Freescale Semiconductor
Application Note

Document Number: AN3292
Rev. 0, 08/2006

XGATE Library: SCI Emulation

Emulating up to Four SCI Channels Using the Timer

by: Daniel Malik
MCD Applications, East Kilbride

The XGATE SCI emulation package is a collection of header and source files, in the C programming language, that enable the user to create virtual SCI (serial communication interface) peripherals by using the ECT (enhanced capture timer) or TIM (timer) peripheral. This is useful in situations where the application requires more SCI communication channels than readily available on-chip of the particular S12X family member.

Contents

1 Introduction 1
1.1 SCI Data Encoding 1
1.2 Support for LIN (Local Interconnect Network) 2
1.2.1 Tx Timing Accuracy and Rx Timing Tolerance 2
1.2.2 Break Characters 3
2 Emulation of SCI Functionality 3
2.1 Reception 3
2.2 Transmission 6
3 Using the Emulated SCI (EMSCI) Driver 8
3.1 Compile Time Configuration 8
3.2 XGATE Code 9
3.3 Run-Time Interface 10
3.3.1 Initialization 10
3.3.2 Polling 10
3.3.3 Special Characters 11
3.3.4 Interrupts 11
4 Performance and Required Resources 12
4.1 XGATE Performance 12
4.2 Required Resources 13
5 References 14

1 Introduction

1.1 SCI Data Encoding

The SCI peripheral uses asynchronous transfers and NRZ (non-return-to-zero) data encoding. Figure 1 shows a typical byte transfer using the SCI.

Each of the bits used to transfer data using the SCI occupies the same amount of time. The number of bits (start bit, stop bit, and data bits) transferred per second is known as the symbol rate and is expressed in baud (Bd).

freescale™ semiconductor

When there is no transfer in progress, the Rx or the Tx line typically stays in the high state (logic 1). Each data transfer begins with a logic 0 start bit. This enables the receiver to reliably identify the beginning of the transfer in the completely asynchronous system. Following the start bit, individual data bits are transferred starting with the least significant bit and ending with the most significant bit. There is a space of at least one bit time of logic 1 between any two transfers. This is referred to as the stop bit. The space between transfers does not have to be a multiple of the bit time, the minimum length is the only requirement it must satisfy.

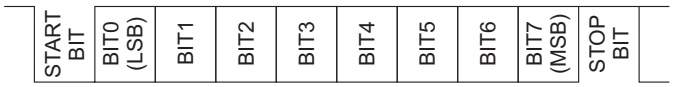

Figure 1. Byte Transfer Using the SCI Peripheral

The number of bits transferred in one data transfer is typically 8 (one byte), but other lengths are possible as well (typically 7 and 9 bits). The most significant bit (MSB) can optionally be used as a parity bit to improve reliability and noise immunity of the communication path.

1.2 Support for LIN (Local Interconnect Network)

The LIN bus belongs to a group of network architectures frequently used in the automotive industry and uses the SCI peripheral for its implementation. The LIN bus specification implies several requirements the SCI peripheral must satisfy to be able to participate in a LIN communication:

- Tx timing accuracy and Rx timing tolerance
- Ability to transmit and receive break characters

1.2.1 Tx Timing Accuracy and Rx Timing Tolerance

To ensure compliance with the LIN standard the LIN master must transmit at a symbol rate which deviates no more than ±0.5% from the nominal rate.

The LIN master must also receive data correctly from the slave devices connected to the bus even when the symbol rate of the slave device deviates up to ±1.5% from the nominal rate.

The transmission path of the SCI peripheral is typically implemented in such a way that the only source of inaccuracy is the clock source used to time the duration of the individual bits. Assuming that the clock source of the SCI peripheral deviates from the nominal frequency by the maximum allowed amount (0.5%), then the receiver (which uses the same clock source) must be able to accommodate at least ±2% deviation from the nominal rate.

The simplest possible implementation of an SCI receiver only synchronizes the receiver's clock with the falling edge at the beginning on the start bit. Then it simply samples the Rx line to receive the individual data bits.

This imposes certain limits on the maximum symbol rate deviation the receiver will tolerate. The situation when receiving from devices running at both faster than nominal and slower than nominal clock is considered in Figure 2. Figure 2 shows symbol rate deviation of ±4.5% while transferring 8 data bits. In this case the sampling window for the stop bit shrinks to only 10% of the nominal bit time. Tolerance of

±5% and higher symbol rate deviation is impossible as there would be no time left for sampling the stop bit. The situation gets worse for higher number of data bits than 8.

