AN204776

F²MC-16FX Family, I2C

This application note describes how to communicate via I2C with a Serial EEPROM.

Contents

1 Introduction....1
2 24C04....2
2.1 EEPROM....2
2.2 Connection to MB963xx....3
2.3 Addressing....3
2.4 Example Code....3
3 Additional Information....8
4 Document History....9
Worldwide Sales and Design Support....10
Products....10
PSoC® Solutions....10
Cypress Developer Community....10
Technical Support....10

1 Introduction

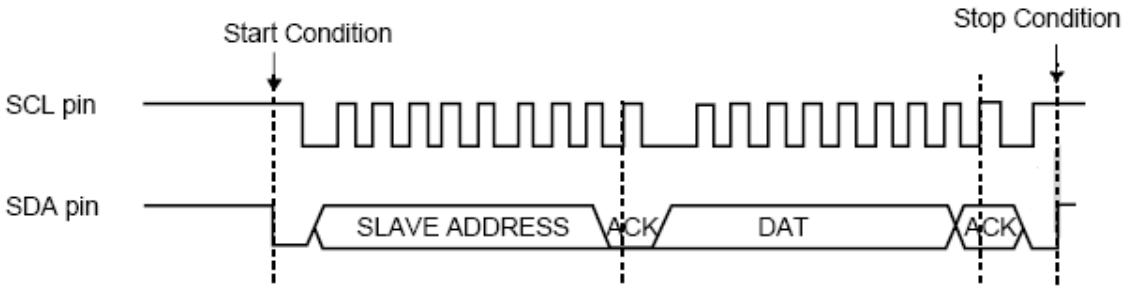
I2C is a two-wire serial bus. There is no need for chip select or arbitration logic, making it cheap and simple to implement in hardware. The two I2C signals are serial data (SDA) and serial clock (SCL) and are both bi-directional. These are open drain output hence one need to connect pull up register on the SDA and SCL line. For slow communication speeds, the internal pull-up resistors may be used.

The device that initiates a transaction on the I2C bus is termed the master. The master normally controls the clock signal. A device being addressed by the master is called a slave.

Each I2C-compatible hardware slave device comes with a predefined device address, the lower bits of which may be configurable at the board level. The master transmits the device address of the intended slave at the beginning of every transaction. Each slave is responsible for monitoring the bus and responding only to its own address.

Data transfer is initiated with the START bit (S) when SDA is pulled low while SCL stays high. Then, SDA sets the transferred bit while SCL is low and the data is sampled (received) when SCL rises. When the transfer is complete, a STOP bit (P) is sent by releasing the data line to allow it to be pulled up while SCL is constantly high.

Figure 1. Timing Diagram



The master begins the communication by issuing the start condition (S). The master continues by sending a unique 7-bit slave device address, with the most significant bit (MSB) first. The eighth bit after the start, read/not-write (0/1), specifies whether the slave is now to receive (0) or to transmit (1). The receiver, acknowledging receipt of the previous byte, issues an ACK bit. Then the transmitter (slave or master, as indicated by the bit) transmits a byte of data starting with the MSB. At the end of the byte, the receiver (whether master or slave) issues a new ACK bit. This 9-bit pattern is repeated if more bytes need to be transmitted.

In a write transaction (slave receiving), when the master is done transmitting all of the data bytes it wants to send, it monitors the last ACK and then issues the stop condition (P). In a read transaction (slave transmitting), the master does not acknowledge the final byte it receives. This tells the slave that its transmission is done. The master then issues the stop condition.

This application note describes how to communicate via I2C with a Serial EEPROM. In this note a 24C04 EEPROM from Turbo IC is used.

Please note, that this document only gives a rough overview about the communication. The described source codes were written for understanding not for code size or speed. Neither interrupts nor timers were used. Time critical program codes are always performed by simple flag polling or wait loops.

