CODE TIME TECHNOLOGIES

Abassi RTOS

I2C Support

Disclaimer

Code Time Technologies Inc. provides this document “AS IS” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.
Code Time Technologies Inc. does not warrant that the contents of this document will meet your requirements or that the document is error-free. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. Code Time Technologies Inc. may make improvements and/or changes in the product(s) and/or program(s) described in the document at any time. This document does not imply a commitment by Code Time Technologies Inc. to supply or make generally available the product(s) described herein.

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the I2C driver used by Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The standalone version of the I2C driver is also described here.

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`???_i2c.h`	Include file for the I2C driver (??? is target dependent)
`???_i2c.c`	"C" file for the Abassi I2C driver (??? is target dependent)
`Demo_8_<PROC>_<TOOL>.c`	"C" file for testing. *<TOOL>* is the name of the build environment tool-set, *<PROC>* is the processor.
`SAL.h`	Include file for the standalone abstraction layer (supplied with standalone package only)
`SAL.c`	"C" file for the standalone abstraction layer (supplied with standalone package only)
`ISRhandler_???.s`	"ASM" add-on file for the standalone version. It contains support for both the driver and the demo application (supplied with standalone package only)

1.2 Features

The I2C driver API and build options are kept the same across all target platforms. Target specific extra functionality is not described in this document; refer to the code itself. When possible (i.e. GPIO muxing with I2C lines), if the platform I2C module does not support bus clearing upon bus lock-up, the driver will manually perform the bus clearing by muxing GPIO lines to the I2C clock line and toggling it. As much as possible, and depending on the target controller capabilities, the driver aborts transfers upon encountering an error and report the issue; e.g. TX FIFO under-run or RX FIFO overrun. Also monitored is the transfer time; if the transfer takes much longer time than the expected exchange time then the driver will abort the transfer and report the issue.

1.3 Limitations

Some controller cannot support some of the features described in this document. Please refer to the specific driver code for a description / list of these limitations; this is described near the top of the files `??_i2c.h` and `??_i2c.c`.

[1] When Abassi is mentioned in this document, unless explicitly stated, it always means Abassi, mAbassi and µAbassi.

2 Target Set-up

All there is to do to configure and enable the use of the I2C driver in an application based on Abassi is to include the following file in the build:

- `???_i2c.c` (For Abassi & standalone)
- `SAL.c` (For Standalone)
- `ISRhandler_???.s` (For Standalone)

and to set-up the include search directory order making sure the file `???_i2c.h` is found (and `SAL.h` for the standalone)

The I2C driver may or may not, depending on the target platform, be independent from other include files.

2.1 Build Options

There are a few build options that allow the I2C driver to be configured for the needs of the target application. The following table lists all of them:

Table 2-1 Build Options

File Name	Default	Description
OS_PLATFORM	Target dependent	Number indicating the target platform. Refer to `???_i2c.h` and `Platform.h` to see the list of supported platforms and the default one.
I2C_CLK	Target dependent	Clock frequency of the I2C controller.
I2C_MAX_DEVICES	Target dependent	Number of I2C controllers(s) available in the target platform.
I2C_LIST_DEVICE	Target dependent	Bit field selecting the I2C controller(s) to use. The default value is dependent on the build option `OS_PLATFORM`.
I2C_N_TRIES	1	When a bus transfer failure is detected, number of times the driver will try the same transfer again.
I2C_USE_MUTEX	1	Boolean used to activate the I2C driver internal protection for exclusive device access.
I2C_OPERATION	0x10101	Bit field defining how the I2C driver operates.
I2C_ISR_RX_THRS	50	Threshold in percentage of the RX FIFO size to trigger the RX interrupt.
I2C_ISR_TX_THRS	50	Threshold in percentage of the TX FIFO size to trigger the TX interrupt.
I2C_MIN_4_RX_DMA	2	Minimum number of bytes to read in order to use DMA transfers instead of polling.
I2C_MIN_4_TX_DMA	4	Minimum number of bytes to write in order to use DMA transfers instead of polling.
I2C_MIN_4_RX_ISR	4	Minimum number of bytes to read in order to use the interrupts instead of polling.