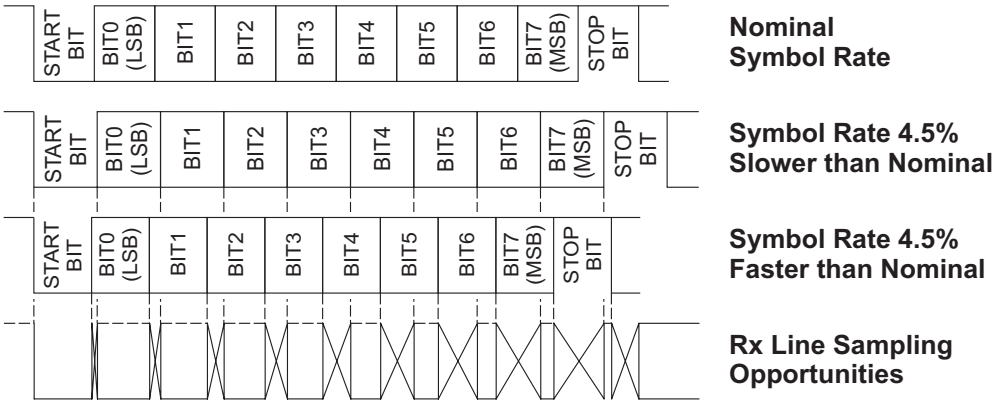

Figure 2. Reception of a Byte Considering Transfers at Both Slower and Faster Symbol Rates

1.2.2 Break Characters

A break character is defined as a continuous transmission of 0 for a specified duration of time. The duration is typically a multiple of the bit time. To make sure that the break character cannot be confused with a normal data transfer, the number of 0 bits transmitted in a break character must be higher than the number of data bits in a data transfer plus one (i.e., at least 10 for 8-bit data transfers).

The LIN standard makes use of the break character to denote beginning of a frame. To avoid confusion with normal data transfers, the LIN standard specifies that the break character must be at least 13 bit times long.

2 Emulation of SCI Functionality

This chapter describes one of the many possible algorithms for implementing SCI emulation in software. This particular algorithm makes use of features present on the standard timer (TIM) or the enhanced capture timer (ECT) peripherals. The emulation algorithms use the timer peripheral to capture changes of state of the input Rx lines and to generate changes of the output Tx lines. The input capture feature of the timer triggers interrupt execution of the reception algorithm and the output compare feature triggers interrupt execution of the transmission algorithm. While this division of the functionality into reception and transmission algorithm is very descriptive, it is not 100% accurate (as will be explained in Section 2.2, “Transmission”). The first section of the transmission algorithm is designed to resolve a particular situation which occurs during reception.

2.1 Reception

The reception of data does not have to be performed with bus cycle accuracy; however, the timing constraints depicted in Figure 2 must be satisfied (i.e., the state of the Rx line must be sampled at some point within the available sampling window). To allow the widest possible use of the emulation algorithm,

minimal restrictions must be imposed on the rest of the application when the emulation algorithm is created as part of a library. The reception algorithm described in this chapter allows for almost one full bit time of interrupt latency.

The reception algorithm uses the input capture feature of the timer module. This feature enables the application to capture the time of an edge on the input signal. The software processes the time differences between the edges to reconstruct the signal and receive the data.

A simplified flowchart of the reception algorithm is shown in Figure 3.

The receive algorithm is executed every time the timer detects an edge on the Rx line input.

First, the interrupt routine checks whether the edge was captured because of a fast transient on the Rx line. It does so by comparing the last known state of the Rx line with its current state. If the two states are the same, it means that the line has quickly changed its state and returned back to its current state. In this case the interrupt flag of the timer is cleared and the edge is ignored.

Once the routine determines that the current transition of the Rx line is not just a glitch, it checks whether a reception is currently in progress. If this is not the case, then the edge corresponds to the beginning of a new character.

If a new character is being received the routine checks whether the state of the start bit is 0 rather than 1. If a correct start bit is received, then the routine records the time of the edge detection and finishes.

