Freescale Semiconductor
Application Note

Document Number: AN3291
Rev. 2, 04/2007

How to Use IIC Module on M68HC08, HCS08, and HCS12 MCUs

by: Stanislav Arendarik
Application Engineer
Roznov, Czech Republic

1 Introduction

This application note is an example of how to use the IIC module on Freescale's MCUs. The IIC module can be used in the master or slave mode, respectively. In this case, the master mode communicates with the serial EEPROM because the IIC bus is used mainly for communication between an MCU and a IIC peripheral device. The IIC bus can communicate directly between two MCUs, but the SPI bus is better for this task.

This application note summarizes the common IIC bus states and definitions and provides an example of how to communicate with serial EEPROMs (24C16 and 24C512). You can easily use another IIC device instead of an EEPROM, but you must change the IIC address byte that characterizes the slave device.

Contents

1 Introduction 1
2 IIC Bus Summary 2
2.1 IIC Bus Terminology 2
2.2 Bit Transfer 2
2.3 START and STOP Conditions 3
2.4 Bus Communication 3
2.5 Control Byte 3
2.6 Address Byte 4
2.7 Acknowledge 4
2.8 Read/Write Format 5
3 IIC Software Routines for MCUs 5
3.1 Initialization of the IIC 6
3.2 Write Function 7
3.3 Read Function 9
3.4 Example with Interrupt Used 13
3.4.1 MCU as Master 13
3.4.2 MCU as Slave 16
4 Conclusion 17

© Freescale Semiconductor, Inc., 2007. All rights reserved.

freescale™ semiconductor

2 IIC Bus Summary

The IIC bus is a bidirectional, two-wire bus based on the wired AND (open drain) connection between master and slave devices. For proper functionality, pull-up resistors are used. Each transfer originates from the master device and is acknowledged or answered by the slave device.

2.1 IIC Bus Terminology

The following IIC bus terminology is used in this application note:

- **Transmitter** — Device that sends data to the bus. A transmitter can put data on the bus of its own accord (master transmitter), or respond to a request for data from another device (slave transmitter).
- **Receiver** — Device that receives data from the bus.
- **Master** — Component that initializes a transfer, generates the clock signal, and terminates the transfer. A master can be a transmitter or a receiver.
- **Slave** — Device addressed by master. A slave can be a receiver or a transmitter.
- **Multi-master** — Ability for more than one master to co-exist on the bus without collision or data loss.
- **Arbitration** — Prearranged procedure that authorizes only one master at a time to take control of the bus.
- **Synchronization** — Prearranged procedure that synchronizes the clock signals provided by two or more masters.
- **SDA** — Data signal line (Serial DAta).
- **SCL** — Clock signal line (Serial CLock).

The SDA and SCL are bidirectional lines connected to a positive supply voltage via a current source or pull-up resistor. When the bus is free, both lines are high. Data on the IIC bus can transfer at rates up to 100 kbit/s in standard mode, up to 400 kbit/s in fast mode or up to 3.4 Mbit/s in high-speed mode. The number of devices connected to the bus depends on only the bus capacitance; the limit is 400 pF.

2.2 Bit Transfer

The data on the SDA line must be stable during the clock's high period. The SDA line's high or low state changes only when the clock signal on the SCL line is low. See Figure 1.

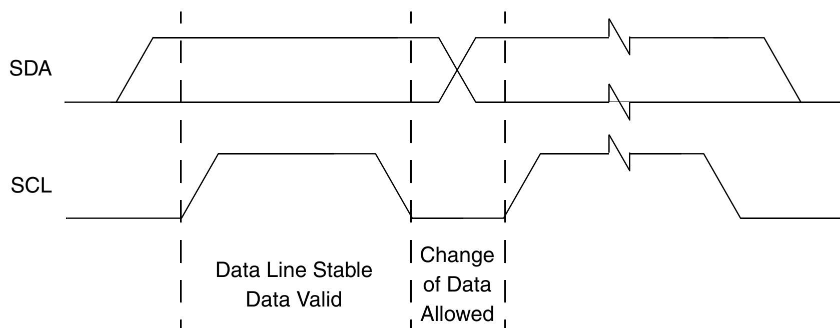

Figure 1. Bit Transfer

2.3 START and STOP Conditions

Within the IIC bus procedure, unique situations arise, which are defined as START and STOP conditions (see Figure 2). While the SCL line is high, a high-to-low transition on the SDA line is a START condition. While the SCL line is high, a low-to-high transition on the SDA line is a STOP condition. The master always generates the START and STOP conditions. The bus is free if the SDA and SCL lines remain at a high level after STOP and before START conditions.

