t IPS:1;
            uint16_t IEFIE:1;
            uint16_t IEF:1;
            uint16_t TOFIE:1;
            uint16_t TOF:1;
            uint16_t TCFIE:1;
            uint16_t TCF:1;
      } bitFields;
      uint16_t uintValue;
} TmrStatusCtrl_t;
```

Description

Describes the status and control of the Timer module.

7.5.2.1 Structure Elements

- bitFields - a structure with following fields:
  - OEN - output enable
  - OPS - output polarity select
  - FORCE - force the OFLAG output
  - VAL - forced OFLAG value
  - EEOF - enable external OFLAG force
  - MSTR - master mode
  - CAPMODE - input capture mode
  - INPUT - external input signal
  - IPS - input polarity select
  - IEFIE - input edge flag interrupt enable
  - IEF - input edge flag
  - TOFIE - timer overflow flag interrupt enable
  - TOF - timer overflow flag
  - TCFIE - timer compare flag interrupt enable
  - TCF - timer compare flag
- uintValue - the value corresponding to the bitFields structure.

7.5.3 TmrComparatorStatusCtrl_t

Structure

```
typedef union TmrComparatorStatusCtrl_tag{
    struct{
        uint16_t CL1:2;
        uint16_t CL2:2;
        uint16_t TCF1:1;
        uint16_t TCF2:1;
        uint16_t TCF1EN:1;
        uint16_t TCF2EN:1;
        uint16_t RESERVED:5;
        uint16_t FILT_EN:1;
        uint16_t DBG_EN:2;
    } bitFields;
    uint16_t uintValue;
} TmrComparatorStatusCtrl_t;
```

Description

Describes the comparator status and control of the Timer module.

7.5.3.1 Structure Elements

bitFields	is a structure with following fields:
CL1	compare load control 1;
CL2	compare load control 2;
TCF1	timer compare 1 interrupt source;
TCF2	timer compare 2 interrupt source;
TCF1EN	timer compare 1 interrupt enable;
TCF2EN	timer compare 2 interrupt enable;
RESERVED	reserved
FILT_EN	input filter enable;
DBG_EN	debug actions enable;
uintValue	is the value corresponding to the bitFields structure.

7.5.4 TmrCallbackFunction_t

Structure

```
typedef void (*TmrCallbackFunction_t)(TmrNumber_t tmrNumber);
```

Description

Callback function for the timer module driver.

7.6 Exported Macros

7.6.1 SetComp1Val()

The macro writes the value *val* in the COMP1 register corresponding to timer *tmrNumber*.

7.6.1.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the COMP1 register.

Returns

None

7.6.2 SetComp2Val()

The macro writes the value *val* in the COMP2 register corresponding to timer *tmrNumber*.

7.6.2.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the COMP2 register.

Returns

None

7.6.3 SetCaptureVal()

The macro writes the value *val* in the CAPTURE register corresponding to timer *tmrNumber*.

7.6.3.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the CAPTURE register.

Returns

None

7.6.4 SetLoadVal()

The macro writes the value *val* in the LOAD register corresponding to timer *tmrNumber*.

7.6.4.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the LOAD register.

Returns

None

7.6.5 SetHoldVal()

The macro writes the value *val* in the HOLD register corresponding to timer *tmrNumber*.

7.6.5.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the HOLD register.

Returns

None

7.6.6 SetCntrVal()

The macro writes the value *val* in the COUNTER register corresponding to timer *tmrNumber*.

7.6.6.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the COUNTER register.

Returns

None

7.6.7 SetCompLoad1Val()

The macro writes the value *val* in the COMPARE LOAD1 register corresponding to timer *tmrNumber*.

7.6.7.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the COMPARE LOAD1 register.

Returns

None

7.6.8 SetCompLoad2Val()

The macro writes the value *val* in the COMPARE LOAD2 register corresponding to timer *tmrNumber*.

7.6.8.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be modified.

Type

uint16_t

Name

val

In/Out

input

Description

The value of the COMPARE LOAD2 register.

Returns

None

7.6.9 GetComp1Val()

The macro reads the value *val* from the COMP1 register corresponding to timer *tmrNumber*.

7.6.9.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be read.

Returns

The value of the COMP1 register.

7.6.10 GetComp2Val()

The macro reads the value *val* from the COMP2 register corresponding to timer *tmrNumber*.

7.6.10.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be read.

Returns

Value of COMP2 register

7.6.11 GetCaptureVal()

The macro reads the value *val* from the CAPTURE register corresponding to timer *tmrNumber*.

7.6.11.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be read.

7.6.12 GetLoadVal()

The macro reads the value *val* in the LOAD register corresponding to timer *tmrNumber*.

7.6.12.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be read.

Returns

The value of the LOAD register.

7.6.13 GetHoldVal()

The macro reads the value *val* from the HOLD register corresponding to timer *tmrNumber*.

7.6.13.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

timer that will be read

Returns

The value of the HOLD read.

7.6.14 GetCntrVal()

The macro reads the value *val* from the COUNTER register corresponding to timer *tmrNumber*.

7.6.14.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be read.

Returns

The value of the COUNTER register.

7.6.15 GetCompLoad1Val()

The macro reads the value *val* from the COMPARE LOAD1 register corresponding to timer *tmrNumber*.

7.6.15.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

timer that will be read

Returns

The value of the COMPARE LOAD1 register.

7.6.16 GetCompLoad2Val()

The macro reads the value *val* from the COMPARE LOAD2 register corresponding to timer *tmrNumber*.

7.6.16.1 Macro Arguments

Type

TmrNumber_t

Name

tmrNumber

In/Out

input

Description

The timer that will be read.

Returns

The value of the COMPARE LOAD2 register.

7.7 TMR Driver API Function Descriptions

7.7.1 TmrInit ()

Prototype

```
void TmrInit (void);
```

Description

The function is called to initialize the RAM memory space needed for the driver. Actually this is not a interface function. It will be called only once, for all the timer driver instances. If this call is skipped, unpredictable behavior can appear when trying to access timer driver functions.

CAUTION

This function must be called prior to any other call to the timer driver interface functions.

7.7.2 TmrEnable()

Prototype

```
TmrErr_t TmrEnable
(
    TmrNumber_t tmrNumber
);
```

Description

The function is called to enable the corresponding *(tmrNumber)* timer module.

There are some tests performed before enabling the timer module. If the Timer instance number to open exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If timer module is already enabled, the function exits with *gTmrErrTimerIsEn_c* value.

If all of these tests have been passed, the function enables the timer specified as parameter and exits with *gTmrtErrNoError_c* value.

CAUTION

This function shall be called prior to any call of other driver functions. If it is called more than once, a *gTmrErrTimerIsEn_c* will be returned.

7.7.2.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the timer instance to enable.

7.7.2.2 Returns

Type

TmrErr_t

Description

The status of the operation.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrTimerIsEn_c

7.7.3 TmrDisable ()

Prototype

```
TmrErr_t TmrDisable
(
    TmrNumber_t tmrNumber
);
```

Description

The function is called to disable the corresponding *(tmrNumber)* timer module.

There are some tests performed before disabling the timer module. If the Timer instance number to open exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with

gTmrErrInvalidParameter_c value. If timer module is already disabled, the function exits with *gTmrErrTimerIsDis_c* value.

If all of these tests have been passed, the function disables the timer specified as parameter and exits with *gTmrtErrNoError_c* value.

CAUTION

If it is called more than once, a *gTmrErrTimerIsDis_c* will be returned.

7.7.3.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the timer instance to disable.

7.7.3.2 Returns

Type

TmrErr_t

Description

The status of the operation.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrTimerIsDis_c

7.7.4 TmrSetMode ()

Prototype

```
TmrErr_t TmrSetMode
(
    TmrNumber_t tmrNumber,
    TmrMode_t tmrMode
);
```

Description

The function is called to set the working mode for the Timer peripheral.

There are some tests performed before configuring the Timer peripheral. If the Timer peripheral instance number exceeds the gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If Timer module is not enabled, the function exits with the *gTmrErrTimerIsDis_c* value. If timer is started already the function exits with *gTmrErrTimerBusy_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received and returns the *gTmrErrNoError_c* value.

7.7.4.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to set mode.

Name

tmrMode

In/Out

input

Description

timer mode

7.7.4.2 Returns

Type

TmrErr_t

Description

The status of the operation.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrTimerIsDis_c
- gTmrErrTimerBusy_c

7.7.5 TmrSetConfig ()

Prototype

```
TmrErr_t TmrSetConfig
(
    TmrNumber_t tmrNumber,
    TmrConfig_t* pconfig
);
```

Description

The function is called to set the parameters (operating mode, primary input source, secondary input source, count frequency, count length, count direction, co-channel initialization, output mode) for the Timer peripheral.

There are some tests performed before configuring the Timer peripheral. If the Timer peripheral instance number exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If the pointer to the structure which contain the configuration parameters is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gTmrErrNoError_c* value.

7.7.5.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to configure.

Name

pConfig

In/Out

input

Description

The pointer to a structure containing the configuration parameters.

7.7.5.2 Returns

Type

TmrErr_t

Description

The status of the operation.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.6 TmrSetStatusControl ()

Prototype

```
TmrErr_t TmrSetStatusControl
(
    TmrNumber_t tmrNumber,
    TmrStatusCtrl_t* pStatusCtrl
);
```

Description

The function is called to set the control parameters (output polarity, master mode, outputs, capture mode, interrupts) for the Timer peripheral.

There are some tests performed before configuring the Timer peripheral. If the Timer peripheral instance number exceeds the gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If the pointer to the structure containing the configuration parameters is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the control structure and returns the *gTmrErrNoError_c* value.

7.7.6.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to configure.

Name

pStatusCtrl

In/Out

input

Description

The pointer to a structure containing the control parameters.

7.7.6.2 Returns

Type

TmrErr_t

Description

The status of the operation.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.7 TmrSetCompStatusControl ()

Prototype

```
TmrErr_t TmrSetCompStatusControl
(
    TmrNumber_t tmrNumber,
    TmrComparatorStatusCtrl_t* pCompStatusCtrl
```

```
);
```

Description

The function is called to set the comparator control parameters (compare load control, interrupts) for the Timer peripheral.

There are some tests performed before configuring the Timer peripheral. If the Timer peripheral instance number exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If the pointer to the structure containing the configuration parameters is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the control structure and returns the *gTmrErrNoError_c* value.

7.7.7.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to configure comparator.

Name

pCompStatusCtrl

In/Out

input

Description

The pointer to a structure containing the comparator configuration parameters.

7.7.7.2 Returns

Type

TmrErr_t

Description

The status of the operation.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.8 TmrSetCallbackFunction ()

Prototype

```
TmrErr_t TmrSetCallbackFunction
(
    TmrNumber_t tmrNumber,
    TmrEvent_t tmrEvent,
    TmrCallbackFunction_t pCallbackFunc
);
```

Description

The function is called to set the callback function for a specific event of the Timer peripheral.

There are some tests performed before setting the callback function. If the Timer peripheral instance number exceeds gTmrMax_c (number of Timer peripherals present on platform) or the event *tmrEvent* is grater than *gTmrMaxEvent,* the function exits with value *gTmrErrInvalidParameter_c*.

If all of these tests have been passed, the function is setting the internal driver callback pointers with the one received as parameter and returns the *gTmrErrNoError_c* value.

7.7.8.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to set the callback function.

Name

tmrEvent

In/Out

input

Description

The timer event associated with the callback.

Name

pCallback

In/Out

input

Description

The pointer to the callback function.

7.7.8.2 Returns

Type

TmrErr_t

Description

Function execution status.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c

7.7.9 TmrWriteValue ()

Prototype

