Freescale Semiconductor
Application Note

Document Number: AN3810
Rev. 0, 04/2009

Emulation of a SCI Module Using the IO Processor (IOP) in the MPC5510 Family

by: Oscar Luna
PMMC Software Engineer
GDL

Contents

1 Introduction 1
1.1 SCI Communication Protocol 1
2 Emulated SCI Module Principles 2
2.1 Tx Timing Accuracy and Rx Timing Tolerance 2
2.2 Reception Algorithm 3
2.3 Transmission Algorithm 5
3 Emulated SCI (EMSCI) Configuration 7
3.1 Pre-Compile Definitions 7
3.2 EMSCI Channel Configuration 9
4 EMSCI Runtime Interface 10
4.1 Initialization 10
4.2 Run-Time Structure 10
4.3 Polling 11
5 Required Resources 12
6 Sample Application 12
6.1 Introduction 12
6.2 Initialization 12
6.3 Scheduler Initialization 13
6.4 Sample Application 13
7 References 15

1 Introduction

This application note describes generating an emulated serial communication interface (SCI) by using the EMIOS peripheral and IOP in the MPC5510 family. The main objective of this application note is to enable users that require more SCI than is supported in the hardware and the ability to easily emulate in software using the IOP.

1.1 SCI Communication Protocol

The SCI peripheral uses an asynchronous serial transmission protocol. A unique start signal is sent prior each byte and a unique stop signal is sent after each byte.

The start bit indicates the receiving mechanism of the next frame sent. The stop bit sets a waiting time for receiving the next character.

After the stop bit is sent, the line may remain in an idle state for an indeterminate amount of time. Figure 1 shows a complete SCI frame transfer.

© Freescale Semiconductor, Inc., 2008. All rights reserved.

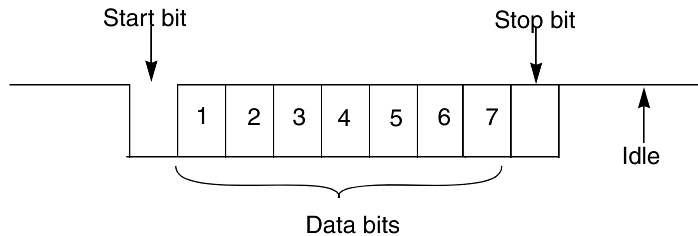

Figure 1. SCI byte Frame

Typically the number of bits transferred is 8, other lengths are also possible. The most significant bit can be optionally used as a parity bit to improve reliability and noise immunity in the communication path.

A baud rate is used to configure the transfer speed and each transmitted bit has the same amount of time.

2 Emulated SCI Module Principles

Several implementations are possible to implement an emulated SCI module. This application note makes use of independent EMIOS channels in transmit and receive algorithms.

The transmit algorithm uses the EMIOS channels output compare mode to generate bits at a certain baud rate with an accurate response. The transmit algorithm also changes the line to the appropriate logic level.

The reception algorithm uses the EMIOS module input capture mode to detect any input change in the line. The reception algorithm can identify if a glitch is present on the line and remove it to avoid incorrect data reception.

The split in functionality is not as independent as described. The first section of the transmission algorithm resolves a particular situation that occurs during reception. This particular process in the transmission algorithm is explained in Section 2.2, “Reception Algorithm”.

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 that 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 if the symbol rate of the slave device deviates up to ±1.5 % from the nominal rate.

The SCI peripheral transmission path is 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 the SCI peripheral clock source deviates from the nominal frequency by the maximum allowed amount (0.5 %), the receiver that uses the same clock source must then be able to accommodate at least ±2 % deviation from the nominal rate.

The simplest possible implementation of an SCI receiver synchronizes only the receiver clock with falling edge at the beginning of the start bit. It then samples the Rx line to receive the individual data bits. This imposes certain limits on the maximum symbol rate deviation the receiver tolerates. Figure 2 shows symbol rate deviation of ±4.5 % while transferring 8 data bits.

The sampling window for the stop bit reduces to 10 % of the nominal bit time. Tolerance of ±5 % and higher symbol rate deviation is impossible. There is no time left for sampling the stop bit. The situation gets complicated for higher number of data bits than 8.