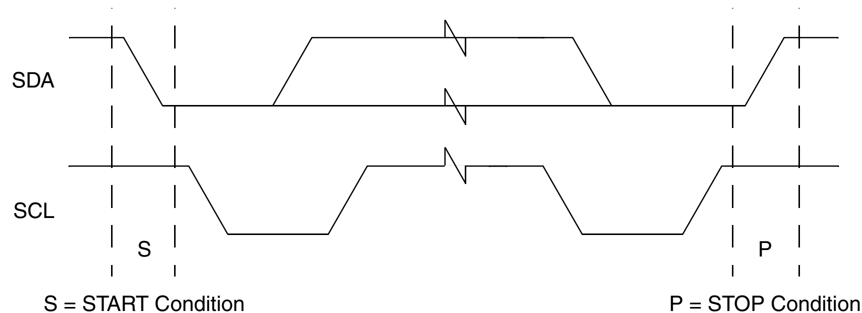

Figure 2. START and STOP Conditions

2.4 Bus Communication

The bus transfer protocol is based on the byte transfers followed by an acknowledge bit (ACK). The byte transfer starts with the most significant bit (MSB) to the least significant bit (LSB). The SCL positive clock impulses clock each bit. The 9th SCL impulse clocks the ACK bit.

Every bus transfer originates from a master device which sends the START condition to the bus. The the next consecutive bytes' sequence depends on the data being transferred through the IIC bus.

2.5 Control Byte

The control byte always follows the START condition. The control byte selects and activates specific devices on the bus; see Figure 3 for the control-byte structure. The first four bits represent the slave device's control code. For example, the control byte for the serial EEPROM is A0 (control code is '1010' binary). The next three bits represent the chip selects of the connected EEPROMs on the bus. The EEPROM uses these three bits for multiple device operations, but they are used as the higher bits of the internal EEPROM address if the memory address byte is wider than 8 or 16 bits. This address depends on the EEPROM capacity. For example, if the capacity is in the range of 16 kbits, these three bits represent the highest three bits of the address.

The 8th bit is the R/W (read/write) bit. This bit determines the requested operation for the slave device:

- If bit equals 0, this means a write operation
- If bit equals 1, this means a read operation.

The last 9th bit is the ACK bit and, in this case, always generated by the slave.

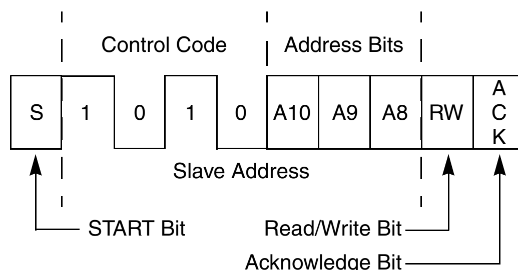

Figure 3. Control-Byte Structure

2.6 Address Byte

The address byte brings the address information to the slave device. This byte determines the real address in the EEPROM, to or from the data to be written or read. The address information structure depends on the connected EEPROM capacity. The EEPROMs up to 16 kbits use one address byte, but EEPROMs with higher capacity use two byte addresses (address H and address L). Address H is sent first. The structure of address bytes for a one-byte address are in Figure 4, and in Figure 5 for two byte addresses. The EEPROMs type 24C16 and 24C512 are used as communication devices.

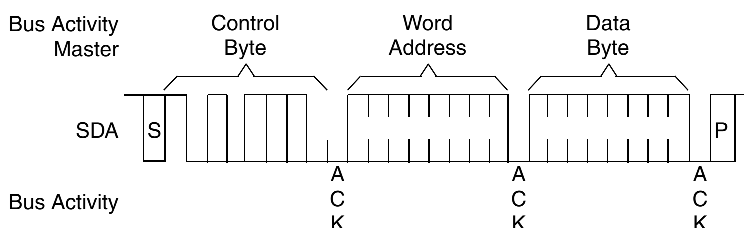

Figure 4. One Byte Address IIC Write Format

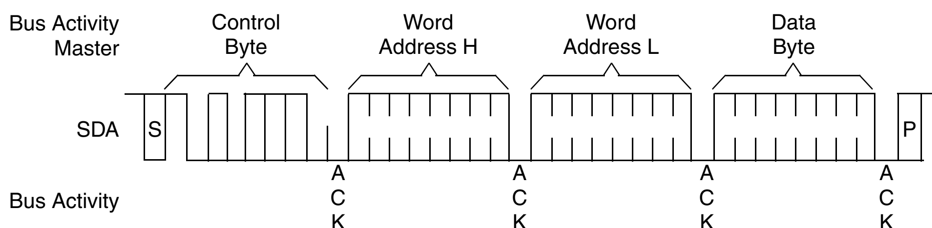

Figure 5. Two Byte Address IIC Write Format

2.7 Acknowledge

The acknowledge bit (ACK) is the low-level impulse on the SDA line during the 9th pulse on the SCL line. The master always generates the SCL pulses. The receiver generates ACK pulses. This means in the write operation the slave device generates ACK after each byte transfers. In the read operation, the slave generates ACK when the control byte and address byte are sent from the master; the master generates ACK when it receives the data from the slave.

