Device driver library for Zilog® Z8 Encore!®

API-Documentation

Version 1.00-RC 1

Version 1.00-RC 1

CYA for trademarks

Zilog™
Z8Encore™
I2C™

Document date: 2009-12-01 08:37:00

Version 1.00-RC 1

Content

1. Copyright

Contact: Nils Paulsson
Addr: Engelbrektsgatan 7A, SE-58221, Linkoping, Sweden.
Web : http://ddrz8e.sourceforge.net/

2. Main modules

2.1. Analog to Digital Converter

Prefix

`adc_`, `ADC_`

Files

adc.h, adc.c, adc_inline.c

Description

The *Analog to Digital Converter* module provides support for configuration and control of the built-in analog to digital converter. The module supports polled as well as interrupt driven one-shot and continuous conversion.

Configuration Parameters

`CFG_ADC`

Most Important Functions / Macros

Hardware resources

`ADC_0`

2.2. Binary

Prefix

`bin_`, `b`, `B`

Files

binary.h, binary.c, binary_inline.c

Description

The *Binary* module provides support for binary number notation within C-code. The module also has a few functions for basic 8-bit rotate and multiply utilizing embedded assembler. The main reason for these functions is to eliminate dependencies on external libraries since the ZDS II C-compiler sometimes fallback to library functions as opposed to compiling the statements directly into assembler.

Binary number notation uses the prefixes *b* or *B* to specify that an integer is given in binary form. All values between `b0000/B0000` and `b11111111/B11111111` are supported. The binary number representation does not include any type definition. Consequently, when type casting is omitted the binary numbers are regarded as *int*, unless the compiler has been configured otherwise.

NOTE

The rotate and multiply functions only support dynamic frames. Static frames are not supported owing to differences in calling convention.

Configuration Parameters

None

Most Important Functions / Macros

Hardware resources

2.3. Debugging

Prefix

dbg_, DBG_

Files

debug.h, debug.c, debug_inline.c

Description

The *Debugging* module's main responsibility is to provide various ASSERT() macros. Since the design philosophy of *Ddr-Z8e* is that the hardware resources are allocated by configuration, the *Debug* module support more than one type of assertation. The lowest level is critical assertations dealing with conditions that cannot be resolved at compile time and that must not occur during execution. The module also supports a higher level of assertation. This is specifically aimed for validating conditions that can be expected to occur during development but shouldn't occur in a deployed system. An example of this is invalid name references to hardware resources.

Configuration Parameters

CFG_DEBUG
CFG_DEBUG_ILLEGAL_INSTR
CFG_DEBUG_STATIC_PARAMS

Most Important Functions / Macros

Hardware resources

2.4. Digital IO

Prefix

io_, IO_

Files

io.h, io.c, io_inline.c

Description

The *Digital IO* module provides the framework for using the built-in digital I/O ports. This includes reading/writing bits and bytes, configuring ports, setting port function, etc.

Configuration Parameters

```
CFG_IO
```

Most Important Functions / Macros

Hardware resources

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

2.5. Flash Memory

Prefix

```
fls_, FLS_
```

Files

flash.h, flash.c, flash_inline.c, flash_options.c

Description

The *Flash Memory* module provides support for reading from and writing to the built-in flash memory as well as configuring the flash option bits in the MCU.

Configuration Parameters

```
CFG_CPU_FREQUENCY
CFG_FLS
CFG_FLS_OPTION_BITS
```

Most Important Functions / Macros

Hardware resources

```
FLS_0
```

2.6. I2C

Prefix

`i2c_, I2C_`

Files

i2c.h, i2c.c, i2c_inline.c

Description

The *I2C* module handles configuration, initialization and communication with peripheral circuits using the built-in I2C serial communication controller. The module is transaction oriented and handles communication in an interrupt driven non-blocking fashion. The controller can only operate as an I2C master.

Configuration Parameters

`CFG_I2C`

Most Important Functions / Macros

Hardware resources

`I2C_0`

2.7. Interrupt

Prefix

`irq_, IRQ_`

Files

interrupt.h, interrupt.c, interrupt_inline.c

Description

The *Interrupt* module provides support for the built-in interrupt controller and related logic.

Configuration Parameters

`None`

NOTE
This module is always enabled, configuration wise, when included in a project since there is no reason to disable it

Most Important Functions / Macros

Hardware resources

`IRQ_0`
`IRQ_1`

```
IRQ_2
```

2.8. Register

Prefix

```
REG_, reg_
```

Files

register.h

Description

This module defines the data types and macros necessary for accessing the hardware registers of the built-in Z8 Encore! peripherals.

Configuration Parameters

```
None
```

Most Important Functions / Macros

Hardware resources

2.9. Serial (RS-232)

Prefix

```
ser_, SER_
```

Files

serial.h, serial.c, serial_inline.c

Description

The *Serial* module handles configuration, initialization and communication with external devices through the RS-232 interface. The module supports simultaneous communication on both ports in an interrupt driven non-blocking fashion.

Configuration Parameters

```
CFG_SER
CFG_SER_0
CFG_SER_0_NOFLOW
CFG_SER_0_RTS_BIT
CFG_SER_0_RTS_CTS
CFG_SER_0_RTS_PORT
CFG_SER_0_RX_TIMEOUT
CFG_SER_0_TX_TIMEOUT
```

```
CFG_SER_1
CFG_SER_1_NOFLOW
CFG_SER_1_RTS_BIT
CFG_SER_1_RTS_CTS
CFG_SER_1_RTS_PORT
CFG_SER_1_RX_TIMEOUT
CFG_SER_1_TX_TIMEOUT
```

Most Important Functions / Macros

Hardware resources

```
SER_0
SER_1
```

2.10. Serial Peripheral Interface (SPI)

Prefix

`spi_, SPI_`

Files

spi.h, spi.c, spi_inline.c

Description

The *Serial Peripheral Interface* module handles configuration, initialization and communication with peripheral circuits using the built-in SPI serial communication controller. The module is transaction oriented and handles communication in an interrupt driven non-blocking fashion. The controller can operate as both a master and a slave.

Configuration Parameters

```
CFG_SPI
CFG_SPI_ERRORS_ALL
CFG_SPI_ERROR_COLLISION
CFG_SPI_ERROR_OVERRUN
CFG_SPI_ERROR_SLAVE_ABORT
```

Most Important Functions / Macros

Hardware resources

```
SPI_0
```

2.11. Timer

Prefix

`tmr_, TMR_`

Files

timer.h, timer.c, timer_inline.c

Description

The *Timer* module provides support for configuration and use of the built-in generic timers.

Configuration Parameters

CFG_TMR

Most Important Functions / Macros

Hardware resources

TMR_0
TMR_1
TMR_2
TMR_3

2.12. Watchdog Timer

Prefix

WDT_

Files

wdt.h, wdt.c, wdt_inline.c

Description

The *Watchdog Timer* module provides support for configuration and use of the built-in watchdog timer.

Configuration Parameters

CFG_WDT

Most Important Functions / Macros

Hardware resources

WDT_0

3. Add-on modules

3.1. LCD

Prefix

`lcd_`, `LCD_`

Files

lcd.h, lcd.c, lcd_init.c

Description

The *Serial LCD* module provides basic support for the Milford Instruments Ltd serial LCD Driver Board (part # 6-201). The module uses one of the serial RS-232 ports for communication.

NOTE
This module is not officially supported in version 1.00 of the library. The module is, however, included in the distribution.

Configuration Parameters

```
CFG_LCD
CFG_LCD_SER_0
CFG_LCD_SER_1
CFG_LCD_2400_BAUD
CFG_LCD_9600_BAUD
```

Most Important Functions / Macros

Hardware resources

`SER_0` or `SER_1`

3.2. Simple Numerical to Text Conversion

Prefix

`txt_`

Files

text.h, text.c, text_inline.c

Description

The *Simple Numerical to Text Conversion* module provides simple numerical to text conversion for unsigned hexadecimal numbers. The main purpose is as an aid during development when the standard `printf()` carries too much overhead.

Configuration Parameters

CFG_TXT

Most Important Functions / Macros

Hardware resources

3.3. Ticker

Prefix

tck_, TCK_

Files

ticker.h, ticker.c, ticker_inline.c

Description

The *Ticker* module is an interval counter. The module can use any of the 4 built-in timers as well as any of the built-in baud rate generators (2*RS-232, I2C, SPI) as clock source. The main purpose of this module is to provide functionality for decently accurate time measurements (milliseconds) and fine-grained delays. The accuracy is between 0 and +2 ticks and the ticker resolution can be configured at initialization.

Configuration

```
CFG_TCK
CFG_TCK_I2C
CFG_TCK_SER_0
CFG_TCK_SER_1
CFG_TCK_SPI
CFG_TCK_TICK_8
CFG_TCK_TICK_16
CFG_TCK_TICK_32
CFG_TCK_TMR_0
CFG_TCK_TMR_1
CFG_TCK_TMR_2
CFG_TCK_TMR_3
```

Most Important Functions / Macros

Hardware resources

I2C_0, SER_0, SER_1, SPI_0, TMR_0, TMR_1, TMR_2 or TMR_3

4. Software Configuration

Ddr-Z8e is a highly configurable software package, allowing tailor made and compact solutions. All configurations are made in the user provided file `configure.h`. This file has to be included in any project utilizing *Ddr-Z8e*. The configuration file is divided into 6 sections:

Basic System Setup
Basic Definitions
Debug Support
Module Support Definitions
Module Dependency Definitions
Module Configuration Definitions.

Basic System Setup defines such parameters as MCU clock frequency and memory model. The only purpose of the *Basic Definitions* section is to specify the **_NULL** macro. In *Debug Support* the various levels of assertation are defined. The *Module Support Definitions* section specifies what modules should be enabled. *Module Dependency Definitions* is an internal section that ensures that no module dependencies are violated. Finally, in the *Module Configuration Definitions* section each module can be specifically configured for certain needs in a project. The various configurations in the *Module Configuration Definitions*-section only have impact when the entire module has been enabled in the *Module Support Definitions* section. Thus, all configurations can be left intact when a module is disabled.

4.1. Parameters usage

All configuration parameters are defined using the C-preprocessor. A configuration parameter can therefore be one of three types; *switch*, *value* or *combined*.

A *switch* is a parameter that can only be defined or not defined. A parameter is defined using the preprocessor directive `#define` and a parameter can be undefined using the preprocessor directive `#undef`. A parameter can of course also be undefined by removing or commenting out the entire `#define PARAMETER_X` statement from `configure.h`. When a parameter is defined it usually means that the corresponding functionality is be enabled. When undefined, the functionality is consequently disabled. Typical *switch*-parameters are the parameters in the *Module Support Definitions*-section such as `CFG_SPI`, which enables the SPI-support.

A *value* is a parameter that specifies a certain setting. One such parameter is the parameter `CFG_CPU_FREQUENCY`, which specifies the core clock frequency of the MCU. The construct of a valid setting is specific for each parameter. The fact that a *value*-parameter is defined or not defined has no meaning in any other sense than that it may be a configuration error when defined or undefined in the wrong context.

A *combined* configuration parameter has a meaning both in regards to its defined/undefined state as well as the value of the parameter. An example of this type is `CFG_SER_1_TX_TIMEOUT`. This parameter specifies the transmission timeout for UART 1. When defined, its integer value specifies how many times the serial (RS-232) device driver should try to send data before timing out. When this parameter is undefined, the support for timeout is disabled entirely and the serial device driver will try forever to transmit data.

Some configuration parameters are mutually exclusive, i.e. they can't be enabled at the same time. One such case is `CFG_MODEL_SMALL` and `CFG_MODEL_LARGE`. The first specifies that the library is compiled using the small memory model and the second specifies that the library is compiled using the large memory model. Obviously, both models can't be used at the same time for a given source code module.

Ddr-Z8e takes some precautions trying to prevent invalid configurations. However, owing to limitations in the C-preprocessor it isn't possible to perform a waterproof validation of the configuration file wherefore it is possible to have erroneous settings in `configure.h`.

4.2. Naming conventions

The configuration parameter names all start with the prefix `CFG_` followed by the prefix of the module to which they apply. Since the parameters are all preprocessor definitions the names are also capitalized. For example, the switch that enables the RS-232 module is named `CFG_SER` and the parameter that sets the receiver timeout value for UART number 0 is named `CFG_SER_0_RX_TIMEOUT`.

4.3. Basic System Setup Parameters

Parameter	Description	Type	Possible Value	Unit	Comments
CFG_CPU_FREQUENCY	Specifies the core MCU clock frequency. The supported values are the frequencies at which *Ddr-Z8e* has predefined support. When using a nonstandard MCU frequency baud rate generators, and other system clock dependent settings have to be set by specific values provided by the specific application.	Value	unsigned long > 0 Standard values supported by *Ddr-Z8e*: 18432000L 20000000L	Hertz	For frequencies other than the predefined baud rates and other frequency dependent constants are not valid.
CFG_MODEL_SMALL	Specifies that the program is compiled for small memory model.	Switch			Mutually exclusive: CFG_MODEL_LARGE
CFG_MODEL_LARGE	Specifies that the program is compiled for large memory model.	Switch			Mutually exclusive: CFG_MODEL_SMALL
CFG_INLINE_MACRO	Specifies that *Ddr-Z8e* should use inline macros for all applicable device driver functions resulting in fast and compact code. When this parameter is NOT defined standard C calling convention is used to facilitate debugging.	Switch			

4.4. Basic Definitions

Parameter	Description	Type	Possible Value	Unit	Comments
_NULL	Specifies the NULL-pointer. This was needed for older versions of ZDS II to work properly.	Value	(void *) 0x0000		

4.5. Debug Support Parameters

Parameter	Description	Type	Possible Value	Unit	Comments
CFG_DEBUG	Specifies that the highest level of debug assertation should be enabled. In reality this means that the **DBG_ASSERT()** macro is enabled. This macro should be used in cases which it is under no circumstances wise to disable validation of an expression. When an assertation fails to validate an expression the function **dbg_Failed()** in the debug module is called. This function can be tailored according to various needs.	Switch			
CFG_DEBUG_ILLEGAL_INSTR	Specifies that attempts to execute illegal instructions should be captured. This is sometimes useful when debugging code utilizing function pointers. When an illegal instruction is found, the interrupt handler **dbg_IllegalInstruction()** located in *debug.c* is called and the MCU is brought to halt-mode as default action.	Switch			
CFG_DEBUG_STATIC_PARAMS	Specifies that basic assertation should be performed, i.e. the macro **DBG_ASSERT_STATIC()** is enabled. This macro is specifically aimed for validating expressions that are expected to be non-dynamic once the development phase has ended. A typical case where this macro is used in *Ddr-Z8e* is to verify names of the hardware resources in a call to a function or a macro. For example, the macro **IRQ_CLEAR(***IrqName***)** utilize **DBG_ASSERT_STATIC()** to	Switch			

Parameter	Description	Type	Possible Value	Unit	Comments
	validate that *IrqName* is a valid irq reference. However, when the software development cycle has ended, it is reasonable to assume that all illegal name references have been found wherefore the assertation can be removed, thereby reducing code size. When an assertation fails to validate an expression the function **dbg_StaticFailed()** in the debug module is called. This function can be tailored according to various needs.				

4.6. Module Support Definitions

Parameter	Description	Type	Possible Value	Unit	Comments
CFG_ADC	Enables support for the built-in analog to digital converter.	Switch			
CFG_FLS	Enables support for reading and writing to the built-in flash memory.	Switch			
CFG_I2C	Enables support for serial communication using the built-in I2C controller.	Switch			
CFG_IO	Enables the module for digital I/O.	Switch			
CFG_LCD	Enables support for Milford Instrument's serial LCD Driver Board (part # 6-201).	Switch			Add-on module
CFG_SER	Enables support for RS-232 communication using the built-in UARTs.	Switch			
CFG_SPI	Enables support for serial communication using the built-in Serial Peripheral Interface (SPI) controller.	Switch			
CFG_TCK	Enables support for the ticker clock.	Switch			Add-on module
CFG_TMR	Enables support for the built-in timers.	Switch			
CFG_TXT	Enables basic numerical to hexadecimal text conversion support.	Switch			Add-on module
CFG_WDT	Enables support for the built-in Watch Dog Timer (WDT)	Switch			

NOTE
Even though a specific module has been enabled and configured in `configure.h`, it's source code file is not automatically included in the ZDSII project. This has to be done manually from within the ZDS II environment.

4.7. Module Dependency Definitions

Parameter	Description	Possible Value	Unit	Comments
This section sees to that the inter-module dependencies are handled correctly and should therefore be left unchanged.	THIS HAS NOT BEEN IMPLEMENTED YET AND MAY OR MAY NOT BE SO IN THE FUTURE			

4.8. Module Configuration Definitions

Subsection CLG_FLS

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_FLS_OPTION_BITS	Specifies the settings of the flash-option bits. When undefined, the default flash option bits settings are in effect.	Combined	Any of the predefined values or the result of an bitwise AND-operation between 2 or more of the predefined values.		All flash option bits set to 1.	Predefined Values: FLS_WDT_INTERRUPT FLS_WDT_ALWAYS_ON FLS_VBO_DISABLE_ON_STOP FLS_CODE_READ_PROTECT FLS_WRITE_PROTECT FLS_OSC_RC_NETWORK FLS_OSC_LOW_POWER FLS_OSC_MED_POWER

Subsection CFG_LCD

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_LCD_SER_0	Specifies that RS232 controller 0 should be used for communicating with the serial LCD.	Switch				Mutually exclusive: CFG_LCD_SER_1 Note: CFG_SER must be defined as well in the *Module Support Definitions* section and CFG_SER_0 must bed defined in the *Module Configuration Definitions* section.
CFG_LCD_SER_1	Specifies that RS232 controller 1 should be used for communicating with the serial LCD.	Switch				Mutually exclusive: CFG_LCD_SER_0 CFG_SER must be defined as well in the *Module Support Definitions* section and CFG_SER_1 must bed defined in the *Module Configuration Definitions* section.
CFG_LCD_2400_BAUD	Specifies a baud rate of 2400 bits/s for communicating with the LCD controller.	Switch				Mutually exclusive: CFG_LCD_9600_BAUD
CFG_LCD_9600_BAUD	Specifies a baud rate of 9600 bits/s for communicating with the	Switch				Mutually exclusive: CFG_LCD_2400_BAUD

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
	LCD controller.					

Subsection CFG_SER

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_SER_0	Enables support for RS232-controller number 0.	Switch				
CFG_SER_1	Enables support for RS232-controller number 1.	Switch				

Subsection CFG_SER -- CFG_SER_0

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_SER_0_TX_TIMEOUT	When defined, specifies the timeout for trying to initiate a transmission using RS232-controller 0. When this period of time has ended a timeout error occur. When this parameter is NOT defined, the MCU will wait indefinitely. This parameter usually only has a meaning when using hardware flow control (RTS/CTS). It does, however, also work when using no flow control, e.g. for very long and slow transaction.	Combined	unsigned long > 0	Number of attempts to send a specific byte. The value of this depends on the MCU core frequency as well as the overall MCU utilization. A timeout of 0xffffffL corresponds to about 5-6 seconds using 18.432 MHz core clock frequency and limited workload		
CFG_SER_0_RX_TIMEOUT	When defined, specifies the amount of time to wait for a byte to be received by RS232-controller 0. When this time has elapsed and no data has arrived, a timeout error will be generated.	Combined	unsigned long > 0	Number of attempts to receive a specific byte. The value of this depends on the MCU core frequency as well as the overall MCU		

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
	When this parameter is NOT defined, the MCU will wait indefinitely. This parameter has a meaning when using BOTH hardware flow control (RTS/CTS) and no flow control at all.			utilization. A timeout of 0xffffffL correspond to about 5-6 seconds using 18.432 MHz core clock frequency and limited workload		
CFG_SER_0_NOFLOW	Specifies that no flow control should be used when communicating over RS232-controller 0.	Switch				Mutually exclusive: CFG_SER_1_RTS_CTS
CFG_SER_0_RTS_CTS	Specifies that hardware flow control (RTS/CTS) should be used when communicating over RS232-controller 0.	Switch				Mutually exclusive: CFG_SER_0_NOFLOW

Subsection CFG_SER -- CFG_SER_0 -- CFG_SER_0_RTS_CTS

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_SER_0_RTS_PORT	Specifies to witch port the outgoing RTS signal is tied for RS232-controller 0.	Value	IO_A IO_B IO_C IO_D IO_E IO_F IO_G IO_H	Named port reference.		This setting only has meaning for when hardware flow control is enabled
CFG_SER_0_RTS_BIT	Specifies the port pin to which the RTS-signal is connected using the port specified by **CFG_SER_0_RTS_POR**.	Value	0X01 (b00000001) 0X02 (b00000010) 0X04 (b00000100) 0X08 (b00001000) 0X10 (b00010000) 0X20 (b00100000) 0X40 (b01000000) 0X80 (b10000000)	Bit number		This setting only has meaning for when hardware flow control is enabled

Subsection CFG_SE -- CFG_SER_1

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_SER_1_TX_TIMEOUT	When defined, specifies the timeout for trying to initiate a transmission using RS232-controller 1. When this period of time has ended a timeout error occur. When this parameter is NOT defined, the MCU will wait indefinitely. This parameter usually only has a meaning when using hardware flow control (RTS/CTS). It does, however, also work when using no flow control, e.g. for very long and slow transaction.	Combined	unsigned long > 0	Number of attempts to send a specific byte. The value of this depends on the MCU core frequency as well as the overall MCU utilization. A timeout of 0xffffffL corresponds to about 5-6 seconds using 18.432 MHz core clock frequency and limited workload		
CFG_SER_1_RX_TIMEOUT	When defined, specifies the amount of time to wait for a byte to be received by RS232-controller 1. When this time has elapsed and no data has arrived, a timeout error will be generated. When this parameter is NOT defined, the MCU will wait indefinitely. This parameter has a meaning when using BOTH hardware flow control (RTS/CTS) and no flow control at all.	Combined	unsigned long > 0	Number of attempts to receive a specific byte. The value of this depends on the MCU core frequency as well as the overall MCU utilization. A timeout of 0xffffffL correspond to about 5-6 seconds using 18.432 MHz core clock frequency and limited workload		
CFG_SER_1_NOFLOW	Specifies that no flow control should be used when communicating over RS232-controller 1.	Switch				Mutually exclusive: CFG_SER_1_RTS_CTS
CFG_SER_1_RTS_CTS	Specifies that hardware flow control (RTS/CTS)	Switch				Mutually exclusive: CFG_SER_1_NOFLOW

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
	should be used when communicating over RS232-controller 1.					

Subsection CFG_SER -- CFG_SER_1 -- CFG_SER_1_RTS_CTS

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_SER_1_RTS_PORT	Specifies to witch port the outgoing RTS signal is tied for RS232-controller 1.	Value	IO_A IO_B IO_C IO_D IO_E IO_F IO_G IO_H	Named port reference.		This setting only has meaning for when hardware flow control is enabled.
CFG_SER_1_RTS_BIT	Specifies the port pin to which the RTS-signal is connected using the port specified by **CFG_SER_1_RTS_POR**.	Value	0X01 (b00000001) 0X02 (b00000010) 0X04 (b00000100) 0X08 (b00001000) 0X10 (b00010000) 0X20 (b00100000) 0X40 (b01000000) 0X80 (b10000000)	Bit number		This setting only has meaning for when hardware flow control is enabled.

