MOTOROLA
SEMICONDUCTOR
APPLICATION NOTE

Order this document
by AN1724/D

Implementing SCI Receive and Transmit Buffers in C

by Mark Maiolani, Motorola, East Kilbride

1 Introduction

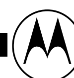
Several devices in the Motorola M68HC16 and M68300 microcontroller families include asynchronous serial communications channels. On the MC68331 device, this functionality is included in the serial communications interface (SCI) part of the queued serial module (QSM).

Although the SCI only buffers a single transmission or reception of eight or nine bits, typical applications require data to be transmitted and received in multiple byte ‘packets’, such as text strings. If the main CPU software directly passes data to and from the SCI, then it may have to wait for SCI transmissions or reception to complete before it can write the next byte for transmission or read the received byte. This is not efficient use of the CPU, which could be processing other tasks.

One way to resolve this problem is to implement software buffers for the SCI transmitter and receiver which service the SCI module via interrupt. To transmit a packet of data, the main program would simply add the data to the transmit buffer, and continue processing without having to wait for actual transmission. Similarly, a packet of received data could be read from the receive buffer and processed at a time convenient for the CPU.

This application note describes one implementation of SCI buffer software with the following characteristics:

- Configured for CPU32 devices with QSM (including MC68331, MC68332, MC68F333 and MC68336)
- Written in C language to simplify porting to CPU16 devices with QSM
- Simple user interface
- Circular transmit buffer
- Circular receive buffer
- All SCI maintenance carried out by interrupt
- Buffer sizes easily adjusted
- SCI interrupts can remain enabled during buffer reads/writes


MOTOROLA

© MOTOROLA INC, 1997

2 The SCI Buffer Software

The SCI buffer software consists of one main source file (SCIBUFF.C), and a single include file of defines for the QSM module (QSM.H). All of the buffer operations are built into six support functions, with a main() function included only as an example of the initialization sequence and calling conventions for these functions.

The following support functions are included in the software:

- sciinit (void)
- qinit (queue_struct *sptr)
- qstat (queue_struct *sptr)
- rx_byte (byte *b)
- tx_byte (byte)
- sciint (void)

3 SCI Initialization

Sciinit performs initialization of the SCI hardware and unlike the other support functions, does not access the main transmit and receive buffers created by the software. Any changes to the SCI baud rate or other SCI settings can be made by modifying this function. The function is called without any passed or returned parameters as shown below.

```
sciinit();                       /* Initialize the SCI in the QSM module */
```

4 Main Library Functions

The functions qinit, qstat, rx_byte, and tx_byte are called by the main user software as library type functions.

Qinit is used to initialize a specified buffer to its empty state, either as part of the main initialization sequence, or later during operation to perform a ‘flush buffer’ function. The buffer is specified by passing its address to the function, e.g.:

```
qinit(&rxbuff);                                  /* Initialize RX buffer */
```

Qstat returns the number of valid data entries in a specified buffer. An example use of this function is if the user program is required to process received data in packets of n entries. Qstat would be periodically called to check if the required n entries are available, before continuing to read and process the data. The qstat function can also be used to provide an early warning that a buffer is almost full before it reaches an overrun condition. As with qinit, the buffer is specified by passing its address.

Rx_byte is used to read a byte of data from the SCI receive buffer, and update the buffer pointers. The function return value should be monitored to confirm that a byte was available and has been successfully read. A returned value of zero indicates that no data was read. Typical usage of rx_byte is shown below.

```
if (rx_byte(&rxchar))
   { /* Data read into rxchar successfully */ }
else
   { /* Data read unsuccessful */ }
```

Tx_byte is used to write a byte to the SCI transmit buffer. The function also ensures that the SCI transmit interrupt is enabled, as the interrupt is automatically disabled by function sciint when all transmit buffer data has been transmitted. As with rx_byte, a non-zero return value indicates successful operation, with the passed data byte added to the buffer. The example code below continually re-tries to write data to the transmit buffer until tx_byte flags a successful operation.

```
while (!tx_byte(rxchar));           /* Try to transmit until successful */
```

5 The SCI Interrupt Handler

Sciint is the SCI interrupt handler for both transmit and receive interrupts and should never be conventionally called from software. The two main SCI interrupt sources for the function are receive data register full (RDRF), which indicates that a byte of data has been received and may be read from the SCI, and transmit data register empty (TDRE), indicating that a new byte of data may be written to the SCI for transmission. The third interrupt source is the receiver overrun flag (OR), which indicates an error in that data is being received faster than it can be processed by the interrupt handler.

