AN2824
Application note

STM32F10xxx devices: advanced I²C examples

Introduction

The aim of this application note is to:

- Provide I²C firmware examples based on polling, interrupts and DMA, covering the four I²C communication modes available in the STM32F10xxx, that is, slave transmitter, slave receiver, master transmitter and master receiver.
- Provide recommendations on the correct use of the I²C peripheral.

This application note applies to STM32F101xx and STM32F103xx medium- and high-density microcontrollers that feature two I²C interfaces (controllers). Throughout this document, these devices are referred to collectively as STM32F10xxx.

The application note is organized in three parts. The first part gives an overview of I²C events and some recommendations on using the I²C efficiently. The second part describes I²C firmware examples using polling, interrupts and DMA. The third part provides an example using DMA and other STM32F10xxx-controlled resources (multislave communication, ADC conversion, temperature sensor).

Contents

List of tables

List of figures

1 I²C events overview

1.1 Slave mode

Slave transmitter

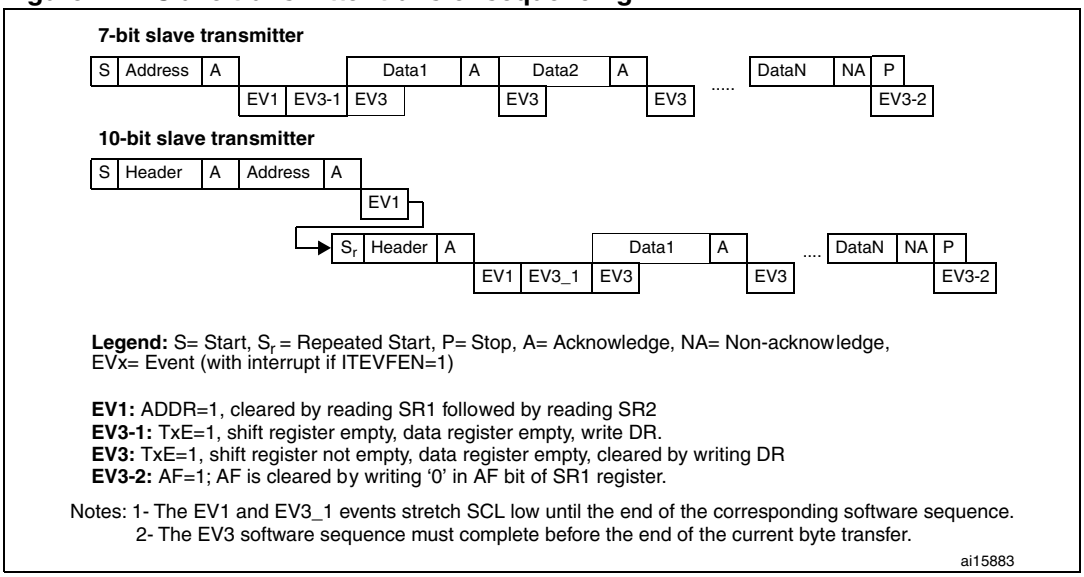
Following the address reception and after clearing ADDR, the slave sends bytes from the DR register to the SDA line via the internal shift register.

The slave stretches SCL low until ADDR is cleared and DR filled with the data to be sent (EV1, EV3 in *Figure 1: Slave transmitter transfer sequencing*).

When the acknowledge pulse is received, the TxE bit is set by hardware with an interrupt if the ITEVFEN and the ITBUFEN bits are set.

If TxE is set and some data were not written in the I2C_DR register before the end of the next data transmission, the BTF bit is set and the interface waits until BTF is cleared by a read to I2C_SR1 followed by a write to the I2C_DR register, stretching SCL low.

Figure 1. Slave transmitter transfer sequencing



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV1: ADDR=1, cleared by reading SR1 followed by reading SR2
EV3-1: TxE=1, shift register empty, data register empty, write DR.
EV3: TxE=1, shift register not empty, data register empty, cleared by writing DR
EV3-2: AF=1; AF is cleared by writing '0' in AF bit of SR1 register.