Subsection CFG_SPI

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_SPI_ERRORS_ALL	Specifies that all SPI error conditions below should be reported by **spi_GetError()**.	Switch				
CFG_SPI_ERROR_COLLISION	Specifies that only collision error should be reported by **spi_GetError()**.	Switch				
CFG_SPI_ERROR_OVERRUN	Specifies that only overrun error should be reported by **spi_GetError()**.	Switch				
CFG_SPI_ERROR_SLAVE_ABORT	Specifies that only slave abort error should be reported by **spi_GetError()**.	Switch				

Subsection CFG_TCK

Parameter	Description	Type	Possible Value	Unit	Default Value	Comments
CFG_TCK_TICK_8	Specifies that the ticker should be an 8-bit variable. The ticker thus counts from 0 to 0xff and then begins at 0 again.	Switch				Mutually exclusive: CFG_TCK_TICK_16 CFG_TCK_TICK_32
CFG_TCK_TICK_16	Specifies that the ticker should be a 16-bit variable. The ticker thus counts from 0 to 0xffff and then begins at 0 again.	Switch				Mutually exclusive: CFG_TCK_TICK_8 CFG_TCK_TICK_32
CFG_TCK_TICK_32	Specifies that the ticker should be a 32-bit variable. The ticker thus counts from 0 to 0xffffffff and then begins at 0 again.	Switch				Mutually exclusive: CFG_TCK_TICK_8 CFG_TCK_TICK_16
CFG_TCK_TMR_0	Specifies that timer 0 should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_1 CFG_TCK_TMR_2 CFG_TCK_TMR_3 CFG_TCK_SER_0 CFG_TCK_SER_1 CFG_TCK_SPI CFG_TCK_I2C Note: CFG_TMR must be in the *Module Support Definitions* section as well.
CFG_TCK_TMR_1	Specifies that timer 1 should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_0 CFG_TCK_TMR_2 CFG_TCK_TMR_3 CFG_TCK_SER_0 CFG_TCK_SER_1 CFG_TCK_SPI CFG_TCK_I2C Note: CFG_TMR must be in the *Module Support Definitions* section as well.

CFG_TCK_TMR_2	Specifies that timer 2 should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_0 CFG_TCK_TMR_1 CFG_TCK_TMR_3 CFG_TCK_SER_0 CFG_TCK_SER_1 CFG_TCK_SPI CFG_TCK_I2C Note: CFG_TMR must be in the *Module Support Definitions* section as well.
CFG_TCK_TMR_3	Specifies that timer 3 should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_0 CFG_TCK_TMR_1 CFG_TCK_TMR_2 CFG_TCK_SER_0 CFG_TCK_SER_1 CFG_TCK_SPI CFG_TCK_I2C Note: CFG_TMR must be in the *Module Support Definitions* section as well.
CFG_TCK_SER_0	Specifies that the baud rate generator of serial controller 0 should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_0 CFG_TCK_TMR_1 CFG_TCK_TMR_2 CFG_TCK_TMR_3 CFG_TCK_SER_1 CFG_TCK_SPI CFG_TCK_I2C Note 1: CFG_SER must be defined in the *Module Support Definitions* section as well. Note 2: CFG_SER_0 must **NOT** be defined since that parameter is used for enabling serial communication using controller 0. The controller can't be used for

						serial communication and ticker clock source at the same time. Note 3: If the only purpose of the serial-module is to provide a ticker clock source the file *serial.c* must **NOT** be included in the project. However, if one RS232-controller is used for serial communication and the other one as ticker clock source the file *serial.c* **SHOULD** be included in the project.
CFG_TCK_SER_1	Specifies that the baud rate generator of serial controller 1 should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_0 CFG_TCK_TMR_1 CFG_TCK_TMR_2 CFG_TCK_TMR_3 CFG_TCK_SER_0 CFG_TCK_SPI CFG_TCK_I2C Note 1: CFG_SER must be defined in the *Module Support Definitions* section as well. Note 2: CFG_SER_1 must **NOT** be defined since that parameter is used for enabling serial communication using controller 1. The controller can't be used for serial communication and ticker clock source at the same time. Note 3: If the only purpose of the serial-module is to provide a ticker clock source the file *serial.c* must **NOT** be included in the project. However, if one RS232-controller is used for serial communication and the other one as ticker clock source the file *serial.c* **SHOULD** be included in the project.

CFG_TCK_SPI	Specifies that the baud rate generator of the SPI controller should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_0 CFG_TCK_TMR_1 CFG_TCK_TMR_2 CFG_TCK_TMR_3 CFG_TCK_SER_0 CFG_TCK_SER_1 CFG_TCK_I2C Note 1: CFG_SPI must be defined in the *Module Support Definitions* section as well. Note 2: The file *spi.c* must **NOT** be included in the project. Communication using the SPI controller is not allowed at the same time the baud rate generator is used as ticker time source.
CFG_TCK_I2C	Specifies that the baud rate generator of the I2C controller should be used as the ticker time source.	Switch				Mutually exclusive: CFG_TCK_TMR_0 CFG_TCK_TMR_1 CFG_TCK_TMR_2 CFG_TCK_TMR_3 CFG_TCK_SER_0 CFG_TCK_SER_1 CFG_TCK_SPI Note 1: CFG_I2C must be defined in the *Module Support Definitions* section as well. Note 2: The file *i2c.c* must **NOT** be included in the project. Communication using the I2C controller is not allowed at the same time the baud rate generator is used as ticker time source.

5. Functions and Macros

5.1. Naming Convention

All function and macro names are prefixed with the corresponding module prefixes specified in chapters 2 and 3. For example, all functions and macros in the interrupt module are prefixed with either `irq_` or `IRQ_`. Function names start with the lower case prefix and macros start with the uppercase. Macros are named using all capital letters having words separated by an underscore:

```
IRQ_SET_VECTOR()
IRQ_CLEAR()
```

Functions are named using lower case letter except for the first letter of each word in the name:

```
irq_Enable()
irq_GetStatus()
```

5.2. Inline macros vs. C-functions

A benefit of using C-macros is that it allows compact and performance centric inlining of code snippets without the overhead of the standard C calling convention. One drawback of macros is that there is no type checking performed. Another is lack of code isolation. This can make it very difficult to find even simple bugs during development. To facilitate both the need for compact/efficient code and ease of debugging *Ddr-Z8e* provides two ways of calling a macro. One uses the C calling convention providing type checking and code isolation. The other includes the bare macro directly in the program source code. Which version to be used is determined by the configuration parameter `CFG_INLINE_MACRO` (Chapter 4.3).

Since C doesn't support function inlining the way C++ does another solution is needed. Let's look at the following macro:

```
//#inline unsigned char SER_TX_IS_ENABLED(reg_addr Uart)
#define SER_TX_IS_ENABLED(Uart) \
   ( \
   *REG_OFFSET(Uart,SER_CTRL_0) & SER_CTRL_TX \
   )
```

This is a macro that checks if transmission is enabled for RS232-controller `Uart` and returns 1 if so, other wise 0 is returned. The macro doesn't look too complicated but note that the macro itself contains yet another macro, i.e. `REG_OFFSET(Uart,SER_CTRL_0)`. This inclusion of macros within macros can quickly generate large and complex code fragments that are barely visible outside the preprocessor and compiler. By wrapping the macro in a standard C function we get both type checking and some code isolation that can be very useful during the development phase:

```
/* Functional macro wrapper */
unsigned char ser_TxIsEnabled_wrapper(reg_addr Uart)
   {

   /* Invoke macro and return result */
   return(SER_TX_IS_ENABLED(Uart));
   }
```

The wrapper functions of each device driver module is located in the corresponding "*_inline.c" file. E.g. for the RS232 module that would be the file serial_inlince.c.

The final missing piece to allow configurability of which version to use, inline macro or function call, is to define yet another macro pointing to either the wrapper function or the original macro:

```
/* SER_TX_IS_ENABLED() */
#ifdef CFG_INLINE_MACRO

   /* Using inline macros. */
   #define ser_TxIsEnabled(Uart) SER_TX_IS_ENABLED(Uart)
#else

   /* Using wrapper functions. */
   unsigned char ser_TxIsEnabled_wrapper(reg_addr Uart);
   #define ser_TxIsEnabled(Uart) ser_TxIsEnabled_wrapper(Uart)
#endif
```

This final macro (ser_TxIsEnabled(Uart)) is then the one that should be used in user applications.

The code for all these wrapper functions and extra macros is generated by a Perl script (InlineCodeGenerator.pl) located in <*Ddr-Z8e root*>\Utilities\InlineCodeGenerator\. For the inline code generator to know for which macros to create wrappers and what data types to use, each macro has a function call definition attached:

```
//#inline ReturnDataType MacroName(ParamDataTypeA ParamNameA)
```

The inline-files (*_inline.c) are entirely created by the generator. The configurable user macros (e.g. ser_TxIsEnabled(Uart)) are also created by the inline generator but appended to each module's include file, e.g. serial.h. There is more information about the inline generator within the Perl script.

5.3. API documentation

adc_Close()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
void adc_Close();
```

Return

Description

`adc_Close()` shuts down and disables the A/D converter. Associated interrupt is disabled and the setting for call back function is reset. The ADC is shut down immediately without concerns for possible ongoing conversions.

Example

```
unsigned short AdcValue;

// Setup analog port
Io_SetAltFunct(IO_B,IO_B_ALT_AD_0);

// Initialize the adc-module (no callback)
adc_Init(_NULL);
adc_SelectInp(ADC_SEL_0);

// Perform an analog to digital conversion
adc_Start();
while (adc_IsBusy())
   ;

// Get the digital value
AdcValue = adc_GetValue();

// Shut down analog to digital converter
adc_Close();
```

adc_GetValue()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
unsigned short adc_GetValue(void);
```

Return

The numerical value from the latest analog to digital conversion.

Description

`adc_GetValue()` reads the 10-bit numerical value from the latest analog to digital conversion. Only the 10 least significant bits (LSB) are valid.

Example

```
unsigned short AdcValue;

// Setup analog port
Io_SetAltFunct(IO_B,IO_B_ALT_AD_0);

// Initialize the adc-module (no callback)
adc_Init(_NULL);
adc_SelectInp(ADC_SEL_0);

// Perform an analog to digital conversion
adc_Start();
while (adc_IsBusy())
   ;

// Get the digital value
AdcValue = adc_GetValue();

// Convert to hexadecimal text and display the result
txt_htoa(AdcValue, Text);
…
```

adc_Init()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
void adc_Init(void (*CallBack)(unsigned short AdcValue));
```

Return

Description

`adc_Init()` initializes the A/D controller with default setting and makes it ready for operation. The `CallBack` parameter is a pointer to a user written function of the type `void (*CallBack)(unsigned short AdcValue)`. This function will be invoked once a one-shot conversion has finished. The call back function executes during interrupt time wherefore the same precautions apply the call back function as to any interrupt handler. The call back invocation can be disabled by specifying `_NULL` as the parameter to `adc_Init()`. When the A/D controller is operated in continuous mode the call back is only invoked after the first A/D conversion.

The call back function is of the type `CallBack(unsigned short AdcValue)`, where `AdcValue` is the numerical result from the latest A/D-conversion.

The default setting after initialization is:

- one-shot A/D conversion
- internal voltage reference
- analog channel 0 (zero) selected

Example

```
void CallBack(unsigned short AdcValue)
   {

   // Process the value in some fashion
   Process(AdcValue);

   // Start another A/D conversion
   adc_Start();
   }


void main(void)
   {

   // Initilize ADC
   Io_SetAltFunct(IO_B,IO_B_ALT_AD_0);
   adc_Init(CallBack);

   // Start the first A/D conversion
   adc_Start();
   }
```

adc_IsBusy()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
unsigned char adc_IsBusy();
```

Return

Status of ongoing A/D conversion

Description

`adc_IsBusy()` returns a non-zero value when the ADC is occupied with performing an A/D conversion. When zero (0) is returned the ADC is ready to perform another A/D conversion.

Example

```
// Start an A/D conversion
adc_Start();

// Wait for it to complete
while (adc_IsBusy())
   ;

// Read the value
AdcValue = adc_GetValue();
```

adc_Reset()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_Reset()
```

Return

Description

This macro resets the A/D-controller and puts it into default state and configuration. This means:

- one-shot conversion
- conversion shut off
- internal voltage reference
- analog input channel 0 selected

Example

adc_SelectInp()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_SelectInp(unsigned char AdChannel);
```

Return

Description

`adc_SelectInp()` specifies which of the analog input signals that should be targeted for conversion. The parameter `AdChannel` specifies the name of the analog input. Valid names are:

`ADC_SEL_0`	(Alternate function `IO_B_ALT_AD_0`)
`ADC_SEL_1`	(Alternate function `IO_B_ALT_AD_1`)
`ADC_SEL_2`	(Alternate function `IO_B_ALT_AD_2`)
`ADC_SEL_3`	(Alternate function `IO_B_ALT_AD_3`)
`ADC_SEL_4`	(Alternate function `IO_B_ALT_AD_4`)
`ADC_SEL_5`	(Alternate function `IO_B_ALT_AD_5`)
`ADC_SEL_6`	(Alternate function `IO_B_ALT_AD_6`)
`ADC_SEL_7`	(Alternate function `IO_B_ALT_AD_7`)
`ADC_SEL_8`	(Alternate function `IO_H_ALT_AD_8`)
`ADC_SEL_9`	(Alternate function `IO_H_ALT_AD_9`)
`ADC_SEL_10`	(Alternate function `IO_H_ALT_AD_10`)
`ADC_SEL_11`	(Alternate function `IO_H_ALT_AD_11`)

Default analog input after the A/D-controller has been initialized with `adc_Init()` or reset by `adc_Reset()` is `ADC_SEL_0`.

Example

```
// Configure alternate function on the pin that corresponds to
// analog input channel 10, i.e. Port H, Bit 2.
io_SetAltFunct (IO_H, IO_H_ALT_AD_10);

// Initlialize the A/D-controller
adc_Init(CallBack);

// Select A/D-input number 10
adc_SelectInp(ADC_SEL_10);
```

adc_SetContinuous()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_SetContinuous();
```

Return

Description

`adc_SetContinuous()` configures the A/D-converter for continuous operation at which a new conversion automatically is commenced as soon as a previous has finished.

Example

```
unsigned short AdcValue;

// Init ADC and analog port
io_SetAltFunct(IO_B,IO_B_ALT_AD_0);
adc_Init(_NULL);
adc_SelectInp(ADC_SEL_0);

// Specify continuous operation
adc_SetContinuous();

// Read values
adc_Start();
AdcValue = adc_GetValue();
while(1)
   {

   // Wait for different voltage reading
   while(AdcValue == adc_GetValue())
      ;

   // Get the new voltage level
   AdcValue = adc_GetValue();
   }
```

adc_SetExtVref ()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_SetExtVref();
```

Return

Description

`adc_SetExtVref()` configures the A/D controller to use an external voltage reference, provided through the VREF pin on the MCU. Default setting is to use internal voltage reference.

Example

```
// Init ADC and analog port
Io_SetAltFunct(IO_B,IO_B_ALT_AD_0);
adc_Init(_NULL);
adc_SelectInp(ADC_SEL_0);

// Specify external voltage reference
adc_SetExtVref();

// Read values
adc_Start();
```

adc_SetIntVref()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_SetIntVref();
```

Return

Description

`adc_SetIntVref()` specifies that the A/D-controller should use the internal voltage reference during A/D-conversions. This is the default setting.

Example

```
// Init ADC and analog port
Io_SetAltFunct(IO_B,IO_B_ALT_AD_0);
adc_Init(_NULL);
adc_SelectInp(ADC_SEL_0);

// Specify external voltage reference
adc_SetExtVref();

// Read values
adc_Start();

// Go back to internal voltage reference
adc_SetIntVref();
```

adc_SetOneShot()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_SetOneShot();
```

Return

Description

`adc_SetOneShot()` configures the A/D-converter for one-shot operation. In one-shot operation the voltage level of the active analog input is read, converted to digital representation and then the A/D-converter shuts down. When a call back function is used (see `adc_Init()`) this will also be invoked once the conversion is done. This is the default mode of operation.

Example

```
unsigned short Counter;

// Specify continuous operation
adc_SetOneShot();

// Read 256 values
adc_Start();
for (Counter  = 0;Counter < 256;Counter++)
   {
   Delay();
   Process(adc_GetValue());
   }

// Stop A/D conversion
adc_Stop();

// Specify one-shot operation
adc_SetOneShot();
```

adc_Start()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_Start();
```

Return

Description

`adc_Start()` starts an A/D conversion. When the controller is configured for continuous operation, conversions carries on until a call to `adc_Stop()` is issued. When configured for one-shot mode, the A/D-converter automatically shuts down once the conversion has finished and a new call to `adc_Start()` has to be issued for yet another A/D-conversion.

Example

```
// Initialize A/D controller (one-shot operation is default)
adc_Init(_NULL);

// Start a new conversion
adc_Start();

// Wait until the conversion is done
while (adc_IsBusy())
   ;

// Do something with the value
Process (adc_GetValue());

// Continuous operation
adc_SetContinuous();

// Start continuous conversion
adc_Start();

// Perform some processing
…

// Stop the A/D converter
adc_Stop();
```

adc_Stop()

Module

Analog to Digital Converter

Synopsis

```
#include "adc.h"
adc_Stop();
```

Return

Description

`adc_Stop()` ends A/D conversions when the controller is operating in continuous mode. In one-shot mode this macro has no effect.

Example

See `adc_Start()`

bin_LRotate()

Module

Binary

Synopsis

```
#include "binary"
unsigned char bin_LRotate(unsigned char Byte, unsigned char NumRotate);
```

Return

The value of `Byte` rotated `NumRotate` positions to the left.

Description

`bin_LRotate()` takes the parameter `Byte` and rotate its content `NumRotate` positions to the left. Carry over bits are rotated back to the least significant bit position. The function utilizes the built-in assembler instruction for rotating 8-bit numbers.

NOTE
`bin_LRotate()` only supports dynamic frames owing to differences in calling convention for static frames.

Example

```
unsigned char Byte;
unsigned char NumRotates;
unsigned char RtnVal;

Byte = 0x01; // b00000001
NumRotates = 0x09;
RtnVal = 0;

// This will rotate Byte 9 times. After 7 rotations, the most
// significant bit will wrap around and start from the least
// significant bit-position  again.
RtnVal = bin_LRotate(Byte, NumRotates); // RtnVal is now == b00000010
```

bin_RRotate()

Module

Binary

Synopsis

```
#include "binary"
unsigned char bin_RRotate(unsigned char Byte, unsigned char NumRotate);
```

Return

The value of `Byte` rotated `NumRotate` positions to the right.

Description

`bin_LRotate()` takes the parameter `Byte` and rotate its content `NumRotate` positions to the right. Carry-over bits are rotate back to the most significant bit position. The function utilizes the built-in assembler instruction for rotating 8-bit numbers.

NOTE
`bin_RRotate()` only supports dynamic frames owing to differences in calling convention for static frames.

Example

```
unsigned char Byte;
unsigned char NumRotates;
unsigned char RtnVal;

Byte = 0x80; // b10000000
NumRotates = 0x09;
RtnVal = 0;

// This will rotate Byte 9 times. After 7 rotations, the most
// significant bit will wrap around and start from the most
// significant bit-position  again.
RtnVal = bin_RRotate(Byte, NumRotates); // RtnVal is now == b01000000
```

bin_UMultiply()

Module

Binary

Synopsis

```
#include "binary.h"
unsigned short bin_UMultiply(unsigned char Byte1, unsigned char Byte2);
```

Return

`Byte1` multiplied by `Byte2`.

Description

This function performs an unsigned multiplication between two 8-bit integers and returns the result as an `unsigned short`. `bin_UMultiply()` utilizes the built-in assembler instruction for unsigned multiplications.

NOTE
`bin_UMultiply()` only supports dynamic frames owing to differences in calling convention for static frames.

Example

```
unsigned char Byte 1;
unsigned char Byte2;
unsigned short Product;

Byte1 = 5;
Byte2 = 5;
Product = bin_UMultiply(Byte1, Byte2); // Product = 25
Product = bin_UMultiply(255, 255); // Product = 65025
```

DBG_ASSERT ()

Module

Debugging

Synopsis

```
#include "debug.h"
DBG_ASSERT(Expression);
```

Return

Description

`DBG_ASSERT()` assures that `Expression` evaluates to a non-zero (true) value before commencing execution of subsequent code. If `Expression` evaluates to zero (false) the function `dbg_Failed()` is called and, depending on implementation, further program execution is potentially halted.

The difference between `DBG_ASSERT()` and the common C-version `ASSERT()` is that `DBG_ASSERT()` is tailored for use in RAM-limited execution environment. Instead of referring to the `__FILE__` definition (see http://en.wikipedia.org/wiki/Assert.h), to point out in what file something went wrong, `DBG_ASSERT()` instead uses file numbers. The reason is that if `__FILE__` was used, a copy of the current filename, including path, would be inserted at every single call to `DBG_ASSERT()`. Consequently, the RAM memory would in a very short time be filled with identical copies of `__FILE__`. File numbers are defined by setting `DBG_FILE_NUM` to a suitable file number identifying the specific file. `DBG_FILE_NUM` apply to `*.c` files only. `DBG_FILE_NUM` numbers from 100 to 299 are reserved for *Ddr-Z8e* files. Any other file number can be used in user code.

The `DBG_ASSERT()` macro can be disabled in the configuration file `configure.h`. As a result, all code and expressions used in `DBG_ASSERT()`-statements are removed during compilation resulting in reduced code size. This does, of course, also mean that the corresponding validity tests are disabled, which may or may not be feasible. As a mean of increasing the flexibility, the *Debug* module supports two levels of assert-macros. `DBG_ASSERT()` is considered as the lowest level of expression validation and is therefore normally **not** disabled in production code. The other level of validation is handled by `DBG_ASSERT_STATIC()`, which is directly targeted at development code.

Example

```
void Copy(void *Source, void *Dest, int ByteCount)
   {

   // Assure that there are no NULL-pointers
   DBG_ASSERT(Source != _NULL && Dest != _NULL);

   // Copy the data
   …
   }
```

DBG_ASSERT_STATIC()

Module

Debugging

Synopsis

```
#include "debug.h"
DBG_ASSERT_STATIC(Expression);
```

Return

Description

`DBG_ASSERT_STATIC()` ensures that `Expression` evaluates to a non-zero (true) value before commencing execution of subsequent code. If `Expression` evaluates to zero (false) the function `dbg_FailedStatic()` is called and, depending on implementation, further program execution is potentially halted. `DBG_ASSERT_STATIC()` should be viewed as a less stringent form of assertation than `DBG_ASSERT()`.

`DBG_ASSERT_STATIC()` is specifically aimed for use during the development stage in a project to validate expressions and parameters that can be expected to be bug free once the project reaches production quality. One such example is validating resource names in function and macro calls. Thus, there is a need to catch this and similar errors during development. However, in production code invalid references can be expected to be eliminated. Consequently, in production code, `DBG_ASSERT_STATIC()` can, arguably, be disabled resulting in smaller and faster code. Disabling `DBG_ASSERT_STATIC()` is of course a decision that has to be made for each separate case. Enabling/disabling of `DBG_ASSERT_STATIC()` is done in the file `configure.h`.

Example

```
void OpenUart(char UartName)
   {

   // Check uart name
   DBG_ASSERT_STATIC(UartName==SER_0 || UartName==SER_1);

   // Start the uart engine
   ser_Init(UartName);
   }
```

DBG_FILE_NUM

Module

Debugging

Synopsis

```
#include ""
DBG_FILE_NUM
```

Return

Description

`DBG_FILE_NUM` specifies the file number to which a file is referred in case a `DBG_ASSERT()` or `DBG_ASSERT_STATIC()` statement fails. The file `number` is the global number reference. Numbers are assigned manually and there is no built in protection against accidentally giving two files the same file number. File numbers should only be given to `*.c` files, **not** `*.h` files. Numbers between 100 and 299 are reserved for files belonging to *Ddr-Z8e*.

Example

File 1:

```
#define DBG_FILE_NUM 5

void main(void)
   {

   // This will always generate an error (unless the
   // code optimizer removes the statement)
#line 3
   DBG_ASSERT_STATIC(1==0);
   }
```

debug.c:

```
unsigned char dbg_StaticFailed (unsigned char FileNum, int LineNum)
{

// FileNum is now == 5 and LineNum is == 3
…
}
```

dbg_Failed()

Module

Debugging

Synopsis

```
#include "debug.h"
unsigned char dbg_Failed (unsigned char FileNum, int LineNum);
```

Return

Depends on implementation

Description

`dbg_Failed()` handles failed `DBG_ASSERT()` statements, i.e. when an assertation fails, `dbg_Failed()` is automatically invoked. `FileNum` specifies the number of the file in which the error occurred. File numbers are defined by `DBG_FILE_NUM`. `LineNum` holds the number of the code line at which the error occurred.

`dbg_Failed()` is located in `debug.c` can be tailored for specific needs. The default implementation only prevents further program execution by entering an endless HALT-loop. Interrupts are though still enabled.

Example

