Freescale Semiconductor, Inc.

Engineering Bulletin

EB614/D
Rev. 1, 2/2003

*SCI Interrupt Errata
Workaround for HCS12
Family Devices*

MOTOROLA
intelligence everywhere™

digital dna™

By: Gordon Doughman
Field Applications Engineer, Software Specialist
Dayton, Ohio

Introduction

The purpose of this engineering bulletin is to document one possible workaround associated with the serial communications interface (SCI) interrupt errata (MUCts00510) in some of the currently available HCS12 Family of devices. The affected devices and associated mask sets are listed in **Table 1**.

Table 1. Devices Affected by Errata MUCts00510

Device	Mask Set
MC9S12DP256B, MC9S12DT256B, MC9S12DJ256B, MC9S12DG256B, MC9S12A256B	0K36N, 0K79X, 1K79X
MC9S12DT128B, MC9S12DB128B, MC9S12DJ128B, MC9S12DG128B, MC9S12A128B	0L85D
MC9S12H256, MC9S12H128	0K78X

NOTE: *The mask set errata for each particular HCS12 Family device should be thoroughly examined to determine whether the described errata applies to the specific device being used in a design. The mask set is identified by a five-character code consisting of a version number, a letter, two numerical digits, and a letter, for example 0F74B.*

© Motorola, Inc., 2003

Freescale Semiconductor, Inc.

Symptom

The symptom exhibited by the SCI interrupt errata is a failure of the hardware logic to assert an interrupt if an even number of unmasked SCI interrupt sources are asserted. Most often the errata manifests itself when the SCI is being used in a buffered, interrupt driven manner and the SCI transmit and receive interrupts are asserted simultaneously. For example, if the TIE and RIE bits in the SCICR2 control register are both set, enabling transmit and receive interrupts, and TDRE and RDRF bits become set simultaneously, the interrupt request to the CPU will not be asserted. This condition can lead to spurious or missing interrupts. In the case of a spurious interrupt, the request gets deasserted before the CPU can fetch the SCI vector and the SWI vector is fetched.

When an interrupt is missed, SCI interrupts will not be recognized unless and until an odd number of sources are enabled and asserted. In most cases this will happen when an additional character is received causing an SCI receiver overrun to occur resulting in the OR bit in the SCISR1 register being set. While the receiver overrun will cause SCI interrupts to once again be recognized, it results in a character being lost from the received data stream.

Problem Description

The source of this errata is caused by a logic error in the SCI interrupt structure. Within the SCI module, five sources of interrupts (RDRF, OR, IDLE, TDRE, TC) must be OR'd together to generate a single interrupt request to the CPU. The logic error results from the fact that the interrupts were 'added' together rather than OR'd. Using a one bit adder for all five sources results in an output of '1' from the adder only when an odd number of inputs are equal to one (i.e., $1 + 1 = 0$; $1 + 1 + 1 = 1$).

Workaround

While MUCts00510 states that there are no general workarounds for this errata, it does state that the occurrence can be reduced by the use of slower baud rates and ensuring that interrupt service routines execute as quickly as possible. An additional alternative might involve the use of a combination of interrupts and polling or polling alone, however, either of these two methods may tend to impose severe constraints on the real-time performance of a system. Although these suggestions may work for a limited set of applications, there are numerous situations where these restrictions cannot be tolerated.

The workaround described here utilizes one of the MCU's timer channels to generate interrupts for transmitted data while using the interrupt capability of the SCI for the reception of data. While this workaround doesn't provide 100% immunity against spurious or missed interrupts for received data, it greatly reduces the possibility of its occurrence during the normal transmission and reception of SCI data. The manner in which the receive interrupt service routine is written, there is no possibility of having both the RDRF and OR interrupts asserted simultaneously since received characters in excess of the receive buffer size are simply discarded. However, if interrupts remain masked for more than two character times, it is possible for both the RDRF and OR bits to become set, inhibiting receiver interrupts until the SCISR1 register is read, followed by a read of the SCIDRL register.