I2C_MIN_4_TX_ISR	2	Minimum number of bytes to write in order to use the interrupts instead of polling.
I2C_MIN_4_RX_ISR	2	Minimum number of bytes to read in order to use the interrupts instead of polling.
I2C_MIN_4_TX_ISR	2	Minimum number of bytes to write in order to use the interrupts instead of polling.
I2C_MULTICORE_ISR	0	Boolean to enable/disable the ISR handler to be used by multiple cores
I2C_TOUT_ISR_ENB	1	Boolean to enable/disable the timeout check for transfers done through polling
I2C_REMAP_LOG_ADDR	1	Boolean to enable/disable the conversion from logical to physical address with DMA transfers
I2C_ARG_CHECK	1	Boolean to enable/disable the check on the validity of the API function arguments
I2C_DEBUG	0	Boolean controlling the sending of progress / debug messages to stdout.

2.1.1 OS_PLATFORM

The build option OS_PLATFORM informs the I2C driver about the platform it is operating on. There are three benefits ensuing from the presence of this build option:

- The I2C driver implicitly knows the total number of I2C devices on the platform.
- The I2C driver is able to configure and reset the I2C devices without intervention of the application.
- Provides the implicit information on which GPIO port to use when the driver has to manually toggle the I2C data and clock lines to try to clear a bus lock-up.

The information on the numbering used for OS_PLATFORM is available in the Platform.txt and Platform.h files also supplied as part of the distribution.

2.1.2 I2C_MAX_DEVICES

The build option I2C_MAX_DEVICES informs the I2C driver on how many I2C controllers (devices) are on the target platform. If this build option is not set, then the I2C driver will rely on the build option I2C_LIST_DEVICE (Section 2.1.4). If the build option I2C_LIST_DEVICE is also not set, then the I2C driver will rely on the OS_PLATFORM value (Section 2.1.1).

2.1.3 I2C_CLK

The build option I2C_CLK defines the clock frequency the I2C controller operates with. A default value is set according to the target platform specified by OS_PLATFORM. If the module clock frequency is different from the default value, all there is to do is defined the build option I2C_CLK and set it to the clock frequency in Hz.

2.1.4 I2C_LIST_DEVICE

The build option I2C_LIST_DEVICE informs the I2C driver about the individual I2C controllers (devices) that are used by the application. When the target platform has multiple I2C devices, enabling only the devices used by the application offers a main benefit:

- Minimize the data memory required by the driver, as there is no need to reserve memory for the queue descriptors / buffers / interrupt handlers and semaphores or optional mutexes of unused devices.

This build option is a bit field, where the bit position represents the I2C device number; device numbering starts at 0. When the corresponding bit is cleared (reset to 0) it specifies the device is not used; when the corresponding bit is set to 1 then the device is used. The following table shows the valid combinations for a target platform with 2 I2C devices:

Table 2-2 Build Options

I2C_LIST_DEVICE	I2C #0	I2C #1
1	In use	Not used
2	Not used	In use
3	In use	In use

If the build option `I2C_LIST_DEVICE` is not externally defined, the default value will be set according to the build option `I2C_MAX_DEVICES` (Section 2.1.2). If the build option `I2C_MAX_DEVICES` is also not set, then `I2C_LIST_DEVICE` will be set according to the build option `OS_PLATFORM` (Section 2.1.1) and will make all the I2C devices available on the target platform.

2.1.5 I2C_N_TRIES

The I2C bus is not a foolproof bus. Bus transfer failures can happen, and when a failure is detected, the driver can be configured at build time to retry the transfer a number of times. The build option `I2C_N_TRIES` specifies if retries are performed and when so, the maximum number of times it retries. Setting the build option `I2C_N_TRIES` to a value of 0 will not make the driver retry a transfer upon failure. A positive value of ***N*** will make the driver retry a maximum number of ***N*** retries.

2.1.6 I2C_USE_MUTEX

In an RTOS environment, the driver can provide exclusive access protection to the I2C device(s) through its internal mutex(es). By default, the build option `I2C_USE_MUTEX` is set to a non-zero value, meaning the driver uses one mutex per device as the exclusive access protection mechanism. Defining and setting the build option `I2C_USE_MUTEX` to a zero value will configure the driver to not use mutexes, therefore the application has to enforce there be no concurrent accesses to the same device.