6 Data Objects

Defines for the data sizes byte, word and lword are used throughout the program when accessing 8-, 16- and 32-bit values. These defines are generally used when accessing data objects such as hardware registers whose primary characteristic is their size and the use of any C mathematical type is inappropriate.

```
typedef unsigned char byte;    /* Define byte, word, and long word types */
typedef unsigned short int word;
typedef unsigned long lword;
```

The struct type queue_struct is used for both receive and transmit buffers. This struct includes the buffer itself, together with the pointers and full flag required for buffer maintenance. The structure typedef is shown below:

```
/*     Typedef for tx and rx queue structures */
typedef struct {
       word in;                          /* Queue input/write pointer */
       word full;                                  /* Queue full flag */
       word out;                         /* Queue output/read pointer */
       byte q[qsize];                                  /* Queue array */
       } queue_struct;
```

The byte array q[qsize] forms the actual buffer. As supplied, the code supports 8-bit SCI transfers. If 9-bit SCI transfers were to be buffered, the q[qsize] array could be defined as a word array, although using a 16-bit word to store 9-bit data is an inefficient use of data RAM. In the case of using the ninth SCI bit for parity checking or address selection for received messages, it may be more efficient to decode and generate the extra bit in the sciint routine, and keep the buffers eight bits wide.

The two queue pointers are used to determine where buffer data should be written to or read from, and how much of the buffer space is used or empty. The full flag is required to differentiate between buffer completely empty and completely full conditions. The functional definitions of these variables are as follows:

Structure element .in is the pointer to the next free buffer entry which can be written. It is used as an index to the q[] array.

Element .full is a flag which is used to indicate whether the buffer is completely empty or completely full whenever the buffer input pointer equals the output. The flag has no meaning whenever the buffer pointers are not equal. This is shown in **Table 1**, along with the two main conditional tests which have to be evaluated in the software when reading from and writing to the buffers.

Table 1 Buffer Conditions

Buffer Variables		Notes	Conditional Tests	
Buffer.in == Buffer.out	**Buffer.full**		**Buffer Data Available**	**Buffer Not Full**
0	x	Buffer partially used	Yes	Yes
1	0	Buffer empty	No	Yes
1	Non-zero	Buffer full	Yes	No

Element .out is the pointer to the next valid buffer entry to be read. As with .in it is used to index the q[] array.

Figure 1 shows the state of the buffer variables in four different conditions, from the buffer being completely empty through to the buffer being completely full.

Figure 1-A shows the buffer in its empty, initialized state. In this case, none of the q[] array entries hold valid data, and the in pointer equals the out pointer, with full == zero.

Figure 1-B shows the buffer with a single entry used. This could be the case of a receive buffer after an SCI byte has been received and before it has been read by the user program, or a transmit buffer after the user program has added a byte to the buffer and before it has been passed to the SCI for transmission. The in pointer is incremented to point to the next free buffer location, and the out pointer is left pointing to the valid data in the buffer. As the in pointer does not equal the out pointer, the full variable has a ‘don’t care’ value.

Figure 1-C shows the buffer filled to the stage where there is only a single free entry remaining. Buffer data has been added causing the .IN pointer to wrap past the last entry to the start of the buffer, and increment until there is a single buffer entry left unused.

Figure 1-D shows the buffer completely full, with every buffer entry holding data and no free entries left to accept new data. The buffer variables are the same for this case as 1-A (buffer empty), except that the .FULL element is non-zero.

