*Application Note*

*AN2318/D*
*Rev. 0, 8/2002*

# *Using the I²C Bus with HCS12 Microcontrollers*

**by Grant M More**
**Applications Engineering**
**Freescale, East Kilbride**



## Introduction

The I²C bus is a simple two-wire bi-directional serial communication medium that is intended for inter-IC communication over short distances. This is typically, but not exclusively, between any number of devices on the same printed circuit board, the limiting factor being the bus capacitance.

Freescale have taken advantage of this flexible standard by including hardware support for I²C bus interfacing on most of the HCS12 range of 16-bit microcontrollers. Full details of the I²C module can be found in the I²C block user guide on the Freescale website.

## Hardware and Connections

Connecting an HCS12 device to an I²C bus is simple. Connect the SCL pin on the MCU to the serial clock line of the bus. Connect the SDA pin on the MCU to the serial data line of the bus. It is important to ensure that all devices that will use the bus to communicate are referenced to a common electrical ground. There is no requirement to terminate an I²C bus.

The I²C bus operates using the wired-AND principle. The bus lines are held in a logic 1 state (usually 5v) by obligatory pull-up resistors external to devices on the bus. The nodes can drive the bus by switching on a pull-down transistor and driving either of the bus lines to ground (logic 0). When the transistor is turned off, the bus line returns to the logic 1 state. The choice of value of the pull-up resistors depends on bus capacitive loading. The greater the bus capacitive loading, the lesser the value of the pull-up resistors must be. See **Figure 1. I²C Bus and Interface Hardware**.

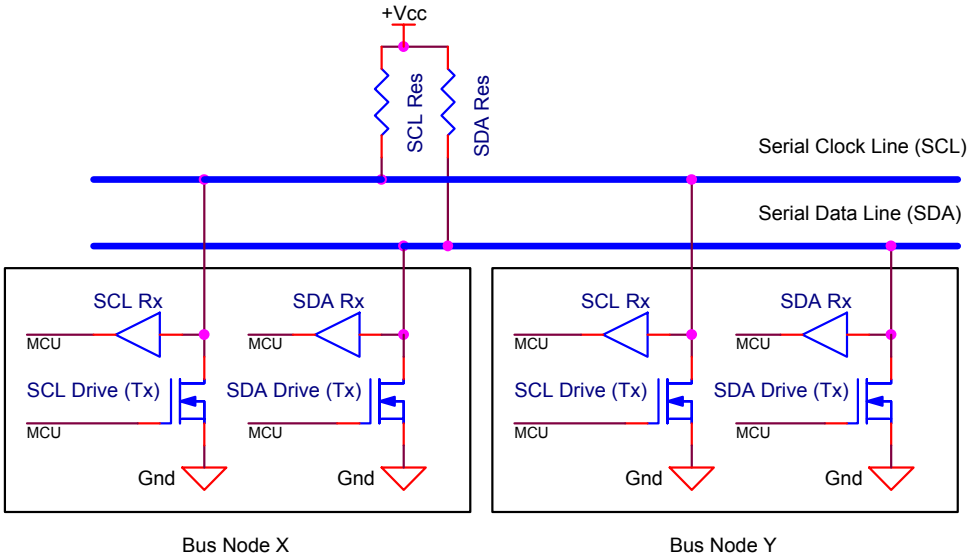


**Figure 1. I²C Bus and Interface Hardware**

The HCS12 I²C module is designed in accordance with the 100kbit/s standard-type I²C bus specification. This will allow 100kbit/s communication on the bus at the maximum capacitive bus loading of 400pF. Higher communication speeds are possible, but reduced bus loading is necessary. In this case, it is the responsibility of the user to ensure that correct pull-up resistor values and bus loading are observed.

For 100kbit/s communication on a fully loaded (400pF) bus, we recommend using pull-up resistors of no greater than 2kΩ. On a lightly loaded bus (100pF) this value can be increased to 8kΩ. Exact details of pull-up resistor values are detailed in the specification document for the I²C bus.