2.1.7 I2C_OPERATION

The build option `I2C_OPERATION` is used to configure how the I2C driver operates. This build option is a bit field. The meaning of each bits, (bit position #0 is the LSBit), is described in the following table:

Table 2-3 I2C_OPERATION bit definitions

Bit #	Description
0	Interrupts are disabled during a read (receive) burst. To not disable the interrupts during a read burst, bit #0 must be reset to zero. To disable the interrupts during a read burst, bit #0 must be set to 1.
1	Interrupts are used to empty the controller RX FIFO when performing a read (receive) operation and/or interrupts are used to report the end of transmission. To not allow the use of interrupts when reading the RX FIFO, bit #1 must be reset to zero. To allow use of interrupts when reading RX FIFO, bit #1 must be set to 1.
2	DMA is used to empty the controller RX FIFO when performing a read (receive) operation. To not allow the use of the DMA to read the RX FIFO, bit #2 must be reset to zero. To allow the use of the DMA to read the RX FIFO, bit #2 must be set to 1. The end of the transfer can be polled or blocked on an interrupt depending on the setting of bit #1.

8	Interrupts are disabled during a write (send) burst. To not disable the interrupts during a write burst, bit #8 must be reset to zero. To disable the interrupts during a write burst, bit #8 must be set to 1.
9	Interrupts are used to fill the controller TX FIFO when performing a write (send) operation and/or interrupts are used to report the end of transmission. To not allow the use of interrupts when filling the TX FIFO, bit #9 must be reset to zero. To allow the use of interrupts when filling TX FIFO, bit #9 must be set to 1.
10	DMA is used to fill the controller TX FIFO when performing a write (send) operation. To not allow the use of the DMA to fill the TX FIFO, bit #9 must be reset to zero. To allow the use of the DMA to fill the TX FIFO, bit #9 must be set to 1. The end of transfer can be polled or blocked on an interrupt depending on the setting of bit #9.
16	When bit #16 is set to 1, the bus clear operation will be performed upon an I2C transfer failure. If bit #16 is clear to 0, no bus clear will be performed.

2.1.8 I2C_ISR_RX_THRS

The build option `I2C_ISR_RX_THRS` is used to set the threshold, or watermark, at which the data receive interrupts are triggered. When interrupts are used to read from the I2C bus (`I2C_OPERATION` bit #1 set to 1), the RX interrupt is triggered when the RX FIFO holds more than a preset number of bytes. The build option `I2C_ISR_RX_THRS` specifies this threshold, in percentage of the FIFO size. Therefore only values between 0 and 100 are accepted for `I2C_ISR_RX_THRS`.

Each application has an optimal value for the RX threshold. To maximize the performance, the interrupt handler should ideally be entered exactly when the FIFO is full or very close to be full. As there is always a bit of latency between the time an interrupt is raised and when the interrupt handler starts retrieving data, the optimal threshold should be set to the number of bytes that are transferred on the I2C bus in the latency duration. Assuming a FIFO of 32 bytes and assuming an interrupt latency of 10 *u*s with a I2C bus of 3.2 MHz, then 4 bytes are read in 10 *u*s. This optimal threshold is located at 32-4 bytes, which is 28 or 88% of 32. In this example, the optimal value to set `I2C_ISR_RX_THRS` is 88.

Setting this threshold too low has the effect of increasing the number of interrupts and setting it too high will or could provoke overflows of RX FIFO, therefore loss of received bytes. Unless the controller performs clock stretching, if the RX FIFO overflows, bytes will be lost and the driver will abort the transfer.

This build option is ignored if bit #1 in `I2C_OPERATION` (Section 2.1.7) is reset to zero.

2.1.9 I2C_ISR_TX_THRS

The build option `I2C_ISR_TX_THRS` is used to set the threshold, or watermark, at which the data write interrupt is triggered. When interrupts are used to write to the I2C (`I2C_OPERATION` bit #9 set to 1), the TX interrupt is triggered when the write FIFO holds less than a preset number of bytes. The build option `I2C_ISR_TX_THRS` specifies this threshold, in percentage of the FIFO size. Therefore only values between 0 and 100 are accepted for `I2C_ISR_TX_THRS`.

Each application has an optimal value for the TX threshold. To maximize the performance the interrupt handler should in theory be starts filling the FIFO exactly when the FIFO is empty or very close to be empty. As there is always a bit of latency between the time an interrupt is raised and when the interrupt handler starts writing data, the optimal threshold should be set to the number of bytes that are transferred on the I2C bus in the latency duration.