Notes: 1- The EV1 and EV3_1 events stretch SCL low until the end of the corresponding software sequence.
2- The EV3 software sequence must complete before the end of the current byte transfer.

ai15883

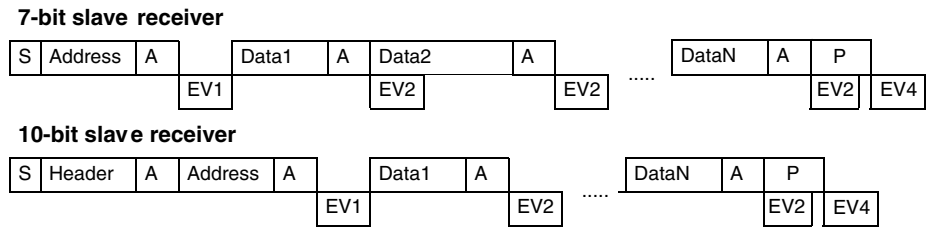
Slave receiver

Following the address reception and after clearing ADDR, the slave receives bytes from the SDA line into the DR register via the internal shift register. After each byte the interface generates in sequence:

- An acknowledge pulse if the ACK bit is set
- The RxNE bit is set by hardware and an interrupt is generated if the ITEVFEN and ITBUFEN bit is set.

If RxNE is set and the data in the DR register is not read before the end of the next data reception, the BTF bit is set and the interface waits until BTF is cleared by a read from I2C_SR1 followed by a read from the I2C_DR register, stretching SCL low (see *Figure 2: Slave receiver transfer sequencing*).

Figure 2. Slave receiver transfer sequencing



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV1: ADDR=1, cleared by reading SR1 followed by reading SR2
EV2: RxNE=1 cleared by reading DR register.
EV4: STOPF=1, cleared by reading SR1 register followed by writing to the CR1 register

Notes: 1- The EV1 event stretches SCL low until the end of the corresponding software sequence.
2- The EV2 software sequence must complete before the end of the current byte transfer.

ai15884

Closing slave communication

After the last data byte is transferred, a Stop Condition is generated by the master. The interface detects this condition, sets the STOPF bit and generates an interrupt if the ITEVFEN bit is set.

Then the interface waits for a read from the SR1 register followed by a write to the CR1 register, stretching SCL low (EV4 in *Figure 2: Slave receiver transfer sequencing*).

1.2 Master mode

Master transmitter

Following the address transmission and after clearing ADDR, the master sends bytes from the DR register to the SDA line via the internal shift register.

The master waits until the first data byte is written into I2C_DR (EV8_1 in *Figure 3: Master transmitter transfer sequencing*).

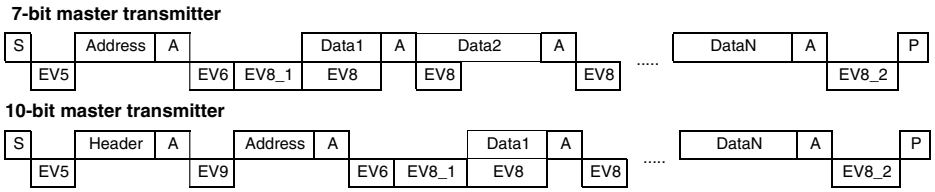
When the acknowledge pulse is received, the TxE bit is set by hardware and an interrupt is generated if the ITEVFEN and ITBUFEN bits are set.

If TxE is set and a data byte was not written in the DR register before the end of the last data transmission, BTF is set and the interface waits until BTF is cleared by a read from I2C_SR1 followed by a write to I2C_DR, stretching SCL low

Closing the communication

After writing the last byte to the DR register, the STOP bit is set by software to generate a Stop condition (EV8_2 in *Figure 3: Master transmitter transfer sequencing*). The interface goes automatically back to slave mode (M/SL bit cleared).