```
// Standard simplistic assertation handler for critical assertations
unsigned char dbg_Failed (unsigned char FileNum, int LineNum)
   {

   // Do nothing
   do
      {
      asm("HALT");
      }
   while(1);

   // In case a newer version of the compiler complains about
   //non-returning value
   return(0);
   }
```

dbg_FailedStatic()

Module

Debugging

Synopsis

```
#include "debug.h"
unsigned char dbg_FailedStatic (unsigned char FileNum, int LineNum);
```

Return

Depends on implementation

Description

`dbg_FailedStatic()` is the handler of a failed `DBG_ASSERT_STATIC()` statements, i.e. when an assertation fails, `dbg_FailedStatic()` is automatically invoked. `FileNum` specifies the number of the file in which the error occurred. File numbers are defined by the `DBG_FILE_NUM` macro. `LineNum` holds the number of the code line at which the error occurred.

`dbg_FailedStatic()`, located in `debug.c`, can be tailored for specific needs. The default implementation only prevents further program execution by entering an endless HALT-loop. Interrupts are though still enabled.

Example

```
// Standard simplistic assertation handler for static assertations
unsigned char dbg_StaticFailed (unsigned char FileNum, int LineNum)
   {

   do
      {
      asm("HALT");
      }
   while(1);

   // In case a newer version of the compiler complains about
   // non-returning value
   return(0);
   }
```

dbg_IllegalInstruction()

Module

Debugging

Synopsis

```
#include "debug.h"
void dbg_IllegalInstruction(void);
```

Return

Description

`dbg_IllegalInstruction()` is an interrupt handler for the Illegal Instruction-trap. This interrupt service routine is invoked when the MCU tries to execute an illegal machine-code instruction. `dbg_IllegalInstruction()` is located in `debug.c` and the interrupt handler's response can be tailored according to various needs. The default implementation of `dbg_IllegalInstruction()` stores the content of the flag-register in register R12. The address of the offending instruction is stored in register RR10.

The invocation of `dbg_IllegalInstruction()` is enabled/disabled in the configuration file `configure.h` (`CFG_DEBUG_ILLEGAL_INSTR`). There is, however, usually no need to enable the debug support for this condition. In cases when the illegal instruction trap is handled by user programs the invocation of `dbg_IllegalInstruction()` should be disabled.

Example

```
#pragma interrupt
void dbg_IllegalInstruction(void)
   {
   IRQ_SETVECTOR(IRQ_TRAP,dbg_IllegalInstruction);

   // Store the flag register in R12
   asm ("   POP R12");

   // Store the program counter in R10 and R11 (RR10)
   asm ("   POP R10");
   asm ("   POP R11");

   // Stop execution
   while(1)
      asm(" HALT");
   }
```

fls_Init()

Module

Flash

Synopsis

```
#include "flash.h"
fls_Init();
```

Return

Description

`fls_Init()` initializes the flash memory controller by setting the programming frequency of the flash memory. Only MCU core frequency of either 18.432 MHz or 20.000 MHz are supported (see `CFG_CPU_FREQUENCY` in `configure.h`). If the controller is using a different frequency, the flash frequency registers has to be set manually in `flash.h`.

Example

```
   …

// Update the flash memory with new content
fls_Init();
fls_UnlockPage((void *)0x0034);
if (fls_IsUnlocked())
   {
   fls_Write ((unsigned char rom *) 0x34, SourceData, 0x04);
   }
fls_Lock();
```

fls_IsUnlocked()

Module

Flash

Synopsis

```
#include "flash.h"
fls_IsUnlocked();
```

Return

Returns a non-zero value (true) when the flash memory controller is unlocked and zero (false) when the controller is locked.

Description

Example

```
// Init and unlock flash controller
fls_Init();
fls_UnlockPage((void *)0x0034);
if (fls_IsUnlocked())
   {
…
```

fls_Lock()

Module

Flash

Synopsis

```
#include "flash.h"
fls_Lock();
```

Return

Description

`fls_Lock()` locks the flash memory controller, thereby preventing any of the flash memory content to be updated or overwritten.

Example

```
unsigned char SourceData = {0,0,0,0};

// Update the flash memory with new content
fls_Init();
fls_UnlockPage((void *)0x0034);
if (fls_IsUnlocked())
   {
   fls_Write ((unsigned char rom *) 0x34, SourceData, 0x04);
   }
fls_Lock();
```

fls_UnlockPage()

Module

Flash

Synopsis

```
#include "flash.h"
fls_UnlockPage(void *FlashMemAddress);
```

Return

Description

`fls_UnlockPage()` unlocks the flash memory controller thereby allowing new content to be written to the flash memory. FlashMemAddress points to a memory address within the flash memory page that should be unlocked. Which specific address, within a given page, FlashMemAddress points out is not important. The entire page, to which the memory address belongs, will be unlocked. When writing flash memory data that is split over two or more pages, care must be taken to handle page boundaries correctly. For example, unlocking page 0 (address 0 – 511) and then writing two bytes starting at address 511 won't work. The first byte will be written but the second will not. That byte should be written to address 512 but that address belongs to the sequential page, which is locked. The correct way to do this is:

- unlock page 0 by calling `fls_UnlockPage((void *) 511)`
- write to address 511
- unlock page 1 by calling `fls_UnlockPage( (void *) 512)` (page 0 will be locked)
- write to address 512

Before the flash memory controller can be unlocked it has to be initialized by `fls_Init()`.

Example

```
// Update the flash memory with new content
fls_Init();
fls_UnlockPage((void *) 0x0034);
if (fls_IsUnlocked())
   {
   fls_Write ((unsigned char rom *) 0x34, SourceData, 0x04);
   …
```

fls_Read()

Module

Flash

Synopsis

```
#include "flash.h"
void fls_Read (unsigned char *Dest, unsigned char rom *Source, unsigned
char Count);
```

Return

Description

`fls_Read()` reads `Count` number of bytes from `Source` and store them in `Dest`. `Source` is a pointer to an address in the program memory (i.e. flash memory) and `Dest` is an address pointer in RAM. It is not necessary to issue a `fls_Init()` macro before reading flash memory.

Example

```
unsigned char Vectors[4];

// Reads the IRQ-vectors for ports C1 and C0
fls_Read (Vectors, (unsigned char rom *) 0x34, 4);
```

fls_Write()

Module

Flash

Synopsis

```
#include "flash.h"
void fls_Write (unsigned char rom *Dest, unsigned char *Source,
unsigned char Count);
```

Return

Description

`fls_Write()` copies `Count` number of bytes from `Source` and store them in `Dest`. `Source` is a pointer to an address in RAM and `Dest` is a pointer to an address in program memory (i.e. flash memory). Before any data can be written to the flash memory the flash memory controller has to be initialized and unlocked.

Example

```
unsigned char SourceData = {0,0,0,0};

// Update the flash memory with new content
fls_Init();
fls_UnlockPage();
if (fls_IsUnlocked())
   {
   fls_Write ((unsigned char rom *) 0x34, SourceData, 0x04);
   }
fls_Lock();
```

i2c_BirqDisable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_BirqDisable();
```

Return

Description

`i2c_BirqDisable()` disables the I2C baud rate generator interrupt.

Example

i2c_BirqEnable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_BirqEnable();
```

Return

Description

`i2c_BirqEnable()` enables the I2C baud rate generator interrupt requests allowing the I2C-controller to be used as a timer. The controller may however **not** be used as timer and for I2C communication simultaneously.

Example

```
// Reset timer
i2c_Reset();
i2c_SetBaudrate(40000); // About 2 ms interrupt timer when 18.432 MHz

// Setup IRQ
irq_SetPriority (IRQ_I2C, IRQ_PRIO_LOW);
irq_Enable (IRQ_I2C);
i2c_BirqEnable();
```

i2c_ClearAbort()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_ClearAbort();
```

Return

Description

`i2c_ClearAbort()` clears an abort condition. The abort condition is a result from an interrupted I2C transaction. The most common reason is that the I2C slave was not ready to participate in a transaction.

Example

```
…

if (i2c_GetStatus() == I2C_STATUS_ABORTED)
   {

   // Handle the aborted transaction
   …

   // Clear condition
   i2c_ClearAbort();
   }

…
```

i2c_Close()

Module

I2C

Synopsis

```
#include "i2c.h"
void i2c_Close(void);
```

Return

Description

`i2c_Close()` shutdown and resets the built-in I2C controller. Before shutting down the controller, `i2c_Close()` will first wait for any ongoing transaction (issued by `i2c_Tranceive()`) to finish. `i2c_Reset()` should be used to unconditionally shut down the IC2 controller.

Example

…

```
// Done
PrintI2cStatus();
ser_Write(SER_PORT, (unsigned char*) "Done\r\n", 6);
ser_Write(SER_PORT, (unsigned char*) NewLine, 2);
ser_Write(SER_PORT, (unsigned char*) NewLine, 2);

// Wait for TX to finish
while (ser_TxIsBusy(SER_PORT))
   ;

// Shut down
i2c_Close();
ser_Close(SER_PORT);
```

…

i2c_Disable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_Disable();
```

Return

Description

`i2c_Disable()` disables the I2C transmitter and receiver leaving the current controller configuration intact.

Example

```
// Change I2C configuration to 40 kHz
i2c_Disable();
i2c_SetBaudRate(I2C_40_KBAUD);
i2c_Enable();
```

i2c_Enable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_Enable();
```

Return

Description

`i2c_Enable()` enables the I2C transmitter and receiver using the current controller configuration.

Example

```
// Change I2C configuration to 40 kHz
i2c_Disable();
i2c_SetBaudRate(I2C_40_KBAUD);
i2c_Enable();
```

i2c_FilterEnable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_FilterEnable();
```

Return

Description

`i2c_FilterEnable()` enables the low pass filter on the SDA and SCL input signals.

Example

```
// Initialize the controller
i2c_Init();

// Change I2C configuration to 500 Hz
i2c_Disable();
i2c_SetBaudRate(9220);
i2c_FilterEnable();
i2c_Enable();
```

i2c_FilterDisable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_FilterDisable();
```

Return

Description

`i2c_FilterDisable()` disables the low pass filter on the SDA and SCL input signals. This is the default setting.

Example

i2c_FlushData()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_FlushData();
```

Return

Description

`i2c_FlushData()` clears the data register of the I2C controller.

Example

```
…
// Write the slave address to the I2C controller
*REG_OFFSET(I2C_0,I2C_DATA) = i2c_SlaveAddress;

// Clears the data register immediately
i2c_FlushData();

…
```

i2c_GetBaudRate()

Module

I2C

Synopsis

```
#include "i2c.h"
unsigned short i2c_GetBaudRate();
```

Return

`i2c_GetBaudRate()` returns the current baud rate setting of the I2C controller.

Description

Example

```
unsigned short BRate;

i2c_Init();
BRate = i2c_GetBaudRate();

// BRate is now equal to the default baud rate,
// which is equal to I2C_100_KBAUD
…
```

i2c_GetSlaveAddrTransResp()

Module

I2C

Synopsis

```
#include "i2c.h"
unsigned char i2c_GetSlaveAddrTransResp();
```

Return

The response from a *Send slave address only*-transaction. Possible values are `I2C_STATUS_ACKN` and `I2C_STATUS_NACKN`.

Description

`i2c_GetSlaveAddrTransResp()` returns the status of a I2C slave when responding to a *Send slave address only*-transaction previously issued by `i2c_Tranceive()`. `i2c_GetSlaveAddrTransResp()` will block until the transaction has finished before returning the status.

Calling `i2c_GetSlaveAddrTransResp()` without previously having issued a *Send slave address only*-transaction is invalid use and will result in a `DBG_ASSERT()` trap.

Example

```
…

// Write some content to eeprom
i2c_Tranceive(SlaveAddr, MemContent, 31, 0);
while (i2c_IsTransBusy())
   ;

// Check if eeprom is busy writing previous transaction
// to memory cells by issuing a slave address transaction
i2c_Tranceive(SlaveAddr, _NULL, 0, 0);

// Get the status. Eeprom not ready => status should be NACKN
Status = i2c_GetSlaveAddrTransResp();

…
```

i2c_GetStatus()

Module

I2C

Synopsis

```
#include "i2c.h"
unsigned char i2c_GetStatus();
```

Return

`i2c_GetStatus()` returns the status of an ongoing I2C transaction. The status can be one of the following:

`I2C_STATUS_UNKNOWN`
The I2C controller is in an unknown condition. This typically happens before the controller has been initialized by `i2c_Init()`.

`I2C_STATUS_IDLE`
The controller is idle and there is no ongoing transaction

`I2C_STATUS_BUSY`
The controller is busy with an ongoing transaction

`I2C_STATUS_ABORTED`
A previous issued transaction has been aborted.

Description

Example

```
void PrintI2cStatus(void)
   {

   // Send I2C-status to serial port
   switch (i2c_GetStatus())
      {
      case I2C_STATUS_UNKNOWN:
         ser_Write(SER_PORT, (unsigned char*)"I2C_STATUS_UNKNOWN",20);
         break;

      case I2C_STATUS_IDLE:
         ser_Write(SER_PORT, (unsigned char*)"I2C_STATUS_IDLE", 17);
         break;

      case I2C_STATUS_BUSY:
         ser_Write(SER_PORT, (unsigned char*)"I2C_STATUS_BUSY", 17);
         break;
```

```
      case I2C_STATUS_ABORTED:
         ser_Write(SER_PORT, (unsigned char*)"I2C_STATUS_ABORTED", 20);
         break;

      default:
         ser_Write(SER_PORT, (unsigned char*)"INVALID I2C STATUS", 20);
         break;
      }
   }
```

i2c_Init()

Module

I2C

Synopsis

```
#include "i2c.h"
void i2c_Init(void);
```

Return

Description

`i2c_Init()` initializes the built-in I2C-controller, the corresponding IO-pins and sets communication parameters to default values, i.e. 100 kbits/s and low pass filter disabled. Controller settings can later on be modified to fit specific needs.

NOTE
The current version (ver. 1.00-RC 1) of the I2C-module only supports 7-bit addressing.

Example

```
// Initialize the controller
i2c_Init();

// Change I2C configuration to 500 Hz
i2c_Disable();
i2c_SetBaudRate(9220);
i2c_FilterEnable();
i2c_Enable();
```

i2c_IrqTxEnable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_IrqTxEnable();
```

Return

Description

`i2c_IrqTxEnable()` enables the transmitter interrupt because the data register is empty, potentially needing more data written to it for transmission.

Example

```
…
// Start the transaction
i2c_SendStart ();

// Check if to enable TX-interrupt
if ((i2c_SlaveAddress & I2C_SL_ADDR_READ) == 0)
   i2c_IrqTxEnable();

// Check if it is necessary to set the NACKN-bit owing to
// this beeing a 1-byte short RX
else if (i2c_RxByteCount == 1)
   i2c_SendNackn ();
…
```

i2c_IrqTxDisable()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_IrqTxDisable();
```

Return

Description

`i2c_IrqTxDisable()` disables the I2C transmitter interrupt.

Example

```
…
// Send slave address only transaction
if ((i2c_TxByteCount + i2c_RxByteCount) == 0)
   {

   // Set transaction type.
   i2c_TransactionType = I2C_TRANS_SLAVE_ADDR;

   // End slave address transaction accordingly
   i2c_IrqTxDisable ();
   i2c_SendStartStop ();
   return;
   }
…
```

i2c_IsHwBusy()

Module

I2C

Synopsis

```
#include "i2c.h"
unsigned char i2c_IsHwBusy();
```

Return

I2C hardware busy status

Description

`i2c_` IsHwBusy `()` returns a non zero value (true) when the I2C hardware is busy transmitting or receiving data. 0 is returned when the hardware is idle.

The busy status is however only from a hardware perspective. There may still be an ongoing transaction in progress issued from software.

Example

…

```
i2c_Tranceive(SlaveAddr, MemContent, 0, 37);

// Wait for the transaction to complete
while(i2c_IsTransBusy())
   ;

// Wait for the hardware to finish up
while (i2c_IsHwBusy())
   ;

// Shut down I2C controller
i2c_Close();
```

…

i2c_IsTransBusy()

Module

I2C

Synopsis

```
#include "i2c.h"
unsigned char i2c_IsTransBusy();
```

Return

Transaction busy status

Description

`i2c_IsTransBusy()` returns a non zero value (true) when there is an ongoing I2C transaction, previously issued by `i2c_Tranceive()`. 0 is returned when there is no ongoing transaction and a new call to i2c_Tranceive will not block. `i2c_IsTransBusy()` is used for synchronizing sequential transactions preventing potential race conditions.

The busy status is however only from a software perspective. `i2c_IsTransBusy()` can return 0 (idle) but the I2C hardware may still be sending the last byte of a previous transaction.

Example

```
…

i2c_Tranceive(SlaveAddr, MemContent, 0, 37);

// Wait for the transaction to complete
while(i2c_IsTransBusy())
   ;

// Wait for the hardware to finish up
while (i2c_IsHwBusy())
   ;

// Shut down I2C controller
i2c_Close();

…
```

i2c_Reset()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_Reset();
```

Return

Description

`i2c_Reset()` resets the I2C controller to a known default state. This means:

- Data register cleared, i.e. set to 0 (zero)
- I2C transmitter and receiver disabled
- Baud rate interrupt disabled
- Transmit interrupt disabled
- Low pass signal filer disabled
- Baud rate set to 0xFFFF

Example

```
// Reset the i2c-controller.
i2c_Reset();

// Initialize the SCL/SDA-ports
io_SetAltFunct(IO_A, IO_A_ALT_I2C);

// Default operation parameters
i2c_SetBaudRate(I2C_100_KBAUD);

// Initialzie variables
i2c_Buffer = _NULL;
i2c_TxByteCount = 0;
```

i2c_SendNackn()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_SendNackn();
```

Return

Description

`i2c_SendNackn()` issues a Not Acknowledge condition on the I2C bus after the next data byte has been read from the I2C slave.

Example

…

```
// Start the transaction
i2c_Status = I2C_STATUS_BUSY;
i2c_SendStart();

// Check if to enable TX-interrupt
if ((i2c_SlaveAddress & I2C_SL_ADDR_READ) == 0)
   i2c_IrqTxEnable();

// Check if it is necessary to set the NACKN-bit owing to
// this being a 1-byte short RX
else if (i2c_RxByteCount == 1)
   i2c_SendNackn();
```

…

i2c_SendStart()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_SendStart();
```

Return

Description

`i2c_SendStart()` sends a start condition on the I2C bus. Once issued the start condition will be cleared after the start condition has been sent. If the both the data register and I2C shift registers are empty the start condition will not be transmitted until new data is written to the data register.

Example

```
…
// Write the slave address to the I2C controller
*REG_OFFSET(I2C_0,I2C_DATA) = i2c_SlaveAddress;

// Start the transaction
i2c_Status = I2C_STATUS_BUSY;
i2c_SendStart();
…
```

i2c_SendStartStop()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_SendStartStop();
```

Return

Description

`i2c_SendStartStop()` sets both the start and stop bits of the i2C controller. This is useful e.g. for slave address only transactions.

Example

…

```
// Write the slave address to the I2C controller
*REG_OFFSET(I2C_0,I2C_DATA) = i2c_SlaveAddress;

// Send slave address only transaction
i2c_SendStartStop();
```

…

i2c_SendStop()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_SendStop();
```

Return

Description

`i2c_SendStop()` sends a stop condition on the I2C bus after the data in the I2C shift register has been transmitted or after a byte has been received.

Example

```
…
// Ongoing RX transaction that ends
if (i2c_RxByteCount == 1)
   {

   // Read the last byte received.
   *i2c_Buffer = *REG_OFFSET(I2C_0,I2C_DATA);
   i2c_Buffer++;
   i2c_RxByteCount--;

   // End RX transaction
   if (i2c_TxByteCount == 0)
      {
      i2c_SendStop ();
…
```

i2c_SetBaudRate()

Module

I2C

Synopsis

```
#include "i2c.h"
i2c_SetBaudRate(unsigned short BaudRate);
```

Return

Description

`i2c_SetBaudRate()` sets the baud rate registers of the I2C controller. This rate is dependent of the system clock. The following predefined rates are supported for 20 MHz CPU frequency as well as 18.432 MHz:

```
I2C_10_KBAUD
I2C_40_KBAUD
I2C_100_KBAUD
I2C_400_KBAUD
```

For other clock frequencies than the supported ones above, the baud rate values have to be calculated according the Z8 Encore! documentation. The core CPU frequency is set in the `configure.h` configuration file using the parameter `CFG_CPU_FREQUENCY`.

Example

```
// Change I2C configuration to 40 kHz
i2c_Disable();
i2c_SetBaudRate(I2C_40_KBAUD);
i2c_Enable();
```

i2c_Tranceive()

Module

I2C

Synopsis

```
#include "i2c.h"

void i2c_Tranceive (unsigned char SlaveAddress, unsigned char *Buffer,
unsigned short TxByteCount, unsigned short RxByteCount);
```

Return

Description

`i2c_Tranceive()` starts a non-blocking transaction over the I2C bus. `SlaveAddress` identifies the I2C slave device to communicate with, `Buffer` is the address where data to transmit (write) and receive (read) is stored, `TxByteCount` is the number of bytes to transmit in the transaction and `RxByteCount` is the number of bytes to receive. `i2c_Tranceive()` only supports 7 bits slave addressing.

Since `Buffer` should hold data to write as well as data to read it has to be large enough to hold both, i.e. `Buffer` must have a length of at least `TxByteCount` + `RxByteCount`. Further more, since the transaction is non-blocking, `Buffer` has to be allocated and its integrity must be intact for the entire lifetime of the transaction. The following code snippet is an example of how **not** to do:

```
void MyFync(void)
   {
   unsigned char MySlaveAddr;
   unsigned short MyTxCnt;
   unsigned short MyRxCnt;
   volatile unsigned char MyBuffer[10];

   …

   i2c_Tranceive (MySlaveAddr, MyBuffer, MyTxCnt, MyRxCnt);
   return;
   }
```

In this example there is a race condition. `i2c_Tranceive()` returns immediately after it has started the transaction but very likely before the transaction has ended. When `MyFunc()` returns at the next statement `Buffer` is freed since it's allocated on the stack, being a local variable. `i2c_Tranceive()` will consequently use an invalid buffer for the remaining lifetime of the I2C transaction. To safely check when the transaction has finished use `i2c_IsTransBusy()`.

Before a transaction can start `Buffer` must hold the data to transmit starting at index `[0]` (zero) and ending at index `[TxByteCount – 1]`. When a transaction has finished, received data is stored in the buffer starting at index `[TxByteCount]` and ending at `[TxByteCount + RxByteCount – 1]`. Any data previously stored between the indices `[TxByteCount]` and `[TxByteCount + RxByteCount – 1]` will be overwritten during a transaction. The `Buffer` after a valid transaction will have the following content:

TxData [0 to TxByteCount – 1]	RxData [TxByteCount to TxByteCount + RxByteCount – 1]

Transmitted data will thus be the same as before the transaction but received data will not.

Five different transaction types are supported:

- Send slave address only
- Write only
- Read only
- Write – Read
- Read – Write

All transactions start with a START-condition and end with a STOP-condition. The Write – Read and Read – Write transactions are special in the meaning that no STOP-condition is sent going from writing data to reading data and vice versa. Instead a repeated START-condition is used to switch between the two. The entire transaction as such is, however, still ended by a STOP-condition.

Which kind of transaction to perform is for the first four types determined by the parameters `TxByteCount` and `RxByteCount` according to the table below.

Value of `TxByteCount`	**Value of `RxByteCount`**	**Transaction type**
= 0	= 0	Send slave address only
> 0	= 0	Write only
= 0	> 0	Read only
> 0	> 0	Write - Read

To perform a Read – Write transaction the read bit (defined as constant `I2C_SL_ADDR_READ`) of the slave address also has to be set in addition to `TxByteCount` and `RxByteCount` both being > 0.