If a reception is already in progress, the routine calculates the time difference between the last and currently recorded edge. It subsequently uses this information to calculate how many bit times have elapsed between the two edges and updates the shift register accordingly.

The routine must follow a different flow depending on the current state of the Rx line. If the timer has captured a rising edge, then the Rx line was low until this edge was detected, and the routine has simply to shift the corresponding number of 0 bits into the shift register. However, if the timer has detected a falling edge, the edge might be the beginning of the next character to be received. In this case, the routine must check whether a complete character was received after shifting each 1 bit into the shift register.

The algorithm allows each received sequence of 1 or 0 bits to be up to half a bit time longer or shorter than nominal. This allows maximum possible flexibility in terms of symbol rate inaccuracy.

The reception algorithm will detect a complete reception of a character only at the beginning of the next character. This is clearly unacceptable, as it would fail to receive the last character in a message. Reception of the last character is dealt with in the transmission interrupt. The transmission interrupt service routine checks whether the current time is past the end of the stop bit. If the transmission routine detects that a reception is still in progress even after the time when the stop bit should have been received, it terminates the reception and transfers the received character into the Rx buffer. The reception algorithm ensures that the periodic transmit interrupt is enabled during an active reception.

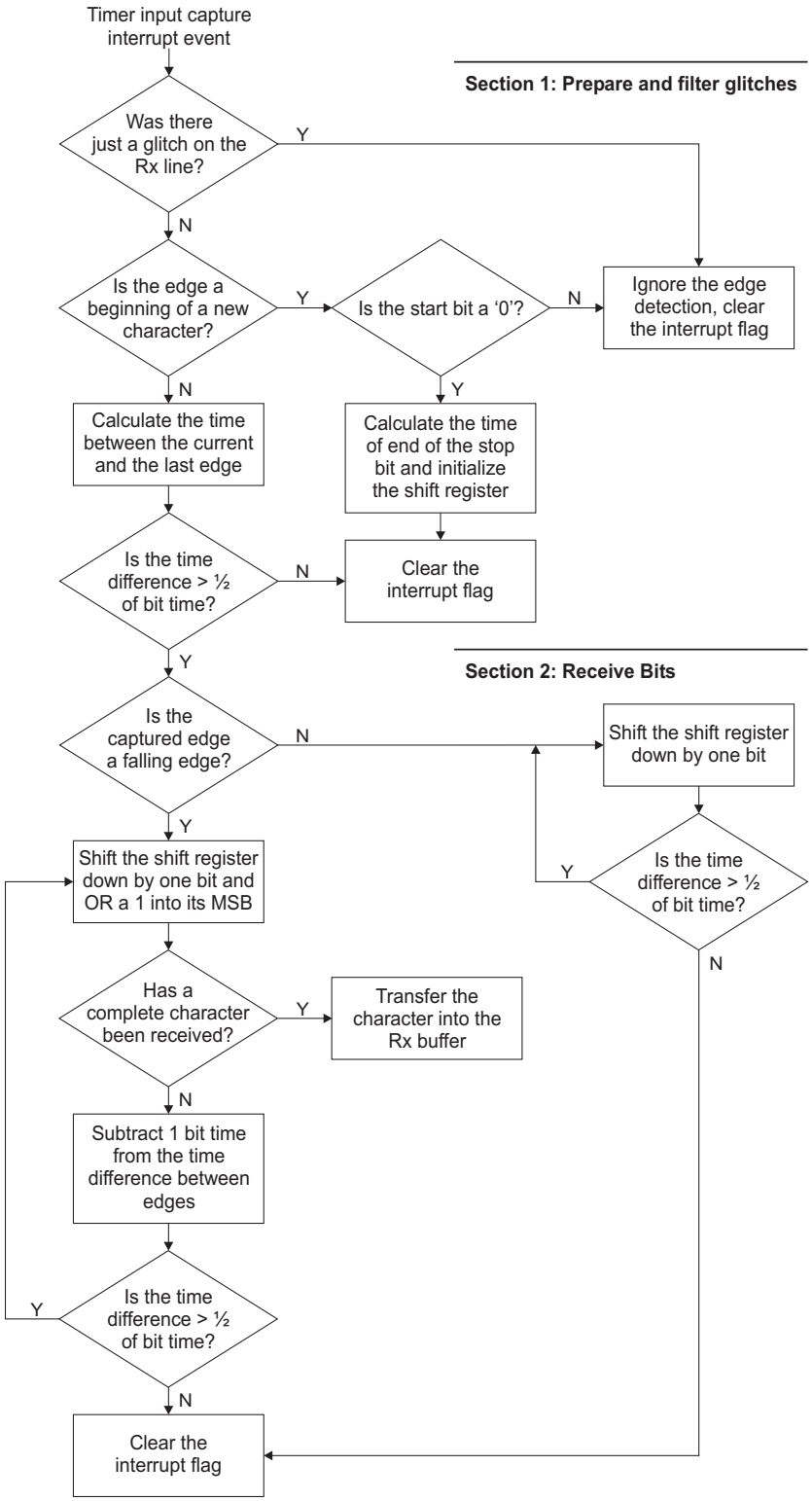

Figure 3. Simplified Flowchart of the Reception Algorithm

2.2 Transmission

Unlike the reception process, the transmission must be performed as accurately as possible. This enables the transmitter to operate correctly even in situations where all of the allowed symbol rate inaccuracy is eaten up by the inaccuracy of the clock source of the microcontroller. For this reason the transmission algorithm uses the output compare functionality of the timer to make sure that the symbol rate is bus cycle accurate. The use of the timer makes the emulated SCI even more accurate than the hardware SCI peripheral in situations where the bus clock frequency is not an exact multiple of 16 times the symbol rate.