2 24C04

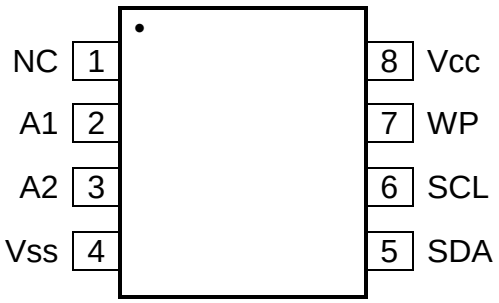
This Chapter Describes How to Communicate With the 24C04 EEPROM

2.1 EEPROM

The 24LC04 serial EEPROM from Turbo IC has 4096-Bit memory size, organized as 512 x 8 Bits.

The 24LC04 has the following pin-out:

Figure 2. Pin Diagram of EEPROM 24LC04



Pin Names:

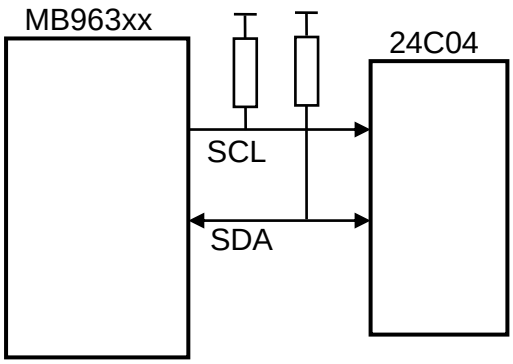
Table 1. Pin Names of EEPROM 24LC04

NC	Not connected
A1	Address select 1
A2	Address select 2
Vss	Ground
SDA	Serial Data
SCL	Serial Clock
WP	Write Protect
Vcc	Power Supply (+ 5 volts)

2.2 Connection to MB963xx

The EEPROM can be connected as in the following schematic. Please note, that no power supply pins and other MCU-Pins are drawn than those for the connection to the EEPROM.

Figure 3. Connection Block Diagram



The internal pull-up resistors of the MCU may be used, if the 50 kΩ fits to the communication speed. Also, the Port Input Enable Register (PIER) must be switched on.

2.3 Addressing

The EEPROM uses the 8-bit addressing scheme of the I2C bus. The 4 most significant bits are fixed to “1010”. The A1 and A2 lines of the chip select the next two address bits. The next bit is the most significant bit of the memory address. The last address bit select Read or Write operation, as defined by the I2C standard.

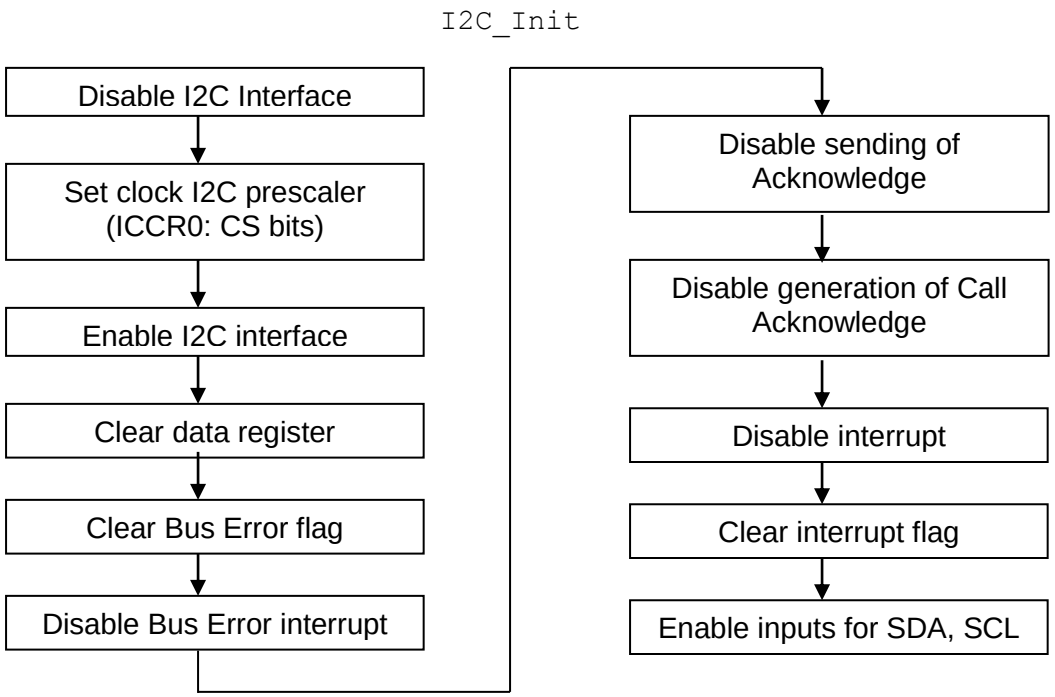
The actual address that is written to or read from is transmitted as the first data byte. It can only be set by a write operation, i.e. if a certain memory address needs to be read, a write access for that address is required before reading it.

