AN205502

F²MC-8FX Family MB95200 Series I²C SW Realization using GPIO

This application note describes how to use the GPIO to realize I²C function on the MB95200 series.

Contents

1 Introduction....1
2 Overview of I²C....1
2.1 Background....1
2.2 Protocol....2
3 I²C Driver....4
3.1 Peripheral Usage....4
3.2 Driver Code....4
4 Typical Application....11
4.1 HW Design....11
4.2 Sample Code....11
4.3 I²C Wave (E2PROM Read and Write Wave)....13
5 Notes on Items....14
6 More Information....14

1 Introduction

In this document, we will introduce how to use the GPIO to realize I²C function on the MB95200 series.

2 Overview of I²C

2.1 Background

In consumer electronics, telecommunications and industrial electronics, there are often many similarities among different designs. The following blocks are nearly included in every system.

Some intelligent control, usually a single-chip microcontroller

General-purpose circuits like LCD drivers, remote I/O ports, RAM, E2PROM, or data converters

Application-oriented circuits, such as digital tuning and signal processing circuits for radio and video systems, or DTMF generators for telephones with tone dialing

In these blocks listed above, the control and communication among different chips may be performed. To maximize hardware efficiency and circuit simplicity, Philips developed a simple bi-directional 2-wire bus for efficient inter-IC control. This bus is called "inter IC" or "I²C-bus".

Now, Philips' IC includes more than 150 CMOS and bipolar I²C-bus compatible used to perform corresponding functions mentioned above. All I²C-bus compatible with devices build in an on-chip interface which supports them to communicate directly each other via I²C-bus. This concept has solved many problems related to interface when designing digital control circuits.

Here are some features of I²C-bus:

Only two lines are required: a serial data line (SDA) and a serial clock line (SCL).

For each device connected to the bus, only an independent address and a simple master/slaver relationship are needed. The master can operate as a transmitter or a receiver.

A true multi-master mode can be realized only by bus arbitration and collision detection.

In 8-bit bi-directional serial transfer, data transfer rate can be up to 100 Kbit/s in standard-mode and up to 400 Kbit/s in fast-mode and up to 3.4 Mbit/s in high speed mode.

On-chip bus filtering function can preserve the integrity of communication data on the bus.

The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance of 400 pF.

2.2 Protocol

2.2.1 START Condition and STOP Condition

In I²C communication, START (S) condition and STOP (P) condition should be set respectively.

SDA line is changed from HIGH to LOW while SCL is HIGH; this case is defined as a START condition.

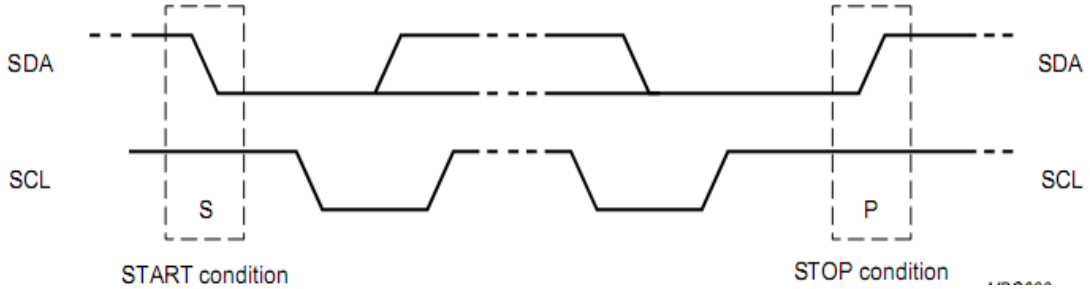
SCL is changed from LOW to HIGH while SCL is HIGH; this case is defined as a STOP condition.

The START condition and the STOP condition are always generated by the master. The bus is busy when the START condition is generated and is free when the STOP condition is generated.

The bus also remains busy if a repeated START (Sr) is generated instead of a STOP condition. In the case above, the function of the START (S) condition and the repeated START (Sr) condition are regarded as the same.