## Configuring the HCS12 I2C Nodes

**Node Address (IBAD Address Register)**

Each node on the I2C bus requires a unique address to allow it to be addressed as a slave device by a master device. The HCS12 MCU devices support 7-bit addressing. The slave address of the node can be set by writing the address value to the IBAD slave address register. Note that IBAD[7:1] should contain the 7-bit address. Writes to bit 0 are meaningless and will be ignored as it is reserved for future use. The 8th bit transmitted by a master in the address data is used to inform a slave of the read/write status during the impending data transfer.

**Frequency Divider (IBFD Frequency Divider Register)**

Once the I2C bus transfer rate has been chosen, the HCS12 nodes need to be configured such that they all communicate on the bus at the same clock frequency, even though the I2C modules are possibly being clocked at differing frequencies by the MCU. The bus serial clock frequency has exactly the same value as the bus transfer rate. For example, a bus with a transfer rate of 100kbit/s will have a serial clock frequency of 100kHz. This value can be used in conjunction with the frequency of the internal MCU bus clock (driving the I2C module) to generate a divider value, the SCL Divider. See **Figure 2. Equation Used to Calculate SCL Divider Value**.

SCL Divider = CPU Bus Clock Frequency ÷ IIC Bus Serial Clock Frequency

**Figure 2. Equation Used to Calculate SCL Divider Value**

Unless explicitly set by the PLL, the CPU bus clock frequency on HCS12 devices is the oscillator clock frequency divided by two. Refer to the HCS12 CRG block user guide for full details.

Once the SCL divider value has been calculated, the value for the IBFD frequency divider register can be selected from a lookup table. This is available in the I2C block user guide or in software tools soon to accompanying this documentation. It is possible to have more than one valid IBFD combination for a specific SCL divider value. The variable in this case is the SDA hold time, which is the time (in clock cycles) from the falling edge of SCL to the change in value of the SDA line. This value can be found in the lookup table. The choice of value here will depend on the other devices on the bus with respect to setup and hold times. Refer to the documentation for these devices. Greater bus loading may also affect this value in that greater loading may effectively reduce the setup and hold times as the bus takes longer to charge and discharge, hence the hold value may have to be artificially inflated. Note that the I2C bus specification dictates that the maximum hold time when using a 100kHz I2C bus is 3.45µS.

As an example, consider an HCS12 device clocked from a 16MHz crystal (no PLL in use):

- $I^2C$ Bus Communication Rate = 100kbit/s (frequency = **100kHz**)
- HCS12 Bus Clock Frequency = 16MHz ÷ 2 = **8MHz**

The SCL divider can be calculated:

- SCL Divider = 8MHz ÷ 100kHz = **80**

From the lookup table in the $I^2C$ block user guide, there are five corresponding IBFD values for this SCL divider value:

- IBFD = $14 SDA hold time = 17 clock cycles = **2.125µS** @ 8MHz (VALID)
- IBFD = $18 SDA hold time = 9 clock cycles = **1.125µS** @ 8MHz (VALID)
- IBFD = $47 SDA hold time = 20 clock cycles = **2.5µS** @ 8MHz (VALID)
- IBFD = $4B SDA hold time = 18 clock cycles = **2.25µS** @ 8MHz (VALID)
- IBFD = $80 SDA hold time = 28 clock cycles = **3.5µS** @ 8MHz (NOT VALID)

The first four are valid IBFD values as the SDA hold time is within the 3.45µS specification. When IBFD is $80, the SDA hold time will contravene this specification, thus should not be used. The choice of SDA hold time from the four remaining possibilities will then depend upon the other devices on the bus and the bus loading.

A software tool for calculating the value of IBFD will be available soon.

## Communicating Using the HCS12

### Introduction

There are a number of possible ways to communicate using the HCS12 $I^2C$ module. The most efficient method, with respect to MCU core usage time and delegation of tasks to dedicated hardware, is to use the $I^2C$ interrupt vector. This ensures that the HCS12 core is only used when data is received or has to be transmitted. The remainder of this document will concentrate on this methodology. A polling approach may be used to monitor the hardware flags in the IBSR status register, but this method is not as efficient and is not recommended.