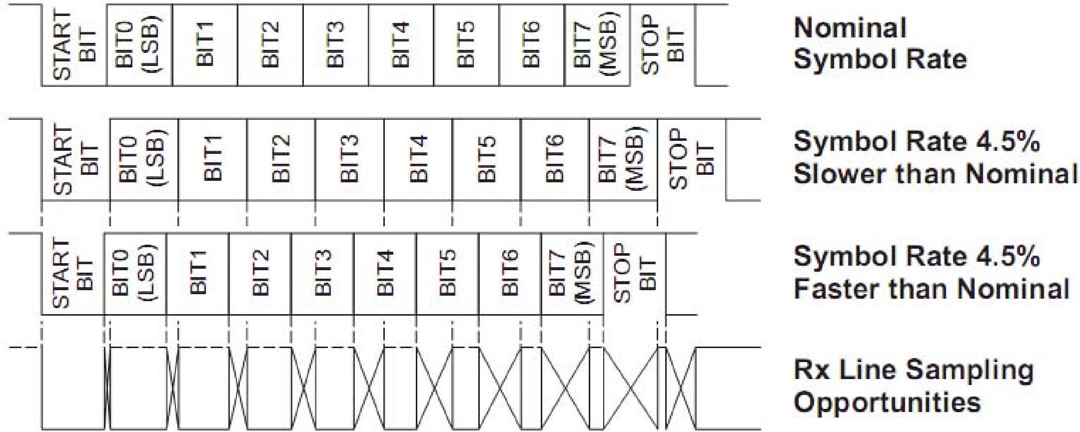

Figure 2. SCI Reception Byte at Both Slower and Faster Symbol Rates

2.2 Reception Algorithm

The reception algorithm uses the the EMIOS module input capture mode. This function enables the application to capture the time of an edge on any EMIOS input capture pin and has the ability to accept or reject data frames that are not within the allowable length range.

