Freescale Semiconductor
Application Note

Document Number: AN3419
Rev. 0, Draft A, 02/2007

System Integrity Techniques for the S12XE

by: Steve McAslan
MCD Applications, East Kilbride

1 Introduction

The system designer is responsible for the correct operation of a microcontroller-based system. If an operational breakdown causes bodily harm, the system integrity will likely be the most important aspect of the design. In other cases, cost or functionality may be higher considerations. Many aspects of an embedded system's operation can influence its operational integrity.

This application note examines how you can use the on-chip capabilities of the S12XE to address memory integrity, operational correctness, and correct software behavior. The S12XE MCU family introduces features that can improve operational robustness and allow early detection and prevention of potential hazards in the system.

Contents

2 Memory Integrity

The application software provides most of the functionality of the system. It is the aspect of an embedded system with the greatest influence on system integrity.

In most S12X applications, the CPU software and important system data are stored in the on-chip flash. To help maintain the integrity of the on-chip flash, the S12XE includes an automatic Error Correction Coding (ECC) system in hardware and a margin-checking system that checks how completely a flash cell is programmed or erased.

2.1 ECC System

The ECC system, always enabled, detects and corrects some errors that may arise in the flash. It is applied to the P-flash in groups of four consecutive words known as phrases. Each phrase is assigned a syndrome value that allows error correction of one error and detection of two errors across the phrase.

Each time its contents are read, the flash module checks the consistency of the data and the syndrome. If a single bit has changed, the error is corrected. This error correction is transparent to you and does not disturb the normal read process of the P-flash. The flash module also sets a single-bit error flag. You can choose to receive an interrupt if this occurs.

If while reading the flash the module detects that two bits are in error, it sets a double-bit fault flag. Because these errors cannot be corrected, the value read from memory will contain an incorrect value. To avoid this problem, the user software may enable an interrupt when the module finds a double-bit error. If the error word contains an instruction opcode, the interrupt prevents execution of the incorrect opcode and allows the CPU to take corrective action. The flash error detection and correction process is shown in Figure 1.

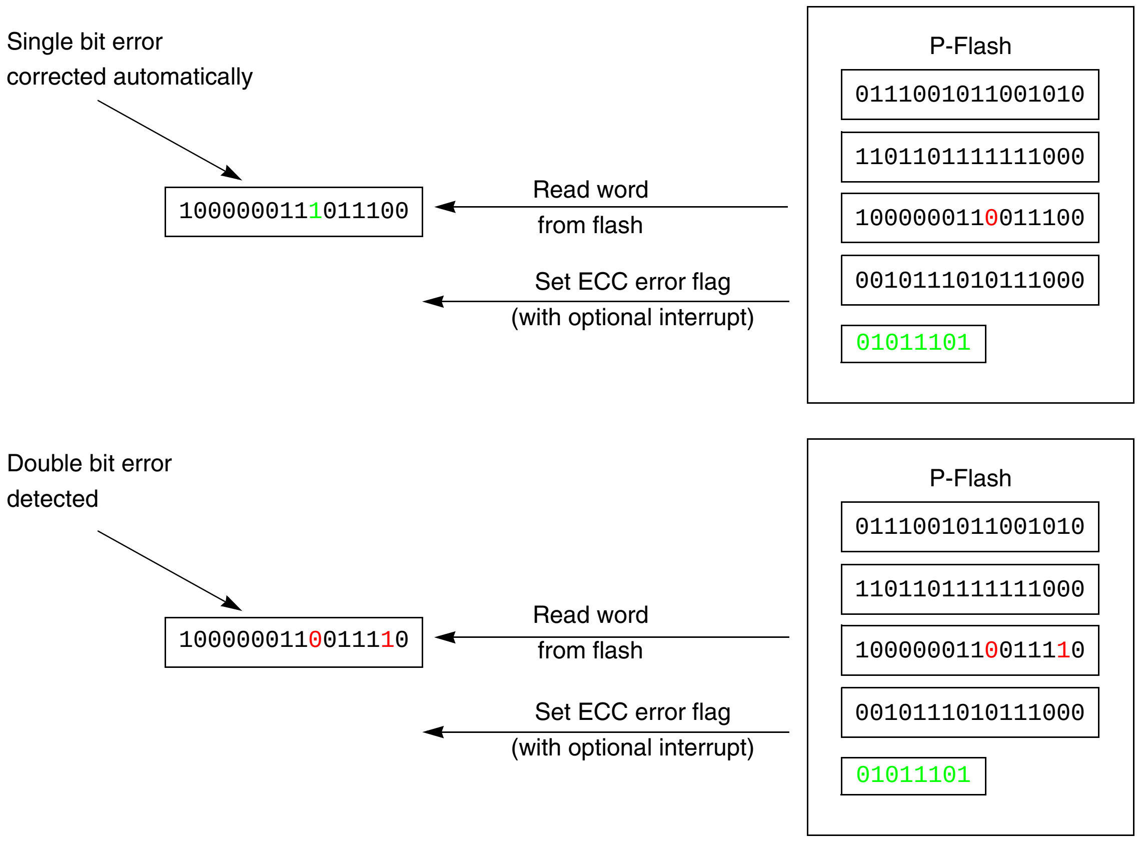

Figure 1. Flash ECC Process

The syndrome is automatically created each time the software programs a phrase in P-flash. The P-flash programming operation is shown in Figure 2.