Once the HCS12 $I^2C$ modules are configured, it is necessary to enable the module. This is accomplished by setting the IBEN bit in the IBCR register. If the interrupt driven approach is to be adopted, the IBIE interrupt enable bit in the IBCR register must also be set.

## Buffers and Packets

I$^2$C communication can be made simple with the use of transmit and receive buffers, and through the use of packet communication. This embraces the idea of organising the stream of data into packets, which are groups of data of specific length. This is shown in **Figure 3. Continuous Data Stream vs. Packet Transmission**. This demonstrates sixteen units (bytes) of data transmitted as a data stream and as four discrete four byte packets.

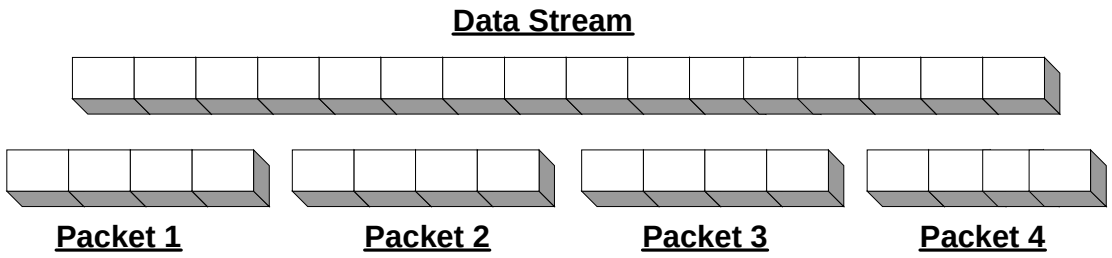


**Figure 3. Continuous Data Stream vs. Packet Transmission**

The devices on the bus must aware of how much data must be transmitted and received in each packet. The size of the packets can be transmitted actively on the bus and controlled by software, or stored by the devices as a constant. In either circumstance, the data transmitted and received is unchanged, what is varied is the size of groups in which the data is transmitted.

In the software example accompanying this documentation, packets are defined as communications that carry eight bytes of user data (not including the address of the slave). It is useful to declare a transmit and a receive data array and assigning flags and pointers to those arrays. By doing this data can be buffered and manipulated in a manageable way which complements use of the interrupt service routine. See **Figure 4. Transmit and Receive Buffer Arrangement**.

Each array consists of a number of elements, with a variable pointer (position pointer) used to fill and empty the arrays, and a fixed pointer (end pointer) that is used to denote the end of the array. Transmission and reception complete situations can be detected when the pointers both point to the same array element.

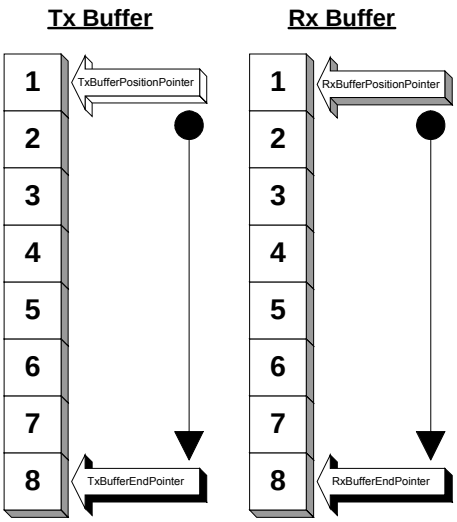


**Figure 4. Transmit and Receive Buffer Arrangement**

The buffers provide a useful interface between the bus and the software environment. When transmitting, the data to be transmitted can be loaded into the array independent of the $I^2C$ bus, and then transmitted when the array is full. The array is transmitted as a packet by the interrupt service routine, minimising bus usage and maximising efficiency. When receiving, the packet appears as a full buffer of data. A flag is raised to declare that the buffer is full and a packet of data has been received when the position pointer points to the same array element as the end pointer.