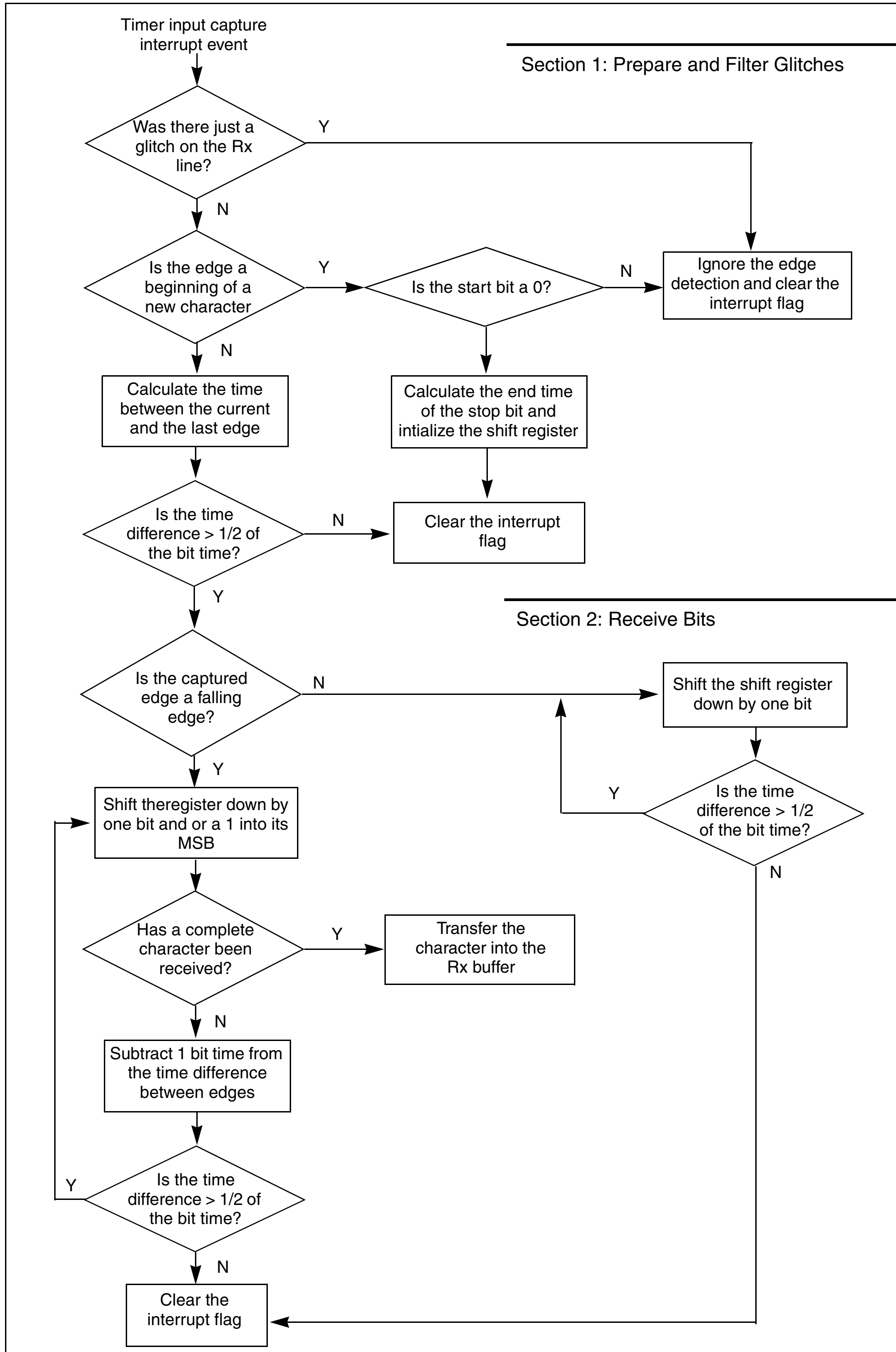

Figure 3. Simplified Receive Algorithm

Every time an edge in the Rx line is detected, the reception algorithm is executed. First, the algorithm checks whether the incoming signals is a fast transient on the line. If so, it does not validate the incoming data and the interrupt flag is cleared. The received algorithm calculates if a transient signal is detected by comparing the last known state of the Rx line with its current state. If the two states are the same this means that a transient occurred as the line quickly changed its state and returned back to its current state.

When the detected edge is validated, the algorithm verifies if a reception is already in progress. If a reception is not in progress, the detected edge then corresponds to a new character.

Every time a new character is validated, the algorithm checks whether the start bit is a 0 or 1. The reception flow continues only if the start bit has been correctly received, and if so the routine records the edge time of the start bit for future references for any incoming data.

If a reception is already in progress, the routine calculates the time difference between the last and currently recorded edge. This recorded information is used during the complete reception process, validating how many bit times have elapsed between two edges, and accordingly updating the shift register.

The algorithm follows different flows depending on the current state of the Rx line. When a rising edge is captured by the EMIOS channel, the routine knows that the previous state of the Rx line was low and the routine shift the corresponding number of 0’s into the shift register. On the other hand, when the EMIOS channel detects a falling edge, the routine knows that the previous state on the Rx line was high. The routine takes a different action whether to shift the 1’s into shift register. The routine validates if the received character is within the allowed tolerance range.

The reception algorithm can receive each sequence of 1 or 0 bits to be upto half a bit time longer or shorter than nominal. This allows maximum possible flexibility of symbol rate inaccuracy.

The reception algorithm detects a complete character reception only . However, the last character reception is dealt within the transmission interrupt. Section 2.3, “Transmission Algorithm” details how the transmit algorithm manages the completion of the last character during reception.

The transmit interrupt service routine checks whether the current time has past the end of the stop bit. When the transmission routine detects a reception is still in progress after the time stop bit ends, it terminates the reception and transfers the received character into the Rx buffer. The reception algorithm ensures the periodic transmit interrupt is enabled during an active reception.

2.3 Transmission Algorithm

The transmission algorithm uses the EMIOS module output compare mode to generate the output signals accurately. This enables the transmitter to operate correctly even in situations where all of the allowed symbol rate inaccuracy is consumed by the microcontroller clock source.

Figure 4 is a flow chart of a transmit algorithm.