2.4 Example Code

The following code shows how to establish a communication to and from the EEPROM using the MB96F348.

2.4.1 Initial Functions and Declarations

Flowchart



C Code

```
void I2C_Init(void)
{
  PIER04_IE4 = 1;// SDA0 pin to input (MB96340 Series)
  PIER04_IE5 = 1;// SCL0 pin to input (MB96340 Series)

  ICCR0_EN  = 0;// stop I2C interface
  ICCR0_CS4 = 1;// CS4..0 : set prescaler
  ICCR0_CS3 = 1;
  ICCR0_CS2 = 1;
  ICCR0_CS1 = 1;
  ICCR0_CS0 = 1;
  ICCR0_EN  = 1;// enable I2C interface

  IDAR0 = 0;// clear data register

  IBCR0_BER  = 0;// clear bus error interrupt flag
  IBCR0_BEIE = 0;// bus error interrupt disabled
  IBCR0_ACK  = 0;// no acknowledge generated
  IBCR0_GCAA = 0;// no call acknowledge is generated
  IBCR0_INTE = 0;// disable interrupt
  IBCR0_INT  = 0;// clear transfer interrupt request flag
}
```

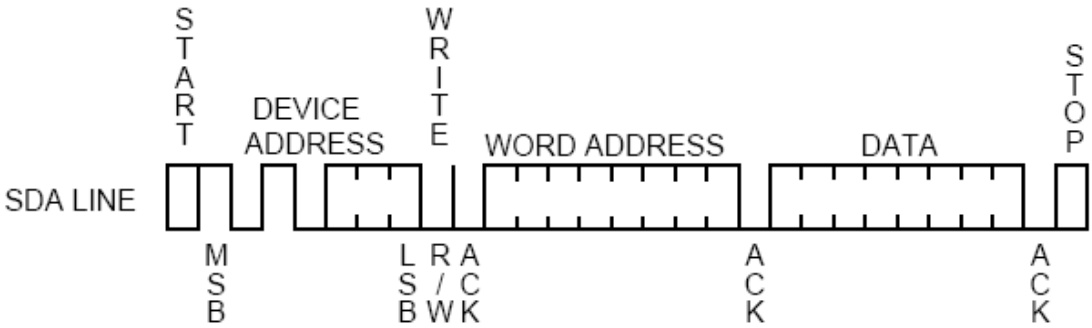
Note: The Port Input Enable Register (PIER) of the I2C pins must be enabled.