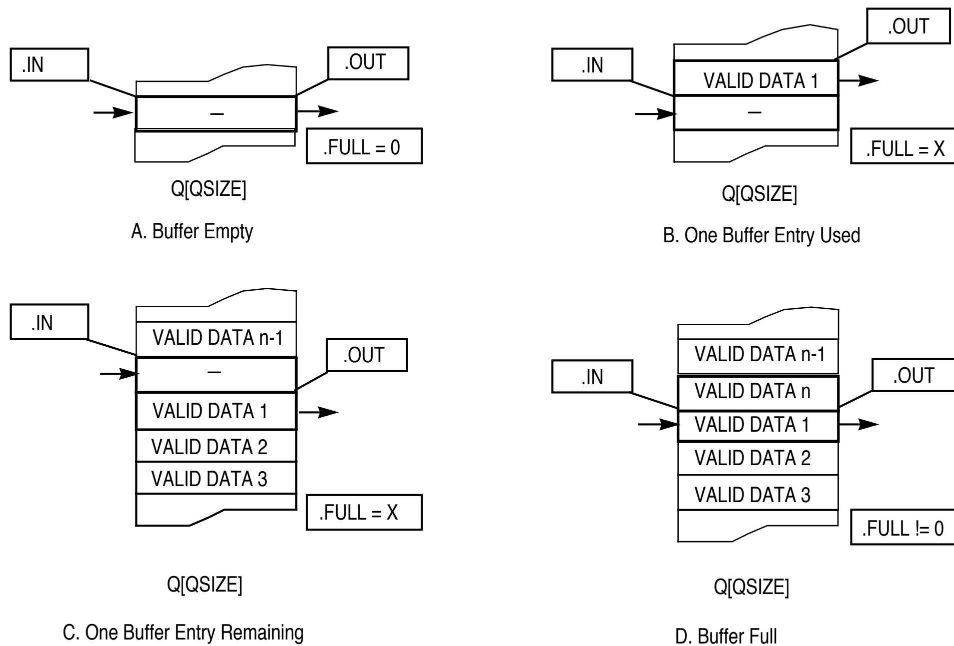


Figure 1 Buffer Use Examples

7 Reading Buffer Data

The most complex task performed by the software involves reading data from, and writing data to the receive and transmit buffers. As identical buffer structures are used for both receive and transmit, the code sequence to access them is the same in both cases.

Data is read from a buffer when user software reads received data from a buffer using rx_byte, or when data is read from the transmit buffer and passed to the SCI module by the sciint function in response to the SCI TDRE interrupt. The task can be described by the following high level pseudo-code.

```
If (Buffer data is available)
   {
   Read the byte from Buffer.Q
   Update Buffer.OUT and clear Buffer.FULL in an indivisible operation
   Return the byte read
   }
```

As can be seen from **Table 1**, the condition (buffer data is available) can be broken down into ((buffer partially used) OR (buffer full)). This is coded as:

```
if ((Buffer.IN!=Buffer.OUT) || Buffer.FULL)
```

As long as only a single function call is used to read from a particular buffer (e.g. rx_byte for the receive buffer, and sciint for the transmit buffer), then it can be guaranteed that if the condition is evaluated as true, it will remain true even if other functions interrupt to write to the buffer.

8 Writing Buffer Data

Data is written to a buffer when user writes data to a transmit buffer using tx_byte, or when SCI receive data is written to the receive buffer by the sciint function in response to the SCI RDRF interrupt. The task can be described by the following high level pseudo-code.

```
 If (Buffer Not Full)
   {
   Write the byte to the Buffer.Q
   If (Buffer now full)
            {
            Update  Buffer.IN  and  set  Buffer.FULL  in  an  indivisible
operation
            }
   else
            {
            Update Buffer.IN only
            }
   }
```

As can be seen from **Table 1**, the condition (Buffer Not Full) can be broken down into ((Buffer Partially Used) OR (Buffer Empty)). This is coded as:

```
if ((Buffer.IN!=Buffer.OUT) || !Buffer.FULL)
```

As long as only a single function call is used to write to a particular buffer (e.g. tx_byte for the transmit buffer, and sciint for the receive buffer), then it can be guaranteed that if the condition is evaluated as true, it will remain true even if other functions interrupt to read from the buffer.

9 Coherency Requirements

To ensure coherency, certain operations must be implemented as single indivisible CPU32 instructions so that they cannot be interrupted when partially completed. An example from the read buffer data pseudo-code is:

```
Update Buffer.OUT and clear Buffer.FULL in an indivisible operation
```

Implementing these indivisible operations requires knowledge of the assembly code and variable assignments used by the C compiler. In this SCI buffer software, the buffer structures are defined so that the buffer variables in, full and out are arranged as consecutive words in memory. The CPU32 core can then perform a 32-bit access to read or write the two variables in and full, or full and out, in a single indivisible operation. The following C code shows a write operation to buffer variables full and out.

```
Buffer_full_out = (lword)rxout;           /* prepare for coherent write..*/
*(lword *)(&Buffer.full) = Buffer_full_out;/* of rxqout with rxqfull zero
*/
```

The resultant assembly code enters with D1 holding rxout and A0 holding the address of Buffer.full. Register D0 holds the temporary storage variable, Buffer_full_out.

```
CLR.L       D0
MOVE.W      D1,D0
LEA.L       (2,A5),A0
MOVE.L      D0,(A0)
```

The requirement for indivisible operations can be seen when reading the last available data entry from a buffer (**Figure 1-B**). If the operation was divisible and another function interrupted after Buffer.OUT had been incremented but before Buffer.FULL had been cleared, it would read a condition where ((Buffer.OUT == Buffer.IN) AND (Buffer.FULL !=0)). This would indicate that the buffer was completely full rather than completely empty, and may result in errors such as received data being lost due to an incorrectly sensed data overrun condition.