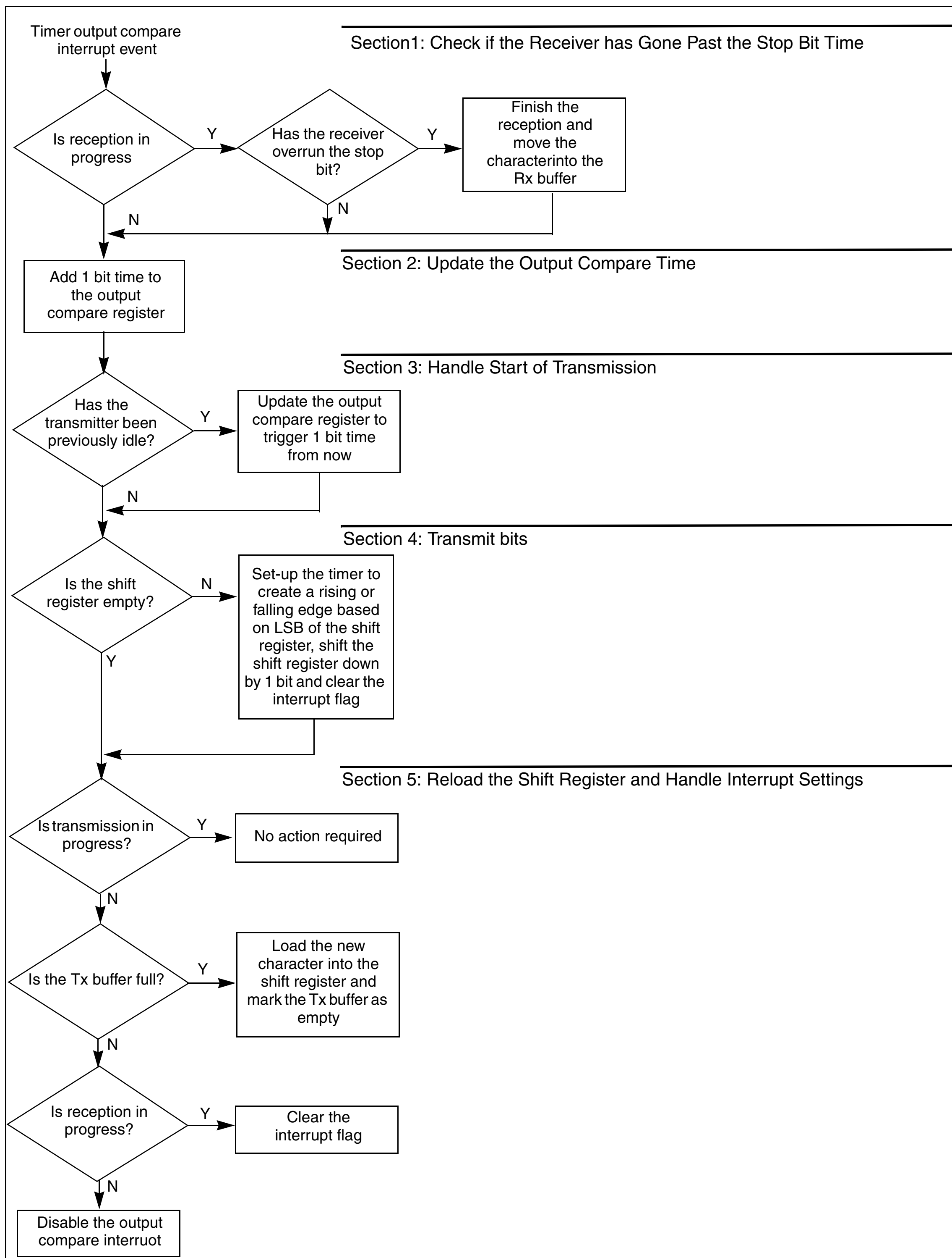

Figure 4. Simplified Flowchart of Transmission Algorithm

The transmission operation is enabled every time a particular emulated SCI channel is invoked. Every time an emulated SCI channel is used, an eMIOS channel is triggered to operate in output compare mode and starts to count until an output compare interrupt occurs.

When an output compare interrupt occurs, the transmit algorithm is executed and starts sending bits. The interrupt service routine first verifies the state of the receiver and finishes character reception if there are no further edges on the Rx line. The transmit algorithm has a section of code from the reception algorithm that validates a stop bit has been received, therefore the transmit algorithm checks the receiver routine finishes. After the algorithm validates the end of the receive routine, the service routine adds one bit time to the output compare time. If a transmission is already in progress this ensures the next bit time is transmitted after the previous character.