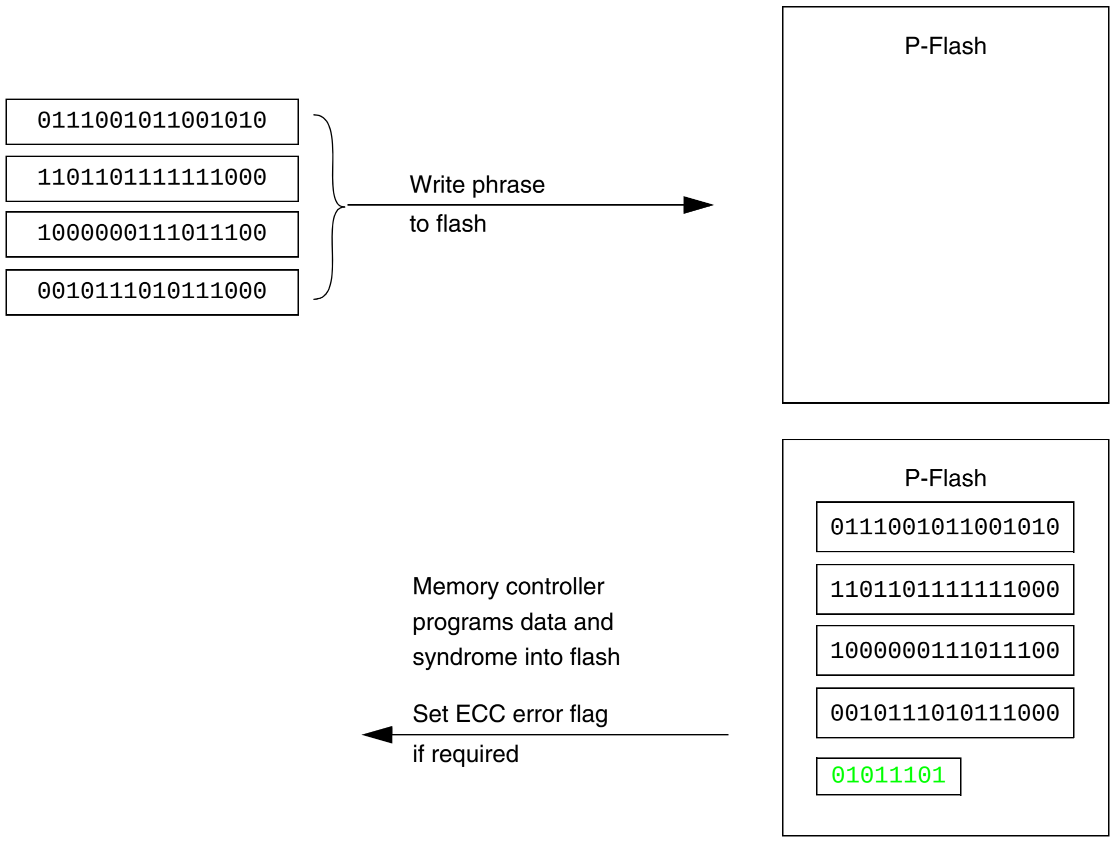

Figure 2. P-flash Programming Sequence

The on-chip D-flash also has an automatic ECC system, but each word in D-flash is protected by its own syndrome. The module detects single- and double-bit errors and these errors can cause interrupts if necessary.

ECC errors can be detected only if the software reads the flash. Performing periodic block reads of flash helps find these errors, especially in infrequently accessed flash. Because ECC is always performed on an eight-byte phrase, reading a single 16-bit word that spans the last byte of one phrase and the first byte of the next phrase performs ECC checks on all bits of the two phrases.

If an ECC error occurs, you can choose a response dependent on the nature of the error. For example, a system with a single-bit error may continue operating, while a system with a double-bit error may perform a graceful power-down.

2.2 Margin Checking

The S12XE allows a detailed examination of the actual programmed or erased state of each flash cell. This is achieved by setting the logic margin to a different value than that of normal operation.

The MCU determines the binary value of a flash cell by sensing the current through the cell. In normal operation, the MCU uses a measurement level that allows for the maximum discrimination between normally programmed or erased values. The charge values vary across a narrow distribution for a correctly programmed or erased cell. If the cell is marginally programmed or erased, the actual cell charge margin may be smaller than expected (see Figure 3). This decreased margin could be further reduced by normal variations of the MCU caused by voltage or temperature fluctuations, or by the gradual change in charge on the cell over the course of many years.

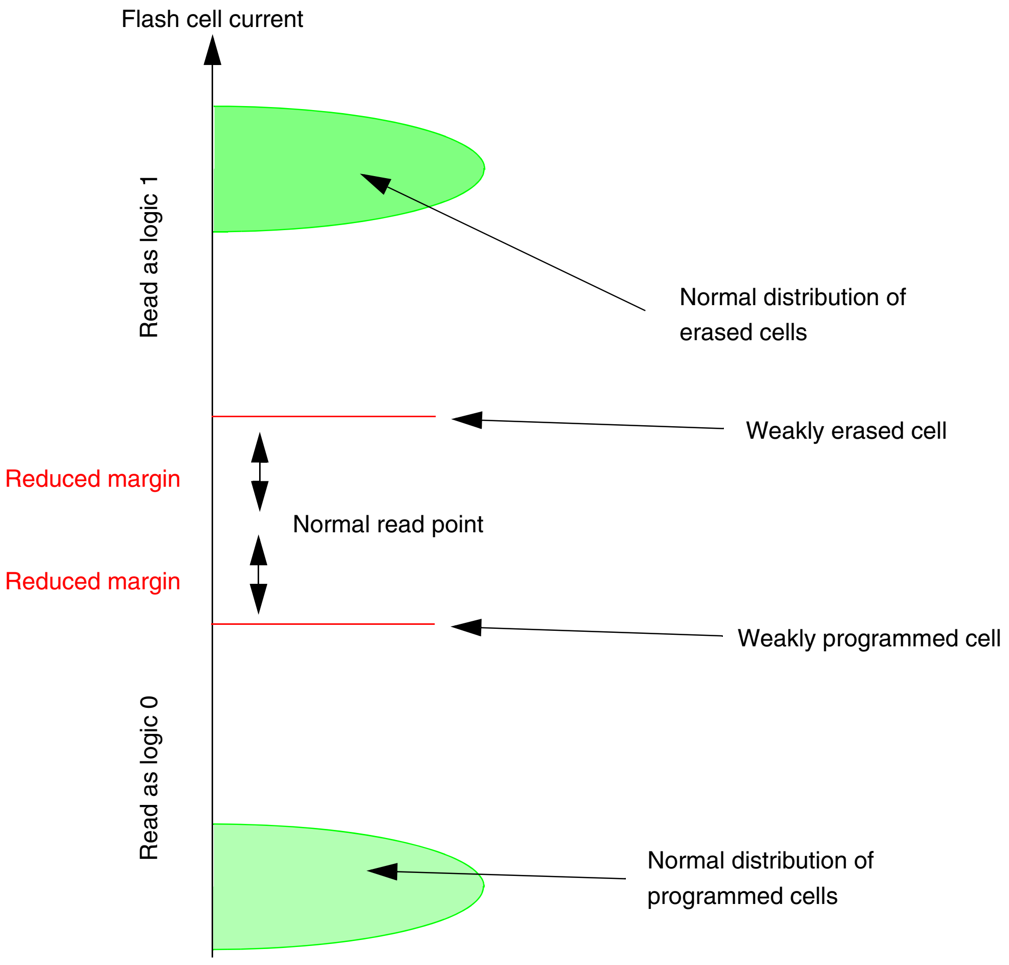

Figure 3. Effect of Marginally Programmed Cells

The MCU provides two additional measurement levels that allow for a more accurate examination of the cell's state. One of the two additional levels is closer to the programmed state (see Figure 4). This allows the user software to detect a cell with a smaller-than-expected margin, a margin that can be used to provide advanced warning of an incorrect reading. The erased margin level works in exactly the same way but towards the erased state. The alternative margin reads may be used in diagnostic routines that read the flash and ensure there are no ECC errors.

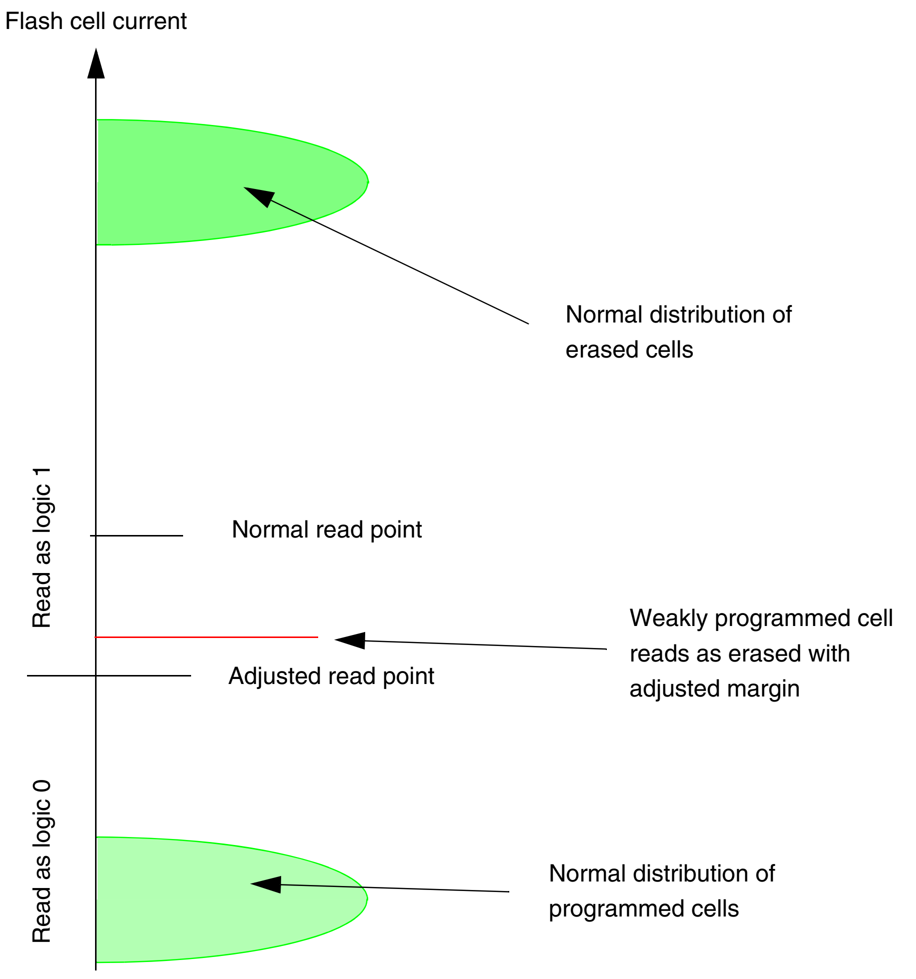
Flash cell current
Normal distribution of erased cells
Read as logic 1
Normal read point
Weakly programmed cell reads as erased with adjusted margin
Adjusted read point
Read as logic 0
Normal distribution of programmed cells