A simplified flowchart of the transmission algorithm is shown in Figure 4.

The transmit algorithm is executed every time the timer generates an output compare interrupt.

The interrupt service routine verifies the state of the receiver and finishes reception of the character if there are no further edges on the Rx line input (as described at the end of Section 2.1, “Reception”).

After the receiver check is finished, the service routine adds one bit time to the output compare time. If a transmission is already in progress this will ensure that the next bit is transmitted exactly one bit time after the previous character.

If the transmitter has previously been idle, the output compare time for the previous bit might be outdated. In this case the service routine updates the output compare register to trigger after another bit time (from the current timer value). This means that if the transmitter is idle and the user fills the Tx buffer the actual transmission will start after a little over one bit time.

If the shift register is not empty, the service routine sets the timer to force the Tx line output to 0 or 1 (based on the LSB of the shift register) on the next output compare trigger. Then it shifts the shift register right by one bit (i.e., the bit value just transferred to the output compare timer logic is discarded). The service routine also clears the interrupt flag at this point, since the transmission is in progress and the timer will set the flag again as soon as the output compare logic triggers.

If the transmit shift register is not empty yet, there is no additional tasks to perform and the service routine ends. However if the output shift register is empty, the service routine tries to reload the shift register from the Tx buffer. If there is no data in the Tx buffer, then no more data is to be transmitted. In that case, the routine considers whether a reception is currently in progress. If a reception is in progress, then the periodic output compare interrupt is kept running and the interrupt flag is cleared. If no reception is in progress, then the whole emulated peripheral is idle and the output compare interrupt is disabled completely as it is no longer needed.

