Freescale Semiconductor
Application Note

AN3224
Rev. 0, 3/2006

Tutorial: Introducing the XGATE Module to Consumer and Industrial Application Developers

by: Ross Mitchell
Systems Engineering Manager, Transportation and Standard Products Group

1 Introduction

A challenge for many embedded applications is to perform a significant number of tasks with very short latency times. Direct Memory Access modules (DMA) offer part of the solution by enabling data for interrupt sources to be read or written automatically via hardware control. DMA is limited in functionality, however, and usually only executes read or write commands before awaiting its next interrupt. Hardware in each module can be added to clear flags, but little else is possible. In embedded systems, such an interrupt event often involves additional logical process steps such as validating a signal or making a modification to the data before moving it to a final destination. Thus, an interrupt with DMA support often does half the job, and the CPU is left to interrupt the main execution routines to complete these tasks. Such interrupt handling takes CPU performance away from other functions and can often require critical timing of its own to ensure deterministic system operation, forcing complex software dependencies in the application.

The XGATE was born out of this need to greatly improve application responsiveness and coherency through a reduction in the interrupt loading on the main CPU, by allowing sequences of interrupt instructions to be executed in parallel with the normal CPU application execution.

Table of Contents

This product incorporates SuperFlash® technology licensed from SST.

2 What is the XGATE Used For?

The XGATE is thus targeted at performing fast interrupt handling, thereby reducing the load on the main CPU performing interrupt handling.

Most real-time embedded applications require many interrupt driven processes to service simple functions, often at high rates of execution. These are most often associated with human interface functions, actuator control feedback, and communications with other parts of the system. The XGATE is designed to take the load off the main CPU for these events.

One impressive feature of the XGATE is its general instruction set, which allows complex sequences to be developed. The XGATE is more than an intelligent DMA controller, as it provides the capability of a sophisticated I/O co-processor. There are limitations to consider, when developing applications using the XGATE in parallel with the main CPU12X core. These will be explored later; however, they are easily understood and do not impact typical functions for which the XGATE was designed.

The XGATE provides a degree of deterministic behavior for applications with high levels of interrupt activity, by off-loading several of these routines, many of which will be short duration tasks, from the CPU.

3 XGATE Concept

Interrupts from the interrupt controller hardware can be routed to the XGATE or to the CPU12X. Any interrupts routed to the XGATE will remove the load of that interrupt from the main CPU, and the XGATE will handle the entire interrupt routine.

As can been seen from the diagram in Figure 1, a switch directs the interrupt signals to either the XGATE or the CPU12X. If the XGATE is chosen, it executes the requested routine and, when this is completed, awaits the next request.

In Figure 1, we can see that there is a register to enable the XGATE for a specific interrupt, and the level of interrupt priority is set to one of seven levels. Any simultaneous interrupt request will be serviced according to the interrupt level — the highest is the most important and will be run first. These interrupt levels are the same for the CPU12X or XGATE.

The XGATE is a CPU in its own right. It is completely software programmable and is fully supported by an ANSI C compiler. It runs as soon as the interrupt source for the XGATE becomes present. After completing the interrupting task, it stops all its clocks again to await the next event, thus reducing power consumption.

The XGATE is a co-processor. It makes direct use of, and has direct access to, almost all of the memory-mapped registers and on-chip memory available to the main CPU. The XGATE’s most innovative feature is its unique way of interfacing to the MCU’s existing on-chip RAM. The internal bus of the MCU allows for interleaved access to the RAM on alternate bus access states, such that the main CPU, when running at full speed, accesses the RAM only half the time. The XGATE has access to the RAM on the other half of the bus cycle and in any cycle during which the main CPU does not access the RAM.

In fact, in a normal application execution, the main CPU accesses RAM much less often than every bus cycle, as it is usually fetching instructions from FLASH memory and reading and writing registers. The XGATE can read and write the RAM on typically seven out of every eight bus cycles.

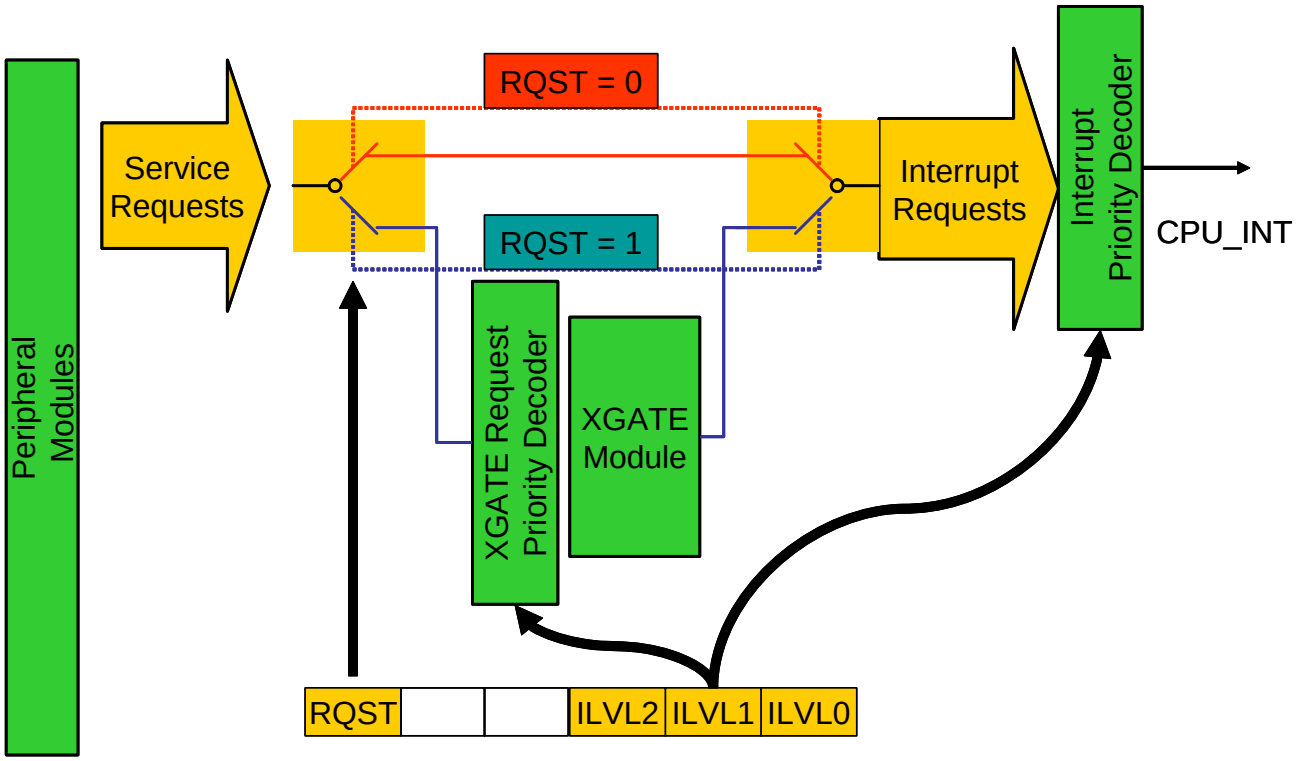

Figure 1. Interrupt Paths Using CPU or XGATE

You may now be thinking, “Why not run the XGATE out of FLASH memory?” We must consider the speed of access to different memory types. The RAM memory can respond to addressing of its memory and deliver the data much more quickly than FLASH memory. FLASH accesses are unable to keep up with the demands of an 80 MHz bus access, while the RAM and the MCU registers can handle this access speed. Running the XGATE using code executing in FLASH rather than RAM would reduce the XGATE’s performance by at least fifty percent. Furthermore, the CPU spends a lot of time accessing FLASH (but is not held back by doing this as it runs at 40 MHz); however, the XGATE would be held back every time it every time it had to access the FLASH memory. RAM is therefore a much preferred memory type to run the XGATE code from.

The XGATE is clocked at twice the bus frequency, and its performance is typically not impacted by the need to share the same RAM resources. However, it is important to realize that this processor is second in priority to RAM access, and the main CPU always wins. This affects determinism; however, this is quite manageable, and we will explain how this works in the next sections.

The schematic in Figure 2 shows how the XGATE is connected to the other microcontroller resources.