```
TmrErr_t TmrWriteValue
(
    TmrNumber_t tmrNumber,
    uint16_t value
);
```

Description

The function is called to write a new value to the timer register to be used as the base for its next count.

There are some tests performed before starting the operation. If the Timer peripheral instance number exceeds gTmrMax_c(number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value.

7.7.9.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to write to.

Name

value

In/Out

input

Description

The value to be used as the base for the next count.

7.7.9.2 Returns

Type

TmrErr_t

Description

Function execution status.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c

7.7.10 TmrGetMode ()

Prototype

```
TmrErr_t TmrGetMode
(
    TmrNumber_t tmrNumber,
    TmrMode_t *pTmrMode
);
```

Description

The function is called to get the working mode for the Timer peripheral specified as parameter.

There are some tests performed before getting the configuration. If the Timer peripheral instance number exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with

gTmrErrInvalidParameter_c value. If the pointer to the variable where the timer mode shall be placed is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function is filling the variable according with the mode value in the peripheral hardware registers and returns the *gTmrErrNoError_c* value.

7.7.10.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the timer instance to get configuration from.

Name

pTmrMode

In/Out

output

Description

The pointer to a variable where current timer mode configuration will be placed.

7.7.10.2 Returns

Type

TmrErr_t

Description

Function execution status.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.11 TmrGetConfig ()

Prototype

```
TmrErr_t TmrGetConfig
(
    TmrNumber_t tmrNumber,
    TmrConfig_t* pconfig
);
```

Description

The function is called to get the parameters (operating mode, primary input source, secondary input source, count frequency, count length, count direction, co-channel initialization, output mode) for the Timer peripheral specified as parameter.

There are some tests performed before getting the configuration. If the Timer peripheral instance number exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function is filling the structure with the configuration parameters according with the values in the peripheral hardware registers and returns the *gTmrErrNoError_c* value.

7.7.11.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the timer instance to get configuration from.

Name

pconfig

In/Out

output

Description

The pointer to a structure where the configuration parameters shall be placed.

7.7.11.2 Returns

Type

TmrErr_t

Description

Function execution status.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.12 TmrGetStatusControl ()

Prototype

```
TmrErr_t TmrGetStatusControl
(
    TmrNumber_t tmrNumber,
    TmrStatusCtrl_t* pStatusCtrl
);
```

Description

The function is called to get the status of the Timer peripheral instance.

There are some tests performed before getting the configuration. If the Timer peripheral instance number exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function is filling the structure with the status parameters according with the values in the peripheral hardware registers and returns the *gTmrErrNoError_c* value.

7.7.12.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the timer get status from.

Name

pStatusCtrl

In/Out

input

Description

The pointer to a structure where the status parameters shall be placed.

7.7.12.2 Returns

Type

TmrErr_t

Description

Function execution status.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.13 TmrGetCompStatusControl ()

Prototype

```
TmrErr_t TmrGetCompStatusControl
(
    TmrNumber_t tmrNumber,
    TmrComparatorStatusCtrl_t* pCompStatusCtrl
);
```

Description

The function is called to get the comparator control parameters (compare load control, interrupts) from the Timer peripheral.

There are some tests performed before configuring the Timer peripheral. If the Timer peripheral instance number exceeds the gTmrMax_c (number of Timer peripherals present on platform), the function exits

with *gTmrErrInvalidParameter_c* value. If the pointer to the structure containing the configuration parameters is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the control structure and returns the *gTmrErrNoError_c* value.

7.7.13.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to get comparator status.

Name

pCompStatusCtrl

In/Out

input

Description

The pointer to a structure where the comparator status parameters shall be placed.

7.7.13.2 Returns

Type

TmrErr_t

Description

The status of the operation.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.14 TmrReadValue ()

Prototype

```
TmrErr_t TmrReadValue
(
    TmrNumber_t tmrNumber,
    uint16_t* pvalue
);
```

Description

The function is called to read a new value from the counter register.

There are some tests performed before starting the operation. If the Timer peripheral instance number exceeds gTmrMax_c (number of Timer peripherals present on platform), the function exits with *gTmrErrInvalidParameter_c* value. If the pointer to the memory where the new value will be placed is not initialized, the function exits with *gTmrErrNullPointer_c* value.

If all of these tests have been passed, the function gets the count value from peripheral hardware register and returns the *gTmrErrNoError_c* value.

7.7.14.1 Function Parameters

Name

tmrNumber

In/Out

input

Description

The number of the Timer instance to read from.

Name

pvalue

In/Out

input

Description

The memory place where the new value from timer register will be placed.

7.7.14.2 Returns

Type

TmrErr_t

Description

Function execution status.

Possible Values

- gTmrErrNoError_c
- gTmrErrInvalidParameter_c
- gTmrErrNullPointer_c

7.7.15 TmrIsr()

Prototype

```
void TmrIsr (void);
```

Description

The function will be set as interrupt routine for the timer peripheral. This will be done using the interface provided by the ITC driver.

Chapter 8
Inter-integrated Circuit (I2C) Driver

8.1 Overview

The inter-IC (IIC or I2C) bus is a two-wire—serial data (SDA) and serial clock (SCL)— bidirectional serial bus that provides a simple efficient method of data exchange between the system and other devices, such as microcontrollers, EEPROMs, real-time clock devices, A/D converters, and LCDs. The two-wire bus minimizes the interconnections between devices. The synchronous, multi-master bus of the I2C allows the connection of additional devices to the bus for expansion and system development. The bus includes collision detection and arbitration that prevent data corruption if two or more masters attempt to control the bus simultaneously.

8.1.1 Hardware Module

The module has configuration registers that must be programmed to set the usage. For detailed information on the I2C module, see the *MC1322x Reference Manual* (MC1322xRM).

The I2C interface includes the following features:

- Two-wire interface
- Multi-master operational
- Arbitration lost interrupt with automatic mode switching from master to slave
- Calling address identification interrupt
- START and STOP signal generation/detection
- Acknowledge bit generation/detection
- Bus busy detection
- Software-programmable clock frequency
- Software-selectable acknowledge bit
- On-chip filtering for spikes on the bus

8.1.2 Driver Functionality

The driver allows the developer to use the I2C functions easily. It allows the developer to:

- Enable/disable I2Cmodule
- Set the master/slave operation mode
- Configure clock frequency
- Write data to I2C bus

- Read data from I2C bus
- Register callback for I2C event

8.2 Include Files

Table 8-1 shows the I2C driver files that must be included in the application C-files in order to have access to the driver function calls.

Table 8-1. I2C Driver Include Files

Include File Name	Description
I2C_Interface.h	Global header file that defines the public data types and enumerates the public functions prototypes.

8.3 I2C Driver API Functions

Table 8-2 shows the available API functions for the I2C Driver.

Table 8-2. I2C Driver API Function List

I2C Driver API Function Name	Description
I2c_Init()	The function is called to init the status of the module and clear I2C event callback.
I2c_Enable()	The function is called to enable I2C module.
I2c_Disable()	The function is called to disable the I2C module.
I2c_RecoverBus()	The function is called to recover I2C bus.
I2c_SetConfig()	The function is called to set the parameters for the I2C peripheral.
I2c_SetCallbackFunction()	The function is called to set the callback function for the I2C peripheral.
I2c_GetStatus()	The function is called to get the status of the I2C module.
I2c_CancelTransfer()	The function is called to cancel transmitting/receiving data.
I2c_SendData()	The function is called to send a sequence of bytes.
I2c_ReceiveData()	The function is called to receive a sequence of bytes.
I2c_Isr()	The interrupt service routine for the I2C peripheral.

8.4 Driver Exported Constants

8.4.1 I2cErr_t

Constant Structure

```
typedef enum
{
    gI2cErrNoError_c = 0,
    gI2cErrModuleIsEn_c,
    gI2cErrModuleIsDis_c,
    gI2cErrNullPointer_c,
    gI2cErrInvalidOp_c,
    gI2cErrBusBusy_c,
    gI2cErrNoDevResp_c,
    gI2CErrNoAckResp_c,
    gI2CErrTransferInProgress_c,
    gI2cErrModuleBusy_c,
    gI2cErrArbLost_c,
    gI2cErrMax_c
}I2cErr_t;
```

Description

Specifies the possible return values for the I2C driver API functions.

Constant Values

The constant values are self explanatory.

8.4.2 I2cTransferMode_t

Constant Structure

```
typedef enum
{
    gI2cSlvTransfer_c = 0,
    gI2cMstrReleaseBus_c,
    gI2cMstrHoldBus_c,
    gI2cTransferModeMax_c
}I2cTransferMode_t;
```

Description

Specifies the possible modes of transfer.

Constant Values

- gI2cSlvTransfer_c: data will be transferred in slave mode
- gI2cMstrReleaseBus_c: data will be transferred in master mode and the module will release the bus after transfer.

- gI2cMstrHoldBus_c: data will be transferred in master mode and the module will hold the bus after transfer.

8.4.3 I2cResponse_t

Constant Structure

```
typedef enum
{
    gI2cAckResponse_c = 0,
    gI2cNoAckResponse_c
} I2cResponse_t;
```

Description

Specifies if acknowledge response is used or not.

Constant Values

- gI2cAckResponse_c: I2C acknowledge reponse.
- gI2cNoAckResponse_c: I2C no acknowledge response.

8.4.4 I2cOperation_t

Constant Structure

```
typedef enum
{
    gI2cWrite_c = 0,
    gI2cRead_c
} I2cOperation_t;
```

Description

Specifies the operation type performed by the device (Write or read).

Constant Values

- gI2cWrite_c: I2C write operation.
- gI2cRead_c: I2C read operation.

8.4.5 I2cBusStatus_t

Constant Structure

```
typedef enum
{
    gI2cBusIdle_c = 0,
    gI2cBusBusy_c
} I2cBusStatus_t;
```

Description

Specifies the current status of the I2C bus.

Constant Values

- gI2cBusIdle_c: I2C bus is idle.
- gI2cBusBusy_c: I2C bus is busy.

8.4.6 I2cTransferType_t

Constant Structure

```
typedef enum {
    gI2cTransmitData_c = 0,
    gI2cReceiveData_c
}I2cTransferType_t;
```

Description

Specifies the possible types of transfer.

Constant Values

- gI2cTransmitData_c: module transmits data
- gI2cReceiveData_c: module receives data

8.5 Exported Structures and Data Types

8.5.1 I2cConfig_t

Structure

```
typedef struct I2cConfig_tag{
    uint8_t  slaveAddress;
    uint8_t  freqDivider;
    uint8_t  saplingRate;
    bool_t   i2cInterruptEn;
    bool_t   i2cBroadcastEn;
}I2cConfig_t;
```

Description

Describes the configuration of the I2C module.

8.5.1.1 Structure Elements

The structure elements are self explanatory.

8.5.2 I2cCallbackFunction_t

Structure

```
typedef  void (*I2cCallbackFunction_t)(uint16_t transfBytesNo, uint16_t status);
```

Description

Callback function for the I2C module driver.

8.6 Driver Exported Macros

8.6.1 I2cGetFDRVal()

The frequency divide value is calculated based on SYSTEM_CLOCK and I2C_CLOCK values () that must be defined by the user.

Macro Arguments

None

Returns

The divider value for the frequency divide register (FDR).

8.6.2 I2cGetDFSRVal()

The value for the digital filter sampling rate register (DFSR).

Macro Arguments

None.

Returns

0x01

8.7 I2C Driver API Function Descriptions

8.7.1 I2c_Init ()

Prototype

```
void I2c_Init (void);
```

Description

The function is called to clear I2C registers, clear I2C events callbacks and to init the status of the module. It shall be called only once. If this call is skipped, unpredictable behavior can appear when trying to access I2C driver functions.

CAUTION

This function must be called prior to any other call to other I2C driver interface functions !!!

8.7.2 I2c_Enable ()

Prototype