## Interrupts

The HCS12 I$^2$C module has one interrupt vector. This vector is taken when any one of three conditions occur. The conditions are:

TCF — Byte transfer complete

This flag is set and the I$^2$C interrupt routine is called when a byte of data (or an address) is successfully transferred to or from the I$^2$C module. The flag is raised after the ninth clock cycle has occurred (the acknowledge cycle). This interrupt source can be used to sequentially increment the packet position pointers and load the data to or from the IBDR data register for transfer or reception, implementing an automatic packet handling routine.

IAAS — Module addressed as slave

This flag is set and the I$^2$C interrupt routine is called when an address received on the I$^2$C bus matches the address previously written to the IBAD address register.

IBAL — Bus arbitration lost

This flag is set and the I$^2$C interrupt routine is called when a collision has been detected on the bus and the module has lost master status on the bus. This occurs when a master samples an unexpected condition on the bus which was not driven by the master's own output stage. The exact reasons for this situation occurring are detailed in the I$^2$C block user guide.

When an interrupt occurs, it is the responsibility of the software to identify the specific cause of the interrupt and take appropriate action. The I$^2$C interrupt service routine is arguably the most important part of the I$^2$C communication system on the HCS12, as the module has been designed to be driven by actions taken after interrupts occur.

The flow for the recommended interrupt service routine is shown in **Figure 5. Recommended Interrupt Service Routine Flow Diagram**.