2.4.2 Write EEPROM

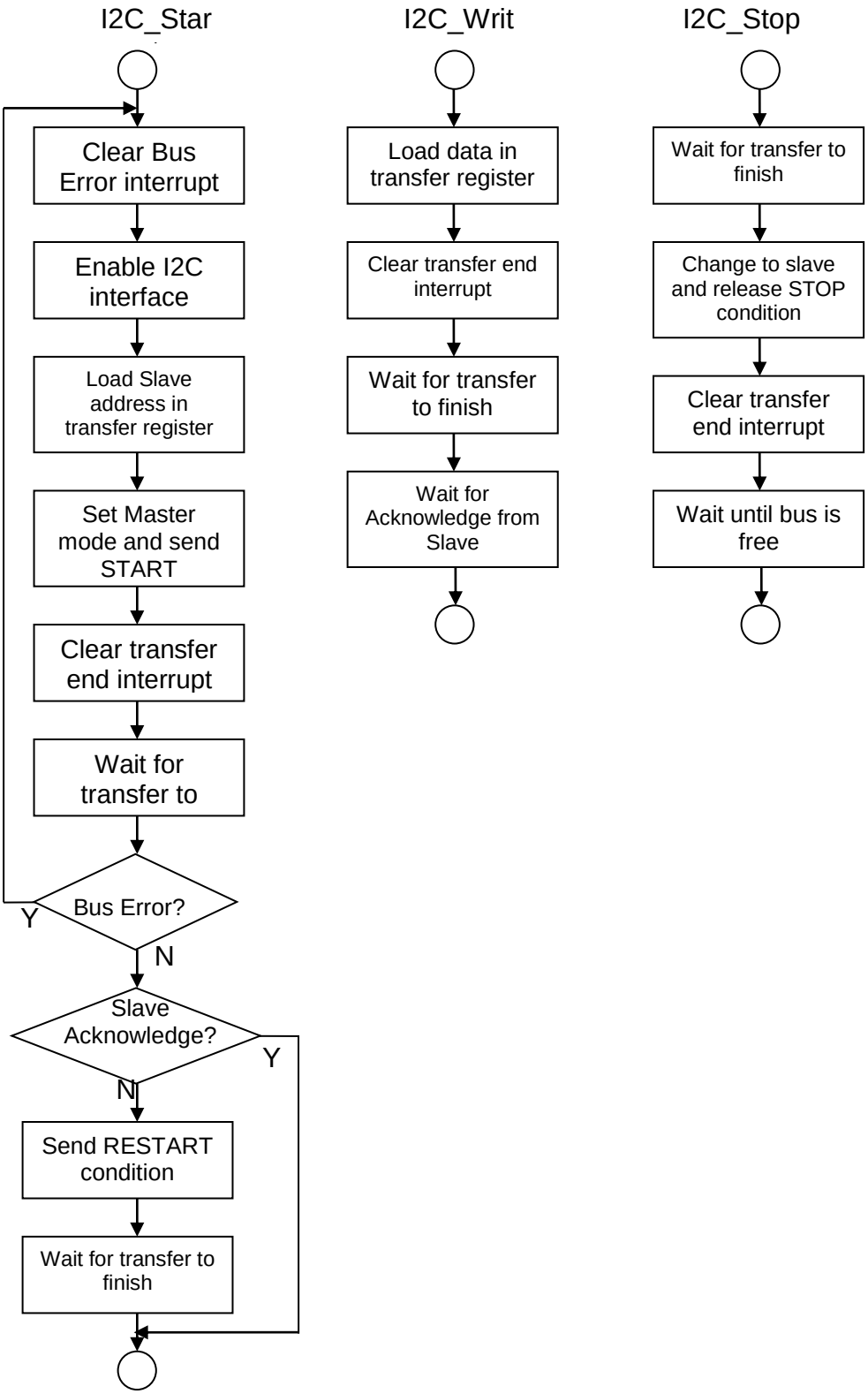
Writing to the EEPROM is done in several steps. First, the I2C controller is set to Master mode and the slave address is configured. Then, the memory address is transmitted to the EEPROM. Next, actual data to write is transmitted. Finally, the I2C controller is switched back to slave mode.

Timing Diagram

Figure 4. Byte Write



Flowchart


I2C_Star
Clear Bus Error interrupt
Enable I2C interface
Load Slave address in transfer register
Set Master mode and send START
Clear transfer end interrupt
Wait for transfer to
Bus Error?
Y
N
Slave Acknowledge?
Y
N
Send RESTART condition
Wait for transfer to finish
I2C_Writ
Load data in transfer register
Clear transfer end interrupt
Wait for transfer to finish
Wait for Acknowledge from Slave
I2C_Stop
Wait for transfer to finish
Change to slave and release STOP condition
Clear transfer end interrupt
Wait until bus is free