Figure 4. Adjusted Margin Level

3 Correct Software Behavior

In real world systems, it is generally impossible to prove that application software is error-free. This is especially true when multiple software components from several suppliers are integrated into a single device. Individual components may operate correctly on their own, but incorrectly when used together. Potential errors include modification of the private variables of another task, incorrectly calling another task, and changing the configuration of on-chip peripherals.

The S12XE provides a protection scheme that prevents software components from inappropriately accessing memory and peripherals. The scheme consists of restricted permissions on CPU operation and access descriptors in the memory protection unit (MPU).

3.1 CPU Permissions

In an embedded system the CPU executes software that initializes and executes the application while handling interrupt events. During initialization, the CPU needs to modify RAM contents and configure peripheral registers and interrupts. During execution of the application, the CPU should not modify or access unused parts of the MCU. The CPU needs to modify the stack and clear flags when responding to interrupts. In general, the CPU's required capabilities will vary according to the type of software being run.

The S12XE addresses these requirements by providing three operating states with different levels of permissions (see Table 1). The state is controlled by the U bit in the CCR and the SVSEN bit in the MPUSEL register.

Table 1. CPU Operating States

State	U	SVSEN	Restrictions
Supervisor	0	X	None
Protected Supervisor	0	1	Descriptor ranges only

User	1	X	Descriptor ranges only;
			No interrupt control;
			No low-power capability (STOP or WAIT)

3.1.1 Supervisor State and Protected Supervisor State

At reset, the MCU sets the CPU to Supervisor state and sets the MPU so that there is a descriptor for all memory. In Supervisor state, the CPU can access any resource on- or off-chip without restriction (as with CPUs in other S12 and S12X devices), making it an ideal state for initialization purposes and for use with kernal and interrupt tasks.

The CPU can enter Protected Supervisor state by setting the SVSEN bit in the MPUSEL register. This prevents the CPU from accessing areas of memory that have not been specified by the MPU; but the CPU can change this bit at any time and is not subject to other limitations. This state may be useful for providing protection in kernel or interrupt tasks.

3.1.2 User State

When the CPU sets the U bit, it enters User state. This state limits the access rights of the CPU as well as its ability to change its state, the U bit, the I bit or X bit (in the CCR) via any opcode, execute the STOP or WAI opcodes, or access areas of memory not specified by the MPU. The only way to return to Supervisor state from User state is to receive an interrupt. User state is used during the execution of standard application code, such as user tasks. After the CPU is in the appropriate protected state, the MPU controls the access permissions to resources on- or off-chip.

An interrupt causes the CPU to return to Supervisor state. In this state, the CPU can perform activities necessary for an interrupt such as modifying interrupt flags and copying data from peripherals. When the CPU returns from interrupt, the U-bit returns to its pre-interrupt condition. This is achieved by stacking the U bit as part of the CCR. It is not possible to force a return to Supervisor state by executing an RTI while in User state.

3.2 MPU Descriptors

The MPU provides eight descriptors, each of which defines an accessible memory range. Each descriptor applies to one or more bus masters. On the S12XE, the bus masters are the CPU (in Supervisor state and User state) and the XGATE.

A descriptor consists of a memory range (defined by a start and an end address) that applies across the entire global memory space with a resolution of eight bytes. It also includes one or more bus masters, a control to make the range read-only, and a control that allows the bus master to execute code within the range. Without these additional controls, every address within the range would be readable, executable, and writable to the relevant masters. See Figure 5 for the content of the descriptors.

M0	M1	M2	M3	Start address [22:3]

WP	NEX	0	0	End address [22:3]

M0	Protected Supervisor state (CPU)
M1	User state (CPU)
M2	XGATE
M3	Not used on S12XEP100

WP	Write protected
NEX	No execution

Figure 5. MPU Descriptors

In Supervisor state, the CPU configures the descriptors required by the next User state task. After completing the configuration, the CPU executes the User state task. The CPU typically jumps to the start of the task by modifying the return address on the stack and executing an RTI. This approach is shown in Figure 6. The CPU can alternatively set the U-bit and jump directly to the User state task.

