**8bit & 32bit**

**ABOV I2C API**

**User Guide**

Application Note

Version 1.00

Contents

[1 Overview 3](#_Toc34054472)

[2 Function block diagram 4](#_Toc34054473)

[2.1 I2C Function Description 4](#_Toc34054474)

[2.2 I2C Master Mode 5](#_Toc34054475)

[2.3 I2C Slave Mode 6](#_Toc34054476)

[3 How to use in main application 7](#_Toc34054477)

[3.1 Public Macro 7](#_Toc34054478)

[3.2 I2C Initialization 7](#_Toc34054479)

[3.3 I2C Master – Write / Read 8](#_Toc34054480)

[3.4 I2C Slave – Set Transmit Buffer 8](#_Toc34054481)

[3.5 Example of use in main application 9](#_Toc34054482)

[4 How to port to other device 10](#_Toc34054483)

[4.1 Main (main.c) 10](#_Toc34054484)

[4.2 Header file (DeviceName\_I2C\_API\_Vx.xx.h) 10](#_Toc34054485)

[4.3 Source file (DeviceName\_I2C\_API\_Vx.xx.c) 11](#_Toc34054486)

[5 APPENDIX 18](#_Toc34054487)

[5.1 Example code for I2C Master Function with I2C BUS EEPROM 18](#_Toc34054488)

# Overview

This document describes the firmware for the ABOV\_I2C\_Driver.

It has the goal of helping the developer to understand how this firmware works, and how to tailor the application.

# Function block diagram

## I2C Function Description

|  |
| --- |
|  |

1. START and STOP conditions are always generated by the master.  
 The bus is busy between START and STOP condition.

2. Every byte put on the SDAn line must be 8-bits long and it has to be followed by an ACK bit.  
 The transmitter releases the SDAn line(High) during the acknowledge clock pulse.  
 So the receiver is able to pull down the SDAn line during the acknowledge clock pulse.

3. The I2C is byte-oriented and interrupt-based. So byte complete, Interrupt is generated.  
 And clock line held low while interrupts are served by slave.  
 Data transfer continues when the slave is ready for another byte of data and releases clock line.

## I2C Master Mode

|  |
| --- |
| main.c    I2C IRQ |

## I2C Slave Mode

|  |
| --- |
| main.c    I2C IRQ |

# How to use in main application

## Public Macro

The macros on the left side are user specific.

Modification is required for the user application.

|  |  |
| --- | --- |
| #define I2C\_DEVICE\_ADDRESS 0xA0  #define I2C\_SLAVE\_OWN\_ADDR 0xA0  #define I2C\_SPEED 10000  #define I2C\_MAX\_BUFFER\_SIZE 20  #define I2C\_MAX\_CHANNEL 2  #define I2C\_CH0 0  #define I2C\_CH1 1 | #define I2C\_ACK\_DISABLE 0  #define I2C\_ACK\_ENABLE 1  #define I2C\_WRITE\_MODE 0  #define I2C\_READ\_MODE 1  #define I2C\_IDLE 0  #define I2C\_BUSY 1  #define I2C\_FALSE 0  #define I2C\_TRUE 1 |

## I2C Initialization

‘USI\_I2C\_Initial()’ function is required to use I2C functionality.

This function includes the following.

- Peri. Clock Enable / I2C Block Reset / I2C Interrupt Enable

- I2C Enable / I2C Ack Enable / Set Slave Own Address / Set I2C Speed / Set I2C Data Hold Time

|  |
| --- |
| [example of use]  USI\_I2C\_Initial(I2C\_CH0, I2C\_SPEED, I2C\_SLAVE\_OWN\_ADDR, I2C\_ACK\_ENABLE); |

## I2C Master – Write / Read

Using ‘USI\_I2C\_MasterTransferData()’ function, you can use both I2C Write and I2C Read functions.

This function includes the following parameter.