2.8 Read/Write Format

There are two types of data transfers on the IIC bus:

- Read from the slave
- Write to the slave

When the write operation executes, a data byte follows the address byte. This data byte contains the actual data to be written to the address in the address byte; see Figure 4 and Figure 5.

When the read operation executes in the first part of the communication protocol:

1. The master writes the actual address to the slave.
2. The master sends the repeat START condition to the bus, followed by the control byte with R/W equal to 1.
3. The slave device sends ACK and immediately sends the actual requested data.

See Figure 6 for the read-operation structure.

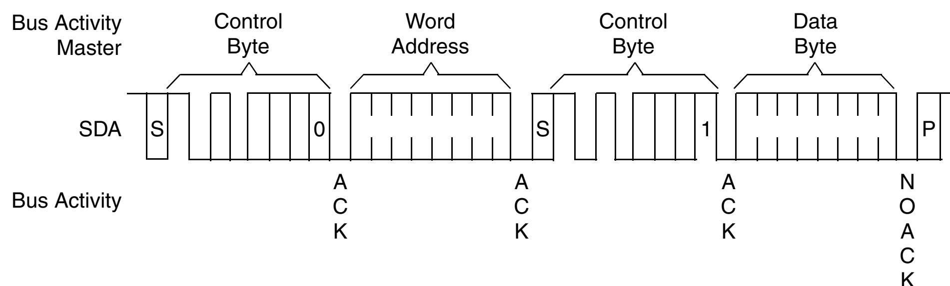

Figure 6. Read-Operation Structure

See Figure 4, Figure 5, and Figure 6 for read- and write-byte operations. The second style of data transfer is page-write and page-read. For more information on these styles refer to the slave-device data sheet.

3 IIC Software Routines for MCUs

Detailed descriptions of all IIC modules can be found in the MCU data sheet. A short review follows:

- **Address register** — Used differently in the HC908 than in the S08 and S12 MCU types. In the HC908 core, the address register's content is sent automatically after the START condition, and the data register content follows the condition. This address register also serves as its own IIC-slave address in slave mode. In the S08 and S12 core, the address register is active only in the slave mode and comprises its own IIC slave address of the MCU on the bus. This can be used by the multi-MCU connection on the bus.
- **Frequency divider register** — Determines the SCL clock speed. The bus frequency acts as the source frequency for the IIC module.
- **Control register** — Controls all IIC module functionality.
- **Status register** — Comprises the information about the actual bus state.
- **Data register** — Serves as the input and output point for all information transferred through the bus.

3.1 Initialization of the IIC

The main task is to set the right speed of the SCL clock signal. These clocks are based on the bus frequency. The next task is to set the IIC module functionality — to use or not use the interrupt service routine, activate the IIC module, and set the slave mode. Refer to Example 1, Example 2, and Example 3.

Example 1. Software for HC908 MCU (MC68HC908AP64 Used)

```
void init_I2C (void)
{
MMCR1_MMEN = 1;       // Enable IIC;
MMCR1_MMCLRBB = 1;    // Clear bus busy flag;
MMCR1_MMTXAK = 1;
MMCR2_MMRW = 0;       // R/W bit = 0;
MMCR2_MMAST = 0;      // Slave mode actually;
MMCR2_MMNAKIF = 0;
MMCR2_MMALIF = 0;     // Clear flags;
MMSR = 0;             // Clear all flags (Status register);
MMFDR = 3;            // Set speed to 25kHz for Bus = 4MHz;
}
```

Example 2. Software for 9S12 MCU (MC9S12DP256B Used)

```
void Init_I2C(void)
{
IBFD = 0x4C;          // frequency divider register: Bus = 8MHz => SCL = 91kHz;
IBAD = 0x00;          // slave address of this module;
IBCR = 0x80;          // enable I2C module and No interrupts;
}
```

Example 3. Software for the 9S08 MCU (MC9S08GB60 Used)

```
void Init_I2C (void)
{
IIC1C_IIC1EN = 1;     // Enable I2C;
IIC1C_TXAK = 1;       // not generate ACK by master after transfer;
IIC1C_MST = 0;        // Slave mode actually;
IIC1F = 0x99;         // Set speed to 50kHz for Bus = 18.8743MHz;
                      // 12.5k->0x39; 50k->0x99; 100k->0x59;
IIC1S_SRW = 0;        // R/W bit = 0;
}
```

3.2 Write Function

The write to slave at an address will be discussed first. To maintain this task, activate communication with the slave by sending the START bit followed by the slave-address byte. These bits' content are discussed in Section 2.6, "Address Byte." When the master sends the byte, the slave sends the ACK bit during the 9th SCL impulse. The slave must acknowledge each byte the master sends.