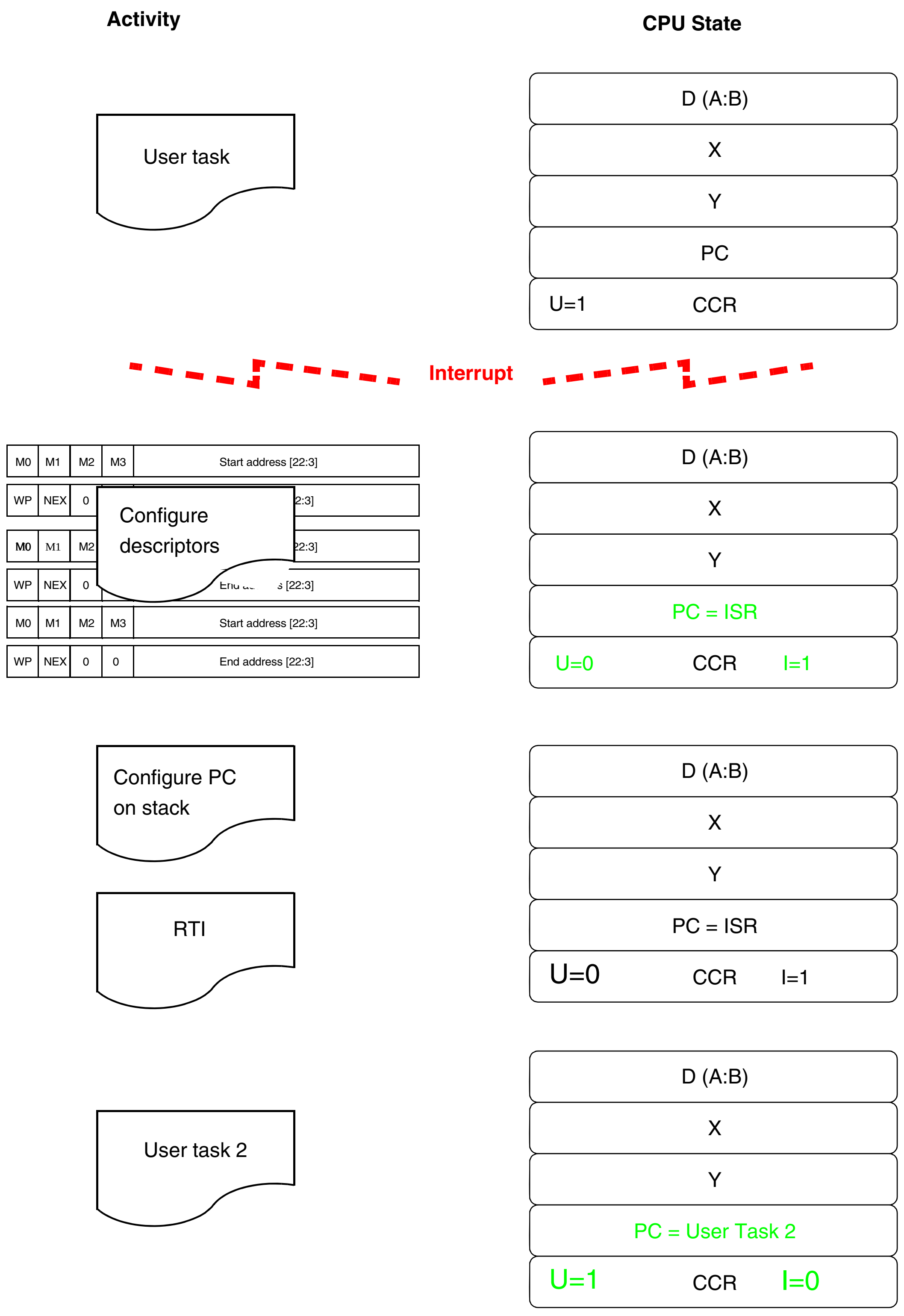

Figure 6. Typical Flow When Using MPU

Software integrators must pay attention to the configuration of descriptors. For a User state task, this requires a task code descriptor and a task data descriptor. In many cases, however, this is not enough. When coding in C, the linker may add library calls to implement certain functions, so there must also be a descriptor that fulfills that requirement. There may also be task-specific constants stored in emulated EEPROM or in flash that also require a descriptor.

There are no restrictions on the CPU when in Supervisor state. All memory is accessible so no specific requirements for descriptors exist. But for a Protected Supervisor state task, the CPU must have access to the task code, variables, interrupt vectors, stack and register space, and library code and constants.

The system integrator must also configure the correct type of access. The NEX-bit in each descriptor prevents the bus master executing code from within the descriptor range. This bit must be cleared for code spaces like application and library code. The WP-bit prevents the bus master from writing to the memory range. This is useful when sharing global variables in RAM or for allowing tasks access to the current input status of ports (but preventing them from changing the port configuration). See Table 2 for the combinations that these bits create.

Table 2. Combinations of Permissions

WP	NEX	Access	Example of use
0	0	Read, write, execute	Executing code from RAM
0	1	Read, write	Stack and data regions
1	0	Read, execute	Code in flash
1	1	Read	Data in flash

It is possible to combine descriptors by overlapping their memory ranges. The overlapped region accumulates the restrictions of the source descriptors. In Figure 7, descriptor 0 applies to CPU User state from address $40_0000 to $41_BFFF with read-only and execute permissions (typically a flash code configuration). Descriptor 1 also applies to CPU User state, but goes from address $40_8000 to $41_FFFF with read/write and no execute permissions (typically a RAM data configuration). Because the descriptors overlap, an overlapped range is created from $40_8000 to $41_BFFF. This accumulates the restrictions from the two descriptors and so is a read-only and no execute region (typically flash data configuration). Figure 8 shows an example of the use of descriptors in a simple application that contains a scheduler, two CPU tasks, and an XGATE thread.

PD0

M0	M1	M2	M3	Start address [22:3]
0	**1**	**0**	**0**	**100_0000_0000_0000_0000_0**

WP	NEX	0	0	End address [22:3]
0	**1**	**0**	**0**	**100_0001_1011_1111_1111_1**

PD1

M0	M1	M2	M3	Start address [22:3]
0	**1**	**0**	**0**	**100_0000_1000_0000_0000_0**

WP	NEX	0	0	End address [22:3]
1	**0**	**0**	**0**	**100_0001_1111_1111_1111_1**

Overlapped range

M0	M1	M2	M3	Start address [22:3]
0	**1**	**0**	**0**	**100_0000_1000_0000_0000_0**

WP	NEX	0	0	End address [22:3]
1	1	**0**	**0**	**100_0001_1011_1111_1111_1**

Figure 7. Effect of Overlapping Descriptors

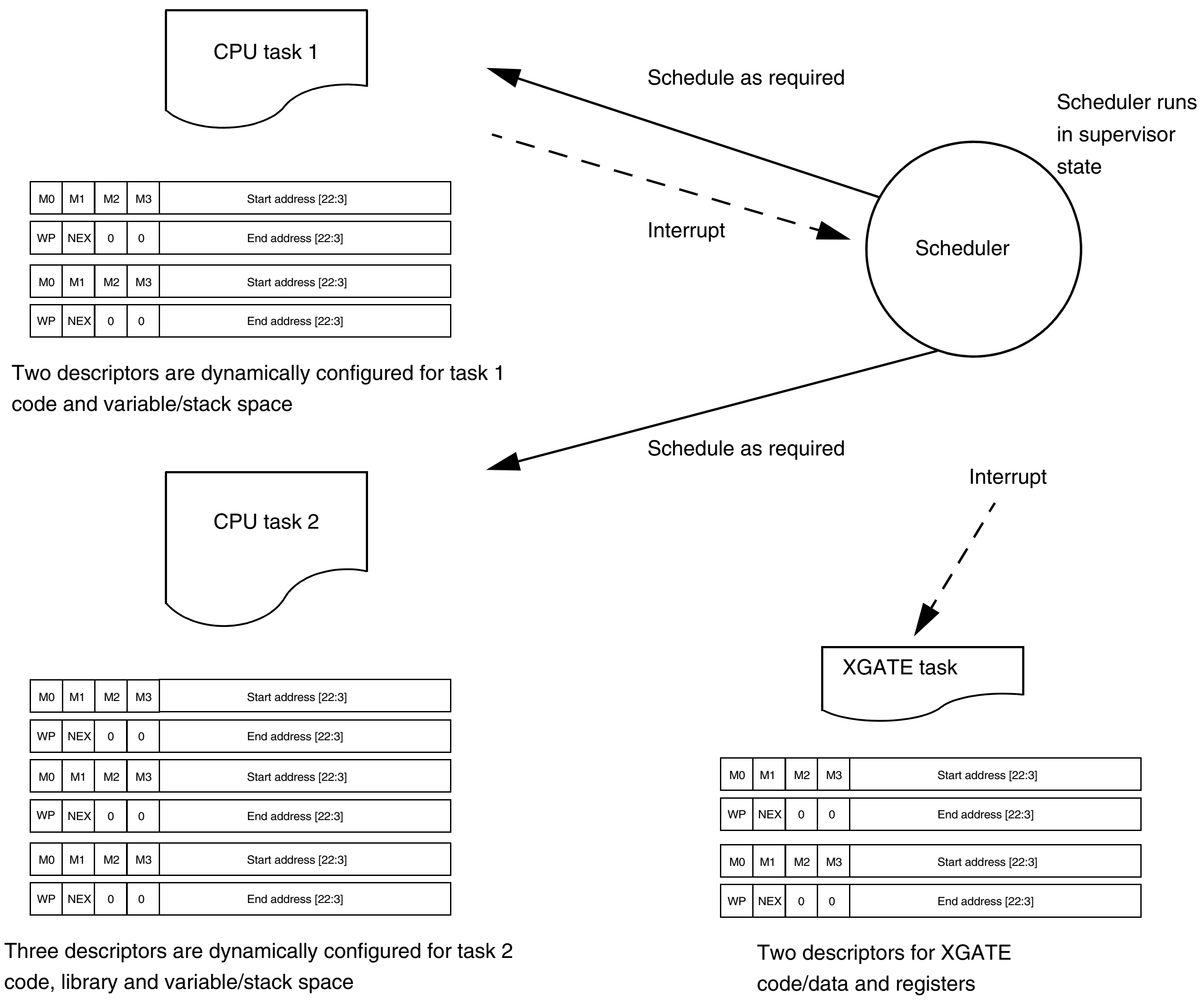
CPU task 1
Schedule as required
Scheduler runs in supervisor state
M0 M1 M2 M3 Start address [22:3]
WP NEX 0 0 End address [22:3]
Interrupt
Scheduler
Two descriptors are dynamically configured for task 1 code and variable/stack space
Schedule as required
Interrupt
CPU task 2
XGATE task
Three descriptors are dynamically configured for task 2 code, library and variable/stack space
Two descriptors for XGATE code/data and registers