It is easy to detect the START condition and the STOP condition by devices connected to the bus, if they build in necessary hardware interface. However, if the controller does not have such interface, it must sample the SDA line at least twice per clock period so that the master can recognize the START condition and the STOP condition.

Figure 1. START Condition and STOP Condition



2.2.2 Byte Format

Each byte on the SDA line should be 8bits long. The number of bytes that can be transferred at a time is arbitrary. Each byte should be followed by an acknowledge bit. Data transfer starts from the most significant bit (MSB first). If the slaver can not receive or transfer a complete byte data, for example, an internal interrupt processing, SCL line should be set as LOW to force the master into a wait sate. Data transfer can be performed until the slaver is ready for next byte data and releases SCL line.

In some cases, you can use a format different from I²C-bus format (for example, CBUS compatible devices). A message which starts with such an address can be terminated by generating a STOP condition, even during a byte transfer. In this case, a no acknowledge is generated.

2.2.3 Acknowledge

An acknowledge is necessary for data transfer. The clock pulse related to acknowledge is generated by the master. During the acknowledge clock pulse, the transmitter releases the SDA line (HIGH), the receiver pulls down the SDA line and always remains LOW during the HIGH period of the clock pulse. Please note that setup time and hold time should also be considered.

Usually, a receiver which has been addressed generates an acknowledge every time after a byte data is received. An exception is that the receiver accesses on CBUS addressing and sends a message.

If a slaver can not recognize the slave address (for example, it is performing some real-time function), the SDA line should be left HIGH by the slaver. In this case, the master should terminate data transfer by generating a STOP condition or start a new transfer by generating a repeated START condition.

If a slave-receiver does not receive any byte data for some time after recognizing the slave address, the master must terminate data transfer again.

If the slaver generates a no acknowledge immediately after the first byte, the slaver will set SDA as HIGH, so the master should generate a STOP condition or a repeated START condition.

If the master-receiver also performs data transfer, it should send a no acknowledge to the slave-transmitter when the slaver has sent the last data. The slave-transmitter must release the SDA line so that the master can generate a STOP condition or a repeated START condition.

Figure 2. Data Transfer on the I²C-bus

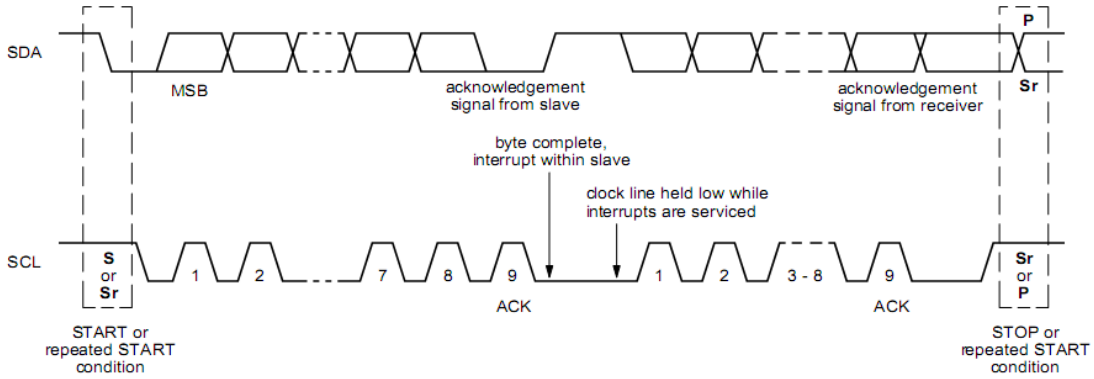
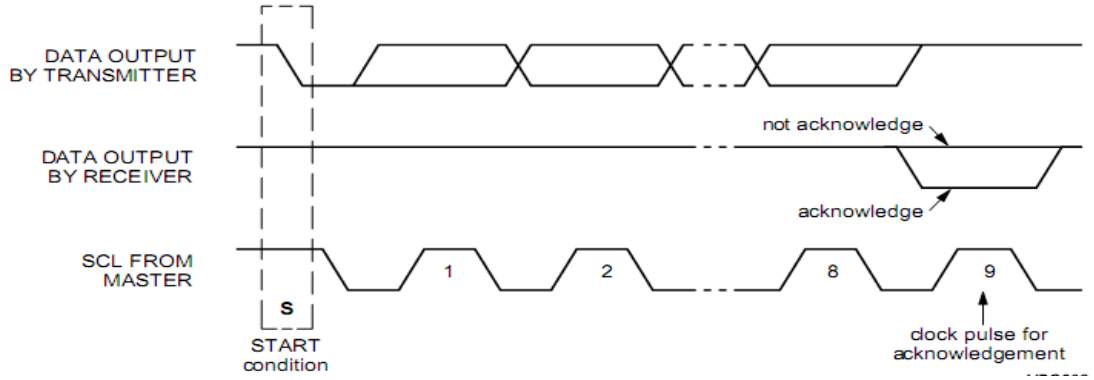