```
I2cErr_t I2c_Enable (void);
```

Description

The function is called to enable I2C module.

There are some tests performed before enabling the I2C module. If the I2C module is already enabled, the function exits with *gI2cErrModuleIsEn_c* value.

If all of these tests have been passed, the function enables the I2C module and exits with *gI2cErrNoError_c* value.

8.7.2.1 Returns

Type

I2cErr_t

Description

The status of the operation.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsEn_c

8.7.3 I2c_Disable ()

Prototype

```
I2cErr_t I2c_Disable (void);
```

Description

The function is called to disable the I2C module.

There are some tests performed before disabling the I2C module. If I2C module is already disabled, the function exits with *gI2cErrModuleIsDis_c* value. If I2C module is busy, which means that previous operation (transmit/receive) is not finished (regardless the module is master or slave) or module is master over the I2C bus, the function exits with *gI2cErrModuleBusy_c* value.

If all of these tests have been passed, the function disables the I2C module and exits with *gI2cErrNoError_c* value.

8.7.3.1 Returns

Type

I2cErr_t

Description

The status of the operation.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsDis_c
- gI2cErrModuleBusy_c

8.7.4 I2c_SetConfig ()

Prototype

```
I2cErr_t I2c_SetConfig
(
    I2cConfig_t *pI2cConfig
);
```

Description

The function is called to set the parameters for the I2C peripheral (enable/disable interrupts, enable/disable broadcast capability, set I2C frequency divider, I2C sampling rate and slave address).

There are some tests performed before configuring the I2C peripheral. If the I2C module isn't enabled the function exits with *gI2cErrModuleIsDis_c* value. If the pointer to the structure containing the configuration parameters is not initialized, the function exits with *gI2cErrNullPointer_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gI2cErrNoError_c* value.

8.7.4.1 Function Parameters

Name

pI2cConfig

In/Out

input

Description

The pointer to a structure containing the configuration parameters.

8.7.4.2 Returns

Type

I2cErr_t

Description

The status of the operation.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsDis_c
- gI2cErrNullPointer_c

8.7.5 I2c_RecoverBus ()

Prototype

```
I2cErr_t I2c_RecoverBus (void);
```

Description

The function is called to recover I2C bus. Forces the I2C module to become the I2C bus master and drive the SCL signal (even though SDA may already be driven, which indicates that the bus is busy).

There are some tests performed before configuring the I2C peripheral. If the I2C module isn't enabled the function exits with *gI2cErrModuleIsDis_c* value.

If all of these tests have been passed, the function forces the I2C module to become the I2C bus master and returns the *gI2cErrNoError_c* value.

8.7.5.1 Returns

Type

I2cErr_t

Description

The status of the operation.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsDis_c

8.7.6 I2c_SetCallbackFunction ()

Prototype

```
I2cErr_t I2c_SetCallbackFunction
(
    I2cCallbackFunction_t pI2cCallback
);
```

Description

The function is called to set the callback function for the I2C peripheral.

There are some tests performed before setting the callback function. If the module is not enabled, the function exits with *gI2cErrModuleIsDis_c* value.

If this test has been passed, the function will set the callback pointers with the one received in the structure and returns the *gI2cErrNoError_c* value.

8.7.6.1 Function Parameters

Name

pI2CCallback

In/Out

input

Description

The pointer to a data containing the address of the callback function.

8.7.6.2 Returns

Type

I2cErr_t

Description

Function execution status.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsDis_c

8.7.7 I2c_SendData ()

Prototype

```
I2cErr_t I2c_SendData
(
    uint8_t slaveDestAddr,
    uint8_t *i2cBuffData,
    uint16_t dataLength,
    I2cTransferMode_t transferMode
);
```

Description

This function is used to send a sequence of bytes to the device specified by *slaveDestAddr* parameter (if master mode is used) or to the device which will address the module (if slave mode is used). The function attempts to send *dataLength* bytes which are read from *i2cBuffData* buffer.

For the master mode the function checks if the bus is busy (there is another master device that is transmitting data) and returns *gI2cErrBusBusy_c* in this case. Otherwise, the communication begins with RESTART sequence if the previous transfer was in master mode and the module didn't release the bus (the type of the previous transfer was *gI2cMstrHoldBus_c*). After that, the response from the slave device is checked. If the slave device doesn't acknowledge the byte sent the function exits with *gI2cErrNoDevResp_c* error, otherwise the function start sending data from buffer received as parameter. If the receiving device doesn't acknowledge at least one byte sent during transmission (except last byte) the function returns *gI2cErrNoAckResp_c*. The module sends STOP sequence at the end of data transfer if *transferType* value is equal to *gI2cMstrReleaseBus_c*, otherwise STOP command is not sent on the bus.

For the slave mode, all data bytes are sent when the device is addressed as slave (data is transmitted using interrupts) and the desired operation is to read data from slave device.

There are some tests performed before sending data on the I2C bus. If the pointer to memory containing data is not initialized, the function exits with *gI2cErrNullPointer_c* value. If the previous transaction is not completed the function exits with *gI2cErrModuleBusy_c* value. If the function is called to transmit data as slave and the interrupts are not enabled the function exits with *gI2cErrInvalidOp_c* value.

If all of these tests have been passed and all data bytes were sent the function returns the *gI2cErrNoError_c* value. After ending of transmitting data, the callback (if not NULL) is called with number of bytes successfully sent and the status of transfer as parameters.

8.7.7.1 Function Parameters

Name

slaveDestAddr

In/Out

input

Description

The destination address of the slave device, only if the module is configured as master (if the module is configured as slave, this parameter will be ignored).

Name

i2cBuffData

In/Out

input

Description

The pointer to the memory location where the data to be send is stored.

Name

dataLength

In/Out

input

Description

length of data

Name

transferMode

In/Out

input

Description

the type of data transfer

8.7.7.2 Returns

Type

I2cErr_t

Description

The status of the operation.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsDis_c
- gI2cErrModuleBusy_c
- gI2CErrNullPointer_c
- gI2cErrInvalidOp_c
- gI2cErrBusBusy_c
- gI2cErrNoAckResp_c
- gI2cErrNoDevResp_c

8.7.8 I2c_ReceiveData ()

Prototype

```
I2cErr_t I2c_ReceiveData
(
    uint8_t slaveDestAddr,
    uint8_t *i2cBuffData,
    uint16_t dataLength,
    I2cTransferMode_t transferMode
);
```

Description

This function is used to receive a sequence of bytes from the device specified by *slaveDestAddr* parameter (if master mode is used) or from the device which will address the module (if slave mode is used). The function attempts to receive *dataLength* bytes which will be stored in *i2cBuffData* buffer.

For the master mode the function checks if the bus is not busy (there is another master device that is transmitting data) and returns *gI2cErrBusBusy_c* if the bus is busy. Otherwise, the communication begins with RESTART sequence if the previous transfer was in master mode and the module doesn't release the bus (the type of the previous transfer was *gI2cMstrHoldBus_c*). After *I2cOpenDevice* is called, the response from the slave device is checked. After communication ends, the module sends STOP sequence at the end of data transfer if *transferType* value is equal to *gI2cMstrReleaseBus_c*, otherwise STOP command is not sent on the bus.

For the slave mode, all data bytes are read when the device is addressed as slave (data is transmitted using interrupts) and the desired operation is to read from a master device.

There are some tests performed before reading data from the I2C bus. If the pointer to memory containing the data is not initialized, the function exits with *gI2cErrNullPointer_c* value. If the previous transaction is not completed the function exits with *gI2cErrModuleBusy_c* value. If the function is called to receive data as slave and the interrupts are not enabled the function exits with *gI2cErrInvalidOp_c* value.

If all of these tests have been passed and all data bytes were received the function returns the *gI2cErrNoError_c* value.

8.7.8.1 Function Parameters

Name

slaveDestAddr

In/Out

input

Description

The destination address of the slave device from where the data will be received, only when module is working in master mode (if transfer type desired is slave mode this parameter will be ignored).

Name

i2cBuffData

In/Out

input

Description

The pointer to a memory location where data will be placed.

Name

dataLength

In/Out

input

Description

data length

Name

transferMode

In/Out

input

Description

The type of data transfer.

8.7.8.2 Returns

Type

I2cErr_t

Description

The status of the operation.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsDis_c
- gI2cErrModuleBusy_c
- gI2cErrNullPointer_c
- gI2cErrBusBusy_c
- gI2cErrInvalidOp_c
- gI2cErrArbLost_c
- gI2cErrNoDevResp_c

8.7.9 I2c_GetStatus()

Prototype

```
I2cErr_t I2c_GetStatus
(
    uint8_t *status
);
```

Description

The function is called to get the status of the I2C module.

There are some tests performed before getting the status. If the pointer to the location where the status will be placed is not initialized, the function exits with *gI2cErrNullPointer_c* value.

If all of these tests have been passed, the function is filling the structure with the status of the I2C module and returns the *gI2cErrNoError_c* value.

8.7.9.1 Function Parameters

Name

status

In/Out

output

Description

The pointer to a memory location where the I2C module's status shall be placed.

8.7.9.2 Returns

Type

I2cErr_t

Description

Function execution status.

Possible Values

- gI2cErrNoError_c
- gI2cErrNullPointer_c

8.7.10 I2c_CancelTransfer ()

Prototype

```
I2cErr_t I2c_CancelTransfer
(
    I2cTransferType_t transferType
);
```

Description

The function is called to cancel transmitting/receiving data.

There are some tests performed before canceling the transfer operation. If the function is called when the module is master and interrupts are not enabled the function exits with *gI2cErrInvalidOp_c* value.

If all of these tests have been passed, the transfer is cancelled and the function returns the *gI2cErrNoError_c* value.

8.7.10.1 Function Parameters

Type

I2cTransferType_t

Name

transferType

In/Out

input

Description

The type of the transfer that will be canceled.

8.7.10.2 Returns

Type

I2cErr_t

Description

Function execution status.

Possible Values

- gI2cErrNoError_c
- gI2cErrModuleIsDis_c

- gI2cErrInvalidOp_c

8.7.11 I2c_Isr ()

Prototype