Figure 8. Example of a Simple System Using the MPU

As seen in the figure, the tasks can call the scheduler via an interrupt opcode provided by the S12XE called SYS that is not inhibited by the I-bit in the CCR. In this configuration, the scheduler is considered trusted code and can run in Supervisor state.

3.3 MPU Protection Response and Recovery

If the MPU detects that a bus master has accessed resources outside of permissions defined by the descriptors, it causes an interrupt that prevents intended access. If the CPU violates the memory protection, it receives a non-maskable MPU access interrupt.

The CPU interrupt sets the AEF-bit in the MPU flag register (MPUFLG) and loads the address where the violation occurred into the MPUSTAT[0:2] registers. The MPUFLG register also contains information about write or execute violations in the WPF and NEXF bits, respectively.

The XGATE is stopped if it violates the protection configuration, and an XGATE software error interrupt flag is set so that the CPU can manage the error. In this case, the AEF bit is not set and the MPUSTAT[0:2] registers are not loaded with the violation address. The CPU can read the XGATE core registers (including PC and CCR) directly from its register space to determine the XGATE violation condition.

When defining memory protection, you must consider an error recovery strategy. You may need to place the application in a known good state or perform a graceful shutdown. You may be able to shut down the task and give a warning to seek assistance if the error occurs during a non-critical task. Ultimately, the error-recovery strategy depends on the application requirements.

In Protected Supervisor state, CPU access to MCU resources is controlled by descriptors. These must be configured carefully to allow operation in normal and error conditions. The MCU helps in cases of CPU violation by automatically creating an explicit descriptor for the register space within the MCU. This means that the CPU recovery routine can make adjustments to the register space when in Protected Supervisor state even if such access is not normally allowed. Descriptors for the error interrupt vector and the error recovery ISR must be included to allow the CPU to process the error interrupt correctly.

If the interrupt vector is not included in a descriptor range, then the original application violation leads to an interrupt that causes another interrupt when it attempts to fetch the vector and so on. This also applies to the ISR code and to any other resources the ISR requires.

Problems can arise because the CPU and XGATE perform pre-fetches of opcodes to maximize the performance of the MCU. Because the MPU cannot distinguish between pre-fetch opcodes that execute and those that do not, several extra bytes must be added beyond the end of the memory range. The CPU reference manual contains information about its queue and the microcontroller reference manual contains information for the XGATE.

3.4 Operational Correctness

The MPU protects against software that accesses invalid resources. However, even if a function has not accessed an invalid resource, it may continue to operate incorrectly. Errors causing incorrect operation or slow responses in the system can arise in variable contents or stacks because of unusual combinations of system events. You can address this type of error by providing diagnostics that can detect or correct undesired behavior that occurs once the system is in use. Often provided at reset, these diagnostics perform detailed checks on the status of the system before the operation begins. Hardware interlocks like the COP watchdog or ECC protection on the S12XE are two of the diagnostics available when the application is running.

You can also add functional checks to running systems. In traditional single-core MCUs, this capacity is limited by the additional load the checks place on the CPU as well as the fact that they are running on the processor whose performance they are checking. With these limitations in mind, you can see why the dual core S12X family brings specific advantages for embedded diagnostics.

4 Correct RAM contents

4.1 Errors in Embedded Systems

The operational times for many embedded systems extend to weeks or even longer between resets. These systems are at risk of accumulating small errors that eventually disrupt normal operation. These errors affect volatile memory and are caused by memory leaks, stack overflow, array index overrun, and invalid pointer values. The number of errors can be reduced in embedded systems, though, because dynamic memory may be allocated on the stack and so this can be classified as a stack overrun. Similarly, it is possible to combine array overruns and invalid variable values because the overrun is an issue only if it damages another variable.

It is possible for the MPU to detect these errors if the software stack and variables have some natural separation in memory. This makes it easier to separate protection descriptors. It is common practice, however, to group a task's RAM requirements under a single descriptor.

4.2 XGATE Co-Processor

Because the XGATE co-processor shares access to RAM, it can validate its contents and identify some of the errors listed above. In Figure 9, the XGATE verifies the correct extent of the stack by examining the contents of a buffer zone.

```
@interrupt void XG_StackCheck(void)
{
       char i;
       /* Check that each word in the stack buffer zone is
       at the default value */
       for (i=0;i<stackMax;i++)
       if (stackBuffer[i] != stackDefault)
       /* Call response function to handle */
       XG_StackFail(i);
}
```

Figure 9. Stack Overflow Testing

Figure 10 shows how a secondary processor can validate variable contents and detect errors on the primary core. The code declaration shows an array and a far (24-bit) pointer placed beside each other in memory. There is no range checking on the received message length, and a message longer than four bytes causes an overflow into the pointer. Because the pointer is used rarely and because a five byte message is rarely received, the error can go undetected for an extended time.

The XGATE code checks the pointer value and sends an interrupt to the CPU to signal that the value is incorrect. It can also restore the pointer to a known good value, can ensure that peripheral configuration registers have the correct values, and can verify that the CPU is operating at a functional level.

CPU Code

```
unsigned char message[4];
far char* menu;
interrupt void SCIRx()
{
         char byte;
         /Assume no receive errors
         byte = SCI->scisr1.byte;
         byte = SCI->scidrl.byte;
         if (index == 0)
         {
         index = 1;
         message[0] = byte
         } else
         {
         message[index] = byte;
         index++;
         }
         if (message[0] == index)
         index = 0;}
```

XGATE Checking Code

```
union u_farptr
{
unsigned long l;
unsigned int i[2];
}farptr;

interrupt void Checker(void)
{
// Check far pointers are valid
         farptr = menu;
         if (farptr.i[0] !=
menupage)
         _SIF(SW1);
}
```

Figure 10. Variable Validation

4.3 Functional Correctness

A dual core system provides opportunities for functional correctness that do not exist in single-core systems, such as the double-checking of algorithms, dual-key authorization, and challenge-response authentication.

