AN2824
Application note

STM32F10xxx I²C optimized examples

Introduction

The aim of this application note is to provide I²C firmware optimized 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 and to provide recommendations on the correct use of the I²C peripheral.

This application note applies to STM32F101xx and STM32F103xx medium, high and XL density microcontrollers, STM32F105/107xx connectivity line and STM32F100xx value line devices. Throughout this document, these devices are referred to collectively as STM32F10xxx.

The application note is organized in three parts. The first part describes the I2C master programming examples using Polling, DMA and Interrupts. The second part describes the I2C slave programming examples using DMA and Interrupts. The third part is an overview of the content of the firmware accompanying this application note. .

Contents

List of tables

List of figures

1 I2C master programming examples (DMA, interrupts, polling)

1.1 Overview

The purpose of this section is to describe the firmware examples of I2C master transmitting and receiving data using polling, DMA and interrupts, provided with this application note.

Flowcharts of Master Transmitter/Receiver in all modes (DMA, Polling, Interrupts) are also provided.

You can modify these examples to adapt them to your application requirements.

1.2 Description of the examples

1.2.1 Polling

Master receiver

The master sends the START condition on the bus by setting START bit. The interface waits for the SB flag to be set and then cleared by writing the slave address in the DR register. The interface waits for the ADDR flag to be set then cleared by reading the SR1 and SR2 status register. After that, the master waits for the RXNE flag to be set in order to read data from the data register (EV7).

The EV7 software sequence must complete before the end of the current byte transfer. In case EV7 software sequence can not be managed before the current byte end of transfer, it is recommended to use BTF instead.

In order to close the communication, the software must guarantee the ACK bit is cleared in time in order to receive the last byte with a NACK. For this purpose, method 2 described in the device reference manuals is used: with this method, DataN_2 is not read, 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 is cleared before the DataN Acknowledge pulse.
- After this, 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 data 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 starts DataN reception in the shift register.
- DataN received (with a NACK)
- Program START/STOP
- Read DataN-1

- RxNE = 1
- Read DataN

Note: Due to the "Wrong data read into data register" limitation described in the device errata sheet, interrupts should be masked between STOP programming and DataN-1 reading. Please refer to the device errata sheet for more details.

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

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.

Note: The EV6_3 software sequence must complete before the current byte end of transfer. To ensure this, the interrupts should be masked between ADDR clearing and STOP/START programming.

Case of two bytes to be received:

- 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 DataN-1
- Read DataN

Note: 1 Due to the "Wrong data read into data register" limitation described in the device errata sheet, interrupts must be masked between STOP programming and DataN-1 reading. Please refer to the device errata sheet for more details.

2 The EV6_1 software sequence must complete before the ACK pulse of the current byte transfer. To ensure this, interrupts must be disabled between ADDR clearing and ACK clearing.