Figure 3. Master transmitter transfer sequencing



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN = 1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register with Address.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.
EV8_1: TxE=1, shift register empty, data register empty, write DR.
EV8: TxE=1, shift register not empty, data register empty, cleared by writing DR register
EV8_2: TxE=1, BTF = 1, Program Stop request. TxE and BTF are cleared by hardware by the Stop condition
EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

Notes: 1- The EV5, EV6, EV9, EV8_1 and EV8_2 events stretch SCL low until the end of the corresponding software sequence.
2- The EV8 software sequence must complete before the end of the current byte transfer.

ai15881

Master receiver

Following the address transmission and after clearing ADDR, the I²C interface enters Master Receiver mode. In this mode the interface receives bytes from the SDA line into the DR register via the internal shift register. After each byte the interface generates in sequence:

- An acknowledge pulse if the ACK bit is set
- The RxNE bit is set and an interrupt is generated if the ITEVFEN and ITBUFEN bits are set (EV7 in *Figure 4: Master receiver transfer sequencing*).

If the RxNE bit is set and the data in the DR register is not read before the end of the last data reception, the BTF bit is set by hardware and the interface waits until BTF is cleared by a read in the SR1 register followed by a read in the DR register, stretching SCL low.

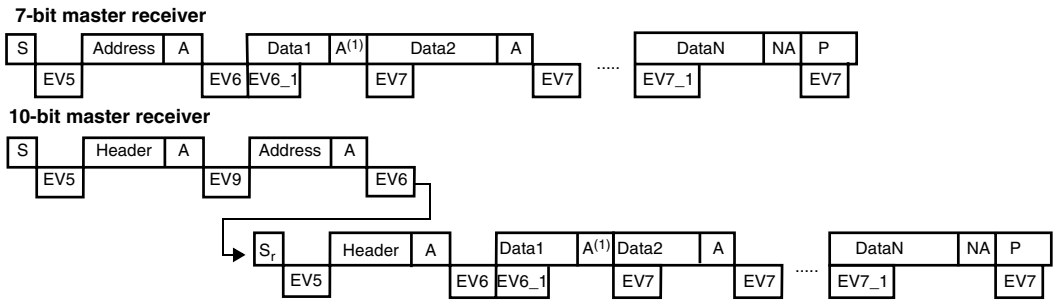
Closing the communication

The master sends a NACK for the last byte received from the slave. After receiving this NACK, the slave releases the control of the SCL and SDA lines. Then the master can send a Stop/Re-Start condition.

- In order to generate the non-acknowledge pulse after the last received data byte, the ACK bit must be cleared just after reading the second last data byte (after second last RxNE event).
- In order to generate the Stop/Re-Start condition, software must set the STOP/START bit just after reading the second last data byte (after the second last RxNE event).
- In case a single byte is to be received, the Acknowledge disable and the Stop condition generation are made just after EV6 (in EV6_1, just after ADDR is cleared).

After the Stop condition generation, the interface goes automatically back to slave mode (M/SL bit cleared).

Figure 4. Master receiver transfer sequencing



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register.

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.

EV6_1: no associated flag event, used for 1 byte reception only. The Acknowledge disable and Stop condition generation are made just after EV6, that is after ADDR is cleared.

EV7: RxNE=1 cleared by reading DR register.

EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request

EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

ai15882

In Master receiver mode, it could happen that the application software does not clear the ACK bit before the last byte acknowledge pulse is received. To handle this case, a procedure is proposed in *Section 1.2.1*.

1.2.1 Closing communications safely when STM32™ is master receiver

In the case where the STM32 is Master Receiver and application software does not guarantee that the acknowledge is disabled before the slave sends the last byte, it is recommended not to read DataN_2, so that after DataN_1, the communication is stretched (both RxNE and BTF are set). Then, clear the ACK bit before reading DataN-2 in DR to ensure it will be cleared before the DataN acknowledge pulse. After that, just after reading DataN_2, set the STOP/ START bit and read DataN_1. After RxNE is set, read DataN.