The next byte (second byte) is the internal address of the memory array. This address can be one or two bytes, depending on memory size. The slave must acknowledge each byte the master sends.

The third (fourth for large memory) is the data byte — this data will write to a specified address. The slave ACK follows this data byte. When the master receives this ACK bit, it generates the STOP bit. At this moment, the slave internally disconnects from the IIC bus and internally processes the write operation. While the slave processes the requested operation (write) it cannot answer the next master request. Therefore, the master must wait for the amount of time defined in the slave-device data sheet, or it must periodically send requests to the slave and test for the right answer (ACK from the slave).

Example 4. Write Byte for HC908 MCU (MC68HC908AP64 and 24C16 Used)

```
void I2C_write_byte (byte addr, byte wr_data)
{
   MMCR2_MMRW = 0;               // Set Write mode;
   MMADR = own_sl_addr;          // set combined address of Slave for write;
   MMDTR = addr;                 // set address of Slave to read;
      //-------start of transmit bytes to IIC bus-----
   MMCR2_MMAST = 1;              // Start transfer - Master bit = 1;
   while (!(MMSR_MMTXBE));       // wait till data transferred;
   while (MMSR_MMRXAK);          // wait for ACK from slave;
      //-----Slave ACK occurred------------
   MMDTR = wr_data;              // write data byte into EEPROM;
   while (!(MMSR_MMTXBE));       // wait till data transferred;
   MMDTR = 0xFF;                 // generate SCL impulse for slave to send ACK bit;
   while (MMSR_MMRXAK);          // wait for ACK from slave;
   MMCR2_MMAST = 0;              // STOP bit;
}
```

Example 6 and Example 7 are for a larger type of EEPROM (24C512). The word address is divided into two parts (two bytes): low byte and high byte, as defined in Example 5.

Example 5. Word Address Divided into Low Byte and High Byte

```
typedef union
{
  word EE_Address;
  struct
  {
    byte Address_H;
    byte Address_L;
  }Bytes;
}tAddr;

tAddr sADDR;
#define Address sADDR.EE_Address
#define Addr_L  sADDR.Bytes.Address_L
#define Addr_H  sADDR.Bytes.Address_H
```

Example 6. Write Byte for 9S08 MCU (MC9S08GB60 and 24C16 Used)

```
void IIC_write_byte(word addr, byte data)
{
  Address = addr;          // load address;
  temp = (Addr_H & 0x07) << 1;
  IIC1C_TXAK = 0;          // RX/TX = 1; MS/SL = 1; TXAK = 0;
  IIC1C |= 0x30;           // And generate START condition;
        //-------start of transmit first byte to IIC bus-----
  IIC1D = IIC_SLAVE | temp;  // Address the slave and set up for master transmit;
  while (!IIC1S_IICIF);  // wait until IBIF;
  IIC1S_IICIF=1;           // clear the interrupt event flag;
  while(IIC1S_RXAK);       // check for RXAK;
        //-----Slave ACK occurred------------
  IIC1D = Addr_L;             // Send low byte of the word address;
  while (!IIC1S_IICIF);  // wait until IBIF;
  IIC1S_IICIF=1;           // clear the interrupt event flag;
  while(IIC1S_RXAK);       // check for RXAK;
        //-----Slave ACK occurred------------
  IIC1D = data;
  while (!IIC1S_IICIF);  // wait until IBIF;
  IIC1S_IICIF=1;           // clear the interrupt event flag;
  while(IIC1S_RXAK);       // check for RXAK;
        //-----Slave ACK occurred------------
  IIC1S_IICIF=1;           // clear the interrupt event flag;
  IIC1C_MST = 0;           // generate STOP condition;
}
```