Code

```
voidI2C_Acknowledge(void)
{
while(IBSR0_LRB == 1);// no answer from slave, program will get stuck here
// a timeout mechanism should be implemented here
}

void I2C_Start(unsigned char slave_address)
{
do
  {
    IBCR0_BER = 0;                // clear bus error interrupt flag
    ICCR0_EN  = 1;                // enable I2C interface
    IDAR0 = slave_address;        // slave_address is sent out with start condition

    IBCR0_MSS = 1;                // set master mode and set start condition
    IBCR0_INT = 0;                // clear transfer end interrupt flag

while(IBCR0_INT == 0);        // look if transfer is in process
  }
while (IBCR0_BER == 1);         // retry if Bus-Error detected

while(IBSR0_LRB == 1)           // no acknowledge means device not ready
  {                               // maybe last write cycle not ended yet
    IBCR0_SCC = 1;                // try restart (= continue)
while (IBCR0_INT == 0);       // wait that transfer is finished
  }
}

void I2C_Stop(void)
{
while (IBCR0_INT == 0);         // wait that transfer is finished
  IBCR0_MSS = 0;                  // change to slave and release stop condition
  IBCR0_INT = 0;                  // clear transfer end interrupt flag
  while (IBSR0_BB);               // wait till bus free
}

void I2C_Write(unsigned char value)
{
  IDAR0 = value;                  // load data or address into register
  IBCR0_INT = 0;                  // clear transfer end interrupt flag
while (IBCR0_INT == 0);         // look if transfer is in process
  I2C_Acknowledge();              // wait for Acknowledge
}

void main(void)
{
  // write to EEPROM

  I2C_Start(0xA0 | WRITE);        // send slave address

  I2C_Write(0x00);                // Address where to write to

  I2C_Write(65);                  // Databyte 'A'
  I2C_Write(66);                  // Databyte 'B'
  I2C_Write(67);                  // Databyte 'C'

  I2C_Stop();
}
```

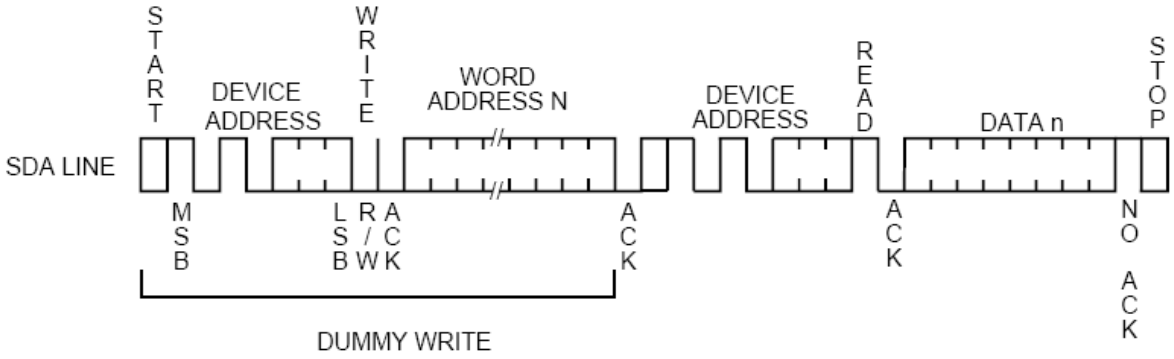
2.4.3 Read from EEPROM

Reading from the EEPROM is performed in several steps. First, a START condition is generated. A write command containing the memory address is sent next. This is followed by a RESTART condition and the actual read cycles. Please note that after the last read byte, no-acknowledge has to be generated by the Master. Finally, a STOP condition is generated.