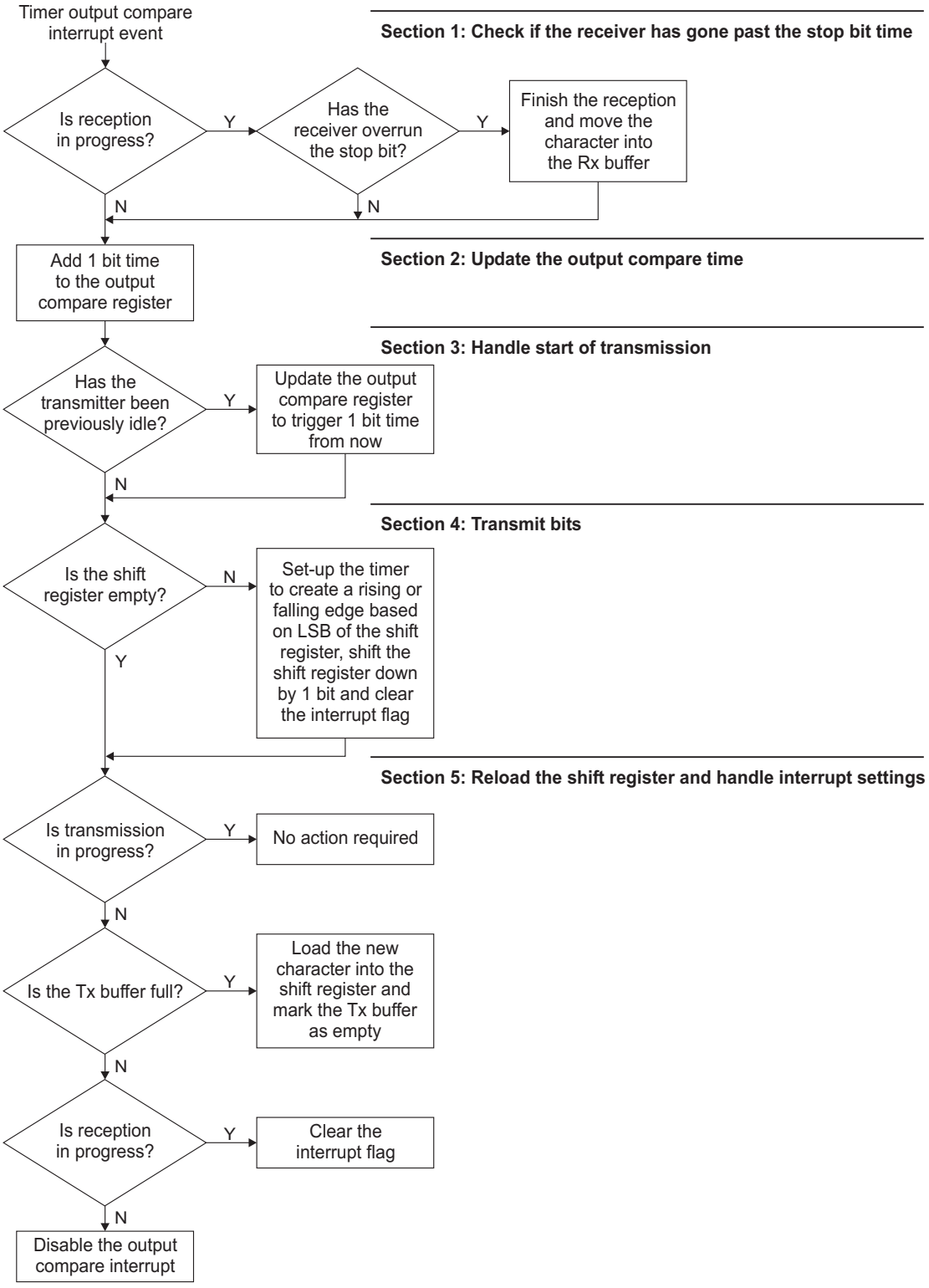

Figure 4. Simplified Flowchart of the Transmission Algorithm

3 Using the Emulated SCI (EMSCI) Driver

This section describes how to integrate the EMSCI driver into an application and how to use it.

3.1 Compile Time Configuration

All compile time configuration of the driver is performed by changing symbols definitions in file “emsci.h”. The symbol definitions that can be modified by the user in order to change the behavior of the driver are detailed below.

EMSCI_BUS_FREQUENCY

This symbol defines the bus frequency of the device in Hz. The driver uses this symbol to generate the correct emulated SCI symbol rate.

EMSCI_BAUD_RATE

This symbol defines the symbol rate in baud as required by the user. Typical values for LIN communication are 9600 Bd and 19200 Bd.

EMSCI_BIT_COUNT

This symbol defines the number of data bits transferred in a single transmission/reception. The allowed range is 1–14. The most frequent value is 8 (one byte per transfer). Other commonly used values are 7 (basic ASCII character communications) and 9 (byte transfers with parity).

EMSCI_TIMER_PRESCALER

To ensure correct operation of the driver the number of timer ticks per emulated SCI symbol must not be higher than 32767/(EMSCI_BIT_COUNT+1.5). The reason is that if zero data is received, the time of all the data bits plus the start bit transmitted at the lowest possible symbol rate must fit into a signed integer variable. This condition would not hold true for high bus speeds and low symbol rates. In such case the it is possible to pre-divide the timer clock by making use of the EMSCI_TIMER_PRESCALER parameter. Observe that the following equation is satisfied.

$$\frac{\text{EMSCI\_BUS\_FREQUENCY}}{2^{\text{EMSCI\_TIMER\_PRESCALER}} \cdot \text{EMSCI\_BAUD\_RATE}} < \frac{32767}{\text{EMSCI\_BIT\_COUNT} + 1.5}$$

If this is not the case, then the value of the EMSCI_TIMER_PRESCALER parameter must be increased. The allowed range for this parameter is 0–7.