In general, the buffer software has been written to allow a buffer write operation to interrupt a buffer read operation, and vice versa. This allows the SCI interrupts to be assigned any required priority level and remain enabled at all times.

The only user restriction is that a read operation should not interrupt another read operation, and a write operation should not interrupt another write operation. Supporting this operation would require additional code and semaphore flags so that the first routine to access the buffer would ‘lock out’ subsequent attempted accesses until the first routine completes and releases the buffer.

No code modifications are required if all user buffer reads (i.e. calling the rx_byte) occur in main level code or at the same interrupt level, and similarly all buffer writes (i.e. calls to tx_byte) occur in main level code or at the same interrupt level.

10 Restrictions in Accessing Data Objects

In order to ensure correct operation of the SCIBUFF software under all conditions, the types of accesses that the various functions can make to the buffer variables have to be fully defined. Access restrictions are required to guarantee that if one function which uses buffer accesses interrupts a second function during its own buffer access, no incorrect operation will occur.

Table 2 Data Access Types by Function

Data Element	Function				
	qinit (&rxbuff)	rx_byte	sciint	tx_byte	qinit (&txbuff)
rxbuff.in	write	read	read / ++	—	—
rxbuff.full	write	read / clear	read / set	—	—
rxbuff.out	write	read / ++	read	—	—
rxbuff.q[]	—	read	write	—	—
txbuff.in	—	—	read	read / ++	write
txbuff.full	—	—	read / clear	read / set	write
txbuff.out	—	—	read / ++	read	write
txbuff.q[]	—	—	read	write	—

11 Further Enhancements

The full source code includes detailed comments to allow further user modification. One potential user addition would be extra support for error conditions. As supplied, the software will detect all overrun conditions (SCI hardware overrun, software receive buffer overrun, and software transmit buffer overrun), but no recovery action is taken. Any recovery is dependent on the overall application and must therefore be added by the use.