You should ensure that the outcome of an algorithm is correct in all circumstances for certain applications, particularly in environments where the microcontroller affects the active or passive safety of a system. Figure 11 shows an example in which both cores compute the outcome of an algorithm but must agree on it before it is used. Run different versions of the algorithm on each core to ensure independent outcomes. In some cases, an approximation of the correct outcome may be sufficient as a margin check. Even if identical high-level code is used for both algorithms, the machine-dependent implementation is completely different because of the architectural differences between the CPU and XGATE on the S12XE. This provides an added measure of protection against algorithmic or compiler errors.

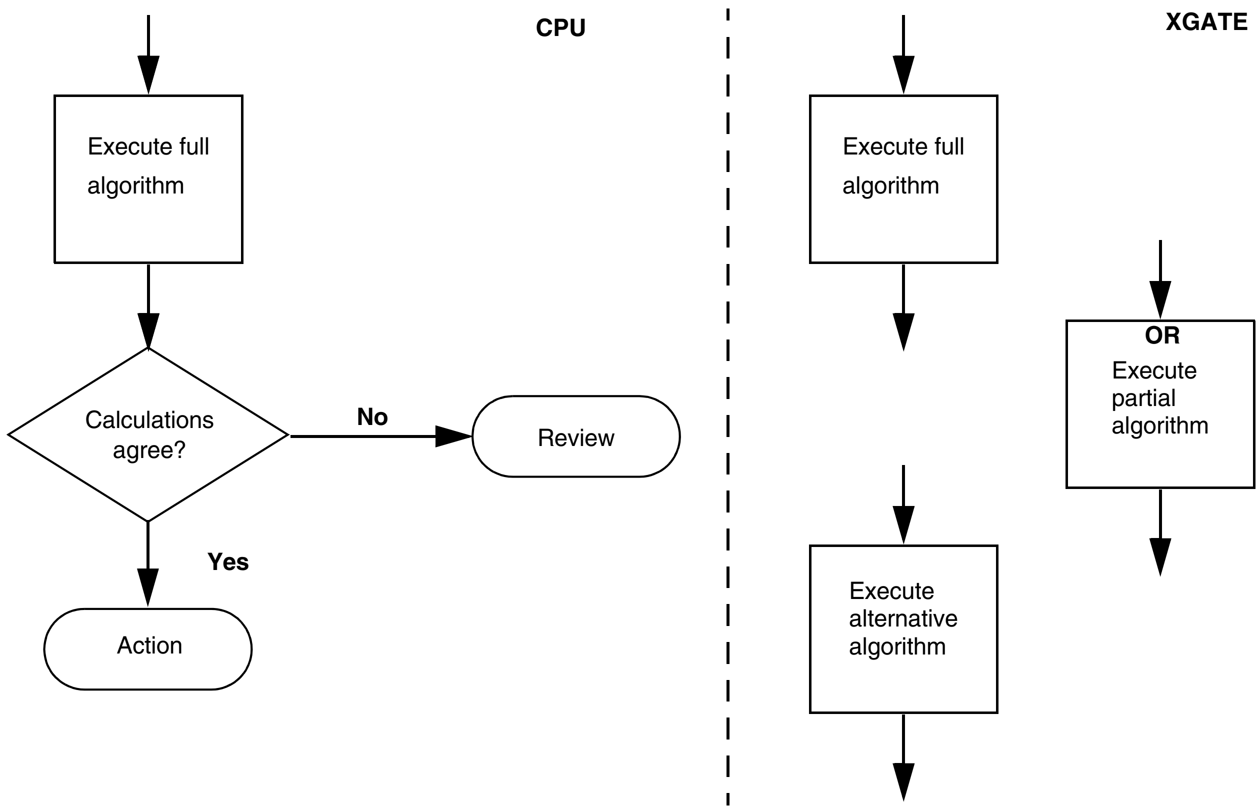

Figure 11. Example of Algorithm Checking

In the approach illustrated above, a single core determines the validity of the algorithm output. If an error occurs during the operation of this hardware or software, it is conceivable that this decision may not be correctly evaluated. In this case, a dual-key approach may offer additional integrity. Seen in Figure 12, this approach allows both cores to present their calculated results to an independent checkpoint such that both can agree on the conclusion before the specified activity occurs.

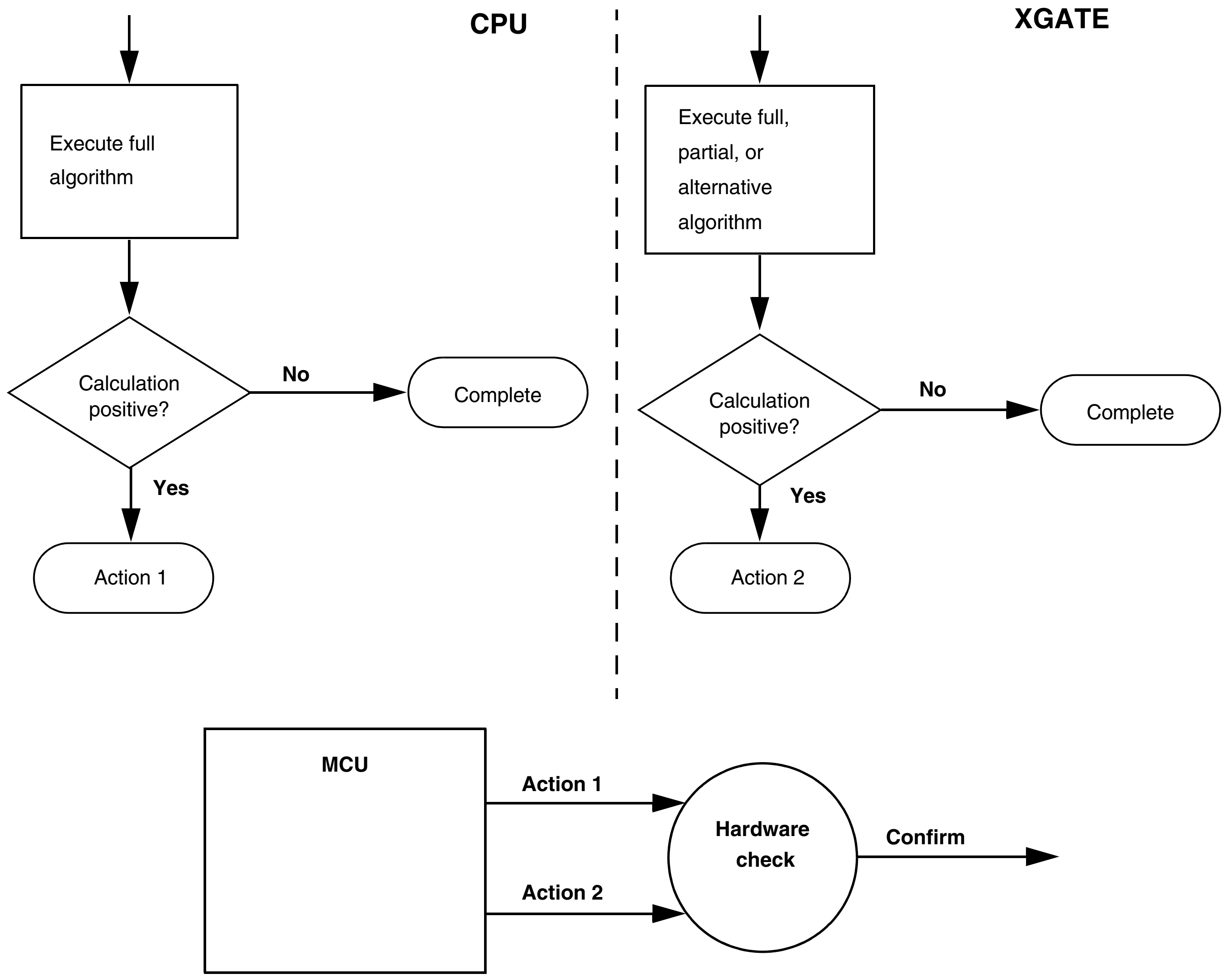

Figure 12. Dual-Key Output

You can use a variation of the cryptographic challenge-response approach to assess the accuracy of the primary core's activity. The secondary core can challenge the primary core to provide information on its status, and then check that the response was provided in a correct and timely manner. One useful challenge might be to return its program counter, stack the pointer values and any other useful internal registers, perform a checksum on a fixed area of memory, and verify the interrupt vector locations. The returned information would be checked against the values known to or calculated by the secondary core.

