Application Note

AN2685
Rev. 0, 3/2004

How to Configure and Use the XGATE on S12X Devices

MOTOROLA
intelligence everywhere™
digital dna™

By Joachim Krücken,
8/16-bit Products Division
Munich

Introduction

The HCS12X microcontroller family offers many enhancements over the HCS12 family; principal among these is the XGate peripheral processor. The XGATE is a programmable core that operates independently of the main CPU, has access to all of the S12X peripherals, and features a RISC instruction set. This application note describes how to configure and use the XGATE.

The document begins with a discussion of the issues surrounding data coherency in dual core systems and how these can be addressed by software.

It goes on, in the following section, to provide some general information and advice on how to set up and initialize the XGATE module.

Finally, it describes the XGATE's various power saving modes, and provides a comparison of their capabilities and differences.

Data Coherency

Data coherency is the state whereby a set of data is seen as consistent and complete by any process that wishes to examine it. In practice, this means that access to the set is forbidden if the data is currently being changed and so is not internally consistent.
Example: One process enters some data into a memory area and increments a variable holding the number of entries, while the other process takes the data out of that memory area and decrements this entry count variable. A typical code sequence looks like the following:

1. Read the variable into an internal processor register
2. Increment or decrement the variable

3. Write the variable back to memory

In a multi-tasking system, an interrupt might cause a task switch immediately after reading the variable. The variable might then be read in the other task, decremented and written back, thereby causing the variable to end up in an inconsistent state. In single processor systems, such code sequences are typically protected by disabling and enabling the interrupt before and after the critical code sequence.
When dealing with any dual-processor architecture, this method does not work. Strictly speaking, almost every kind of peripheral has this kind of issue. For example, an SCI receiver fills the receive buffer when a new byte is received; at the same time, the CPU might read the receive buffer and get conflicting data. In the case of peripherals, in most cases special flags indicate whether data is available or can be transmitted.
In short, each time two or more processes can access the same resource simultaneously, special care has to be taken.

The XGATE and the HCS12X CPU access the RAM in a time-multiplexed way. Within one S12X CPU cycle (25 ns), the XGATE can either: access the RAM twice within 12.5 ns, if the S12X CPU does not address the RAM; or at least one time, with one access dedicated to the S12X CPU and the other to the XGATE. This nice feature enhances the throughput of XGATE as well as of the CPU, but the complexity of the data coherency problem is also increased.

This application note describes several application relevant techniques that can be used to resolve data coherency issues.

Simple Buffer Scheme

A simple way to exchange data between two processors and avoid data coherency problems is to use a buffer. The two processors must honour an underlying agreement that one side writes to the buffer only if the other side has flagged the buffer as empty, and the other side reads from the buffer only if it has received the full signal.

Transmit Buffer

Typical applications that can use this simple scheme are LIN transmission and SPI masters. The flow for a transmit buffer would be as follows:

1. CPU fills the buffer.
2. CPU signals the XGATE module that the data is ready for transmission. (This can be caused, for instance, by enabling the transmit request inside the peripheral.)
3. The XGATE module gets a transmit service request.
4. The XGATE module sends out the data to the peripheral.
5. Once finished, the XGATE disables transmit request of the peripheral.
6. The XGATE signals to the CPU (via an interrupt, for example) that the transfer is completed.

7. The CPU can now start again at step 1.

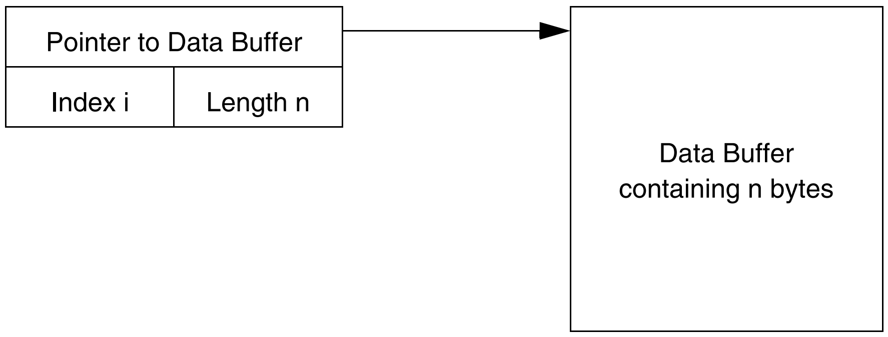

Figure 1. Transmit or Receive Buffer Data Structure

Receive Buffer

The flow for receiving data from a peripheral device and storing it into a buffer is very similar. However, care must be taken that no new data is received before the buffer is read. This is not an issue for a LIN node, or for SPI protocols, for example, as the sending of data is actually controlled by a transmit process, i.e. controlled by the CPU, itself. If the receive process runs asynchronous to the CPU activities, different methods must be used, as shown below. For receive processes, the same structure as for transmit can be used.