Here, we can see a representation of the XGATE interfacing with the memory and peripherals. The “p” connector in the diagram indicates access where the CPU has priority over the XGATE (except for port replacement registers), and where the XGATE has the normal bus access of 40 MHz max on the S12X. The “t” symbol represents access where the XGATE can run at twice the access rate of the CPU. The XGATE can interface to one of the FLASH memory blocks, the RAM, and the peripherals, but is not able to interface directly to the EEPROM on S12X devices. In practice, most of the memory addresses are on

the same bus, but the role of the XGATE is usually to manage the data flow of the peripherals, to deliver the results of the interrupt handler to a buffer or memory address in RAM, for subsequent processing by the main CPU.

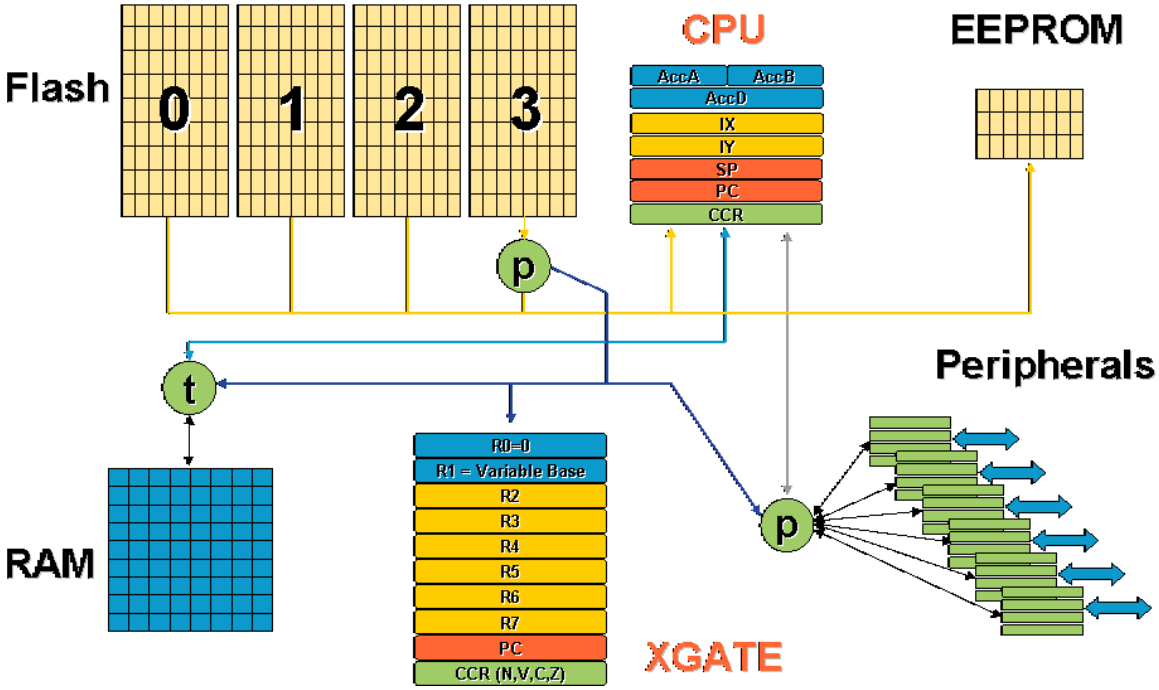

Figure 2. Data Bus Interfacing with XGATE, CPU12X, Memory, and Peripheral Modules

Let’s look at how we can best use this added functionality.

The XGATE runs at twice the frequency of the MCU bus and is, therefore, a CPU with twice the bus clock frequency. Most of the XGATE instructions take one clock cycle to complete, thus getting optimal performance from the faster bus access. The XGATE CPU also has optimized instructions and a register configuration to suit execution of very short routines with strong focus on efficient bit and byte manipulation, so its performance for these types of interrupt handling tasks can be higher than the host CPU. Indeed, in a like-for-like measurement of typical word, byte and bit manipulation operations, performance is around 8 MIPS (million instructions per second) for the CPU12X (40 MHz bus) and 13 MIPS for the XGATE (80 MHz clock) — the co-processor has better execution performance than the main CPU.

Now we have introduced a number of potential concerns for most embedded developers. We have two separate CPUs, with different instruction sets and different CPU architectures, running at different speeds, but using the same memory and peripherals.

This is neatly handled by a number of considered design choices.

The main CPU normally wins, and the XGATE always has to wait until access is free. This means there is occasionally some delay, which reduces the XGATE performance slightly. All code written for the XGATE should take this factor into account.

The use of C programming puts most of the architecture differences into the hands of the C compiler developers. The XGATE CPU architecture is focused on speed of execution and a small instruction set. The XGATE instructions have been developed to achieve this high speed, and to provide highly optimized implementation in C. For developers looking for that last tweak to the efficiency of the XGATE code, it is easily possible to insert XGATE assembly language into the code to force specific instructions to be used.

Since all the peripherals for the MCU are the same for both CPUs, it is simple to move C code for the S12X CPU to the XGATE CPU. In fact, this feature makes this very easy to develop XGATE routines, as the C code for both is almost identical — we will discuss the differences later.

So, we need some instructions to run, and we want these to run at the maximum XGATE CPU speed, so they should be run from RAM. Since the RAM is not permanent memory, the code is stored in FLASH memory and downloaded into RAM after reset of the MCU. To avoid corruption of the XGATE code after loading the code into RAM, the S12X allows this address range to be protected from writes by the main CPU after download, thereby removing the tiny risk of code runaway destroying the XGATE code. This memory protection feature operates in a similar manner to the FLASH and EEPROM protection.

In general terms, we now have the XGATE connected to the memory of the MCU, in much the same way as the main CPU, and able to run twice as fast as the main CPU when accessing RAM.

We should take a look now at how best to use it, and then, later, at some important things to avoid and to understand for the total MCU.

4 Virtual Peripherals

We have now a possibility to create functionality with the XGATE, and the I/O and timer peripheral functions, that could be considered equivalent to a dedicated hardware peripheral.

These “virtual peripherals” allow for enormous flexibility for the S12X MCU and can greatly enhance the flexibility of the applications, whilst taking advantage of the added processing power of this I/O co-processor.

4.1 Multi-Channel PWM

Pulse width modulation (PWM) on many I/O pins is strictly limited by the physical hardware of the MCU, or we must use software to drive these pulses. As we want a low-cost filter on the output, we usually want short PWM periods, and this can result in a very high loading on the main CPU, resulting from interrupt calls. The XGATE can remove this load completely from the main CPU, and can run the PWM generation for many channels from just a single hardware timer, thus making very efficient use of the MCU resources.

The following two implementation examples illustrate the performance of the XGATE for this type of functionality.

Example 1. Flexible routine delivering 117 channels of 8-bit PWM updated at 80 Hz, each with selectable port assignment, duty cycle, and period.

Here we have ten bytes of data assigned to each PWM channel to control port assignment, duty cycle, period, etc., and a very short routine that reads the data and executes the function defined by the data table. The routine is only 94 bytes in length and requires 58.2% of the XGATE performance to execute. Here a

full table for 117 LEDs takes a data space of 1170 bytes, yet the executable routine is just 8% of this size. This is a good example of the flexibility of the XGATE instruction set.

This routine has an effective interrupt rate of 2.4 million interrupts per second. In this case, the routine reads through the table at a rate of once every 49 µs, to maintain 8-bit resolution at 80 Hz, as the PWM is generated using software control based on a single timer compare. The routine takes 28.4 µs to check the 117 PWM channels for change of output state. This can be scaled down linearly for a reduced number of channels. Thus, for 100 channels of flexible 8-bit PWM control refreshed at 80 Hz, the execution time is approximately 24.3 µs.

In the second example routine, discussed below, the code was optimized for speed of execution, to reduce the time spent in the XGATE routine.

Example 2. One hundred channels of 10-bit PWM at 100 Hz, with grouping of outputs and a common period for maximum performance, is an alternate approach.

This routine has been optimized for performance and provides less flexibility than the former routine. It is therefore considerably longer at 1010 bytes, but requires only two bytes of data per PWM to set up, thus taking 202 bytes of data for 100 channels. The total load on the XGATE for this is 86%, but the resolution is four times better and the PWM rate is faster than in the first example.

