Freescale Semiconductor
Application Note

AN3144
Rev. 0, 10/2005

Using XGATE to Implement a Simple Buffered SCI

by: Steve McAslan
TSPG MCD Applications Engineering

1 Introduction

The XGATE module allows a new approach to implementing device drivers on the S12X family. Since the XGATE is fully programmable in C, the functionality of the drivers may range from simple DMA activities to complex protocol and data handling. This application note shows how to implement a very simple interrupt-driven buffer for the SCI module, and illustrates the three steps required to configure the XGATE to service the peripheral interrupts.

2 The XGATE and Peripheral Handling

The architecture of the S12X and the interaction between the XGATE and the CPU are discussed extensively in several application notes, notably AN2615, AN2685 and AN2734, and interested readers should refer to those for a detailed description of the MCU.

In summary, the XGATE is a 16-bit RISC processor that can execute program code when an interrupt occurs on the MCU. For example, the serial communications

Table of Contents

Note:
Source code for the software examples in this application note can be found in the file AN3144SW.zip, available at http://www.freescale.com.

freescale™ semiconductor

interface (SCI) receiving a byte can raise an interrupt that will cause the XGATE to execute code. When executing its code, the XGATE can read and write the contents of the RAM and peripheral registers. This means that the XGATE can copy data to or from RAM and peripherals and, so, can implement a simple DMA or buffer operation independently of the CPU. It is also possible to create very algorithmically complex functions for the XGATE, as it is the MCU software designer who provides the code that the XGATE executes. By convention, an XGATE interrupt handler is known as a thread.

As in traditional microcontroller architecture, the CPU also has the capability of handling these interrupts, so the software designer has full control of whether the CPU or the XGATE will be the target for the interrupt event. In addition, the XGATE can raise an interrupt and direct this to the CPU, which allows events to be handled on two levels: first, activity by the XGATE; and second, a higher level function performed by the CPU. An example of this behavior is illustrated in this application note.

Finally, it is worth noting that the XGATE is optimized for data-handling operations and can execute with a bus cycle time that is one half of the CPU bus cycle time. By careful software design, use of the XGATE can significantly improve the performance of the S12X by freeing the CPU from handling many real-time interrupt events.

3 Three Steps to Use XGATE

Based on the description given above, there are three simple steps that the software designer must take to allow the XGATE to handle an interrupt. The steps must be followed for each individual interrupt created by a peripheral.

1. Direct the interrupt event to the XGATE.
2. Create a thread to handle the interrupt.
3. Initialize the XGATE interrupt vector table to point to the thread.

In fact, since the XGATE is a peripheral itself. it must also be enabled; however, this is a one-time only operation, typically executed shortly after reset.

Let us look at each step in turn...

3.1 Direct the Interrupt Event to the XGATE

Each interrupt source has the option of having a handler executed by the CPU or the XGATE. After reset all interrupts are directed to the CPU.

Each interrupt source (i.e., each interrupt that has a unique vector) has a configuration register associated with it in the interrupt controller. This defines the priority of the interrupt and also contains a single bit called RQST that determines whether the interrupt goes directly to the CPU (bit clear) or to the XGATE (bit set). To avoid filling the memory map with lots of unique registers, the S12X interrupt controller arranges them into multiple banks. To change the contents of a register, first select the correct bank and then write the value into the correct register.

In most cases, this can be done simply with a macro, as shown in Figure 1.

```
#define ROUTE_INTERRUPT(vec_adr, cfdata)                \
  INT_CFADDR= (vec_adr) & 0xF0;                         \
  INT_CFDATA_ARR[((vec_adr) & 0x0F) >> 1]= (cfdata)
```

Figure 1. Interrupt Routing Macro

Once the macro is defined, you can use it to initialize all the interrupt vectors to their designed values as shown in Figure 2.

```
#define SCI0_VEC  0xD6 /* vector address= $xxD6 */
  ROUTE_INTERRUPT(SCI0_VEC, 0x81); /* RQST=1 and PRIO=1 */
```

Figure 2. Interrupt Routing Macro (in Use)

3.2 Create a Thread to Handle the Interrupt

Creating threads for XGATE is almost identical to creating an interrupt handler for the CPU. If the thread is created in assembler, the assembly code for the CPU and XGATE will be completely different (since they are two completely different processors); for portability reasons, this approach is not recommended. If the thread is created in C, the only difference is that the function definition for the thread can contain a parameter that is passed from the vector table to the XGATE before the thread begins to execute.

