... the world's most energy friendly microcontrollers

I2C Master and Slave Operation

AN0011 - Application Note

Introduction

The EFM32 I2C module allows simple, robust and cost effective communication between integrated circuits using only one data and one clock line.

This application note demonstrates how to use the EFM32 I2C module to talk to an I2C temperature sensor. It also includes a software example where two EFM32s are connected; each EFM32 will operate in either slave or master mode and talk to the other one.

This application note includes:

- This PDF document
- Source files (zip)
  - Example C-code
  - Multiple IDE projects

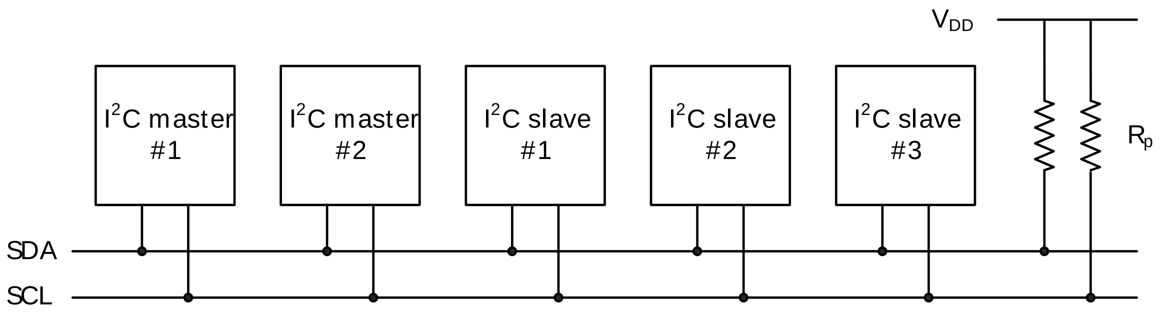
1 I2C Theory

1.1 General

The I2C allows connection of up to 128 individually addressable devices using only two bi-directional lines: clock (SCL) and data (SDA). The only additional hardware required is a pull-up resistor for each of the lines. Each of the connected devices can be either a master or slave device. Only master devices are allowed to drive the clock line.

The I2C protocol and the EFM32 I2C module feature several mechanisms for handling bus conflicts and contention. A possible I2C connection scheme is illustrated in Figure 1.1 (p. 2) .

Figure 1.1. I2C Bus



At the physical layer both SCL and SCA lines are in open-drain, hence the pull-up resistors. Increasing the number of devices on the I2C bus will also increase the line capacitance and thus reduce the slew-rate. The slew-rate can be controlled by changing the drive strength in the GPIO module for the I2C pins. The size of the pull-up resistors can be calculated as a function of the maximum rise time allowed for the given bus speed and the estimated bus capacitance Cb as shown in Equation 1.1 (p. 2)

Pull-up Resistor Equation

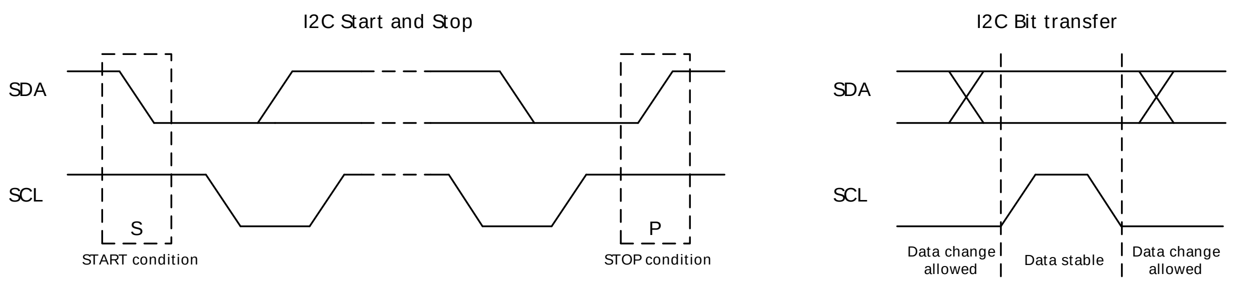
$$Rp(max) = tr/0.8473 \times Cb \tag{1.1}$$

The maximal rise times for 100 kHz, 400 kHz and 1 MHz I2C are 1 µs, 300 ns and 120 ns respectively.

1.2 I2C Signals/Frames

START and STOP conditions are used to initiate and stop transactions on the I2C-bus. All transactions on the bus begin with a START condition (S) and end with a STOP condition (P). As illustrated in Figure 1.2 (p. 3), a START condition is generated by pulling the SDA line low while SCL is high, and a STOP condition is generated by pulling the SDA line high while SCL is high.
Also illustrated in Figure 1.2 (p. 3) is I2C bit transfer. Note that data must be stable for the whole duration of the SCL high period.