Example 7. Write Byte for 9S12 MCU (MC9S12XDT512 Used)

```
void IIC_write_byte(word addr, byte data)
{
  Address = addr;             // load address;
  IIC0_IBCR_TXAK = 0;         // TXAK = 0;
  IIC0_IBCR |= 0x30;         // RX/TX = 1; MS/SL = 1; TXAK = 0;
                              // And generate START condition;
        //-------start of transmit first byte to IIC bus-----
  IIC0_IBDR = slavewrite;     // Address the slave and set up for master transmit;
```

```
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;          // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);     // check for RXAK;
        //-----Slave ACK occurred------------
  IIC0_IBDR = Addr_H;        // Send high byte of the word address;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;          // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);     // check for RXAK;
        //-----Slave ACK occurred------------
  IIC0_IBDR = Addr_L;        // Send low byte of the word address;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;          // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);     // check for RXAK;
        //-----Slave ACK occurred------------
  IIC0_IBDR = data;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;          // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);     // check for RXAK;
        //-----Slave ACK occurred------------
IIC0_IBCR_MS_SL = 0; // generate STOP condition;
}
```

3.3 Read Function

The read function is similar to the write function. First, write the address (byte or word) into the slave. An ACK bit from the slave always follows these write cycles. When the address properly writes to the slave, the master generates the REPEAT START condition. Then, the master again sends the control byte but with the R/W bit equal to 1. This means the read-from-slave function. The ACK bit from the slave follows this byte. The master must then switch to receive mode and perform the read data from the IIC-data register (IICDR) function. This instruction generates the SCL pulses for the slave to send data from the requested address.

There are two possible options:

- If the master does not need the next data byte from the slave, it sends the NOACK (ACK = 1) and the STOP condition.
- If the master wants to read the next data byte from the slave, it generates the ACK bit (ACK = 0).

The slave recognizes this ACK bit and moves the internal address pointer to the next address. Then, the master performs the read data from the IICDR function to generate the next series of SCL pulses for the slave to send data. This sequence can perform infinitely.

Example 8. Read Byte for HC908 Family (HC908AP64 Used)

```
byte I2C_read_byte (byte addr)//output is byte "rd_data";
{
   MMCR2_MMRW = 0;            // Set Write mode;
   MMADR = own_sl_addr;       // set combined address of Slave for write;
   MMDTR = addr;              // set address;
        //-------start of transmit first byte to IIC bus-----
   MMCR2_MMAST = 1;           // Start transfer - Master bit = 1;
   while (!(MMSR_MMTXBE));    // wait till data transferred;
   while (MMSR_MMRXAK);       // wait for ACK from slave;
```

```
        //-----Slave ACK occurred------------
   MMCR2_MMRW = 1;             // set read operation;
   MMCR1_REPSEN = 1;           // enable repeat Start bit;
   MMCR1_MMTXAK = 0;           // Master will generate ACK;
   MMCR2_MMAST = 1;            // Start transfer - Master bit = 1;
        //------start of transmit Repeat start & "A1" to IIC bus-------
   MMDTR = 0xFF;               // send repeated start and combined address;
   while (!(MMSR_MMTXBE));     // wait till data transferred;
   while (MMSR_MMRXAK);        // wait for ACK from slave;
   MMDTR = 0xFF;               // send SCL clocks for the slave to send data;
   MMCR1_MMTXAK = 1;           // Disable master ACK after read byte from Slave;
   while (!(MMSR_MMRXBF));     // wait till data received;
   rd_data = MMDRR;            // read received data;
   MMCR2_MMAST = 0;            // generate STOP bit - End of transfer;
   return rd_data;
}
```

Example 9 and Example 10 are for a larger type of the EEPROM (24C512). The word address is divided into two parts (two bytes): low byte and high byte (see Example 5).

Example 9. Read Byte for 9S08 Family (MC9S08GB60 and 24C16 Used)

```
typedef union
{
   word EE_Address;
   struct
   {
      byte Address_H;
      byte Address_L;
   }Bytes;
}tAddr;

tAddr ADDR;

#define Address ADDR.EE_Address
#define Addr_L  ADDR.Bytes.Address_L
#define Addr_H  ADDR.Bytes.Address_H

byte IIC_read_byte(word addr)
{
  Address = addr;
  temp = (Addr_H & 0x07) << 1;
  IIC1C_TXAK = 0;           // RX/TX = 1; MS/SL = 1; TXAK = 0;
  IIC1C |= 0x30;            // And generate START condition;
  IIC1D = IIC_SLAVE | temp;    // Address the slave and set up for master transmit;
  while (!IIC1S_IICIF);     // wait until IBIF;
  IIC1S_IICIF=1;            // clear the interrupt event flag;
  while(IIC1S_RXAK);        // check for RXAK;
        //-----Slave ACK occurred------------
  IIC1D = Addr_L;           // Send low byte of word address;
  while (!IIC1S_IICIF);     // wait until IBIF;
  IIC1S_IICIF=1;            // clear the interrupt event flag;
  while(IIC1S_RXAK);        // check for RXAK;
        //-----Slave ACK occurred------------
  IIC1C_RSTA = 1;           // set up repeated start;
  IIC1D = IIC_SLAVE | temp | 1;  // (slave_address) | (RW = 1);
  while (!IIC1S_IICIF);     // wait until IBIF;
```

```
  IIC1S_IICIF=1;           // clear the interrupt event flag;
  while (IIC1S_RXAK);      // check for RXAK;
        //-----Slave ACK occurred------------
  IIC1C_TX = 0;            // set up to receive;
  IIC1C_TXAK  = 1;         // acknowledge disable;
  RD_data = IIC1D;         // dummy read;
  while (!IIC1S_IICIF);    // wait until IBIF;
  IIC1S_IICIF=1;           // clear the interrupt event flag;
  IIC1C_MST = 0;           // generate stop signal;
  RD_data = IIC1D;         // read right data;
  return RD_data;
}
```