The thread for XGATE shown in Figure 3 will look very familiar to experienced embedded software engineers. In fact, this code could be compiled and executed on the CPU without modification.

```
interrupt void SCI_Thread(void)
{
  /* read the status register - required to clear the int flag */
  SCI0SR1;
    /* write the next byte of data */
  SCI0DRL = '*';
}
```

Figure 3. XGATE Thread

As shown in Figure 3, the thread ignores the additional parameter available from the vector table. To use this parameter, simply declare it in the function header as shown in Figure 4.

```
interrupt void SCI_Thread(tBuffer* Data)
{
  if (Data->size > 0)
  {
    /* read the status register - required to clear the int flag */
    SCI0SR1;
    /* write the next byte of data */
    SCI0DRL = Data->character[Data->size-1];
    Data->size--;
    if (Data->size == 0)
    {
      SCI0CR2_SCTIE = 0; //Disable SCI interrupt
      _sif(); //Send interrupt to CPU
    }
  }
}
```

Figure 4. XGATE Thread with Parameter

The value of the Data parameter is initialized from the XGATE vector table. The parameter is a 16-bit value that can be interpreted as a scalar value or a pointer, or can be completely ignored.

3.3 Initialize the XGATE Interrupt Vector Table to Point to the Thread

It is important to note that the XGATE vector table is completely independent from the CPU vector table.

An XGATE vector table has two entries for each unique vector. The first entry is a 16-bit pointer to the thread and the second entry is the value of the parameter that is passed when the interrupt occurs. A typical vector table definition is shown in Figure 5.

The XGATE vector table can be placed anywhere in the XGATE memory map, but the XGVBR register must point to the correct start address of the table.

```
const XGATE_TableEntry XGATE_VectorTable[] =
{
  {ErrorHandler, 0x09},  // Channel 09 - Reserved
  {ErrorHandler, 0x0A},  // Channel 0A - Reserved
...
  {ErrorHandler, 0x6A},  // Channel 6A - SCI1
 {(XGATE_Function)SCI_Thread, (int) &Buffer0}, // Channel 6B - SCI0
  {ErrorHandler, 0x6C},  // Channel 6C - SPI0
...
  {ErrorHandler, 0x78},  // Channel 78 - Real Time Interrupt
  {ErrorHandler, 0x79},  // Channel 79 - IRQ
};
```

Figure 5. XGATE Vector Table

As seen in Figure 5, many of the vectors may be unused. In this case it is good practice to define a thread that will allow a graceful recovery. For the SCI0 entry, we can see a pointer to our thread (SCI_Thread) and the value of our parameter (in this case the address of our data buffer).

4 A Simple Example

The remainder of this application note introduces an example to demonstrate a simple SCI implementation for the XGATE.

The example project is named Simple SCI.mcp and contains two user source files: one for the CPU (main.c) and one for XGATE (xgate.cxgate). Within the xgate.cxgate file there is a compile switch that allows us to build a very simple example or a more complex buffered example. We will first consider the simple example.

In the main.c file we can see there are three functions: SetupXGATE, main, and SCI_Handler. The latter function is applicable only in the buffered case and will be discussed later.

SetupXGATE initializes the XGATE module and directs the SCI interrupt to the XGATE (which is step 1 in our sequence).