Even if a system is behaving correctly, there may be an issue with its response time.

4.4 Timeliness

The overall loading of the processor influences the timeliness of a system response. A system designer must ensure that the highest priority tasks can directly access the CPU while also preventing lower priority tasks from being blocked. This can be difficult to achieve in a fixed priority scheme. Dynamic scheduling schemes allow you to modify a task's priority based on the overall system loading.

XGATE can help improve the system performance by adjusting task and request priorities dynamically. A general benefit of a second processing core is that the kernel's task priority algorithm can execute on the second core, which in turn reduces the execution time of the kernel itself. This reduces the task-switching time of the system. A more specific benefit of the second core can be in dynamically allocating and changing priorities of the service requests in the system.

Consider a system where a very important task must be completed by a given deadline. A standard approach would be to disable interrupts during this period, which makes the system completely unresponsive to external requests. As seen in Figure 13, the S12XE allows the XGATE to manage service requests differently during this period. A typical response may be to acknowledge the request and indicate that the system will deal with it when it is less busy. This approach relies on the CPU being aware that it is under heavy load.

This approach may not be possible, for example, in systems where there are several tasks that all need to execute with interrupts active. In this case the XGATE can monitor the interrupt response time of the CPU and adjust the priority of the incoming requests such that they are all serviced within a known time. The XGATE can monitor interrupt flag states and determine if the servicing of these is slower than expected.

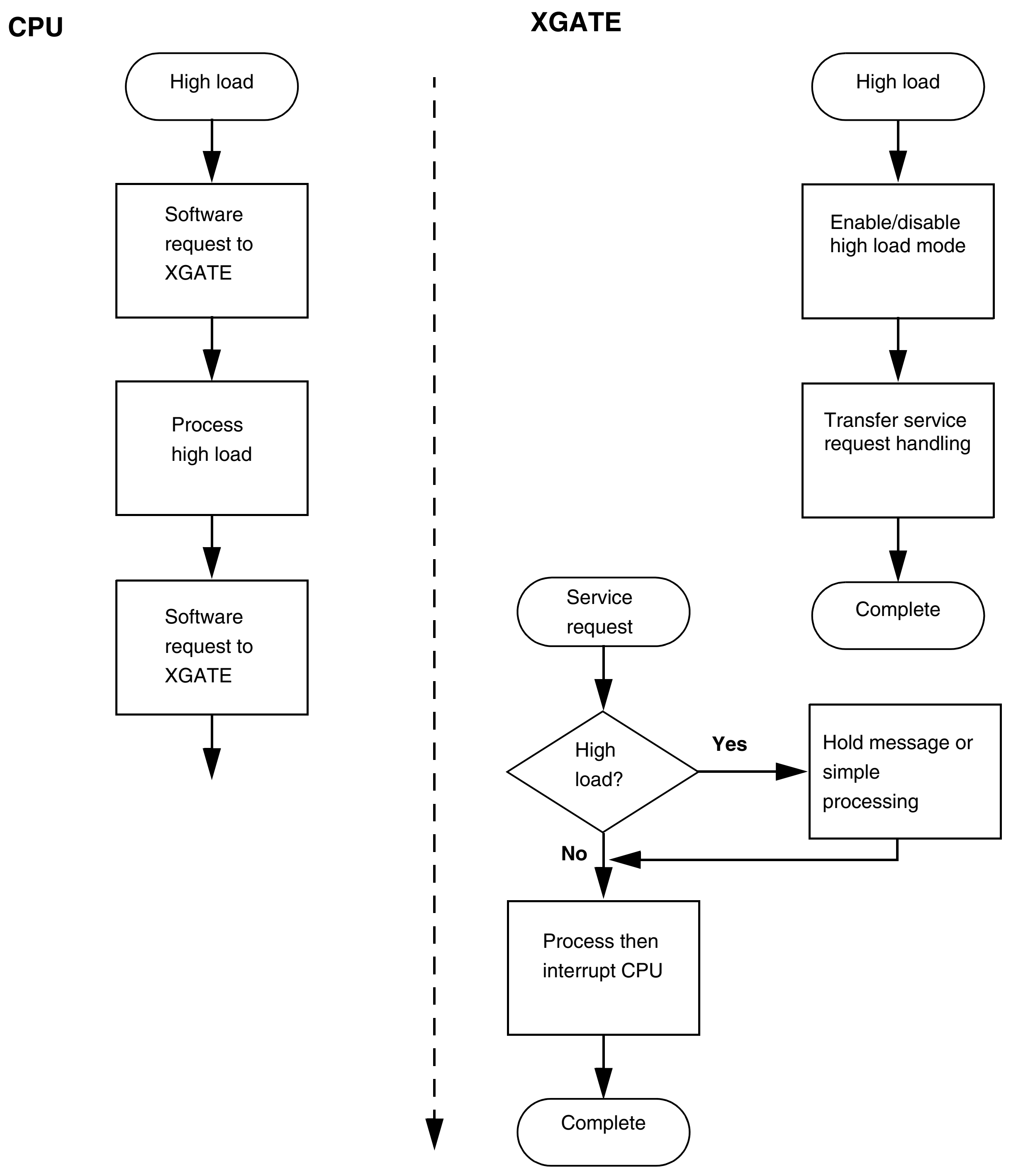
CPU
XGATE
High load
Software request to XGATE
Process high load
Software request to XGATE
High load
Enable/disable high load mode
Transfer service request handling
Complete
Service request
High load?
Yes
Hold message or simple processing
No
Process then interrupt CPU
Complete

Figure 13. Request Handling in High-Load Conditions

4.5 Error Recovery

Depending on the application requirements, XGATE has several options if it discovers a CPU software malfunction. The least intrusive action would be to set a flag to warn the CPU that there is, for example, evidence of a small excursion into the stack buffer. The CPU may be able to execute a recovery algorithm that re-initializes the dynamic memory and interrupts stack pointers. If the CPU program flow is defective, then this flag may not be tested, in which case the XGATE could detect this condition and escalate the response. See Figure 14 for a potential flow of this approach.