Figure 1 through **Figure 6** present the C source code for buffered, interrupt-driven SCI routines that use XOn/XOff handshaking for data flow control. Listing 1 contains the SCI initialization function InitSCI() which accepts a single unsigned long integer value representing the desired baud rate. Before initializing the SCI and timer hardware, the variables used to manage the transmit and receive queues must be initialized. Initialization of the SCI hardware is simply a matter of writing the baud rate register with the proper value, enabling the transmitter and receiver and enabling SCI receiver interrupts. Notice that the value for the baud rate register is calculated based on the value of SysClk, which is equal to the MCUs bus clock frequency.

Configuring the timer system in the SCI initialization code is not necessarily the correct location for an application using the timer for other functions, however, it helps to illustrate the required timer settings for this example. In this example, the timer system is enabled and the prescaler is configured to divide the bus clock by eight. A prescaler value of eight was chosen to allow communication rates as low as 600 baud to be supported. If other system timer functions require a different prescaler value, the range of supported baud rates will change.

Note that the TFFCA bit in the TCSR1 register is set to enable automatic output compare flag clearing in the transmit interrupt service routine. If this setting is incompatible with the requirements of other system timer functions, the transmit data interrupt service routine will have to be modified to manually clear the timer channel 1 interrupt flag.

Finally, a value is calculated for the variable OCVal. This value contains the number of timer ticks equal to one character time at the specified baud rate. The constant value of 8 in the equation is the divisor value of the timer prescaler. A change in the timer prescaler divisor ratio will require a change in this value. The value of 1 added at the end of the equation is used to compensate for integer math truncation and ensures that the output compare delay time is slightly longer than one character time. This prevents the transmit interrupt service routine from having to check the TDRE bit before writing a character to the transmit data register.

```
void InitSCI(ulong Baud)
 {

 /* Variable Declarations */

 /* Begin Function InitSCI() */

RxIn = RxOut = TxIn = TxOut = 0;    /* set the Rx & Tx queue indexes to zero */
RxBAvail = RxBufSize;               /* the receive buffer is empty */
TxBAvail = TxBufSize;               /* the transmit buffer is empty */
XOffSent = 0;
XOffRcvd = 0;

SCI0.Baud = (SysClk / 16) / Baud;   /* calculate the SCI Baud register value */
                                    /* using the system clock rate */
SCI0.CR2 = TE + RE + RIE;           /* enable both the transmitter & receiver */

TSCR1 = 0x90;                       /* enable timer system, fast flag clear option */
TSCR2 = 0x03;                       /* set timer prescaler to /8 */
TIOS |= 0x02;                       /* configure timer channel 1 as an OC */

OCVal = ((SysClk / 8) / (Baud / 10)) + 1;  /* sets the OC1 tic count (using /8 prescaler)
                                    /* to generate a 1 character time interrupt */

return;                             /* return */

 }      /* end InitSCI */
```

Figure 1 `InitSCI()` **Function**

Figure 2 contains the source code for the `putchar()` function. As the first statement of the function shows, if the transmit queue is full, the routine waits until space is available before returning. While this ensures that no data is lost from the transmitted data stream, it can have a detrimental effect on the rest of the system, especially if the host has halted the transmission of data by sending an XOff character. A careful analysis of the communications requirements of both the host and HCS12 must be performed to ensure that a deadlock situation cannot occur. If space in the transmit queue is available, the character is placed in the next available location and the queue index is updated. If the character was placed in the last location of the queue, the index is set to zero so that the next character is placed in the first location.

If an XOff has not been received from the host, the timer channel 1 interrupt is enabled, causing the character to be transmitted. Notice that if the `TDRE` bit in the SCI status register is set, indicating that the character just placed in the queue may be immediately transmitted, the timer output compare one register is written with a value that is two greater than the current value of the `TCNT` register. This causes an output compare interrupt to occur shortly after the output compare register is written. Note that the value added to the `TCNT` value, two, was chosen based on the setting of the timer prescaler. This value

provides a maximum delay of 16 bus clocks (2 * 8) and a minimum delay of eight bus clocks, which is more than enough time for the compiled code to read the TCNT register, add two to the value and write the result to the TC1 register before the value of TCNT catches up. If a smaller prescaler divisor ratio is used, this constant value will need to be increased.

