Freescale Semiconductor
Application Note

AN3145
Rev. 0, 2/2006

XGATE Library: Using the Freescale XGATE Software Library

by: Steve McAslan
MCD Applications, East Kilbride

1 Introduction

The Freescale S12X family introduces true multi-core capability to the 16-bit microcontroller market. Its unique architecture allows simple partitioning of an application between real-time driver activities on the XGATE and algorithmic and control tasks on the CPU. This document describes Freescale's standard approach to providing driver software examples for the XGATE, and explains how to select and use drivers from the library.

Table of Contents

2 Library Concept

Freescale intends the XGATE library to be a collection of useful and optimal drivers for a variety of different application needs. Conceptually, use of the library involves following three simple steps:

1. Selecting the appropriate driver from the library.
2. Configuring the driver parameters to meet the application needs.
3. Integrating the driver into the final application.

freescale™ semiconductor

Each driver is designed and documented in such a way that the user can easily complete each of the three steps.

To simplify driver selection, each driver comes with an application note that describes the driver functionality, the MCU resources required to use the driver (and whether or not these can be shared), and a specification of XGATE load, code size and data size. The drivers include standard real-time processing tasks and “virtual peripherals”, where the XGATE provides the functionality of a peripheral that does not exist on silicon. In the latter case, the interface with the CPU makes it appear as if a real hardware peripheral exists.

To allow easy configuration, the application note describes the static and dynamic variables used by the driver.

Finally, integration into the final application is helped by a clear description of the interrupt sources used by XGATE, if and how parameters are passed into threads, data coherency processes used by the driver, and CPU function calls and behaviors.

All of the library application notes share a common format when describing each element, and this allows users to quickly build up an understanding of the total resource demands of the drivers.

The remainder of this document describes the common format used to describe each driver, advice on approaches to integration, and guidelines for developing or modifying XGATE threads.

3 Library Format

Freescale designs the drivers so that they use common approaches to peripheral definition, performance specification, and documentation approach.

3.1 Peripheral Definition Format

Each MCU supported by the library has a dedicated peripheral definition file that uses a standard format for access to on-chip modules. In fact, this header includes references to a separate file for each module included on the MCU. In the case where multiple copies of a module are present on the chip, then the definition for the module is included only once, but referenced multiple times. This approach makes it simpler to have consistency across MCUs and allows the user to quickly locate the definition of a peripheral register.

In practice, then, the peripheral definitions are contained in multiple files that are included in the software project. Figure 1 shows the structure of the peripheral headers.