Figure 6 shows the contents of SetupXGATE(). The first step is to tell XGATE where its vectors are — that is, we initialize XGVBR to point to the vector table that we will create. This is typically a lower address than the defined start address of the C structure, since in most cases XGATE has fewer vectors than its maximum, and so the unused vector space can be used for other purposes. Following this, we change the SCI interrupt configuration so that it goes to the XGATE instead of the CPU — we use the macro previously defined in Figure 1. In this example, we are handling a single interrupt so only one macro call is made; in a typical application we would expect to initialize several more interrupts in this way. Lastly, we enable the XGATE module with the configuration that we require.

```
static void SetupXGATE(void)
{
  /* initialize the XGATE vector block and
     set the XGVBR register to its start address */
  XGVBR= (unsigned int)(void*__far)(XGATE_VectorTable - XGATE_VECTOR_OFFSET);

  /* switch SCI0 interrupt to XGATE */
  ROUTE_INTERRUPT(SCI0_VEC, 0x81); /* RQST=1 and PRIO=1 */

  /* enable XGATE, its interrupts and the debug's freeze mode */
  XGMCTL= 0xFBC1; /* XGE | XGFRZ | XGIE */
}
```

Figure 6. SetupXGATE

The main() routine contains the various calls to initialize the application and the main loop for the CPU. It firstly enables interrupts on the CPU, although this is required for the buffered example only. Next it calls SetupXGATE.

The following step is to initialize the buffer for our buffered example. Finally, it initializes the SCI and enables the SCI interrupt. From then on, the CPU will sit in an endless loop while XGATE services the SCI interrupt. The XGATE will receive the first SCI interrupt, informing it that the SCI transmit buffer is empty, and queue the next character.

We turn to the xgate.cxgate file to see the remaining two steps in our sequence. Step 2 was to create a thread, and the code for the simple example is shown in Figure 3. The code simply clears the interrupt flag and sends a '*' character out on the SCI. When this character has been sent, the SCI will raise another buffer empty interrupt, which the XGATE will service in the same way, and so on.

The final step in the sequence is to initialize the XGATE vector; this was shown in Figure 5. The SCI vector location in the table contains a pointer to our thread named SCI_Thread. The second parameter in the table is ignored in the simple example.

The output from the SCI in this case is a string of '*' characters, as shown in Figure 7.

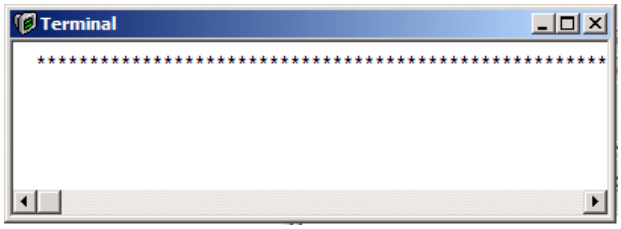

Figure 7. Example SCI Output on CodeWarrior Simulator

5 Buffered Example

The Simple SCI.mcp project contains a more complex example that implements a buffered SCI. In this configuration, the CPU initializes a small buffer that the XGATE transmits. On completion, the XGATE sends an interrupt to the CPU to allow it to configure the next buffer content. This is shown diagrammatically in Figure 8.