Figure 1. Flowchart of master receiving more than 2 bytes using polling

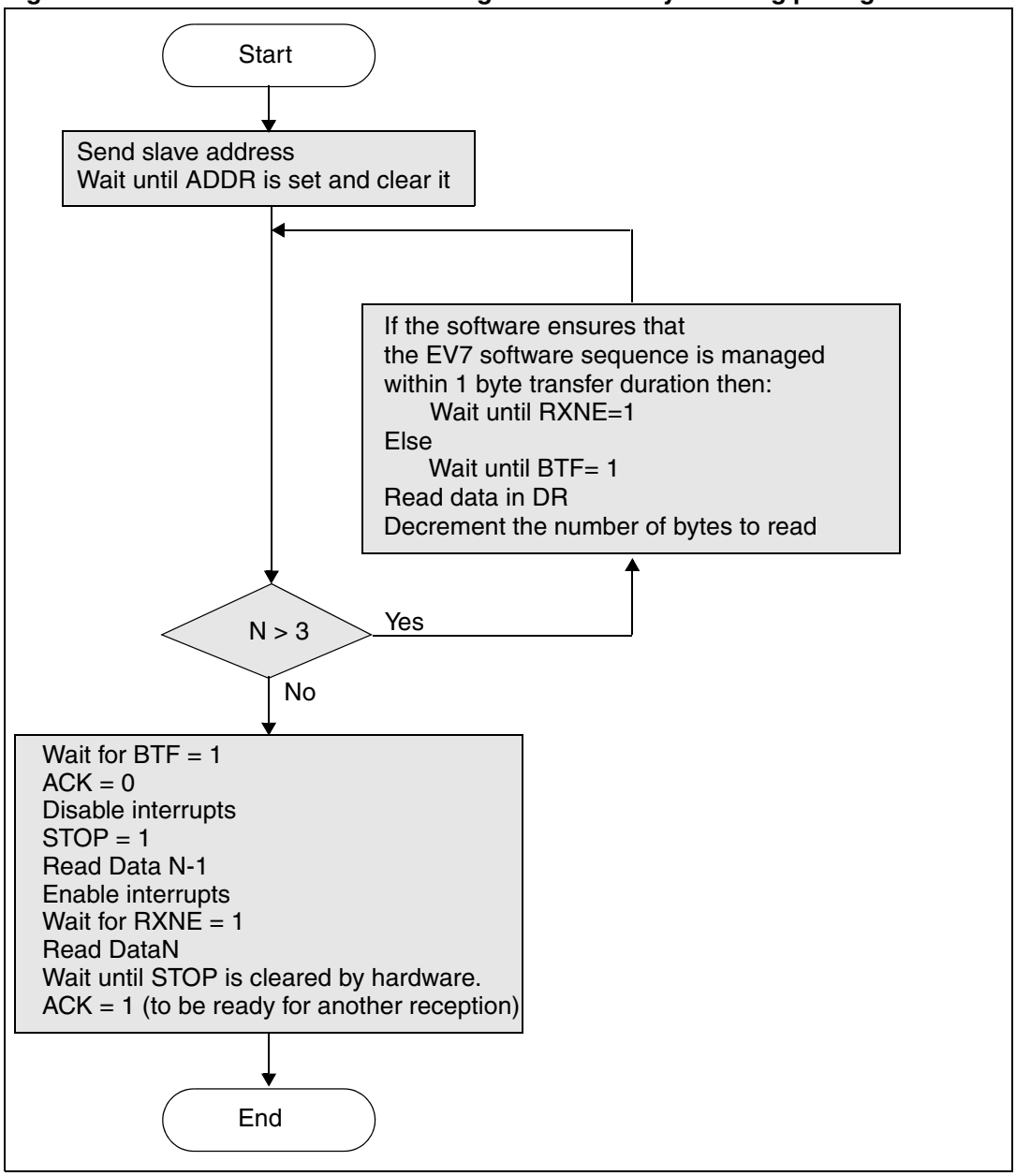
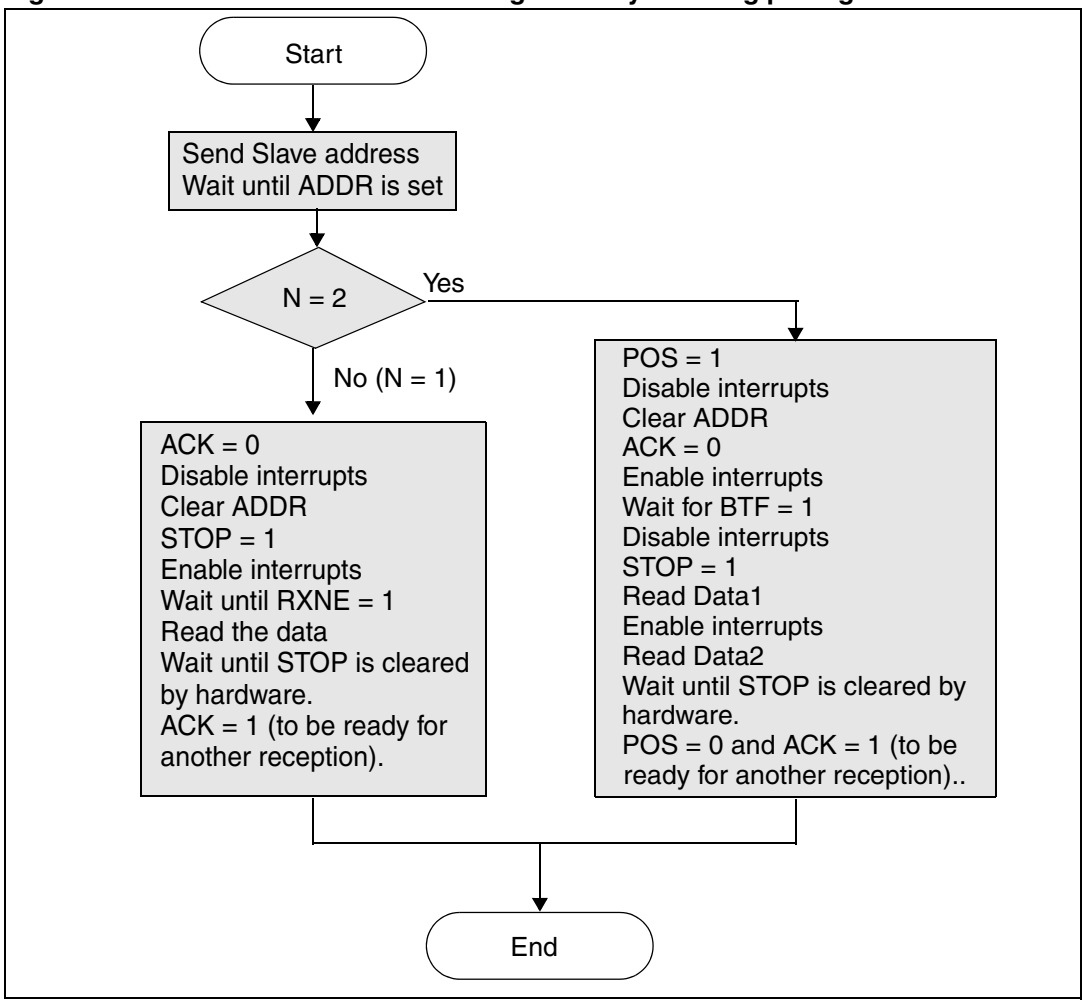