1. The XGATE services a receive service request from the peripheral.
2. The XGATE module fills the buffer until a predefined number of bytes is filled in or an end marker is received.
3. The XGATE informs the CPU (via an interrupt, for example) that the transfer is completed.
4. The CPU can then fetch the data from the buffer for further processing.

Guarding Technique

Another technique, not using semaphores, is to allow overwriting of the data. The reading side must set a “guard” before reading data, and verify that this guard is still set after the read sequence is completed. The advantage of this technique, compared to a semaphore based technique (see below), is that the process reading the data never gets stopped, and does not even need to block interrupts. The disadvantage of this is that, potentially, the reading of data must be repeated. Before a process starts to fill a buffer, the variable `Guard` is incremented. After the data is completely written into the buffer, the variable `Guard` is incremented again. Before the read process starts to read the data buffer, it creates a copy of the current guard (`OldG`). Assuming the variable `Guard` has been initialized to $0000, an odd number indicates that a write is just in progress, so the read process keeps polling the variable `Guard` until it becomes even, at which time it starts to read the data. At the end of the read process, the variable `Guard` is compared to the saved one, `OldG`. If they are not equal, then a write has occurred while reading, thereby corrupting the read data. In this case, the process must be repeated. **Figure 2** shows the flow.

This technique can be used in XGATE to S12X CPU or S12X to XGATE direction.

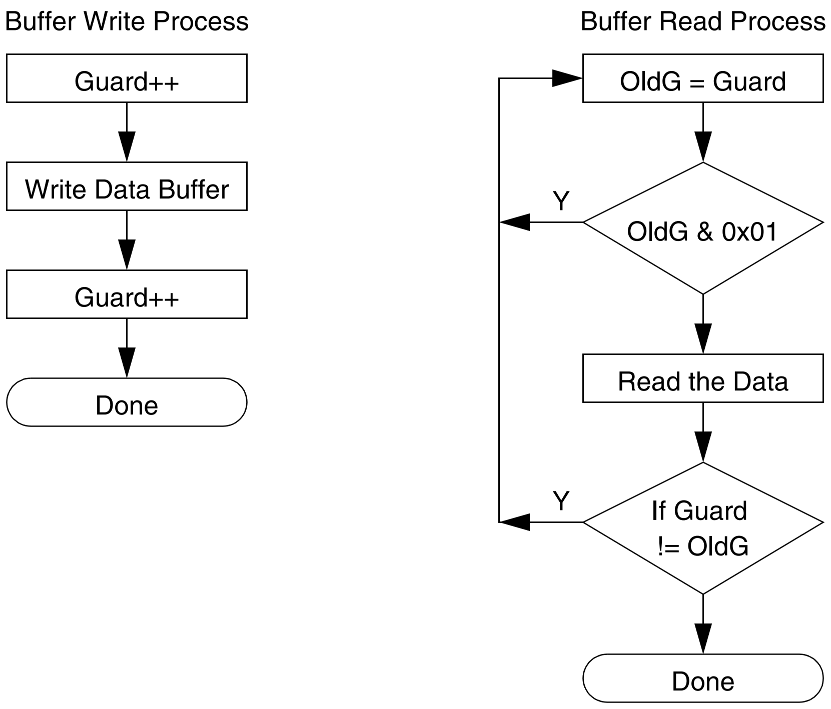

Figure 2. Flowchart Guarding Technique

Mutex (Mutually Exclusive Variable) or Semaphores

In many cases, concurrent access to the same resource is avoided by using a mutex (mutually exclusive RAM variable) or a semaphore[1]. However, for the S12X CPU and the XGATE module it is impossible to use a read-modify-write mutex or semaphore in RAM to indicate exclusive access to any resources, because it takes several cycles to read a RAM variable, test its contents, and write the modified variable back to RAM. Between the read and the write, the variable is “in transition” in an internal register of either the CPU or the XGATE module. If, now, the other module accesses the variable while being in transition, a write-back will result in an inconsistent state.[2] The XGATE

1. While the term semaphore is typically associated with a data type being able to count items, and a mutex can have only three states, the term semaphore is used here interchangeably with mutex.

provides a set of eight hardware semaphores designed specifically for this purpose. Such a semaphore can be in one of the following three states.