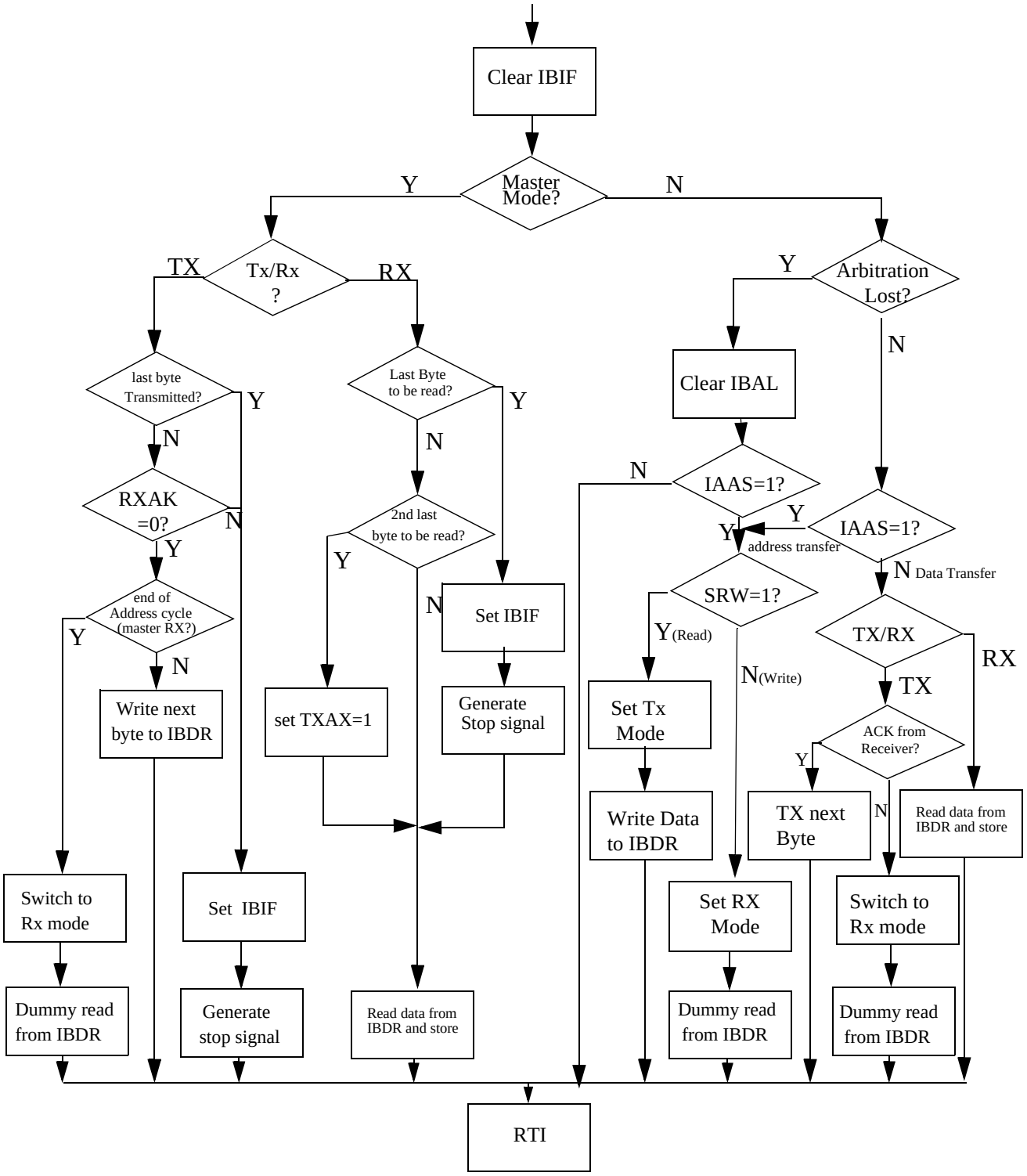


**Figure 5. Recommended Interrupt Service Routine Flow Diagram**

The majority of this flow chart can be easily translated into software with a number of if-else statements. A few global flags are required to track progress of packet transmission and reception. In the example, the flags used are as follows:

*TxCompleteflag* — This flag is set when the module is the bus master and the last byte in the transmit buffer has been transmitted. It is used to flag the ISR to ensure that a stop signal is generated before returning to non-ISR code at the end of packet transmission.

*RxCompleteflag* — This flag is similar to the TxCompleteflag except that this flag is used to signal to the ISR the fact that the bus master is finished receiving data from a slave device.

*TxBufferemptyflag* — This flag is used to signal to the calling function that the ISR has transmitted the packet on the bus and that the calling function can now stop waiting for the packet to transmit.

*RxBufferfullflag* — This flag is raised to signal to the non-interrupted code that a packet of data has been received. The flag is used to make the calling function wait until a packet of data has been received.

*MasterRxFlag* — This flag is set by the calling function to make the ISR aware that the only data to be transmitted is the address of the slave that should transmit data back to the master. This ensures that the module switches into receive mode as soon as the address is transmitted.

While the implementation of the ISR is relatively straightforward, there are a small number of areas that require a little more effort. These are detailed below.

## Transmission using the ISR

When the ISR recognises that the master device is to transmit data to a slave device, it is necessary to check the status of the transmit packet position pointer before transmitting data. By performing a comparison between the packet end pointer and the position pointer, it is possible to detect when the contents of the buffer have been transmitted. Refer to **Figure 6. ISR Master Transmit Segment**.

```
Iic.ibdr.byte = *TxPacketpositionptr;                  /* write byte to IBDR */
if (TxPacketpositionptr == TxPacketendptr)             /* if last data Tx'd */
{
TxCompleteflag = SET;                                  /* set the ISR transmit complete flag */
}
else                                                   /* there is still data to be Tx'd */
{
TxPacketpositionptr++;                                 /* move to next byte to Tx */
}
```

**Figure 6. ISR Master Transmit Segment**

The transmit complete flag is raised to signal to the ISR the fact that transmission is complete when in master mode. This allows the ISR to detect this on the final pass and generate a stop condition. The stop condition is generated by clearing the MSSL bit in the IBCR control register. A stop condition, by definition, is when the module returns to slave status by releasing the SCL line then the SDA line.  A code example of this is shown in **Figure 7. Generation of a Stop Condition**.

```
if(TxCompleteflag == SET)                              /* if last byte has been Tx'd */
{
TxCompleteflag = CLEAR;                                /* clear ISR Tx complete flag */
Iic.ibcr.byte = (IBEN|TXRX);                           /* send stop (clear MSSL) */
TxBufferemptyflag = SET;                               /* set Tx buffer empty flag */
}
```