This performance-optimized routine has an effective interrupt rate of 10.2 million interrupts per second, more than four times the performance of the flexible routine in the first example.

This routine takes 8.6 µs to execute 100 PWM channels of 10-bit resolution.

We can see from the PWM examples above that some trade-offs must be made in the XGATE code, as is the case for any CPU. However, if we want low latency times so that other routines can run, we must reduce the time taken by the XGATE to execute code and complete the interrupt handling routine. Balancing the use of RAM is also an important consideration. If 24 µs is acceptable for the other routine's latency, the much shorter first algorithm could significantly reduce the amount of RAM required, and could be important if there is a limited amount of RAM available. However, most developers would probably opt for the longer, but more efficient routine, as this is relatively short and allows for future flexibility of the XGATE application code.

In both examples, the S12X CPU could be held in STOP mode, in which case the PWM will still run fully.

4.2 Serial Communication Protocol Handler

Other functions typically seen with the XGATE are serial communications. Here we have messages being received, with identifiers to be compared, then data to be moved to an appropriate area in memory, determined by any matching of identifiers and payload data.

The XGATE routines are usually longer for this type of operation, as it may be necessary to search for and compare data, and then to search message destination tables to direct the distribution of the payload data. A full CAN implementation for an XGATE attached to the msCAN module may check 100 or more identifiers, but it can handle this in the time between separate messages from the CAN buses, and still have performance to spare. In this case, a typical 10 ms CAN frame period for one low speed CAN bus gives plenty of time for the XGATE to handle large tables. As the CAN message handling routine can be complex, this will often take longer to execute than typical interrupt routines, thus establishing the worst case latency for all other interrupts.

CANOpen is a good example of enhancing the performance of a standard peripheral. The msCAN module has only three transmit and five receive buffers, whereas CANOpen requires a minimum of sixteen of each. Software can extend the CAN capability, but the CPU12X will use much of its application performance to create this additional functionality, and this will limit the performance available for the main application code.

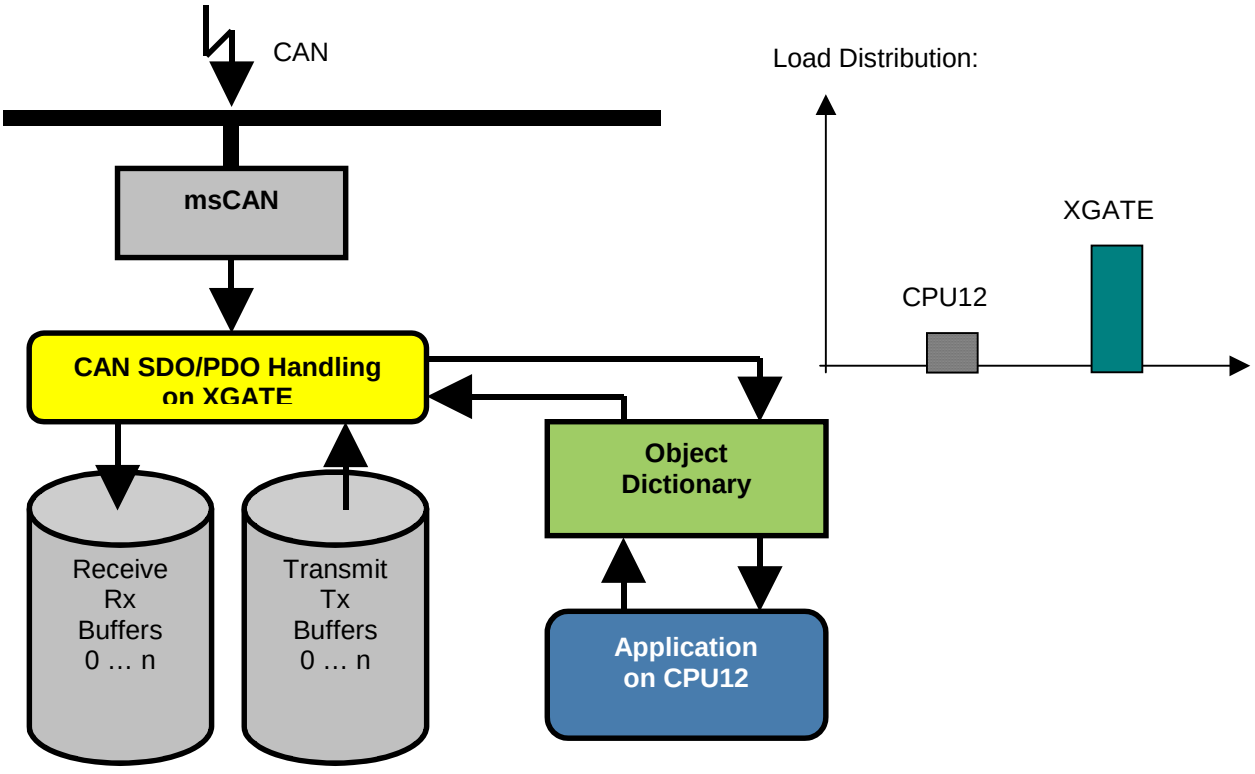

Figure 3. Sharing the CANOpen Application Load with the XGATE

Figure 3 illustrates the CANOpen functionality using XGATE to increase the number of msCAN buffers to meet the demands of CANOpen. The resulting loading on the CPU12X is greatly reduced.

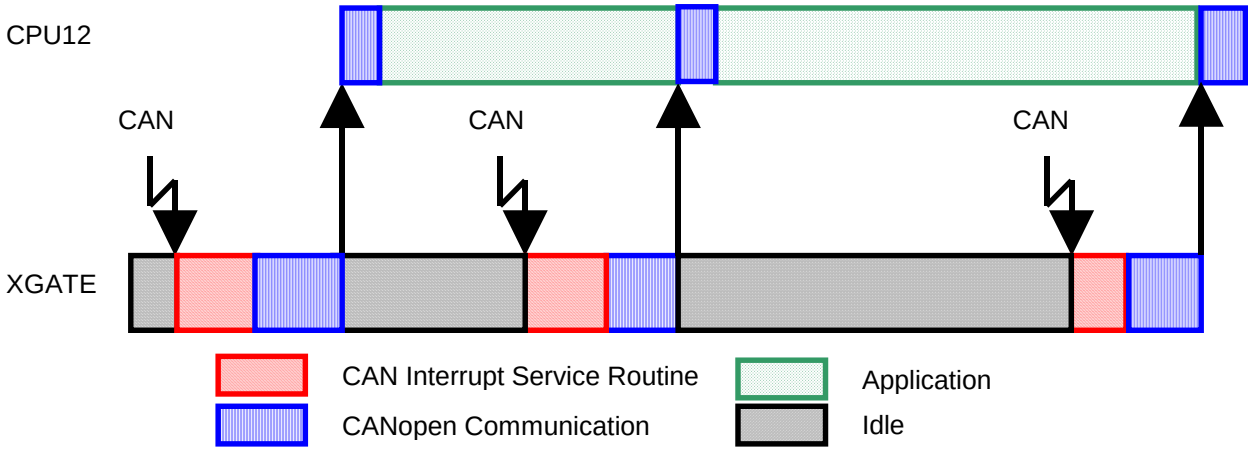

Figure 4. Releasing CPU Performance

To illustrate the point further, the schematic in Figure 4 shows the CPU12X activity (with time advancing from left to right). As the CAN message is received, the XGATE interrupt routine runs first, then the CANOpen routine follows to identify the packet and place the data into the appropriate memory buffers for access by the application code. The code running on the CPU12X has all the time between CAN interrupts to run the application, and leaves handling the interrupts to the XGATE. This can result in a significant improvement in application code performance, when the CAN messages are arriving, on average, once every 100 µs (the theoretical maximum for high-speed CAN). Applications that can benefit from this include real-time industrial applications, such as those driving automated equipment, where closed-loop control systems require fast system responses. Without the XGATE, a much faster CPU would be required, usually at significantly higher cost.

4.3 CAN Gateway

The CAN gateway can be seen in industrial systems — typically in the form of a two CAN network topology. The msCAN module found on Freescale 8-bit and 16-bit MCUs has limited filtering capability, and large distributed industrial systems can have large numbers of message identifiers.