If the transmitter has previously been idle, the output compare time for the previous bit may be outdated. In this case, the service routine updates the output compare register to trigger after another bit time (from the current timer value). If the transmitter is idle and the user fills the transmit buffer the actual transmission starts after one bit time.

If the shift register is not empty, the service routine sets the timer to force the transmit line to a 0 or 1 state (based on the shift register LSB) on the next output compare trigger. It then shifts the shift register right by one bit. The bit value just transferred to the output compare timer logic is discarded. The service routine also clears the interrupt flag because the transmission is in progress the timer sets the flag again as soon as the output compare logic triggers.

If the transmit shift register is not empty, 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 transmit buffer.

If there is no data in the transmit buffer no more data is to be transmitted. In this case, the routine considers whether a reception is currently in progress. If a reception is in progress, the periodic output compare interrupt is then kept running and the interrupt flag is cleared. If no reception is in progress, the whole emulated peripheral is then idle and the output compare interrupt is disabled completely. It is no longer needed.

3 Emulated SCI (EMSCI) Configuration

This section describes how to configure the EMSCI driver to operate during run-time and describes pre-compile statements.

3.1 Pre-Compile Definitions

The EMSCI driver has many pre-compile definitions to enable or disable certain functionalities in the driver. All pre-compile definitions are located in file Emsci_Cfg.h with the exception of the macro named SYS_FREQ located in Mcu.h file.

SYS_FREQ — This definition holds the operating system frequency. This field must be captured at Hz units:

```
#define SYS_FREQ  (uint32_t) 64000000U  /* Hz units */
```

EMSCI_BUS_FREQUENCY — This definition is required to allow the EMSCI driver to calculate the correct baud rate. This field depends directly on the value captured in the SYS_FREQ macro.

EMSCI_BAUD_RATE — The timer output compare counter uses this value to generate the specific baud rate indicated by this macro definition. The baud rate value must be captured as shown below:

```
#define EMSCI_BAUD_RATE    19200
```

EMSCI_BIT_COUNT — The number of data bits to transmit or receive. The transmit algorithm uses this variable to send the amount of data bits through the Tx line. The receive algorithm makes use of this value to determine the number of data bits on the Rx line.

Values range goes from 0 to 14 bits. The typical value is 8, however there are other ranges such as 7 bits (basic ASCII communications) and 9 bits (byte transfers with parity).

```
#define EMSCI_BIT_COUNT    8
```

EMSCI_TIMER_PRESCALER — To ensure correct driver operation, the number of timer ticks per emulated SCI symbol must not be higher than 32767/(EMSCI_BIT_COUNT+1.5). 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 does not hold true for high bus speeds and low symbol rates. In such case, it is possible to pre-divide the timer clock by making use of the EMSCI_TIMER_PRESCALER parameter.

$$\frac{EMSCI_{BusFrecuency}}{(2^{EMSCI_{TimePrescaler}})(EMSCI_{BaudRate})} < \frac{32767}{EMSCI_{BusCount} + 1.5}$$

Eqn. 1

If Equation 1 is not satisfied, 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 number of EMSCI enabled by the initialization routine in the driver. Range allowed is from 1 to 4 channels. This parameter is located at Emsci_Cfg.h file.

```
#define EMSCI_CHANNELS    4
```

EMSCI_USE_INTERRUPTS — This parameter defines whether the EMSCI driver invokes a predefined software interrupt (not implemented in the hardware) after a transmission is finished or when a complete frame has been received.

```
#define EMSCI_USE_INTERRUPTS   1  /* Interrupts are enabled */
```

Each EMSCI channel has its own transmit and receive interrupts. Each interrupt is configured as below:

```
#define EMSCI_TX_FNC_0       TxInterrupt_0
#define EMSCI_RX_FNC_0       RxInterrupt_0
```

EMSCI_TX_FNC_0 — This definition is used by the driver to jump into the predefined function (TxInterrupt_0, as in the above code) after the transmit algorithm finishes the in-progress transmission.

EMSCI_RX_FNC_0 — This definition is managed exactly the same way as the transmit algorithm. EMSCI_TX_FNC_0 and EMSCI_RX_FNC_0 are strictly used by EMSCI channel 0. EMSCI channels 1–3 use their own software based interrupts. EMSCI channel 1 uses EMSCI_TX_FNC_1 and EMSCI_RX_FNC_1, definitions.