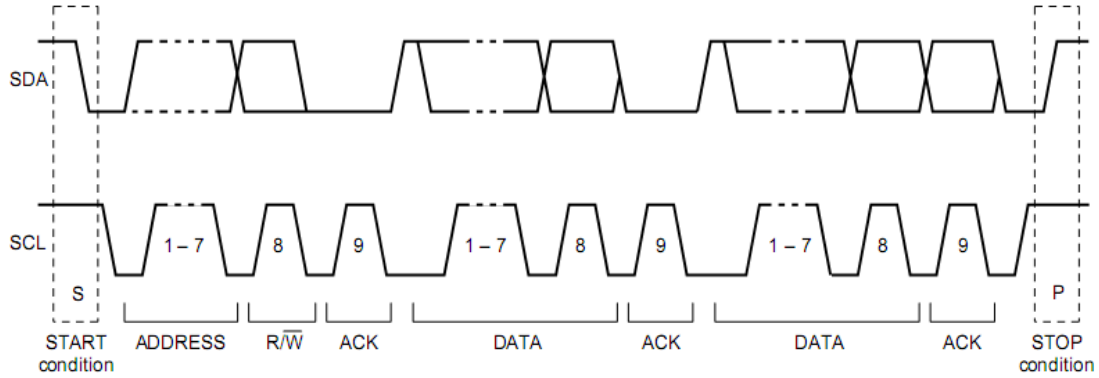
Figure 3. Acknowledge on the I²C-bus



2.2.4 A Complete Data Transfer

Data transfer format is shown in Figure 4 as below. A slave address is transferred after the START condition (S) is generated. This address is 7 bits long followed by the eighth bit which is a data direction bit (R/W) ("0": Write, '1': Read). Data transfer is always terminated by the STOP condition (P) generated by the master. However, if the master still needs to perform data transfer on the bus, it can generate a repeated START condition (Sr) and address another slaver without generating a STOP condition first. This way above can be used for various read/write formats.

Figure 4. A Complete Data Transfer on the I²C-bus



3 I²C Driver

3.1 Peripheral Usage

The MCU pins are used as below.

P62, used as SDA;

P63, used as SCL;

3.2 Driver Code

3.2.1 General Definition

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;        /* Unsigned  8 bit quantity */
typedef signed   char  INT8S;        /* Signed    8 bit quantity */
typedef unsigned int   INT16U;       /* Unsigned 16 bit quantity */
typedef signed   int   INT16S;       /* Signed   16 bit quantity */
typedef unsigned long  INT32U;       /* Unsigned 32 bit quantity */
typedef signed   long  INT32S;       /* Signed   32 bit quantity */