**Figure 7. Generation of a Stop Condition**

When in slave mode, it is sufficient to raise the transmit buffer empty flag when transmission is complete because the slave device does not generate the stop condition on the bus, and thus the ISR routine will not be executed again once the last byte of data has been transmitted. Refer to **Figure 8. ISR Slave Transmit Segment**.

```
if (TxPacketpositionptr == TxPacketendptr)              /* if last data Tx'd */
{
TxBufferemptyflag = SET;                                /* set the Tx buffer empty flag */
}
```

**Figure 8. ISR Slave Transmit Segment**

## Reception using the ISR

Reception is relatively straightforward after taking a few points into consideration. Firstly, with the exception of the situation when arbitration is lost, when a module is interrupted and switched to receive mode, the contents of the IBDR data register must be read to allow the bus master to transmit the next byte. This includes the situation where a slave receives an address. In this case the IBDR should be read using a dummy variable and the contents ignored (the contents will be the address of the slave module and can be discarded).

When receiving as a master, it is important to give the slave device indication of when to stop transmitting data. This is accomplished by not acknowledging the slave when it transmits the last byte of data. An example of this is shown in **Figure 9. Instructing Slave to Stop Transmitting**. Here the active low acknowledge bit is made active high (effectively disabling it) just before the receive packet position pointer is incremented to point to the last byte.

```
if(RxPacketpositionptr == (RxPacketendptr - 1))
{
Iic.ibcr.bit.txak = SET;                                /* disable active low acknowledge */
}
```

**Figure 9. Instructing Slave to Stop Transmitting**

The receive functionality is based on the same principles as the transmit functionality, the key difference being the filling of the receive buffer rather than the emptying of the transmit buffer.

### Bus Master Mode – Transmitting to a Slave Device

To initiate communication on the $I^2C$ bus, it is necessary for the module that is initiating the communication to assert itself and become master on the bus. Before attempting to grab the bus, it is important to check that no other devices are communicating to minimise any possibility of data collision. Wait until the IBB bit in the IBSR status register has cleared before switching the module to master mode.

Master status on the bus can be accomplished by setting the MSSL bit in the IBCR register. The TXRX bit should also be set at the same time as the next thing to do is transmit the address of the slave which the master is to communicate with. This creates a start condition on the bus (SDA then SCL lines driven low by master). At this stage it is necessary to transmit the address of the slave that is to be communicated with. Once this has been accomplished, the $I^2C$ interrupt service routine will perform the necessary actions to transfer the contents of the transmit buffer. At this stage wait for the transmit buffer to empty, signalling that transfer is complete. An example of code to perform this function is shown in **Figure 10. Bus Master Transmit to Slave Function**. The transmit packet position pointer should be set to point to the first element in the array after reception in anticipation of the transmission of the next packet.

```
while(Iic.ibsr.bit.ibb == SET)                  /* while bus is busy */
{                                               /* wait until bus free */
}
Iic.ibcr.byte = (IBEN|IBIE|MSSL|TXRX);          /* grab bus */
Iic.ibdr.byte = slaveAddress;                   /* address the slave(rx) */
while(TxBufferemptyflag == CLEAR)               /* wait for tx complete */
{
}
TxPacketpositionptr = &TxPacket[0];             /* point to first element */
TxBufferemptyflag = CLEAR;                      /* clear flag */
```

**Figure 10. Bus Master Transmit to Slave Function**

### Bus Master Mode – Receiving from a Slave Device

Making a slave device transmit data requires a very similar approach to transmitting to a slave device. The key difference is the slave address. The address depends on whether the slave is to receive data from or transmit data to the master addressing it. If the slave is to transmit data, the address of the slave must be OR'ed with the value 1 to ensure that the last bit of the address indicates to the slave that it must transmit data during the impending transfer. Of course, when receiving from a slave, the destination of the data is the receive buffer, hence the flag raised when reception is complete is the receive buffer full flag. Both procedures are otherwise the same. See **Figure 11. Bus Master Receive from Slave Function**.

```
MasterRxFlag = SET;                                 /* set master rx flag */

while(Iic.ibsr.bit.ibb == SET)                      /* while bus is busy */
{                                                   /* wait until bus free */
}
Iic.ibcr.byte = (IBEN|IBIE|MSSL|TXRX);              /* grab bus */
Iic.ibdr.byte = (slaveAddress | 0x01);              /* address the slave(tx) */
while(RxBufferfullflag == CLEAR)                    /* wait for rx complete */
{
}
RxPacketpositionptr = &RxPacket[0];                 /* point to 1st element */
MasterRxFlag = CLEAR;                               /* reset master rx flag */
```