1. Unused, released
2. Assigned to S12X CPU process
3. Assigned to XGATE process

The transition between these three states is handled as follows.

1. Set semaphore.
   XGATE provides a dedicated instruction called SSEM with either a 3-bit immediate value or a register as operand. The carry flag is set if the XGATE could successfully lock the semaphore, and is cleared if the semaphore is already locked.
   The S12 CPU accesses the semaphore via the XGSEM peripheral register and requires a two step approach to set and check the semaphore, using the C-Macros shown below.
2. Release semaphore.
   XGATE provides a dedicated instruction called CSEM with either a 3-bit immediate value or a register as operand to release the semaphore. Again, the S12 CPU accesses the semaphore via the XGSEM peripheral register.

The hardware assures a clear priority, in case the two “get” commands are issued at the same time. The “release” command should be issued only by the process associated with the semaphore.

```
#define SET_SEM(x) (XGATE.XGSEM = 0x0101 << (x))
#define TST_SEM(x) (XGATE.XGSEM & 0x0001 << (x))
#define REL_SEM(x) (XGATE.XGSEM = 0x0100 << (x))

        do {SET_SEM(2);         /* try to allocate semaphore 2 */
          } while (!TST_SEM(2));/* out of a pool of 0..7 */
        /* Protected software region starts                  */
        /* now do a short piece of work as short as possible */
        /* End of protected software region                  */
        REL_SEM(2);

Same sequence in XGATE Assembler
LOOP1:  SSEM    #2              ; try to lock semaphore
        BCC     LOOP1           ; retry if locked
        ....                    ; protected software region starts here
        CSEM    #2              ; release semaphore
```

2. Why does this not happen in a normal multitasking operation system using semaphores or alike to guarantee exclusive access of one task to a shared resource? Most CPUs have an non-interruptible “read, test, modify, write” instruction so the variable is never “in transition”. The HCS12 must emulate this by using disable and enable interrupts, around the read-modify-write sequence, while the HCS12X CPU has a dedicated instruction (BTAS).

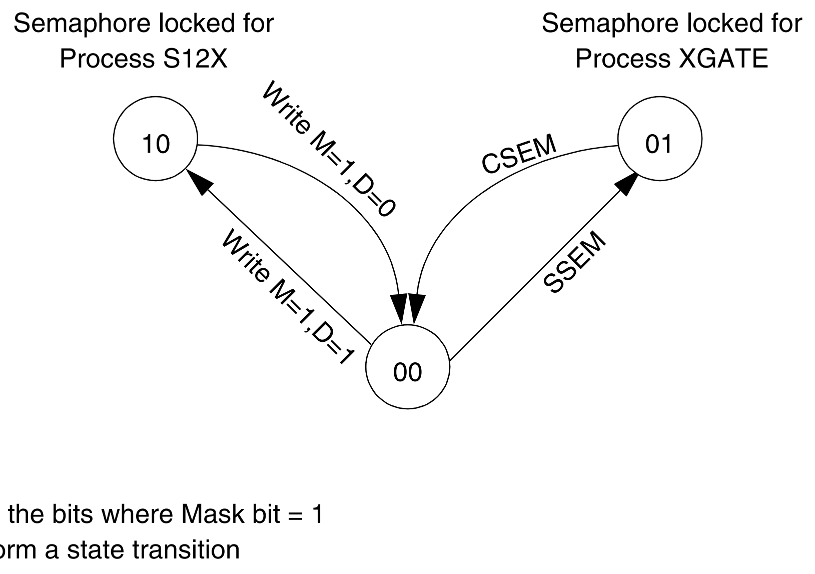
Semaphore locked for Process S12X
Semaphore locked for Process XGATE
10
01
00
Write M=1,D=0
CSEM
Write M=1,D=1
SSEM
Only the bits where Mask bit = 1 perform a state transition

Figure 3. Semaphore States

FIFO

A well known technique for de-coupling two asynchronous data streams is a First-In-First-Out (FIFO) data structure. A FIFO is most useful if a stream of data has to be received, where the individual bytes (or whole messages) are received in bursts, too fast for the CPU to handle, even though the overall performance of the CPU is more than sufficient to sustain the average data rate. By using a FIFO, the latency requirements of the CPU can be reduced significantly. A typical example is the 16550 UART found in almost every personal computer, which uses a 16-byte deep FIFO to buffer the incoming and outgoing data.

