APPLICATION NOTE

AN713
XA interrupts

Author: Kent Lowman

1997 Feb 10

Philips
Semiconductors

PHILIPS

XA interrupts AN713

Author: Kent Lowman

CONTENTS

1. INTRODUCTION

This document will discuss the XA Interrupt Structure from two different perspectives. First we will look at the XA Family Interrupt Structure since it is important to have an understanding of all the interrupt options available in the XA Family. This general discussion will introduce us to all available interrupt options in the XA Family. We will cover the details of Exception Interrupts, Trap Interrupts and Software Interrupts since these will generally be standard across the XA Family. However, specific implementations for Event Interrupts will not be covered, since each XA derivative may have a unique subset of Event Interrupts available.

Next we will look in detail at the XA-G3 Interrupt Structure since this is the first available member of the XA derivative family. This discussion will cover the function and detail of all interrupts included on the XA-G3. We will assume an understanding of the general structure and function of XA interrupts as given in the section on XA Family Interrupts. Event Interrupts that are unique to the XA-G3 will be covered in detail.

2. XA FAMILY INTERRUPT STRUCTURE

This section covers all the interrupt options available in the XA Family. It should be used as background material to gain familiarity with the way XA interrupts function. Please refer to the sections on specific XA derivatives for details of their individual interrupt structures.

The XA Family offers a very powerful Interrupt Structure with various levels of user programmable configuration and control. This allows for great flexibility in various applications but does require the user to provide adequate initialization and set-up for interrupts to function as expected. This required initialization is more detailed than that needed for microcontrollers such as the 8051 which have a much simpler interrupt structure.

2.1. XA Family Interrupts

The XA architecture defines four kinds of interrupts. These are listed below in order of intrinsic priority:

- Exception Interrupts
- Trap Interrupts
- Event Interrupts
- Software Interrupts

Exception interrupts reflect system events of overriding importance. Examples are stack overflow, divide-by-zero, and Non-Maskable Interrupt. Exceptions are non maskable and are always processed immediately as they occur, regardless of the Execution Priority of currently executing code.

Trap interrupts are processed as part of the execution of a TRAP instruction. Since the Trap interrupt is non-maskable the interrupt vector is always taken when the TRAP instruction is executed.

Event interrupts reflect less critical hardware events, such as a UART needing service or a timer overflow. These events may be associated with some on-chip device or an external interrupt input. Event interrupts are maskable and are processed only when their priority is higher than that of currently executing code. Event interrupt priorities are software selectable by writing bits in the IPA (Interrupt Priority) register for each interrupt source. In this section we will generically refer to the IPA register but in most XA derivatives this will actually be a group of registers (IPA0–IPAn) based on the number of event interrupts available. Each event interrupt can be set to one of 16 priority levels by writing four bits in the IPA register assigned to the interrupt event. A priority level of zero effectively disables the interrupt since the priority must be greater than the Execution Priority of the code that is currently executing for the interrupt to be serviced.

Software interrupts are an extension of event interrupts, but are caused by software setting a request bit in a Special Function Register or SFR. Software interrupts are also processed only when their priority is higher than that of currently executing code. Software interrupt priorities are fixed at levels from 1 through 7. Thus code with an Execution Priority of 8 or higher can NOT be interrupted by any of the Software Interrupts.

All forms of interrupts trigger the same sequence: First, a stack frame containing the address of the next instruction and then the current value of the PSW (Program Status Word) is pushed on the System Stack. A vector containing a new PSW value and a new execution address is fetched from code memory. The new PSW value entirely replaces the old, and execution continues at the new address, e.g., at the specific interrupt service routine. Since the execution address for the Interrupt Service Routine (ISR) is only 16 bits wide, the ISR for all XA interrupt sources must begin in Page 0 of code memory (the first 64K byte page). This allows a faster interrupt response time than if a full 32 bit ISR address was fetched. Notice that all XA ISR's always begin in Page 0 of code memory independently of whether the XA is operating in Page 0 Mode or not. Page 0 Mode is a special mode where total XA code memory is limited to 64K bytes.