Example

```
void WriteReadEeprom()
   {
   volatile unsigned char SlaveAddr = CONTROL_CODE | EEROM_ADDR;
   volatile unsigned char MemAddress[2] = {0x00,0x00};
   volatile unsigned char MemContent[] =
      "  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";


   // Initialize the controller
   i2c_Init();

   // Specify the memory address of the eeprom to access
   MemContent[0] = MemAddress[0];
   MemContent[1] = MemAddress[1];
```

```
// Write 2 bytes (addr to read from)
// and read 37 bytes from that address
i2c_Tranceive(SlaveAddr, MemContent, 2, 37);

// Wait for the i2c transaction to finish
// MonitorI2c();
while (i2c_IsTransBusy())
   ;

// Do something useful with the data
…

// Shut down
i2c_Close();
}
```

io_ClearAltFunct()

Module

Digital IO

Synopsis

```
#include "io.h"
io_ClearAltFunct(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_ClearAltFunct()` disables the alternate functions of port pins `BitMask` in port `PortID` and restore the associated port pins to standard digital I/O operation. Several alternate functions can be cleared at the same time for a given port by merging the bit masks with a bitwise OR operation. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

There are predefined bit masks for all alternate functions. Some defined bit masks do however clear the alternate functions for several port pins where so is motivated. These are marked with an asterisk below:

Port `IO_A`

```
IO_A_ALT_TMR_0_IN
IO_A_ALT_TMR_0_OUT
IO_A_ALT_UART_0_DE
IO_A_ALT_UART_0_CTS
IO_A_ALT_UART_0      *
IO_A_ALT_I2C         *
```

Port `IO_B`

```
IO_B_ALT_AD_0
IO_B_ALT_AD_1
IO_B_ALT_AD_2
IO_B_ALT_AD_3
IO_B_ALT_AD_4
IO_B_ALT_AD_5
IO_B_ALT_AD_6
IO_B_ALT_AD_7
```

Port `IO_C`

```
IO_C_ALT_TMR_1_IN
IO_C_ALT_TMR_1_OUT
IO_C_ALT_SPI          *
IO_C_ALT_TMR_2_IN
IO_C_ALT_TMR_2_OUT
```

Port `IO_D`

```
IO_D_ALT_TMR_3_IN
IO_D_ALT_TMR_3_OUT
IO_D_ALT_UART_1_DE
IO_D_ALT_UART_1_CTS
IO_D_ALT_UART_1       *
IO_D_ALT_WDT
```

Port `IO_H`

```
IO_H_ALT_AD_8
IO_H_ALT_AD_9
IO_H_ALT_AD_10
IO_H_ALT_AD_11
```

NOTE
Not all IO-ports and all port- pins have alternate functions.

Example

```
#include "binary.h"
…
void main(void)
   {

   // Reset port IO_A (not necessary but a good habit)
   io_Reset(IO_A);

   // Enable the I2C protocol on port IO_A (IO_A_ALT_I2C == b11000000)
   io_SetAltFunct(IO_A, IO_A_ALT_I2C);

   // Use the I2C-bus

   // Restore the port pins to operate as standard digital IO-pins
   io_ClearAltFunct(IO_A, IO_A_ALT_I2C);
   }
```

io_ClearBits()

Module

Digital IO

Synopsis

```
#include "io.h"
io_ClearBits(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_ClearBits()` clears (drives low) the port pins `BitMask` for output port `PortID`. A 1 means that the corresponding port-pin is driven low. A 0 (zero) means that the port-pin is left unchanged. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Configure all IO-pins in port IO_B as output pins.
io_SetDataDirOut(IO_B, b11111111);

// Write 0x00 to port IO_B
io_Outp(IO_B,b00000000);

// All of the pins in IO_B is driven low

// Set bits 2 and 7 to 1 for output port IO_B
io_SetBits(IO_B, b10000100);

// IO_B is now driving pins 2 and 7 high

// Clear bit 2 in port IO_B
io_ClearBits(IO_B, b00000100);

// IO_B is now only driving bit 7 high
```

io_ClearHiCurrent()

Module

Digital IO

Synopsis

```
#include "io.h"
io_ClearHiCurrent(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_ClearHiCurrent()` disables the ability of port pins `BitMask` in port `PortID` to drive extra current. A 1 means that the corresponding port-pin is configured for normal drive mode. A 0 (zero) means that the port-pin is left unchanged. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Enable bit 0 in port IO_A to drive extra current
io_SetHiCurrent(IO_A, b00000001);

// Restore bit 0 in port IO_A to nominal current level.
io_ClearHiCurrent(IO_A, b00000001);
```

io_ClearOpenDrain()

Module

Digital IO

Synopsis

```
#include "io.h"
io_ClearOpenDrain(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_ClearOpenDrain()` restores the port pins `BitMask` in port `PortID` to normal non-open drain operation mode. A 1 means that the corresponding port-pin is configured non-open drain mode. A 0 (zero) means that the port-pin is left unchanged. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Enable bit 0 in port IO_A to operate in open drain mode
io_SetOpenDrain(IO_A, b00000001);

// Restore bit 0 in port IO_A to normal mode
io_ClearOpenDrain(IO_A, b00000001);
```

io_ClearStopMod()

Module

Digital IO

Synopsis

```
#include "io.h"
io_ClearStopMod(Reg_addr PortID, unsigned char BitMask)
```

Return

Description

`io_ClearStopMod()` disables port pins `BitMask` in port `PortID` to be used as sources of stop mode recovery. A 1 means that the corresponding port-pin is cleared from stop mode operation mode. A 0 (zero) means that the port-pin is left unchanged. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Sets bit 0 and 7 in port IO_A to be used as a stop mode recovery
// source. I.e. when the MCU is in stop mode and pins 0 or 7 in port
// IO_A sense a transition, the MCU is awaken from stop mode.
io_SetStopMod(IO_A, b10000001);

// Disable bit 0 in port IO_A to be used for stop mode recovery.
io_ClearStopMod(IO_A, b00000001);

// Now only pin 7 works as stop mode recovery source for IO_A
```

io_GetAltFunct()

Module

Digital IO

Synopsis

```
#include ""
unsigned char io_GetAltFunct(reg_addr PortID);
```

Return

The alternate function settings for port `PortID`.

Description

`io_GetAltFunct()` returns the alternate function configuration for the 8 port pins in `PortID`. A 1 means that the corresponding port-pin is configured for its alternate function. A 0 (zero) means that the port-pin is configured for standard digital I/O. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

unsigned char AltFunctConfig = 0;

// Enable analog input 0 in port IO_B (IO_B_ALT_AD_0 == b00000001)
io_SetAltFunct(IO_B, IO_B_ALT_AD_0);

// Get the alternate function configuration for port IO_B
AltFunctConfig = IO_GetAltFunct(IO_B);

// AltFunctConfig is now == IO_B_ALT_AD_0 (b00000001) unless port IO_B
// has been configuredfor other alternate functions as well else where
// in the code.
```

io_GetDataDir()

Module

Digital IO

Synopsis

```
#include "io.h"
unsigned char io_GetDataDir(reg_addr PortID);
```

Return

The input/output configurations for the port pins of port `PortID`.

Description

`io_GetDataDir()` returns the input/output configuration for the 8 port pins in `PortID`. A 1 (one) means that the corresponding port pin is configured for output. A 0 (zero) means that the corresponding port pin is configured for input. Default setting is that all pins of a port are configured for input. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

unsigned char PinConfig = 0;

// Configure pins/bits 0-3 for output and pins 4-7 for
// input in port IO_A
io_SetDataDirOut(IO_A, b00001111);
io_SetDataDirIn(IO_A, b11110000); // Not necessary after io_Reset()

// Get the pin configuration for port IO_A
PinConfig = io_GetDataDir(IO_A);

// PinConfig is now == b00001111
```

io_GetHiCurrent()

Module

Digital IO

Synopsis

```
#include "io.h"
unsigned char io_GetHiCurrent(reg_addr PortID);
```

Return

The hi-current configurations for the port pins of port `PortID`.

Description

`io_GetHiCurrent()` returns the hi-current configuration for the port pins in `PortID`. A 1 (one) means that the corresponding port pin is configured for hi-current operation. A 0 (zero) means that the corresponding port pin is configured for low current operation. Default setting is that all pins in a port are configured for low current operation. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

unsigned char CurrentConfig = 0;

// Configure pins/bits 0-3 for high-current output
io_SetDataDirOut(IO_A, b00001111);
io_SetHiCurrent(IO_A, b00001111);

// Get the pin configuration for port IO_A
CurrentConfig = IO_GET_HI_CURRENT(IO_A);

// CurrentConfig is now == b00001111 unless the
// port has been differently configured elsewhere
// in the code
```

io_GetOpenDrain()

Module

Digital IO

Synopsis

```
#include "io.h"
unsigned char io_GetOpenDrain(reg_addr PortID);
```

Return

The open-drain configurations for the port pins of port `PortID`.

Description

`io_GetOpenDrain()` returns the open-drain configuration for the port pins in `PortID`. A 1 (one) means that the corresponding port pin is configured for open-drain operation. A 0 (zero) means that the corresponding port pin is configured for normal mode operation. Default setting is that all pins in a port are configured for normal mode operation. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

unsigned char DrainConfig = 0;

// Configure pins/bits 0-3 for open-drain in port IO_A
io_SetDataDirOut(IO_A, b00001111);
io_SetOpenDrain(IO_A, b00001111);

// Get the pin configuration for port IO_A
DrainConfig = io_GetOpenDrain(IO_A);

// DrainConfig is now == b00001111 provided that the port
// hasn't been configured differently elsewhere in the code
```

io_GetStopMod()

Module

Digital IO

Synopsis

```
#include "io.h"
unsigned char io_GetStopMod(reg_addr PortID)
```

Return

The stop mode recovery configurations for the port pins of port `PortID`.

Description

`io_GetStopMod()` returns the *Stop Mode Recovery* configuration for the port pins in `PortID`. A 1 (one) means that the corresponding port pin is enabled as a source for *Stop Mode Recovery*. A 0 (zero) means that the corresponding port pin is disabled as a source for *Stop Mode Recovery*. Default setting is that all pins in a port are disabled. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

unsigned char StopModeConfig = 0;

// Configure all IO-pins in port IO_A as input pins.
// (not really necessary since this is default)
io_SetDataDir(IO_A, b00000000);

// Configure pins/bits 4-7 in port IO_A as stop mode recovery sources
io_SetStopMod(IO_A, b11110000);

// Get the pin configuration for port IO_A
StopModeConfig = io_GetStopMod(IO_A);

// StopModeConfig is now == b11110000 unless IO_A
// has been differently configured elsewhere
```

io_Inp()

Module

Digital IO

Synopsis

```
#include "io.h"
unsigned char io_Inp(reg_addr PortID);
```

Return

The input value of port `PortID`.

Description

`io_Inp()` reads and returns the input value of port `PortID`. Note that only port pins configured as input pins are sampled. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

unsigned char PortInput;

// Configure all IO-pins in port IO_A as input pins.
// Not really necessary since this is default after io_Reset()
io_SetDataDirIn(IO_A, b11111111);

// Read the input value form port IO_A
PortInput = io_Inp(IO_A);
```

io_Outp()

Module

Digital IO

Synopsis

```
#include "io.h"
io_Outp(reg_addr PortID, unsigned char Byte)
```

Return

Description

`io_Outp()` writes the value of `Byte` to the port `PortID`. Only port-pins configured for output are written. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Configure all IO-pins in port IO_B as output pins.
io_SetDataDirOut(IO_B, b11111111);

// Write 0x11 to port IO_B
io_Outp(IO_B,0x11);
```

io_SetAltFunct()

Module

Digital IO

Synopsis

```
#include "io.h"
io_SetAltFunct(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_SetAltFunct()` configures the alternate functions of port pins `BitMask` for port `PortID`. A 1 (one) in `BitMask` enables the corresponding alternate function whereas a 0 (zero) disables it. Pins that are not specified by `BitMask` are left unchanged. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

There are predefined bit masks for all alternate functions. Some defined bit masks do however set the alternate functions for several port pins where so is motivated. These are marked with an asterisk below:

Port `IO_A`

```
IO_A_ALT_TMR_0_IN
IO_A_ALT_TMR_0_OUT
IO_A_ALT_UART_0_DE
IO_A_ALT_UART_0_CTS
IO_A_ALT_UART_0      *
IO_A_ALT_I2C         *
```

Port `IO_B`

```
IO_B_ALT_AD_0
IO_B_ALT_AD_1
IO_B_ALT_AD_2
IO_B_ALT_AD_3
IO_B_ALT_AD_4
IO_B_ALT_AD_5
IO_B_ALT_AD_6
IO_B_ALT_AD_7
```

Port `IO_C`

```
IO_C_ALT_TMR_1_IN
IO_C_ALT_TMR_1_OUT
IO_C_ALT_SPI          *
IO_C_ALT_TMR_2_IN
IO_C_ALT_TMR_2_OUT
```

Port `IO_D`

```
IO_D_ALT_TMR_3_IN
IO_D_ALT_TMR_3_OUT
IO_D_ALT_UART_1_DE
IO_D_ALT_UART_1_CTS
IO_D_ALT_UART_1       *
IO_D_ALT_WDT
```

Port `IO_H`

```
IO_H_ALT_AD_8
IO_H_ALT_AD_9
IO_H_ALT_AD_10
IO_H_ALT_AD_11
```

NOTE
Not all port pins have alternate functions. Enabling a non-existent alternate function may result in unpredictable behavior of the MCU.

Example

```
// Reset port IO_A
io_Reset(IO_A);

// Enable the pins for RS-232 communication in port IO_A
// using RTS/CTS handshaking.
io_SetAltFunct(IO_A, IO_A_ALT_UART_0 | IO_A_ALT_UART_0_CTS);
```

io_SetBits()

Module

Digital IO

Synopsis

```
#include "io.h"
io_SetBits(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_SetBits()` drives the output bits in `BitMask` high in port `PortID`. If the bits are not configured for output the macro doesn't have any effect. Bits that are not specified by `BitMask` are left unchanged. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Configure all IO-pins in port IO_B as output pins.
io_SetDataDirOut(IO_B, b11111111);

// Write 0x00 to port IO_B
io_Outp(IO_B,b00000000);

// All of the pins in IO_B is driven low

// Set bits 2 and 7 to 1 for output port IO_B
io_SetBits(IO_B, b10000100);

// Set bit number 0 as well but leave the rest intact
io_SetBits(IO_B, b00000001);

// IO_B is now driving pins 0,2 and 7 high
```

io_SetDataDirIn()

Module

Digital IO

Synopsis

```
#include "io.h"
io_SetDataDirIn(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_SetDataDirIn()` specifies which pins in port `PortID` should be configured as input pins. A 1 in `BitMask` specifies that the corresponding port pin should be an input pin. Pins that are not specified by `BitMask` are left unchanged. Default settings for all port pins are to be configured as input-pins. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

unsigned char DataIn;

// Configure IO-pins 0-3 in port IO_B as output pins.
io_SetDataDirOut(IO_B, b00001111);

// Configure IO-pins 4-7 in port IO_B as input pins
io_SetDataDirIn(IO_B, b11110000);

// Write 0x08 to port IO_B
io_Outp(IO_B,b00001000);

// Read 4 bits from IO_B
DataIn = io_Inp(IO_B);

// The 4 bits are now stored in DataIn in bits 4-7
```

io_SetDataDirOut()

Module

Digital IO

Synopsis

```
#include "io.h"
io_SetDataDirOut(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_SetDataDirOut()` specifies which pins in port `PortID` should be configured as output pins. A 1 in `BitMask` specifies that the corresponding port pin should be an output pin. Pins that are not specified by `BitMask` are left unchanged. Default settings for all port pins are to be configured as input-pins. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Configure IO-pins 0-3 in port IO_B as output pins.
io_SetDataDirOut(IO_B, b00001111);

// Write 0x09 to port IO_B
io_Outp(IO_B,b00001001);
```

io_SetHiCurrent()

Module

Digital IO

Synopsis

```
#include "io.h"
io_SetHiCurrent(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_SetHiCurrent()` specifies which pins in port `PortID` that should be configured for high current drive. A 1 in `BitMask` specifies that the corresponding port pin should be a high-current drive pin. Pins that are not specified by `BitMask` are left unchanged. Default settings for all port pins are to be configured as normal current driving pins. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
#include "binary.h"

…

// Configure all IO-pins in port IO_B as output pins
io_SetDataDirOut(IO_B, b11111111);

// Configure IO-pin 3 in port IO_B as high current pin
io_SetHiCurrent(IO_B, b00001000);

// Configure IO-pin 6 in port IO_B as high current as well
io_SetHiCurrent(IO_B, b01000000);

// Now both pin 3 and pin 6 are high current pins
```

io_SetOpenDrain()

Module

Digital IO

Synopsis

```
#include "io.h"
io_SetOpenDrain(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_SetOpenDrain()` specifies which pins in port `PortID` that should be configured for open-drain mode. A 1 in `BitMask` specifies that the corresponding port pin should be an open-drain pin. Pins that are not specified by `BitMask` are left unchanged. Default settings for all port pins are to be configured for non-open-drain mode. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
// Configure IO-pin 3 in port IO_B as open-drain pin
io_SetOpenDrain(IO_B, b00001000);

// Configure IO-pin 6 in port IO_B as open-drain pin
io_SetOpenDrain(IO_B, b01000000);

// Now both pin 3 and pin 6 are open-drain pins
```

io_SetStopMod()

Module

Digital IO

Synopsis

```
#include "io.h"
io_SetStopMod(reg_addr PortID, unsigned char BitMask);
```

Return

Description

`io_SetStopMod()` specifies which pins in port `PortID` that should be configured for triggering the microcontroller to recover from *Stop Mode*. A 1 in `BitMask` specifies that the corresponding port pin should function as a trigger. Pins that are not specified by `BitMask` are left unchanged. Default setting for all port pins are not to be configured as *Stop Mode* recovery triggers. Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
…

// Configure IO-pins 0 and 3 in port IO_B as Stop Mode recovery
// triggers
io_SetdataDirIn(IO_B, b00001001);
io_SetStopMod(IO_B, b00001001);

// Any high to low or low to high transition on pins
// 0 and 3 in port IO_B will now trigger the MCU to recover from a
// Stop Mode state
```

io_Reset()

Module

Digital IO

Synopsis

```
#include "io.h"
void io_Reset (reg_addr PortID);
```

Return

Description

`io_Reset()` resets port `PortID` to a known and predetermined state. This means that:

- All port pins are configured as input pins
- All alternate functions are disabled
- All pins are set to non-open-drain mode
- All pins are disabled as sources for *Stop Mode Recovery*
- All pins are set to normal current operation, i.e. low
- Output data register is set to 0x00
- Address register is set to 0x00
- Control register is set to 0x00

Valid values for `PortID` are:

```
IO_A
IO_B
IO_C
IO_D
IO_E
IO_F
IO_G
IO_H
```

Example

```
// Reset port IO_A
io_Reset(IO_A);
```

IRQ_CLEAR()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
IRQ_CLEAR(Vector);
```

Return

Description

`IRQ_CLEAR()` is a macro that clears the interrupt request bit in the interrupt controller that corresponds to the interrupt source `Vector`. This is usually not necessary for Z8 Encore! hardware revisions 642 and later since the interrupt will automatically be cleared by the MCU itself. Valid values for `Vector` are:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

NOTE
`IRQ_CLEAR()` translates to a single assembler statement in accordance with the Z8 Encore! specification.

Example

```
#pragma interrupt
void InterruptHandler(void)
   {

   // Set IRQ vector
   IRQ_SET_VECTOR(IRQ_TMR_0, InterruptHandler);

   // Process interrupt
   …

   // Clear IRQ and return
   IRQ_CLEAR(IRQ_TMR_0);
   }
```

irq_Di()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
irq_Di();
```

Return

Description

`irq_Di()` disables the interrupt controller on a global basis. As a consequence, no interrupt requests will be serviced by the MCU during this time. This can be useful e.g. for protecting a code segment that needs to be executed without interruption from other competing tasks. The interrupt controller can be re-enabled again by `irq_Ei()`. Interrupt requests that are issued after a call to `irq_Di()` are in a pending stage and will be serviced once `irq_Ei()` has been executed.

Example

```
volatile unsigned short Ticker = 0;

#pragma interrupt
void TimerInterruptHandler(void)
   {

   // Set IRQ vector
   IRQ_SET_VECTOR(IRQ_TMR_0, TimerInterruptHandler);

   // Increment the ticker
   Ticker++;
   }

void DelayMs  (unsigned short Delay)
   {

   // The Ticker variable can't be guaranteed to be set to zero
   // in an atomic way, i.e. using exactly one assembler instruction
   // only. Therefore, accidental invocation of the timer interrupt
   // handler must be not be allowed during the assignment to prevent
   // a race condition.
   irq_Di();
   Ticker = 0;
   irq_Ei();

   while (Ticker < Delay)
      ;
   }
```

```
void main(void)
   {
   unsigned short Delay = 0x1fff;

   // Initialize and start the ticker timer (1 IRQ/1 ms on 18.432MHz)
   tmr_SetMode(TMR_0, TMR_MODE_CONTINUE);
   tmr_SetPrescaler(TMR_0, TMR_PRES_DIV_8);
   tmr_SetCounter(TMR_0, 2304);
   tmr_SetReload(TMR_0, 2304);
   irq_Enable (IRQ_TMR_0);
   tmr_Enalbe(TMR_0);

   // Delay 1 ms
   DelayMs (1);

   // Continue with the program
   …
   }
```

irq_Disable()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
irq_Disable (unsigned short Irq);
```

Return

Description

`irq_Disable()` disables the particular interrupt specified by `Irq`. `irq_Disable()` is generally used as a way to turn off interrupt request responses for a specific hardware device when it has finished a task, e.g. a write transaction over the I2C-bus. Valid values for `Irq` are:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

NOTE
`irq_Disable()` is **not** well suited for temporary protecting code segments from concurrent access or execution. These situations are much better handled by `irq_Di()` and `irq_Ei()`.