```
void I2c_Isr(void);
```

Description

The function will be set as interrupt routine for the I2C peripheral. This will be done using the interface provided by the ITC driver.

Function Parameters

None

Returns

None

Chapter 9
Synchronous Serial Interface (SSI) Driver

9.1 Overview

The SSI is a full-duplex, serial port that allows the chip to communicate with a variety of serial devices. These serial devices can be standard CODer-DECoder (CODECs), Digital Signal Processors (DSPs), microprocessors, peripherals, and popular industry audio CODECs that implement the inter-IC sound bus standard (I2S).

An SSI is typically used to transfer samples in a periodic manner. The SSI consists of independent transmitter and receiver sections with independent clock generation and frame synchronization.

9.1.1 Hardware Module

The module has configuration registers that must be programmed to set the usage. For detailed information on the SSI module, see the *MC1322x Reference Manual* (MC1322xRM).

The SSI includes the following features:

- Synchronous transmit and receive sections with shared internal/external clocks and frame syncs, operating in Master or Slave mode.
- Normal mode operation using frame sync
- Network mode operation allowing multiple devices to share the port with as many as 32 time slots
- Gated Clock mode operation requiring no frame sync
- Separate Transmit and Receive FIFOs. Each of the FIFOs is 8x24 bits.
- Programmable data interface modes such like I2S, LSB, MSB aligned
- Programmable word length (8, 10, 12, 16, 18, 20, 22 or 24 bits)
- Program options for frame sync and clock generation
- Programmable I2S modes (Master, Slave).
- Programmable internal clock divider
- Time Slot Mask Registers for reduced CPU overhead (for Tx and Rx both)
- SSI power-down feature
- IP Interface for register accesses, compliant to SRS 3.0.2 standard

9.1.2 Driver Functionality

The driver allows the developer to use the SSI functions easily. It allows the developer to:

- Enable/Disable the SSI module

- Set the master/slave operation mode
- Set the normal/network operation mode
- Configure the clock output generation mode (for master operation)
- Configure the transmit / receive clock frequency
- Configure the data transfer operation including data size and FIFO usage
- Write data to SSI bus
- Read data from SSI bus
- Register callbacks for SSI events

9.2 Include Files

Table 9-1 shows the SSI driver files that must be included in the application C-files in order to have access to the driver function calls.

Table 9-1. I2C Driver Include Files

Include File Name	Description
Ssi_Interface.h	Global header file that defines the public data types and enumerates the public functions prototypes.

9.3 SSI Driver API Function Descriptions

Table 9-2 shows the available API functions for the SSI Driver.

Table 9-2. SSI Driver API Function List

SSI Driver API Function Name	Description
SSI_Init()	The function is called to init the SSI module.
SSI_Enable()	The function is called to enable/disable the SSI module.
SSI_SetConfig()	The function is called to set the configuration parameters for the SSI peripheral.
SSI_SetClockConfig()	The function is called to set the configuration for the SSI clock.
SSI_SetTxRxConfig()	The function is called to set the configuration for the SSI TX/RX path.
SSI_TxData()	The function is called to send a sequence of bytes over the SSI bus
SSI_RxData()	The function is called to read data from the SSI bus.
SSI_StartContinuousRx()	The function is called to start a continuous receive sequence from the SSI bus.
SSI_Abort()	The function is called to abort a TX or RX operation.
SSI_SetTxCallback()	The function is called to set/reset the TX completion callback function.
SSI_SetRxCallback()	The function is called to set/reset the RX completion callback function.
SSI_SetContinuousRxCallback()	The function is called to set/reset the continuous RX callback function.
SSI_ISR()	The interrupt service routine provided by the SSI driver

9.4 Driver Exported Constants

9.4.1 SsiErr_t

Constant Structure

```
typedef enum
{
    gSsiErrNoError_c = 0,
    gSsiAlreadyEnabled_c,
    gSsiAlreadyDisabled_c,
    gSsiErrNullPointer_c,
    gSsiErrDataLength_c,
    gSsiErrWordSize_c,
    gSsiErrInvalidConfiguration_c,
    gSsiErrNoCallback_c,
    gSsiErrInvalidOpType_c,
    gSsiErrBusy_c,
    gSsiErrNotBusy_c,
    gSsiErrMax_c
} SsiErr_t;
```

Description

It specifies the possible return values for the SSI driver API functions.

Constant Values

The constant values are self explanatory.

9.4.2 SsiMode_t

Constant Structure

```
typedef enum
{
    gSsiNormalMode_c = 0,
    gSsiI2SMasterMode_c,
    gSsiI2SSlaveMode_c,
    gSsiModeMax_c
} SsiMode_t;
```

Description

Specifies the possible values of the working mode.

Constant Values

- gSsiNormalMode_c: the SSI module works in normal mode.
- gSsiI2SMasterMode_c: the SSI module works in I2S master mode
- gSsiI2SSlaveMode_c: the SSI module works in I2S slave mode

9.4.3 SsiWordSize_t

Constant Structure

```
typedef enum
{
    gSsiWordSize8bit_c = 0,
    gSsiWordSize16bit_c = 1,
    gSsiWordSizeInvalid_c = 2,
    gSsiWordSize32bit_c = 3,
    gSsiWordSizeMax_c
} SsiWordSize_t;
```

Description

Specifies the possible values for the word size used in the TX/RX functions. This enumerations refers to the possible word sizes for the provided TX/RX buffers and must be correlated with the HW configured SSI word sizes (SsiClockConfig_t.bit.ssiWL).

Constant Values

- gSsiWordSize8bit_c: Data will be taken from the TX buffer (/placed in the RX buffer) with byte (8bits) alignment.
- gSsiWordSize16bit_c: Data will be taken from the TX buffer (/placed in the RX buffer) with short (16bits) alignment.
- gSsiWordSize32bit_c: Data will be taken from the TX buffer (/placed in the RX buffer) with long (32bits) alignment.

9.4.4 SsiOpType_t

Constant Structure

```
typedef enum
{
    gSsiOpTypeTx_c = 0,
    gSsiOpTypeRx_c,
    gSsiOpTypeMax_c
}SsiOpType_t;
```

Description

Specifies the operation type for the SSI_SetTxRxConfig() and SSI_Abort() functions.

9.5 Driver Exported Structures

9.5.1 SsiConfig_t

Constant Structure

```
typedef struct SSIConfig_tag
{
    bool_t      ssiInterruptEn;
    SsiMode_t   ssiMode;
    bool_t      ssiNetworkMode;
    bool_t      ssiGatedTxClockMode;
    bool_t      ssiGatedRxClockMode;
} SsiConfig_t;
```

Description

Describes the configuration of the SSI module.

Usage

Uses as a parameter type in SSI_SetConfig() function.

9.5.1.1 Structure Elements

ssiInterruptEn	enable/disable SSI module interrupts. (TRUE/FALSE)
ssiMode	configures the SSI module working mode.
ssiNetworkMode	configures the usage of the network mode (TRUE/FALSE)
ssiGatedTxClockMode	The transmitter works in the Gated mode (TRUE/FALSE) (clock generation is stopped between frames in case if the internal clock is selected)
ssiGatedRxClockMode	The receiver works in the Gated mode (TRUE/FALSE) (clock generation is stopped between frames in case if the internal clock is selected)

9.5.2 SsiClockConfig_t

Constant Structure

```
typedef union SSIClockConfig_tag
{
    uint32_t ssiClockConfigWord;
    struct{
        uint32_t      ssiPM: 8;
        uint32_t      ssiDC : 5 ;
        uint32_t      ssiWL : 4 ;
        uint32_t      ssiPSR : 1;
        uint32_t      ssiDIV2 : 1;
    } bit;
} SsiClockConfig_t;
```

Description

Describes the configuration of the SSI clock.

Usage

Used as a parameter type in SSI_SetClockConfig() function.

9.5.2.1 Structure Elements

ssiDIV2	enable/disable the usage of the divide-by-two divider. (1/0)
ssiPSR	enable/disable the usage of the divide-by-eight prescaler. (1/0)
ssiWL	Word Length Control - configures the length of the data words being transmitted by the SSI
ssiDC	Frame Rate Divider Control – configures the word transfer rate (normal mode) / number of words per frame (network mode).
ssiPM	Prescaler Modulus Select – configures the divider in clock generation

Finally the internal bit clock results from the following formula:

$$f_{INT\_BIT\_CLK} = f_{SYS\_CLK} / [(DIV2 + 1) \times (7 \times PSR + 1) \times (PM + 1) \times 2]$$

$$f_{FRAME\_SYN\_CLK} = (f_{INT\_BIT\_CLK}) / [(DC + 1) \times WL]$$

9.5.3 SsiTxRxConfig_t

Constant Structure

```
typedef union SSITxRxConfig_tag
{
    uint16_t      ssiTxRxConfigWord;
    struct{
        uint16_t      ssiEFS : 1;
        uint16_t      ssiFSL : 1;
        uint16_t      ssiFSI : 1;
        uint16_t      ssiSCKP : 1;
        uint16_t      ssiSHFD: 1;
        uint16_t      ssiCLKDIR : 1 ;
        uint16_t      ssiFDIR : 1 ;
        uint16_t      ssiFEN : 1;
        uint16_t      ssiReserved2: 1;
        uint16_t      ssiBIT0 : 1;
        uint16_t      ssiRxEXT : 1;
    } bit;
} SsiTxRxConfig_t;
```

Description

Describes the configuration of the SSI Tx/Rx operation.

Usage

Used as a parameter type in SSI_SetTxRxConfig() function.

9.5.3.1 Structure Elements

ssiRXEXT	Rx data extension for LSB aligned data. This field is ignored for TX configuration 0 = Sign extension turned off 1 = Sign extension turned on
ssiBIT0	0 = TX/RX shifting begins with bit 15/31. 1 = TX/RX shifting begins with bit 0. In both modes, the shifting direction is controlled by the ssiTSHFD bit.
ssiFEN	enable/disable the usage of the TX/RX FIFO. (1/0)
ssiFDIR	0 = TX/RX frame sync is external 1 = TX/RX frame sync is internal
ssiCLKDIR	0 = TX/RX clock is external 1 = TX/RX clock is internal
ssiSHFD	0 = Data transmitted/received MSB first 1 = Data transmitted/received LSB first
ssiSCKP	0 = Data clocked out/in on rising edge 1 = Data clocked out/in on falling edge
ssiFSI	0 = frame sync active high 1 = frame sync active low
ssiFSL	0 = frame sync is one word long 1 = frame sync is one clock bit long
ssiEFS	0 = frame sync issued one data bit before TX/RX 1 = frame sync issues as the first data bit is transmitted

9.5.4 SsiTxCallback_t

Constant Structure

```
typedef   void (*SSITxCallback_t)(void);
```

Description

Used as a function pointer type for the SSI TX callback.

Usage

Used as a parameter type in the SSISetTxCallback() function.

9.5.5 SsiRxCallback_t

Constant Structure

```
typedef   void (*SSIRxCallback_t)(void);
```

Description

Used as a function pointer type for the SSI RX callback.

Usage

Used as a parameter type in the SSISetRxCallback() function.

9.5.6 SsiContinuousRxCallback_t

Constant Structure

```
typedef void* (*SSIContinuousRxCallback_t)(uint8_t *length);
```

Description

Used as a function pointer type for the continuous SSI RX callback.

Usage

Used as a parameter type in the SSI_SetContinuousRxCallback() function.

9.6 Driver Exported Macros

9.6.1 SSI_SET_BIT_CLOCK_FREQ()

Structure

Please refer to the definition of *SSI_SET_BIT_CLOCK_FREQ(sys_freq, bit_clk_freq)* in Ssi_Interface.h

Description

Can be used for filling the SsiClockConfig_t structure with values for the targeted SSI clock frequency. It only fills the ssiDIV2, ssiPSR, ssiPM fields. Further the ssiDC and ssiWL should be filled accordingly.

Macro Arguments

- sys_freq = system frequency in Hz
- bit_clk_freq = SSI desired bit clock frequency in Hz

Returns

A 32bit value for *SsiClockConfig_t* structure filling with automatic setting for ssiDIV2, ssiPSR, ssiPM

9.6.2 SSI_DEFAULT_CLOCK_CONFIG()

Structure

```
#define SSI_DEFAULT_CLOCK_CONFIG (uint32_t) ( DIV2 | PSR | WL | DC | PM )
```

Description

Default configuration constant for the SSI Clock structure (see section 16.6.8 SSI Transmit and Receive Clock Control Register (STCCR) in the MC1322x reference manual).

For default clock configuration, assign this macro to the SsiClockConfig_t.ssiClockConfigWord instance, then call the Ssi_SetClockConfig() function.

Macro Arguments

None

Returns

A 32bit value for `SSIClockConfig_t` structure filling with default settings:

- DIV2 = 1 Initial divide-by-2 prescaler used
- PSR = 1 Divide-by-8 Prescaler used
- WL = 7 16 Bits/Word

- DC = 1 1 word per frame
- PM = 1 Modulus divider of 1

9.6.3 SSI_DEFAULT_TX_CONFIG()

Structure