The XGATE can provide 100% filtering via a look-up table. For CAN messages arriving every 100 µs (eight bytes per message and a 29-bit identifier), this loading for two CAN buses is significant for 8-bit or 16-bit MCUs. The XGATE can reduce this to approximately zero load for the main CPU, with just 258 bytes of code. For a CAN message of six bytes of data, this can be around 3 µs per CAN message for the XGATE code — a loading of just 6%, worst case.

4.4 Quadrature Decoder

Many motor control applications require continuous input from position and velocity sensors. A quadrature decoder may provide position feedback and can generate tens of thousands of pulses (interrupts) per second, for small, rapidly spinning electric motors. This would be an ideal XGATE function, as it can handle the interrupt event on each of the two sensors, to provide direction and position changes, with no load being placed the main CPU. A third sensor can provide an index signal for the rotational reference point. The task of reading the two sensor inputs each time there is an interrupt, and

performing simple forwards/backwards computation based on the subsequent states of the inputs, takes little time when using the XGATE (the routine takes less than 0.5 µs to execute), and is an ideal example of an XGATE function.

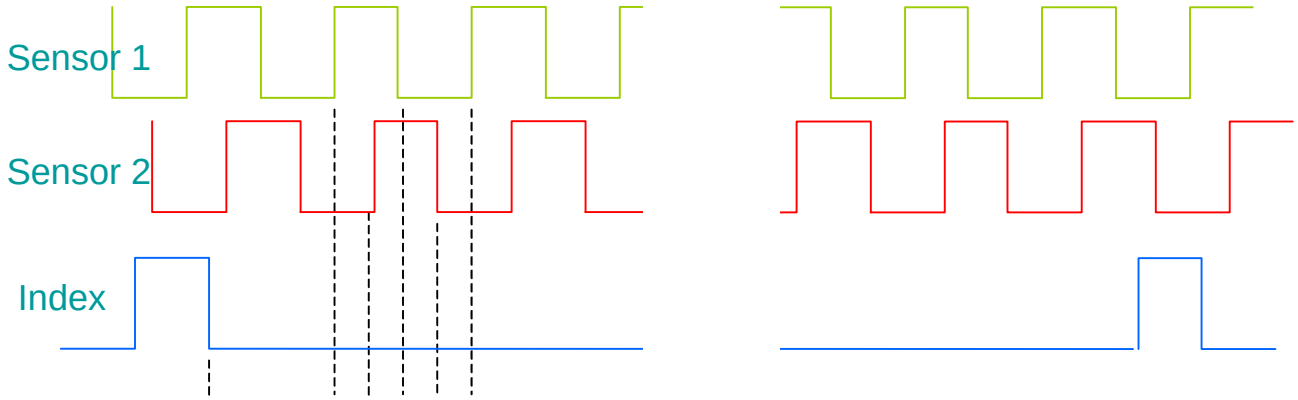

Figure 5. Quadrature Encoder Signals

As can be seen from the diagram in Figure 5, the interrupts are frequent; however, there can be some latency in checking the input pins, as long as the highest input rate is met. For motor control, this can be high, so it is important to check the allowable latency of such a routine, if you plan to run several routines on the XGATE.

4.5 Synchronous Serial Communications

The XGATE can provide a low-level driver function for many serial communications.

SPI operation in master mode is easy to achieve, as the clock and data are fully driven by the XGATE, and only a timer is required to provide the baud rate.

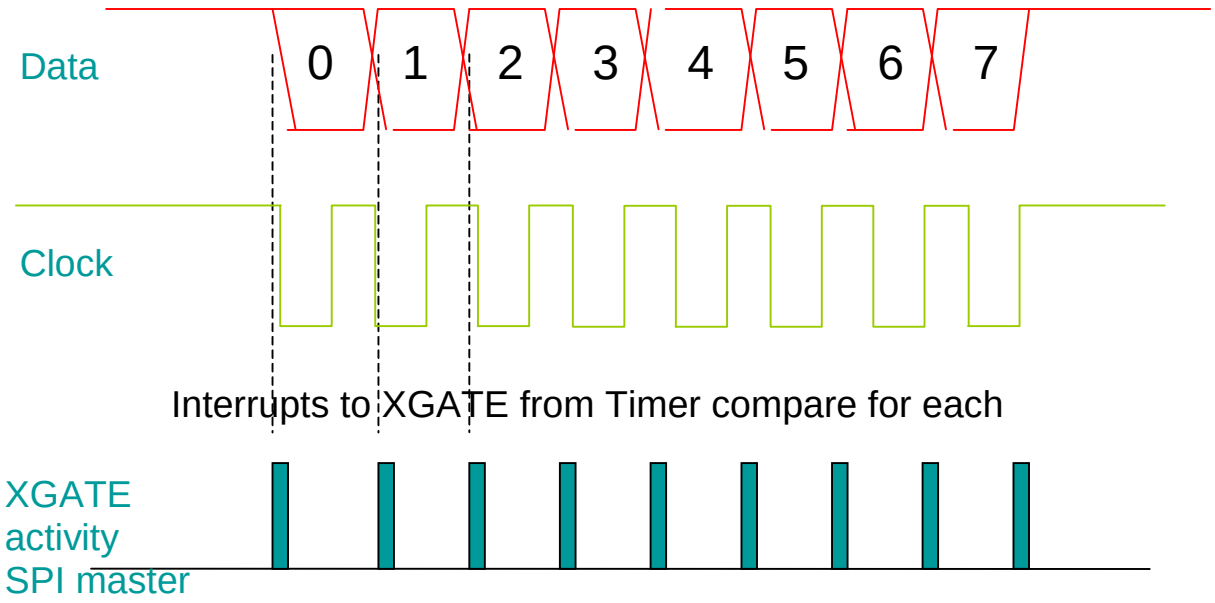

Figure 6. Serial Peripheral Interface (SPI) Master

In most cases, the master can deliver slightly varying clock periods in this protocol, with no impact on the slave provided the minimum time between clocks is maintained. Thus, another XGATE routine slightly delaying the SPI master routine is normally acceptable. However, this routine is usually run very frequently and may be significantly delayed by another XGATE function which could result in large variations in bit rates for this software SPI function. For many applications, the master is driving an SMOS product or dumb peripheral, and data rates can be well below 100 kBaud. In such cases, each bit can be handled separately.

Do be careful, therefore — typically, the SPI clocks run at 1 MHz, which leaves only 80 cycles for the XGATE, and it is easy for other routines to delay this by a significant amount. To avoid this, one possible approach is to send an entire byte without leaving the XGATE routine. This takes approximately 9 µs to complete, and would have a lot of dead time in the routine, which effectively wastes XGATE performance; however, if all the other routines can accept a worst case delay of 9 µs, this could be the best option.

Responding as a slave in this example takes more careful consideration, as it must react swiftly to each clock from the incoming SPICLK signal delivered by the SPI master. This defines the worst case latency, and again we need to know that the other routines will not interfere with this reception.

The XGATE can still easily manage SPI slave operation, as this typically involves collecting the data value on a port pin (MOSI) or outputting data on another port pin (MISO), each time a clock transition is received on a third pin (SCLK). Care must be taken for a slave SPI implementation, as missing a clock edge by half a cycle will generate erroneous reads and cause incorrect data to be transmitted back to the SPI master.