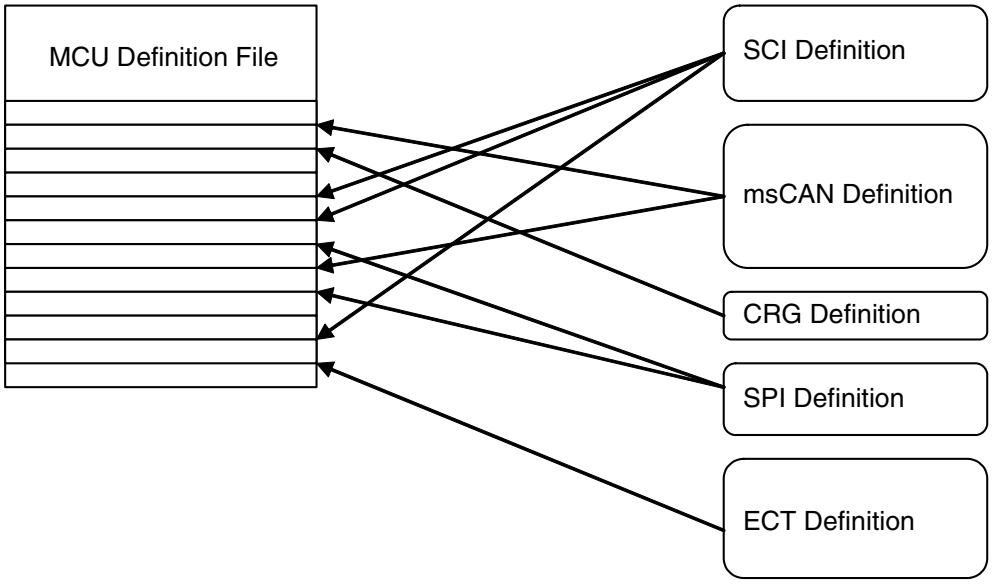

Figure 1. Peripheral Definition Concept

In this example we can see that there are multiple SCI, SPI and msCAN modules on this MCU and so the MCU Definition file refers to each of these more than once. The C implementation of this concept can be seen in Figure 2.

```
volatile tCRG     CRG           @0x0034;        /* Clock & Reset Gen */
volatile tTIMER   Timer         @0x0040;        /* ECT Module */
volatile tATD16   ATD1          @0x0080;        /* ATD */
volatile tIIC     IIC1          @0x00B0;        /* IIC Module 1 */
volatile tSCI     SCI2          @0x00B8;        /* SCI Module 2 */
volatile tSCI     SCI3          @0x00C0;        /* SCI Module 3 */
volatile tSCI     SCI0          @0x00C8;        /* SCI Module 0 */
volatile tSCI     SCI1          @0x00D0;        /* SCI Module 1 */
volatile tSPI     SPI0          @0x00D8;        /* SPI Module 0 */
volatile tIIC     IIC0          @0x00E0;        /* IIC Module 0 */
volatile tSPI     SPI1          @0x00F0;        /* SPI Module 1 */
volatile tSPI     SPI2          @0x00F8;        /* SPI Module 2 */
volatile tFLASH   Flash         @0x0100;        /* Flash Registers */
volatile tEEPROM  Eeprom        @0x0110;        /* EEPROM Registers */
```

Figure 2. Peripheral Declaration Example

The C code inserts the peripheral definition at the correct address for the MCU and reuses identical structure definitions for repeated modules.

Each of the variable declarations is a structure defined in the module definition file so access to the specific peripheral register is made through this structure. The definition of each of these structures varies according to the specific requirements of the module but in general the format will be as shown in Figure 3.

```
/* write value into peripheral */

modulename.registername.byte = value;


/* write value in bitfield of register */

modulename.registername.bit.bitnames = bitfield;
```

Figure 3. Format for Register Access

The upper assignment sets the peripheral register value to the whole byte given, whereas the lower assignment selectively sets or clears only the bits identified. Let's look at two specific examples of this approach.

Consider the case of the PIT module which has, amongst others, a counter register (PITMTLD0) and an interrupt register (PITTF). Figure 4 shows how a write to the counter register would be programmed and how flag 2 in the interrupt flag register would be cleared.

```
/* Configure counter register value */

PIT. pitmtld0.byte = 0x81;


/* Clear channel 2 flag */

PIT. Pittf.bit. ptf2 = 1;
```

Figure 4. Example of Register Access

In most cases it is possible to write to (or read from) a register in either format depending on which is the most convenient. Figure 5 shows an alternative approach where the code clears individual bits in the counter register and performs a write to the whole byte of the interrupt flag register.

```
* Configure counter register value */

PIT. pitmtld0.byte = 0x00;

PIT. pitmtld0.bit. _0 = 1;

PIT. pitmtld0.bit. _7 = 1;


/* Clear channel 2 flag */

PIT. pittf.byte = 0x04;
```

Figure 5. Alternative Example of Register Access

Obviously, the best way to use these definitions will depend on the practicality of the approach, the behavior of register and the efficiency of the code generated.

Of course, there is no requirement to use this register format for the application code but it is available for use if required. If you make any changes to the XGATE driver as supplied then you should use this format.

3.2 Resource Definition Format

Freescale specifies each driver using six standard parameters that provide a clear summary of the resources it requires. These are:

- Code size: the code size required to implement the driver measured in bytes.
- Data size: the total amount of memory used to store variables or constants.
- Maximum execution time: the maximum time that the driver takes to execute in its worst case scenario assuming code in RAM and XGATE having 100% access.
- Maximum latency: the maximum time that can elapse between successive runs of the driver assuming code in RAM and XGATE having 100% access.
- XGATE load: the worst case percentage of time that the driver uses the XGATE assuming code in RAM and XGATE having 100% access.
- Peripheral use: Identifies the peripherals required by the driver during its operation.