```
#define SSI_DEFAULT_TX_CONFIG (uint16_t)(TXBIT0 | TFEN | TFDIR | TXDIR | TSHFD |
TSCKP | TFSI | TFSL | TEFS)
```

Description

Default configuration constant for the SSI TX structure (see section16.6.6 SSI Transmit Configuration Register (SSI_STCR) in the MC1322x reference manual).
For default TX configuration, assign this macro to the SsiTxRxConfig_t.ssiTxRxConfigWord instance, then call the Ssi_SetTxConfig() function.

Macro Arguments

None

Returns

A 32bit value for `SsiTxRxConfig_t` structure filling with default settings:

- TXBIT0 = 1 LSB aligned
- TFEN = 1 Transmit FIFO enabled
- TFDIR = 1 Frame Sync generated internally
- TXDIR = 1 Transmit Clock generated internally
- TSHFD = 0 Data transmitted MSB first
- TSCKP = 1 Data clocked out on falling edge of bit clock.
- TFSI = 1 Transmit frame sync is active low
- TFSL = 0 Transmit frame sync is one-word long
- TEFS = 0 Transmit frame sync initiated as the first bit of data is transmitted

9.6.4 SSI_DEFAULT_RX_CONFIG()

Structure

```
#define SSI_DEFAULT_TX_CONFIG (uint16_t)(RXEXT | RXBIT0 | RFEN | RFDIR | RXDIR |
RSHFD | RSCKP | RFSI | RFSL | REFS)
```

Description

Default configuration constant for the SSI RX structure (see section16.6.6 SSI Receive Configuration Register (SSI_SRCR) in the MC1322x reference manual).

For default RX configuration, assign this macro to the *SsiTxRxConfig_t.ssiTxRxConfigWord* instance, then call the Ssi_SetRxConfig() function.

Macro Arguments

None

Returns

A 32bit value for `SSITxRxConfig_t` structure filling with default settings:

- RXEXT = 0 Sign extension turned off
- RXBIT0 = 1 LSB aligned
- RFEN = 1 Receive FIFO enabled
- RFDIR = 0 Frame Sync is external
- RXDIR = 0 Receive Clock is external
- RSHFD = 0 Data received MSB first
- RSCKP = 1 Data latched on rising edge of bit clock
- RFSI = 1 Receive frame sync is active low
- RFSL = 0 Receive frame sync is one-word long
- REFS = 0 Receive frame sync is initiated one bit before the data is received

9.7 SSI Driver API Function Descriptions

9.7.1 SSI_Init ()

Prototype

```
void SSI_Init(void);
```

Description

The function is called to initialize the RAM memory space for the SSI driver (patch function table and global variable space). If this call is skipped, unpredictable behavior can appear when trying to access SSI driver functions.

CAUTION

This function must be called prior to any other call to the SSI driver interface functions.

Function Parameters

None

Returns

None

9.7.2 SSI_Enable ()

Prototype

```
SsiErr_t SSI_Enable
(
    bool_t enable
);
```

Description

The function is called to enable / disable the SSI module.

There are some tests performed before enabling the SSI module. If the SSI module is already enabled / disabled, the function exits with *gSsiErrAlreadyEnabled_c / gSsiErrAlreadyDisabled_c* value.

If all of these tests have been passed, the function enables the SSI module and exits with *gSsiErrNoError_c* value.

CAUTION

This function shall be called after SSI_Init() prior to any call to other driver functions, and it should be called only once. If it is called more than once, a *gSsiErrAlreadyEnabled_c / gSsiErrAlreadyDisabled_c* will be returned.

9.7.2.1 Function Parameters

Name

enable

In/Out

input

Description

This selects between enable and disable requests.

9.7.2.2 Returns

Type

SsiErr_t

Description

The status of the operation.

Possible Values

- gSsiErrNoError_c

- gSsiErrAlreadyEnabled_c
- gSsiErrAlreadyDisabled_c

9.7.3 SSI_SetConfig ()

Prototype

```
SsiErr_t  SSI_SetConfig
(
    SsiConfig_t* pSsiConfig
);
```

Description

The function is called to set the parameters for the SSI peripheral.

There are some tests performed before configuring the SSI peripheral. If the pointer to the structure containing the configuration parameters is not initialized, the function exits with *gSsiErrNullPointer_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gSsiErrNoError_c* value.

9.7.3.1 Function Parameters

Name

pSSIConfig

In/Out

input

Description

The pointer to a structure containing the configuration parameters.

9.7.3.2 Returns

Type

SsiErr_t

Description

The status of the operation.

Possible Values

- gSsiErrNoError_c

- gSsiErrNullPointer_c

9.7.4 SSI_SetClockConfig ()

Prototype

```
SsiErr_t SSI_SetClockConfig
(
    SsiClockConfig_t* pSsiClockConfig
);
```

Description

The function is called to set the parameters for the SSI clock.

There are some tests performed before getting the configuration. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gSsiErrNullPointer_c* value.

If the network mode was previously selected (SSISetConfig function) and the SSISetClockConfig.bit.ssiDC is 0, the function exits with gSsiErrInvalidConfiguration_c value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gSsiErrNoError_c* value.

9.7.4.1 Function Parameters

Name

pSSIClockConfig

In/Out

input

Description

The pointer to a structure containing the configuration parameters for the SSI clock.

9.7.4.2 Returns

Type

SsiErr_t

Description

Function execution status.

Possible Values

- gSsiErrNoError_c
- gSsiErrNullPointer_c
- gSsiErrInvalidConfiguration_c

9.7.5 SSI_SetTxRxConfig ()

Prototype

```
SsiErr_t SSI_SetTxRxConfig
(
    SsiTxRxConfig_t* pSsiTxRxConfig,
    SsiOpType_t opType
);
```

Description

The function is called to set the parameters for the SSI TX/RX operation.

The selection is made trough the opType parameter which should specify either gSsiOpTypeTx_c or gSsiOpTypeRx_c. In case if other values are provided for this parameter, the *gSsiErrInvalidOpType_c* error is returned.

There are some tests performed before getting the configuration. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gSsiErrNullPointer_c* value. If the normal mode was previously selected (*SsiConfig_t.ssiMode = gSsiNormalMode_c*), the *ssiDC* was set to 0 (*SsIClockConfig_t.bit.ssiDC = 0*) and the frame sync generation is requested as word-length (*SSITxRxConfig_t.bit.ssiFSL = 0*) the function exits with *gSsiErrInvalidConfiguration_c* value as the SSI hardware can generate only bit-clock-length frame sync in this case.

If another TX operation is pending and the Tx configuration was selected, the function returns *gSsiBusy_c*.

If another RX operation is pending and the Rx configuration was selected, the function returns *gSsiBusy_c*.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gSsiErrNoError_c* value.

9.7.5.1 Function Parameters

Name

pSsiTxRxConfig

In/Out

input

Description

The pointer to a structure containing the configuration parameters for the RX/TX operation.

Name

opType

In/Out

input

Description

Selects between Tx or Rx configuration.

9.7.5.2 Returns

Type

SsiErr_t

Description

Function execution status.

Possible Values

- gSsiErrNoError_c
- gSsiErrNullPointer_c
- gSsiErrInvalidConfiguration_c
- gSsiErrInvalidOpType_c
- gSsiBusy_c

9.7.6 SSI_SetTxCallback ()

Prototype

```
SsiErr_t SSI_SetTxCallback
(
    SSITxCallback_t ssiTxCallback
);
```

Description

The function is called to set the callback function for the SSI write operation.

The function is setting the internal driver callback pointer and returns the *gSsiErrNoError_c* value. If a NULL pointer is provided as function parameter, the effect is the canceling of the previously set callback function.

CAUTION

The callback function will be called from interrupt context and should perform accordingly. Execution time should be reduced to minimum and the global variables modified from this function should be declared as volatile. Other driver functions should not be called from the callback function.

9.7.6.1 Function Parameters

Name

ssiTxCallback

In/Out

input

Description

The pointer to the callback function.

9.7.6.2 Returns

Type

SsiErr_t

Description

Function execution status.

Possible Values

- gSsiErrNoError_c

9.7.7 SSI_SetRxCallback ()

Prototype

```
SsiErr_t SSI_SetRxCallback
(
    SSIRxCallback_t ssiRxCallback
);
```

Description

The function is called to set the callback function for the SSI read operation.

The function is setting the internal driver callback pointer and returns the *gSsiErrNoError_c* value. If a NULL pointer is provided as function parameter, the effect is the canceling of the previously set callback function.

CAUTION

The callback function will be called from interrupt context and should perform accordingly. Execution time should be reduced to minimum and the global variables modified from this function should be declared as volatile. Other driver functions should not be called from the callback function.

9.7.7.1 Function Parameters

Name

ssiRxCallback

In/Out

input

Description

The pointer to the callback function.

9.7.7.2 Returns

Type

SsiErr_t

Description

Function execution status.

Possible Values

- gSsiErrNoError_c

9.7.8 SSI_SetContinuousRxCallback ()

Prototype

```
SsiErr_t SSI_SetContinuousRxCallback
(
    SSIContinuousRxCallback_t ssiContinuousRxCallback
);
```

Description

The function is setting the internal driver callback pointer and returns the *gSsiErrNoError_c* value. If a NULL pointer can be specified with the result of canceling the previously set callback function.

CAUTION

The callback function will be called from interrupt context and should perform accordingly. Execution time should be reduced to minimum and the global variables modified from this function should be declared as volatile. Other driver functions should not be called from the callback function._

9.7.8.1 Function Parameters

Name

ssiContinuousRxCallback

In/Out

input

Description

The pointer to a structure containing the pointer to the callback function.

9.7.8.2 Returns

Type

SsiErr_t

Description

Function execution status.

Possible Values

- gSsiErrNoError_c

9.7.9 SSI_TxData ()

Prototype

```
SsiErr_t SSI_TxData
(
    void* pData,
    uint8_t length,
    SsiWordSize_t wordSize,
    uint32_t timeSlot
);
```

Description

This function is used to send a sequence of words on the SSI bus. The function attempts to send *length* words which are read from *pData* (*wordSize* aligned).

There are some tests performed before sending data on the SSI bus. If the pointer to memory containing the data is not initialized, the function exits with the *gSsiErrNullPointer_c* value. If the data length is provided as 0, the function exists with the *gSsiErrDataLength_c* error. If an invalid *wordSize* is specified (outside the *SsiWordSize_t* type), the function exits with *gSsiErrWordSize_c* value.

If another TX operation is in pending, the function exists with the *gSsiErrBusy_c* error code.

If all of these tests have been passed and all data bytes were sent the function returns the *gSsiErrNoError_c* value.

Further, the function schedules a SSI transmission. If the TX FIFO is activated, the function will manage its usage in order to maximize the data output rate.

If the provided package cannot be transmitted in one step, the transmission will be completed by the SSI_ISR() routine.

When the transmission is completed, the driver will call the writeCallback function if it was configured by the user.

CAUTION

This function will not buffer the provided data internally.

9.7.9.1 Function Parameters

Name

pData

In/Out

input

Description

The pointer to a memory location containing the data to be transmitted.

Name

length

In/Out

input

Description

data length

Name

wordSize

In/Out

input

Description

The word size for the provided buffer elements.

Name

timeSlot

In/Out

input

Description

The time slot for transmission in network mode.

9.7.9.2 Returns

Type

SsiErr_t

Description

The status of the operation.

Possible Values

- gSsiErrNoError_c
- gSsiErrNullPointer_c
- gSsiErrDataLength_c
- gSsiErrWordSize_c
- gSsiErrBusy_c

9.7.10 SSI_RxData ()

Prototype