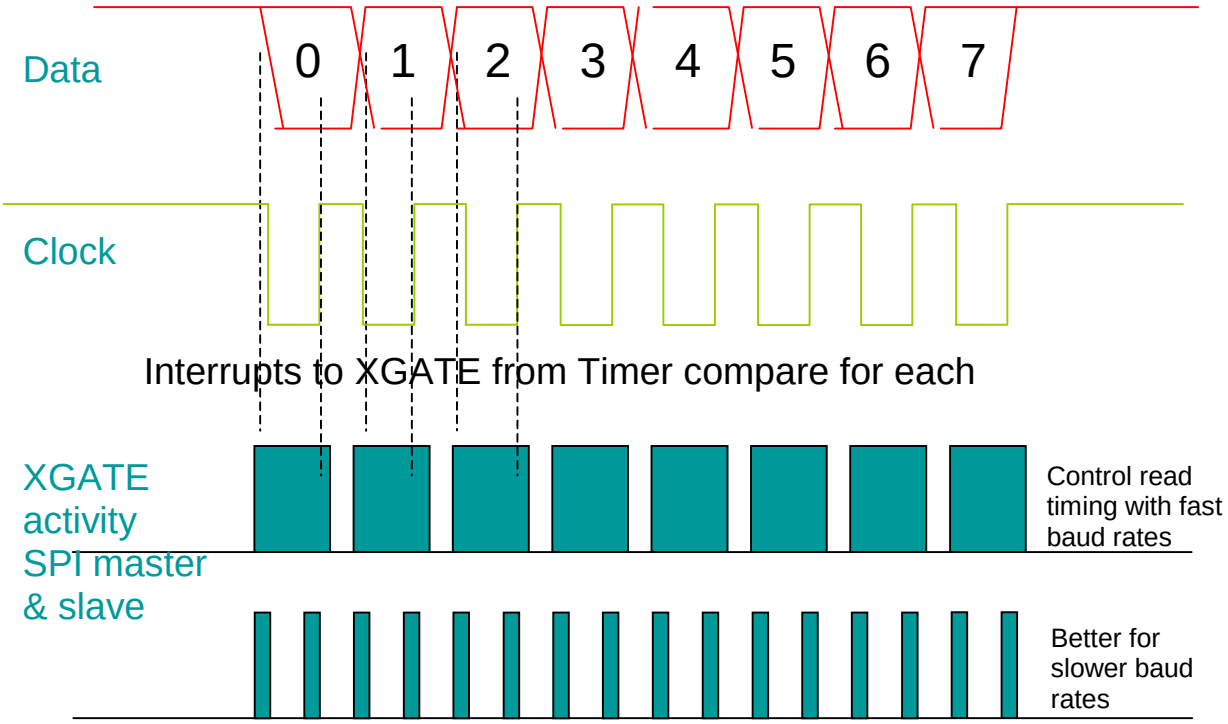

Figure 7. Serial Peripheral Interface (SPI) Slave

The diagram in Figure 7 shows two ways to deal with the combination of master and slave SPI. The lower part of the timing diagram illustrates the loading of the XGATE and shows that master and slave have events at different times. For a slow SPI, the lower of the two traces is acceptable but, for higher speed communications, the jitter effect of other XGATE routines, or even just the CPU common access to RAM, could cause enough delay to demand a different approach and require the master send and slave read to be one routine, to preserve timing. As we can see from Figure 7, for high baud rates, this effectively hogs the XGATE while transmitting and receiving data.

SPI slave operation is, therefore, very demanding for the XGATE, but it does demonstrate nicely some limitations of the co-processor that we can chose to accept and manage carefully to achieve our desired functionality. In most cases, a dedicated SPI peripheral is used with the XGATE, providing a queue mechanism and perhaps providing chip select functionality.

4.6 Asynchronous Serial Communications

Software SCIs or UARTs can be handled by the XGATE by “bit-bashing” the I/O ports directly, just as for the SPI. This is similar to the synchronous communications function described earlier but, of course, the timing of the edges for each bit is critical.

This imposes strict limitations on the XGATE code; however, the baud rates are usually a maximum of 19,200. With a need to remain within 20% of the baud rate for individual bits and 2% overall, this results in a worst case latency, either to set or to sample each bit, of 10.4 µs. This is the allowable variation in bit timing, such that there will be no impact on sending and receiving correctly eight bits of data at 19,200 baud.

The XGATE code to send SCI data needs two timer values that can be generated by a single timer compare: one for the bit period; and another for the break detection time, which is usually a multiple of the bit time. The routine to transmit a bit is very short, being required only to set the output state and increment the bit counter. Receiving the data requires the sample to be mid way through the timing for a bit period, which effectively doubles the interrupt rate as this would be another XGATE interrupt, separate from the transmit interrupt. Again, this routine is very short. As the worst case time between the interrupts (send/receive and 19,200 baud) is 26 µs, this allows for another routine taking up to approximately 36 µs (26+ 10.4 µs allowable delay), before the timing of the XGATE routine impacts the reception and transmission of SCI data at 19,200 baud.

This is a good example of how to allow for the other routine execution and meet maximum allowable delays when mixing XGATE routines.

There are a few other things to consider; and we will deal with these later, but this first pass should have given you good understanding of how much you can do with the XGATE in your application.

4.7 LIN Protocol Handler

The Local Interconnect Network (LIN) protocol is well known in automotive applications, and, due to its cost and deployment alongside CAN, is starting to appear in other non automotive applications.

LIN assumes a slightly modified standard UART or SCI. The main difference for a master node is the generation of a longer break signal than other asynchronous communication protocols tend to use. This is required to accommodate variable clock frequencies on the slave. The protocol has a simple frame format of: break, baud rate synch byte, identifier, payload data of two to eight bytes, and finally a checksum byte.

As the SCI peripherals are controlled byte by byte, the protocol benefits from the fact that the XGATE handles the interrupts and queues data for transmission and buffers received data.

The XGATE routine that executes the protocol is just 213 bytes long and executes in just 0.9 µs per byte. With two bytes of payload data, this works out at just 0.66% of the XGATE performance at 80 MHz for a two-byte LIN message.

4.8 Queue Management

Management of queues for peripherals can often be a significant load for the main CPU. Many MCU architectures opt for queues on the peripherals to reduce this loading on the CPU. The XGATE can provide this function with user defined flexibility, without loading the main CPU.

Typically, the search routines can take a while, so it is a good idea to try to find a way to reduce the execution times of these routines, to avoid lengthy latency times for other routines. Binary searches of ordered data can be very effective, as can splitting the task into a number of steps to avoid having a single long routine.

4.9 LCD Display Control

Direct driving of up to four back-plane LCD panels is possible with the XGATE; a typical example is shown in Figure 8.

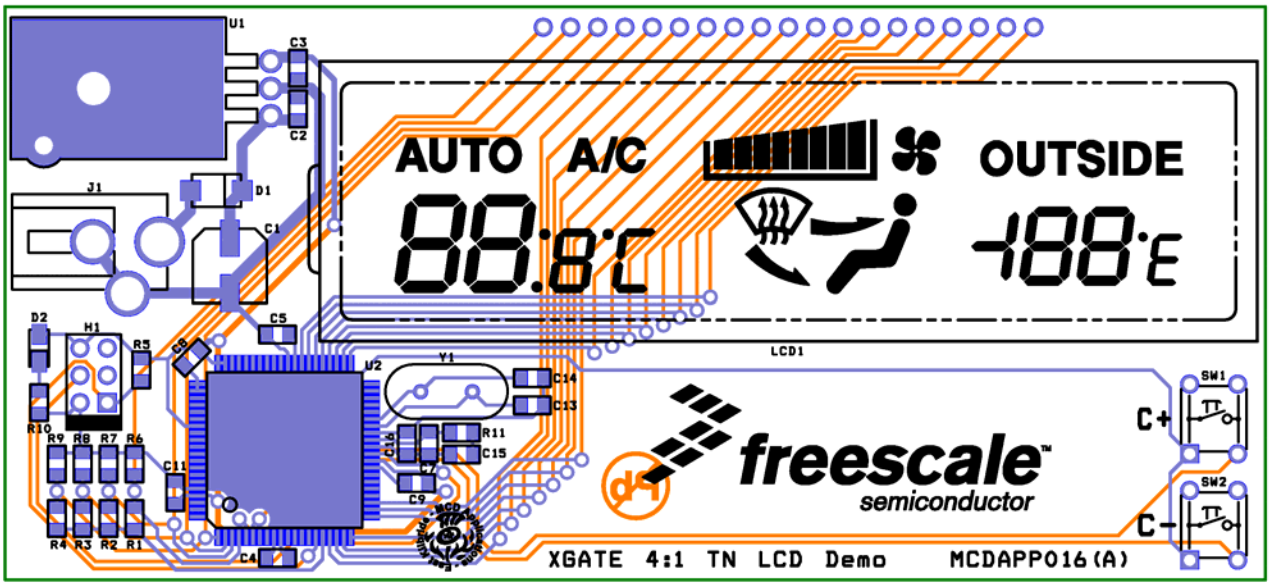

Figure 8. PCB Demo Board where the XGATE Controls the LCD Panel Directly

This LCD has 60 segments driven from four backplanes and fifteen front planes. It is refreshed at a frame rate of 64 Hz. The control algorithm allows for direct control of each segment (ON/OFF — one bit per segment), and contrast adjustment with 16-bit accuracy.