Finally, if XOn/XOff handshaking support is not desired, the `if (XOffRcvd == 0)` statement and associated open and close braces may simply be removed.

```
int putchar(int c)

 {

/* Variable Declarations */

/* Begin Function putchar() */

while (TxBAvail == 0)           /* if there's no room in the xmit queue */
 ;                              /* wait here */

TxBuff[TxIn++] = c;             /* put the char in the buffer, inc buffer index */
if (TxIn == TxBufSize)          /* buffer index go past the end of the buffer? */
 TxIn = 0;                      /* yes. wrap around to the start */
TxBAvail--;                     /* one less character available in the buffer */
if (XOffRcvd == 0)              /* if an XOff has not been received... */
 {
  SEI();                        /* mask interrupts */
  if ((SCI0.SR1 & TDRE) != 0)   /* if Xmit data register is empty... */
   TC1 = TCNT + 2;              /* generate an 'immediate' TxD interrupt */
  TIE |= 0x02;                  /* enable TC1 interrupts */
  CLI();                        /* enable global interrupts */
 }

return(c);                      /* return the character we sent */

}     /* end putchar */
```

Figure 2. `putchar()` Function

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

The `getchar()` function, shown in **Figure 3**, begins by comparing the number of bytes remaining in the receive queue with the size of the queue. If the two are equal, it indicates there is no data in the receive queue. In this case, the `while()` loop is repeated until a character is received. When a character is received, the `RxBAvail` variable is decremented in the receive interrupt service routine, causing the compare to fail. At this point, the character is removed from the receive queue and placed in a local variable. Before returning the character to the calling routine, a check is performed to determine if an XOn character needs to be sent to the host. If an XOff has previously been sent and the number of characters in the receive queue is greater than or equal to `XOnCount`, the `SendXOn` flag is set to a non-zero value so that an XOn character will be sent to the host the next time a transmit interrupt occurs. Note that if the TDRE bit is set, indicating that the XOn character may immediately be transmitted, the timer output compare one register is written with a value that is two greater than the current value of the `TCNT` register. This causes an output compare interrupt to occur shortly after the output compare register is written.

One of the problems with standard `getchar()` function is the fact that if it is called with no characters in the receive queue, the function will simply wait until a character is received before returning to the calling program. For many embedded applications, this behavior would prove to be unacceptable. In an effort to enhance the usability of `getchar()` in the embedded environment, an additional routine, `SCIcharsAvail()`, is also included in **Figure 3**. This routine returns the number of characters in the receive queue by subtracting the number of available bytes in the receive queue from the receive queue size.

```
int getchar(void)
 {

 /* Variable Declarations */

 uchar c;                       /* holds the character we'll return */

 /* Begin Function getchar() */

 while (RxBAvail == RxBufSize)   /* if there's no characters in the Rx buffer */
  ;                             /* just wait here */

 c = RxBuff[RxOut++];        /* get a character from the Rx buffer & advance the Rx index */
 if (RxOut == RxBufSize)     /* index go past the physical end of the buffer? */
  RxOut = 0;                 /* yes wrap around to the start */
 RxBAvail++;                 /* 1 more byte available in the receive buffer */

 if ((XOffSent != 0) && (RxBAvail >= XOnCount)) /* OK for host to send data again? */
  {
   SEI();                       /* mask interrupts */
   SendXOn = 1;
   if ((SCI0.SR1 & TDRE) != 0)   /* if Xmit data register is empty... */
    TC1 = OTCNT + 2;            /* generate an 'immediate' TxD interrupt */
   TIE |= 0x02;                /* enable TC1 interrupts */
   CLI();                       /* enable global interrupts */
  }

 return(c);                 /* return the character retrieved from receive buffer */

 }     /* end getchar */


uint SCIcharsAvail(void)

 {

 /* Begin Function SCIcharsAvail() */

  return(RxBufSize - RxBAvail);     /* return the number of bytes in the receive buffer */

 }     /* end SCIcharsAvail */
```