The new PSW value may include a new setting of PSW bit **SM** (System Mode), allowing handler routines to be executed in System or User mode, and a new value of PSW bits **IM3–IM0**, reflecting the Execution Priority of the new task. These capabilities are basic to multi-tasking support on the XA.

Returns from all interrupts should in most cases be accomplished by the RETI instruction, which pops the System Stack and continues execution with the restored PSW context. All interrupt service routines will normally be executed in System Mode. If an RETI instruction is executed from an ISR running in User Mode an exception interrupt will be generated.

The XA architecture contains sophisticated mechanisms for deciding when and if an interrupt sequence actually occurs. As described below, Exception Interrupts are always serviced as soon as they are triggered. Event Interrupts are deferred until their Execution Priority is higher than that of the currently executing code. For both exception and event interrupts, there is a systematic way of handling multiple simultaneous interrupts. Software Interrupts and Trap Interrupts occur only when program instructions generating them are executed, so there is no need for conflict resolution within these two interrupt classes.

2.2. The Interrupt Mask (Execution Priority)

The PSW operating mode flags (shown below) set several aspects of the XA operating mode including the Interrupt Mask or Execution Priority. The terms Interrupt Mask and Execution Priority are two different ways of defining the same thing. Interrupt Mask refers to the fact that all interrupts with a priority equal to or lower than this value are “masked” and are not allowed to occur. Execution Priority refers to the fact that for any interrupt (or task) to be allowed to run, it must have a higher priority than the Execution Priority of the task that is currently running. The four Interrupt Mask bits (**IM3–IM0**) identify the Execution Priority of the code that is currently executing. The XA interrupt controller compares the current setting of the IM bits to the priority of any pending interrupts to decide whether to initiate an interrupt sequence. The value 0 in the IM bits indicates the lowest Execution Priority, or fully interruptable code. The value 15 (or 0F hexadecimal) indicates the highest Execution Priority, which is not interruptable by maskable event interrupts. However, note that an Execution Priority of 15 does not inhibit servicing of Exception Interrupts or Traps since these are non-maskable.

PSWH (401h) – bit addressable

SM	TM	RS1	RS0	IM3	IM2	IM1	IM0

PSW operating mode flags

All of the flags in the upper byte of the PSW (**PSWH**), except the bits **RS1** and **RS0** (Register Bank Select), may be modified only by code running in system mode.

2.3. PSW Initialization

At reset, the initial XA PSW value is loaded from the reset vector located at address 0 in code memory. The initial **PSWH** value sets the stage for system software initialization and its value requires great attention. **PSWL** contains only status flags which do not require initialization. Therefore, the initial value of **PSWL** is generally of no special system-wide importance and may be set to zero or some other value. Philips recommends that the PSW initialization value in the reset vector sets **IM3–IM0** to all 1’s so that XA initialization code is set as the highest Execution Priority process (and therefore can not be interrupted by any source other than an exception or trap). It is also recommended that the reset vector set the **SM** bit to 1, so that execution begins in System Mode. This gives an initial PSW value of 8F00H for normal operation. At the conclusion of the user initialization code, the Execution Priority is typically reduced, often to 0, to allow all other maskable interrupt driven tasks to run.

Here’s an example set of declarations that create the recommended initial value of **PSWH**:

```
system_mode     equ     8000h
max_priority    equ     0F00h
initial_PSW     equ     system_mode + max_priority
```

2.4. Interrupt Service Data Elements

There are two data elements associated with XA interrupts. The first is the stack frame created when each interrupt is serviced. The second is the interrupt vector table located at the beginning of code memory. Understanding the structure and contents of each is essential to the understanding of how XA interrupts are processed.