If interrupts are enabled, it is necessary to create the functions where the interrupt definitions (`EMSCI_TX_FNC_xx / EMSCI_RX_FNC_xx`)jump after a transmit or receive service has been executed.

3.2 EMSCI Channel Configuration

Each of the four available emulated SCI channels need to be configured independently for proper operation. The EMSCI channel configuration structure is located at Emsci_Cfg.c and Emsci_Cfg.h files.

The EMSCI configuration structure is composed of two different sections. Both sections have the same variables but are designated to configure transmit and receive channels of an EMSCI unit.

Each of the transmit and receive channels have an individual resource assigned. These resources are detailed within this section. Figure 5 shows the configuration structure used by the driver to initialize each emulated SCI channel.

```
typedef struct
{
   uint32_t u32Emsci_Unit;
   uint32_t u32Tx_EmiosChannel;
   uint32_t u32Tx_InterruptNum;
   uint32_t u32Tx_OutputPin;
   uint32_t u32Tx_OutputPinFnc;
   void (*TxEndNotification)(void);
   uint32_t u32Rx_EmiosChannel;
   uint32_t u32Rx_InterruptNum;
   uint32_t u32Rx_InputPin;
   uint32_t u32Rx_InputPinFnc;
   void (*RxEndNotification)(void);
} Emsci_ConfigType;
```

Figure 5. Emulated SCI Configuration Data Structure

The `u32Emsci_Unit` variable indicates what emulated SCI unit is used. This variable is used mainly by the driver initialization routine to know which emulated SCI unit needs to be configured.

The `u32Tx_EmiosChannel` and `u32Rx_EmiosChannel` variables indicate the number of the EMIOS channel used to emulate as a transmit or receive channel.

The `u32Tx_InterruptNum` and `u32Rx_InterruptNum` variables contain the physical interrupt vector number of the assigned EMIOS channel.

The `u32Tx_OutputPin` and `u32Rx_OutputPin` variables indicate the pin to be configured as an EMIOS channel. These pins are configured as output compare and or input capture modes. The chosen pins are required to support input capture or output compare modes.

The `u32Tx_OutputPinFnc` and `u32Rx_OutputPinFnc` variables contain the value to enable the EMIOS functionality of a particular pin.

The `TxEndNotification` and `RxEndNotification` fields are function pointer variables that serve to jump into a specific predefined function after a transmission or reception routine finishes. These functions must be previously declare in the application code.

4 EMSCI Runtime Interface

4.1 Initialization

The EMSCI driver needs to be properly configured before running. EMIOS timers have to be initialized before hand because the EMSCI driver is entirely based on the use of the EMIOS module for correct emulation of the SCI.

The driver is initialized by invoking function vfnemsci_init. This initialization routine makes use of the configuration data structure from the Emsci_Cfg.c file. The routine uses the configuration parameters to setup EMIOS channels to emulate the EMSCI channels.

Configuration structure setup is detailed in Section 3.2, “EMSCI Channel Configuration”.

The EMSCI initialization routine takes control of the configuration GPIO pins configuration that operate as EMIOS and handling the proper configuration of EMIOS channels. This routine also resets the run-time structure used by each emulated SCI channel.

4.2 Run-Time Structure

The EMSCI driver makes use of data structure during run-time operation for handling the four emulated SCI channels. Each of the emulated SCI channels uses their own run-time structure to handle the EMSCI channel timings and signals. The emulated SCI data structure used during run-time is shown in Figure 6.

```
typedef struct
{
  uint16_t rx_buffer;
  uint16_t tx_buffer;
  uint16_t rx_shift;
  uint16_t tx_shift;
  uint16_t rx_last_edge_time;
  uint16_t rx_last_edge:1;
  uint16_t tx_in_progress:1;
  uint16_t rx_in_progress:1;
  uint16_t rx_bit_counter:5;
  uint16_t unused:8;
} temsci_data;
```

Figure 6. Emulated SCI Data Structure Used During Runtime

The Tx and Rx buffers (`tx_buffer , rx_buffer`), are used to save receive or transmit information. The Tx buffer contains the complete byte to send, meanwhile all received bits from the Rx line are stored in the Rx buffer.