This is illustrated below:

When 3 bytes remain to be read:

- RxNE = 1 => Nothing (DataN-2 not read)
- DataN-1 received
- BTF = 1 because both Shift and DR registers are full: DataN-2 in DR and DataN-1 in the shift register => SCL tied low: no other data will be received on the bus.
- Clear ACK bit
- Read DataN-2 in DR => This will launch the DataN reception in the shift register
- DataN received (with a NACK)
- Program START/STOP
- Read DataN-1
- RxNE = 1
- Read DataN

This procedure ensures that ACK is cleared before the DataN acknowledge pulse. It is implemented in the software provided with this application note.

The procedure described above is valid for N>2. Cases where a single byte or two bytes are to be received should be handled differently.

Case of a single byte to be received:

- In the ADDR event, clear the ACK bit
- Clear ADDR
- Program the STOP/START bit.
- Read the data after the RxNE flag is set.

Case of two bytes to be received:

With POS= 1, you can program NACK during first byte reception, that is, after clearing ADDR. The NACK should be programmed before the end of first byte reception.

- Set POS and ACK
- Wait for the ADDR flag to be set
- Clear ADDR
- Clear ACK
- Wait for BTF to be set
- Program STOP
- Read DR twice

The procedures described above can be implemented using either polling or interrupts, taking into account that most I^2C firmware events should be managed before the end of transfer of the current byte.

If two or more bytes are to be received by the master, DMA can be used to automatically generate the NACK on the last received byte.

1.3 Some recommendations

- In Slave mode, it is not recommended to use polling because the Slave does not know in advance when it will be addressed by the Master and how many data bytes it will receive or transmit from/to the Master.
- In Slave Transmitter/Receiver mode, when using DMA, if you want to disable the DMA in order to update the DMA counter for example, this should be done between the ADDR event (ADDR is set) and clearing the ADDR flag. This is the only period when the Slave has control of the line (SCL is stretched). Otherwise, there is a risk of stopping DMA while the Master is transmitting or receiving data.
- In Slave Transmitter/Receiver mode, when using DMA, you should make sure that the number of bytes written in the DMA channel counter register is greater than the number of bytes to be transmitted or received by the Master.
- When using DMA, Master reception of a single byte is not supported.
- When using I^2C with interrupts, the I^2C interrupts should have the highest priority in the application in order to make them uninterruptible.

2 I²C firmware configuration for different communication modes (polling, DMA and interrupts)

2.1 Overview

The purpose of this section is to provide an example of I²C firmware using polling, interrupts and DMA.

In this software, I²C communication is set up between two STM32F10xxx devices. The first device operates as a master transmitter/receiver and the second one, as a slave transmitter/receiver.

“STM32F10xxx I2C1” or “I2C1” is used here to refer to the I2C1 interface of the device while “STM32F10xxx I2C2” or “I2C2” refers to the I2C2 interface.

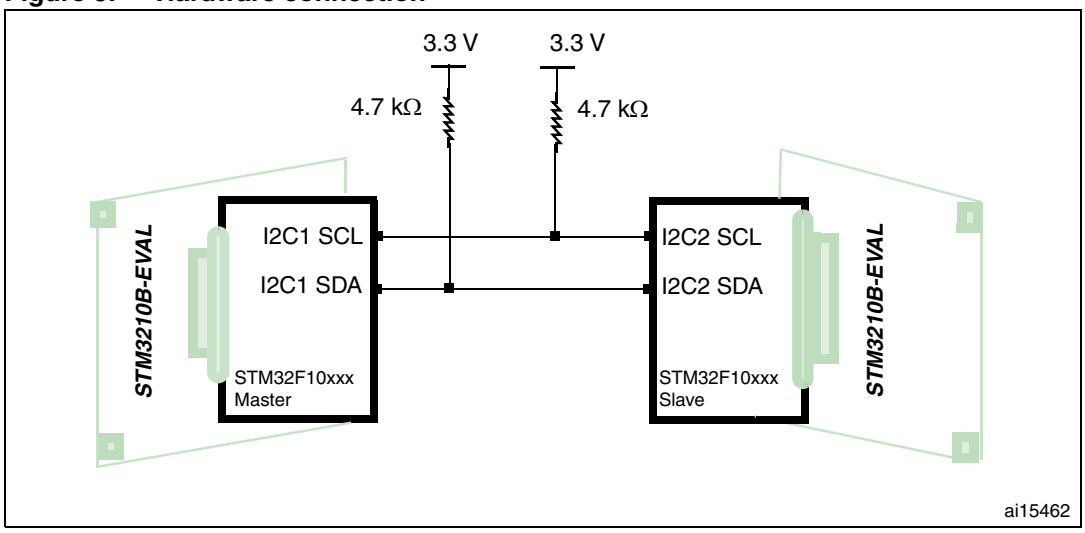
For more details, please refer to the medium- and high-density STM32F101xx and STM32F103xx datasheets and reference manual available from: *http://www.st.com*.