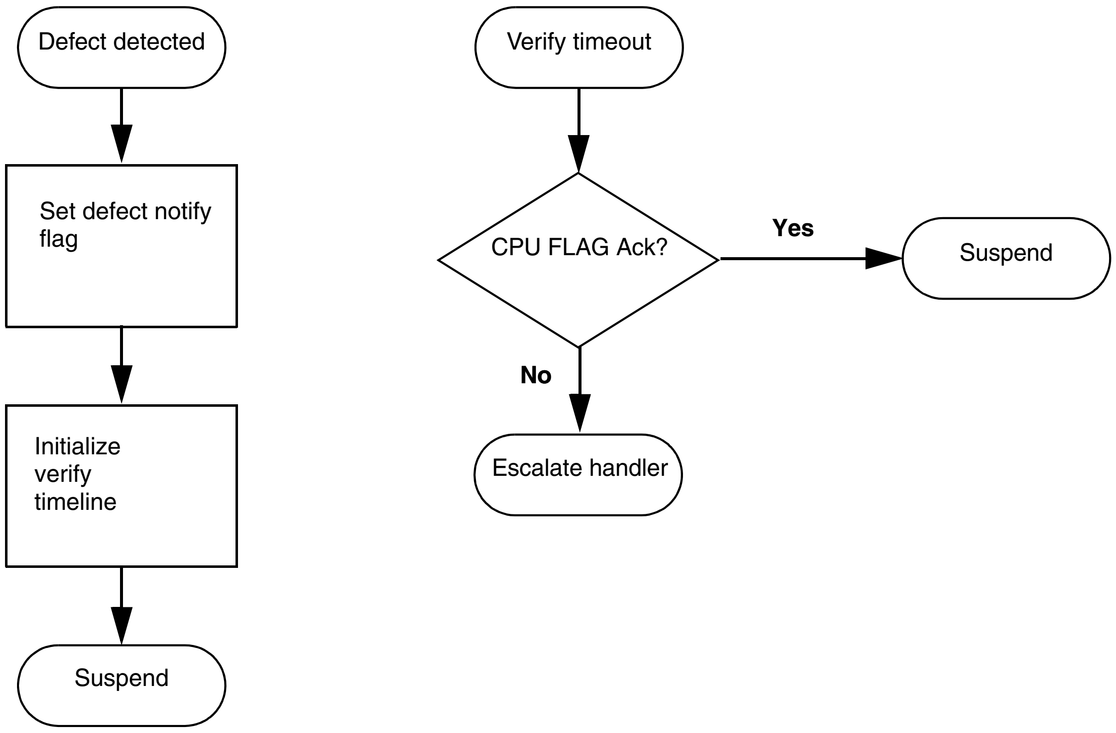

Figure 14. Potential Response to Low Level Defect

Sending a low-priority interrupt to the CPU would be quicker and would increase the likelihood of the CPU responding at the cost of using the stack space. A defective CPU flow could prevent the servicing of this interrupt, interrupts could be disabled incorrectly, or a high-priority interrupt could block the core. But the XGATE could still detect this null response and escalate the recovery activity. Remove the low-level interrupt before taking this action. See Figure 15 for a potential flow.

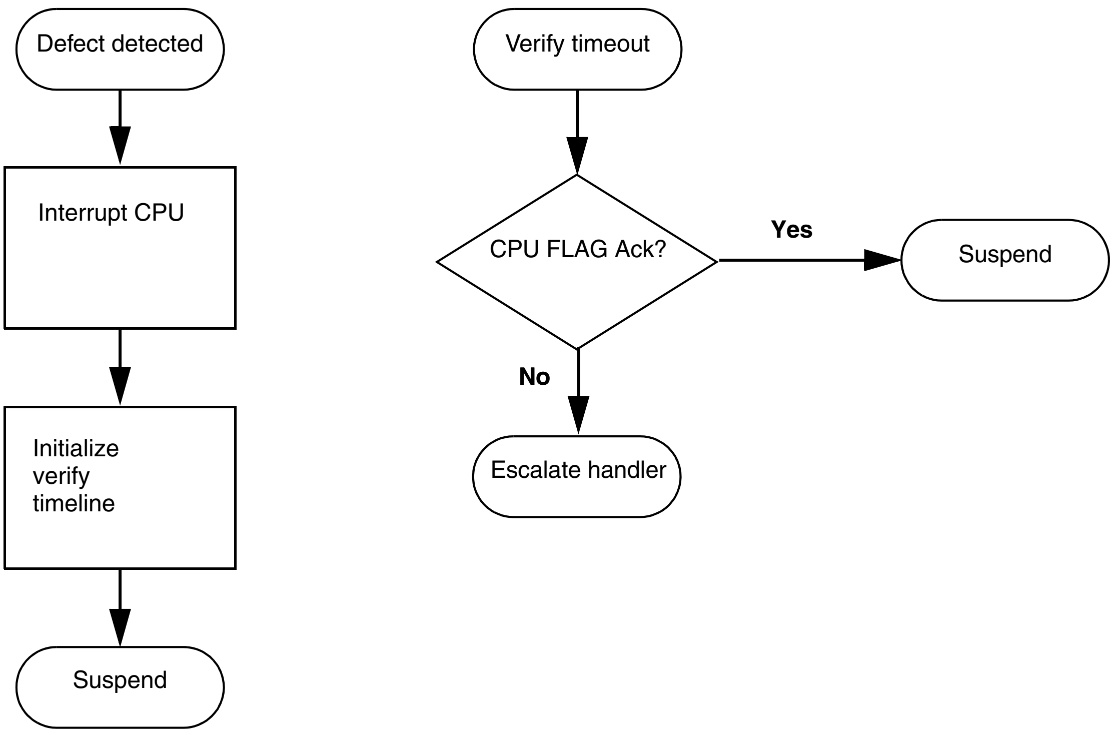

Figure 15. Potential Second-Level Response

By reserving the highest-priority interrupt for problem recovery, the MCU can ensure that interrupt blocking is kept to a minimum. However, non-maskable and disabled interrupts would remain blocked. Unlike the low-priority approach, this interrupt interferes with high-level activities on the CPU. For example, the execution of an important communications interrupt could be stalled while the diagnostic interrupt was being serviced with the potential loss of data. As with the low-level approach, the XGATE can monitor the lack of a CPU response to the interrupt and escalate again afterwards. In the ultimate step, the XGATE can force the whole MCU to reset if necessary, but this aborts all current operations and can cause data loss and disruption. In some embedded systems this may not be a permitted behavior. As shown in Figure 16, the most convenient method for resetting the MCU is to violate the watchdog servicing conditions. This results in an instant reset and subsequent reconfiguration of the system.

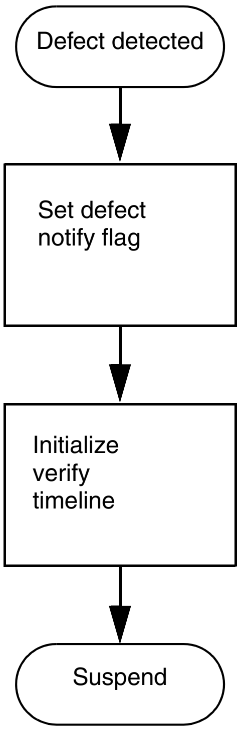

Figure 16. Potential Fourth-Level Response

5 Summary of Techniques

This application provides a summary of hardware and software resources available to you on the S12XE family of microcontrollers. Many of the techniques require no more than configuration of the hardware provided by the S12XE, while others use software techniques and the unique architecture of the family.

Ultimately, the requirements of the application determine how many of these approaches are required and how much effort the system designer must put into the integrity of the system.

6 References

- Improving System Integrity Through the Use of a Peripheral Co-Processor, S. McAslan. Proceedings of embedded world conference 2006.
- Saftey Drives Automotive Body Electronics Computers, R. Kalman, automotivedesignline.com, May 2006
- MC9S12XEP100 Reference Manual, Rev. 1.04, Freescale Semiconductor

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

Document Number: AN3419
Rev. 0, Draft A
02/2007