Figure 3. `getchar()` and `SCIcharsAvail()` Functions

The SCI receive interrupt service routine, presented in **Figure 4**, contains the code necessary to manage the data and flow control characters received from a host computer. Notice that much of the interrupt service routine involves the management of data flow from the host. After reading the SCI status register and retrieving the byte from the data register, the character is examined to determine whether it is one of the two flow control characters. If the received byte is an XOff, timer TC1 interrupts are disabled and the variable XOffRcvd is set to a non-zero value. This prevents any additional characters from being

sent to the host computer while allowing calls to the `putchar()` function to continue to fill the transmit queue. If an XOn character is received, indicating that the host is once again able to accept data, a check is made to see whether any data remains in the transmit queue. If data is available for transmission, the TDRE bit in the SCI status register is examined and if set, the TC1 register is written with a value causing an output compare to occur shortly after the output compare register is written. After TC1 interrupts are enabled, the `XOffRcvd` variable is written to zero, allowing TC1 interrupts to be enabled each time a character is placed in the transmit queue. Notice that neither of the flow control characters is placed in the receive queue, making the XOn/XOff flow control completely transparent to the application.

To allow XOn/XOff flow control handshaking to accommodate hardware FIFOs that may be part of the host's serial communications system, an XOff character must be sent to the host while there is still enough room in the receive queue to accept data the size of the FIFO without losing any data. So, before placing the received byte into the queue, the space remaining in the queue is compared to the value `XOffCount`. If the remaining space is less than or equal to this value, the necessary actions are taken to have the XOff character transmitted to the host. Notice that two different variables, `SendXOff` and `XOffSent`, are necessary to accomplish this. The `SendXOff` variable is used to signal the transmit interrupt service routine to send an XOff to the host even if there are characters in the transmit queue waiting to be sent. The `XOffSent` variable is primarily used by the `getchar()` function to determine whether an XOn must be sent to the host. In addition, notice that the state of `XOffSent` is part of the equation used in determining whether an XOff must be sent to the host. Including this as part of the logic ensures that only a single XOff character is sent to the host.

The final action of the receive interrupt service routine is to place the received character into the receive queue. However, notice that a check is first made to ensure that there is room in the queue. If the queue is full, the character is simply discarded.

The concluding piece of code, presented in **Figure 5**, is the transmit interrupt service routine. Like the receive interrupt service routine, much of the code is dedicated to management of the XOn/XOff flow control. After reading the SCI status register, a check is made to determine whether a request is pending to send an XOn or XOff character to the host system. If neither request is pending, a character is removed from the transmit queue and written to the SCI data register. The final action performed by the transmit interrupt service routine is to configure timer channel 1 to generate an output compare in one character time. Notice that the timer is setup to generate the output compare regardless of whether any additional characters remain in the transmit queue. If the transmit queue is empty, TC1 interrupts are disabled, preventing the output compare from generating an interrupt.

Freescale Semiconductor, Inc.

```
ISR RxISR(void)
 {

 /* Variable Declarations */

 uchar c;

 /* Begin Function RxISR() */

 /* code to check for receive data errors (NF, OR, etc.) can be inserted here */

 c = SCI0.SR1;          /* read the SR so RDRF will clear when we read DRL */
 c = SCI0.DRL;          /* get the received character */

 if (c == XOff)         /* host want us to stop sending data? */
  {
   TIE &= ~0x02;        /* yes. disable transmit interrupts */
   XOffRcvd = 1;        /* continue to place chars in xmit queue but don't send them */
   return;
  }
 else if (c == XOn)     /* host want us to start sending data? */
  {
   if (TxBAvail != TxBufSize)    /* anything left in the Tx buffer? */
    {
     if ((SCI0.SR1 & TDRE) != 0) /* if Xmit data register is empty... */
      TC1 = TCNT + 2;            /*  generate an 'immediate' TxD interrupt */
      TIE |= 0x02;              /* enable TC1 interrupts */
    }
   XOffRcvd = 0;
   return;
  }

 if ((RxBAvail <= XOffCount) && (XOffSent == 0))     /* send an XOff? */
  {
   SendXOff = 1;              /* set flag so Tx ISR sends an XOff */
   XOffSent = 1;              /* set the flag showing that an XOff's been sent */
   if ((SCI0.SR1 & TDRE) != 0) /* if Xmit data register is empty... */
    TC1 = TCNT + 2;            /* generate an 'immediate' TxD interrupt */
   TIE |= 0x02;              /* enable TC1 interrupts */
  }

 if (RxBAvail != 0)             /* if there are bytes available in the Rx buffer */
  {
   RxBAvail--;                 /* reduce the count by 1 */
   RxBuff[RxIn++] = c;         /* place the received byte in the buffer */
   if (RxIn == RxBufSize)      /* reached the physical end of the buffer? */
    RxIn = 0;                  /* yes. wrap around to the start */
  }

 }    /* end RxISR */
```