2.2 Hardware environment

Figure 5 shows the hardware connection between the I2C1 of an STM32F10xxx and the I2C2 of another STM32F10xxx. The I2C1 and I2C2 data lines (SDA) are connected together. The I2C1 and I2C2 clock (SCL) lines are also connected together. A pull-up resistor is connected to each line (SDA and SCL).

There are no specific boards (single printed-circuit board) fitted with two STM32F10xxx devices, so we simply connected two STM3210B-EVAL boards together as shown in *Figure 5: Hardware connection*.

Figure 5. Hardware connection



2.3 I²C firmware description

The FirmwareExamples folder is structured as follows:

- src subfolder: contains the source files
  - *driver.c*: file containing the I2C1 master and I2C2 slave read and write routines (using DMA and polling) and DMA1 Channels configured for I2C1 and I2C2 transmission/reception. See list of functions in *Table 1*.
  - *main.c*: file in which the system clocks, I2C1 master, I2C2 slave and interrupts are configured
  - *stm32f10x_it.c*: file in which the I2C1 and I2C2 events (master/slave transmitter/receiver) and error (acknowledge failure, bus error, overrun, arbitration loss) interrupts are handled.
- inc subfolder: contains the header files
  - *driver.h*: Header file for driver.c. It contains also the defined I²C clock, the defined slave address and which mode is used (master transmitter using polling, slave receiver using DMA etc...see *Section 2.4*)
  - *stm32f10x_it.h*: Headers of the interrupt handlers
  - *stm32f10x_conf.h*: configuration file
- EWARMv5, RVMDK and RIDE subfolders: contain tool-dependent preconfigured projects and workspaces.

Table 1. List of functions

Software routine	Purpose
I2C_Master_BufferRead	Master receives a buffer of bytes from the slave using DMA or Polling
I2C_Master_BufferWrite	Master sends a buffer of bytes into the slave suing DMA or polling mode.
I2C_Master_BufferRead1Byte	Master receives 1 byte from the slave using polling.
I2C_Master_BufferRead2Byte	Master receives 2 bytes from the slave using polling.
I2C_Slave_BufferRead	Slave receives a buffer of bytes from the master using DMA.
I2C_Slave_BufferWrite	Slave sends a buffer of bytes to the master using DMA.
DMA_Channel7_Configuration	DMA1 Channel 7 configured for I2C1 master reception.
DMA_Channel6_Configuration	DMA1 Channel 6 configured for I2C1 master transmission.
DMA_Channel5_Configuration	DMA1 Channel 5 configured for I2C2 slave reception.
DMA_Channel4_Configuration	DMA1 Channel 4 configured for I2C2 slave transmission.

2.4 How to use the firmware

In the *driver.h* header file, just uncomment one of the following lines:

`#define DMA_Master_Transmit`: in order to use the STM32 I2C1 as Master Transmitter using DMA.

`#define DMA_Master_Receive`: in order to use the STM32 I2C1 as Master Receiver using DMA.

`#define DMA_Slave_Transmit`: in order to use the STM32 I2C2 as Slave Transmitter using DMA.

`#define DMA_Slave_Receive`: in order to use the STM32 I2C2 as Slave Receiver using DMA.

`#define IT_Master_Transmit`: in order to use the STM32 I2C1 as Master Transmitter using Interrupts.

`#define IT_Master_Receive`: in order to use the STM32 I2C1 as Master Receiver using Interrupts.