Example 10. Read Byte for 9S12 Family (MC9S12DT512 and 24C512 Used)

```
typedef union
{
   word EE_Address;
   struct
   {
      byte Address_H;
      byte Address_L;
   }Bytes;
}tAddr;

tAddr ADDR;

#define Address ADDR.EE_Address
#define Addr_L  ADDR.Bytes.Address_L
#define Addr_H  ADDR.Bytes.Address_H

void IIC_read_byte(word addr)
{
  Address = addr;
  IIC0_IBCR_TXAK = 0;         // TXAK = 0;
  IIC0_IBCR |= 0x30;         // RX/TX = 1; MS/SL = 1; TXAK = 0;
                              // And generate START condition;
  IIC0_IBDR = slavewrite;     // Address the slave and set up for master transmit;
  while (!IIC0_IBSR_IBIF);    // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);      // check for RXAK;
        //-----Slave ACK occurred------------
  IIC0_IBDR = Addr_H;         // Send high byte of word address;
  while (!IIC0_IBSR_IBIF);    // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);      // check for RXAK;
        //-----Slave ACK occurred------------
  IIC0_IBDR = Addr_L;         // Send low byte of word address;
  while (!IIC0_IBSR_IBIF);    // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);      // check for RXAK;
        //-----Slave ACK occurred------------
  IIC0_IBCR_RSTA = 1;         // set up repeated start;
  IIC0_IBDR = slaveread;      // (slave_address) | (RW = 1);
  while (!IIC0_IBSR_IBIF);    // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while (IIC0_IBSR_RXAK);     // check for RXAK;
```

```
        //-----Slave ACK occurred------------
  IIC0_IBCR_TX_RX = 0;          // set up to receive;
  IIC0_IBCR_TXAK  = 1;          // acknowledge disable;
  RD_data = IIC0_IBDR;          // dummy read;
  while (!IIC0_IBSR_IBIF);      // wait until IBIF;
  IIC0_IBSR_IBIF=1;             // clear the interrupt event flag;
  IIC0_IBCR_MS_SL = 0;          // generate stop signal;
  RD_data = IIC0_IBDR;          // read right data;
}
```

Example 11 gives a routine for 9S12 MCU core and 24C512 EEPROM block-write and block-read functions.