- I2C channel number / slave device address

- write data buffer / write data length / read data buffer / read data length

|  |
| --- |
| [example of use]  (case1) I2C Write 4-byte  🡪 USI\_I2C\_MasterTransferData(I2C\_CH0, I2C\_DEVICE\_ADDRESS, \*u8TxDat, 4, \*u8RxDat, 0);  (case2) I2C Read 4-byte  🡪 USI\_I2C\_MasterTransferData(I2C\_CH0, I2C\_DEVICE\_ADDRESS, \*u8TxDat, 0, \*u8RxDat, 4);  (case3) I2C Write 1-byte , Read 3-byte  🡪 USI\_I2C\_MasterTransferData(I2C\_CH0, I2C\_DEVICE\_ADDRESS, \*u8TxDat, 1, \*u8RxDat, 3);  (case4) I2C Write 3-byte , Read 1-byte  🡪 USI\_I2C\_MasterTransferData(I2C\_CH0, I2C\_DEVICE\_ADDRESS, \*u8TxDat, 3, \*u8RxDat, 1); |

## I2C Slave – Set Transmit Buffer

Using ‘USI\_I2C\_SetSlaveData()’ function, you can set transmit buffer data.

The buffer data can be modified by the user.

|  |
| --- |
| [example of use]  USI\_I2C\_SetSlaveData(I2C\_CH0); |

## Example of use in main application

This is example of use in main application.

|  |
| --- |
|  |

# How to port to other device

## Main (main.c)

1) Include header file 🡪 #include "DeviceName\_I2C\_API\_Vx.xx.h"

2) Call ‘USI\_I2C\_InterruptHandler()’ function at the I2C IRQ.

🡪 Example) void I2C0\_Handler(void){ USI\_I2C\_InterruptHandler(I2C\_CH0); }  
 void I2C1\_Handler(void){ USI\_I2C\_InterruptHandler(I2C\_CH1); }

3) Use functions as below in user application.

🡪 Example) USI\_I2C\_Initial(I2C\_CH0, I2C\_SPEED, I2C\_SLAVE\_OWN\_ADDR, I2C\_ACK\_ENABLE);  
 USI\_I2C\_MasterTransferData(I2C\_CH0, I2C\_DEVICE\_ADDRESS, \*u8TxDat, 4, \*u8RxDat, 0);  
 USI\_I2C\_SetSlaveData(I2C\_CH1);

## Header file (DeviceName\_I2C\_API\_Vx.xx.h)

It needs to change public typedef as below.

I2C Control Register and I2C Status Register may differ from product to product,

So the corresponding bits must be checked.

|  |
| --- |
|  |

## Source file (DeviceName\_I2C\_API\_Vx.xx.c)

8-bit MCU and 32-bit MCU comparison

1) void USI\_I2C\_Initial(uint8\_t ch, uint32\_t speed, uint8\_t addr, uint8\_t ack)

It needs to change name of the register below to match the device.

>> I2CSAR(I2C Slave Address), I2CSCLHR(SCL High Period Register), I2CSCLLR(SCL Low Period Register)  
 I2CSDAHR(SDA Hold Time Register)  
 >> For 32-bit MCU, It needs to add peripheral clock enable register.

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

2) void USI\_I2C\_SoftwareReset(uint8\_t ch, uint8\_t enable)

It needs to change name of the register below to match the device.

>> For 8-bit MCU, I2CMR(I2C Mode Control Register) has reset function bit.  
 >> For 32-bit MCU, It needs to add SCU register configure.

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

3) void USI\_I2C\_Enable(uint8\_t ch, uint8\_t enable)

It needs to change name of the register below to match the device.

>> I2CMR(I2C Mode Control Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

4) void USI\_I2C\_ConfigureInterrupt(uint8\_t ch, uint8\_t enable)

It needs to change name of the register below to match the device.

>> For 8-bit MCU, It can configure interrupt by I2CMR and IE1.  
 >> For 32-bit MCU, It can configure interrupt by NVIC.

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

5) void USI\_I2C\_ConfigureAcknowledge(uint8\_t ch, uint8\_t enable)

It needs to change name of the register below to match the device.