12 Code Listings

12.1 SCI Buffer Software Listing

```
/*
   SCIBUFF.C
   Interrupt driven C language routines to implement circular receive and
   transmit buffers for SCI channel of QSM. As 100% C code is used, the rx
   and tx buffers can be easily re-sized, or directly accessed.

   Platform : MC68332 or similar CPU32 + QSM micrcontroller
   Compiler : HiWare CM68K CPU32 C Compiler ver 2.6
   Author   : Mark Maiolani, Motorola EKB, April 96

   Revisions: 13/11/95  - Initial creation from CPLOT.C routines
              04/04/96  - Structs used for rx and tx queues
              05/04/96  - SCI interrupt handler added
              10/04/96  - rx_stat function added, initial test of rx functions
                          queue initialization and status functions generalised
                          to access a specified structure
              11/04/96  - SCI tdre interrupt handler added
                          basic rx and tx functionality tested
              17/04/96  - Comment fixes only
              16/07/96  - SCI receiver OR handlers added
                          Comments updated

*/

/*    Typedefs */
typedef unsigned char byte;                 /* Define byte, word, and lon word types */
typedef unsigned short int word;
typedef unsigned long lword;

/*    Include files */
#include "qsm.h"
```

```
/*    Defines */
#define vbr 0x000      /* Vector Base Register value */
#define scivec 0x56    /* SCI receive vector number  */
#define qsize  160     /* Queue size - bytes */

/*      Typedef for tx and rx queue structures */
typedef struct {
      word in;          /* Queue input/write pointer */
      word full;        /* Queue full flag */
      word out;         /* Queue output/read pointer */
      byte q[qsize];    /* Queue array */
      } queue_struct;

/*    Global Variables */
queue_struct rxbuff, txbuff;       /* Transmit and receive queue structures */

/*    Function prototypes */
void       qinit      (queue_struct *sptr);/* Initialize a specified buffer */
word       qstat      (queue_struct *sptr);/* Return the status of a specified buffer */
void       sciinit    (void);          /* SCI module initialization */
void       sciint     (void);          /* SCI interrupt handler (TX and RX) */
char       rx_byte    (byte *b);       /* Read a byte from the RX buffer */
char       tx_byte    (byte);          /* Write a byte to the TX buffer */

/* Main initialization and usage: */
main()
        {
      byte rxchar, tempbyte;
      word tempw;

      qinit(&rxbuff);  /* Initialize RX buffer */
      qinit(&txbuff);  /* Initialize TX buffer */

      /* Set up SCI interrupt vectors for SCI exception handler sciint() */
      *(long *)((scivec * 4) + vbr) = (long)sciint;

      sciinit();       /* Initialize the SCI in the QSM module */

      asm{
            MOVE.W  #$2500,SR; enable CPU interrupts / level 6+7
      }

      while (1)
            {
            if ((qstat(&rxbuff))>(qsize/2))    /* Wait till rx queue 1/2 full */
                  {
                  while (rx_byte(&rxchar))     /* Then tx data until rx empty */
                        {
                        while (!tx_byte(rxchar));/* then tx until successful */
                        }
                  }
            }
      }

char  rx_byte(byte *rxbyte)
      {
      /* Read a received byte from the RX buffer */
      /* Return 0 if no byte available, or non zero if byte read OK */
```

```
	lword rxq_full_out;
	word rxin, rxfull, rxout;

	/* Read rx buffer variables into locals */
	rxin   = rxbuff.in;
	rxfull = rxbuff.full;
	rxout  = rxbuff.out;

	if ((rxin!=rxout)||rxfull)         /* IF (rx data avaialable) */
		{
		*rxbyte = rxbuff.q[rxout];   /* read data byte */
		rxout++;                     /* update pointer..*/
		if (rxout>(qsize-1)) rxout = 0;/* check for wraparound */
		rxq_full_out = (lword)rxout; /* prepare for coherent write..*/
		*(lword *)(&rxbuff.full) = rxq_full_out; /* of rxqout with rxqfull zero */
		return 1;                    /* return 1 for successful read */
		}
	else
		{
		return 0;                    /* return 0 as no byte available */
		}
	}

void  qinit (queue_struct *qvar)
	/* Initialize specified queue */
	{
	qvar->in   = 0x0002;  /* Set in=out, and full=0, ie. buffer empty */
	qvar->full = 0x0000;
	qvar->out  = 0x0002;
	}

word  qstat (queue_struct *qvar)
	/* Return the number of valid bytes in a specified buffer */
	{
	word qin, qfull, qout;

	/* Read q buffer variables into locals */
	qin   = qvar->in;
	qfull = qvar->full;
	qout  = qvar->out;

	if (qin>qout)
		{
		/* Data in buffer, no pointer wrap */
		return (qin-qout);
		}
	if (qin<qout)
		{
		/* Data in buffer with pointer wrap */
		return (qsize-qout+qin);
		}
	/* qin must be = qout */
	if (qfull)
		{
		/* Buffer full */
		return (qsize);
		}
	/* Buffer empty */
	return (0);
	}
```

```
char  tx_byte(byte txbyte)
      {
      /* Queue a byte into the TX buffer for transmission */
      /* Return 0 if buffer already full, or non zero if byte queued OK */

      word txin, txfull, txout;
      lword txq_in_full;

      /* Read rx buffer variables into locals */
      txin   = txbuff.in;
      txfull = txbuff.full;
      txout  = txbuff.out;

      if ((txin!=txout)||!txfull)  /* IF (tx buffer not full) */

            /* Add data to tx buffer as space is available */
            {

            txbuff.q[txin] = txbyte;
            txin++;
            if (txin>(qsize-1)) txin = 0; /* check for wraparound */

            if (txin==txout)
            /* Buffer is now full, need to set FULL flag coherently with update of txin */
                  /* Also generate warning that buffer is now full */
                  {
                  txq_in_full = (((lword)txout)<<16)|1;  /* prepare for coherent write..*/
                  *(lword *)(&txbuff.in) = txq_in_full;/* of txin with txqfull non-zero */