**Figure 11. Bus Master Receive from Slave Function**

### Bus Slave Mode – Receiving from a Bus Master

This mode is the default passive mode of an $I^2C$ device, the mode that the device should revert to as a slave to ensure the device can be addressed by a bus master. The majority of the data receive operation is carried out by the $I^2C$ interrupt service routine, but the buffer for receiving the data must be reset and configured to accept the incoming data. Once the buffer is prepared, wait for the packet to be received. This can be done by monitoring the state of the receive buffer full flag. When the flag is set, the packet has been received and the flag can be cleared. The data can then be extracted from the receive buffer. See **Figure 12. Slave Module Wait While Message Received Function**. Note that the MCU is held in a while loop waiting for the flag to set. This is for simplicity of this example, normally the MCU would be busy dealing with other tasks and this flag would be polled at regular intervals.

```
while(RxBufferfullflag == CLEAR)                    /* wait until packet received */
{
}
RxBufferfullflag = CLEAR;                           /* clear packet received flag */
```

**Figure 12. Slave Module Wait While Message Received Function**

**Bus Slave Mode – Transmitting to a Bus Master**

This procedure is almost the same as the receive from a bus master routine, except that data must be available in the transmit buffer for the interrupt service routine to transmit. Ensure that the transmit buffer position pointer is set to point to the first element of the transmit array before transfer begins. Wait for the transmit buffer empty flag to be set to indicate that transmission is complete.

**Loss of Bus Arbitration**

If a module loses master status on the bus, the IBAL bit in the IBSR will be set and the $I^2C$ interrupt service routine will be called. In this situation a 1 must be written to the IBAL bit to clear the flag. If the device has not been addressed as a slave (IAAS = 0) then the device is not being addressed by the master that it lost arbitration to. In this instance the module will automatically revert to slave mode and the ISR should be terminated.

Should the device find that it has lost arbitration on the bus, but that the device that it lost arbitration to addresses it, the module should behave as if the other master had addressed it as a slave. This is outlined as part of the interrupt service routine shown in **Figure 5. Recommended Interrupt Service Routine Flow Diagram**.

It is the responsibility of the software programmer to take necessary steps to prevent loss of data when loss of bus arbitration occurs. A possible solution would be for the transmitter that lost arbitration to set a data corrupted by arbitration lost flag and to attempt retransmission of the data until the flag is cleared by a successful transmit buffer empty condition. In this case the receiver would ignore incoming data until the receive buffer is full. Note that the buffer position pointers on both the transmitter and receiver must be reset each time a loss of bus arbitration occurs.

**Clock Stretching**

Slave devices can control the rate at which data to transmitted and received on the bus by clock stretching. HCS12 devices support this principle in that the SCL clock line is held low by a slave until the contents of the IBDR data register are read. The reading of this data register is controlled by software and thus it is possible, by controlling the delay between the interrupt service routine call and the read of the IBDR register, for a slave to control the transmit rate of a master device. This is of particular use when devices of different families, speeds and architectures are used on the bus. A slower device can hold off a bus master while data from a previous transmission is stored or processed.

**Repeated Start**

The HCS12 microcontroller series supports repeated start functionality. Although not explicitly demonstrated in the accompanying code, repeated start conditions can be used to replace stop conditions if the master wishes to continue transmission on the bus after it has finished communicating with a particular slave. Write a 1 to the RSTA bit in the IBCR control register to generate a repeated start condition instead of the normal clearing of MSSL to generate a stop condition.

[redacted]

### *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



freescale™ semiconductor