The main issue with any FIFO is that at least one common bit of information is required to be updated by the process filling the FIFO, and by the process draining the FIFO. This section shows a way to implement a FIFO. The basic data structure is shown in **Figure 4**. The index `putidx` shows the location where new items are written into the data buffer. The index `getidx` shows the location where items are read from the FIFO. In this example, a variable `num` is used, denoting the number of entries in the FIFO. While the difference between the `putidx` and `getidx` can be used to calculate the number of entries in the FIFO, using a dedicated variable holding the number of entries in the FIFO is typically simpler to implement in software.The variable `num` is incremented by the process filling the FIFO and decremented by the process draining the FIFO, and is, therefore, a resource shared by two independent processes. The example uses the mutex to protect critical code region as shown in **"Mutex (Mutually Exclusive Variable) or Semaphores" on page 4**.

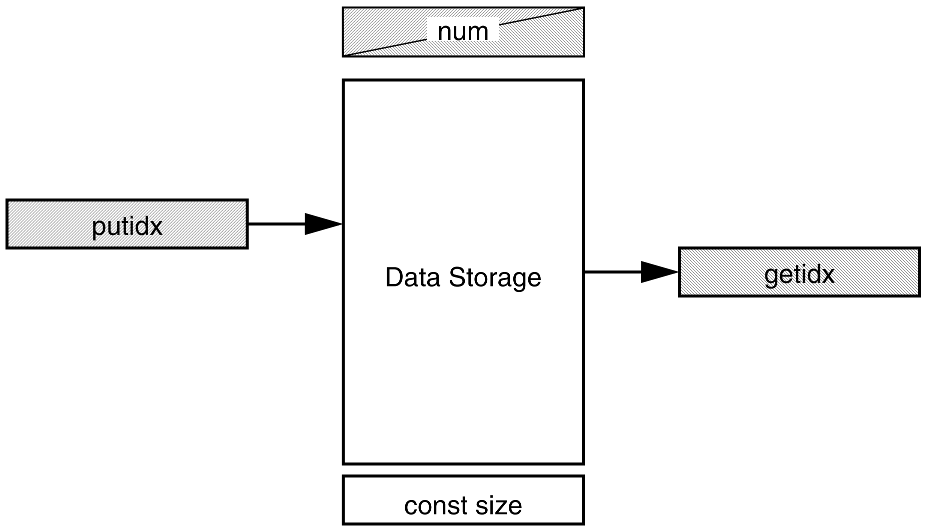

Figure 4. FIFO Data Structure

Receive FIFO

The following code example shows an implementation using byte wide entries and a maximum of 256 entries in this FIFO. It can easily be adapted to have a larger number of entries or more complex entries. The XGATE stores a byte into the FIFO while the CPU reads it out from there.

Putting Entries Into FIFO by XGATE

```
; Entry Conditions for this code sequence
; R1 points to data structure
; R2 to hardware (not shown here)
; R3 points to FIFOs data space
; R4 contains the byte to be stored in to the FIFO (also not shown how its loaded)
        LDL     R5,(R1,#num)     ; get number of entries in the FIFO
        LDL     R7,(R1,#size)
        CMP     R7,R5            ; check if there is space left
        BHI     SPACEOK          ; at least one item space left
; overflow error can be raised here
; further checks can be made here for high watermarks typically for XON/XOFF protocols
SPACEOK LDL     R6,(R1,#putidx) ; get the index where to put this in the buffer
        STB     R4,(R3,R6+)      ; Store byte to buffer an increment put index
        CMP     R6,R7            ; check on overflow
        BLO     LBL1
        CLR     R5               ; cyclical increment
LBL1    EQU     *
        STB     R6,(R1,#putidx) ; bump the next store index
; since the variable items is also updated by the reading side precautions must be
; taken to avoid a clash
; try to set a semaphore
```

```
SEMLOP  SSEM    #FIFOSEM        ; try to lock it
        BCC     SEMLOP
        LDB     R5,(R1,#num)    ; get value again in protected region
        INC     R5              ; since it might got changed in the mean time
        STB     R5,(R1,#num)    ; update length byte
        SIF                     ; signal to the HCS12 at least one byte is available
        CSEM    #FIFOSEM        ; unlock semaphore, leave protection
                                ; further activities like clearing request flag,...
```