                  }
            else
                  {
                  /* Buffer not full yet, so don't set FULL */
                  txbuff.in = txin;
                  }
            /* Enable transmitter interrupt (TIE bit set) */
            QSM_SCCR1 = 0x00ac;
            return (1);
            }
      else
            /* Return error code as no buffer space to queue data */
            {
            return (0);
            }
      }

void  sciinit()
      /* Power-up initialization of SCI */
      {
      /* IARB=7 supv access only */
      QSM_MCR = 0x0087;
      /* SCI interrupt at level 6 with vector number (scivec) */
      QSM_QILR_QIVR = scivec + 0x0600;
      /* Select baud rate using:
            Baud rate = system clock / (32 * SCCR0 value) */
      /* QSM_SCCR0 = 0x0037; *//* 9600 baud @ 16.778 MHz */
      QSM_SCCR0 = 0x0034;    /* 9600 baud @ 16. MHz    */

      /* Enable receiver interrupt (RIE bit set)
            This enables interrupts generated by RDRF and/or OR flags */
```

```
      QSM_SCCR1 = 0x002c;
      }

#pragma TRAP_PROC
      /* pragma required as sciint is an exception handler */
void  sciint()
      /* SCI RX interrupt handler */
      /* Note : The SCI interrupt handler will only clear a single
            interrupt source each time it is executed, by using an else-if structure.
            This ensures that the transmitter routine is not called  unnecessarily
            if the SCI transmitter is idle (TDRE set) but with no transmit buffer
            data pending, and the interrupt source therefore disabled */
      {
      word status, qin, qfull, qout;
      word scidata;
      lword q_long;

      /* Read status reg to determine source of exception */
      status=QSM_SCSR;

      if (status & 0x0040)   /* if RDRF (rx data register full) flag set */
            {

            scidata = QSM_SCDR;/* Read received data (+ clear RDRF) */

            if (status & 0x0008)/* if OR (receiver OverRun) flag set */

            /* Note : typically RDRF will be set at the same time as OR, in which case
            the RDRF handler will clear OR.
            OR may be set on its own if an overrun is generated in between QSM_SCSR being
            read with RDRF set (normal receive), and RDRF being cleared by reading SCDR.
            Because of this possibility, we need an independent OR flag test and
            clear mechanism */

                  {
                  /* Any additional receiver overrun handling should be added here
                        and in the separate OR handler. */
                  }

            else
                  {

                  /* Note : This s/w flow discards any received data which has caused
                  a receiver overrun, as it may be regarded as a non-recoverable error.
                  The s/w could be modofied by removing the previous 'else' statement
                  so that no data is discarded, although this may result in missing data
                  from the receive buffer when overrun occurs. */

                  /* Read rx buffer variables into locals */
                  qin   = rxbuff.in;
                  qfull = rxbuff.full;
                  qout  = rxbuff.out;

                  if ((qin!=qout)||!qfull)   /* IF (rx buffer not full) */
                        /* data received, but OK. as space in rx buffer */
                        {
                        rxbuff.q[qin] = (byte)scidata;
                        qin++;
                        if (qin>(qsize-1)) qin = 0; /* check for wraparound */
```

```
                if (qin==qout)
                /* Buffer now full, so set FULL flag coherently with update of qin */
                      /* Also generate warning that buffer is now full */
                      {
                      q_long = (((lword)qout)<<16)|1;/* prepare for coherent write..*/
                      *(lword *)(&rxbuff.in) = q_long;/*of qin with rxqfull non-zero*/
                      }
                else
                      {
                      /* Buffer not full yet, so don't set FULL */
                      rxbuff.in = qin;
                      }
                }
          else
                /* Data received, but buffer full (buffer overrun) so flag error */
                /* Any additional handler for buffer overrun should be added here */
                {
                }
          }
    }

else if (status & 0x0008)    /* if OR (receiver OverRun) flag set */

/* Note : typically RDRF will be set at the same time as OR, in which case
the RDRF handler will clear OR.
OR may be set on its own if an overrun is generated in between QSM_SCSR being
read with RDRF set (normal receive), and RDRF being cleared by reading SCDR.
Because of this possibility, we need an independent OR flag test and
clear mechanism */

    {

    scidata = QSM_SCDR;    /* Read received data to clear OR */

    /* Any additional receiver overrun handling should be
    added here and in the RDRF + OR handler */

    }

else if (status & 0x0100)    /* if TDRE (tx data register empty) flag set */
    {
    qin   = txbuff.in;
    qfull = txbuff.full;
    qout  = txbuff.out;

    if ((qin!=qout)||qfull)      /* IF (tx data avaialable) */
          {
          scidata = txbuff.q[qout];    /* read data byte */
          qout++;                      /* update pointer..*/
          if (qout>(qsize-1)) qout = 0;/* check for wraparound */
          q_long = (lword)qout;        /* prepare for coherent write..*/
          *(lword *)(&txbuff.full) = q_long; /* of txqout with txqfull zero */

          QSM_SCDR = scidata;    /* Write tx data to SCI (+ clear TDRE)*/
          }
    else              /* no more tx data pending.. */
          {
          /* ..so disable transmitter interrupt (clear TIE bit) */
          QSM_SCCR1 = 0x002c;
          }
```