```
SsiErr_t SSI_RxData
(
    void* pData,
    uint8_t length,
```

```
        SsiWordSize_t wordSize,
        uint32_t timeSlot
    );
```

Description

This function is used to receive a sequence of bytes from the SSI bus. The function attempts to receive *length* bytes which will be stored in the location pointed by *pData*.

There are some tests performed before receiving data from the SSI bus. If the pointer to memory containing the data is not initialized, the function exits with *gSsiErrNullPointer_c* value. If the data length is provided as 0, the function exists with the *gSsiErrDataLength_c* error. If an invalid wordSize is specified (outside the *SsiWordSize_t* type), the function exits with *gSsiErrWordSize_c* value.

If another RX operation is in pending, the function exists with the *gSsiErrBusy_c* error code.

If all of these tests have been passed the function returns the *gSsiErrNoError_c* value.

Further, the function schedules a SSI reception. If the RX FIFO is activated, the function will manage its usage in order to optimize the data input rate. The receive operation will be completed by the SSI_ISR() routine.

9.7.10.1 Function Parameters

Name

pData

In/Out

input

Description

The pointer to a memory location where data will be placed.

Name

length

In/Out

input

Description

data length

Name

wordSize

In/Out

input

Description

The word size for the provided buffer elements.

Name

timeSlot

In/Out

input

Description

The time slot for receive operation in network mode.

9.7.10.2 Returns

Type

SsiErr_t

Description

The status of the operation.

Possible Values

- gSsiErrNoError_c
- gSsiErrNullPointer_c
- gSsiErrDataLength_c
- gSsiErrWordSize_c
- gSsiErrBusy_c

9.7.11 SSI_StartContinuousRx ()

Prototype

```
SsiErr_t SSI_StartContinuousRx
(
    void * pData,
    uint8_t length,
    SsiWordSize_t wordSize,
    uint32_t timeSlot
);
```

Description

This function is used to start a continuous receive sequence. The SSI driver will use the hardware RX FIFO (if enabled) to store the received data and inform the application each time an amount of *length* words are available. The application callback function will be called with a pointer to the received data when the number of *length* words is available in the RX FIFO. Also, the application callback function will return a new data length and data pointer for the next indication and 0 (or NULL) if the process should be stopped. If the new provided data length is not consistent with the FIFO activation (1-8 with the RX FIFO active and 1 without RX FIFO), the RX process is stopped.

If the RX FIFO is not activated, this function will report each received word (*length* only accepted as 1).

There are some tests performed before receiving data on the SSI bus. If the data length is provided outside the requested range (1-8 with the RX FIFO active and 1 without RX FIFO), the function exists with the *gSsiErrDataLength_c* error. If an invalid wordSize is specified (outside the *SsiWordSize_t* type), the function exits with *gSsiErrWordSize_c* value.

If there is no callback function configured for the continuous RX process, the function exists with *gSsiNoCallback_c* error. If another RX operation is in pending, the function exists with the *gSsiErrBusy_c* error code.

If all of these tests have been passed the function returns the *gSsiErrNoError_c* value.

Further, the function schedules a SSI reception.

9.7.11.1 Function Parameters

Name

pData

In/Out

input

Description

The pointer to a memory location where data will be placed.

Name

length

In/Out

input

Description

data length

Range

- 1-8 with RX FIFO activated
- 1 with RX FIFO deactivated

Name

wordSize

In/Out

input

Description

The word size for the buffer elements which will be reported at the end of the RX process.

Name

timeSlot

In/Out

input

Description

The time slot for receive operation in network mode.

9.7.11.2 Returns

Type

SsiErr_t

Description

The status of the operation.

Possible Values

- gSsiErrNoError_c
- gSsiErrDataLength_c
- gSsiErrWordSize_c
- gSsiNoCallback_c
- gSsiErrBusy_c

9.7.12 SSI_Abort ()

Prototype

```
SsiErr_t SSI_Abort
(
    SsiOpType_t opType
);
```

Description

This function is used to abort a TX/RX sequence. The selection is made trough the *opType* parameter which should specify either *gSsiOpTypeTx_c* or *gSsiOpTypeRx_c*. In case if other values are provided for this parameter, the *gSsiErrInvalidOpType_c* error is returned.

For TX abort, it will cancel the current transmission of the data previously requested with SSI_TxData() function by turning off the HW SSI transmitter. The BUS transmission will stop after the current frame. The TX callback function is not called in this case. If there is no transmission currently pending, the function exits with *gSsiErrNotBusy_c* value.

For Rx abort, it will cancel the current reception of the data previously requested with SSI_RxData() function. The RX callback function is not called in this case. If there is no data reception currently pending, the function exits with *gSsiErrNotBusy_c* value.

9.7.12.1 Function Parameters

Name

opType

In/Out

input

Description

Selects between Tx or Rx abort.

9.7.12.2 Returns

Type

SsiErr_t

Description

The status of the operation.

Possible Values

- gSsiErrNoError_c
- gSsiErrNotBusy_c
- gSsiErrInvalidOpType_c

9.7.13 SSI_ISR ()

Prototype

```
void SSI_ISR(void);
```

Description

The function will be set as interrupt routine for the SSI peripheral. This will be done using the interface provided by the ITC driver.

This function should be registered as SSI interrupt service routine through the ITC provided AssignHandler service:

```
Int_AssignHandler(gSsiInt_c, SSI_ISR);
```

Further the interrupts should be enabled at the ITC level:

```
Itc_EnableInterrupt(gSsiInt_c);
```

And the interrupts globally enabled:

```
Int_EnableAll();
```

Chapter 10
Analog to Digital Converter (ADC) Driver

10.1 Overview

The purpose of the ADC module is to manage the interface to external sensors. The ADC module will scan multiple channels and control the warm up of the analog portions of the A to D system. The ADC module can be set to interrupt the ARM based on programmed compare values that can be set for up to 8 channels. The primary A to D can be programmed to convert and monitor 9 channels (8 GP-ADC pins plus battery). The secondary A to D can be programmed to convert and monitor the 8 GP-ADC pins.

10.1.1 Hardware Module

The module has configuration registers that must be programmed to set the usage. For detailed information on the ADC module, see the *MC1322x Reference Manual* (MC1322xRM).

The ADC has the following features:

- 12 bit resolution. Effective number of bits 10.5
- Input voltage range: Vref_high, Vref_low.
- Maximum input of 3.6v
- Minimum input of 0.0v
- Integral non-linearity 0.25 bit.
- Differential non-linearity 0.5 bit
- Maximum current: 4ma
- Sample rate maximum 20us
- 8 bit prescaler to provide the time base for the 32 bit timers.
- Two independent channels, each with a 32 bit timer.
- Primary ADC has 9 channels. 8 GPIO plus battery.
- Secondary ADC has 8 channels. 8 GPIO.
- Active channels for each A to D are programmable.
- A maximum of 8 active monitors can generate a IRQ trigger.
- A 8 deep FIFO for recording data.
- IRQ’s can be generated by the channel compare values, FIFO status, 32 bit timers.

10.1.2 Driver Functionality

The driver allows the developer to use the ADC functions easily. It allows the developer to:

- Turn the ADC module ON/OFF
- Set the bus clock divide ratio for the ADC timers
- Set the ADC clock divider the converter clock
- Set the ON time for the ADC
- Set the conversion time
- Set the ADC module for automatic or manual operation
- Select the channels that will be converted during each converter sequence
- Select whether an interrupt should be generated after completion of each sequence of conversions
- Turn the timer for the sequence ON/OFF

10.2 Include Files

Table 10-1 shows the ADC driver files that must be included in the application C-files in order to have access to the driver function calls.

Table 10-1. ADC Driver Include Files

Include File Name	Description
ADC_Interface.h	Global header file that defines the public data types and enumerates the public functions prototypes.

10.3 ADC Driver API Functions

Table 10-2 shows the available API functions for the ADC Driver.

Table 10-2. ADC Driver API Function List

ADC Driver API Function Name	Description
Adc_Init()	The function is called to initialize the ADC driver.
Adc_Reset ()	The function is called to reset the ADC hardware.
Adc_SetConfig ()	The function is called to set the parameters for the ADC peripheral.
Adc_SetConvCtrl()	The function is called to set the conversion control of the ADC module.
Adc_SetCompCtrl()	The function is called to set the ADC module to monitor ADC pins for changes.
Adc_SetFifoCtrl()	The function is called to set the threshold of the ADC FIFO.
Adc_GetFifoStatus()	The function is called to get the FIFO status and the FIFO fill level.
Adc_ReadFifoData ()	The function is called to read data from the ADC FIFO.
Adc_StartManualConv()	The function is called to configure the manual conversion mode of the ADC.
Adc_ManualRead()	The function is called to get the conversion result value after manual mode was set.
Adc_SetCallback()	The function is called to set the callbacks for ADC events.
Adc_Isr()	The interrupt service routine for the ADC peripheral.

10.4 Driver Exported Constants

10.4.1 AdcErr_t

Constant Structure

```
typedef enum
{
    gAdcErrNoError_c = 0,
    gAdcErrWrongParameter,
    gAdcErrNullPointer_c,
    gAdcErrModuleBusy_c,
    gAdcInvalidOp_c,
    gAdcErrMax_c
} AdcErr_t;
```

Description

Specifies the possible return values for the ADC driver API functions.

Constant Values

The constant values are self explanatory.

10.4.2 AdcChannel_t

Constant Structure

```
typedef enum
{
    gAdcChan0_c = 0,
    gAdcChan1_c,
    gAdcChan2_c,
    gAdcChan3_c,
    gAdcChan4_c,
    gAdcChan5_c,
    gAdcChan6_c,
    gAdcChan7_c,
    gAdcBatt_c,
    gAdcMaxChan_c
} AdcChannel_t;
```

Description

Specifies the possible values of the channels.

Constant Values

The constant values are self explanatory.

10.4.3 AdcModule_t

Constant Structure

```
typedef enum
{
    gAdcPrimary_c = 0,
    gAdcSecondary_c,
    gAdcMaxModule_c
} AdcModule_t;
```

Description

Specifies the possible values for ADC module.

Constant Values

The constant values are self explanatory.

10.4.4 AdcModuleStatus_t

Constant Structure

```
typedef enum
{
    gAdcModuleOff_c = 0,
    gAdcModuleOn_c
} AdcModuleStatus_t;
```

Description

Specifies the possible values for the status of ADC.

Constant Values

The constant values are self explanatory.

10.4.5 AdcMode_t

Constant Structure

```
typedef enum
{
    gAdcAutoControl_c = 0,
    gAdcManualControl_c
} AdcMode_t;
```

Description

Specifies the possible values of ADC working mode.

Constant Values

The constant values are self explanatory.

10.4.6 AdcCompType_t

Constant Structure

```
typedef enum
{
    gAdcCompGrater_c = 0,
    gAdcCompLess_c
} AdcCompType_t;
```

Description

Specifies the possible values for compare type.

Constant Values

The constant values are self explanatory.

10.4.7 AdcSeqMode_t

Constant Structure

```
typedef enum
{
    gAdcSeqOnConvTime_c = 0,
    gAdcSeqOnTmrEv_c
} AdcSeqMode_t;
```

Description

Specifies the possible values for sequencer mode.

Constant Values

The constant values are self explanatory.

10.4.8 AdcChanStatus_t

Constant Structure

```
typedef enum
{
    gAdcInactiveChan_c = 0,
    gAdcActiveChan_c
} AdcChanStatus_t;
```

Description

Specifies the possible values for ADC channels state.

Constant Values

The constant values are self explanatory.

10.4.9 AdcRefVoltage_t

Constant Structure

```
typedef enum
{
    gAdcBatteryRefVoltage_c = 0,
    gAdcExtRefVoltage_c
} AdcRefVoltage_t;
```

Description

Specifies the possible values for ADC reference voltage.

Constant Values

The constant values are self explanatory.

10.4.10 AdcFifoStatus_t

Constant Structure

```
typedef enum
{
    gAdcFifoEmpty_c = 0,
    gAdcFifoFull_c,
    gAdcFifoFilled_c
} AdcFifoStatus_t;
```

Description

Specifies the possible values for ADC FIFO status.

Constant Values

The constant values are self explanatory.

10.4.11 AdcEvent_t

Constant Structure