Getting Bytes Out of the FIFO by the S12 CPU

```
; Entry:
; Typically the FIFO might be drained in an interrupt handler
; X register points to the FIFO control structure
; Y points to FIFO Data space
        LDAA    num,X    ; get number of entries
        BNE     ONEPLUS  ; at least one item in FIFO
; underrun error, should never occur
; further comparisons can be made here for a low watermark if e.g. the transmission has
; been blocked using e.g. XON/XOFF protocol
ONEPLUS LDAB    getidx,X ; where to pick data from
        LDAA    B,Y      ; get the byte item
; Content of Accumulator A must now be stored into the next level data structure
        INCB
        CMPB    size,X   ; check length
        BLO     LBL1
        CLRB             ; cyclical increment
LBL1    STAB    getidx,X
; now we need to updated the number of items left in the FIFO
; since this can clash with the filling side it must be locked using the
; same semaphore as above
SEMLOP  MOVW    #$0101<<FIFOSEM,XGSEM           ; try to lock semaphore
        BRCLR   XGSEM+1,#1<<FIFOSEM,SEMLOP      ; repeat if not successful
        DEC     num,X                           ; decrement number of entries left
        BNE     LEFTOVERS                       ; leave the interrupt asserted
        MOVB    #CLAERIF,XGIF                   ; clear interrupt in case last entry removed
        MOVW    #$0100<<FIFOSEM,XGSEM
 unlock semaphore
        RTS
```

Power Saving Modes

The S12X features some new ways to save power along with an XGATE I/O Processor.

To reduce power consumption, all clocks for the XGATE are turned off while the XGATE is waiting for a new thread to start. Consequently, a STOP instruction, such as on the S12 CPU, is not required.

From the overall system point of view, the four power saving modes shown in Table 1 can be classified into two distinct types: STOP and PSEUDO-STOP, on one hand; and WAIT and RUN, on the other hand.

The STOP and PSEUDO-STOP modes reduce the power consumption as much as possible, by turning off all internal clocks with the exception of some support modules.

In WAIT mode and RUN mode, a large portion of the internal clock distribution network (clock tree) stays alive.

Table 1. Power Saving Modes

Module	STOP	PSEUDO-STOP	WAIT	RUN
Voltage Regulator	Reduced Power	Reduced Power	Full Performance	Full Performance
Crystal Oscillator	Stopped	Reduced Oscillation Amplitude	Full Drive	Full Drive
VCO	Stopped	Runs if Clock Monitor detected a fail and self clock mode is enabled	Runs if PLLON and PLLWAI=0 or Clock Monitor detected a fail and self clock mode is enabled	Runs if PLLON or Clock Monitor detected a fail and self clock mode is enabled
PLL	Stopped	Stopped	Runs if PLLON and PLLWAI=0	Runs if PLLON
API (internal RC oscillator)	Runs if APIFE=1	Runs if APIFE=1	Runs if APIFE=1	Runs if APIFE=1
S12CPU	No Clocks	No Clocks	No Clocks	Clocked
XGATE	No Clocks	No Clocks	Clocked active	Clocked if active
RTI	Stopped	Runs if PRE=1	Runs if RTIWAI=0	
COP	Stopped	Runs if PCE=1	Runs if COPWAI=0	

Table 1. Power Saving Modes

Module	STOP	PSEUDO-STOP	WAIT	RUN
Other Peripheral Module	Stopped	Stopped	Most modules feature a lock Module_Stops_In_ Wait bit	Clocked (if enabled for sure)

STOP

The system STOP mode is entered if the CPU executes the STOP instruction and the XGATE is not executing a thread.

Wake-up Capabilities

System wake-up from STOP can be achieved via external inputs on certain pins. On the S12XDP512, these pins are: XIRQ and IRQ; pins on ports H, J, and P; and pins associated with the SCI or CAN modules, if these modules are enabled. Depending on the setting of the RQST bit in the interrupt module, either the S12 SPU or the XGATE is woken up.

The 9S12XDP512 features also a purely internal API (Asynchronous Periodic Interrupt) with an accuracy of approximately +-5% over temperature and voltage, once trimmed.

In the event of wake-up, the user has several current-saving choices.