>> I2CMR(I2C Mode Control Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

6) void USI\_I2C\_Start(uint8\_t ch, uint8\_t enable)

It needs to change name of the register below to match the device.

>> I2CMR(I2C Mode Control Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

7) void USI\_I2C\_Stop(uint8\_t ch, uint8\_t enable)

It needs to change name of the register below to match the device.

>> I2CMR(I2C Mode Control Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

8) void USI\_I2C\_Send7bitAddress(uint8\_t ch, uint8\_t dev\_addr, uint8\_t direction)

It needs to change name of the register below to match the device.

>> I2CDR(I2C Data Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

9) void USI\_I2C\_MasterTransferData(uint8\_t ch, uint8\_t dev\_addr, uint8\_t \*write\_data, uint8\_t write\_len, uint8\_t \*read\_data, uint8\_t read\_len)

It needs to change name of the register below to match the device.

>> I2CSR(I2C Status Register)

|  |
| --- |
| \* 8-bit MCU |
| \* 32-bit MCU |

10) void USI\_I2C\_InterruptHandler(uint8\_t ch)

It needs to change name of the register below to match the device.

>> I2CSR(I2C Status Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

11) static void USI\_I2C\_MasterProcess(uint8\_t ch)

It needs to change name of the register below to match the device.

>> I2CSR(I2C Status Register), I2CDR(I2C Data Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

12) void USI\_I2C\_SetSlaveData(uint8\_t ch)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

13) static void USI\_I2C\_SlaveProcess(uint8\_t ch)

It needs to change name of the register below to match the device.

>> I2CSR(I2C Status Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

14) uint8\_t USI\_I2C\_GetInterruptStatus(uint8\_t ch)

It needs to change name of the register below to match the device.

>> I2CSR(I2C Status Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

15) void USI\_I2C\_ConfigureGerneralCall(uint8\_t ch, uint8\_t enable)

It needs to change name of the register below to match the device.

>> I2CSAR(I2C Slave Address Register)

|  |  |
| --- | --- |
| \* 8-bit MCU | \* 32-bit MCU |

# APPENDIX

## Example code for I2C Master Function with I2C BUS EEPROM

1) Devices

- Example 1) Master : MC97FGx16 (ABOV 8-bit MCU) / Slave : BR24G01-3 I2C BUS EEPROM (ROHM)

- Example 2) Master : A31G12x (ABOV 32-bit MCU) / Slave : BR24G01-3 I2C BUS EEPROM (ROHM)

2) Function Description

- Write Cycle

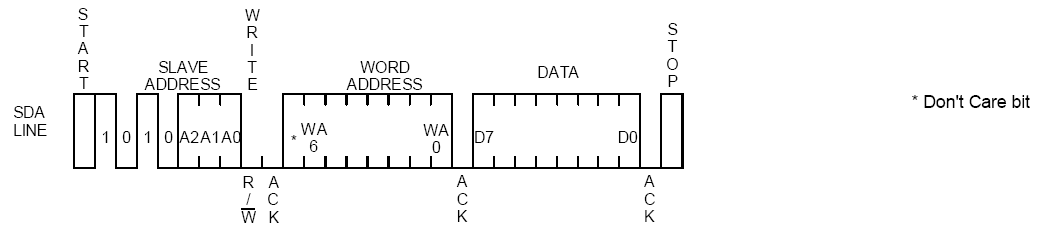
: Arbitrary data can be written to this EEPROM. When writing only 1 byte,

Byte Write is normally used, and when writing continuous data of 2 bytes or more,

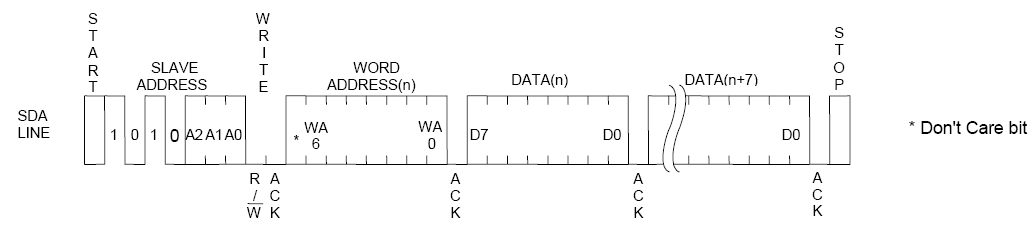
simultaneous write is possible by Page Write cycle. The maximum number of bytes is specified

per device of each capacity. Up to 8 arbitrary bytes can be written.