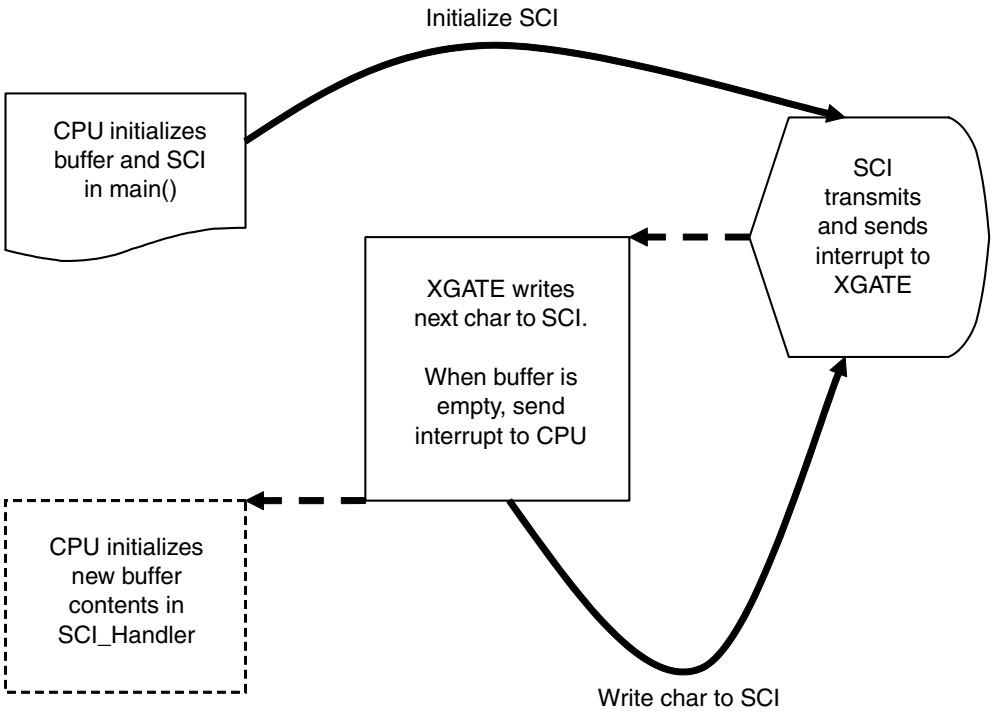

Figure 8. Structure of Buffered Example

The definition of the buffer is in the xgate.h file and is shown in Figure 9.

```
typedef struct
{
  unsigned char size;
  unsigned char character[8];
} tBuffer;
```

Figure 9. Definition of Buffer

The buffer is a structure containing an array of up to eight characters and a size parameter indicating how many entries in the array are used. Returning to main(), we can see in Figure 10 the buffer being initialized.

```
//Initial first SCI Buffer
 Buffer0.size = 4;
 Buffer0.character[0]='1';
 Buffer0.character[1]='2';
 Buffer0.character[2]='3';
 Buffer0.character[3]=' ';
```

Figure 10. Buffer Initialization

Figure 4 contains the thread that the XGATE uses to transmit the buffer. Note that the buffer itself is passed as a parameter from the XGATE vector table. Each interrupt will cause the XGATE to write the next character into the SCI and then terminate — when XGATE is not executing a thread, it is completely stopped.

When the final character has been written to the SCI, the XGATE sends an interrupt to the CPU using its SIF opcode. The final task for the XGATE is to disable the SCI interrupt since there is no more data to transmit.

In this case, the interrupt to the CPU is received on the same channel as the original interrupt was on; in other words, the CPU fetches the SCI vector and begins to execute code from its SCI_Handler function. The XGATE has the capability to direct any channel interrupt to the CPU using another form of the SIF opcode, but this is not discussed here. Note also that there is no requirement to use the SIF opcode at all, as in the case of the simple example.

The SCI_Handler function is in the main.c file and is shown in Figure 11.

```
interrupt void SCI_Handler()
{
  unsigned char temp;
  //Clear XGATE interrupt flag - SCI0 is channel $6B
  XGIF1 = 0x0800;

  //Initialize buffer with new values
  Buffer0.size = 4;
  temp = Buffer0.character[0];
  Buffer0.character[0] = Buffer0.character[1];
  Buffer0.character[1] = Buffer0.character[2];
  Buffer0.character[2] = temp;

  /* Enable the SCI Tx interrupt to start the transfer */
  SCI0CR2_SCTIE = 1;
}
```

Figure 11. CPU Interrupt Handler

The first task of any interrupt handler is normally to clear the flag that caused the interrupt. Although the CPU fetched the SCI interrupt vector, the interrupt did not actually come from the SCI module so it does not clear an SCI flag; instead, it clears the channel flag in the XGATE. This system of associating the XGATE interrupt with a peripheral channel allows a close correlation between the application behavior and the software. From the perspective of the CPU, the interrupt function appears to be servicing a buffered SCI rather than a simple SCI.