Figure 1.2. Start, Stop and Data



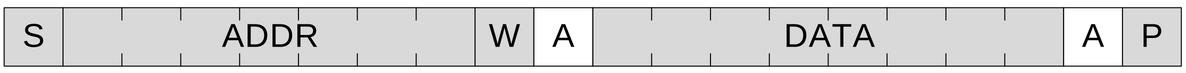
A master initiates a transfer by sending a START followed by the address of the slave it wishes to contact and a single R/W bit telling whether it wishes to read from (R/W = 1) or write to the slave (R/W = 0).

After the 7-bit address and the R/W bit, the master releases the bus, allowing the slave to acknowledge the request. During the next bit-period, the slave pulls SDA low (ACK) if it acknowledges the request, or keeps it high if it does not acknowledge it (NACK). Following the address acknowledge, either the slave or master transmits data, depending on the value of the R/W bit. After every 8-bit byte transmitted on the SDA line, the transmitter releases the line to allow the receiver to transmit an ACK or a NACK. Both the data and the address are transmitted with the most significant bit first.

The master ends the transmission after a (N)ACK by sending a STOP condition on the bus. After a STOP condition, any master can initiate a new transfer.

An example of a master writing to a slave is shown in Figure 1.3 (p. 3). The identifiers used are: S - Start bit, ADDR - Address, W - Write(0), A - ACK, N - NACK, DATA - Data, P - Stop bit.

Figure 1.3. Master Write to Slave



For further details about I2C and the EFM32 I2C module, please see the EFM32 reference guide.

1.3 Arbitration

As the I2C bus is a multi-master bus it is possible that two devices initiate a transfer at the exact same time (e.g. RTC tick). When this happens the first device attempting to transmit a logical 1 while another device transmits a logical 0 will lose arbitration. The device attempting to transmit 1 will detect that the line is low when it should actually be high so assumes that another master is active and immediately stops its transfer. This device will then wait until the next STOP condition before trying to transmit again.

1.4 Clock stretching

An addressed slave device may hold the clock line (SCL) low after receiving (or sending) a byte, indicating that it is not yet ready to process more data. The master communicating with the slave will try to raise the clock to transfer the next bit but will verify that the clock line will remain low. The master will then have to wait for the slave to release the line so that the clock signal can be transmitted. If a master wants to slow down the rate of data transfer it just delays the next clock edge.

2 Software Examples

2.1 Master Operation with Interrupts

This software example uses the efm32_i2c library to read a DS75 digital thermometer mounted on the EFM32_G890_DK development kit. The tempsens driver includes an interrupt routine to demonstrate how to use the efm32_i2c library in interrupt driven mode.

2.1.1 Program Flow

The EFM32 reads the temperature sensor every 2 seconds and stays in a low power mode between measurements. The measurement results are converted to either Celsius or Fahrenheit and displayed on the segment LCD.

The efm32_i2c library can be used both in polled and interrupt driven mode. This example uses the interrupt driven approach, but still has a while loop which blocks the function reading the temperature until the transfer is finished. The I2C interrupt routine is entered every time the I2C module generates an interrupt, then it is up to the state machine in the efm32_i2c library to handle and clear the active interrupt flags.

Interrupts are generated every time the I2C-peripheral is finished with an autonomous task, or when it detects fault conditions which should be handled by software. Not all possible interrupts are handled or used by this code example.

2.1.2 Connections

The software example is specifically made for the development kit and includes the board support package to enable the correct connections and initialize the temperature sensor from software.

2.2 Master and Slave Operation

This software example makes the EFM32 operate in both master and slave configuration. Two EFM32 I2C modules are connected and set up to both transmit (master mode) and receive data between each other (slave mode) using a common I2C bus.

The code example included is written for the EFM32_Gxxx_DK development kit, but is easily ported to any EFM32 part with an I2C-peripheral.