The Tx and Rx shift variables are used by the driver to compose and decompose the data from the Tx and Rx buffers. During the transmit process, start and stop bits are inserted in the shift process and contrary to this process the receiving routine removes the start and stop bits to store only the data bits into the Rx buffer.

The Rx last edge time and last edge indicators are used by the reception algorithm to keep track of the time elapsed from the last edge detected and the edge polarity.

The Tx and Rx in-progress indicators are used by the driver to decide whether transmission and reception algorithms are needed 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 if the data is not received, it is because of lack of edges on the Rx line.

4.3 Polling

The emsci.h file contains polling macros that retrieve status of a particular transmit or receive emulated SCI channel. This file also contains two macros to write and read data into and from a particular emulated SCI channel.

4.3.1 EMSCI_TX_BUFFER_EMPTY (emsci_no)

This macro enables the user to test if 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 to 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.

4.3.2 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 tests as full. The buffer tests again as empty once the data is transferred to the shift variable and the transmission is initiated.

4.3.3 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 automatically full after data is received through the corresponding emulated SCI channels.

4.3.4 EMSCI_RX (emsci_no, result_var)

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

5 Required Resources

The emulation of the SCI driver requires the usage of several EMIOS channels the user needs to be aware of. These EMIOS channels cannot be assigned to handle other tasks within the users application because the performance of the emulated SCI driver is considerably affected.

Table 1 shows the assigned EMIOS channels that emulate SCI channels.

Table 1. Emios Channel Association with Emulated SCI Channels

Emulated SCI Event	Associated Interrupt Channel	Vector Interrupt
EMSCI0 Rx	EMIOS Ch0	58
EMSCI0 Tx	EMIOS Ch1	59
EMSCI1 Rx	EMIOS Ch2	60
EMSCI1 Tx	EMIOS Ch3	61
EMSCI2 Rx	EMIOS Ch4	62
EMSCI2 Tx	EMIOS Ch5	63
EMSCI3 Rx	EMIOS Ch6	64
EMSCI3 Tx	EMIOS Ch7	65

EMSCI events are not necessarily conneceted to specific EMIOS channels. The user can re-map the four EMSCI channels to any available EMIOS channel on the microcontroller. The EMIOS channel remapping can be done by modifying the second, third, and fourth parameters located on each element array of the configuration structure. These parameters are located in Emsci_Cfg.c file.

6 Sample Application

6.1 Introduction

The sample application proposed in this application note demonstrates operation of the EMSCI driver with a scheduler running on the second core of the MPC5510 microcontroller.

6.2 Initialization

The sample application first initializes the EMSCI driver before running scheduler tasks. The first step is to invoke the initialization function, emsci_init.

The emsci_init function configures all four emulated SCI channels by assigning EMIOS channels 0–7 to each emulated SCI channel. EMIOS channel assignation is described in Table 1. Microcontroller pins from PortC (0–7) are configured to operate in output compare and input capture modes.

A baud rate operation of 9600 bps is configured for all transmitters and receivers. EMSCI run time structures are initialized to avoid any type of malfunctions in the driver.

```
/* Initialize Pwm module */
vfnemsci_init(Emsci_ChannelConfig);
```

6.3 Scheduler Initialization

After initialization execution, the scheduler initialization function `vfnScheduler_Init` runs and configures EMIOS channel number 8 as a time base reference.

A macro definition `LOOP_TIME_50ms` located in the Emios.h file calculates the value of the EMIOS channel 8 counter to generate an interrupt service every 50 ms. This macro makes use of the actual system clock used by the sample application of 64 MHz.

After EMIOS channel 8 is configured, the scheduler remains in a hold state until the function `vfnStart_Scheduler` is invoked. This function enables the EMIOS 8 timer to start counting and enables the complete operation of the scheduler.

Figure 7 shows the correct initialization and execution of the scheduler.

```
vfnScheduler_Init();  /* Initialize Scheduler timebase
*/
vfnStart_Scheduler(); /* Start Tasks execution
*/
```

Figure 7. Schedule Initialization Procedure

6.4 Sample Application

The sample application consists of sending different bytes through all four EMSCI channels. Simultaneously all receivers validate the bytes sent by the transmitters. This sample application is to show the basic operation of the EMSCI driver.