Example

```
#pragma interrupt
void ser_TxIrqHandler0(void)
   {

   // Set vector for this uart
   IRQ_SET_VECTOR(IRQ_UART_0_TX,ser_TxIrqHandler0);

   // Check if there is more data to send
   if (MoreData())
      {

      // Send next data byte
      *REG_OFFSET(SER_0,SER_TX_DATA) = GetNextByte();
      }
   else
      {

      // End transmission
      irq_Disable (IRQ_UART_0_TX);
      }
   }
```

irq_Ei()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
irq_Ei();
```

Return

Description

`irq_Ei()` enables, on a global basis, interrupt requests to be serviced by the interrupt controller. This is usually used to handle concurrency issues.

Example

```
volatile unsigned short Ticker = 0;

#pragma interrupt
void TimerInterruptHandler(void)
   {

   // Set IRQ vector
   IRQ_SET_VECTOR(IRQ_TMR_0, TimerInterruptHandler);

   // Increment the ticker
   Ticker++;
   }


void DelayMs  (unsigned short Delay)
   {

   // The Ticker variable can't be guaranteed to be set to zero
   // in an atomic way, i.e. using exactly one assembler instruction
   // only. Therefore, accidental invocation of the timer interrupt
   // handler must be not be allowed during the assignment to prevent
   // a race condition.
   irq_Di();
   Ticker = 0;
   irq_Ei();

   while (Ticker < Delay)
      ;
   }
```

```
void main(void)
   {
   unsigned short Delay = 0x1fff;

   // Initialize and start the ticker timer (1 IRQ/1 ms on 18.432MHz)
   tmr_SetMode(TMR_0, TMR_MODE_CONTINUE);
   tmr_SetPrescaler(TMR_0, TMR_PRES_DIV_8);
   tmr_SetCounter(TMR_0, 2304);
   tmr_SetReload(TMR_0, 2304);
   irq_Enable (IRQ_TMR_0);
   tmr_Enalbe(TMR_0);

   // Delay 1 ms
   DelayMs (1);

   // Continue with the program
   …
   }
```

irq_Enable()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
void irq_Enable (unsigned short Irq);
```

Return

Description

`irq_Enable()` enables the particular interrupt specified by `Irq`. This function only enables interrupts at the interrupt controller level. For some hardware devices, e.g. the RS-232 controllers, it is necessary to enable interrupt generation at the device level as well. Valid values for `Irq` are:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

Example

```
#pragma interrupt
void ser_RxIrqHandler(void)
   {

   // Set vector for this isr
   IRQ_SET_VECTOR(IRQ_UART_1_RX,ser_RxIrqHandler);

   // Service interrupt
   …
   }

void main(void)
   {
   // Change the priority to low priority
   irq_SetPriority (IRQ_UART_1_RX, IRQ_PRIO_LOW);

   // Enable the IRQ-bit in the interrupt controller for
   // the RS232 controller 1 receiver interrupt
   irq_Enable(IRQ_UART_1_RX);

   // Enable receiver interrupt in the RS232 controller
   ser_IrqRxEnable(SER_1);

   // Enable the RS232 receiver
   ser_RxEnable(SER_1);

   …
   }
```

irq_GetEdge()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
unsigned char irq_GetEdge(unsigned char BitMask);
```

Return

The setting of the interrupt edge register for the bit(s) specified by `BitMask`.

Description

The port pins of ports `IO_A` and `IO_D` can be configured to generate interrupt requests when their levels change from high to low or vice versa. `irq_GetEdge()` returns the trigger configuration for these port pins. The `BitMask` parameter specifies which bits (0 - 7) the edge configuration should be returned for. An edge setting of one (1) means that the corresponding bit in `IO_A` or `IO_D` issues an interrupt request when its input voltage level goes from low to high. A setting of zero (0) means that the interrupt request is issued when the input level goes from high to low.

Example

```
#include "binary.h"

…

#pragma interrupt
void InterruptHandler(void)
   {

   // Set IRQ vector
   IRQ_SET_VECTOR(IRQ_A3_D3, InterruptHandler);

   // Service interrupt
   …
   }


void main(void)
   {
   unsigned char EdgeSetting;

   // Reset port D. All bits set as input by default
   io_Reset (IO_D);

   // Use bit number 3 in port IO_D as interrupt source.
   // (The other bits use port IO_A as interrupt sources)
   irq_SetIoSource(b00001000);
```

```
// Use the rising edge (low to high voltage transition) as
// interrupt trigger for bit 3
// (The other bits use high to low transition)
irq_SetEdge(b00001000);

// Enable interrupt
irq_Enable(IRQ_A3_D3);

// Read the edge-setting for bit number 3 and bit number 4
EdgeSetting = irq_GetEdge(b00011000);

// EdgeSetting is now == b00001000 unless the edge setting
// also has been set in code elsewhere
}
```

irq_GetIoSource()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
unsigned char irq_GetIoSource(unsigned char BitMask);
```

Return

The interrupt source pin configuration for ports `IO_A` and `IO_D`

Description

`irq_GetIoSource()` returns which port pins of ports `IO_A` and `IO_D` that are configured as interrupt request sources. A 0 (zero) specifies that the corresponding pin number for port `IO_A` can be used as interrupt request source and a 1 (one) selects the same pin but for `IO_D` as interrupt request source. Default configuration is that only port IO_A can be used as interrupt source (all bits 0 – 7 set to zero). The parameter `BitMask` specifies which port pins (0-7) to retrieve the configuration for.

Example

```
#include "binary.h"

…

void main(void)
   {
   unsigned char IoSourceSetting;

   // Reset port A and D. All bits set as input by default
   io_Reset (IO_A);
   io_Reset (IO_D);

   // Use pins 0-3 in port IO_D as interrupt source
   // and pins 4-7 in port IO_A as interrupt source.
   irq_SetIoSource(b00001111);

   // Get the configuration for bits 3 and 4
   IoSourceSetting  = irq_getIoSource(b00011000);

   // IoSourceSetting is now == b00001000;

   …
```

irq_GetPriority()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
unsigned short irq_GetPriority (unsigned short Irq);
```

Return

The interrupt priority setting for interrupt `Irq`.

Description

`irq_GetPriority()` returns the current interrupt priority setting for interrupt request `Irq`. The priority settings are stored internally in the *Interrupt* module wherefore the priority can be retrieved disregarding if an interrupt is enabled or disabled. Valid interrupt priorities are:

`IRQ_DISABLED`	- Interrupt disabled
`IRQ_PRIO_LOW`	- Low priority
`IRQ_PRIO_NORM`	- Normal priority
`IRQ_PRIO_HIGH`	- High priority

Default priority for all interrupts are `IRQ_PRIO_NORM`. An interrupt having the priority `IRQ_DISABLED` is entirely disabled and `irq_Enable()` has no effect. `irq_GetStatus()` will though still return a valid interrupt request status. Valid values for `Irq` are:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
```

```
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

Example

```
// Set IRQ vector
IRQ_SET_VECTOR(IRQ_A2_D2, IrqHandler);

// Check the priority
if (irq_GetPriority(IRQ_A2_D2) != IRQ_PRIO_HIGH)
   {

   // Disable interrupt
   irq_Disable(IRQ_A2_D2);

   // Set the highest interrupt priority
   irq_SetPriority (IRQ_A2_D2, IRQ_PRIO_HIGH);
   }

// Enable interrupt having the highest priority
irq_Enable(IRQ_A2_D2);
```

irq_GetStatus()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
unsigned char irq_GetStatus(unsigned short Irq);
```

Return

Status of interrupt request `Irq`.

Description

`irq_GetStatus()` retrieves the interrupt request status for the interrupt request source `Irq`. A non-zero status (!= 0) means that an interrupt request is waiting to be serviced. When the status is zero (0) no request is pending for interrupt `Irq`. Valid values for `Irq` are:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

Example

```
#include "binary.h"

…

void main(void)
   {

   // Reset port D. All bits set as input by default
   io_Reset (IO_D);

   // Use bit number 3 in port IO_D as interrupt source.
   Irq_Disable(IRQ_A3_D3);
   irq_SetIoSource(b00001000);

   // Use the rising edge (low to high voltage transition) as
   // interrupt trigger for bit 3
   irq_SetEdge(b00001000);
   while (1)
      {

      // The IRQ_A3_D3 interrupt is disabled and no
      // interrupt service routine will be invoke. It is however
      // still possible to use the interrupt status as a polling
      // method to check if bit 3 in port IO_D has sensed a
      // low to high voltage level transition.
      while (irq_GetStatus(IRQ_A3_D3) == 0)
         ;

      // When the program reaches this point a low to high
      // voltage transition has occurred.

      // Respond to the transition in a meaningful way
      …

      }
   }
```

IRQ_INTERRUPT()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
IRQ_INTERRUPT(Irq);
```

Return

Description

`IRQ_INTERRUPT()` is a macro that generates an interrupt request of the type `Irq` by setting the corresponding interrupt request bit in the interrupt controller. If the interrupt `Irq` has previously been enabled by `irq_Enable()` the interrupt request will be serviced. Valid values for `Irq` are:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

Example

IRQ_SET_VECTOR()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
IRQ_SET_VECTOR(Irq,Handler);
```

Return

Description

`IRQ_SET_VECTOR()` is an assembler macro (generated by the C preprocessor) that specifies the address of the interrupt service routine to be invoked when servicing the interrupt request `Irq`. `Handler` should be a pointer to a function of the type:

```
void InterruptHandler(void).
```

`IRQ_SET_VECTOR()` does not result in any executable code but rather a data segment that is written to the interrupt table memory when a program is uploaded to the MCU. `IRQ_SET_VECTOR()` can therefore be placed anywhere in the source code file that implements the interrupt handler. A good place is at the beginning of the interrupt handler. An interrupt vector should be set one time only in a program. Trying to set the same vector several times can potentially result in an error when uploading the compiled program from ZDS II. Valid values for `Irq` are:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
```

```
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

Example

```
#pragma interrupt
void TimerInterruptHandler(void)
   {

   // Set the interrupt vector
   IRQ_SET_VECTOR(IRQ_TMR_0, TimerInterruptHandler);

   // Service the interrupt
   …
   }
```

irq_SetEdge()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
irq_SetEdge(unsigned char BitMask);
```

Return

Description

`irq_SetEdge()` specifies if an interrupt request should be generated at the rising or falling edge of a voltage transition for the pins `BitMask` of port `IO_A` and `IO_D`. A 0 (zero) configures the pin to issue an interrupt request at the rising edge and a 1 (one) issue an interrupt request at the falling edge. Default setting is that all port pins configured as interrupt request sources are configured to issue an interrupt at the falling edge.

Example

```
…

unsigned char EdgeSetting;

// Reset port D. All bits set as input by default
io_Reset(IO_D);

// Use bits number 3 and 4 in port IO_D as interrupt source.
// (The other bits use port IO_A as interrupt sources)
irq_SetIoSource(b00011000);

// Use the rising edge (low to high voltage transition) as
// interrupt trigger for bits 3 and 4
// (The other bits use high to low transition)
irq_SetEdge(b00011000);

// Set IRQ vector and enable interrupt
IRQ_SET_VECTOR(IRQ_A3_D3, InterruptHandler);
irq_Enable(IRQ_A3_D3);

…
```

irq_SetIoSource()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
irq_SetIoSource(unsigned char BitMask)
```

Return

Description

`irq_SetIoSource()` specifies which port pins of port `IO_A` and `IO_D` that should be configured as sources for interrupt requests. The bits are mutually exclusive between `IO_A` and `IO_D`. Thus, each bit only can only be used by either `IO_A` or `IO_D`. `BitMask` specifies which bits should be associated with what port. A 0 (zero) selects port `IO_A` as interrupt request source and a 1 (one) selects `IO_D`. Default configuration is that all 8 bits (0 - 7) are associated with `IO_A`.

Example

```
#include <binary.h>
#include <interrupt.h>

void main(void)
   {
   unsigned char EdgeSetting;

   // Reset port A and D. All bits set as input by default
   io_Reset (IO_A);
   io_Reset (IO_D);

   // Use bits number 0-3 in port IO_D as interrupt source
   // and bits number 4-7 in port IO_A as interrupt source.
   irq_SetIoSourceIRQ_SET_IO_SOURCE(b00001111);

   // Use the rising edge (low to high voltage transition) as
   // interrupt trigger for all bits.
   irq_SetEdge(b11111111);

   // Set IRQ vectors and enable interrupts
   IRQ_SET_VECTOR(IRQ_A0_D0, IrqHandler0);
   irq_Enable(IRQ_A0_D0);

   …

   IRQ_SET_VECTOR(IRQ_A7_D7, IrqHandler7);
   irq_Enable(IRQ_A7_D7);
   }
```

irq_SetPriority()

Module

Interrupt

Synopsis

```
#include "interrupt.h"
void irq_SetPriority (unsigned short Irq, unsigned short Priority);
```

Return

Description

`irq_SetPriority()` sets the priority level `Priority` of interrupt request `Irq`. The priority can be one of the following four levels:

```
IRQ_DISABLED      - Interrupt disabled
IRQ_PRIO_LOW      - Low priority
IRQ_PRIO_NORM     - Normal priority
IRQ_PRIO_HIGH     - High priority
```

The priority is administrated by the *Interrupt* module internally and stays in effect until it is altered. The priority `IRQ_DISABLED` disables an interrupt entirely and `irq_Enable()` has no effect with this setting. Default priority for all interrupts is `IRQ_PRIO_NORM`. The `Irq` parameter can have one of the following values:

```
IRQ_A0_D0
IRQ_A1_D1
IRQ_A2_D2
IRQ_A3_D3
IRQ_A4_D4
IRQ_A5_D5
IRQ_A6_D6
IRQ_A7_D7
IRQ_ADC
IRQ_C0
IRQ_C1
IRQ_C2
IRQ_C3
IRQ_DMA
IRQ_I2C
IRQ_RESET
IRQ_SPI
IRQ_TMR_0
IRQ_TMR_1
IRQ_TMR_2
IRQ_TMR_3
IRQ_TRAP
IRQ_UART_0_RX
IRQ_UART_0_TX
```

```
IRQ_UART_1_RX
IRQ_UART_1_TX
IRQ_WDT
```

Example

```
#pragma interrupt
void InterruptHandler(void)
   {

   // Set interrupt vector
   IRQ_SET_VECTOR(IRQ_SPI, InterruptHandler);

   // Service interrupt
   …
   }

void main(void)
   {

   // When the program starts executing the default
   // interrupt priority is IRQ_PRIO_NORM

   // Change interrupt priority to its highest level
   irq_SetPriority(IRQ_SPI,IRQ_PRIO_HIGH);

   // Enable interrupt
   irq_Enable(IRQ_SPI);

   // The SPI-interrupt is now serviced at the highest priority level

   // Disable interrupt
   irq_Disable(IRQ_SPI);

   // Set low interrupt priority
   irq_SetPriority(IRQ_SPI,IRQ_PRIO_LOW);

   // Enable interrupt
   irq_Enable(IRQ_SPI);

   // The SPI-interrupt is now serviced at the lowest priority level

   …
   }
```

REG_OFFSET()

Module

Register

Synopsis

```
#include "register.h"
REG_OFFSET(RegBaseAddr,Offset);
```

Return

Description

`REG_OFFSET()` is a macro used for offset addressing. It returns the memory address of `RegBaseAddr` plus `Offset`. This macro is used for accessing device driver ports, such as status and control ports, in a resource independent and yet compact manner.

Example

```
void SetTimerReload(reg_addr TimerID, unsigned short Value)
   {
   *REG_OFFSET(TimerID,TMR_REL_HI) = (unsigned char) (Value >> 0x08);
   *REG_OFFSET(TimerID,TMR_REL_LO) = (unsigned char) Value;
   }
```

ser_BirqDisable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_BirqDisable(reg_addr Uart);
```

Return

Description

`ser_BirqDisable()` disables baud rate generator interrupt requests for serial controller `Uart`. This is the default setting of the serial controllers. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Reset serial controller 0 (UART 0)
ser_Reset(SER_0);

// Set the UART to generate an interrupt every 10th ms (this is on a
// MCU with 18.432 MHz core clock frequency)
ser_SetBaudRate(SER_0, 11520);

// Setup and enable the interrupt controller
irq_SetPriority (IRQ_UART_0_RX, IRQ_PRIO_LOW);
irq_Enable (IRQ_UART_0_RX);

// Enable baud rate generator interrupt
ser_BirqEnable(SER_0);

// Serial controller 0 does now generate an interrupt every 10th ms
// which can be serviced by an interrupt handler

// Perform some meaningful processing
…

// Disable baud rate generator interrupt
Ser_BirqDisable(SER_0);
irq_Disable (IRQ_UART_0_RX);
```

ser_BirqEnable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_BirqEnable(reg_addr Uart);
```

Return

Description

`ser_BirqEnable()` allows the baud rate generator in serial controller `Uart` to generate interrupt requests in the same way as a timer. Valid values for `Uart` are `SER_0` and `SER_1`. The timer interrupt vector is the same as for Data Received interrupt (`IRQ_UART_0_RX/IRQ_UART_1_RX`).

A serial controller must **not** be used for RS-232 communication and as a timer at the same time. As a consequence, the parameter `CFG_SER_0/CFG_SER_1` (in `configure.h`) for the controller in question must **not** be defined. This is not only a hardware restriction. If e.g. `CFG_SER_0` was defined and serial controller 0 was to be used as a timer as well, the `IRQ_UART_0_RX interrupt` would be assigned to two competing interrupt handlers.

NOTE
The baud rate timer interrupt frequency should **not** be calculated on the assumption that the baud rate and the timer frequency are the same. This is not the case. A baud rate generator programmed for 9600 baud does not generate 9600 interrupts per second when used as a timer. The timer interrupt frequency is much higher. As a rule of thumb, a timer reload value of 11520 generates an interrupt every 10^{th} millisecond on an 18.432 MHz MCU.

Example

```
#pragma interrupt
void InterruptHandler(void)
   {
   IRQ_SET_VECTOR(IRQ_UART_0_RX, InterruptHandler);

   // Service the interrupt
   …
   }

…

// CFG_SER_0 is NOT defined. SER_1 can still be used
// for RS-232 communication

// Reset serial controller 0 (UART 0)
ser_Reset(SER_0);
```

```
// Set the UART to generate an interrupt every 10th ms
// (this is on a MCU with 18.432 MHz core clock frequency)
ser_SetBaudRate(SER_0, 11520);

// Setup and enable IRQ
irq_SetPriority (IRQ_UART_0_RX, IRQ_PRIO_LOW);
irq_Enable (IRQ_UART_0_RX);
ser_BirqEnable(SER_0);

// Serial controller 0 does now generate an interrupt every 10th ms

// Perform some meaningful processing
…
```

ser_ClearError()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_ClearError(reg_addr Uart, unsigned char ErrorMsgBitMask);
```

Return

Description

`ser_ClearError()` clears the error(s) defined by `ErrorMsgBitMask` for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

`ErrorMsgBitMask` specifies which error(s) that should be cleared and can have the following error condition values:

```
SER_ERR_RX_ALL
SER_ERR_RX_BREAK
SER_ERR_RX_FRAMING
SER_ERR_RX_OVERRUN
SER_ERR_RX_PARITY
SER_ERR_RX_TIMEOUT
SER_ERR_TX_TIMEOUT
```

Several error conditions can be cleared at the same time by merging the error conditions with a bitwise OR-operation.

Example

```
…

// Generation of timeout error has to be enabled in configure.h

unsigned short RtnVal;
unsigned char RXChar;

// Init the serial controller (9600, 8, N, 1)
ser_Init(SER_1, 0);

// Read a byte from the serial port
ser_Read(SER_1, &RXChar, 1);

// Check if timeout or overrun error
if (ser_GetError(SER_1, SER_ERR_RX_TIMEOUT | SER_ERR_RX_OVERRUN))
   {

   // Deal with error condition
```

```
    …

    // Clear error condition
    ser_ClearError(SER_1, SER_ERR_RX_TIMEOUT | SER_ERR_RX_OVERRUN);
    }
…
```

ser_Close()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
void ser_Close(reg_addr Uart);
```

Return

Description

`ser_Close()` shuts down serial controller `Uart` entirely and reset all configuration parameters. After a controller has been closed it has to be initialized with `ser_Init()` again before usage. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
unsigned char TXChar[20] = "I live!!! ";
unsigned short RtnVal = 0;

// Init port
ser_Init(SER_0, 0);

// If the fifo is smaller than 11 slots (one slot always has
// to be empty) this call will block until all bytes has been
// put into the fifo.
RtnVal = ser_Write(SER_0, TXChar, 10);
if (RtnVal != 0)
   {

   // An error occured
   ser_Close(SER_0);
   }
```

ser_CtsDisable()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_CtsDisable(reg_addr Uart);
```

Return

Description

`ser_CtsDisable()` disables the use of RTS/CTS hardware flow control at the transmitter level. Valid values for `Uart` are `SER_0` and `SER_1`.

NOTE
Disabling/enabling the CTS signal from user code can cause unpredictable behavior of the serial controller when communication is handled by this device driver. The enabling/disabling of the flow control is automatically handled during initialization by `ser_Init()` based on the setup in `configure.h`.

Example

ser_CtsEnable()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_CtsEnable(reg_addr Uart);
```

Return

Description

`ser_CtsEnable()` enables the use of RTS/CTS hardware flow control at the transmitter level. Valid values for `Uart` are `SER_0` and `SER_1`.

NOTE
Disabling/enabling the CTS signal from user code can cause unpredictable behavior of the serial controller when communication is handled by this device driver. The enabling/disabling of the flow control is automatically handled during initialization by `ser_Init()` based on the setup in `configure.h`.

Example

ser_GetError()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_GetError(reg_addr Uart, unsigned char
ErrorMsgBitMask);
```

Return

Error status for serial controller `Uart` filtered according to `BitMask`.

Description

`ser_GetError()` returns the error status for the serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`. Possible error codes are:

```
SER_ERR_TX_TIMEOUT
SER_ERR_RX_TIMEOUT
SER_ERR_RX_BREAK
SER_ERR_RX_FRAMING
SER_ERR_RX_OVERRUN
SER_ERR_RX_PARITY
SER_ERR_RX_ALL
```

`SER_ERR_TX_TIMEOUT` and `SER_ERR_RX_TIMEOUT` can only be returned when transmitter and/or receiver timeout has been enabled in `configure.h`. If several errors have occurred the error codes are merged by a bit wise OR-operation. The `BitMask` parameter is an error code filter and specifies which error(s) statuses should be returned by `ser_GetError()`.

Example

```
unsigned char Buffer;

// Read one byte from serial controller
unsigned short ser_Read(SER_0, &Buffer, 1);

// Check if a timeout or overrun error occurred
if (ser_GetError(SER_0, SER_ERR_RX_OVERRUN | SER_ERR_RX_TIMEOUT)
   {

   // Handle error condition
   …

   // Clear error condition
   ser_ClearError(SER_0, SER_ERR_RX_OVERRUN | SER_ERR_RX_TIMEOUT);
   }
```

ser_Init()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_Init(reg_addr Uart, char IrEnable);
```

Return

Description

`ser_Init()` initialized serial controller `Uart` and makes it ready for communication. Valid values for `Uart` are `SER_0` and `SER_1`. When `IrEnable` is a non-zero value (true) the infrared encoder/decoder will be enabled as well. This way of enabling IR support prevent spurious communication that may other wise occur when using `ser_IrEnable()` alone. After initialization the serial receiver is on-line and operational. Data received can be read by `ser_Read()`. The serial transmitter is put on hold since there has not been a transmission requested yet. Transmissions are carried out by calling `ser_Write()`.

Default configuration parameters after initialization are:

- 9600 Baud
- 8 data bits
- 1 stop bit
- No parity
- No multiprocessor support
- Normal IRQ priority (`IRQ_PRIO_NORM`)
- Interrupt `IRQ_UART_0_RX` enabled
- Interrupt `IRQ_UART_0_TX` disabled
- RX and TX buffers cleared

In the case that the default configuration is not the preferred, reconfiguration can be done after the serial controller has been initialized.

Example

```
// Initialize
ser_Init(SER_0, 0);

// Temporary disable serial communication
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Reconfigure serial communication
ser_SetBaudRateSER_0, SER_2400_BAUD);
ser_ParityNone(SER_0);

// Disable serial interrupts
```

```
irq_Disable(IRQ_UART_0_RX);

// Change interrupt priority to low
irq_SetPriority (IRQ_UART_0_RX, IRQ_PRIO_LOW);
irq_SetPriority (IRQ_UART_0_TX, IRQ_PRIO_LOW);

// Fire up serial controller again
// Do not enable IRQ_UART_0_TX since that is managed
// by ser_Write().
irq_Enable(IRQ_UART_0_RX);
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_IrDisable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_IrDisable(reg_addr Uart);
```

Return

Description

`ser_IrDisable()` disables the integrated IrDA encoder/decoder for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Init the serial controller with IrDA enabled (9600, 8, N, 1)
ser_Init(SER_1, 1);

// Send some data
…

// Disable IrDA
ser_IrDisable(SER_1);
```

ser_IrEnable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_IrEnable(reg_addr Uart);
```

Return

Description

`ser_IrEnable()` enables the integrated IrDA encoder/decoder for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

NOTE
If the IrDA encode/decoder is enabled before the alternate functions have been enabled for the serial communication IO-pins, spurious signaling may occur. The preferred way to enable IrDA is when initializing the serial controller (`ser_Init()`).