2.4.1. Interrupt Stack Frame

A stack frame is generated, always on the System Stack, for each XA interrupt. The stack frame is stored for the duration of interrupt service and used to return to and restore the CPU state of the interrupted code. There is one case where this is not true. The Exception Interrupt triggered by a Reset event re-initializes the stack pointers, so no stack frame is preserved. This makes the Reset Exception Interrupt unique since it is not terminated with an RETI like all other XA interrupts

The stack frame in the native 24-bit XA operating mode is shown in Figure 1. Three words (6 bytes) are stored on the stack in this case. The first word pushed is the low-order 16 bits of the current Program Counter (PC), i.e., the address of the next instruction to be executed. The next word contains the high-order byte of the current PC. A zero byte is used as a pad since the stack must be word aligned. Since a complete 24-bit address is stored in the stack frame a return to any code location in the 16M byte XA address range is possible. The third word in the XA stack frame contains a copy of the PSW at the instant the interrupt was serviced.

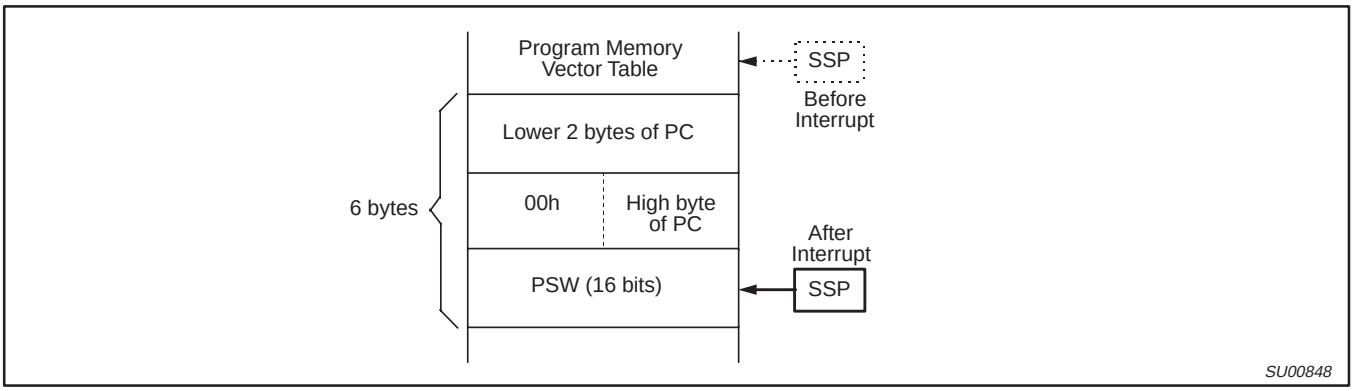

Figure 1. XA Stack Frame – Non Page 0 Mode (24 bit mode)

When the XA is operating in Page 0 Mode (**SCR** bit **PZ** = 1) the stack frame is smaller. In Page 0 Mode, only 16 address bits are used throughout the XA. The stack frame in Page 0 Mode is only four bytes since the High Byte of the PC and the pad byte are not needed. Obviously, it is very important that stack frames of both sizes not be mixed since this would corrupt the return address and therefore the operation of the XA. This is one reason it is recommended that the user set the System Configuration Register (**SCR**) once during XA initialization to select either Page 0 Mode or 24 bit address mode, and leave it unchanged thereafter.

2.4.2. Interrupt Vector Table

The XA uses the first 284 bytes of code memory (addresses 0 – 011B hex) for an interrupt vector table. The table may contain up to 71 double-word entries, each corresponding to a particular interrupt event.

The double-word entries each consist of a 16 bit address of an Interrupt Service Routine (ISR) and a 16 bit PSW replacement value. Because vector addresses are 16-bit, the first instruction of each Interrupt Service Routine must be located in the first 64K bytes of XA memory. The first instruction of all ISR's must also be word-aligned. Note that this is normally handled by the XA assembler, which will insert NOP's automatically to assure word-alignment of ALL labels. The replacement PSW value contains key elements such as the choice of System or User mode for the service routine, the Register Bank selection, and an Interrupt Mask setting.