Figure 2. Flowchart of master receiving 1 or 2 bytes using polling



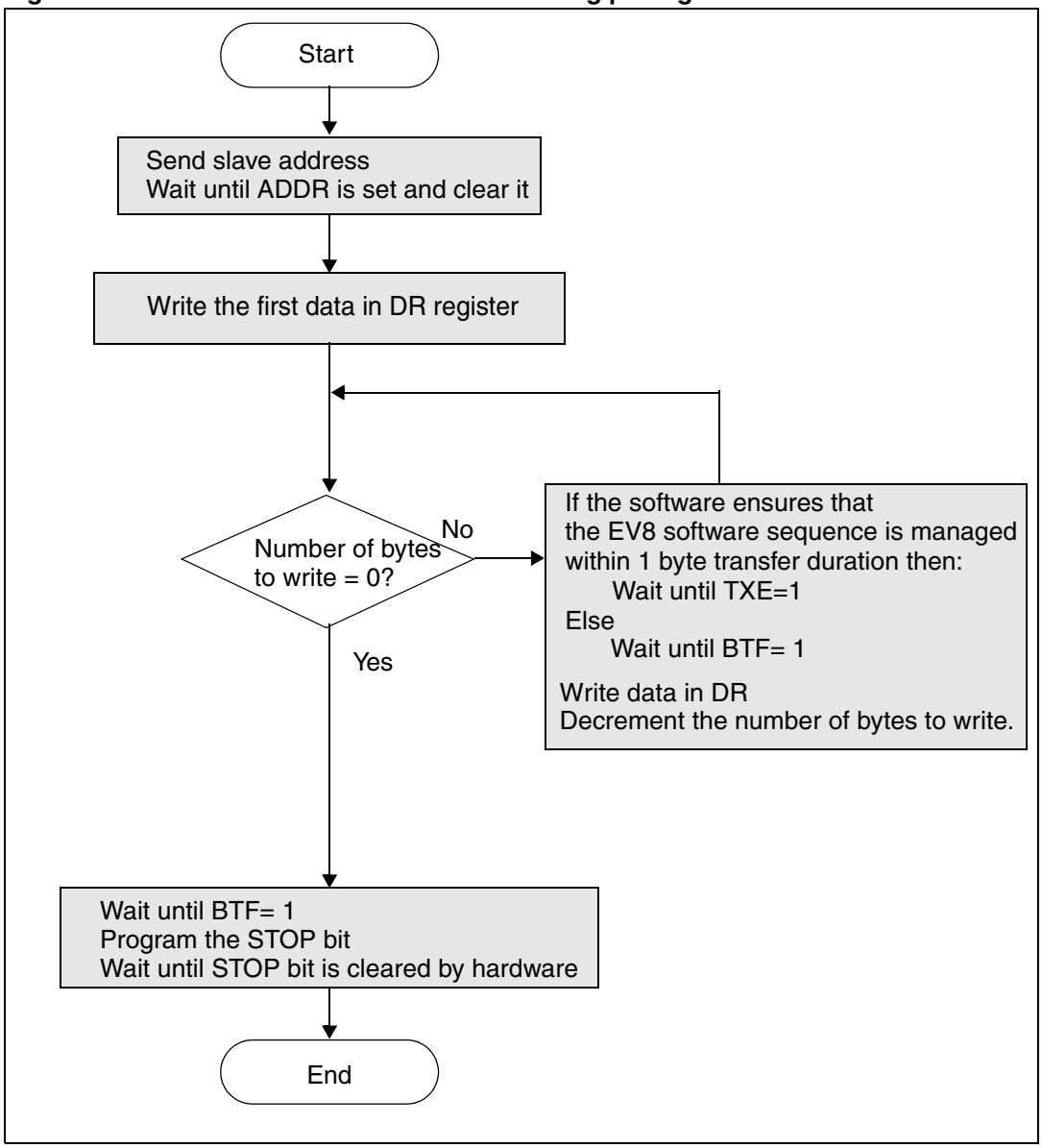
Master transmitter

The master sends the START condition on the bus by setting the START bit. The interface waits for the SB flag to be set and then cleared by writing the slave address in the DR register. The interface waits for the ADDR flag to be set then cleared by reading the SR1 and SR2 status register. After that, the master writes the first data byte in the data register (EV8_1). It then continues by writing the next data bytes in the data register after every TXE (EV8).

The EV8 software sequence must complete before the end of the current byte transfer. In case EV8 software sequence cannot be managed before the current byte end of transfer, it is recommended to use BTF instead.

After the last byte is written to the DR register, the application software must wait until BTF is set (EV8_2: Both DR and shift register are set) before setting the STOP bit to generate a STOP condition.

Figure 3. Flowchart of master transmitter using polling



1.2.2 DMA

Master receiver

DMA requests are generated only for data transfer. In reception, DMA requests are generated by the Data Register becoming full (RXNE = 1).