Example

ser_IrqRxDisable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_IrqRxDisable(reg_addr Uart);
```

Return

Description

`ser_IrqRxDisable()` disables the Data Received interrupt request for serial controller `Uart`. At this stage only receiver errors will generate an interrupt request. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
#pragma interrupt
void ser_RxIrqHandler0(void)
   {

   // Set vector for this isr
   IRQ_SET_VECTOR(IRQ_UART_0_RX,ser_RxIrqHandler0);

   // Check if receive buffer is full
   if (RxBufferIsFull())
      {

      // Put the receiver interrupt on hold
      //(Errors still generate interrupt)
      ser_IrqRxDisable(SER_0);
      }
   else
      {

      // Read the newly received data
      StoreInBuffer(*REG_OFFSET(SER_0,SER_RX_DATA));
      }
   …
```

ser_IrqRxEnable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_IrqRxEnable(reg_addr Uart);
```

Return

Description

`ser_IrqRxEnable()` enables the Data Received interrupt request for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
#pragma interrupt
void ser_RxIrqHandler0(void)
   {

   // Set vector for this isr
   IRQ_SET_VECTOR(IRQ_UART_0_RX,ser_RxIrqHandler0);
   …
   }

void Start SerialReceiver(void)
   {

   // Enable serial receiver
   irq_Enable(IRQ_UART_0_RX);
   ser_IrqRxEnable (SER_0);
   ser_RxEnable (SER_0);
   }
```

ser_LoopDisable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_LoopDisable(reg_addr Uart);
```

Return

Description

`ser_LoopDisable()` disables the internal loop back connection between transmitter and receiver for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

ser_LoopEnable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_LoopEnable(reg_addr Uart);
```

Return

Description

`ser_LoopEnable()` enables the internal loop back connection between transmitter and receiver for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

ser_ParityEven()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_ParityEven(reg_addr Uart);
```

Return

Description

`ser_ParityEven()` specifies that serial controller `Uart` should use even parity when communicating. Valid values for `Uart` are `SER_0` and `SER_1`. Default setting is *No Parity*.

Example

```
// Initialize controller
ser_Init(SER_0, 0);

// Temporary disable receiver and transmitter
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Set 38,4 KBaud communication speed
ser_SetBaudRate (SER_0, SER_38400_BAUD);

// Use even parity
ser_ParityEven(SER_0);

// Re-enable receiver and transmitter
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_ParityNone()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_ParityNone(reg_addr Uart);
```

Return

Description

`ser_ParityNone()` specifies that serial controller `Uart` should not use parity when communicating. Valid values for `Uart` are `SER_0` and `SER_1`. This is the default setting.

Example

```
// Initialize controller
ser_Init(SER_0, 0);

// Temporary disable receiver and transmitter
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Set 38,4 KBaud communication speed
ser_SetBaudRate (SER_0, SER_38400_BAUD);

// Disable parity
ser_ParityNone(SER_0);

// Re-enable receiver and transmitter
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_ParityOdd ()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_ParityOdd(reg_addr Uart);
```

Return

Description

`ser_ParityOdd()` specifies that serial controller `Uart` should use odd parity when communicating. Valid values for `Uart` are `SER_0` and `SER_1`. Default setting is *No Parity*.

Example

```
// Initialize controller
ser_Init(SER_0, 0);

// Temporary disable receiver and transmitter
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Set 38,4 KBaud communication speed
ser_SetBaudRate (SER_0, SER_38400_BAUD);

// Use odd parity
ser_ParityOdd(SER_0);

// Re-enable receiver and transmitter
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_Read()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned short ser_Read(reg_addr Uart, unsigned char *Buffer,
unsigned char Size);
```

Return

The error status of the receiver in serial controller `Uart`.

Description

`ser_Read()` receives `Size` number of bytes from serial controller `Uart` and store the data in `Buffer`. This is a non-blocking function which receives data in the background. Therefore `ser_Read()` returns immediately and most probably before the RX transaction has finished. `ser_Read()` may block in cases for which there already is a previously initiated and ongoing RX transaction. Owing to its non-blocking nature, the transmission buffer `Buffer` must be guaranteed, by the user program, to be both allocated and consistent for the entire life time of the transaction. Failing to do so may results in unpredictable program behavior. The following two code snippets are examples of invalid usage:

Example 1:

```
// Allocate buffer
unsigned char RXBuffer[20];

…

// Transmitt buffer
ser_Read(SER_0, RXBuffer, 11);

// Update buffer with new content
TXBuffer[0] = 'U';   // <------ Buffer is now invalid since it is
                     // updated before the RX transaction has
                     // finished. The 'U' will probably be
                     // overwritten by the first byte received
```

Example 2:

```
void MyFunct(void)
   {

   // Allocate buffer on stack
   unsigned char RXBuffer[20];

   …

   // Transmitt buffer
   ser_Read(SER_0, RXBuffer, 11);
```

```
        // Done
        return;  // <------ Buffer is now invalid since it will
                 // immediately be deallocated from the stack when
                 // returning from MyFunct() before RX transaction
                 // has finished.
        }
```

`ser_Read()` can be configured to timeout the data request preventing the MCU from hanging in case of a communication link shut down. This is done by the parameters `CFG_SER_0_RX_TIMEOUT` and `CFG_SER_1_RX_TIMEOUT` in `configure.h`. Valid values for `Uart` are `SER_0` and `SER_1`.

If an error has occurred a non-zero value is returned by `ser_Read()`. The least significant byte of the return value specifies the number of bytes remaining to be received when the error occurred. The error code can be retrieved by `ser_GetError()`. A value of 0 (zero) is returned when `ser_Read()` was successful.

Example

```
unsigned char RXChar = 0;
unsigned short RtnVal;


// Initialize UART and reconfigure for 38400 BAUD and EVEN PARITY
ser_Init(SER_1, 0);
ser_RxDisable(SER_1);
ser_SetBaudRateSER_1, SER_38400_BAUD);
ser_ParityEven(SER_1);
ser_TxEnable(SER_1);

// Wait for a CR to be received
while (RXChar != 13)
   {

   // Wait for a character
   // This may time-out if time-out is enabled in configure.h
   RtnVal = ser_Read(SER_1, &RXChar, 1);
   if (RtnVal != 0)
      {

      // An error occured
      ser_Close(SER_1);
      return;
      }
   }

// Continue program
…
```

ser_Reset()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_Reset(reg_addr Uart);
```

Return

Description

`ser_Reset()` resets serial controller `Uart`. This means:

- Transmitter disabled
- Receiver disabled
- CTS signal disabled
- Parity disabled
- Parity select set to Even parity
- Transmitter sends one stop bit
- Loop back disabled
- Baud rate generator interrupt disabled
- Multiprocessor mode disabled
- Multiprocessor data filter disabled
- Receive data interrupt disabled
- Ir disabled

In addition, the baud rate generator reload value is set to 0xffff. `ser_Reset()` only affects the dedicated serial controller. No other hardware or software modules are changed. Use `ser_Close()` to shut down the serial device driver in a deterministic fashion. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

ser_RxDisable()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
ser_RxDisable(reg_addr Uart);
```

Return

Description

`ser_RxDisable()` shuts down the receiver for serial controller `Uart`. Configurations will remain intact but the controller will not recognize any attempts to communicate. This is useful for disabling the controller when default configuration is not wanted. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Initialize and reconfigure for 38400 BAUD and EVEN PARITY
ser_Init(SER_1, 0);
ser_RxDisable(SER_1);
ser_SetBaudRate(SER_1, SER_38400_BAUD);
ser_ParityEven (SER_1);
ser_RxEnable(SER_1);
```

ser_RxEnable()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_RxEnable(reg_addr Uart);
```

Return

Description

`ser_RxEnable()` enables the receiver for serial controller `Uart`. Previous configuration will remain unchanged. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Initialize and reconfigure for 38400 BAUD and EVEN PARITY
ser_Init(SER_1, 0);
ser_RxDisable(SER_1);
ser_SetBaudRate(SER_1, SER_38400_BAUD);
ser_ParityEven (SER_1);
ser_RxEnable(SER_1);
```

ser_RxGetByteCount()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_RxGetByteCount(reg_addr Uart);
```

Return

The number of bytes transmitted so far for an ongoing TX transaction.

Description

`ser_RxGetByteCount()` returns the number of bytes that have been received so far by serial controller `Uart` for an ongoing RX transaction.

NOTE
`ser_RxGetByteCount()` does **not** guarantee that the internal variable keeping track of the number of received bytes will be read in an atomic way. Cases for which an atomic read is desired, interrupt first has to be disabled using `irq_Di()`. Keep in mind that globally disabling interrupt for prolonged time may have a negative impact on the performance.

Example

```
unsigned char Data[16];
unsigned char ByteCount = 0;

// Initialize controller
ser_Init(SER_0, 0);

// Send 16 bytes
ser_Read(SER_0, Data, 16);

// Check progress
while (ByteCount < 14)
   {
   PrintProgress(ByteCount);
   ByteCount = ser_RxGetByteCount(SER_0);
   }
```

ser_RxIsBusy()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_RxIsBusy(reg_addr Uart);
```

Return

Non-zero value when the serial device driver is busy receiving data. 0 when the driver is idle.

Description

`ser_RxIsBusy()` is used to check the receiving status of the device driver for serial controller `Uart`. When 0 (zero) is returned there is no ongoing receiving transaction and the RX interrupt handler is disabled. Consequently, when a non-zero value is returned there is data to be received and the interrupt handler is enabled. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
…

// Start a new read transaction if RX is idle.
if (ser_RxIsBusy(SER_0) == 0)
   {
   ser_Read(SER_0, &RxBuffer, 1);
   }

…
```

ser_RxIsEnabled()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_RxIsEnabled(reg_addr Uart);
```

Return

The enable/disable status of serial controller `Uart`.

Description

`ser_RxIsEnabled()` returns a non-zero (true) value when the receiver of serial controller `Uart` is enabled. When the receiver is disabled, zero (false) is returned. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Check that receiver is disabled before reconfiguring it
if (ser_RxIsEnabled(SER_1)
   ser_RxDisable(SER_1);

// Reconfigure serial controller
set_SetBaudRate(SER_1, SER_38400_BAUD);
ser_ParityEven(SER_1);
ser_RxEnable(SER_1);
```

ser_SetBaudRate ()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_SetBaudRate(reg_addr Uart, unsigned short BaudRate);
```

Return

Description

`ser_SetBaudRate()` sets the baud rate generator reload value to `BaudRate` for serial controller `Uart`. The `BaudRate` parameter is thus not the baud rate per se. See the Z8 Encore! documentation for how to calculate the baud rate value. Valid values for `Uart` are `SER_0` and `SER_1`. The following predefined baud rate generator reload values are valid for 18.432 MHz and 20 MHz core clock frequency provided that one of these two system clock frequencies has been for configuration parameter `CFG_CPU_FREQUENCY`:

```
SER_300_BAUD
SER_600_BAUD
SER_1200_BAUD
SER_2400_BAUD
SER_4800_BAUD
SER_9600_BAUD
SER_19000_BAUD
SER_38400_BAUD
SER_57600_BAUD
SER_115200_BAUD
```

Example

```
// Initialize serial controller
ser_Init(SER_0, 0);

// Temporary disable RX and TX
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Set new baud rate
ser_SetBaudRateSER_0, SER_38400_BAUD);

// Reenable RX and TX
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_StopBitsOne()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_StopBitsOne(reg_addr Uart);
```

Return

Description

`ser_StopBitsOne()` configures the serial controller `Uart` to use 1 stop bit when communicating. Valid values for `Uart` are `SER_0` and `SER_1`. This is the default setting.

Example

```
// Initialize controller
ser_Init(SER_0, 0);

// Temporary disable receiver and transmitter
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Set 38,4 KBaud communication speed
ser_SetBaudRate (SER_0, SER_38400_BAUD);

// Use even parity and one stop bit
ser_ParityEven(SER_0);
ser_StopBitsOne(SER_0);

// Re-enable receiver and transmitter
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_StopBitsTwo()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_StopBitsTwo(reg_addr Uart);
```

Return

Description

`ser_StopBitsTwo()` configures the serial controller `Uart` to use 2 stop bits when communicating. Valid values for `Uart` are `SER_0` and `SER_1`. One stop bit is the default setting.

Example

```
// Initialize controller
ser_Init(SER_0, 0);

// Temporary disable receiver and transmitter
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Set 38,4 KBaud communication speed
ser_SetBaudRate (SER_0, SER_38400_BAUD);

// Use even parity and two stop bits
ser_ParityEven(SER_0);
ser_StopBitsTwo(SER_0);

// Re-enable receiver and transmitter
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_TxBreak()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_TxBreak(reg_addr Uart);
```

Return

Description

`ser_TxBreak()` sends a break condition from serial controller `Uart` to the node at the other end of the serial connection. This interrupts any transmission in progress. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
unsigned char Data[4] = {'A', 'B', 'C', 'D'};

// Initialize controller
ser_Init(SER_0, 0);

// Send the 4 bytes
ser_Write(SER_0, Data, 4);

// Check status
if (SomethingWentWrong())
   {

   // Break the transmission
   ser_TxBreak(SER_0);

   // Do something useful
   …

   // Stop sending break
   ser_TxClearBreak(SER_0);
   }
```

ser_TxClearBreak()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_TxClearBreak(reg_addr Uart);
```

Return

Description

`ser_TxClearBreak()` clear a break condition that is being transmitted by serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
unsigned char Data[4] = {'A', 'B', 'C', 'D'};

// Initialize controller
ser_Init(SER_0, 0);

// Send the 4 bytes
ser_Write(SER_0, Data, 4);

// Check status
if (SomethingWentWrong())
   {

   // Break the transmission
   ser_TxBreak(SER_0);

   // Do something useful
   …

   // Stop sending break
   ser_TxClearBreak(SER_0);
   }
```

ser_TxDisable()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_TxDisable(reg_addr Uart);
```

Return

Description

`ser_TxDisable()` disables the transmitter in serial controller `Uart`. All configurations stays intact but the transmitter will not try to send any data until it has been enabled again. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Initialize and reconfigure for 38400 BAUD and EVEN PARITY
ser_Init(SER_0, 0);

// Temporary disable serial communication
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Reconfigure serial communication
ser_SetBaudRate(SER_0, SER_38400_BAUD);
ser_ParityEven(SER_0);
ser_StopBitsTwo(SER_0);

// Fire up serial controller again
ser_RxEnable(SER_0);
ser_TxEnable(SER_0);
```

ser_TxEnable()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_TxEnable(reg_addr Uart);
```

Return

Description

`ser_TxEnable()` enables the transmitter for serial controller `Uart`. Communication parameters previously set are automatically in effect. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Initialize and reconfigure for 2400 BAUD, ODD PARITY
// and 2 stop bits
ser_Init(SER_0, 0);

// Temporary disable serial communication
ser_RxDisable(SER_0);
ser_TxDisable(SER_0);

// Reconfigure serial communication
ser_SetBaudRate(SER_0, SER_2400_BAUD);
ser_ParityOdd (SER_0);
ser_StopBitsTwo (SER_0);

// Fire up serial controller again
ser_RxEnable (SER_0);
ser_TxEnable(SER_0);
```

ser_TxGetByteCount()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_TxGetByteCount(reg_addr Uart);
```

Return

The number of bytes transmitted so far for an ongoing TX transaction.

Description

`ser_TxGetByteCount()` returns the number of bytes that have been transmitted so far by serial controller `Uart` for an ongoing TX transaction.

NOTE
`ser_TxGetByteCount()` does **not** guarantee that the internal variable keeping track of the number of transmitted bytes will be read in an atomic way. Cases for which an atomic read is desired, interrupt first has to be disabled using `irq_Di()`. Keep in mind that globally disabling interrupt for prolonged time may have a negative impact on the performance.

Example

```
unsigned char Data[] = “Yada Yada Yada”;
unsigned char ByteCount = 0;

// Initialize controller
ser_Init(SER_0, 0);

// Send 14 bytes
ser_Write(SER_0, Data, 14);

// Check progress
while (ByteCount < 14)
   {
   PrintProgress(ByteCount);
   ByteCount = ser_TxGetByteCount(SER_0);
   }
```

ser_TxHwIsBusy()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_TxHwIsBusy(reg_addr Uart);
```

Return

True (non-zero) when the transmitter data register or shift register contains data and false (zero) when the data register and the shift register is empty.

Description

`ser_TxHwIsBusy()` reads the status of both the transmitter data register and the shift register of serial controller `Uart`. When either (or both) of the registers contain data a non-zero value is returned. When both registers are empty 0 (zero) is returned. When this is the case, the serial transmitter hardware is entirely at rest. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
…

// Perform a reconfiguration of serial controller SER_0
// in the middle of a program
if (ser_TxIsEnabled(SER_0))
   {

   // Wait for the controller to finish eventual TX transactions
   while (ser_TxIsBusy(SER_0))
      ;
   while (ser_TxHwIsBusy(SER_0))
      ;

   // Disable transmitter
   ser_TxDisable(SER_0);
   }

// Reconfigure speed and enable transmitter again
ser_SetBaudRate(SER_0, SER_2400_BAUD);
ser_TxEnable(SER_0);
```

ser_TxRegIsFull()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_TxRegIsFull(reg_addr Uart);
```

Return

True (non-zero) when the transmitter data register contains data and False (zero) when the data register is empty.

Description

`ser_TxRegIsFull()` reads the status of the transmitter data register for serial controller `Uart`. When the data register contains data a non-zero value is returned. Otherwise 0 (zero) is returned.
`ser_TxRegIsFull()` is useful when it is necessary to have a detailed and up to date control of the serial transmitter. Valid values for `Uart` are `SER_0` and `SER_1`.

NOTE
The transmitter shift register may still contain data when `ser_TxRegIsFull()` returns 0.

Example

ser_TxIsBusy()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_TxIsBusy(reg_addr Uart);
```

Return

Non-zero value when the serial device driver is busy sending data. 0 when the driver is idle.

Description

`ser_TxIsBusy()` is used for checking the transmission status of the device driver for serial controller `Uart`. When 0 (zero) is returned there is no ongoing transmission transaction and the transmission interrupt handler is disabled. Consequently, when a non-zero value is returned there is data awaiting transmission and the interrupt handler is enabled. Valid values for `Uart` are `SER_0` and `SER_1`.

`ser_TxIsBusy()` does however **not** take into account that the hardware may still be busy transmitting the last one or two bytes of data stored in the transmitter data register and the transmitter shift register. The hardware status can be determined by `ser_TxHwIsBusy()` and `ser_TxRegIsFull()`.

Example

```
...

// Perform a reconfiguration of serial controller SER_0
// in the middle of a program
if (ser_TxIsEnabled(SER_0))
   {

   // Wait for the controller to finish eventual TX transactions
   while (ser_TxIsBusy(SER_0))
      ;
   while (ser_TxHwIsBusy(SER_0))
      ;

   // Disable transmitter
   ser_TxDisable(SER_0);
   }

// Reconfigure speed and enable transmitter again
ser_SetBaudRate(SER_0, SER_2400_BAUD);
ser_TxEnable(SER_0);
```

ser_TxIsEnabled()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned char ser_TxIsEnabled(reg_addr Uart);
```

Return

Non-zero (true) when transmitter for serial controller `Uart` is enabled and 0 (false) when it is disabled.

Description

`ser_TxIsEnabled()` returns the enabled/disable status of the transmitter of serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
// Perform a reconfiguration of serial controller SER_0
// in the middle of a program
if (ser_TxIsEnabled(SER_0))
   {

   // Wait for the controller to finish eventual TX transactions
   while (ser_TxIsBusy(SER_0))
      ;
   while (ser_TxHwIsBusy(SER_0))
      ;

   // Disable transmitter
   ser_TxDisable(SER_0);
   }

// Reconfigure speed and enable transmitter again
ser_SetBaudRate(SER_0, SER_2400_BAUD);
ser_TxEnable(SER_0);
```

ser_TxPause()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
ser_TxPause(reg_addr Uart);
```

Return

Description

`ser_TxPause()` cause the serial controller `Uart` to pause transmissions. Transmission is resumed by `ser_TxResume()`. Buffered data is left intact. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

ser_TxResume()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
void ser_TxResume(reg_addr Uart);
```

Return

Description

`ser_TxResume()` resumes a transmission, for serial controller `Uart`, that has previously been halted by `ser_TxPause()`.Valid values for `Uart` are `SER_0` and `SER_1`.

Example

ser_Write()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
unsigned short ser_Write(reg_addr Uart, unsigned char *Buffer, unsigned
char Size);
```

Return

The error status of the transmitter in serial controller `Uart`.

Description

`ser_Write()` transmits `Size` number of bytes from `Buffer` using serial controller `Uart`. This is a non-blocking function which transmits data in the background. Therefore `ser_Write()` returns immediately and most probably before the TX transaction has finished. `ser_Write()` may block in cases for which there already is a previously initiated and ongoing TX transaction. Owing to its non-blocking nature, the transmission buffer `Buffer` must be guaranteed, by the user program, to be both allocated and consistent for the entire life time of the transaction. Failing to do so may results in unpredictable program behavior. The following two code snippets are examples of invalid usage:

Example 1:

```
// Allocate buffer
unsigned char TXBuffer[20] = "I live!!! ";

…

// Transmitt buffer
ser_Write(SER_0, TXBuffer, 11);

// Update buffer with new content
TXBuffer[0] = 'U';   // <------ Buffer is now invalid since it is
                     // updated before TX transaction has finished
```

Example 2:

```
void MyFunct(void)
   {

   // Allocate buffer on stack
   unsigned char TXBuffer[20] = "I live!!! ";

   …

   // Transmitt buffer
   ser_Write(SER_0, TXBuffer, 11);
```

```
        // Done
        return;  // <------ Buffer is now invalid since it will
                 // immediately be deallocated from the stack when
                 // returning from MyFunct() before TX transaction
                 // has finished.
        }
```

`ser_Write()` can be configured to timeout the data transmission, thereby preventing the MCU from hanging in case of a communication link shut down. This is done by the parameters `CFG_SER_0_TX_TIMEOUT` and `CFG_SER_1_TX_TIMEOUT` in `configure.h`. Valid values for `Uart` are `SER_0` and `SER_1`.

If an error has occurred, a non-zero value is returned by `ser_Write()`. The least significant byte of the return value specifies the number of bytes remaining to be transmitted when the error occurred. The error code can be retrieved by `ser_GetError`. A value of 0 (zero) is returned when `ser_Write()` was successful.

Example

…

```
// Send characters to a windows terminal window (or similar)
void WriteSerialData(void)
   {
   unsigned char TXChar[25] = "I'm Alive!!!    \n";
   unsigned long Counter = 0;
   unsigned short LoopCounter = 0;

   // Initialize the serial port using default settings
   ser_Init(SER_PORT, 0);

   // Write some text to the serial port
   for(LoopCounter = 1 ; LoopCounter < 1000 ; LoopCounter++)
      {
      Counter=0;

      // Count the number of wait loops performed before
      // the previous TX transaction finishes
      while (ser_TxIsBusy(SER_0))
         Counter++;

      // Add the wait loop count to the text sent
      // by the UART
      TXChar[20] = '\n';
      TXChar[21] = '\r';
      txt_lhtoa(Counter, (char *) &TXChar[12]);

      // Send the test over the serial connection
      ser_Write(SER_0, TXChar, 22);
      }

   // Wait for TX transaction to finish
   while (ser_TxIsBusy(SER_0))
      Counter++;
```

```
// ser_Close() wait for hardware to finish before
// closing. Hence, no problem with race condition
ser_Close(SER_PORT);
}
```

spi_BirqDisable()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_BirqDisable();
```

Return

Description

`spi_BirqDisable()` disables baud rate generator interrupt requests for the SPI controller. This is the default setting.

Example

spi_BirqEnable()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_BirqEnable();
```

Return

Description

`spi_BirqEnable()` allows the baud rate generator in the SPI controller to generate interrupt requests in the same way as a timer. The timer interrupt vector is the same as the generic SPI interrupt request (`IRQ_SPI`).

The SPI controller must **not** be used for communication and as a timer at the same time. As a consequence, the parameter `CFG_SPI` (in `configure.h`) must **not** be defined. This is not only a hardware restriction. If `CFG_SPI` was defined and the controller was to be used as a timer as well, the `IRQ_SPI` interrupt would be assigned to two competing interrupt handlers.

Note: The baud rate timer interrupt frequency should **not** be calculated on the assumption that the baud rate and the timer frequency are the same. This is not the case. A baud rate generator programmed for 128 KBaud does not generate 128 000 interrupts per second when used as a timer. The timer interrupt frequency is much higher. As a rule of thumb, a timer reload value of 18432 generates an interrupt every 1 millisecond on an 18.432 MHz MCU.

Example

```
#pragma interrupt
void InterruptHandler(void)
   {
   IRQ_SET_VECTOR(IRQ_SPI, InterruptHandler);

   // Service the interrupt
   …
   }

…

// Reset timer (CFG_SPI NOT defined)
spi_Reset ();
spi_SetBaudRate(18432);

// Setup IRQ
irq_SetPriority (TCK_TMR_IRQ, IRQ_PRIO_LOW);
irq_Enable (IRQ_SPI);
spi_BirqEnable();
```

```
// The SPI controller now generates an interrupt
// every millisecond on a 18.432 MHz MCU
…
```

spi_ClearError()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_ClearError();
```

Return

Description

`spi_ClearError()` clear **all** configured error conditions that has occurred during communication. Possible error conditions are:

```
SPI_ERR_COLLISION
SPI_ERR_OVERRUN
SPI_ERR_SLAVE_ABORT
```

More than one error condition can exist at the same time. Once an error has occurred it has to be cleared before any further SPI-communication can take place.