The first 16 vectors, starting at code memory address 0 are reserved for Exception Interrupt vectors. The second 16 vectors are reserved for Trap Interrupts. The following 32 vectors in the table are reserved for Event Interrupts. The final 7 vectors are used for Software Interrupts. A figure presented later will illustrate the XA vector table and the structure of each component vector. Of the vectors assigned to Exceptions, 7 are assigned to events specific to the XA CPU and 9 are reserved. All 16 Trap Interrupts may be used freely since none are reserved. Assignments of Event Interrupt vectors are derivative dependent and vary with the peripheral device complement and pinout of each XA derivative.

Unused interrupt vector locations should typically be set to point to a “null” service routine (an RETI instruction), rather than be overwritten by executable instructions. This is especially true of the exception interrupts, since these are non-maskable and could conceivably occur in a system where the designer did not expect them. If these vectors are routed to an RETI instruction, the system can essentially ignore the unexpected exception or interrupt condition and continue operation.

Note that when using some hardware development tools it may be preferable not to initialize unused vector locations with a “null handler”. This allows the XA development tool to recognize and flag these unexpected interrupt conditions so they can be addressed.

2.5. The Reset Exception Interrupt

Immediately after the –RST line goes high, the XA generates a Reset Exception Interrupt. As a result, the initial PSW and address of the first instruction (the “start-up code”) are fetched from the reset vector in code memory at location 0. Here’s an example in generalized assembler format of the setup for the Reset Exception Interrupt:

```
                code_seg                ;  establish code segment
                org     0h              ;  start at address 0

;  reset_vector

                dw      initial_PSW     ; PSW reset value – normally 8F00H
                dw      startup_code    ; starting address of code

; the XA Interrupt Vector Table goes from 0 –011Bh in code memory

                org     120h            ;  start code at address 120h
                                        ;  (above interrupt vector table)

startup_code:   . . .                   ;  put user startup code here
                . . .
                . . .
; end user startup code by enabling ALL interrupts

                mov.b   PSWH, #80H      ; PSWH run value to allow ALL interrupts
                mov.b   PSWL, #00H      ; PSWL value is not critical
```

The **PSWH** initialization value given in this example sets System Mode (**SM**), selects register bank 0 (any register bank could be used) and clears **TM** so that Trace Mode is inactive.

The startup_code sequence may be followed directly by user startup code or by a simple branch to any application code. At the end of user initialization code remember to lower the Interrupt Mask value in **PSWH** so maskable event interrupts can occur. Do NOT use an RETI instruction at the conclusion of the startup_code sequence even though this is part of the Reset Exception Interrupt handler. The Reset initializes the Stack Pointer (**SP)** and does

not leave an interrupt stack frame. This makes the Reset Exception Interrupt unique since it is not terminated with an RETI like all other XA interrupts.

Notice that the same Reset Exception Interrupt is generated for any of the three possible XA reset sources:

1. Hardware reset via the –RST pin
2. Software reset via the RESET instruction
3. Watchdog Timer generated reset

2.6. XA Interrupt Types

This section describes the four types of XA interrupts. It addresses interrupts that are available in the XA Family but may or may not be present on any given XA derivative.

2.6.1. Exception Interrupts

Exception interrupts reflect events of overriding importance and are always serviced when they occur. Exceptions currently defined in the XA core include: Reset, Breakpoint, Trace, Divide-by-0, Stack overflow, and Return from Interrupt (RETI) executed in User Mode. Nine additional exception interrupts are reserved.