Example 11. Block-Write / Block-Read for 9S12 Core and 24C512 EEPROM (MC9s12XDT512 Used)

```
void IIC_write_block(word addr, byte len)
{
  byte page;

  Address = addr;               // load address;
  page = len;                   // load length of data pack to be written;
  if(page > MAX_PAGE) page = MAX_PAGE;  // set limit;
  IIC0_IBCR_TXAK = 0;           // TXAK = 0;
  IIC0_IBCR |= 0x30;            // RX/TX = 1; MS/SL = 1; TXAK = 0;
                                // And generate START condition;
  IIC0_IBDR = slavewrite;       // Address the slave and set up for master transmit;
  while (!IIC0_IBSR_IBIF);      // wait until IBIF;
  IIC0_IBSR_IBIF=1;             // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);        // check for RXAK;
  IIC0_IBDR = Addr_H;           // Send high byte of word address;
  while (!IIC0_IBSR_IBIF);      // wait until IBIF;
  IIC0_IBSR_IBIF=1;             // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);        // check for RXAK;
  IIC0_IBDR = Addr_L;           // Send low byte of word address;
  while (!IIC0_IBSR_IBIF);      // wait until IBIF;
  IIC0_IBSR_IBIF=1;             // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);        // check for RXAK;
  for(i=0;i<page;i++)
  {
    IIC0_IBDR = WRData[i];
    while (!IIC0_IBSR_IBIF);    // wait until IBIF;
    IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
    while(IIC0_IBSR_RXAK);      // check for RXAK;
  }
  IIC0_IBCR_MS_SL = 0;          // generate stop signal;
}

void IIC_read_block(word addr,word len)
{
  byte dummy;

  Address = addr;
  length = len;
  if(length > MAX_LENGTH) length = MAX_LENGTH;
  IIC0_IBCR_TXAK = 0;          // TXAK = 0;
  IIC0_IBCR |= 0x30;         // RX/TX = 1; MS/SL = 1;
                               // And generate START condition;
```

```
  IIC0_IBDR = slavewrite;     // Address the slave and set up for master transmit;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);      // check for RXAK;
  IIC0_IBDR = Addr_H;        // Send high byte of word address;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);      // check for RXAK;
  IIC0_IBDR = Addr_L;        // Send low byte of word address;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while(IIC0_IBSR_RXAK);      // check for RXAK;
  IIC0_IBCR_RSTA = 1;        // set up repeated start;
  IIC0_IBDR = slaveread;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  while (IIC0_IBSR_RXAK);     // check for RXAK;
  IIC0_IBCR_TX_RX = 0;        // set up to receive;
  dummy = IIC0_IBDR;          // dummy read;
  for(i=0;i<length-1;i++)
  {
    while (!IIC0_IBSR_IBIF); // wait until IBIF;
    IIC0_IBSR_IBIF=1;         // clear the interrupt event flag;
    RDData[i] = IIC0_IBDR;    // save data to RAM;
  }
  IIC0_IBCR_TXAK  = 1;       // acknowledge disable;
  while (!IIC0_IBSR_IBIF);  // wait until IBIF;
  IIC0_IBSR_IBIF=1;           // clear the interrupt event flag;
  IIC0_IBCR_MS_SL = 0;        // generate stop signal;
  RDData[i] = IIC0_IBDR;      // save data to RAM;
}
```

3.4 Example with Interrupt Used

This section shows two options when the IIC interrupt service routine is used:

- MCU as master and write or read from slave used
- MCU as slave on IIC bus

3.4.1 MCU as Master

This routine is used when the main software loop cannot poll the IIC communication sequence. The first action executes in only the main loop; this is the first write into the IIC data register. The IIC-interrupt-service routine then maintains the whole communication. Example 12 explains how to set the MCU.

Example 12. IIC-Module-Initialization Routine

```
void Init_IIC(void)
{
  IICF = 0;         // frequency divider register: Bus = 8MHz => SCL = 400kHz;
  IICA = 0;         // slave address of this module;
  IICS = 0x12;      // clear the IBAL and IICIF flags;
  IICC = 0xC0;      // enable IIC module with interrupt;
```

```
}
```

Example 13 shows the Read_block and Write_block functions for the serial EEPROM 24AA256 and the MC9S08QG8 MCU in the main loop:

Example 13. Read_block and Write_block Functions (24AA256 EEPROM and MC9S08QG8 Used)

```
void IIC_read_block(word addr,byte len)
{
  Address = addr;
  length = len;
  i = 0;                   // for interrupt only;
  IIC_Res_flg = 0;         // for interrupt only;
  flag = 10;               // defines receive function in the interrupt service routine;
  if(length > MAX_LENGTH)  length = MAX_LENGTH;
  IICC_TXAK  = 0;          // RX/TX = 1; MS/SL = 1, TXAK = 0;
  IICC |= 0x30;            // And generate START condition;
  IICD = IIC_SLAVE;        // Address the slave and set up for master transmit;
}

void IIC_write_block(word addr, byte len)
{
  Address = addr;           // load address;
  length = len;             // load length of data pack to be written;
  i = 0;                    // for interrupt only;
  IIC_Res_flg = 0;          // for interrupt only;
  flag = 2;                 // defines transmit function in the interrupt service routine;
  if(length > MAX_PAGE) length = MAX_PAGE;  // set limit;
  IICC_TXAK = 0;            // RX/TX = 1; MS/SL = 1; TXAK = 0;
  IICC |= 0x30;             // And generate START condition;
  IICD = IIC_SLAVE;         // Address the slave and set up for master transmit;
}
```

The flag variable is byte variable used in the IIC interrupt service routine to recognize which byte of communication protocol is processed.

Example 14 shows the read function from the serial EEPROM (24LC16B) and the MCU (S12DP256B) in the main loop.