All of the XGATE channel flags are located in a block within its registers, and flags are cleared by writing a ‘1’ to them. Care should be taken to avoid accidentally clearing more than one flag.

Once the flag is cleared, the CPU performs a simple rotate procedure on the buffer contents — this will allow us to see its effect on the transmitted output. The number of characters in the buffer is set to four once again, then the handler re-enables the SCI interrupt.

Note that there is an opportunity for a configuration conflict to occur here as, on the one hand, the XGATE is disabling the interrupt, and on the other, the CPU is enabling it. The order in which these occur must be carefully considered. In this case, the example is so simple that the XGATE will always disable the interrupt before the CPU re-enables it, so there is no difficulty. In more complex systems, extra care should taken if using this approach — the XGATE provides hardware semaphores for this type of situation.

To build this buffered example, remove the #define SIMPLE macro in the xgate.cxgate file. The output from the SCI is a string of ‘123’ characters in various orders as shown in Figure 12.

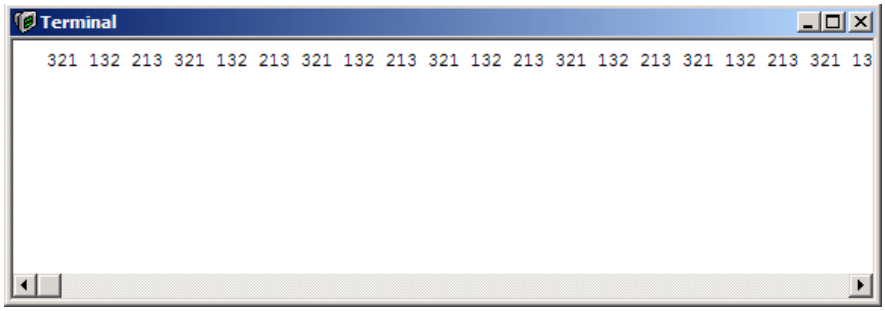

Figure 12. Example SCI Output on CodeWarrior Simulator

As a final note, consider the fact that the buffer was passed as a parameter into the XGATE interrupt handler. It is possible to exploit this approach to build a universal handler for all SCIs by simply extending the content of the buffer structure. Adding a pointer to the SCI within the buffer structure allows us to write a single handler that will implement a buffer scheme on all SCIs. Figure 13 introduces an example structure and a modified thread that takes advantage of this approach.

```
typedef struct
{
  tSCI     *pPort;     /* pointer to an SCI */
  unsigned char size;
  unsigned char character[8];
} tBuffer;

interrupt void SCI_Handler(tBuffer* Data)
{
  if (Data->size > 0)
  {
    /* read the status register - required to clear the int flag */
    Data->pPort->SCISR1;
    /* write the next byte of data */
    Data->pPort->SCIDRL= Data->character[Data->size-1];
    Data->size--;
    if (Data->size == 0)
    _sif(); //Send interrupt to CPU
  }
  else dATA->pPort->SCICR2.bit.tie = 0; //Disable interrupt
```

Figure 13. A Universal SCI Handler

6 Summary

By following the three steps described it is possible to direct any peripheral interrupt on the S12X to the XGATE module. In turn, the XGATE can either completely process the interrupt or, optionally, send a further interrupt to the CPU when it has completed the task.

Using this simple approach allows the CPU to off-load many or all interrupt servicing functions to the XGATE, thus freeing up processing power for use elsewhere in the application. It is this dual-core approach to real-time applications that allows S12X to have significantly higher performance than previous 16-bit solutions.

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

AN3144
Rev. 0, 10/2005

freescale™
semiconductor