EMSCI_CHANNELS

This parameter defines the required number of emulated SCI channels. The allowed range is 1–4. See Section 4.2, “Required Resources” for details on pin assignment.

EMSCI_SEMAPHORE

The architecture of the timer requires use of one of the XGATE semaphores. The allowed range is 0–7. This parameter specifies which semaphore is dedicated to the driver. The specified semaphore can only be reused by the application when the driver is idle (i.e., no transmissions and no receptions are in progress).

EMSCI_USE_INTERRUPTS

The driver is capable of generating CPU interrupts whenever one of the emulated SCI Tx buffers becomes empty or when one of the Rx buffers becomes full. The interrupts will only be generated if this symbol is defined as non-zero. See Section 3.3.4, “Interrupts” for details on interrupt channel assignments.

3.2 XGATE Code

All XGATE code of the EMSCI driver is store in file “emsci.cxgate”. This file contains the interrupt service routines which implement the transmission and reception algorithms described in Section 2, “Emulation of SCI Functionality”. Practical implementation of these algorithms uses a data structure for each emulated SCI peripheral to keep track of the current state of the peripheral and for data storage. The data structure is shown in Listing 1.

The Tx and Rx buffers form the user interface. The transmission algorithm takes data from the Tx buffer and the reception algorithm delivers the received data into the Rx buffer.

The Tx and Rx shift variables are used to serialize and deserialize the data. Stop and start bits are inserted into the shift variable when Tx data is transferred between the Tx buffer and the Tx shift. Conversely the start and stop bits are taken out of the data stream at the end of the reception process.

The Rx last edge time and last edge indicators are used by the reception algorithm to keep track of the time which has elapsed since the last edge was detected and to remember the polarity of the edge.

```
typedef struct {
    unsigned int rx_buffer;        /* buffer for received characters */
    unsigned int tx_buffer;        /* buffer for transmitted characters */
    unsigned int rx_shift;         /* receive shift register */
    unsigned int tx_shift;         /* transmit shift register */
    unsigned int rx_last_edge_time;/* time of the last edge on the Rx pin */
    unsigned int rx_last_edge:1;   /* 0 = last edge was falling; 1 = last edge was rising */
    unsigned int tx_in_progress:1; /* 1 = transmission in progress, 0 = transmitter idle */
    unsigned int rx_in_progress:1; /* 1 = reception currently in progress, 0 = receiver idle */
    unsigned int rx_bit_counter:5; /* count of bits till the end of the character */
    unsigned int unused:8;         /* align the structure to whole number of words */
} temsci_data;
```

Listing 1. Data Structure used for Internal Variables of the Emulated SCI Peripheral

The Tx and Rx in-progress indicators are used to decide whether the reception and transmission algorithms are required to handle the beginning of a data transfer.

The Rx bit counter is used by the transmission algorithm to detect whether the time frame for receiving the data has already elapsed and the data was not yet received because of lack of edges on the Rx input line.

The Tx and Rx interrupt routines are the only entry points of the driver and are attached to the timer output compare interrupt and the timer input capture interrupt respectively. The same interrupt service routines