Example 14. EEPROM (24LC16B) and MCU (S12DP256B) in Main Loop

```
void i2c_read_byte(unsigned char read_addr)
{
  flag =  0;
  temp = read_addr;
  IBCR_MS_SL = 1;          // Set transmit and master mode;
  IBCR_TX_RX = 1;          // And generate start condition;
  IBDR = slave_IIC_addr;
}
```

The temp variable is a common byte variable; the flag variable is used in the IIC interrupt service routine to recognize which communication-protocol byte is processed. Then, the interrupt-service-routine code would be as shown in Example 15.

Example 15. Interrupt-Service-Routine Code for MC9S08QG8 and 24C512 EEPROM

```
__interrupt void isrViic(void)
{
  IICS_IICIF = 1;               // clear the interrupt event flag;
  switch(flag)
  {
    case 2:                     // TX mode;
    {
      IICD = Addr_H;            // Send high byte of word address;
      flag++;
      return;
    }
    case 3:                     // TX mode;
    {
      IICD = Addr_L;            // Send low byte of word address;
      flag++;
      return;
    }
    case 4:                     // TX mode;
    {
      if(length)
      {
        IICD = WR_Data[i];
        length--;
        i++;
        return;
      }
      else
      {
        IICC_MST = 0;           // generate stop signal;
        flag = 0;
        IIC_Res_flg = IIC_OK;   // END of write cycle;
        return;
      }
    }
    case 10:
    {
      IICD = Addr_H;             // Send high byte of word address;
      flag++;
      return;
    }
    case 11:
    {
      IICD = Addr_L;            // Send low byte of word address;
      flag++;
      return;
    }
    case 12:
    {
      IICC_RSTA = 1;           // set up repeated start;
      IICD = IIC_SLAVE | 1;
```

```
      flag++;
      return;
    }
    case 13:
    {
      IICC_TX = 0;              // set up to receive;
      dummy = IICD;             // dummy read;
      flag++;
      return;
    }
    case 14:
    {
      if(length > 1)
      {
        RD_Data[i] = IICD;      // save data to RAM;
        length--;
        i++;
        return;
      }
      else if(length == 1)
      {
        IICC_TXAK  = 1;         // acknowledge disable;
        RD_Data[i] = IICD;       // save data to RAM;
        i++;
        flag++;
        return;
      }
    }
    case 15:
    {
      IICC_MST = 0;              // generate stop signal;
      //RD_Data[i] = IICD;       // save data to RAM;
      flag = 0;
      IIC_Res_flg = IIC_OK;
      return;
    }
  }
}
```

3.4.2 MCU as Slave

This routine is used when the MCU is presented as a slave device on the IIC bus. The MCU needs to catch all IIC bytes intended for it. The recognition of the valid bytes is based on the match of its own slave address saved in the IBAD register. If the master sends an address to the IIC bus, the slave generates an IIC interrupt when this address matches its own slave address in the IBAD register. Then, for example, the slave saves all bytes from the IIC bus to a RAM buffer and processes them later.

Example 16 shows the slave MCU (S12DP256B) connected to a master MCU of the same type (two EVBS12DP256B boards connected together).

The variables defined are:

```
byte RD_data[20];
byte i2c_cnt, temp;
```

Example 16. Slave-Initialization Routine

```
void Init_IIC(void)
{
  IBFD = 0x2B;         // frequency divider register: Bus = 8MHz => SCL = 91kHz;
  IBAD = My_IIC_addr; // slave address of this module;
  IBCR = 0xC0;         // enable IIC module and interrupts, RX mode, Slave mode, ACK = 0;
  IBSR_IBAL = 1;       // clear the IBAL flag;
}
```

Example 17. Interrupt-Service Routine

```
__interrupt void I2C_ISR(void)
{
  if(IBSR_IAAS)
  {
    IBCR_TX_RX = IBSR_SRW;   // set Rx/Tx mode in accordance to received calling address byte
and clear the IAAS bit;
    IBSR = IBSR_IBIF_MASK;   // clear the flag;
    temp = IBDR;             // dummy read to initiate the read data byte;
    i2c_cnt = 0;             // clear the received data counter;
  }
  else
  {
    IBSR = IBSR_IBIF_MASK;   // clear the flag;
    RD_data[i2c_cnt] = IBDR; // save received data to data buffer;
    i2c_cnt++;
  }
}
```

4 Conclusion

This application note explains using the IIC module in Freescale's HC908, 9S08, and 9S12 family MCUs. The user routines are tested on the mentioned MCUs and boards. The serial EEPROMs are often used as IIC devices and serve as good examples. If you need to use something other than an IIC device (for example, a real-time clock [RTC]), change the device address (0xD0 for RTC, 0xA0 for EEPROM) and manage the right byte sequence, as described in the appropriate data sheet.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3291
Rev. 2
04/2007

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2007. All rights reserved.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

freescale™
semiconductor