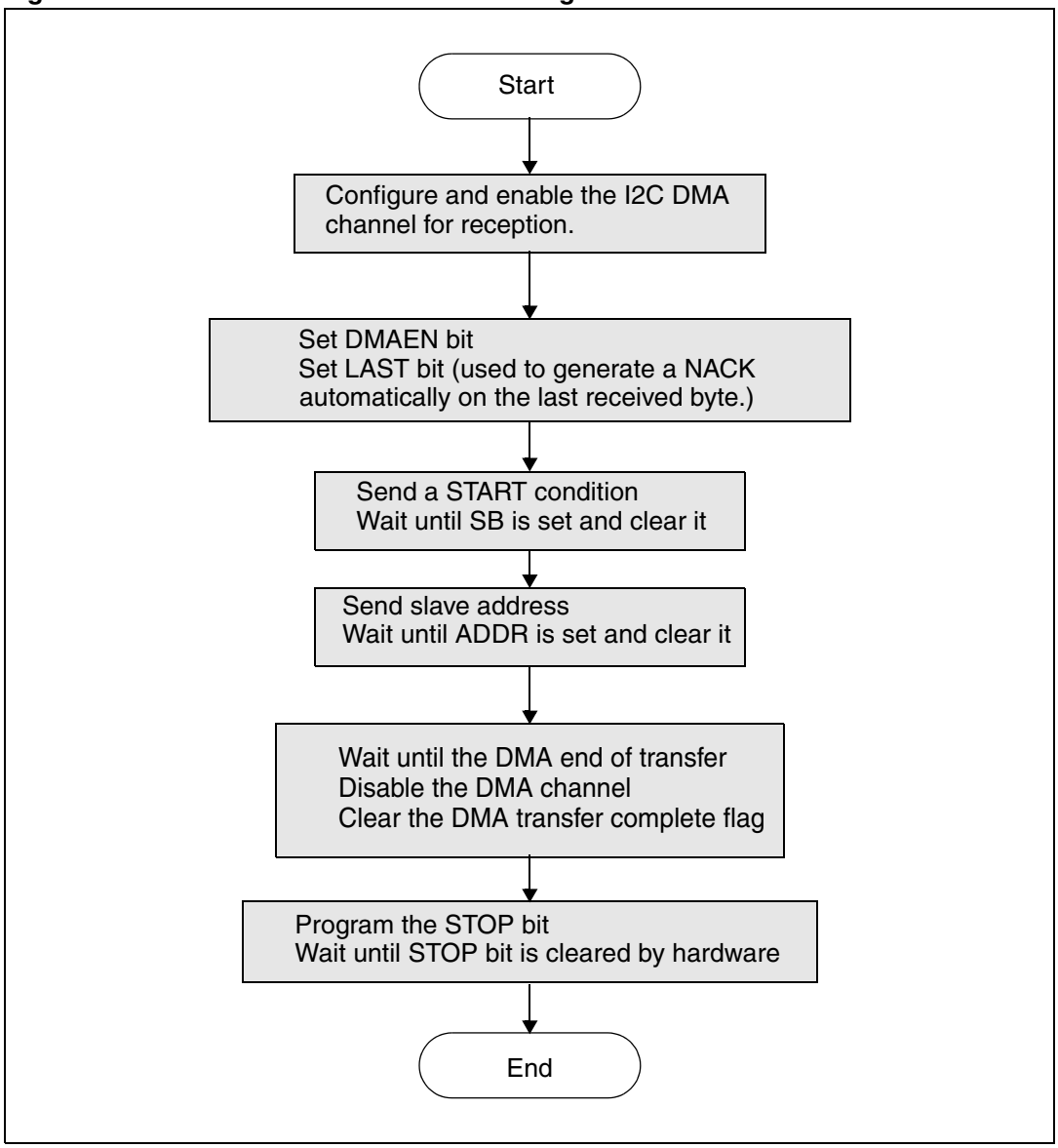
All remaining events (SB, ADDR etc...) must be managed by polling or interrupts. In the examples accompanying the application note, they are managed by polling.

The master sends the START condition on the bus by setting START bit. The interface waits for SB flag to be set and then cleared by writing the slave address in DR register. The interface waits for ADDR flag to be set then cleared by reading SR1 and SR2 status registers. At that point, DMA transfers begin.

After the DMA end of transfer, the STOP bit is set in order to generate a STOP condition.

Note: When using DMA, master reception of a single byte is not supported.