```
typedef enum
{
    gAdcCompEvent_c = 0,
    gAdcSeq1event_c,
    gAdcSeq2event_c,
    gAdcFifoEvent_c,
    gAdcMaxEvent_c
} AdcEvent_t;
```

Description

Specifies the possible events of ADC module.

Constant Values

The constant values are self explanatory.

10.5 Exported Structures and Data Types

10.5.1 AdcConfig_t

Structure

```
typedef struct
{
    uint8_t          adcPrescaler;
    uint8_t          adcDivider;
    uint16_t         adcOnPeriod;
    uint16_t         adcConvPeriod;
    bool_t           adcCompIrqEn;
    bool_t           adcFifoIrqEn;
    AdcMode_t        adcMode;
} AdcConfig_t;
```

Description

This structure configures the ADC module.

10.5.1.1 Structure Elements

adcPrescaler	Specifies the bus clock divide ratio; the clock that drives the timers and sequencers is determined by this divider (prescale clock).
adcDivider	Specifies the clock divider that provides the clock to the analog portion if the ADC (this should be set to a value that provides a 300KHz clock).
adcOnPeriod	Specifies the turn on time for the analog (typical value is 8us; maximum value is 10us). The value is a function of the prescale clock.
adcConvPeriod	Specifies the ADC convert period (should not be set to a value of less than 20 us). The value is a function of the prescale clock.
adcCompIrqEn	Specifies if the compare interrupt should be generated or not.

adcFifoIrqEn	Specifies if the FIFO threshold interrupt should be generated or not.
adcMode	Specifies the desired working mode - automatically control or manual control.

10.5.2 AdcConvCtrl_t

Structure

```
typedef struct
{
    bool_t           adcTmrOn;
    bool_t           adcSeqIrqEn;
    uint16_t         adcChannels;
    uint32_t         adcTmBtwSamples;
    AdcSeqMode_t     adcSeqMode;
    AdcRefVoltage_t  adcRefVoltage;
} AdcConvCtrl_t;
```

Description

This structrure configures conversion sequence.

10.5.2.1 Structure Elements

adcTmrOn	boolean value that specifies if the is active or not.
adcSeqIrqEn	specifies whether an interrupt should be generated after completion of each sequence of conversions.
adcChannels	the value specifies active channels.
adcTmBtwSamples	specifies the time period between samples
adcSeqMode	specifies the sequencing mode - sequence to the next channel based on timer event or based on convert time.
adcRefVoltage	specifies the reference voltage source (battery/external) for ADC module.

sequence of conversions.

10.5.3 AdcCompCtrl_t

Structure

```
typedef struct
{
    AdcChannel_t     adcChannel;
    AdcCompType_t    adcCompType;
    uint16_t         adcCompVal;
} AdcCompCtrl_t;
```

Description

This structrure configures conversion comparison.

10.5.3.1 Structure Elements

adcChannel	the value specifies the channel wherefore the configuration is applied.
adcCompType	specifies the type of comparison (grater/less).
adcCompVal	specifies the value for the comparison.

10.5.4 AdcFifoData_t

Structure

```
typedef struct
{
    AdcChannel_t     adcChannel;
    uint16_t         adcValue;
} AdcFifoData_t;
```

Description

This structrure is used when data is read from the FIFO.

10.5.4.1 Structure Elements

adcChannel	the value indicates which GP - ADC pin the sample was taken on.
adcValue	is a 12 bit value that is the sample stored in FIFO.

10.5.5 AdcEvCallback_t

Structure

```
typedef void (*AdcEvCallback_t)(void);
```

Description

The data type registers the callbacks for ADC event.

10.6 Driver Exported Macros

10.6.1 Adc_DefaultConfig()

Fills the structure passed as parameter with the default configuration values (1Mhz prescale clock, 300 KHz ADC analog clock, 8 us on time, 40 us conversion time, auto working mode, compare interrupt disabled, FIFO interrupt disabled).

10.6.1.1 Macro Arguments

Type

AdcConfig_t

Name

adcConfig

In/Out

output

Description

The structure that will be filled with default configuration parameters.

Type

uint32_t

Name

oscVal

In/Out

input

Description

The value of the oscillator's clock.

Returns

None

10.6.2 Adc_TurnOn()

Turns on the ADC module.

10.6.3 Adc_TurnOff()

Turns OFF the ADC module.

10.6.4 Adc_FifoData()

Read data from FIFO.

10.6.4.1 Macro Arguments

None

Returns

16 bit data from FIFO (first 12 bits represent the value sampled, next 4 bits represent which channel was sampled)

10.6.5 Adc_FifoLevel()

Returns the FIFO level.

10.6.5.1 Macro Arguments

None

Returns

The current FIFO level.

10.6.6 Adc_ChnTriggered()

Reads the channels that were triggered by a compare value.

10.6.6.1 Macro Arguments

None

Returns

The channels that were triggered by a compare value.

10.7 ADC Driver API Function Descriptions

10.7.1 Adc_Init ()

Prototype

```
void Adc_Init (void);
```

Description

The function is called to initialize the ADC driver. Actually this is not a interface function. It will be called only once. If this call is skipped, unpredictable behavior can appear when trying to access ADC driver functions.

CAUTION

This function must be called prior to any other call to the ADC driver interface functions!!!

10.7.2 Adc_SetConfig ()

Prototype

```
AdcErr_t Adc_SetConfig
(
    const AdcConfig_t *adcConfig
);
```

Description

The function is called to set the parameters for the ADC peripheral.

There are some tests performed before configuring the ADC peripheral. If the pointer to the structure containing the configuration parameters is not initialized, the function exits with *gAdcErrNullPointer_c* value. If the module is busy (the module was programmed to sample data from ADC pins and was turned ON), the function exits with *gAdcErrModuleBusy_c* value.

If all of these tests have been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gAdcErrNoError_c* value. Typical configuration parameters are:

- adcOnPeriod = 8 us (maximum 10us)
- adcConvPeriod = 40 us for auto mode
- adcConvPeriod = 20 us for manual mode

10.7.2.1 Function Parameters

Name

adcConfig

In/Out

input

Description

The pointer to a structure containing the configuration parameters.

10.7.2.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrNullPointer_c
- gAdcErrModuleBusy_c

10.7.3 Adc_SetConvCtrl ()

Prototype

```
AdcErr_t Adc_SetConvCtrl
(
    AdcModule_t adcModule,
    const AdcConvCtrl_t *adcConvCtrl
);
```

Description

The function is called to set the conversion control of the ADC module specified as parameter .

There are some tests performed before setting the conversion configuration. If the *adcModule* value is grater or equal than *gAdcMaxModule_c* the function exits with *gAdcErrWrongParameter_c* value. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gAdcErrNullPointer_c* value.

If all of these tests has been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gAdcErrNoError_c* value.

10.7.3.1 Function Parameters

Name

adcModule

In/Out

input

Description

The instance of the ADC module (primary/secondary ADC module) that will be configured.

Name

adcConvCtrl

In/Out

input

Description

The pointer to a structure that contains the conversion control parameters.

10.7.3.2 Returns

Type

AdcErr_t

Description

Function execution status.

Possible Values

- gAdcErrNoError_c
- gAdcErrWrongParameter_c
- gAdcErrNullPointer_c

10.7.4 Adc_SetCompCtrl ()

Prototype

```
AdcErr_t Adc_SetCompCtrl
(
    const AdcCompCtrl_t *adcCompCtrl
);
```

Description

The function is called to set the ADC module to monitor ADC pins for changes.

There are some tests performed before setting the comparison configuration. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gAdcErrNullPointer_c* value.

If the test has been passed, the function sets the peripheral hardware registers according with the values received in the configuration structure and returns the *gAdcErrNoError_c* value.

10.7.4.1 Function Parameters

Name

adcCompCtrl

In/Out

input

Description

The pointer to a structure containing the comparison configuration parameters.

10.7.4.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrNullPointer_c

10.7.5 Adc_SetFifoCtrl ()

Prototype

```
AdcErr_t Adc_SetFifoCtrl
(
    uint8_t adcFifoThreshold
);
```

Description

This function is used to set the threshold of the ADC FIFO.

There are some tests performed before setting the FIFO threshold. If the value of *adcFifoThreshold* exceeds 7 (maxim value of the threshold), the function exits with *gAdcErrWrongParameter_c* value.

If this test has been passed the function returns the *gAdcErrNoError_c* value.

10.7.5.1 Function Parameters

Name

adcFifoThreshold

In/Out

input

Description

The value of the FIFO treshold.

10.7.5.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrWrongParameter_c

10.7.6 Adc_ReadFifoData ()

Prototype

```
AdcErr_t Adc_ReadFifoData
(
    AdcFifoData_t *adcFifoData
);
```

Description

This function is used to read data from the ADC FIFO.

There are some tests performed before reading data from FIFO. If the pointer to the structure where the configuration parameters shall be placed is not initialized, the function exits with *gAdcErrNullPointer_c* value.

If this test have been passed the function returns the *gAdcErrNoError_c* value.

10.7.6.1 Function Parameters

Name

adcFifoData

In/Out

output

Description

The pointer to a structure where FIFO data will be placed.

10.7.6.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrNullPointer_c

10.7.7 Adc_GetFifoStatus ()

Prototype

```
AdcErr_t Adc_GetFifoStatus
(
    AdcFifoStatus_t *fifoStatus,
    uint8_t *adcFifoLevel
);
```

Description

This function is used to get the FIFO status (empty/full/filled) and the fill level.

There are some tests performed before getting FIFO status. If one of the pointers to the memory locations where the values of FIFO level and status will be placed is NULL, the function exits with *gAdcErrNullPointer_c* value.

If this test have been the function returns the *gAdcErrNoError_c* value.

10.7.7.1 Function Parameters

Name

fifoStatus

In/Out

output

Description

The pointer to a memory location where the current FIFO status value will be placed.

Name

adcFifoLevel

In/Out

output

Description

The pointer to a memory location where the current FIFO level value will be placed.

10.7.7.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrNullPointer_c

10.7.8 Adc_StartManualConv ()

Prototype

```
AdcErr_t Adc_StartManualConv
(
    AdcModule_t adcModule,
    AdcChannel_t adcChannel
);
```

Description

This function is used to configure the manual conversion mode of the ADC.

There are some tests performed before configuring the manual conversion mode. If the *adcModule* value is grater or equal than *gAdcMaxModule_c* the function exits with *gAdcErrWrongParameter_c* value. If the current ADC mode is not manual conversion mode, the function exits with *gAdcInvalidOp_c* value. If the function is called before reading the result of the previous started conversion, the function returns *gAdcErrModuleBusy_c* error.

If the test has been passed the function start AD module desired and returns the *gAdcErrNoError_c* value.

10.7.8.1 Function Parameters

Name

adcModule

In/Out

input

Description

The instance of the ADC module (primary/secondary ADC module) that will be configured for manual conversion.

Name

adcChannel

In/Out

input

Description

The channel that will be sampled in manual mode.

10.7.8.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrWrongParameter_c
- gAdcInvalidOp_c
- gAdcErrModuleBusy_c

10.7.9 Adc_ManualRead ()

Prototype

```
AdcErr_t Adc_ManualRead
(
    AdcModule_t adcModule,
    uint16_t *adcConvResult
);
```

Description

This function is used to get the conversion result value after manual mode was set and configured.

There are some tests performed before the proper reading of the conversion result. If the *adcModule* value is grater or equal than *gAdcMaxModule_c* the function exits with *gAdcErrWrongParameter_c* value.

If the pointer to the structure where the configuration parameters are placed is not initialized, the function exits with *gAdcErrNullPointer_c* value. If there was not initiated a manual conversion before, the function exits with *gAdcInvalidOp_c* error.

If the test had been passed the function returns the *gAdcErrNoError_c* value.

10.7.9.1 Function Parameters

Name

adcModule

In/Out

input

Description

The instance of the ADC module (primary/secondary ADC module) that will be configured for manual conversion.

Name

adcConvResult

In/Out

input

Description

The pointer to a memory location where the result of the manual conversion is placed.

10.7.9.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrWrongParameter_c
- gAdcErrNullPointer_c
- gAdcInvalidOp_c

10.7.10 Adc_SetCallback ()

Prototype