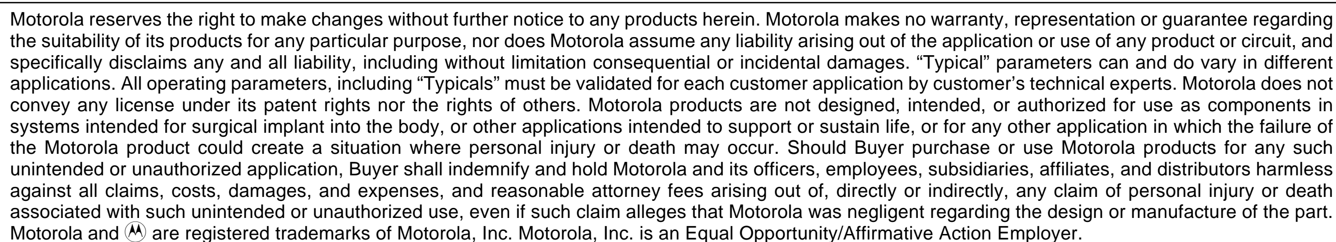
```
        }

    else                    /* some other interrupt source */

        {
        sciinit();          /* something is wrong - re-initialize sci */
        }
    }
```

12.2 QSM Software Listing

```
/*************************************************************
**
**   Definitions for Queued Serial Module (QSM)
**   MM EKB
**   5/4/96
**
*************************************************************/

#define QSM_BASE  (0xfffffc00L)   /* QSM Base Address*/
/* Adjust this address to suit device in use. Note that
if the address is at the top of memory, all 32 bits should
be defined, even though the IMB devices do not require all
address bits. Specifying all 32 bits allows the compiler to
use the more efficient word addressing mode if possible */


#define QSM_MCR         (* (word *)  (QSM_BASE))
#define QSM_TEST        (* (word *)  (QSM_BASE+0x2))
#define QSM_QILR_QIVR   (* (word *)  (QSM_BASE+0x4))
#define QSM_SCCR0       (* (word *)  (QSM_BASE+0x8))
#define QSM_SCCR1       (* (word *)  (QSM_BASE+0xA))
#define QSM_SCSR        (* (word *)  (QSM_BASE+0xC))
#define QSM_SCDR        (* (word *)  (QSM_BASE+0xE))
#define QSM_PORTQS      (* (byte *)  (QSM_BASE+0x15))
#define QSM_PQSPAR_DDR  (* (word *)  (QSM_BASE+0x16))
#define QSM_SRCR0       (* (word *)  (QSM_BASE+0x18))
#define QSM_SPCR1       (* (word *)  (QSM_BASE+0x1A))
#define QSM_SPCR2       (* (word *)  (QSM_BASE+0x1C))
#define QSM_SPCR3_SPSR  (* (word *)  (QSM_BASE+0x1E))

#define QSM_RXRAM       (QSM_BASE+0x100))
#define QSM_TXRAM       (QSM_BASE+0x120))
#define QSM_CMDRAM      (QSM_BASE+0x140))
```

NOTES

NOTES


Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola 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 can and do vary in different applications. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola 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 Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola 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 Motorola was negligent regarding the design or manufacture of the part. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

How to reach us:

USA/EUROPE/Locations Not Listed: Motorola Literature Distribution;
P.O. Box 5405, Denver Colorado 80217. 1-800-441-2447, (303) 675-2140
Mfax™: RMFAX0@email.sps.mot.com - TOUCHTONE (602) 244-6609, U.S. and Canada Only 1-800-774-1848
INTERNET: http://Design-NET.com
JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC,
6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 81-3-3521-8315
ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,
51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298
Mfax is a trademark of Motorola, Inc.

MOTOROLA

AN1724/D