1. Start right away using the minimum VCO clock (bus rate typically 1.5 MHz) to drive the internal clock system. In many cases, only tasks that are not timing-critical have to be performed (for example, reading a switch input). In the majority of these cases, no further action is required and the system can go back to STOP mode. In the case of an active event, the crystal oscillator can be started under software control. Once the oscillator is up and running, the SCMIF (Self Clock Mode Interrupt Flag) is set, and the internal clock system is switched to the output of the crystal oscillator. The software can then turn on the PLL to achieve a higher clock frequency.
2. Start the crystal oscillator Immediately, and wait for stable oscillation (SCMIF set) before proceeding. Typical crystal oscillator stabilization times are 5 –10 ms, depending on the frequency and quality of the crystal.

NOTE: *If the XGATE is woken up by an external event, it can wake up the CPU from STOP by raising an interrupt to the CPU.*

PSEUDO-STOP

The PSEUDO-STOP mode is very similar to the STOP mode, but it keeps the external crystal oscillator and the clock monitor circuit alive, if these are enabled. Additional options allow the RTI and the COP (Computer Operating Properly) Watchdog Timer to continue counting.

Wake-up Capabilities

The wake-up capabilities in PSEUDO-STOP mode are the same as in STOP mode, plus a wake-up via the RTI (Real Time Interrupt). Unlike the API, this RTI allows a precision timebase to be achieved, since it is derived from a crystal oscillator.

The COP can reset the system if it is not serviced properly.

WAIT

In WAIT mode, the voltage regulator is in full performance mode and most of the internal clock distribution network is kept alive. This will raise the current consumption significantly, compared to STOP and PSEUDO-STOP. Most of the peripheral modules feature a Module_Stops_In_Wait bit, allowing the clock to the module to be turned off in WAIT mode.

Wake-up Capabilities

The wake-up capabilities are the same as in PSEUDO-STOP mode, but most of the peripheral modules can raise an interrupt to either the XGATE or the S12 waking up the system.

The CWAI bit allows the clocks to the CPU and BDM to be turned off, thus preventing debug connections to the system in WAIT mode.

RUN Mode

In RUN mode, all enabled modules, including the core, are clocked.

General Setup Guidelines

This section provides a short explanation of how to properly configure and initialize an HCS12X system. It takes the form of a simple recipe; as with all good recipes, it can be adapted to the user's taste.

Initialize Stack Pointer

Firstly, interrupts should be disabled: CLI (just to be sure), and the initial stack pointer should be loaded. In most cases, on the S12X, a value of $4000 (top of RAM+1) is adequate. The interrupt vector base register, if available, can be initialized here as well. If the XIRQ is used as an emergency, non maskable interrupt, it can be enabled now. If the usage of the XIRQ is less critical, it should be enabled along with the other interrupts in **"The Final Countdown" on page 12**.

Initialize I/O Ports

Then, the General Purpose I/O ports (GPIO) should be initialized, to minimize glitches occurring on the output pins.

Initialize RAM Variables

If memory tests, such as basic RAM tests, or FLASH or EEPROM checksum tests, are required, these should be done before the RAM variables are initialized.

Initialize the RAM variables. This is usually done in the start-up routine of "C" (crt.o), even before the main routine is entered. (Note that ANSI-C requires that all static variables are initialized.) Some words of caution here...

1. Peripheral registers should be excluded from this, since some are write once, or the order of the setup is important.
2. If a RAM test is executed within the main routine, this could corrupt the initialization of variables.
3. If possible, code should not rely on automatic initialization.
4. If automatic initialization cannot be avoided, it should be done after the stack initialization.

Most compiler vendors deliver the startup routine source code, to allow users to adapt it to meet their needs.

Download XGATE Code

If the application makes use of the XGATE, the XGATE code should be downloaded into RAM. The RAM protection can be set after downloading, thus preventing an overwrite of XGATE code or S12 RAM variables. The XGATE Vector base register should be set afterwards.

Interrupt Module

Set up the interrupt module by writing the interrupt priority for each vector (if different from the reset condition). If service requests are routed to the XGATE module, the RQST bit of those interrupts must also be set. In addition, it is advisable to initialize the interrupt vector base register, even if the reset value ($FF) is valid for this application.

Peripheral Modules

Initialize all peripheral modules and set their local interrupt enable bits, if required.

The Final Countdown

Initialize and service the watchdog.

NOTE: *Some devices allow the watchdog to be enabled out of reset. In this case, care must be taken to ensure that the watchdog does not fire while executing the sequence above.*

1. Enable STOP instruction, if required.
2. Enable XIRQ, if required.
3. Enable the interrupt (CLI if XGATE is used; XGE=1).

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:
Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu
Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

ASIA/PACIFIC:
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

HOME PAGE:
http://motorola.com/semiconductors

MOTOROLA