NMI is listed in the table of exception interrupts below because NMI is handled by the XA core in the same manner as exceptions, and factors into the precedence order of exception processing. However, the vector address reserved for NMI is actually mapped right in the middle of the Event Interrupt vector address space. This should not cause NMI, which is a non-maskable Exception Interrupt, to be confused with the maskable Event Interrupts. Note that NMI is part of the XA Family Interrupt Structure but is not implemented on the first XA derivative (the XA-G3).

Since exception interrupts are by definition not maskable, they must always be serviced immediately regardless of the Execution Priority level of currently executing code (as defined by the IM bits in the PSW). In the unusual case that more than one exception is triggered at the same time, there is a hard-wired service precedence ranking. This ranking determines which exception vector is taken first if multiple exceptions occur. Of course, being non-maskable, any exception occurring during execution of the ISR for another exception will still be serviced immediately. In this case, the exception vector taken last may be considered the highest priority, since its code will execute first. This LIFO (Last-In-First-Out) system means that an Exception Interrupt with a higher service precedence actually has a higher priority. Even though the Exception with the higher service precedence will be taken last, it will still be serviced first.

Programmers should be aware of the following when writing Exception Interrupt handlers:

1. Since another exception could interrupt a stack overflow ISR, care should be taken in all exception handler code to minimize the possibility of a destructive stack overflow. Remember that stack overflow exceptions only occur once as the stack crosses the lower address limit of 0080h.
2. The Breakpoint (caused by execution of the BKPT instruction, or a hardware breakpoint in an emulation system) and Trace exceptions are intended to be mutually exclusive. In both cases, the handler code will want to know the address in user code where the exception occurred. If a breakpoint occurs during trace mode, or if trace mode is activated during execution of the breakpoint handler code, one of the handlers will see a return address on the stack that points within the other handler code.

Exception Interrupts – Non Maskable

Exception Interrupt	Vector Address	Arbitration Ranking	Service Precedence
Breakpoint	0004h–0007h	1	0
Trace	0008h–000Bh	1	1
Stack Overflow	000Ch–000Fh	1	2
Divide-by-zero	0010h–0013h	1	3
User RETI	0014h–0017h	1	4
<reserved1>	0018h–001Bh	—	—
<reserved2>	001Ch–001Fh	—	—
<reserved3>	0020h–0023h	—	—
<reserved4>	0024h–0027h	—	—
<reserved5>	0028h–002Bh	—	—
<reserved6>	002Ch–002Fh	—	—
<reserved7>	0030h–0033h	—	—
<reserved8>	0034h–0037h	—	—
<reserved9>	0038h–003Fh	—	—
NMI	009Ch–009Fh	1	6
Reset	0000h–0003h	0 (High)	7 always serviced immediately aborts other exceptions

2.6.2. Trap Interrupts

Trap Interrupts are intended to support application-specific requirements, as a convenient mechanism to enter globally used routines, and to allow transitions between User Mode and System Mode. TRAP 0 through TRAP 15 are defined and may be used as required by applications. Trap interrupts are generated by the TRAP instruction. A trap interrupt will occur if and only if the instruction is executed, so there is no need for a precedence scheme with respect to simultaneous traps. A trap acts like an immediate non-maskable interrupt, using a vector to call one of several pieces of code that will be executed in System Mode. This may be used to obtain system services for application code, such as altering the Data Segment register for example. Some XA development software and Real Time Operating Systems may reserve certain Trap instructions for specific system functions. An example of this would be the Hitech XA C compilers use of Trap 15 to access system services.