`#define IT_Slave_Receive`: in order to use the STM32 I2C2 as Slave Receiver using Interrupts.

`#define IT_Slave_Transmit`: in order to use the STM32 I2C2 as Slave Transmitter using Interrupts.

`#define Polling_Master_Transmit`: in order to use the STM32 I2C1 as Master Transmitter using Polling.

`#define Polling_Master_Receive`: in order to use the STM32 I2C1 as Master Receiver using Polling.

The BufferSize is fixed to 6 and the I²C Clock is fixed to 400 kHz.

The BufferSize is used when operating in Master mode. It is not needed in slave mode because the slave will send or receive the number of bytes fixed by the Master.

In slave mode using DMA, the DMA counter register is initialized to 255. It can be any other value which is greater than the number of bytes to be transmitted/received by the Master.

3 Example using DMA

3.1 Overview

In this example, I^2C communication is performed between three STM32F10xxx microcontrollers. One device is operating as the Master receiver and the other two are operating as Slave transmitters.

As mentioned in *Section 2.1: Overview*, “STM32F10xxx I2C1” or “I2C1” is used to refer to the I2C1 interface of an STM32F10xxx microcontroller with two I^2C controllers and “STM32F10xxx I2C2” or “I2C2” refers to the I2C2 interface of another STM32F10xxx microcontroller also featuring two I^2C controllers.

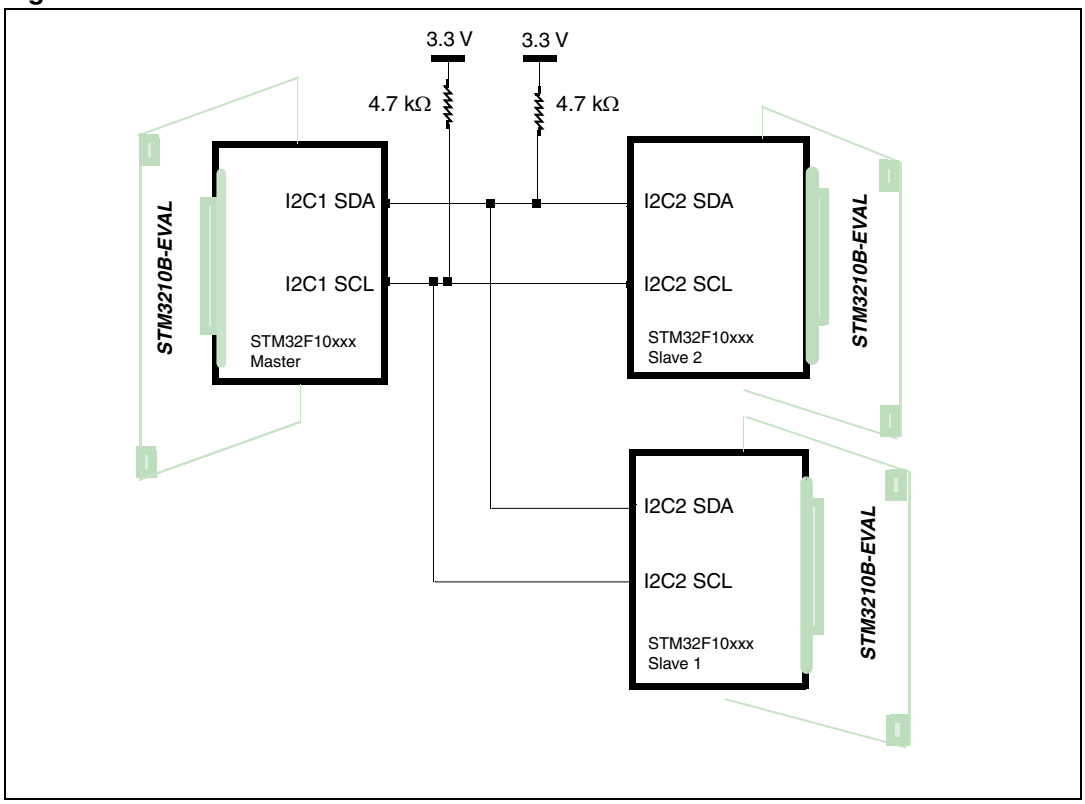
3.2 Hardware environment

Figure 6: Hardware connection shows the connection between an STM32F10xxx I2C1 and two STM32F10xxx I2C2s.