The scheduler is assigned to transmit and receive all bytes at different intervals. These intervals are programmed at intervals of 100 ms, 200 ms, 400 ms, and 800 ms. The first task interval of 100 ms uses EMSCI TX channel 0 to send data. The data sent by TX channel 0 is 0x10.

After 200 m, the scheduler issues the second task to invoke the EMSCI Rx channel 0 to validate the data sent by TX channel 0. On the same scheduler thread, TX channel 1 is called to send data. The data sent by the TX channel 1 is 0x20.

When 400 ms elapse in the scheduler, the third task is executed by using the RX channel 1 to validate the data sent by TX channel 1. If RX channel validates a received value of 0x20, a flag is set to indicate the correct validation of the byte. In this same thread the TX channel 2 sends data. The data sent by TX channel 2 is 0x52.

After 800 ms elapse, the last task is executed by reading data from TX channel 2. The validation is performed by RX channel 2 and a flag is set whenever the received data is correct. TX channel 3 now enters into execution and sends data. The data sent by TX channel 3 is 0x74.

When the last thread is finally executed, the scheduler remains constantly sending and validating receiving bytes.

The EMSCI sample application running in a multi-thread scheduler shows the complete sample application mounted on the scheduler:

```
void main(void)
{
/* Initialise Scheduler handling variables */
  gu8SleepModeEnabled = 0;
  gu8Scheduler_Ctrl   = 0;
  gu8Scheduler_Flag   = 0;

  vfnInit_PLL();
  initIrqVectors();          /* Initialize exceptions: only need to load IVPR */
  initINTC();                /* Initialize INTC for hardware vector mode      */
  vfnEmiosConfig_General_Clocks( ); /* Initialize general Emios clocks     */

/* Initialize Pwm module */
vfnemsci_init(Emsci_ChannelConfig);

vfnScheduler_Init();    /* Initialize Scheduler timebase     */ vfnStart_Scheduler(); /* Start
Tasks execution                          */
enableIrq();           /* Ensure INTC current prority=0 & enable IRQ    */

while (gu8SleepModeEnabled == 0)
  {
       if ((gu8Scheduler_Flag & (uint8_t)0x01) == (uint8_t)0x01)
        {
            /*-- Allow 100 ms periodic tasks to be executed --*/
            EXECUTE_100MS_TASKS();
                 /* Scheduled tasks finished, clear control flag */
            gu8Scheduler_Flag = (uint8_t)0x00;
        }
        else
        {
            if ((gu8Scheduler_Flag & (uint8_t)0x02) == (uint8_t)0x02)
            {
                /*-- Allow 200 ms periodic tasks to be executed --*/
                 EXECUTE_200MS_TASKS();
                      /* Scheduled tasks finished, clear control flag */
                 gu8Scheduler_Flag = (uint8_t)0x00;
            }
            else
            {
                if ((gu8Scheduler_Flag & (uint8_t)0x04) == (uint8_t)0x04)
                {
                    /*-- Allow 400 ms periodic tasks to be executed --*/
                     EXECUTE_400MS_TASKS();
                    /* Scheduled tasks finished, clear control flag */
                    gu8Scheduler_Flag = (uint8_t)0x00;
                }
                else
                {
                    if ((gu8Scheduler_Flag & (uint8_t)0x08) == (uint8_t)0x08)
                    {
                        /*-- Allow 800 ms group A periodic tasks to be executed --*/
                         EXECUTE_800MS_A_TASKS();
                              /* Scheduled tasks finished, clear control flag */
                        gu8Scheduler_Flag = (uint8_t)0x00;
                    }
                    else
                    {
```

```
                    if((gu8Scheduler_Flag & (uint8_t)0x10) == (uint8_t)0x10)
                      {
                         /*-- Allow 800 ms group B periodic tasks to be executed --*/

                          /* Scheduled tasks finished, clear control flag */
                          gu8Scheduler_Flag = (uint8_t)0x00;
                      }
                  }
              }
          }
      }
   }
}
} /* End of main */
```

7 References

AN3292 Appnote (XGATE Library: SCI Emulation), Freescale Semiconductor.

MPC5510 Microcontroller Family Reference Manual.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

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)
www.freescale.com/support

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 China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
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 +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

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

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

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

© Freescale Semiconductor, Inc. 2008. All rights reserved.

AN3810
Rev. 0
01/2009