```
AdcErr_t Adc_SetCallback
(
    AdcEvent_t adcEvent,
    AdcEvCallback_t pCallbackFunction
);
```

Description

This function is used to set the callbacks for ADC events.

There are some tests performed before configuring the manual conversion mode. If the *adcEvent* value is grater or equal than *gAdcMaxEvent_c* the function exits with *gAdcErrWrongParameter_c* value.

If the test had been passed the function returns the *gAdcErrNoError_c* value.

10.7.10.1 Function Parameters

Name

adcEvent

In/Out

input

Description

ADC event

Name

pCallbackFunction

In/Out

input

Description

The pointer to a memory location where the structure containing the event callback is placed.

10.7.10.2 Returns

Type

AdcErr_t

Description

The status of the operation.

Possible Values

- gAdcErrNoError_c
- gAdcErrWrongParameter

10.7.11 Adc_Reset ()

Prototype

```
void Adc_Reset (void);
```

Description

This function is used to software reset the ADC module.

10.7.12 Adc_Isr()

Prototype

```
void Adc_Isr (void);
```

Description

The function will be set as interrupt routine for the ADC peripheral. This will be done using the interface provided by the ITC driver.

Chapter 11
Serial Peripheral Interface (SPI) Driver

11.1 Overview

The Serial Peripheral Interface (SPI) is a high-speed synchronous serial input/output port that allows a serial bit stream of programmed length (1 to 32 bits) to be shifted into and out of the device at a programmed bit-transfer rate in a master or slave mode. The SPI is normally used for communications between the CPU and external devices with SPI support.

11.1.1 Hardware Module

The module has configuration registers that must be programmed to set the usage. For detailed information on the SPI module, see the *MC1322x Reference Manual* (MC1322xRM).

The SPI has the following features:

- Master or slave mode operation
- 4 Bytes of data buffer
- Programmable transmit bit rate
- Serial clock phase and polarity options
- Can handle both 3 and 4 wire connections
- SPI transaction can be polled or interrupt driven
- 32-bit wide IP bus interface to CPU (with 32, 16 and 8 bit IP bus transactions allowed)
- Low Power (SPI Master uses gated clocks. SPI Slave clock derived completely from SPI_SCK)

11.1.2 Driver Functionality

The driver allows the developer to use the SPI functions easily. It allows the developer to:

- Transmit data through SPI bus
- Receive data from SPI bus
- Set the SPI clock

11.2 Include Files

Table 11-1 shows the SPI driver files that must be included in the application C-files in order to have access to the driver function calls.

Table 11-1. SPI Driver Include Files

Include File Name	Description
SPI_Interface.h	Global header file that defines the public data types and enumerates the public functions prototypes.

11.3 SPI Driver API Functions

Table 11-2 shows the available API functions for the SPI Driver.

Table 11-2. SPI Driver API Function List

SPI Driver API Function Name	Description
SPI_Open()	The function is called to open SPI port.
SPI_Close ()	The function is called to close SPI port.
SPI_SetConfig ()	The function is called to configure the SPI port.
SPI_GetConfig ()	The function is called to get the configuration of SPI port.
SPI_WriteSync ()	The function is called to transmit a word through SPI.
SPI_ReadSync ()	The function is called to receive a word through SPI.
SPI_SetTxAsync ()	The function is called to put data to be transmitted into Tx SPI register.
SPI_GetRxAsync ()	The function is called to get last received data from Rx SPI register.
SPI_StartAsync ()	The function is called to start tx/rx operation.
SPI_Abort ()	The function is called to abort current operation.
SPI_SetCallback ()	The function is called to set callback function which will be called from interrupt service routine.
SPI_GetStatus()	The function is called to return the status of the SPI port.
SPI_ISR ()	The interrupt service routine for the SPI peripheral.

11.4 Driver Exported Constants

11.4.1 spiErr_t

Constant Structure

```
typedef enum
{
    gSpiErrNoError_c,
    gSpiErrAlreadyOpen_c,
    gSpiErrPortClosed_c,
    gSpiErrAbortError_c,
    gSpiErrAsyncOperationPending_c,
    gSpiErrWrongConfig_c,
    gSpiErrWrongCallbackFunc_c,
```

```
    gSpiErrNullPointer_c,
    gSpiErrMax_c
} spiErr_t;
```

Description

Specifies the possible return values for the SPI driver API functions.

Constant Values

The constant values are self explanatory.

11.4.2 spiStatus_t

Constant Structure

```
typedef enum
{
    gSpiStatusClosed_c,
    gSpiStatusIdle_c,
    gSpiStatusSyncOperationPending_c,
    gSpiStatusAsyncOperationPending_c,
    gSpiStatusError_c,
    gSpiStatusMax_c
} spiStatus_t;
```

Description

Specifies the possible return values for the SPI driver API functions.

Constant Values

The constant values are self explanatory.

11.5 Exported Structures and Data Types

11.5.1 spiConfig_t

Structure

```
typedef struct
{
    union {
        uint32_t  Word;
        struct {
            uint32_t  DataCount:7;
            uint32_t  reserved0:1;
            uint32_t  ClockCount:8;
            uint32_t  reserved1:16;
        } Bits;
    } ClkCtrl;
```

```
        union {
            uint32_t  Word;
            struct {
                uint32_t  SsSetup:2;
                uint32_t  SsDelay:2;
                uint32_t  SdoInactive:2;
                uint32_t  reserved0:2;
                uint32_t  ClockPol:1;
                uint32_t  ClockPhase:1;
                uint32_t  MisoPhase:1;
                uint32_t  reserved1:1;
                uint32_t  ClockFreq:3;
                uint32_t  reserved2:1;
                uint32_t  Mode:1;
                uint32_t  S3Wire:1;
                uint32_t  reserved3:14;
            } Bits;
        } Setup;
    } spiConfig_t;
```

Description

Configures the SPI module.

Structure Elements

The structure element names are self explanatory.

11.5.2 spiCallback_t

Structure

```
typedef void (*spiCallback_t)(uint32_t Data);
```

Description

This data type registers the callbacks for SPI events.

11.6 SPI Driver API Function Descriptions

11.6.1 SPI_Open ()

Prototype

```
spiErr_t SPI_Open (void);
```

Description

The function is called to open SPI port.

11.6.1.1 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrAlreadyOpen_c
- gSpiErrNoError_c

11.6.2 SPI_Close ()

Prototype

```
spiErr_t SPI_Close (void);
```

Description

The function is called to close SPI port.

11.6.2.1 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrNoError_c

11.6.3 SPI_SetConfig ()

Prototype

```
spiErr_t SPI_SetConfig
(
    spiConfig_t *pConfig
);
```

Description

The function is called to configure a SPI port with information contained in the structure pointed by pConfig.

SPI_SetConfig() does not write to the SPI clock control register. Rather, it only saves the settings to an internal global variable. This variable is then used for each interrupt-driven transaction that is initiated by calling SPI_StartAsync().

11.6.3.1 Function Parameters

Name

pConfig

In/Out

input

Description

The pointer to a configuration structure.

11.6.3.2 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrWrongConfig_c
- gSpiErrNullPointer_c
- gSpiErrNoError_c

11.6.4 SPI_GetConfig ()

Prototype

```
spiErr_t SPI_GetConfig (
    spiConfig_t *pConfig
);
```

Description

The function is called to put the configuration of SPI into the structure pointed by *pConfig*.

11.6.4.1 Function Parameters

Name

pConfig

In/Out

output

Description

The pointer to a configuration structure.

11.6.4.2 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrNullPointer_c
- gSpiErrNoError_c

11.6.5 SPI_WriteSync ()

Prototype

```
spiErr_t SPI_WriteSync
(
    uint32_t Data
);
```

Description

The function is called to transmit a word through SPI. The application remains in this function until the end of transmission.

11.6.5.1 Function Parameters

Name

Data

In/Out

input

Description

32 bit data to be transmitted.

11.6.5.2 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrAsyncOperationPending_c
- gSpiErrAbortError_c
- gSpiErrNoError_c

11.6.6 SPI_ReadSync ()

Prototype

```
spiErr_t SPI_ReadSync
(
    uint32_t *pData
);
```

Description

The function is called to receive a word through SPI. The application remains in this function until the end of transmission.

11.6.6.1 Function Parameters

Name

pData

In/Out

output

Description

A pointer to 32 bit word where will be placed the received data.

11.6.6.2 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrAsyncOperationPending_c
- gSpiErrAbortError_c
- gSpiErrNullPointer_c
- gSpiErrNoError_c

11.6.7 SPI_SetTxAsync ()

Prototype

```
spiErr_t SPI_SetTxAsync
(
    uint32_t Data
);
```

Description

The function is called to put data to be transmitted into the Tx SPI register.

The implementation of the SPI driver is such that an operation is considered complete after the reception of every data word. Because of this, the SPI_StartAsync() should be called after each transaction if multiple words are to be received:

```
SPI_SetTxAsync(...);
```

```
SPI_StartAsync();
```

11.6.7.1 Function Parameters

Name

Data

In/Out

input

Description

32 bit data to be transmitted.

11.6.7.2 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrAsyncOperationPending_c
- gSpiErrNoError_c

11.6.8 SPI_GetRxAsync ()

Prototype

```
spiErr_t SPI_GetRxAsync
(
    uint32_t *pData
);
```

Description

This function is called to get the last data received through SPI.

11.6.8.1 Function Parameters

Name

pData

In/Out

output

Description

Pointer to 32 bit word where it will be placed the received data.

11.6.8.2 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrAsyncOperationPending_c
- gSpiErrNullPointer_c
- gSpiErrNoError_c

11.6.9 SPI_StartAsync ()

Prototype

```
spiErr_t SPI_StartAsync (void);
```

Description

The function is called to start TX/RX operation.

SPI_SetConfig() does not write to the SPI clock control register. Rather, it only saves the settings to an internal global variable. This variable is then used for each interrupt-driven transaction that is initiated by calling SPI_StartAsync().

11.6.9.1 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c,
- gSpiErrAsyncOperationPending_c
- gSpiErrWrongCallbackFunc_c
- gSpiErrNoError_c

11.6.10 SPI_Abort ()

Prototype

```
spiErr_t SPI_Abort (void);
```

Description

The function is called to abort current operation.

11.6.10.1 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrAbortError_c
- gSpiErrNoError_c

11.6.11 SPI_SetCallback ()

Prototype

```
spiErr_t SPI_SetCallback
```

```
(
    spiCallback_t CallbackFunc
);
```

Description

The function is called to set the callback function which will be called from interrupt service routine.

11.6.11.1 Function Parameters

Name

CallbackFunc

In/Out

input

Description

The pointer to the function to be called.

11.6.11.2 Returns

Type

spiErr_t

Description

The status of the operation.

Possible Values

- gSpiErrPortClosed_c
- gSpiErrWrongCallbackFunc_c
- gSpiErrNoError_c

11.6.12 SPI_GetStatus ()

Prototype

```
spiStatus_t SPI_GetStatus (void);
```

Description

The function is called to return the status of the SPI port.

11.6.12.1 Returns

Type

spiStatus_t

Description

The status of the operation.

Possible Values

- gSpiStatusClosed_c
- gSpiStatusIdle_c
- gSpiStatusAsyncOperationPending_c
- gSpiStatusSyncOperationPending_c

11.6.13 SPI_ISR ()

Prototype

```
void SPI_ISR (void);
```

Description

The function will be set as interrupt routine for the SPI peripheral. This will be done using the interface provided by the ITC driver.