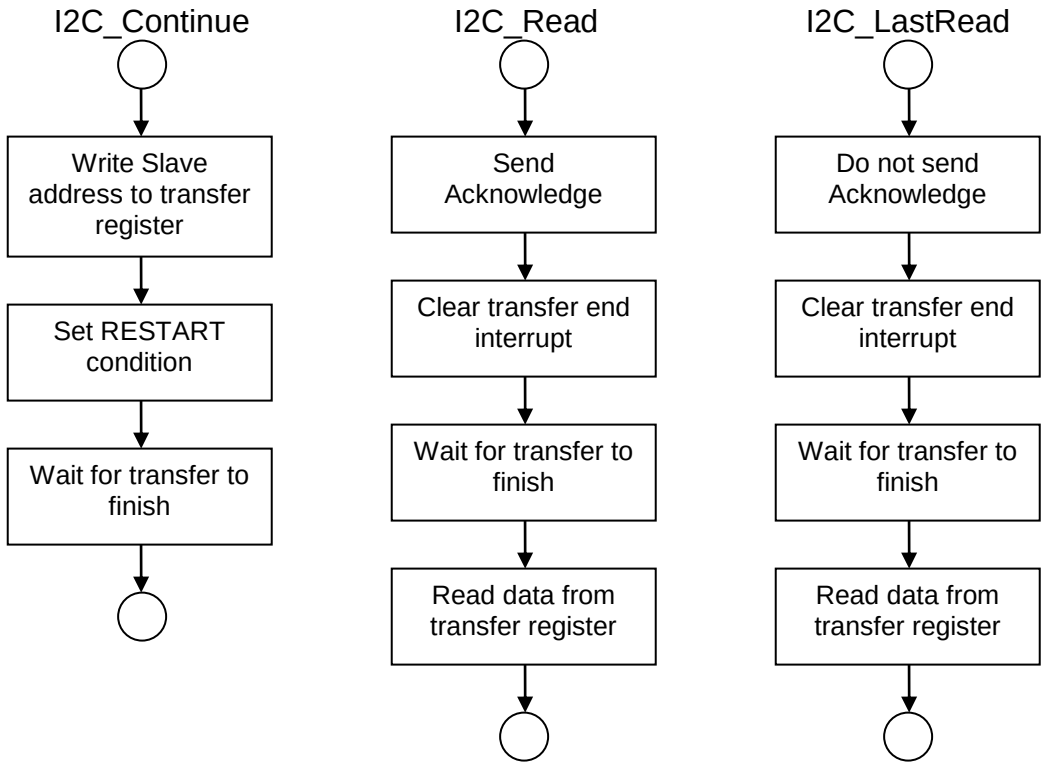
The START, write, and STOP functions are the same as for writing data to the EEPROM. In the following, only the flowchart of the RESTART condition and the actual read function is shown. For the other functions, please refer to the write command.

Timing Diagram

Figure 5. Random Read



2.4.4 Flowchart



Code

```
unsigned char i, result[16];

void I2C_Continue(unsigned char slave_address)
{
  IDAR0 = slave_address;          // slave_address is sent out with start condition
  IBCR0_SCC = 1;                  // restart (= continue)
while (IBCR0_INT == 0);         // wait that transfer is finished
}

unsigned char I2C_Read(void)
{
  IBCR0_ACK = 1;                  // acknowledge has to be sent
  IBCR0_INT = 0;                  // clear transfer end interrupt flag
while (IBCR0_INT == 0);         // wait that transfer is finished
return(IDAR0);                  // read received data out
}

unsigned char I2C_LastRead(void)
{
  IBCR0_ACK = 0;// no acknowledge has to be send after last byte
  IBCR0_INT = 0;// clear transfer end interrupt flag
while(IBCR0_INT == 0);// wait that transfer is finished
return(IDAR0);// read received data out
}

void main(void)
{
  // read from EEPROM

  I2C_Start(0xA0 | WRITE);        // Write Command to ...

  I2C_Write(0x00);                // ... set address from where to read

  I2C_Continue(0xA0 | READ);      // Restart, with READ command

  i = 0;

  result[i++] = I2C_Read();       // receive data from EEPROM
  result[i++] = I2C_Read();       // receive data from EEPROM
  result[i++] = I2C_LastRead();   // receive last byte from EEPROM

  I2C_Stop();
}
```

3 Additional Information

Information about Cypress Microcontrollers can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

The software examples related to this application note is:

96340_i2c

It can be found on the following Internet page:

http://www.cypress.com/products/16FX

4 Document History

Document Title: AN204776 - F²MC-16FX Family, I2C

Document Number: 002-04776

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	MKEA	09/04/2006	Initial Release
			09/11/2006	Revised Overview
			03/05/2007	Revised ACK logic for reading; add Appendix
			06/21/2007	Revised Introduction
			09/27/2007	Updated as per review comments
			07/09/2008	Add information on Port Input Enable and internal pull-up resistors.
*A	5060630	MKEA	12/22/2015	Migrated Spansion Application Note from MCU-AN-300227-E-V15 to Cypress format
*B	5839018	AESATMP8	07/31/2017	Updated logo and Copyright.
*C	6036665	NOFL	01/20/2018	Updated logo. Updated links. Updated Sales page and Copyright year.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2006-2018. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.