Which error condition the SPI module supports can are configured in `configure.h` using the parameters:

```
CFG_SPI_ERRORS_ALL
CFG_SPI_ERROR_COLLISION
CFG_SPI_ERROR_OVERRUN
CFG_SPI_ERROR_SLAVE_ABORT
```

Example

```
#define MEM_WRITE_ENABLE   (unsigned char) b00000110

void main(void)
   {

   // Init slave select port
   io_Reset (IO_E);
   io_SetDataDirOut(IO_E,b10000000);
   io_Outp(IO_E, b10000000);

   // Init SPI as master only. Set 1 Kbaud speed
   spi_Init(_NULL);
   spi_Disable();
   spi_SetBaudRate(SPI_1_KBAUD);
   spi_Enable();
```

```
// Send command to eeprom
Command = MEM_WRITE_ENABLE;
spi_MasterTransceive(&Command, 1);
while (spi_GetStatus() == SPI_STATUS_BUSY)
   ;

// Check for error
if (spi_GetError())
   {

   // Deal with error conditions
   …

   // Clear error
   spi_ClearError();
   }
…
```

spi_Close()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
void spi_Close();
```

Return

Description

`spi_Close()` shuts down the SPI controller, disables SPI interrupts and clear the alternate functions for the shared SPI port pins. When the controller has been closed `spi_GetStatus()` returns `SPI_STATUS_UNKNOWN`.

Example

```
…

// Select slave
spi_SlaveSelHi();

// Read bytes
spi_MasterTransceive(Data, 4);
while (spi_IsBusy() || spi_IsHwBusy())
   ;
spi_SlaveSelLo();

// Close SPI controller
spi_Close();

…
```

spi_Disable()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_Disable();
```

Return

Description

`spi_Disable()` shuts down the SPI controller. Configurations will remain intact but the controller will not recognize any attempts to communicate. This is useful for temporarily disabling the controller for reconfiguration of communication parameters.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spi_Disable();
spi_SetPhaseWhole();
spi_SetClkIdleHi();
spi_SetBaudRate(SPI_64_KBAUD);
spi_Enable();
```

spi_DisableOpenDrain()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_DisableOpenDrain();
```

Return

Description

`spi_DisableOpenDrain()` disables open-drain operation of the four signal pins (SCK, SS, MISO, MOSI). This is the default setting.

Example

spi_Enable()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_Enable();
```

Return

Description

`spi_Enable()` enables the SPI controller. Previously performed configurations are left intact.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spi_Disable();
spi_SetPhaseWhole();
spi_SetClkIdleHi();
spi_SetBaudRate(SPI_64_KBAUD);
spi_Enable();
```

spi_EnableOpenDrain()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_EnableOpenDrain();
```

Return

Description

`spi_EnableOpenDrain()` enables open-drain operations of the four signal pins (SCK, SS, MISO, MOSI). Default setting is open-drain disabled.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spi_Disable();
spi_EnableOpenDrain();
spi_Enable();
```

spi_GetError()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
unsigned char spi_GetError();
```

Return

Error status for the SPI controller.

Description

`spi_GetError()` returns the error condition for the SPI controller. Possible error conditions are:

```
SPI_ERR_COLLISION
SPI_ERR_OVERRUN
SPI_ERR_SLAVE_ABORT
```

More than one error condition can exist at the same time. Reported errors are for that case merged by a bit wise OR-operation.

Which error condition to report can are configured in `configure.h` using the parameters:

```
CFG_SPI_ERRORS_ALL
CFG_SPI_ERROR_COLLISION
CFG_SPI_ERROR_OVERRUN
CFG_SPI_ERROR_SLAVE_ABORT
```

It is sometimes necessary to disable an error condition that would otherwise prevent the SPI-controller form working. One such example is when a slave device has a non-inverted Slave Select pin it is necessary to disable the `SPI_ERR_COLLISION` error condition.

Example

```
#define MEM_READ_ARRAY     (unsigned char) b00000011

// Read the content from eeprom
void ReadEeprom(void)
   {
   char Data[33];
   unsigned short Counter;

   // Init slave select port
   io_Reset (IO_E);
   io_SetDataDirOut (IO_E,b10000000);
   io_Outp(IO_E, b10000000);

   // Init SPI (128 Kbaud default)
   spi_Init(_NULL);
```

```
// Insert read command as the first 3 bytes in the Data array
Data[0] = MEM_READ_ARRAY;
Data[1] = (unsigned char) 0x00; // Start address 0x000
Data[2] = (unsigned char) 0x00;

// Write data to memory
io_Outp(IO_E,b00000000);
spi_MasterTransceive((unsigned char[]) Data, 33);

// Wait for transaction to complete
while (spi_GetStatus() == SPI_STATUS_BUSY)
   ;

// Check if an overrun error occurred
if (spi_GetError() & SPI_ERR_OVERRUN)
   {

   // Handle error condition
   …
   }

…
```

spi_GetStatus()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
unsigned char spi_GetStatus()
```

Return

The status of the SPI controller.

Description

`SPI_GET_STATUS()` returns the status of the SPI-controller. Possible statuses are:

`SPI_STATUS_UNKNOWN`	-SPI controller not initialized
`SPI_STATUS_IDLE`	-SPI master idle
`SPI_STATUS_BUSY`	-SPI master busy with a transaction

Example

spi_HwIsBusy()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
unsigned char spi_HwIsBusy();
```

Return

The hardware status of the SPI controller.

Description

`spi_HwIsBusy()` returns a non-zero value (= Busy) when the SPI-controller transmits data to/from its shift register and zero (0) when SPI-controller hardware is idle. This is different from `spi_IsBusy()` in the sense that `spi_IsBusy()` can report that there is no ongoing transaction but the SPI controller hardware can, however, still be occupied with transmitting the data in its shift register. For such a case `spi_HwIsBusy()` will report *Busy*.

Example

```
...

// Write data
spi_MasterTranceive((unsigned char *) DataBuffer, BufferLength);

// Make sure the transaction has finished
while (spi_IsBusy()
   ;

// Make sure that the hardware also has finished
while (spi_HwIsBusy())
   ;

// Shut down the SPI-controller
spi_Close();
```

spi_Init()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
void spi_Init (unsigned char (*CallBack)(unsigned char RxChar));
```

Return

Description

`spi_Init()` initializes the SPI controller using the default configuration, which means:

- SPI port pins configured for their alternate function (SPI)
- 128 KBaud bit rate (only valid for 18.432 or 20.000 MHz system clock)
- Controller configured as master
- Clock idle at low level
- Half clock phase
- 8 bits character size
- Slave select pin configured for output
- SPI interrupt priority set to `IRQ_PRIO_NORM`
- SPI controller status set to `SPI_STATUS_IDLE`

Changes to the default setting can be made after the controller has been initialized.

When the SPI controller is operated as a master `CallBack` should be set to `_NULL`. When the SPI-controller is a slave, the parameter `CallBack` specifies the address of a call back function to be invoked when data has been received from an external SPI master. `CallBack` is of the type

```
unsigned char (*CallBack)(unsigned char RxChar)
```

where `RXChar` is the character received from the SPI master. The call back function executes during interrupt time wherefore the same limitations apply as for an interrupt handler. The parameter returned from `CallBack()` is transmitted back to the external SPI master upon the next transmission. The call back function has to handle error conditions as well. If an error condition is not cleared by `CallBack()` (see `spi_ClearError()`) the SPI communication will be disabled. After the error condition has been cleared the communication can be re-enabled by calling `spi_IrqEnable()`.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);

// Reconfigure
SPI_DISABLE();
SPI_SET_PHASE_WHOLE();
```

```
SPI_SET_CLK_IDLE_HI();
SPI_SET_BAUD_RATE(SPI_1_KBAUD);
SPI_ENABLE();
```

spi_IrqDisable()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_IrqDisable();
```

Return

Description

`spi_IrqDisable()` prevents the SPI-controller from sending interrupt requests to the interrupt controller.

Example

spi_IrqEnable()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_IrqEnable();
```

Return

Description

`spi_IrqEnable()` enables the SPI-controller to send interrupt requests to the interrupt controller.

Example

```
…
// Start the transaction by sending the first character
spi_Status = SPI_STATUS_BUSY;
*REG_OFFSET(SPI_0,SPI_DATA) = *Buffer;
spi_IrqEnable();
…
```

spi_IrqSend()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_IrqSend();
```

Return

Description

`spi_IrqSend()` causes the SPI-controller to assert a SPI-interrupt request to the interrupt controller.

Example

spi_IsBusy()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_IsBusy();
```

Return

The status of a master transceive SPI-transaction.

Description

`spi_IsBusy()` returns a non-zero value when the SPI-controller is busy handling a transaction issued by `spi_MasterTransceive()`. When no transaction is in progress a 0 (zero) is returned. This is, however, not necessarily the same as all data exchange between master and slave has finished. The SPI controller hardware can under some circumstances still be occupied by transmitting the last bits of data to/from its shift register.

When `spi_IsBusy()` reports *Not Busy* (0) a subsequent call to `spi_MasterTransceive()` will not block.

Example

```
…

// Issue a slave select
io_Outp(IO_E,b00000000);

// Write data to memory
spi_MasterTranceive((unsigned char *) Data, BufferLength);

// Wait for transaction to complete
while (spi_IsBusy())
   {
   ser_Write(SER_PORT, (unsigned char *) "SPI busy\r\n", 10);
   }

…
```

spi_MasterTransceive()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
unsigned char spi_MasterTransceive(unsigned char *Buffer, unsigned
short BufferSize);
```

Return

Description

`spi_MasterTransceive()` sends `BufferSize` number of character stored in `Buffer` to the SPI-slave that is currently selected. At the same time, `BufferSize` number of characters are received from the slave and stored in `Buffer`. Thus, the content of `Buffer` will be sent to the slave and it will also be overwritten with new data received from the slave. The size of Buffer consequently has to be large enough to hold both the data to be transmitted and the data to be received. An example of `Buffer` layout before calling `spi_MasterTransceive()` is:

Cmd1	Cmd2	Dummy	Dummy	Dummy

where `Cmd1` and `Cmd2` are command characters to be sent to the slave. For example, `Cmd1` could be a read command and `Cmd2` could be the start address at which the data should be retrieved. After `spi_MasterTransceive()` has finished `Buffer` could look like:

Dummy	Dummy	Chr1	Chr2	Chr3

where `Chr1`, `Chr2` and `Chr3` is the content of the memory addresses read.

The slave is selected either by the dedicated *Slave Select* signal or by any other port pin used for the same purpose. Note that it may be necessary to take timing requirements into account between repetitive *Slave Select* and *Slave Deselect*.

`spi_MasterTransceive()` is a non-blocking function and returns as soon as the transaction has been setup. This means that `Buffer` must be valid during the entire timeframe of the transaction. The following piece of code is an example of how **not** to perform a transaction:

```
void Communicate(void)
   {
   unsigned char DataToSendReceive[8] = {…};

   // WRONG!!! DataToSendReceive will be invalid as
   // soon as Communicate() returns.
   spi_MasterTransceive(DataToSendReceive, 8);
   }

void main(void)
```

```
{
Communicate();

// Continue program
…
}
```

The array `DataToSendReceive` will be invalid as soon as the program returns from `Communicate()` but the SPI device driver will continue to use the `Buffer` memory as if it was still valid.

Example

```
#define MEM_READ_STATUS    (unsigned char) b00000101

// Read the status of a M95320-WBN6 Serial SPI 4K*8 EEPROM
void main(void)
   {
   unsigned char Data[3];
   unsigned char Command;
   unsigned short Counter = 0;

   // Init slave select port
   io_Reset (IO_E);
   io_SetDataDirOut (IO_E,b10000000);
   io_Outp(IO_E, b10000000);

   // Init SPI
   spi_Init(_NULL);

   // Reconfigure the baud rate from default setting
   spi_Disable();
   spi_SetBaudRate(SPI_1_KBAUD);
   spi_Enable();

   // Setup command for reading memory status
   Data[0] = MEM_READ_STATUS;

   // Clear the bytes that should store memory status
   Data[1] = (unsigned char) 0x00;
   Data[2] = (unsigned char) 0x00;
```

```
// Select the slave circuit
io_Outp(IO_E, b00000000);

// Send/Receive data
spi_MasterTransceive(Data, 3);

// Wait for the transaction to complete (including hardware since
// the slave select is immediately deasserted).
while (spi_IsBusy() || spi_IsHwBusy())
   ;

// Deselect slave circuit
io_Outp(IO_E, b10000000);

// Status now stored in Data[1] and Data[2]
…
}
```

spi_Reset()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_Reset();
```

Return

Description

`spi_Reset()` resets the SPI-controller and puts it in a defined stage. This means:

- SPI disabled
- Slave mode
- Interrupt request generation disabled
- Baud rate timer disabled
- Clock polarity set to *Clock Idle Low*
- Clock phase set to *Half Period*
- Open-drain mode disabled
- Slave select pin configured as an input pin
- 8 data bits per character
- Baud rate set to `0xffff`

Example

spi_SetBaudRate()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetBaudRate(unsigned short BaudRate);
```

Return

Description

`spi_SetBaudRate()` sets the baud rate, at which the SPI-controller will communicate, to `BaudRate`. How the baud is calculated can be found in the documentation of the Z8 Encore!. The slave and the master have to use the same rate to be able to communicate. When operating the MCU core clock at 18.432 MHz or 20.000 MHz the following predefined baud rates are available:

```
SPI_1_KBAUD
SPI_2_KBAUD
SPI_4_KBAUD
SPI_8_KBAUD
SPI_16_KBAUD
SPI_32_KBAUD
SPI_64_KBAUD
SPI_128_KBAUD
SPI_256_KBAUD
SPI_512_KBAUD
SPI_1024_KBAUD
SPI_2048_KBAUD
```

Example

```
// Init SPI
spi_Init(_NULL);

// Set 1 KBaud communication rate
spi_Disable();
spi_SetBaudRate(SPI_1_KBAUD);
spi_Enable();
```

spi_SetCharSize()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetCharSize(unsigned char NumBits);
```

Return

Description

`spi_SetCharSize()` set the size, in number of bits, of each character transmitted over the SPI-bus to `NumBits`. Valid values for `NumBits` are:

```
SPI_1_BITS
SPI_2_BITS
SPI_3_BITS
SPI_4_BITS
SPI_5_BITS
SPI_6_BITS
SPI_7_BITS
SPI_8_BITS
```

Example

```
// Init SPI
spi_Init(_NULL);

// Change size of character from 8 to 4 bits
spi_Disable();
spi_SetCharSize(SPI_4_BITS)
spi_Enable();
```

spi_SetClkIdleHi()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetClkIdleHi()
```

Return

Description

`spi_SetClkIdleHi()` sets the polarity of the SPI clock signal (SCK) to be idle when driven high (CLKPOL=1).

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spi_SlaveSelLo();
spi_Disable();
spi_SetPhaseWhole();
spi_SetClkIdleHi();
spi_SetBaudRate(SPI_64_KBAUD);
spi_Enable();
```

spi_SetClkIdleLo()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include ""
spi_SetClkIdleLo();
```

Return

Description

`spi_SetClkIdleLo()` sets the polarity of the SPI clock signal (SCK) to be idle when driven low (CLKPOL=0). This is the default setting after the controller has been initialized by `spi_Init()`.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spi_Disable();
spi_SetPhaseHalf();
spi_SetClkIdleLo();
spi_SetBaudRate(SPI_128_KBAUD);
spi_Enable();
```

spi_SetModMaster()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetModMaster()
```

Return

Description

`spi_SetModMaster()` sets the SPI controller to operate as a SPI master. This is the default mode of operation.

Example

```
// Set alternate IO port functions for SPI
io_SetAltFunct(IO_C, IO_C_ALT_SPI);

// Set up bit rate
spi_SetBaudRate(SPI_128_KBAUD);

// Master mode
spi_SetModMaster();

// Clock idle low and half phase
spi_SetClkIdleLo();
spi_SetPhaseHalf();

// Slave select out and 8 bits characters
spi_SetCharSize(SPI_8_BITS);
spi_SetSlaveSelOut();
```

spi_SetModSlave()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetModSlave();
```

Return

Description

`spi_SetModSlave()` sets the SPI controller to operate as a SPI slave.

Example

```
// Init SPI controller
spi_Init(SpiSlaveCallback);

// Reconfigure SPI controller as slave
spi_Disable ();
spi_SetModSlave();
spi_Enable();
```

spi_SetPhaseHalf()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetPhaseHalf();
```

Return

Description

`spi_SetPhaseHalf()` specifies that a new data bit should written to the SPI-bus at the start of every half clock cycle (PHASE=0). The data is sampled at the next complete clock cycle. This is the default setting after the controller has been initialized by `spi_Init()` or `spi_Reset()`.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spi_Disable();
spi_SetPhaseHalf();
spi_Enable();
```

spi_SetPhaseWhole()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetPhaseWhole();
```

Return

Description

`spi_SetPhaseWhole()` specifies that a new data bit should written to the SPI-bus at the start of every whole clock cycle (PHASE=1). The data is sampled at the next half clock cycle.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spt_Disable ();
spi_SetPhaseWhole();
spi_Enable();
```

spi_SetSlaveSelIn()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetSlaveSelIn();
```

Return

Description

`spi_SetSlaveSelIn()` configures the designated *Slave Select* pin (SS) to be used as a slave select input pin. This is used when the SPI-controller is operating in slave mode.

Example

```
// Init SPI controller as slave
spi_Init(SpiSlaveCallback);
spi_Disable();
spi_SetSlaveSelIn();
spi_SetModSlave();
spi_Enable();
```

spi_SetSlaveSelOut()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SetSlaveSelOut();
```

Return

Description

`spi_SetSlaveSelOut()` configures the designated *Slave Select* pin (SS) to be used as an output pin. This is the default setting after the controller has been initialized by `spi_Init()`.

Example

```
// Init SPI
spi_Init(_NULL);

// Configure SS as output (redundant)
spi_Disable();
spi_SetSlaveSelOut();
spi_Enable();
```

spi_SlaveSelHi()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SlaveSelHi()
```

Return

Description

`spi_SlaveSelHi()` drives the designated *Slave Select* pin (SS) high. A generic IO pin can also be used to handle Chip Select.

Example

```
// Init SPI controller and change default settings
spi_Init(_NULL);
spi_SlaveSelLo();
spi_Disable();
spi_SetPhaseWhole();
spi_SlaveSelHi();
spi_SetBaudRate(SPI_64_KBAUD);
spi_Enable();

// Select temperature sensor DS1722
// Note DS1722 has a non-inverting chip-select pin
spi_SlaveSelHi();

// Init temperature sensor
spi_MasterTransceive(Data,2);
```

spi_SlaveSelLo()

Module

Serial Peripheral Interface (SPI)

Synopsis

```
#include "spi.h"
spi_SlaveSelLo()
```

Return

Description

`spi_SlaveSelLo()` drives the designated *Slave Select* pin (SS) low. A generic IO pin can also be used to handle Chip Select.

Example

```
// Select the DS1722 temperature sensor (Note DS1722 has a
// non-inverting chip-select pin)
spi_SlaveSelHi();

// Init temp sensor
spi_MasterTransceive(Data,2);

// Wait for transaction to complete
while (spi_IsBusy())
   ;

// Release sensor
spi_SlaveSelLo();
```

tck_Close()

Module

Ticker

Synopsis

```
#include "ticker.h"
void tck_Close(void);
```

Return

Description

`tck_Close()` stops the ticker timer thereby preventing the ticker to tick.

Example

```
…
// Init a 2 millisecond periodic ticker
tck_Init(TCK_2_ms_REL,TCK_2_ms_PRES, 2);

// Do some processing
…

// Wait a while (30 milliseconds)
tck_WaitMs(30);

// Stop and close ticker.
tck_Close();
```

tck_GetResolution()

Module

Ticker

Synopsis

```
#include "ticker.h"
unsigned short tck_GetResolution(void);
```

Return

The resolution of the ticker.

Description

`tck_GetTicks()` returns the number of milliseconds elapsing between two sequential ticker ticks. This value the same as used when initializing the ticker with `tck_Init()`.

Example

tck_GetTicks()

Module

Ticker

Synopsis

```
#include "ticker.h"
tck_TickVar tck_GetTicks(void);
```

Return

The number of ticks that has passed since the ticker was initiated by `tck_Init()`.

Description

`tck_GetTicks()` returns the number of ticker periods that have elapsed since the ticker was initiated. The data type `tck_TickVar` is an unsigned integer variable who's size is specified in the configuration file `configure.h`. Possible sizes are:

```
CFG_TCK_TICK_8     // 0 - 0xff (255)
CFG_TCK_TICK_16    // 0 - 0xffff (65535)
CFG_TCK_TICK_32    // 0 - 0xffffffff (4294967295)
```

`tck_GetTicks()` performs an atomic read of the ticker counter.

Example

```
…

tck_TickVar StartTime;
tck_TickVar ProcessingTime;

// Init a 2 millisecond periodic ticker
tck_Init(TCK_2_ms_REL,TCK_2_ms_PRES, 2);
StartTime = tck_GetTicks();

// Do some processing
…

// Measure how long time the processing took measured as number
// of 2 millisecond cycles the processing took.
ProcessingTime = tck_GetTicks() – StartTime;

…
```

tck_Init()

Module

Ticker

Synopsis

```
#include "ticker.h"
void tck_Init(unsigned short ReLoadVal, unsigned short PreScalVal,
unsigned short Resolution);
```

Return

Description

`tck_Init()` initialize the ticker setting the ticker period to the frequency resulting from the timer reload value `ReLoadVal` and prescaler value `PreScalVal`. Since the Z8 Encore! MCU doesn't have knowledge at what core frequency it is running the ticker needs to know how many *milliseconds* the ticker period represents. This is given by the `Resolution` parameter. Which timer source the ticker uses is given by the ticker configuration in the `configure.h` configuration file. Depending on which time source the ticker uses there are some predefined reload and prescaler value for microcontroller core frequencies 18.432 MHz and 20.000 Mhz:

Timer 0-3

1 millisecond
`TCK_1_ms_REL`
`TCK_1_ms_PRES`

2 milliseconds
`TCK_2_ms_REL`
`TCK_2_ms_PRES`

5 milliseconds
`TCK_5_ms_REL`
`TCK_5_ms_PRES`

10 milliseconds
`TCK_10_ms_REL`
`TCK_10_ms_PRES`

50 milliseconds
`TCK_50_ms_REL`
`TCK_50_ms_PRES`

100 milliseconds
`TCK_100_ms_REL`
`TCK_100_ms_PRES`

200 milliseconds
`TCK_200_ms_REL`
`TCK_200_ms_PRES`

400 milliseconds
TCK_400_ms_REL
TCK_400_ms_PRES

Uart 0-1
1 millisecond
TCK_1_ms_REL
TCK_1_ms_PRES

2 milliseconds
TCK_2_ms_REL
TCK_2_ms_PRES

5 milliseconds
TCK_5_ms_REL
TCK_5_ms_PRES

10 milliseconds
TCK_10_ms_REL
TCK_10_ms_PRES

50 milliseconds
TCK_50_ms_REL
TCK_50_ms_PRES

SPI 0 and I2C 0
1 millisecond
TCK_1_ms_REL
TCK_1_ms_PRES

2 milliseconds
TCK_2_ms_REL
TCK_2_ms_PRES

The reason that not all ticker time sources supports the same predefined ticker periods is that only timers 0-3 have prescalers and that Uart 0-1 as well as SPI 0 and I2c 0 run at different baud rate clocks. For systems running at other core frequencies than 18.432 MHz or 20.000 MHz the reload and prescale values have to be calculated according to the hardware documentation to match a specific ticker resolution.