are used to service the timer interrupts even when multiple SCI channels are emulated. The optional XGATE interrupt service routine parameter is used to differentiate between the different channels. Example of conditional setup for multiple emulated channels is shown in Listing 2.

```
#if (EMSCI_CHANNELS>3)
    {(xgfuncptr)emsci_tx, (xgdataptr)3},     /* Channel 70 - Enhanced Capture Timer channel 7 */
    {(xgfuncptr)emsci_rx, (xgdataptr)3},     /* Channel 71 - Enhanced Capture Timer channel 6 */
#else
    {ErrorHandler, U_XDP},                   /* Channel 70 - Enhanced Capture Timer channel 7 */
    {ErrorHandler, U_XDP},                   /* Channel 71 - Enhanced Capture Timer channel 6 */
#endif
#if (EMSCI_CHANNELS>2)
    {(xgfuncptr)emsci_tx, (xgdataptr)2},     /* Channel 72 - Enhanced Capture Timer channel 5 */
    {(xgfuncptr)emsci_rx, (xgdataptr)2},     /* Channel 73 - Enhanced Capture Timer channel 4 */
#else
    {ErrorHandler, U_XDP},                   /* Channel 72 - Enhanced Capture Timer channel 5 */
    {ErrorHandler, U_XDP},                   /* Channel 73 - Enhanced Capture Timer channel 4 */
#endif
#if (EMSCI_CHANNELS>1)
    {(xgfuncptr)emsci_tx, (xgdataptr)1},     /* Channel 74 - Enhanced Capture Timer channel 3 */
    {(xgfuncptr)emsci_rx, (xgdataptr)1},     /* Channel 75 - Enhanced Capture Timer channel 2 */
#else
    {ErrorHandler, U_XDP},                   /* Channel 72 - Enhanced Capture Timer channel 3 */
    {ErrorHandler, U_XDP},                   /* Channel 73 - Enhanced Capture Timer channel 2 */
#endif
    {(xgfuncptr)emsci_tx, (xgdataptr)0},     /* Channel 76 - Enhanced Capture Timer channel 1 */
    {(xgfuncptr)emsci_rx, (xgdataptr)0},     /* Channel 77 - Enhanced Capture Timer channel 0 */
```

Listing 2. Example of Interrupt Vector Table Setup for Multiple Emulated SCI Peripherals

3.3 Run-Time Interface

3.3.1 Initialization

The timer peripheral used for the emulation must be configured appropriately before the driver can be used. The routine for performing the initialization is named “emsci_setup” and is stored in file “emsci.c”. Once this routine is called from somewhere in the user application the timer is initialized and the emulated SCI peripherals are ready to be used.

3.3.2 Polling

The header file “emsci.h” contains a set of macros for identifying the state of the Tx and Rx buffers and for transmitting and receiving data.

EMSCI_TX_BUFFER_EMPTY(emsci_no)

This macro enables the user to test whether a particular Tx buffer is empty. The parameter of this macro is the number of the emulated SCI to test. The allowed values are 0 - (EMSCI_CHANNELS-1). The value of this macro is zero if the particular Tx buffer is full and non-zero if the buffer is empty.

EMSCI_TX(emsci_no, data)

This macro fills the selected Tx buffer with new data. The data is subsequently transmitted by the corresponding emulated SCI peripheral. After new data is written into a Tx buffer it will test as “full”. The buffer will again test as empty once the data is transferred to the shift variable and the transmission is initiated.

EMSCI_RX_BUFFER_FULL(emsci_no)

This macro enables the user to test whether a particular Rx buffer is full. The value of this macro is zero if the particular Rx buffer is empty and non-zero if the buffer is full. Rx buffers become full automatically after data is received through the corresponding emulated SCI channels.

EMSCI_RX(emsci_no, result_var)

This macro receives data from the selected Rx buffer into a user supplied variable. The Rx buffer will test as empty once the data has been received. Receiving data from the buffer destroys its contents. For example, particular data received through the emulated SCI peripheral can only be read once.

3.3.3 Special Characters

The data processed by the transmission and reception macros are 16-bit wide. Only the lower bits are used for standard data characters. Up to full 16-bit data can be used for transmission and reception of special characters (e.g., LIN break character). The following symbols are pre-defined in the “emsci.h” header file.

EMSCI_BREAK_10

A break character consisting of 10 consecutive 0 bits.

EMSCI_BREAK_13

A break character consisting of 13 consecutive 0 bits. This character can be used by a LIN master to indicate beginning of a new data frame.

3.3.4 Interrupts

The driver is capable of generating CPU interrupts on transmission and reception events. Assignment of these events to interrupt channels is detailed in Table 1.