Setting this threshold too high has the effect of increasing the number of interrupts and setting it too low will or could provoke under-runs of TX FIFO, therefore loss of transmitted bytes. If the controller does not support clock stretching, the TX FIFO will under-run and the bus transaction will either show the end of transfer conditions or a TX FIFO underflow error; due to that, the driver will abort transfer.

This build option is ignored if bit #9 in `I2C_OPERATION` (Section 2.1.7) is reset to zero.

2.1.10 I2C_MIN_4_RX_DMA

The build option `I2C_MIN_4_RX_DMA` is used to set the minimum number of bytes to be read for using DMA transfers. Setting up the DMA for a transfer always involves a certain amount of CPU overhead. When a small number of bytes are to be read, it is highly probable the time required to perform the read itself is less than the overall DMA set-up overhead. When the RX DMA transfers are enabled through the build option `I2C_OPERATION`, if the number of bytes to read is less than the value specified by `I2C_MIN_4_RX_DMA`, the read transfer is performed through polling or interrupts instead of using the DMA.

This build option is ignored if bit #2 in `I2C_OPERATION` (Section 2.1.7) is reset to zero.

When using `i2c_send_recv()` (Section 4.1.4), and when the controller natively supports the combined format, the minimum required of bytes to transfer uses an OR condition between `I2C_MIN_4_RX_DMA` and `I2C_MIN4_TX_DMA`.

2.1.11 I2C_MIN_4_TX_DMA

The build option `I2C_MIN_4_TX_DMA` is used to set the minimum number of bytes to be sent for using DMA transfers. Setting up the DMA for a transfer always involves a certain amount of CPU overhead. When a small number of bytes are to be sent, it is highly probable the time required to perform the sending itself is less than the overall DMA set-up overhead. When the TX DMA transfers are enabled through the build option `I2C_OPERATION`, if the number of bytes to send is less than the value specified by `I2C_MIN_4_TX_DMA`, the transfer is performed through polling or interrupts instead of using the DMA.

This build option is ignored if bit #10 in `I2C_OPERATION` (Section 2.1.7) is reset to zero.

When using `i2c_send_recv()` (Section 4.1.4), and when the controller natively supports the combined format, the minimum required of bytes to transfer uses an OR condition between `I2C_MIN_4_RX_DMA` and `I2C_MIN4_TX_DMA`.

2.1.12 I2C_MIN_4_RX_ISR

The build option `I2C_MIN_4_RX_ISR` is used to set the minimum number of bytes to be read for using the interrupts. The whole interrupt handling always involves a certain amount of CPU overhead: a task becoming blocked, interrupt handler operating, and unblocking the task. When a small number of bytes are to be read, it is highly probable the time required to perform the read itself is less than the overall interrupt overhead. When the RX interrupts are enabled through the build option `I2C_OPERATION`, if the number of bytes to read is less than the value specified by `I2C_MIN_4_RX_ISR`, the read transfer is performed through polling instead of using interrupts.

This build option is ignored if bit #1 in `I2C_OPERATION` (Section 2.1.7) is reset to zero.

When using `i2c_send_recv()` (Section 4.1.4), and when the controller natively supports the combined format, the minimum required of bytes to transfer uses an OR condition between `I2C_MIN_4_RX_ISR` and `I2C_MIN4_TX ISR`.

2.1.13 I2C_MIN_4_TX_ISR

The build option `I2C_MIN_4_TX_ISR` is used to set the minimum number of bytes to be written for using the interrupts. The whole interrupt handling always involves a certain amount of CPU overhead: a task becoming blocked, interrupt handler operating, and unblocking the task. When a small number of bytes are to be written, it is highly probable the time required to perform the write itself is less than the overall interrupt overhead. When the TX interrupt are enabled through the build option `I2C_OPERATION`, if the number of bytes to write is less than the value specified by `I2C_MIN_4_TX_ISR`, the write transfer is performed through polling instead of using interrupts.

This build option is ignored if bit #9 in `I2C_OPERATION` (Section 2.1.7) is reset to zero.