#define BOOL           BOOLEAN
#define BYTE           INT8U
#define UBYTE          INT8U
#define WORD           INT16U
#define UWORD          INT16U
#define LONG           INT32S
#define ULONG          INT32U
#define UCHAR          INT8U
```

#define UINT INT16U

#define DWORD INT32U

#define TRUE 1

#define FALSE 0

#define SDA PDR6_P62

#define SCL PDR6_P63

3.2.2 I²C Routines

void I2C_BusInit()

Return : None.

Parameters : None.

Description : Initialize the I²C.

Example :

I2C_BusInit ();

```
void I2C_BusInit()
{
  DDR6_P62=1; //SDA-OUT
  DDR6_P63=1; //SCL-OUT

  PDR6_P62=0; //SDA-0
  PDR6_P63=0; //SCL-0
}
```

void DelayUS(UINT nDly)

Return : None.

Parameters :

nDly,delay US.

Description : Delay nDly on the I²C.

Example :

DelayUS(3);

```
void DelayUS(UINT nDly)
{
  for(;nDly>0;nDly--){}
}
```

Note: The clock loop varies depending on MCU, please change the ‘DelayUS()’ function to realize the μs delay.

void Start ()

Return : None.

Parameters : None.

Description : Start I²C transfer.

Example :

Start ();

```
void Start()
{
  SCL=0;
  SDA=1;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
  SDA=0;
  DelayUS(2);
}
```

void Stop()

Return : None.

Parameters : None.

Description : Stop I²C transfer.

Example :

Stop ();

```
void Stop()
{
  SCL=0;
  SDA=0;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
  SDA=1;
  DelayUS(2);
}
```

void SetAck()

Return : None.

Parameters : None.

Description : Set an acknowledge on the I²C.

Example :

SetAck ();

```
void SetAck()
{
  SCL=0;
  DelayUS(2);
  SDA=0;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
}
```

BOOL GetAck()

Return : 0,failure 1,success.

Parameters : None.

Description : Get an acknowledge from the I²C.

Example :

BOOL bAck;

bAck=GetAck ();

```
BOOL GetAck()
{
  SCL=0;
  DelayUS(2);
  SDA=1;
  DDR6_P62=0; //SDA-IN
  SCL=1;
  DelayUS(2);

  return SDA? FALSE:TRUE;
}
```

void NoAck()

Return : None.

Parameters : None.

Description : Set a no acknowledge on the I2C.

Example :

NoAck ();

```
void NoAck()
{
  SCL=0;
  DelayUS(2);
  SDA=1;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
}
```

void Write8Bit(UCHAR uDat)

Return : None.

Parameters :

uDat, 8-bit data.

Description : Write a 8-bit data on the I^2C.

Example :

Write8Bit (0x63);

```
void Write8Bit(UCHAR uDat)
{
  UCHAR i;

  SCL=0;
  DDR6_P62=1; //SDA-OUT
  for(i=8;i!=0;i--)
  {
    DelayUS(2);
    SDA=(uDat&0x80)? 1:0;
    DelayUS(2);
    SCL=1;
    DelayUS(2);
    SCL=0;
    uDat<<=1;
  }
}
```

UCHAR Read8Bit()

Return : Data from the I²C.

Parameters : None.

Description : Read a 8-bit data from the I²C.

Example :

UCHAR tmp;

tmp=Read8Bit ();

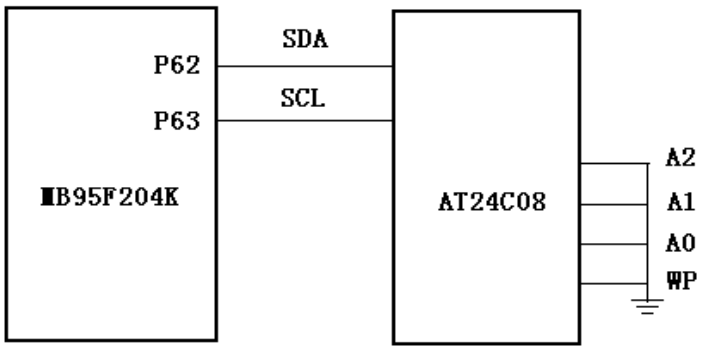
```
UCHAR Read8Bit()
{
  UCHAR i;
  UCHAR uRByte=0;