Example

```
…
// Init a 2 millisecond periodic ticker
tck_Init(TCK_2_ms_REL,TCK_2_ms_PRES, 2);

// Do some processing
…

// Wait a while (30 milliseconds)
tck_WaitMs(30);

// Stop and close ticker.
tck_Close();
```

tck_WaitMs()

Module

Ticker

Synopsis

```
#include "ticker.h"
void tck_WaitMs(unsigned short MilliSec);
```

Return

Description

`tck_WaitMs()` postpones (delays) the execution for `MilliSec` number of milliseconds. The accuracy of the delay is somewhere between 0 and 2 ticker periods.

NOTE
`tck_WaitMs()` only postpones the current path of execution. Concurrent execution paths such as interrupt service routines will not be delayed unless `tck_WaitMs()` is part of the interrupt service routine. This is however not recommended.

Example

```
…

// Init ticker
tck_Init(TickerReloadValue_1ms, TickerPrescaleValue_1ms, 1);

// Wait for 50 ms for peripherals to power up
tck_WaitMs(50);

// Send init command to peripheral unit
SendCommand(0x38);

// Wait 10 ms for peripheral unit to process command
tck_WaitMs(10);

// Send configuration command
SendCommand(0x0c)

// Wait 5 ms for peripheral unit to process command
tck_WaitMs(5);

…
```

tmr_ClearPolarity()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_ClearPolarity(reg_addr TimerID);
```

Return

Description

`tmr_ClearPolarity()` sets (clears) the polarity bit of timer `TimerID` to 0 (zero). This is default value. How this affects the timer output depends on the operation mode and is clarified in the Z8 Encore! product specification. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

```
…
// Set up timer 0
tmr_Reset(TMR_0);
tmr_SetMode (TMR_0, TMR_MODE_CAPTURE);
tmr_SetPrescaler (TMR_0, TMR_PRES_DIV_32);
tmr_SetCounter (TMR_0, 0xffff);
tmr_SetReload (TMR_0, 0xffff);

// Count is captured on the rising edge
// of the timer input signal
tmr_ClearPolarity(TMR_0);

…
```

tmr_Disable()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_Disable(reg_addr TimerID);
```

Return

Description

`tmr_Disable()` disables and stops timer `TimerID`.

Example

```
…
// Setup one shot timer
tmr_Reset(TMR_1);
tmr_SetMode (TMR_1, TMR_MODE_ONE_SHOT);
tmr_SetPrescaler (TMR_1, TMR_PRES_DIV_32);

// Count half of the reload value
tmr_SetCounter (TMR_1, 0x7fff);
tmr_SetReload (TMR_1, 0xffff);

// Setup interrupt
IRQ_SET_VECTOR (IRQ_TMR_1, isr_Handler);
irq_SetPriority (IRQ_TMR_1, IRQ_PRIO_LOW);
irq_Enable (IRQ_TMR_1);

// Start timer
tmr_Enable(TMR_1);

// Use the timer for some processing
…

// Stop the timer
tmr_Disable(TMR_1);
```

tmr_Enable()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_Enable(reg_addr TimerID);
```

Return

Description

`tmr_Enable()` enables and starts timer `TimerID` using its current configuration settings.

Example

```
// Setup one shot timer
tmr_Reset(TMR_1);
tmr_SetMode (TMR_1, TMR_MODE_ONE_SHOT);
tmr_SetPrescaler (TMR_1, TMR_PRES_DIV_32);

// Count half of the reload value
tmr_SetCounter (TMR_1, 0x7fff);
tmr_SetReload (TMR_1, 0xffff);

// Setup interrupt
IRQ_SET_VECTOR (IRQ_TMR_1, isr_Handler);
irq_SetPriority (IRQ_TMR_1, IRQ_PRIO_LOW);
irq_Enable (IRQ_TMR_1);

// Start timer
tmr_Enable(TMR_1);
```

tmr_GetCounter()

Module

Timer

Synopsis

```
#include "timer.h"
unsigned short tmr_GetCounter(reg_addr TimerID);
```

Return

The current timer count value for timer `TimerID`.

Description

`tmr_GetCounter()` reads the current count value of timer `TimerID`. Since the value changes (decreases) during timer operation is read in an atomic manner. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

```
unsigned short CountValue = 0;

// Setup one shot timer
tmr_Reset(TMR_1);
tmr_SetMode (TMR_1, TMR_MODE_ONE_SHOT);
tmr_SetPrescaler (TMR_1, TMR_PRES_DIV_32);

// Set count value
tmr_SetCounter (TMR_1, 0x7fff);

// Start timer
tmr_Enable(TMR_0);

// Wait a little bit
Delay();

// Read the count value (should now be something less than 0x7fff)
CountValue = tmr_GetCounter(TimerID);
```

tmr_GetMode()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_GetMode(reg_addr TimerId);
```

Return

The current operation mode of timer `TimerID`.

Description

`tmr_GetMode()` returns the operation mode of timer TimerID. Valid timer id's are:

```
TMR_0     – Timer 0
TMR_1     – Timer 1
TMR_2     – Timer 2
TMR_3     – Timer 3
```

Returned timer mode is one of the following:

```
TMR_MODE_ONE_SHOT
TMR_MODE_CONTINUE
TMR_MODE_COUNTER
TMR_MODE_PWM
TMR_MODE_CAPTURE
TMR_MODE_COMPARE
TMR_MODE_GATED
TMR_MODE_CAPT_COMP
```

These modes correspond directly to the various modes of the hardware timers.

Example

tmr_GetPolarity()

Module

Timer

Synopsis

```
#include "timer.h"
unsigned char tmr_GetPolarity(reg_addr TimerID);
```

Return

The polarity setting of timer `TimerID`.

Description

`tmr_GetPolarity()` returns the polarity setting of timer `TimerID`. A non-zero value means that the polarity bit is set whereas 0 (zero) means that the polarity bit is cleared. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

tmr_GetPrescaler()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_GetPrescaler(reg_addr TimerID);
```

Return

The prescaler divider value of timer `TimerID`.

Description

`tmr_GetPrescaler()` returns the current prescaler divider value for timer `TimerID`. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Returned prescaler value is one of the following:

`TMR_PRES_DIV_1`	– Divide by 1
`TMR_PRES_DIV_2`	– Divide by 2
`TMR_PRES_DIV_4`	– Divide by 4
`TMR_PRES_DIV_8`	– Divide by 8
`TMR_PRES_DIV_16`	– Divide by 16
`TMR_PRES_DIV_32`	– Divide by 32
`TMR_PRES_DIV_64`	– Divide by 64
`TMR_PRES_DIV_128`	– Divide by 128

Example

tmr_GetPWM()

Module

Timer

Synopsis

```
#include "timer.h"
unsigned short tmr_GetPWM(reg_addr TimerID);
```

Return

The current setting of the PWM register for timer TimerID

Description

`tmr_GetPWM()` returns the current PWM value of timer `TimerID`. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

tmr_GetReload()

Module

Timer

Synopsis

```
#include "timer.h"
unsigned short tmr_GetReload(reg_addr TimerID);
```

Return

The current reload value of timer `TimerID`.

Description

`tmr_GetReload()` returns the current reload or initiation value of timer `TimerID`. This is the value that the timer is reloaded with, e.g. every time a new timer period starts when operated in continuous mode. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

```
…

unsigned short ReloadValue = 0;

// Setup timer
tmr_Reset(TMR_0);
tmr_SetMode (TMR_0, TMR_MODE_CONTINUE);
tmr_SetPrescaler (TMR_0, TMR_PRES_DIV_32);

// Set timer period
tmr_SetCounter (TMR_0, 0xffff);
tmr_SetReload (TMR_0, 0xffff);

// Start timer
tmr_Enable(TMR_0);

// Wait some time
Delay();

// Reload should still be 0xffff
ReloadValue = tmr_GetReload(TMR_0);

…
```

tmr_Reset()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_Reset(reg_addr TimerID);
```

Return

Description

`tmr_Disable()` resets timer TimerID and put it in a known state. This means:

- Count value set to 0x0000
- Reload value set to 0xFFFF
- PWM value set to 0x0000
- Timer disabled
- Polarity cleared
- Prescaler set to 0
- Timer mode set to one-shot

Example

```
…
// Setup one shot timer
tmr_Reset(TMR_1);
tmr_SetMode (TMR_1, TMR_MODE_ONE_SHOT);   // Redundant
tmr_SetPrescaler (TMR_1, TMR_PRES_DIV_32);

// Count half of the reload value
tmr_SetCounter (TMR_1, 0x7fff);
tmr_SetReload (TMR_1, 0xffff);

…
```

tmr_SetCounter()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_SetCounter(reg_addr TimerID, unsigned short Value);
```

Return

Description

`tmr_SetCounter()` sets the count value for timer `TimerID` to `Value`. `tmr_SetCounter()` is not guaranteed to be an atomic operation wherefore setting the count value while the timer is enabled may give unpredictable results. Therefore the count value is preferably set while the timer is disabled by `tmr_Disable()`. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

```
…

// Setup one shot timer
tmr_Reset(TMR_1);
tmr_SetMode (TMR_1, TMR_MODE_ONE_SHOT);
tmr_SetPrescaler (TMR_1, TMR_PRES_DIV_32);

// Set count value
tmr_SetCounter (TMR_1, 0x7fff);

…
```

tmr_SetMode()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_SetMode(reg_addr TimerID, unsigned char Mode);
```

Return

Description

`tmr_SetMode()` configures the operation mode of timer `TimerID` to `Mode`. Valid timer id's are:

```
TMR_0     – Timer 0
TMR_1     – Timer 1
TMR_2     – Timer 2
TMR_3     – Timer 3
```

Valid timer modes are:

```
TMR_MODE_ONE_SHOT
TMR_MODE_CONTINUE
TMR_MODE_COUNTER
TMR_MODE_PWM
TMR_MODE_CAPTURE
TMR_MODE_COMPARE
TMR_MODE_GATED
TMR_MODE_CAPT_COMP
```

These modes correspond directly to the various modes of the hardware timers.

Example

```
…

// Set up timer 0
tmr_Reset(TMR_0);
tmr_SetMode (TMR_0, TMR_MODE_CONTINUE);
tmr_SetPrescaler (TMR_0, TMR_PRES_DIV_32);
tmr_SetCounter (TMR_0, 0xffff);
tmr_SetReload (TMR_0, 0xffff);

// Start timer
tmr_Enable(TMR_0);

…
```

tmr_SetPolarity()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_SetPolarity(reg_addr TimerID);
```

Return

Description

`tmr_SetPolarity()` sets the polarity bit of timer `TimerID` to 1 (one). Default value is 0 (zero). How this affects the timer output depends on the operation mode and is clarified in the Z8 Encore! product specification. Valid timer id's are:

`TMR_0` – Timer 0
`TMR_1` – Timer 1
`TMR_2` – Timer 2
`TMR_3` – Timer 3

Example

```
…
// Set up timer 0
tmr_Reset(TMR_0);
tmr_SetMode (TMR_0, TMR_MODE_CAPTURE);
tmr_SetPrescaler (TMR_0, TMR_PRES_DIV_32);
tmr_SetCounter (TMR_0, 0xffff);
tmr_SetReload (TMR_0, 0xffff);

// Count is captured on the falling edge
// of the timer input signal
tmr_SetPolarity(TMR_0);

…
```

tmr_SetPrescaler ()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_SetPrescaler(reg_addr TimerID, unsigned char Prescale);
```

Return

Description

`tmr_SetPrescaler()` sets the prescaler divider of timer `TimerID` to `Prescale`. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Valid prescale divider constants is one of the following:

`TMR_PRES_DIV_1`	– Divide by 1
`TMR_PRES_DIV_2`	– Divide by 2
`TMR_PRES_DIV_4`	– Divide by 4
`TMR_PRES_DIV_8`	– Divide by 8
`TMR_PRES_DIV_16`	– Divide by 16
`TMR_PRES_DIV_32`	– Divide by 32
`TMR_PRES_DIV_64`	– Divide by 64
`TMR_PRES_DIV_128`	– Divide by 128

Example

```
…
// Set up timer 0
tmr_Reset(TMR_0);
tmr_SetMode (TMR_0, TMR_MODE_CONTINUE);
tmr_SetPrescaler (TMR_0, TMR_PRES_DIV_32);
tmr_SetCounter (TMR_0, 0xffff);
tmr_SetReload (TMR_0, 0xffff);
…
```

tmr_SetPwm()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_SetPwm(reg_addr TimerID, unsigned short Value);
```

Return

Description

`tmr_SetPwm()` sets the time (or count value) at which the PWM output signal for timer `TimerID` should toggle. The entire period time is given by the reload value. When the polarity is set (by `tmr_SetPolarity()`) the output signal starts as high (1) and then transitions to low when the timer count value matches the PWM value set by `tmr_SetPwm()`. The output stays low until the time count value reaches the reload value, wt which the signal returns to high. If the polarity is cleared (set to 0) by `tmr_ClearPolarity()` to opposite/inverted levels are used. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

…

```
// Configure timer TMR_0 to output a PWM signal with a high period
// being 25% of the entire PWM period.
tmr_SetReload (TMR_0, 1000);
tmr_SetPwm (TMR_0, 250);
```

…

tmr_SetReload()

Module

Timer

Synopsis

```
#include "timer.h"
tmr_SetReload(reg_addr TimerID, unsigned short Value);
```

Return

Description

`tmr_SetReload()` sets the reload or initial counting value of time `TimerID` to `Value`. This is the same as the start value for the timer. Valid timer id's are:

`TMR_0`	– Timer 0
`TMR_1`	– Timer 1
`TMR_2`	– Timer 2
`TMR_3`	– Timer 3

Example

```
…
// Setup timer
tmr_Reset(TMR_0);
tmr_SetMode (TMR_0, TMR_MODE_CONTINUE);
tmr_SetPrescaler (TMR_0, TMR_PRES_DIV_32);

// Set timer period
tmr_SetCounter (TMR_0, 0xffff);
tmr_SetReload (TMR_0, 0xffff);

// Turn on interrupt
irq_SetPriority (IRQ_TMR_0, IRQ_PRIO_LOW);
irq_Enable (IRQ_TMR_0);

// Start timer
tmr_Enable(TMR_0);

…
```

txt_htoa()

Module

Text

Synopsis

```
#include "text.h"
char *txt_htoa(unsigned short Value, char *Text);
```

Return

A pointer to `Text`.

Description

`txt_htoa()` converts the unsigned short `Value` to text using hexadecimal representation.

Example

```
…
char Text[12];
txt_htoa(0xf0a3,Text);

// Text is now “f0a3”

…
```

txt_lhtoa()

Module

Text

Synopsis

```
#include "text.h"
char *txt_lhtoa(unsigned long Value, char *Text);
```

Return

A pointer to `Text`.

Description

`txt_lhtoa()` converts the unsigned long `Value` to text using hexadecimal representation.

Example

```
…
char Text[12];
txt_lhtoa(0xff0a3,Text);

// Text is now “000ff0a3”

…
```

WDT_SET_RELOAD()

Module

Watchdog timer

Synopsis

```
#include "wdt.h"
WDT_SET_REALOAD(RelUpr, RelHi, RelLo);
```

Return

Description

`WDT_SET_REALOAD()` is a macro that sets the watchdog reload value (timeout) to the 24-bit integer [`RelUpr`, `RelHi`, `RelLo`], most significant byte first. The setting is atomic and follows the sequence as stated by the Z8 Encore! product specification.

NOTE
Hexadecimal values has to end with an “h” owing to the fact that the provided reload values are inserted directly into assembler code.

Example

```
// Init wdt (0.02 s time-out)
WDT_SET_RELOAD(0h, 03h, e7h);

// Start the wdt.
wdt_Refresh();
```

wdt_GetStatus()

Module

Watchdog timer

Synopsis

```
#include "wdt.h"
unsigned char wdt_GetStatus();
```

Return

The current watchdog timer status from the control register.

Description

`wdt_GetStatus()` reads the content of the WDT control register. Possible values returned are:

```
WDT_STAT_PWR_ON_RST
WDT_STAT_STOP_RECOV
WDT_STAT_TIME_OUT
WDT_STAT_EXT_RST
```

These correspond directly to the bits in the WDT control register. Several bits can be set at the same time.

Example

```
#pragma interrupt
void isr_Handler (void)
   {
   unsigned char RtnVal;

   // Set IRQ vector
   IRQ_SET_VECTOR(IRQ_WDT,isr_Handler);

   // Perform some useful response to the WDT time out
   …

   // Clear the WDT condition by reading the WDT control register
   RtnVal = wdt_GetStatus();
   }
```

wdt_Refresh()

Module

Watchdog timer

Synopsis

```
#include "wdt.h"
wdt_Refresh();
```

Return

Description

`wdt_Refresh()` resets the watchdog timer to start a new countdown from the previously set timeout (reload) value.

Example

```
…
// Init wdt (0.02 s time-out)
WDT_SET_RELOAD(0h, 03h, e7h);

// Start the WDT
wdt_Refresh();
while(1)
   {
   PerformCriticalProcessing();

   // Refresh WDT to prevent it from fireing
   wdt_Refresh();
   }
…
```

6. Afterburner

The documentation in this section is paragraphs and texts that are currently not supported and/or implemented in the official releases. At some point in time the information herein were valid but have since been put on the afterburner. The information may, however, be used in the future.

6.1. Serial module

SER_MPROC_DISABLE()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
SER_MPROC_DISABLE(Uart);
```

Return

Description

`SER_MPROC_DISABLE(Uart)` disable multiprocessor mode for serial controller `Uart`. When multiprocessor mode is disabled the serial controller operates as a standard point-point serial communication controller. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

SER_MPROC_ENABLE()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
SER_MPROC_ENABLE(Uart);
```

Return

Description

`SER_MPROC_ENABLE(Uart)` enables multiprocessor mode for serial controller `Uart`. When the controller is in multiprocessor mode it can operate as a node on a serial network (e.g. RS485). The exact behavior of the controller as a network node is subsequently determined by `SER_MPROC_OFF(Uart)`, `SER_MPROC_ON(Uart)`, `SER_CLEAR_MPROC_BIT(Uart)` and `SER_SET_MPROC_BIT(Uart)`. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
ser_Init(SER_0, 0);

// Temporary disable controller
SER_RX_DISABLE(SER_0);
SER_TX_DISABLE(SER_0);

// Enable multiprocessor mode
SER_MPROC_ENABLE(Uart);

// Only receive data where the multiprocessor bit is set
SER_MPROC_ON(SER_0);

// Fire up the controller again
SER_RX_ENABLE(SER_0);
SER_TX_ENABLE(SER_0);
```

SER_MPROC_OFF ()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
SER_MPROC_OFF(Uart);
```

Return

Description

When the serial controller `Uart` is operated in multiprocessor mode (`SER_MPROC_ENABLE()`), the macro `SER_MPROC_OFF()` will turn off the filter which rejects all data that hasn't got the 9th multiprocessor bit set. In other words, after `SER_MPROC_OFF()` has been called, the serial controller will process all data bytes received. This is the default setting. Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
ser_Init(SER_0, 0);

// Temporary disable controller
SER_RX_DISABLE(SER_0);
SER_TX_DISABLE(SER_0);

// Enable multiprocessor mode
SER_MPROC_ENABLE(Uart);

// Receive both multiprocessor data and non-multiprocessor data
// (default setting)
SER_MPROC_OFF(SER_0);

// Fire up the controller again
SER_RX_ENABLE(SER_0);
SER_TX_ENABLE(SER_0);
```

SER_MPROC_ON()

Module

Serial (RS-232)

Synopsis

```
#include "serial.h"
SER_MPROC_ON(Uart);
```

Return

Description

When the serial controller `Uart` is operated in multiprocessor mode (`SER_MPROC_ENABLE()`), the macro `SER_MPROC_ON()` turns on the filter which rejects all data that hasn't got the 9th multiprocessor bit set. In other words, after `SER_MPROC_ON()` has been called, the serial controller will only process data bytes in which the multiprocessor bit is set. The default setting is that the serial controller processes all data (even non-multiprocessor). Valid values for `Uart` are `SER_0` and `SER_1`.

Example

```
ser_Init(SER_0, 0);

// Temporary disable controller
SER_RX_DISABLE(SER_0);
SER_TX_DISABLE(SER_0);

// Enable multiprocessor mode
SER_MPROC_ENABLE(Uart);

// Only receive data where the multiprocessor bit is set
SER_MPROC_ON(SER_0);

…
```

SER_CLEAR_MPROC_BIT()

Module

Serial (RS232)

Synopsis

```
#include ""
ser_ClearMprocBit(reg_addr Uart);
```

Return

Description

`ser_ClearMprocBit()` sends a 0 (zero) in the multiprocessor bit location (the 9th bit) of the outgoing data stream for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

When the serial controller is used for frame based communication (i.e. RS-485) it is common that the 9th bit should be set for a few bytes only of the frame. Since the `ser_Write()` is buffered it is necessary to assure that the transmitter is idle before setting/clearing the 9th bit.

Example

```
// The fictitious frame is 4 bytes long of which the first has
// the 9th bit set
unsigned char Frame[4] = {1, 2, 3, 4};

// Make sure that the TX buffer is empty
while (SER_TX_IS_BUSY(SER_0))
   ;

// Make sure that the transmitter is done
while (SER_TX_HW_IS_BUSY(SER_0))
   ;

// Set the 9th bit
SER_SET_MPROC_BIT(SER_0);

// Send the first 2 bytes
ser_Write(SER_0, Frame, 2);

// Make sure that the TX buffer is empty
while (SER_TX_IS_BUSY(SER_0))
   ;

// Make sure that the transmitter is done
while (SER_TX_HW_IS_BUSY(SER_0))
   ;

// Clear the 9th bit
SER_CLEAR_MPROC_BIT(SER_0);
```

```
// Send the remaining 2 bytes
ser_Write(SER_0, Frame + 2, 2);
```

SER_SET_MPROC_BIT()

Module

Serial (RS232)

Synopsis

```
#include "serial.h"
SER_SET_MPROC_BIT(Uart);
```

Return

Description

`SER_SET_MPROC_BIT()` sends a 1 in the multiprocessor bit location (the 9th bit) of the outgoing data stream for serial controller `Uart`. Valid values for `Uart` are `SER_0` and `SER_1`.

When the serial controller is used for frame based communication it is common that the 9th bit should be set for a few bytes only of the frame. Since the `ser_Write()` is buffered it is necessary to assure that the transmitter is idle before setting/clearing the 9th bit.

Example

```
// The fictitious frame is 4 bytes long of which the first 2 have
// the 9th bit set
unsigned char Frame[4] = {1, 2, 3, 4};

// Initialize controller
ser_Init(SER_0, 0);

// Make sure that the TX buffer is empty
while (SER_TX_IS_BUSY(SER_0))
   ;

// Make sure that the transmitter is done
while (SER_TX_HW_IS_BUSY(SER_0))
   ;

// Set the 9th bit
SER_SET_MPROC_BIT(SER_0);

// Send the first 2 bytes
ser_Write(SER_0, Frame, 2);

// Make sure that the TX buffer is empty
while (SER_TX_IS_BUSY(SER_0))
   ;

// Make sure that the transmitter is done
while (SER_TX_HW_IS_BUSY(SER_0))
   ;
```

```
// Clear the 9th bit
SER_CLEAR_MPROC_BIT(SER_0);

// Send the remaining 2 bytes
ser_Write(SER_0, Frame + 2, 2);
```

6.2. LCD module

lcd_Close()

Module

Lcd

Synopsis

```
#include "lcd.h"
void lcd_Close(void);
```

Return

Description

`lcd_Close()` shuts down

Example