When using `i2c_send_recv()` (Section 4.1.4), and when the controller natively supports the combined format, the minimum required of bytes to transfer uses an OR condition between `I2C_MIN_4_RX_ISR` and `I2C_MIN4_TX ISR`.

2.1.14 I2C_TOUT_ISR_ENB

The build option `I2C_TOUT_ISR_ENB` is a Boolean controlling if interrupts are shortly re-enabled when checking for timeouts when performing a transfer through polling. When a transfer through polling is performed and the interrupts are disabled during the burst (controlled with `I2C_OPERATION`), this could makes the update of the RTOS timer tick impossible as the timer tick counter is updated through interrupts. The RTOS timer tick won't get updated is the interrupts are disabled on a single core target or when on a multicore target if the core where the driver is operating is the only one handling the RTOS timer tick interrupts. This is one case on a multi-core where it would be advisable to interrupt all core for the RTOS timer tick update.

The setting of `I2C_TOUT_ISR_ENB` does not affect the timeout check when the transfer is interrupt based or DMA based.

2.1.15 I2C_REMAP_LOG_ADDR

When the MMU is set-up to remap memory areas at different addresses from the physical address, it is necessary to convert the logical address to their physical equivalents when using DMA transfers. The build option `I2C_REMAP_LOG_ADDR` is a Boolean that selects if the addresses used by the DMA are converted from logical to physical. By default it is set to a non-zero value (enable). Although the remapping function is a low instruction count, one may want to not perform a redundant remapping when the logical addresses are the same as the physical. The remapping can be turned off setting the build option `I2C_REMAP_LOG_ADDR` to zero.

2.1.16 I2C_ARG_CHECK

The build options `I2C_ARG_CHECK` controls if the driver checks the validity of the API function arguments or not. This build option is a Boolean; when set to a non-zero value, the driver checks the validity of the arguments and returns an error code when the arguments are invalid. When set to a zero value, it does not check the validity of the arguments.

2.1.17 I2C_DEBUG

The build options `I2C_DEBUG` controls the printout of progress and error messages to `stdout`. This build option can have three set-up; when set to a value of zero or less, no messages are sent to `stdout`. When set 1, it sends over `stdout` the set-up information used during initialization and causes of error during the operaton. When set to a value greater than 1, it prints on `stdout` all operations and causes of errors.

3 Transfers

The way transfers are performed, through polling, ISR or DMA and how the end of transfer (EOT) is detected depend on a set of rules based on the number of data to transfer, the value of `I2C_OPERATION` and a few other build options. The descriptions are for the reception direction, the same applies for the transmit direction. When EOT is done with ISR, the driver uses a semaphore on which the calling task blocks until the transfer is completed or upon an expiry timeout / error occurred. The following rules are for the RX direction, the same applies for the TX direction, replacing bit #1 & #2 by bit #9 & #10 and replacing the `RX` by `TX` in the build options token names.

1. If bit 1 is clear (ISR transfers disabled) and bit #2 is clear (DMA transfers disable), the data is always transferred with polling.
2. If bit 1 is set (ISR transfers enable) and bit #2 is cleared (DMA transfers disabled) the transfer will be performed through ISRs only if all these two conditions are met:
   - `nData >= I2C_MIN_4_RX_ISR`
   - `nData > I2C_ISR_RX_THRS * RX_FIFO_SIZE / 100`
3. If bit 1 is clear (ISR transfers disable) and bit #2 is set (DMA transfers enable) the transfer will be performed through DMA only if all these two conditions are met:
   - `nData >= I2C_MIN_4_RX_ISR`
   - `nData > I2C_ISR_RX_THRS * RX_FIFO_SIZE / 100`
4. If bit 1 is set (ISR transfers enable) and bit #2 is set (DMA transfers enable) the transfer will be performed through ISRs, only if all these three conditions are met:
   - `nData >= I2C_MIN_4_RX_ISR`
   - `nData > I2C_ISR_RX_THRS * RX_FIFO_SIZE / 100`
   - The conditions in 3) are NOT met
5. If bit 1 is set (ISR transfers enable) and bit #2 is set (DMA transfers enable) the transfer will be performed through DMA, only if all these two conditions are met (when met, the conditions to validate an ISR transfer are not relevant):
   - `nData >= I2C_MIN_4_RX_ISR`
   - `nData > I2C_ISR_RX_THRS * RX_FIFO_SIZE / 100`

The detection / waiting for the end of transfer (EOT) is performed according to these rules:

1. If bit 1 is clear (ISR transfers disable), EOT is always performed with polling.
2. If bit 1 is set (ISR transfers enabled), EOT is performed with polling if the transfer is done through polling. If the transfer is performed with ISR or DMA, then EOT detection is done through ISRs

NOTE: Most controllers have a TX FIFO and when data is sent, the TX FIFO is always filled in bulk before enabling the transfer. Therefore if the total number of data to transmit all fits in the TX FIFO there won't be any transfer performed through polling, ISR or DMA.