The I2C1 and I2C2 data (SDA) lines are connected together. The I2C1 and I2C2 clock (SCL) lines are also connected together. A pull-up resistor is connected to each line (SDA and SCL).

Like in the case of example 1, there is no a specific board (single PCB) fitted with three STM32F10xxx devices, so we chose to connect an STM3210B-EVAL board (Master) to two other STM3210B-EVAL boards operating as slaves.

Figure 6. Hardware connection



3.3 Example description

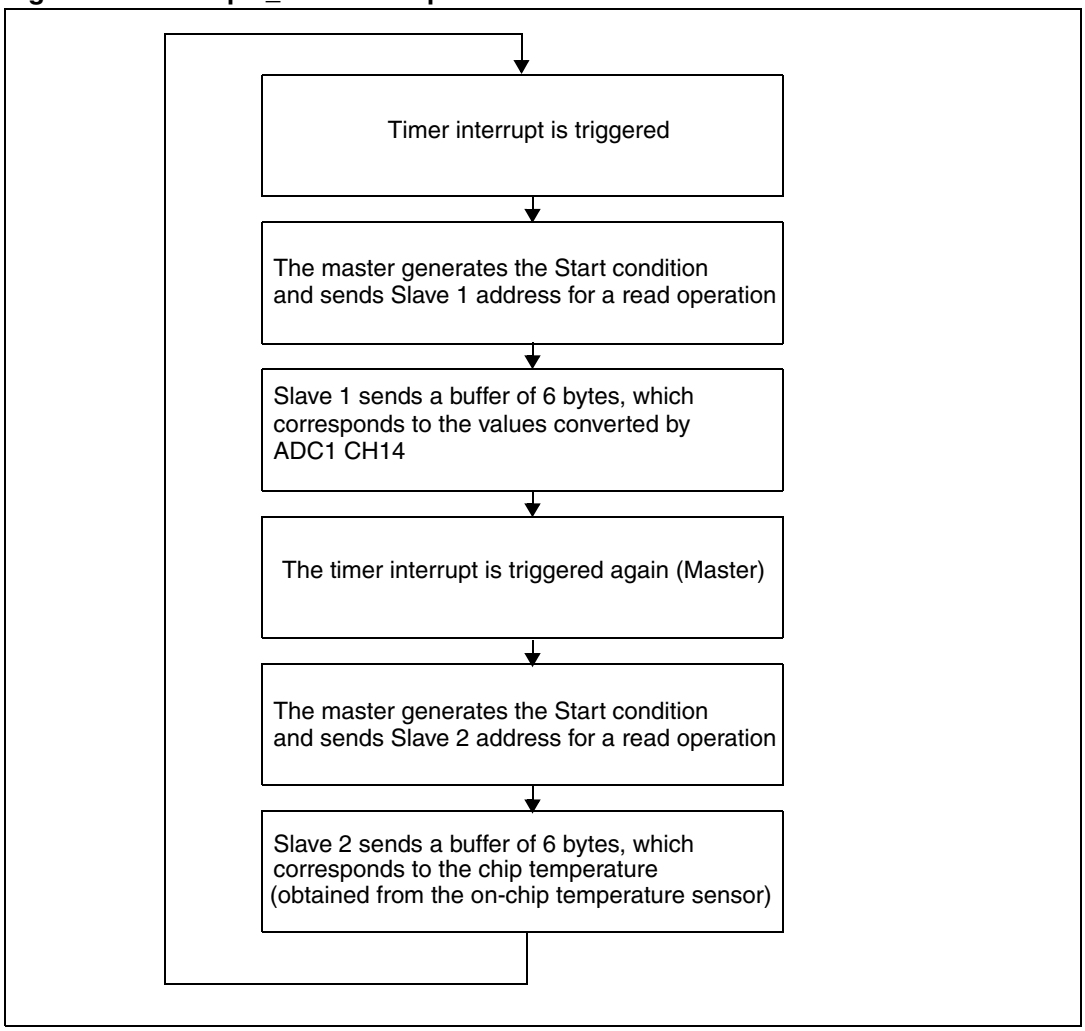
Slave 1 sends the converted values (just the least significant byte) of ADC1 channel 14 (ADC1_CH14) to the Master. ADC1_CH14 is connected to the potentiometer on the board.