Figure 4. Flowchart of master receiver using DMA



Master transmitter

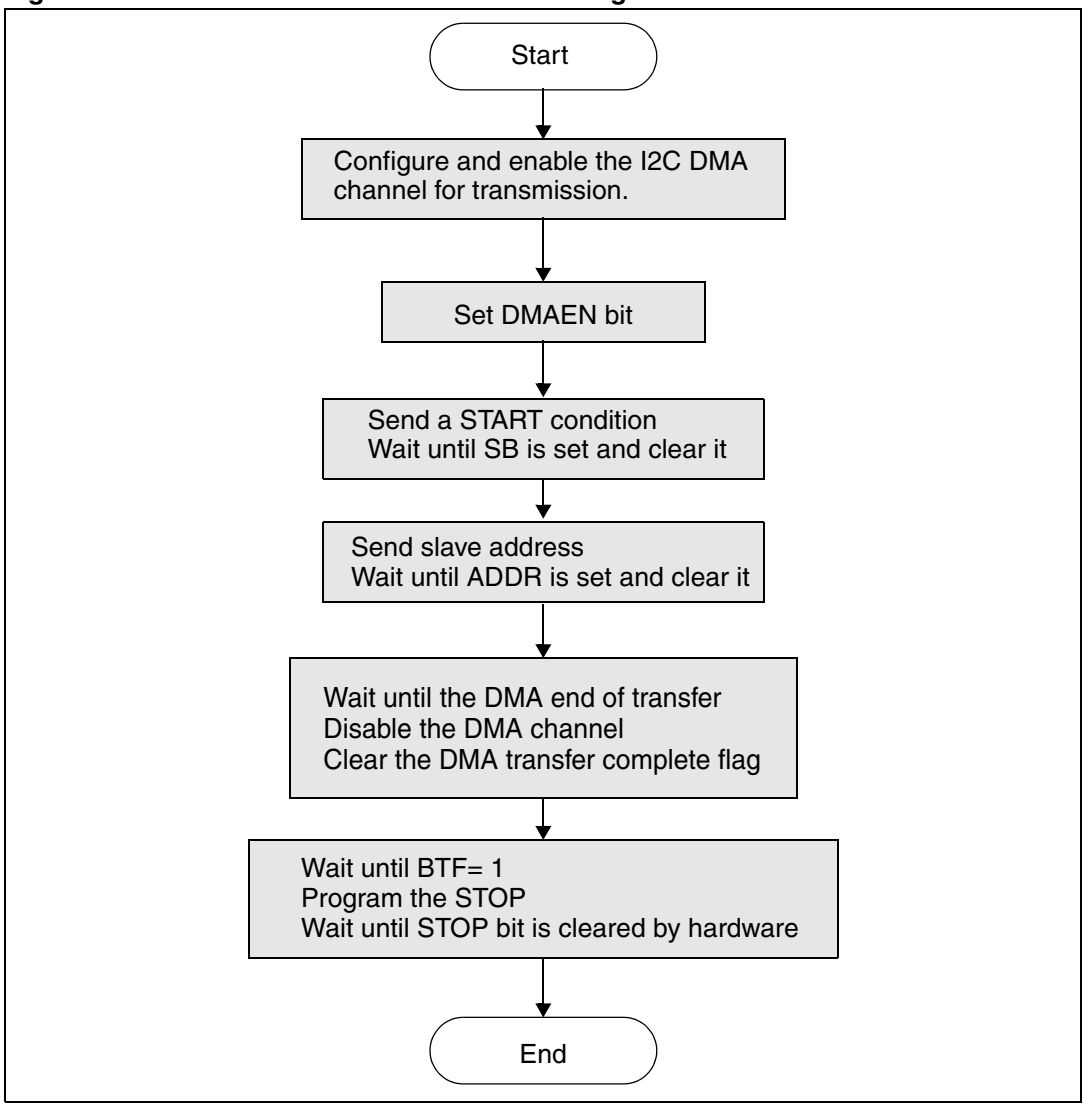
DMA requests are generated only for data transfer. In transmission, DMA requests are generated by the Data Register becoming empty (TXE = 1).

All remaining events (SB, ADDR, etc.) must be managed by polling or interrupts. In the examples accompanying this application note, they are managed by polling.

The master sends the START condition on the bus by setting START bit. The interface waits for the SB flag to be set and then cleared by writing the slave address in DR register. The interface waits for ADDR flag to be set then cleared by reading SR1 and SR2 status register. At that point, DMA transfers begin.

After the DMA end of transfer, the application software must wait until BTF is set (EV8_2: Both DR and shift register are set) before setting the STOP bit to generate a STOP condition.

Figure 5. Flowchart of master transmitter using DMA



Note: In I2C master mode using polling or DMA, I2C errors (BERR, OVR, AF, ARLO) interrupts are enabled and configured. When an error occurs the corresponding error flag is cleared by software.

1.2.3 Interrupt mode

The basic principles of transmission and reception are the same in interrupt mode as in polling. The difference here is that the I2C EVT and BUF interrupts are enabled in addition to the I2C error flags. All I2C events are managed in the I2C event interrupt routine.

The I2C interrupts should have the highest priority in the application in order to make them uninterruptible.

2 I2C slave programming examples (DMA, interrupt)

2.1 Overview

The purpose of this section is to describe the firmware examples provided with this application note, showing an I2C slave transmitting and receiving data using DMA and interrupts.

Flowcharts of Slave Transmitter/Receiver using DMA and Interrupts, are also provided.

Note: In slave mode, it is not recommended to use polling mode, especially for handling the ADDR event because the slave doesn't know in advance when it will be addressed by the Master. Consequently, the application, which normally does other tasks and not just I2C communication, may get stuck waiting for the ADDR flag to be set. The same is true for the other events, for example the slave could also get stuck waiting for the RXNE or TXE flag to be set.