2.2.1 Program Flow

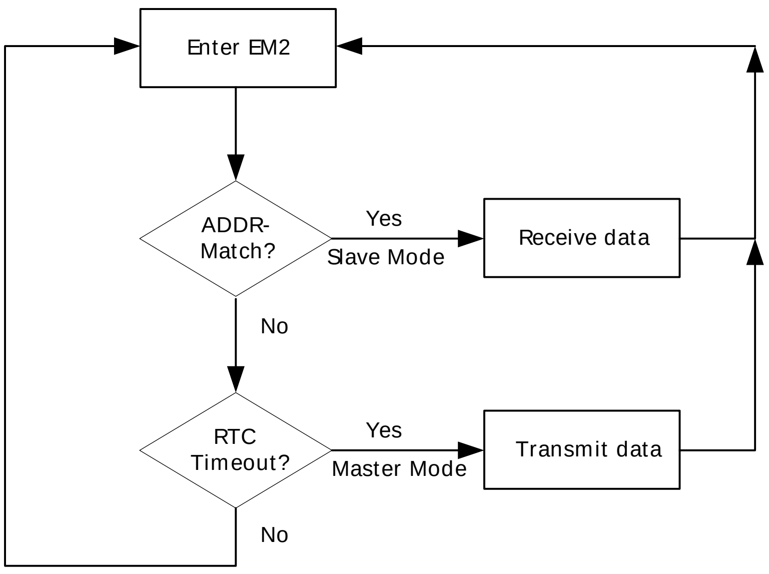
Each EFM32 has enabled I2C address match interrupt and Real Time Counter (RTC) interrupt. Both these interrupts are capable of waking up the EFM32 from Deep Sleep mode (EM2), hence the idle state current consumption is extremely low.

The I2C is continuously monitoring the I2C line. If a start condition followed by the I2Cs defined address, an address match interrupt is issued and data is received until a stop condition is detected.

The Real Time Counter (RTC) is set to wake up the EFM32 regularly. If a reception is not in progress at the time of wake-up, a master transmission is initiated. The address match interrupt is disabled during the transmission and re-enabled afterwards.

The operation is illustrated in the state diagram shown in Figure 2.1 (p. 5)

Figure 2.1. Program state machine



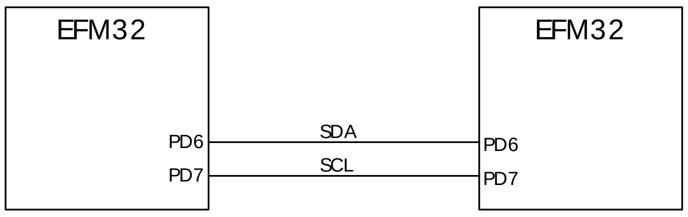
During transmission, the PC0 pin is set in order to determine transfer direction for example by using a logic analyzer. On the starter kit, this pin is connected to LED0, which is lit during transmission.

Please refer to the source code for further details.

2.2.2 Connections

The software example uses location 1 for the SCL (PD7) and SDA (PD6) pins. Connect the PD6 pins and the PD7 pins of the EFM32s, respectively. The configuration is illustrated in Figure 2.2 (p. 5)

Figure 2.2. Connection



In the GPIO module of the EFM32 the pull-up resistors have been enabled, hence external pull-up resistors are not necessary to make the example work. However, external resistors are generally preferable as they keep the lines defined at all times. For example, when the EFM32 is in reset-state, the pull-up configuration in the GPIO module is not available, which leaves the I2C bus undefined.

3 Revision History

3.1 Revision 1.07

2014-05-07

Updated example code to CMSIS 3.20.5

Changed to Silicon Labs license on code examples

Added example projects for Simplicity IDE

Removed example makefiles for Sourcery CodeBench Lite

3.2 Revision 1.06

2013-10-14

New cover layout

3.3 Revision 1.05

2013-05-08

Added software projects for ARM-GCC and Atollic TrueStudio.

3.4 Revision 1.04

2012-11-12

Adapted software projects to new kit-driver and bsp structure.

3.5 Revision 1.03

2012-04-20

Adapted software projects to new peripheral library naming and CMSIS_V3.

3.6 Revision 1.02

2012-03-22

Added description on slew rate control.

Added arbitration and clock stretching sections (1.3 and 1.4).

Added a software example with interrupt driven master operation.

Changed name of application note.

3.7 Revision 1.01

16-11-2010

Removed clearing of RXDATAV interrupt flag through I2C_IFC since this operation is ignored.

Changed example folder structure, removed build and src folders.

Added chip-init function.

3.8 Revision 1.00

20-09-2010.

Initial revision.

A Disclaimer and Trademarks

A.1 Disclaimer

A.2 Trademark Information

B Contact Information

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701

Please visit the Silicon Labs Technical Support web page:
http://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.

Table of Contents

List of Figures

List of Equations

silabs.com

ZERO
ARM Cortex-M0+

TINY
ARM Cortex-M3

GECKO
ARM Cortex-M3

LEOPARD
ARM Cortex-M3

GIANT
ARM Cortex-M3

WONDER
ARM Cortex-M4