The XGATE also generates all the signals for ½ biased drive of the LCD

In this example, the XGATE average load is 0.098% (40 MHz bus) and the CPU is in STOP

This is an example of driving a display where the XGATE performance greatly exceeds what is required. This routine performs ten specific functions that generate eight transitions to maintain the four backplanes and two transitions to maintain contrast timing. Each update takes 1220 cycles to execute (15.3 µs execution time per display refresh) allowing 640 transitions to be performed per second, thus refreshing the display at 64 Hz. In this example, each transition routine takes just 1.25 µs to complete, making this a tiny and very short-lived loading on the XGATE. This is an excellent example of how the XGATE can be used to good effect and kept flexible for other routines to use the remaining performance.

4.10 Encryption Routines

The XGATE was not designed specifically for encryption. However, as long as the latency considerations are understood for long-running routines, the CPU is, in fact, quite a high-performance bit transposition machine and can work well for AES, DES and 3DES.

The XGATE has excellent bit manipulation instructions over a 16-bit field, making this highly efficient for algorithms such as AES. Even so, a run time of over 100 µs will probably cause major problems if the XGATE is to be used for other interrupt routines, as such routines usually require much shorter latency times, to perform correctly. Therefore, take care when performing long algorithms such as encryption, in case there is a direct impact on other functions running on the XGATE.

...breaking the AES into multiple stages can allow other interrupts to be handled.

```
for (i = 0; i = 8; i++)
     {
     enter Xgate AES algorithm for stage i;
     // Other xgate interrups can then run between each partial execution of AES.
     }
```

5 Coding for the XGATE

Converting an interrupt routine from the S12X CPU to the XGATE is quite straightforward.

A typical flow would be as follows:

1. Decide service by XGATE of CPU.
2. Decide interrupt priority.
3. Decide shared data allocation and sharing mechanism.

The C code must be directed to the XGATE C compiler rather than the main CPU; to make this happen, the XGATE C code files have a different extension, “.cxgate”, when using CodeWarrior. The linker handles the dual compiler control by recognizing the different extension for the filename.

To enter an interrupt routine, the XGATE uses two words, the interrupt vector address (as usual), and an additional word (16-bit) for a data parameter associated with the specific vector to be taken.

As the execution code is in RAM, we must take care to optimize usage of this expensive MCU resource. For a single virtual peripheral, there is no major impact; however, it is often desirable to create several of these virtual peripherals. One example would be to create multiple software PWM channels. When writing code for a software PWM on CPU12X, the developer would usually would usually opt for a separate routine in FLASH memory for each PWM output. This works well, as FLASH is a relatively cheap resource on the MCU. However, the XGATE is more efficient when running from RAM, which is a more expensive resource, so a different approach should be taken.

It would be better to have a single PWM routine and allow each PWM output to have data associated with it to uniquely select the specific I/O pin it is associated with, and the appropriate period and pulse width. For eight channels of software PWM, the data might amount to eight times four bytes (for I/O pin identification, period, and duty), but the RAM required to store the executable code would be much greater — probably, at least 100 bytes.

To deal with this kind of situation, XGATE has been provided with a feature that easily enables many sets of data to be assigned to a generic routine. The XGATE design includes a pointer to data in addition to the interrupt routine pointers. In other words, each time the code runs, it can load or store data differently, based on a separate table of addresses being defined for each interrupt vector. This has a significant effect when the routines are complex. This capability is illustrated in Figure 9.

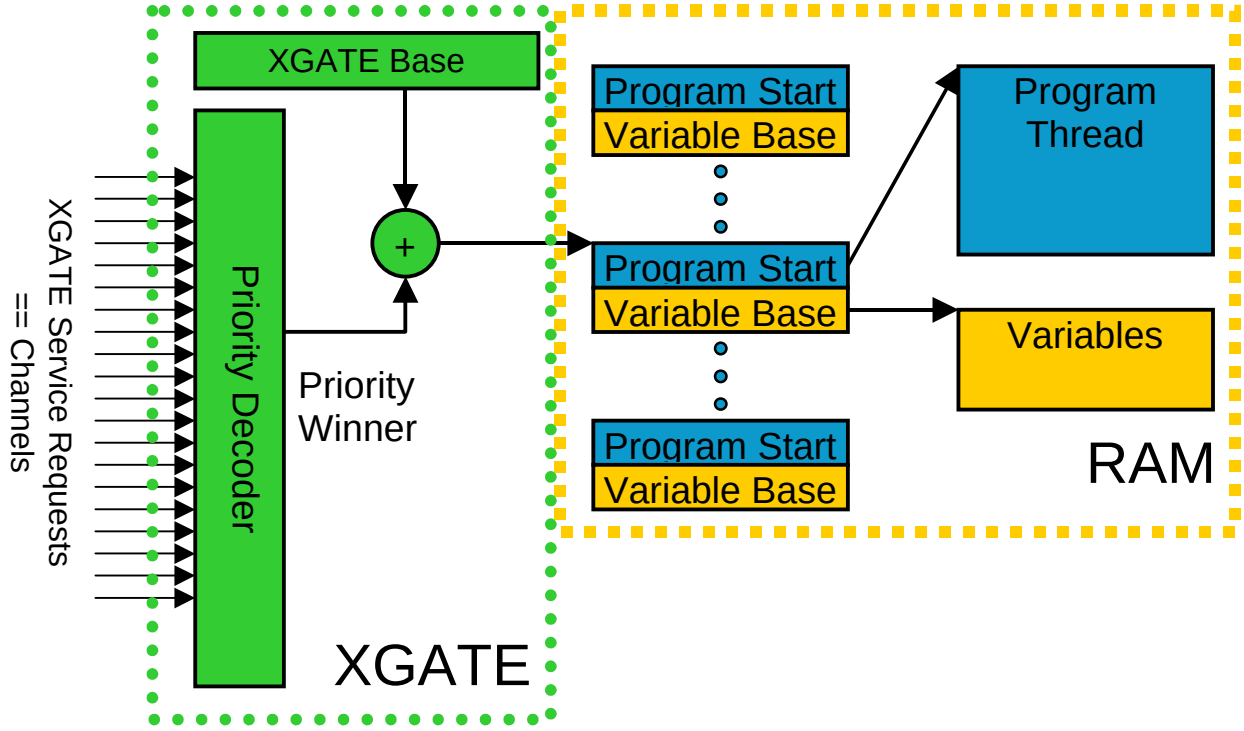

Figure 9. Vector and Data Pointer for XGATE Routines

The interrupt routines can be set with an individual data pointer and separate program start address for each interrupt. Because CPU12X does not have this capability, this is one of the few changes usually to be made to XGATE routines when converting from a routine originally written for the S12X CPU. As the XGATE is a single process CPU, the use of this scheme to reduce RAM does not cause any problems with coherency of data.

The XGATE code will usually be executed from RAM, but must first be loaded into RAM from the FLASH memory at MCU startup. Therefore, a small bootloader routine is also written to the FLASH, and the data packet of the XGATE executable routines is loaded into FLASH with this bootloader routine.

One final check...

Make sure the faster execution of the XGATE will not cause problems with the timing of the overall system, if you originally wrote the routine for the S12 CPU. Good coding practice, of not depending upon cycle counting to meet the timing requirements of the virtual peripheral, helps here. As the timers and other peripherals behave just as they do for the S12 CPU, this is not usually a problem.

Communicating with routines running on the main processor is essential for any dual processor system, and this is provided for by the inclusion of eight semaphore registers. The semaphore concept allows two independent tasks to share the same data elements in a controlled manner; task A takes control of the semaphore associated with the data, and task B must wait for task A to clear the semaphore before any access to the data is possible. In the case of XGATE, task A could run on the XGATE while task B could run on the main CPU; therefore, both tasks could be running simultaneously, and would require this hardware interlock to avoid incorrect modification of the data.