\* Byte Write



\* Page Write



- Read Cycle

: Read cycle is when data of EEPROM is read. Read cycle could be random read cycle

or current read cycle. Random read cycle is a command to read data by designating a specific

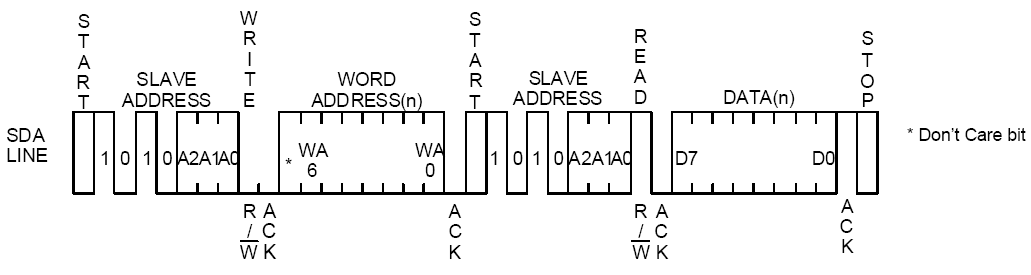
address, and is used generally. Current read cycle is a command to read data of internal address

register without designating an address, and is used when to verify just after write cycle.

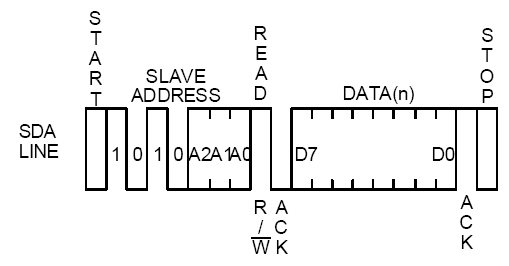
In both the read cycles, sequential read cycle is available where the next address data can be

read in succession.

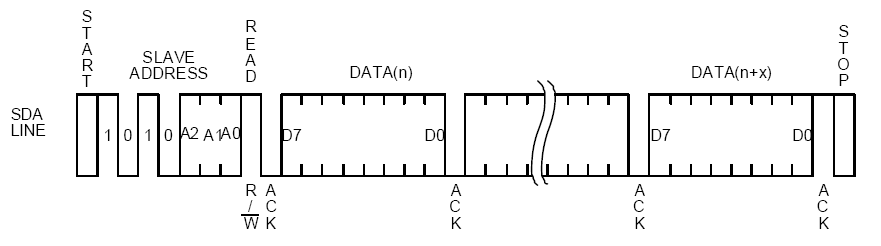
\* Random Read



\* Current Read



\* Sequential Read



**ABOV Disclaimer**

**IMPORTANT NOTICE – PLEASE READ CAREFULLY**

ABOV Semiconductor ("ABOV") reserves the right to make changes, corrections, enhancements, modifications, and improvements to ABOV products and/or to this document at any time without notice. ABOV does not give warranties as to the accuracy or completeness of the information included herein. Purchasers should obtain the latest relevant information of ABOV products before placing orders. Purchasers are entirely responsible for the choice, selection, and use of ABOV products and ABOV assumes no liability for application assistance or the design of purchasers’ products. No license, express or implied, to any intellectual property rights is granted by ABOV herein. ABOV disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of ABOV products in such unauthorized applications. ABOV and the ABOV logo are trademarks of ABOV. All other product or service names are the property of their respective owners. Information in this document supersedes and replaces the information previously supplied in any former versions of this document.

**© 2020 ABOV Semiconductor – All rights reserved**