Figure 4. SCI Receive Interrupt Service routine

```
ISR void TxISR(void)
 {

  /* Variable Declarations */

  uchar c;

  /* Begin Function TxISR() */

  c = SCI0.SR1;               /* read status register so the Xmit hardware works properly */

  if (SendXOn != 0)          /* request to send an XOn to the host? */
   {
    SendXOn = 0;
    XOffSent = 0;                  /* reset the XOff flag */
    SCI0.DRL = XOn;               /* send the character */
    if ((TxBAvail == TxBufSize) || (XOffRcvd != 0))  /* Tx buffer empty, */
                                            /* or host sent an XOFF? */
     TIE &= ~0x02;               /* yes. disable transmit interrupts */
   }
  else if (SendXOff != 0)    /* request to send an XOff to the host? */
   {
    SendXOff = 0;                  /* yes, clear the request */
    SCI0.DRL = XOff;              /* send the character */
    if ((TxBAvail == TxBufSize) || (XOffRcvd != 0))  /* Tx buffer empty, */
                                            /* or host sent an XOFF? */
     TIE &= ~0x02;               /* yes. disable transmit interrupts */
   }
  else
   {
    SCI0.DRL = TxBuff[TxOut++];   /* remove a byte from the buffer & send it */
    if (TxOut == TxBufSize)       /* reached the physical end of the buffer? */
     TxOut = 0;                   /* yes. wrap around to the start */
    if (++TxBAvail == TxBufSize)  /* anything left in the Tx buffer? */
     TIE &= ~0x02;               /* no. disable transmit interrupts */
   }

  TC1 = TCNT + OCVal;            /* character delay to generate the next TxD interrupt */

 }     /* end TxISR */
```

Figure 5. SCI Transmit Interrupt Service Routine

```
#include <ctype.h>
#include "DP256Regs.h"

#define uchar unsigned char
#define ulong unsigned long

#define TDRE 0x80
#define RDRF 0x20
#define RIE 0x20
#define TC 0x40
#define TE 0x08
#define RE 0x04
#define OR 0x08

#define XOn 0x11
#define XOff 0x13

#define RxBufSize 64
#define TxBufSize 32

#define XOnCount RxBufSize - 8
#define XOffCount 18

#define SEI() _asm(" sei")
#define CLI() _asm(" cli")

uint OCVal;                     /* Output Compare timer value for 1 character time */

uchar RxBuff[RxBufSize];      /* receive queue */
uchar TxBuff[TxBufSize];      /* transmit queue */
uchar RxIn;                     /* next available location in the Rx queue */
uchar RxOut;                    /* next character to be removed from the Rx queue */
uchar TxIn;                     /* next available location in the Tx queue */
uchar TxOut;                    /* next character to be sent from the Tx queue */
volatile uchar RxBAvail;      /* number of bytes left in the Rx queue */
volatile uchar TxBAvail;      /* number of bytes left in the Tx queue */
uchar XOffSent;                 /* !=0, an XOff was sent to the host */
uchar XOffRcvd;                 /* !=0, an XOff was received */
uchar SendXOff;                 /* !=0, TxISR should send XOff the next time it's called */
uchar SendXOn;                  /* !=0, TxISR should send XOn the next time it's called */
```

Figure 6. Includes, Defines and Variables

Freescale Semiconductor, Inc.

EB614/D

For More Information On This Product,
Go to: www.freescale.com