The semaphore hardware registers have two significant states: reserved for XGATE, and reserved for the main CPU. The receiving task must wait for the register to be cleared before proceeding and will not be able to change the semaphore register state, thus allowing the software to use this state to determine whether access is or is not allowed to the shared data. This allows safe sharing of data between the CPU and XGATE, in either direction.

In fact, most programmers choose to use software semaphores, which operate in the same way and offer flexibility to the developer. Here, double buffered data can be used to share data, and flags in software indicate the “ownership” of the data and completion of the other function.

And there we have it — the XGATE code is ready to run.

6 Performance Expectations

The XGATE has only a 150–200 ns (operating at 80 MHz) delay on any interrupt handling — this is driven by the hardware. After this, the XGATE executes the code in the appropriate RAM location.

On return from the interrupt routine, the XGATE is immediately able to execute another interrupt, with no delay. The XGATE does not stack its registers, or have to restore them, to service interrupts. This can lead to a twenty cycle improvement in interrupt handling, compared to the CPU12X, and is a considerable advantage of XGATE when handling interrupts.

With a typical access to RAM of 85% of the time (the main CPU has priority) and with an optimized instruction set, for typical uses of XGATE, functions can run up to 4.5 times faster than an S12X CPU. More typically, the XGATE runs at twice the speed of the CPU12X.

Some performance examples are given in Table 1.

Table 1. Performance Examples

Function	Typical Size (Bytes)	Data Size (Bytes)	Typical Execution Time (µs)	Percent of XGATE Performance
100-channel, 10-bit PWM with common fixed period of 100 Hz & static port selection	1010	202	8.3	86%
100-channel, 8-bit PWM at 80 Hz refresh, flexible periods for each channel	94	800	24.3	49.7%
LCD – 60 segments, 64 Hz refresh, four back planes, contrast adjustment	262	26	1.25	0.098%
LIN SCI per byte	213	14 per channel	0.9	0.17%
CAN gateway message routing, messages every 1 ms	258	6 per CAN message + ID table	3.0	0.3%

7 XGATE Running Several Routines

The XGATE has one very important consideration for the programmer — once a task has started in the XGATE, the routine runs until the end of the routine is reached, defined by the “Return To Scheduler”

(RTS) instruction. Thus, we must always be mindful of the possibility that some code will not execute immediately after an interrupt, as an executing XGATE routine must finish before any others can start.

For many industrial applications, a typical way to consider XGATE performance is to allow a time slot for all the operations to complete in, and then to measure the time for each routine to complete. Adding these together will show if the routines can all complete within the allotted time. If they can, the system will be deterministic. Any spare performance available will be obvious.

Table 2 shows an example of running a number of code segments on the XGATE.

Table 2. Example of Running a Number of Code Segments on the XGATE

Function	Total Time (µs)	% of 100µs Period
Gateway tic for 30Tx and 30Rx IDs	17.1	
Receive find frame ID	2.4	
Copy signal to message buffer (byte values)	1.9	
Frame transmit	0.7	
CAN buffer empty ISR	1.2	
CAN receive ISR (1 Tx frame generated)	8.8	
Total XGATE utilization for CAN comms (µs)	32.1	32
Process data object execution on 8 signals	25	25
32 x 10bit PWM channels at 100Hz period	2.7	3
Available resources in XGATE	40.2	40

As can be seen in this example, the CAN communications for two CAN buses and 30 messages takes 32% of the 100 µs time slot. This allows for other routines to run — in this case, the CANOpen PDO execution — on a maximum of eight objects per message (one byte each) and 32 channels, for 7-bit (128 levels) PWM, running at 100 Hz for LED brightness control.

This leaves spare performance of 40%.

It may be required to not fully use the XGATE performance, as latency times could be important. In the example shown in Table 2, the longest routine run time is 25 µs. This means that the worst case latency for the routines is at least 25 µs. Since each routine has a priority level, it could actually be longer than this if they all interrupt while the longest routine is running. Therefore, the lowest priority routine might have to wait until all the other functions have completed before getting allocation of XGATE CPU time. If the lowest priority routine is the PWM control, then it will have a worst case latency of 57 µs (sum of all the other execution times). An LED brightness control might cope with this latency, but not all functions will tolerate such a delay.

8 Debugging the XGATE Code

So now we have our code written and we wish to test it — but there are two processors, so how can we do this easily?

The IDE for S12X development tools has a debugger interface that can show both sets of CPU information. It sounds like a lot of information, but, as we can see from Figure 10, the details are easily viewed.

This CodeWarrior IDE debugger screen shot shows the S12X CPU on the left and the XGATE on the right.

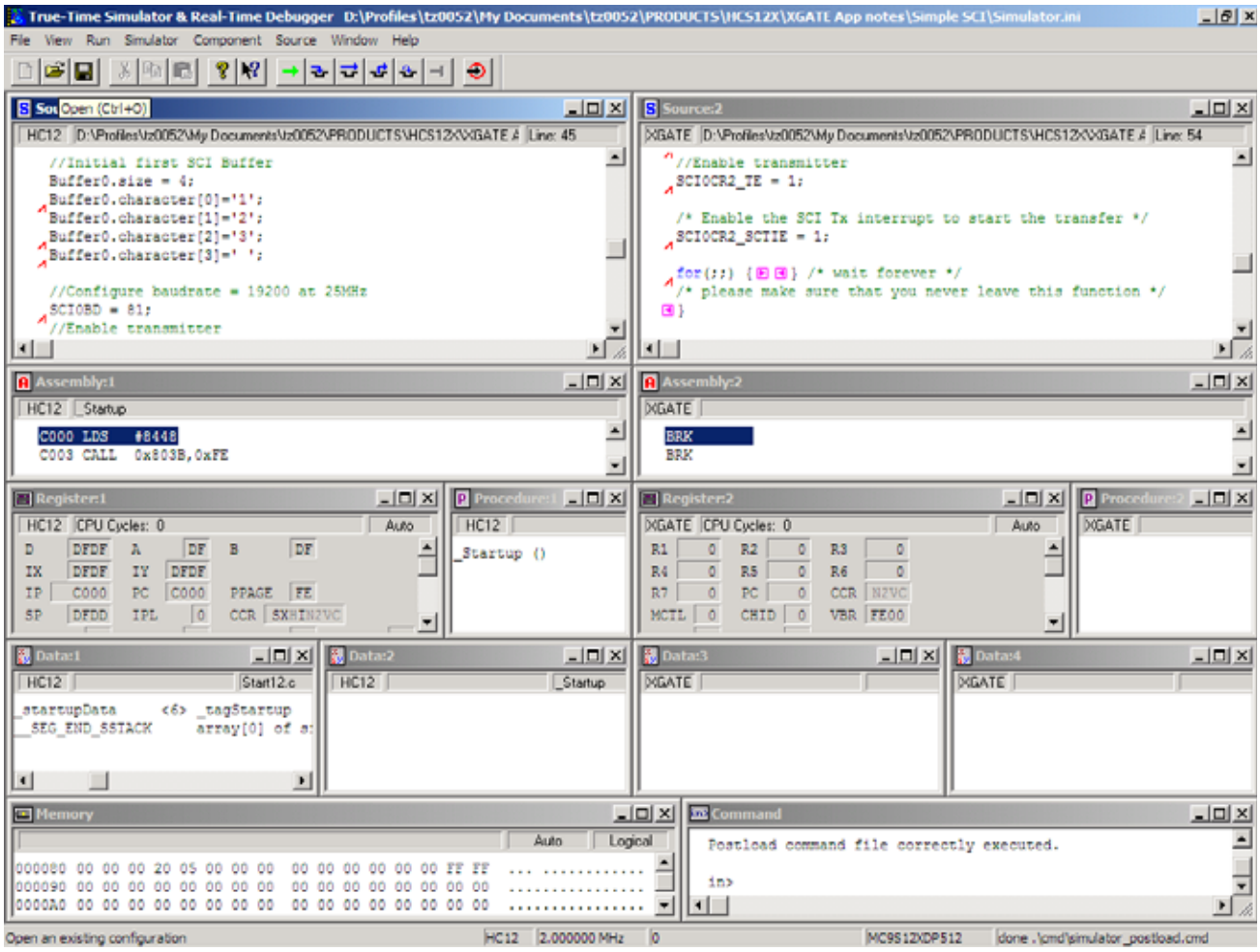

Figure 10. CodeWarrior IDE Debugger Screen Shot

The CodeWarrior tool and the Background Debug Mode (BDM) of the S12X handle the two processors transparently. The BDM interfaces fully to the XGATE in the same way as it manages the main S12X CPU. Thus, it is possible to set breakpoints and trace each processor independently and simultaneously.