The architecture of the XGATE and the availability of these parameters will allow the user to easily manage the MCU resources. Since XGATE only executes code when a relevant event occurs in the system then in most cases the user can simply add the above parameters together to predict the worst case resource loading.

A fundamental contributor to the parameter values is the specification of the operating conditions under which the values were measured. For most drivers the XGATE load and latency will depend on how fast the peripheral is operating during measurement. For example a LIN driver would be capable of operating the SCI peripherals at a range of communications rates and the faster the rate the higher the XGATE load. In these cases, Freescale selects operating conditions that are representative of the typical maximum for the application. In the case of LIN the maximum defined baud rate is 20 kbps but a more typical maximum would be 19.2 kbps. In this example operating at 20 kbps will increase the documented load value and

decrease the latency. Much more common would be a slower LIN network at 9.6 kbps which would decrease the load and increase the allowed latency. Take these operating conditions into account when analyzing the published parameter values.

Let's consider an example where the software designer requires to use three different drivers on the MCU. There is an analog to digital converter driver, an msCAN driver, and an SCI driver that is used across two SCI modules. Table 1 contains example figures that are typical for these types of drivers.

Table 1. Example of Resource Calculation

Driver	Mem_{code}	Mem_{data}	t_{exec}	t_{lat}	Load	Peripherals
ADC	250	24	5µs/100µs	200µs	5%	ATD0
msCAN	200	144	4µs/100µs	500µs	4%	msCAN2
SCI	100	18	1µs/500µs	1ms	0.2%	SCI0
SCI	0	34	1µs/500µs	1ms	0.2%	SCI1
Total	**550**	**220**	**-**	**-**	**9.4%**	**-**

To calculate the total resource requirement for code memory simply add all the individual driver requirements. Note that the XGATE architecture allows simple sharing of threads across multiple peripherals so the SCI driver code is only loaded once. The memory space for data is simply the total for all drivers. The execution time does not have a sensible total since the execution time is for different periods for each driver. It is of course possible to normalize the execution period for all drivers and this will lead to a time value that is equivalent to the total load percentage. The latency total on its own is meaningless, however, you should use this value to determine if the XGATE could be overloaded by the drivers. For example if driver A has a latency of 50 µs but driver B has a maximum execution time of 60 µs then driver A may fail to service the event in time of driver B is executing at that time.

The load total gives an overall indication of the demands on XGATE. The larger this total then the more likely it is that XGATE may have difficulty servicing all system events.

3.3 Documentation Format

The standard documentation approach means that users can go straight to the definition of the driver without having to read through introductory text. An application note provided with the driver provides:

- A general description of the function
- The specification parameters as described above
- A detailed description of the driver function and architecture
- Specification of driver configuration (constants, variables, etc.)
- Specification of each driver function call

The documentation presents each of these in a standard format so that users can easily compare similar drivers and select the most appropriate for their application.

4 Initialization Options

The library provides two options to initialize the driver and peripherals used by each of the drivers. Each can be initialized either individually by the CPU in a function call or in a common thread provided on the XGATE and called by software interrupt 0. The second option allows the XGATE to remove the load of preparing on-chip peripherals from the CPU; however, it does require the user to maintain this thread by adding each driver manually into the thread.

Figure 6 shows a typical example of the process when the CPU performs the initialization for the XGATE driver. The driver includes a CPU function definition in its header files that must be called before the driver can operate. The call to the function will typically take place in the main() function immediately after start up.

```
main()
{
/* Initialize XGATE adc driver */
        xl_adcfilter_init();
…
```

Figure 6. Initializing the XGATE Driver from the CPU

When the XGATE performs the initialization the process is different. In this case each driver also includes an XGATE thread that performs the initialization task and each of these tasks is associated with the XGATE software interrupt number 7. Obviously, it is not possible to associate more than one thread with the same interrupt source so in practice the threads are either all called from a single thread or all the initialization code is copied into a single thread. Figure 7 shows how a single thread can call other threads while Figure 8 shows an example of the initialization thread when modified to contain the code from two drivers. The better approach to take depends on the preference of the software designer; however, the former method may make the code easier to maintain.

```
interrupt void xl_swint7(int features)
{
/* Initialize XGATE adc driver */
        xl_adcfilter_init();
/* Initialize XGATE sci driver */
        xl_scibuffer_init();
…
```