NOTE: Depending on the target controller, when transfers happen, being either for receiving data or for sending data, the controller requires both filling the output data register or the TX FIFO and reading the input data register or the RX FIFO. For such controller, the individual setting for the ISR & DMA in the RX and TX direction applies to the internal operation. i.e. for data reception, as it is required to fill the output data register or the TX FIFO, the ISR / DMA TX setting applies when flling the output data register / TX FIFO.

4 API

In this section, the API of all common I2C driver functions is provided. The next section gives examples on how to use the I2C

4.1.1 i2c_init

Synopsis

```
#include "???_i2c.h"

void i2c_init(int Dev, int AddBits, int Freq);
```

Description

`i2c_init()` is the component used to initialize one I2C module. The module's controller number is indicated by the argument `Dev` and the address width (7 or 10 bits) is indicated by the argument `AddBits`. The I2C bus frequency is specified with the argument `Freq`.

Arguments

`Dev`	Module's controller number (Number starting at 0)
`AddBits`	I2C bus address width. Must be either set to a value of 7 or 10
`Freq`	Frequency of the bus specified in Hz.

Returns

`void`

Component type

Function

Options

Notes

See Also

4.1.2 i2c_recv

Synopsis

```
#include "???_i2c.h"

int i2c_recv(int Dev, unsigned int target, char *buffer, int len);
```

Description

`i2c_recv()` is the component used to perform a read operation (a transfer of data from a slave to the master). The I2C controller to use is indicated by the argument `Dev` and the slave address with the argument `target`. The number of bytes to read from the slave is specified by the argument `len` and the argument `buffer` is the base address of the buffer used to collect the data read from the slave.

Arguments

`Dev`	Module's device number (Number starting at 0)
`target`	I2C address of the slave to read from
`buffer`	Buffer that will hold the data read from the slave. This buffer must be size to at least `len` bytes
`len`	Number of bytes to read from the slave

Returns

`int`	`== 0 :` the transfer was successful
	`!= 0 :` the transfer failed

Component type

Function

Options

Notes

See Also

`i2c_send()`
`i2c_send_recv()`

4.1.3 i2c_send

Synopsis

```
#include "???_i2c.h"

int i2c_send(int Dev, unsigned int target, const char *buffer,
             int len);
```

Description

`i2c_send()` is the component used to perform a write operation (a transfer of data from the master to a slave). The I2C controller to use is indicated by the argument `Dev` and the slave address with the argument `target`. The number of bytes to write to the slave is specified by the argument `len` and the argument `buffer` is the base address of the buffer holding the data to send to the slave.

Arguments

`Dev`	Module's device number (Number starting at 0)
`target`	I2C address of the slave to send to
`buffer`	Buffer holding the `len` bytes to send to the slave.
`len`	Number of bytes to write to the slave

Returns

`int`	`== 0 :` the transfer was successful
	`!= 0 :` the transfer failed

Component type

Function

Options

Notes

See Also

`i2c_recv()`
`i2c_send_recv()`

4.1.4 i2c_send_recv

Synopsis

```
#include "???_i2c.h"

int i2c_send_recv(int Dev, unsigned int target, const char *tx_buf,
                  int tx_len, char *rx_buf, int rx_len);
```

Description

`i2c_send_recv()` is the component used to perform a combined write operation (a transfer of data from the master to a slave) followed by a read operation (a transfer from the slave to the master). The I2C controller to use is indicated by the argument `Dev` and the slave address with the argument `target`. The number of bytes to write to the slave is specified by the argument tx_len and the argument tx_buf is the base address of the buffer holding the data to send to the slave. The number of bytes to read from the slave is specified by the argument rx_len and the argument rx_buf is the base address of the buffer used to collect the data read from the slave.