That is why it is recommended to always manage the ADDR and STOPF events using interrupts and to manage data transfers using interrupts or DMA.

2.2 Description of the examples

2.2.1 Interrupt

All I2C slave transmitter/receiver events are managed by the I2C event interrupt routine.

I2C interrupts should have the highest priority in the application in order to make them uninterruptible.

- **Slave receiver**

Following the address reception and after clearing ADDR, the slave receives bytes from Master device every RXNE event interrupt. When the master generates a STOP condition on the bus, the slave detects this STOP condition when the STOPF flag is set in the SR2 status register.

- **Slave Transmitter**

Following the address reception and after clearing ADDR, the slave transmits bytes to the Master device after every TXE event interrupt. The slave detects the end of transmission when it receives a Non Acknowledge pulse telling it that it must stop transmission (AF flag is set in the I2C error interrupt routine).

For transmission/reception using interrupts, the transmit and receiver counter are initialized every ADDR event.

2.2.2 DMA

- **Slave receiver**

DMA requests are generated only for data transfers. In reception, DMA requests are generated by the Data Register becoming full (RXNE = 1).

Remaining events (ADDR and STOPF) are managed by interrupts.

- **Slave Transmitter**

DMA requests are generated only for data transfers. In transmission, DMA requests are generated by the Data Register becoming empty (TXE = 1).

All remaining events (ADDR and AF) are managed by interrupts.

Provided that the slave doesn't know in advance how many data bytes are to be received/transmitted to/from the master device, the DMA channel transmit/receive end of transfer cannot be detected. So, it's not possible to know when to update the DMA channel counter and memory base address to prepare the next transmission or reception. In order to update the DMA channel counter, the DMA channel must be disabled and of course the DMA channel must not be disabled while the master device is transmitting/receiving data.

The only period during which the slave has control of the line (master can not transmit neither receive) is the period between ADDR event (ADDR flag is set) and clearing the ADDR flag. For this purpose, in the provided slave examples using DMA, the DMA count and memory base address are initialized after ADDR flag is set and before ADDR flag is cleared.

3 Firmware overview

The Optimized I2C Examples folder is structured as follows:

- src subfolder: contains the source files.
  - I2CRoutines.c: containing the I2Cx (I2C1 or I2C2) master and slave initialization, write and read routines (using DMA, interrupts or polling) and DMA1 channels configured for I2Cx (I2C1 or I2C2) transmission/reception. See *Table 1: List of functions*. In the I2C write and read routines, the register are accessed directly. The I2C standard library is used only for the I2C peripheral initialization.
  - main.c: where the I2Cx interrupts in the NVIC are configured and the routines in the I2CRoutines.c are called.
  - stm32f10x_it.c: where the I2Cx (I2C1 and I2C2) event and error interrupts are handled.
- inc subfolder: contains the header files.
  - stm32f10x_it.h: headers of the interrupt handlers
  - stm32f10x_conf.h: configuration file
  - I2CRoutines.h: header file for I2CRoutines.c. It also contains the I2C bit/flag definitions and other definitions that are needed.
- EWARMv5, MDK-ARM and RIDE subfolders: contain tool-dependent preconfigured projects and workspaces.

Table 1. List of functions

Software routine	Purpose
I2C_Master_BufferRead	Reads a buffer of bytes from the slave
I2C_Master_BufferWrite	Sends a buffer of bytes to the slave
I2C_Slave_BufferReadWrite (see notes below)	Enables DMA and event interrupts for data transmission and reception
I2C_LowLevel_Init	Initializes the I2C, GPIOs and DMA channels
I2C_DMAConfig	Updates the buffer size and the buffer base address

Note: 1 In an application where the STM32 is slave transmitter/receiver, the I2C _Slave_BufferReadWrite( ) routine has to be called only once in order to enable the DMA requests or I2C interrupts.

2 If the slave transmits or receives using DMA, uncomment the "#define SLAVE_DMA_USE" line in the stm32f10x_it.c source file.

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: I^2C 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: I^2C firmware description* and *Section 3.4: Firmware details*.
22-June-2010	4	Application note document and firmware completely rewritten.