The data rate for sending debug data to the PC can handle both processors and provides, very effectively, all the information you need to develop and debug your code.

9 Making the Most of the XGATE

The XGATE is best with small routines, because the initial versions cannot stop a routine until it reaches the end of its execution (RTI), with the exception of a brief halt while the main CPU accesses the same memory or peripheral as the XGATE CPU.

If we have long routines running in the XGATE, the worst case latency of any routine running in the XGATE will be at least as long as the time of the longest routine.

An example always helps...

Assume that the XGATE is running four timer input captures in a short routine of duration 80 cycles (1 µs), and there is an AES encryption algorithm running in approximately 8000 cycles (100 µs). If the AES algorithm is not running, the worst case latency is 1 µs (the previous interrupt has just started). However, if the AES algorithm has just started running, we must wait patiently for 100 µs for this to complete, before the timer interrupt can start. In many applications, this would be fatal in the system, and a second timer interrupt could easily have occurred with the second one lost (blocked because the first capture was not reset).

Thus, we should use lots of small routines in the XGATE.

What other issues need special consideration to make this work effectively?

One important timing feature of the XGATE to consider relates to bus collisions. Our code could be halted for some cycles whenever the CPU accesses the same memory as the XGATE. For RAM, this is a very short delay (usually) but, for registers, this could be more significant. If the CPU is performing a read-modify-write instruction, such as BSET which takes five cycles of the main CPU to complete, the XGATE would have to wait for the complete CPU instruction to finish. If we are unlucky, and it occurs when the CPU starts accessing the same address, there could be a maximum XGATE delay of ten cycles. We must take this into account when measuring the performance of the XGATE, as this could be a rare event and be difficult to observe. However, if the XGATE loading is high, or latency requirements of the other routines are critical, this could be very important for the application.

NOTE

There are several very specific situations that occur when accessing port replacement registers, where the XGATE has priority over CPU12X. See the data sheet for full details of how timing is affected for specific address ranges.

One solution is to avoid the same address accesses for XGATE and the CPU, where possible. This can usually be achieved for most functions, but is quite easily overlooked.

Now, what about power consumption, as we have a fast clock and a lot of logic?

The XGATE has clocks running in the module only when it is running. When there is no activity, the XGATE consumes leakage current only, which is almost insignificant compared to the rest of the MCU.

When executing code, it will take approximately the same current as the rest of the MCU, increasing the power consumption while executing code by approximately 35%. For a very heavy loading on the XGATE, we must take care to make sure there is enough heat dissipation at the worst case ambient temperature.

10 Combining Functions on the XGATE

Ideally, we should load up the XGATE with routines to use the CPU fully, and, as we have seen, there are some constraints.

Since the completion of XGATE tasks are usually time critical, latency of any other routines can be very important. Take care of this as a worst case scenario, as the longest latency could be a relatively rare event and not easily seen by observing the MCU functionality.

Next, we might want to check we have enough RAM and no conflicts with peripheral access.

The virtual peripheral software routines on the XGATE can make use of timers frequently, so, to avoid addressing conflicts, check that you have not mistakenly allocated a timer to the main CPU application code. As discussed earlier, bus arbitration can have a major impact on code execution.

Taking all factors into account, the average loading on the XGATE is usually in the range 40–60%.

11 What Products are Supported

Today we support the XGATE on the S12X family of MCUs. This family has a very rich feature set, as can be seen from be seen from the list of S12X family derivatives in Table 3.

Table 3. S12X Family Derivatives and Their Useful Peripheral Features

Device	FLASH / OTP (kB)	EE (Bytes)	RAM (kB)	Timer	I/O	Comms	A/D	PWM
MC9S12XDP512	512	32	4K	1 x 8ch ECT + 4 x 24-bit PIT	up to 119	up to 6 SCI, up to 3 SPI, up to 2 IIC, 5 CAN	2x8ch (112pin), 1x8ch & 1x16ch (144pin)	8ch 8-bit
MC9S12XDT512	512	20	4K	1 x 8ch ECT + 4 x 24-bit PIT	up to 119	up to 6 SCI, up to 3 SPI, 1 IIC, 3 CAN	1x8ch (80pin), 2x8ch (112pin), 1x-ch & 1x16ch (144pin)	8-ch 8-bit (7-ch 8-bit on 80QFP package)
MC9S12XDT256	256	16	4K	1 x 8ch ECT + 4 x 24-bit PIT	up to 119	up to 4 SCI, up to 3 SPI, 1 IIC, 3 CAN	1x8ch (80pin), 2x8ch (112pin), 1x8ch & 1x16ch (144pin)	8-ch 8-bit (7-ch 8-bit on 80QFP package)
MC9S12XD256	256	14	4K	1 x 8ch ECT + 4 x 24-bit PIT	up to 119	up to 4 SCI, up to 2 SPI, 1 IIC, 1 CAN	1x8ch (80pin), 2x8ch (112pin), 1x8ch & 1x16ch (144pin)	8-ch 8-bit (7-ch 8-bit on 80QFP package)
MC9S12XA512	512	32	4K	1 x 8ch ECT + 4 x 24-bit PIT	up to 119	up to 6 SCI, up to 3 SPI, 1 IIC	1x8ch (80pin), 2x8ch (112pin), 1x8ch & 1x16ch (144pin)	8-ch 8-bit (7-ch 8-bit on 80QFP package)
MC9S12XA256	256	16	4K	1 x 8ch ECT + 4 x 24-bit PIT	up to 119	up to 4 SCI, up to 3 SPI, 1 IIC	1x8ch (80pin), 2x8ch (112pin), 1x8ch & 1x16ch (144pin)	8-ch 8-bit (7-ch 8-bit on 80QFP package)

12 Where to Get More Help and Information

Application notes are available to help you understand typical examples of XGATE use. Today, these focus on automotive communications interfaces such as CAN and LIN. However, the list of general purpose routines is growing, and it is well worth checking the Freescale web site for new examples and application notes.

12.1 Application Notes for XGATE Application Development

ID for Application Note	ID for Software	Description
AN2708		An Introduction to the External Bus Interface on the HCS12X
AN3015	AN3015SW	Using the XGATE for Manchester Decoding
AN3144	AN3144SW	Using XGATE to Implement a Simple Buffered SCI
AN2726	AN2726SW	MSCAN Driver for MC9S12XDP512 Using XGATE
AN2685		How to Configure and Use the XGATE on S12X Devices
AN2732		Using XGATE to Implement LIN Communication on HCS12X
AN2734		HCS12X Family Memory Organization
AN3145		XGATE Library: Using the Freescale XGATE Software Library
AN3219	AN3219SW	XGATE Library: TN/STN LCD driver (Driving bare TN and STN LCDs using GPIO pins)
AN3225	AN3225SW	XGATE Library: PWM Driver
AN3226	AN3226SW	XGATE Library: ATD Average

12.2 Web Sites for More Help

Freescale Semiconductor, Inc. for data books, tools and application notes on microcontrollers.

www.freescale.com/mcu

13 Summary of Features

This may be a good time to review what we have learned.

- The XGATE is an I/O co-processor that has access to the same peripheral resources as the main CPU.
- Low latency response and zero delay on exit from interrupts.
- The XGATE must wait for the main CPU to release access to any memory before it can access the same locations.
- The XGATE can best offer deterministic application behavior when it is used with short routines.
- The XGATE consumes power only when running.

- The XGATE can be used for many functions or routines normally run on the main CPU and offers greatly improved performance over a single S12X CPU solution.
- The XGATE is very flexible, with application uses that continue to develop as users explore the myriad of new opportunities provided by a low-cost dual processor MCU.

13.1 Acknowledgements

Many thanks to...

Daniel Malik, **Steve McAslan** and **Martyn Gallop** for contributions to this document.

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

freescale™ semiconductor