Arguments

`Dev`	Module's device number (Number starting at 0)
`target`	I2C address of the slave to read from
`tx_buf`	Buffer holding the tx_len bytes to send to the slave.
`Tx_len`	Number of bytes to write to the slave
`rx_buf`	Buffer that will hold the data read from the slave. This buffer must be size to at least rx_len bytes
`rx_len`	Number of bytes to read from the slave

Returns

`int`	`== 0 :` the transfer was successful
	`!= 0 :` the transfer failed

Component type

Function

Options

Notes

See Also

`i2c_recv()`
`i2c_send()`

4.1.5 I2CintHndl_*n*

Synopsis

```
#include "???_i2c.h"

void I2CintHndl_n(void);
```

Description

`I2CintHndl_n()` is the interrupt handler for the I2C driver (not used by the standalone version). The ***n*** in the name is a numerical value that specifies the device number the interrupt handler is for.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

The interrupt handler should always be attached to the targeted I2C device interrupt and the number of the interrupt handler MUST match the device number. If there is a mismatch, then the application will most likely crash. If the interrupt handler is not attached and the related interrupt is enabled, the I2C driver for this device will not operate at all (not applicable for the standalone version).

See Also

5 Examples

5.1 Initialization

The first step required when using the I2C driver is to initialize the device controller to be used. The second step required (not applicable with the standalone version) is to attach the interrupt handler for the I2C device and enable these interrupts. The following example shown the initialization of controller #2, set to use 10 bit address at the fast mode (400 kHz):

Table 5-1 Initialization

```
    i2c_init(2, 10, 400000);

    OSisrInstall(I2C_INT2, &I2CintHndl_2);
    GICenable(I2C_INT2, 128, 1);
```

5.2 I2C write

The following example shows how to write to a slave device. The slave device used in this example is Maxim's DS1339 I2C real time clock located at address 0x68 on the I2C bus. The Date/time set is:

9h 10min 11 sec / Jan 2nd 2014 / Thursday (day #5)

The arguments used in `i2c_send()` are:

1st:	`2`	Controller #2
2nd	`0x68`	I2C slave address
3rd	`&Buf[0]`	Base address of the buffer holding the data to send
4th	`8`	8 bytes to write to the slave

Table 5-2 I2C write

```
char Buf[8];

  …

  Buf[0] =  0;              /* Start writing at the first register   */
  Buf[1] = 11;              /* Seconds                               */
  Buf[2] = 10;              /* Minutes                               */
  Buf[3] = 11;              /* Hours                                 */
  Buf[4] =  5;              /* Day of the week                       */
  Buf[5] =  2;              /* Date (day)                            */
  Buf[6] =  1;              /* Date (month)                          */
  Buf[7] = 14;              /* Date (Year – 2000)                    */

  if (0 != i2c_send(2, 0x68, &Buf[0], 8)) {
      puts("Write error");
  }

  …
```

5.3 I2C combined write / read

The following example shows how to perform a combined write and read to/from a slave device. The slave device used in this example is Maxim's DS1339 I2C real time clock located at address 0x68 on the I2C bus. The purpose of using the combined write and read with the DS1339 is to first inform the DS1339 what is going to be the first register read in the read operation. For every byte read, the DS1339 will internally increments the register to read. In the example, the meaning / index of the bytes read from the DS1339 is the same as the previous example.

The arguments used in `i2c_send_recv()` are:

1st:	`2`	Controller #2
2nd	`0x68`	I2C slave address
3rd	`&TxBuf[0]`	Base address of the buffer holding the data to send
4th	`1`	1 byte to write to the slave
5th	`&RxBuf[0]`	Base address of the buffer to capture the data read
6th	`8`	8 byte to read from the slave

Table 5-3 I2C combined write / read

```
char RxBuf[1];
char TxBuf[8];

  …

  TXbuf[0] = 0;

  if (0 != i2c_send_recv(2, 0x68, &TxBuf[0], 1, &RxBuf[0], 8)) {
      puts("Write/Read error");
  }

  …
```

6 References

[R1] Abassi RTOS – User Guide, available at http://www.code-time.com
[R2] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R3] µAbassi RTOS – User Guide, available at http://www.code-time.com