Figure 7. Initializing the XGATE Drivers from a Single Thread

```
interrupt void xl_swint7(int features)
{
/* initialize XGATE adc driver */
ATD0.atdctl2.byte = XL_ATD0CTL2;
ATD0.atdctl3.byte = XL_ATD0CTL3;
ATD0.atdctl4.byte = XL_ATD0CTL4;
ATD0.atdctl5.byte = XL_ATD0CTL5;
/* Initialize XGATE sci driver */
        SCI_DMA.pPort = &SCI0;
        /* set SCI baud rate */
SCI_DMA.Status = READY;
…
```

Figure 8. Initializing the XGATE Drivers by Embedding Code

5 Approaches to Integration

As with any software system, the software designer should take care when integrating the drivers into the final application. The first step should always be to clearly understand the capability of each XGATE driver. If there is any doubt about the driver's ability to perform the task intended then it is good practice to evaluate it on its own before beginning integration. For this reason, Freescale provides each driver in the XGATE library with a simple example that users can easily modify for their own purpose.

To integrate suitable drivers into a project, follow these typical steps:

1. Add the XGATE and CPU files to the source code project. In most cases these consist of C source files and header files containing driver and product definitions
2. Add the driver thread vector(s) to the XGATE vector table
3. Declare any shared variables. For example it is common for the CPU to pass a pointer to a buffer to the XGATE. The buffer itself must be created before this operation can occur.
4. Add calls to the driver interface functions. Typically the XGATE driver itself is completely event driven (by peripheral interrupts) but the driver initialization must be explicitly called. In addition some drivers have functions that execute on the CPU and these must be added to the CPU source code.

Finally, tests of the application should include appropriate steps to exercise the XGATE drivers.

6 Good Practice for XGATE Software

This section provides some helpful guidelines and tips when developing or modifying XGATE driver software.

The XGATE processor is fast and easily programmable and so there is a general temptation to write large and complex drivers that fully utilize its capabilities. However, this approach is generally only effective when XGATE is focused on a single task. When XGATE is handling multiple drivers and peripherals then the best approach is to create small focused drivers that run in a short period of time. This allows XGATE to minimize latency and respond to the highest priority tasks in less than 100 ns (at 40 MHz bus speed). The CPU can process the remainder of the task since its interrupt load will be handled by the XGATE.

Unlike the CPU, the XGATE is unable to process data and instructions that are not correctly aligned. Therefore take care when creating or modifying data structures so that pointers with misaligned data bytes are not created. Use the linker “force align” features to keep structures on word boundaries. In a similar way ensure that XGATE code is always linked so that instructions are on even addresses.

Always service the interrupt source inside the associated thread, even when XGATE passes the interrupt on to the CPU. If the interrupt flag is not cleared, then the XGATE thread will likely run again. XGATE can generally respond to interrupts more quickly than the CPU so it will begin processing the interrupt before the CPU has a chance to process the flag.

Use the on-chip memory protection features to ensure that the XGATE and the CPU do not corrupt each other’s code or data contents. In particular ensure that data areas with variable size cannot disturb the operation of the other processor. Examples of this type of data include the CPU and XGATE stacks and buffers with variable record lengths.

When debugging XGATE, be aware that software breakpoints will only operate once the XGATE code is copied into RAM. This may require stopping the application once the setup phase is complete and assigning the breakpoints at that time. This is because software breakpoints rely on replacing the actual XGATE instruction with a break instruction and setting a breakpoint before the code is copied will result in the break instruction being replaced by the instruction when the startup code executes. As an alternative use one of the hardware breakpoints provided in the debug module.

7 Summary

Use of the XGATE library can greatly simplify the task of bringing an application to market when using the S12X family. The library contains useful and efficient drivers to implement common tasks and create “virtual peripherals” and since the XGATE provides the processing power for these tasks the CPU can focus on the main application implementation.

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

AN3145
Rev. 0, 2/2006

freescale™
semiconductor