  SCL=0;
  SDA=1;
  DDR6_P62=0; //SDA-IN
  for(i=8;i!=0;i--)
  {
    DelayUS(2);
    SCL=1;
    DelayUS(2);
    uRByte<<=1;
    uRByte|=(UCHAR)(SDA);
    SCL=0;
  }

  return uRByte;
}
```

4 Typical Application

4.1 HW Design

In this application, MB95F204K is used as MCU chip and E2PROM (AT24C08) is used as the I²C device. The HW is designed as Figure 5.

Figure 5. Hardware Design



4.2 Sample Code

The codes are applied to AT24C08. Please refer to the AT24C08 manual for more information. In the demo, the address for AT24C08 is set as 0xA0 and the baudrate is set as 1 kbit/s.

BOOL Rom_Write(UBYTE romAddr,UBYTE datAddr,UBYTE data)

Return : 0,failure 1,success.

Parameters :

romAddr,the AT24C08 address;

datAddr,the memory space address;

data, data written to the AT24C08.

Description : Write a data to the AT24C08 by I²C.

Example :

```
BOOL bSuc;
bSuc=Rom_Write(0xA0,0x35,0x62);
```

```
BOOL Rom_Write(UBYTE romAddr,UBYTE datAddr,UBYTE data)
{
  Start();
  Write8Bit(romAddr&0xFE); //I2C write address
  if(!GetAck())
    return FALSE;

  Write8Bit(datAddr);
  if(!GetAck())
    return FALSE;
  Write8Bit(data);
  if(!GetAck())
    return FALSE;
  Stop();
  return TRUE;
}
```

UBYTE Rom_Read(UBYTE romAddr,UBYTE datAddr)

Return : data from AT24C08.

Parameters :

romAddr,the AT24C08 address;

datAddr,the memory space address;

Description : Read a data from the AT24C08 by I^2C.

Example :

UBYTE tmp;

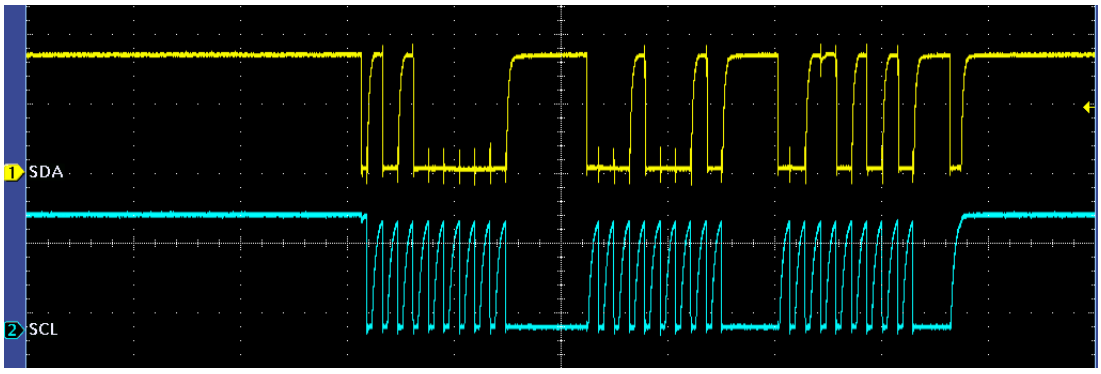
tmp=Rom_Read (0xA1,0x35);

```
UBYTE Rom_Read(UBYTE romAddr,UBYTE datAddr)
{
  UBYTE value;

  Start();
  Write8Bit(romAddr&0xFE); //I2C write address
  if(!GetAck())
    return FALSE;
  Write8Bit(datAddr);
  if(!GetAck())
    return FALSE;