Slave 2 sends the temperature values to the Master, using the on-chip temperature sensor that is internally connected to ADC1_CH16.

The Start condition is generated within a timer interrupt triggered once every 3 milliseconds. Consequently, both Slaves send data alternately every 3 milliseconds, approximately. Once the Master has generated the Start condition and sent the Slave address:

- If Slave1 is addressed, ADC1_CH14 conversions are launched. The number of conversions is to be fixed by the user. In the example, 6 conversions are carried out. DMA1 channel 1 is used to transfer the data to the Slave_Buffer_Tx buffer, which is then sent to the Master.
- If Slave2 is addressed, ADC1_CH16 conversions are launched. The number of conversions is to be fixed by the user. In the example, 6 conversions are carried out. DMA1 channel 1 is used to transfer the data to a table named Buffer. Then the Slave_Buffer_Tx buffer is filled with the temperature values using elements in the Buffer table and the formula provided in the device reference manual. Slave_Buffer_Tx buffer is then sent to the Master.

Figure 7. Example_DMA description



3.4 Firmware details

The DMA example is structured as follows:

- src subfolder: contains the source files
  - *driver.c*: file containing the I2C1 master and I2C2 slave read and write routines and DMA1 channels configured for I2C1 and I2C2 transmission/reception.
  - *main.c*: file in which the system clocks, I2C1 master, I2C2 slave and interrupts are configured
  - *stm32f10x_it.c*: interrupt handler for the I2C1 and I2C2 events (master/slave transmitter/receiver) and errors (acknowledge failure, bus error, overrun, arbitration loss).
- inc subfolder: contains the header files
  - *driver.h*: header file for driver.c. It also contains the definitions for the I^2C clock, the slave address and the mode to be used (master transmitter using polling, slave receiver using DMA etc...see *Section 2.4*)
  - *stm32f10x_it.h*: headers of the interrupt handlers
  - *stm32f10x_conf.h*: configuration file
- EWARMv5, RVMDK and RIDE subfolders: contain tool-dependent preconfigured projects and workspaces.

3.5 How to use the example

In order to compile the project for slave1, just:

- Uncomment `#define slave1` in main.h header file.
- Uncomment `#define DMA_Slave_Transmit` in *driver.h* header file.
- Define the slave address as 0x28 in *driver.h* header file.

In order to compile the project for slave2, just:

- Uncomment `#define slave2` in *main.h* header file.
- Uncomment #define `DMA_Slave_Transmit` in *driver.h* header file.
- Define the slave address as 0x30 in *driver.h* header file.

In order to compile the project for master, just:

- Uncomment `#define master` in *main.h* header file.
- Uncomment `#define DMA_Master_Receive` in *driver.h* header file.

To run the example successfully, perform the following steps:

- Load the code compiled for slave1 into board 1
- Load the code compiled for slave2 into board 2
- Load the code compiled for master into board 3
- Run the code in board 1
- Run the code in board 2
- Run the code in board 3

The user can then use an oscilloscope to display the I^2C data transmitted alternately by slave1 and slave2 to the master.

4 Revision history

Table 2. Document revision history

Date	Revision	Changes
18-Sep-2008	1	Initial release.
04-Mar-2009	2	Added *Section 1.3* recommendations for I^2C use. Content added to *Section 2: I2C firmware configuration for different communication modes (polling, DMA and interrupts)*. Added *Section 3* example using DMA.
03-Nov-2009	3	This application note applies to the whole STM32F10xxx family. Subfolder descriptions modified in *Section 2.3: I2C firmware description* and *Section 3.4: Firmware details*.