Table 1. Association of Emulated SCI Events with CPU Interrupt Channels

Emulated SCI Event	Associated Interrupt Channel
EMSCI0 Rx	0x77 (Timer Ch0)
EMSCI0 Tx	0x76 (Timer Ch1)
EMSCI1 Rx	0x75 (Timer Ch2)
EMSCI1 Tx	0x74 (Timer Ch3)

Table 1. Association of Emulated SCI Events with CPU Interrupt Channels (continued)

Emulated SCI Event	Associated Interrupt Channel
EMSCI2 Rx	0x73 (Timer Ch4)
EMSCI2 Tx	0x72 (Timer Ch5)
EMSCI3 Rx	0x71 (Timer Ch6)
EMSCI3 Tx	0x70 (Timer Ch7)

Rx interrupt events are generated when data is received on the corresponding emulated SCI peripheral. Tx interrupt events are generated when a full Tx buffer becomes empty as a result of the data being transferred to the shift variable of the corresponding emulated SCI peripheral. This means that if a Tx interrupt is acknowledged (i.e., the appropriate interrupt flag is cleared), but the Tx buffer is not filled with new data, the interrupt will not be generated again automatically.

The CPU interrupt routines can use the macros described in Section 3.3.2, “Polling” for transferring data in and out of the Tx and Rx buffers.

Note that the CPU interrupts are not generated by the timer peripheral, but by the XGATE coprocessor. The interrupt flag, which must be cleared, is therefore located in the XGATE register space rather than in the timer space. An empty interrupt service routine for the Rx event of EMSCI0 peripheral is shown in Listing 3.

```
/* Rx interrupt service routine for EMSCI0 */
void interrupt emsci0_rx(void) {
    XGATE.xgif_70=0x0080;                    /* clear the interrupt flag */
}
```

Listing 3. Example of an Empty Interrupt Service Routine for the Rx Event of EMSCI0

4 Performance and Required Resources

The CodeWarrior compiler/debugger tools version 4.5 were used to measure performance of the driver and the required memory size. The example project which was used for the measurement is available in a zip file associated with this application note.

4.1 XGATE Performance

The XGATE load that the driver creates is dependant on the activity on the Tx output and Rx input lines of the emulated SCI peripherals. The load created by the transmission algorithm is independent of the data being transmitted or received. However, the load created by the reception algorithm depends on the number of edges on the Rx input. The load figures presented in Table 2 were measured under the following conditions:

- The driver is configured to transmit/receive 8-bit data at 19200 baud
- The maximum number of four SCI peripherals are emulated

- All transmitters are kept utilized to 100% (data transmitted back-to-back with no gaps)
- All receivers receive data at the maximum possible rate; constant data of 0x55 is received by all of the receivers at all times to maximize the number of edges the driver must process.
- The XGATE code executes from the on-chip RAM while the CPU code executes from the on-chip Flash and makes frequent accesses into the on-chip RAM. The microcontroller operates with a 40-MHz bus frequency.

Table 2. XGATE Load

Algorithm	Load [%]	Max Execution Time [Bus Cycles]
Reception	9.18	48
Transmission	9.75	63

The load figures scale proportionally with the number of emulated SCI peripherals and their symbol rate.

4.2 Required Resources

The current implementation of the driver requires the timer (TIM) or the enhanced capture timer (ECT) for its functionality. Unused channels of the timer peripheral can be used by the application, providing its use of the peripheral is compatible with the driver requirements.

The driver enables emulation of up to four SCI peripherals. The pins assigned to these emulated peripherals are shown in Table 3.

Table 3. I/O Pins Associated with Emulated SCI Functionality

Emulated SCI Signal	Assigned Pin
EMSCI0_RX	Timer channel 0 pin (PT0)
EMSCI0_TX	Timer channel 1 pin (PT1)
EMSCI1_RX	Timer channel 2 pin (PT2)
EMSCI1_TX	Timer channel 3 pin (PT3)
EMSCI2_RX	Timer channel 4 pin (PT4)
EMSCI2_TX	Timer channel 5 pin (PT5)
EMSCI3_RX	Timer channel 6 pin (PT6)
EMSCI3_TX	Timer channel 7 pin (PT7)

The data memory size required by the driver depends only on the number of SCI peripherals being emulated. The required code memory size is independent of the number of channels and stays constant. The required code and data memory sizes are shown in Table 4.

Table 4. Required Memory Size

Memory Content	Required Size [Bytes]
Code (Tx)	354
Code (Rx)	334
Data	12 x EMSCI_CHANNELS

5 References

MC9S12XDP512 Data Sheet, Freescale Semiconductor, Inc., 2005.

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: AN3292
Rev. 0
08/2006

freescale™
semiconductor