  Start();
  Write8Bit(romAddr|0x01); //I2C read address
  if(!GetAck())
    return FALSE;

  value=Read8Bit();
  NoAck();
  Stop();

  return value;
}
```

4.3 I²C Wave (E2PROM Read and Write Wave)

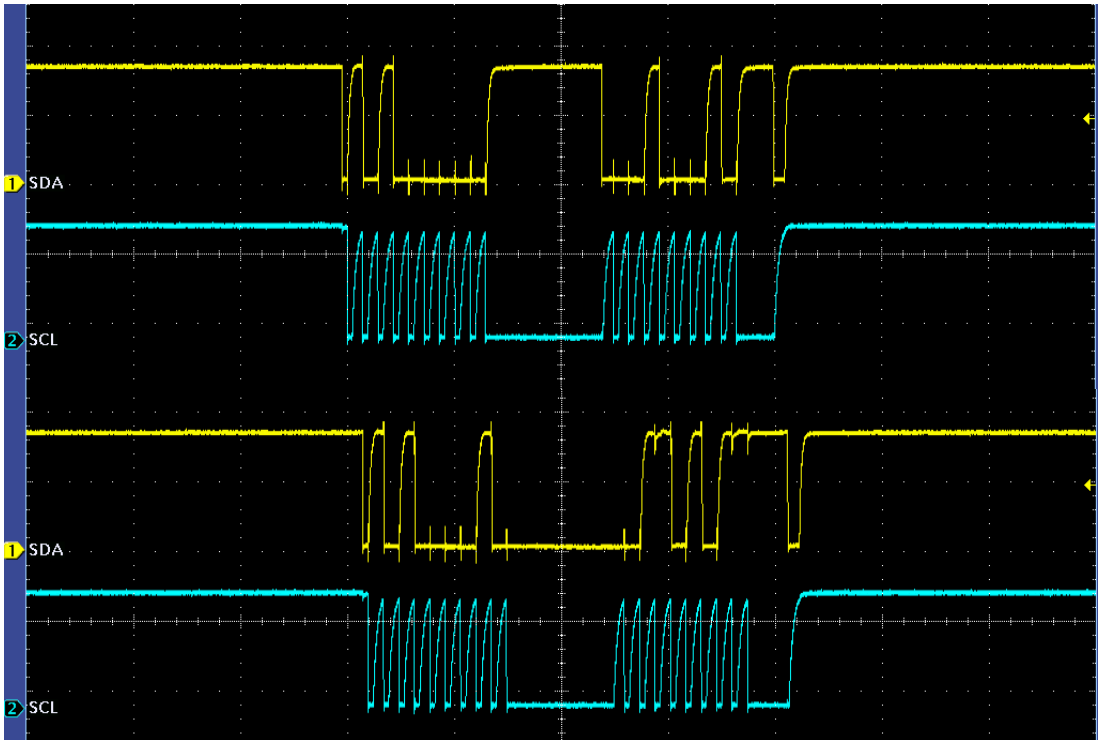
4.3.1 Write Wave-->Rom_Write (0xA0,0X11,0X35)

Figure 6. Write Wave



4.3.2 Read Wave-->Rom_Read (0xA1,0X11)

Figure 7. Read Wave



5 Notes on Items

I²C address bit0 setting :

The bit0 is set as "1" in read mode and is set as "0" in write mode.

In the function “void DelayUS(UINT nDly)”, the ‘nDly’ is different from the other MCU. Please set the ‘nDly’ according to MCU.

6 More Information

For more information on Cypress MB95200 products, please visit following website

www.cypress.com/documentation/application-notes/mb95200-i2c-sw-realization-using-gpio

Document History

Document Title: AN205502 - F²MC-8FX Family MB95200 Series I²C SW Realization using GPIO

Document Number: 002-05502

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	—	HUAL	02/04/2009	Initial release.
			03/24/2009	Modify the format.
*A	5261746	HUAL	05/10/2016	Migrated Spansion Application note from MCU-AN-500034-E-11 to Cypress format.
*B	5848944	MALI	08/09/2017	Updated logo and copyright

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

Cypress Developer Community

Forums | WICED IOT Forums | 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, 2009-2017. 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. 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.