Traps – Non-Maskable

Description	Vector Address	Arbitration Ranking
Trap 0	0040–0043h	1
Trap 1	0044–0047h	1
Trap 2	0048–004Bh	1
Trap 3	004C–004Fh	1
Trap 4	0050–0053h	1
Trap 5	0054–0057h	1
Trap 6	0058–005Bh	1
Trap 7	005C–005Fh	1
Trap 8	0060–0063h	1
Trap 9	0064–0067h	1
Trap 10	0068–006Bh	1
Trap 11	006C–006Fh	1
Trap 12	0070–0073h	1
Trap 13	0074–0077h	1
Trap 14	0078–007Bh	1
Trap 15	007C–007Fh	1

Example of Trap Interrupt:

```
TRAP        #05          ; generate Trap 5 Interrupt

; immediate branch to TRAP05 Interrupt Service Routine (non-maskable)
```

Example of ISR for a Trap Interrupt:

```
TRAP05:     .
            . user code
            .
            RETI
```

Notice that the Execution Priority (**IM3–IM0** value) is not relevant since Traps are non-maskable. When the TRAP instruction is executed the Trap Interrupt will always occur. No user action is required in the ISR to “clear” the Trap before the RETI is executed.

2.6.3. Event Interrupts

On typical XA derivatives, event interrupts will arise from on-chip peripherals and from events detected on external interrupt input pins. Event interrupts may be globally enabled/disabled via the **EA** bit in the Interrupt Enable register (**IE**) and individually masked by specific bits in the **IE** register or registers. When an event interrupt for a peripheral device is disabled but the peripheral is not turned off, the peripheral interrupt flag can still be set by the peripheral. If the peripheral interrupt is re-enabled an interrupt will occur. An event interrupt that is enabled can only be serviced when its Execution Priority is higher than that of the currently executing code. Event Interrupts have 16 priority levels that can be individually set in the Interrupt Priority (IPA) register for the appropriate interrupt source. This allows tight control over the scheduling and occurrence of each maskable XA interrupt source. If more than one event interrupt occurs at the same time, the higher priority setting will determine which one is serviced first. If more than one interrupt is pending at the same priority level, a hardware precedence scheme is used to choose the first to service. Consult the data sheet for a specific XA derivative for details on the hardware precedence scheme or arbitration ranking.

Note that the PSW (including the Interrupt Mask or Execution Priority bits) is loaded from the interrupt vector table when an event interrupt is serviced. Thus, the priority at which the ISR executes could be different from the priority at which the interrupt occurred. Since the occurrence priority is determined by the IPA register setting for that interrupt rather than by the PSW image in the vector table. Normally it is advisable to set the Execution Priority in the interrupt vector to be the same as the IPA register setting that will be used in the code. If the Execution Priority for any ISR is set lower than the Interrupt Priority for that interrupt, then that interrupt will interrupt itself continuously and likely overflow the stack. This can occur since most event interrupts are still pending during part of the ISR.

Event Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable bit	Interrupt Priority	Arbitration Ranking
External interrupt 0	IE0	0080–0083h	EX0	IPA0.3–0	2
Timer 0 interrupt	TF0	0084–0087h	ET0	IPA0.7–4	3
External interrupt 1	IE1	0088–008Bh	EX1	IPA1.3–0	4
Timer 1 interrupt	TF1	008C–008Fh	ET1	IPA1.7–4	5
Timer 2 interrupt	TF2(EXF2)	0090–0093h	ET2	IPA2.3–0	6
<reserved1>		0094–0097h			
<reserved2>		0098–009Bh			
NMI (non-maskable)		009C–009Fh			1
Serial port 0 Rx	RI.0	00A0–00A3h	ERI0	IPA4.7–4	7
Serial port 0 Tx	TI.0	00A4–00A7h	ETI0	IPA5.3–0	8
Serial port 1 Rx	RI.1	00A8–00ABh	ERI1	IPA5.3–0	9
Serial port 1 Tx	TI.1	00AC–00AFh	ETI1	IPA5.7–4	10
<reserved3>		00B0–00B3h			
<reserved4>		00B4–00B7h			
<reserved5>		00B8–00BBh			
<reserved6>		00BC–00BFh			
<reserved7>		00C0–00C3h			
<reserved8>		00C4–00C7h			
<reserved9>		00C8–00CBh			
<reserved10>		00CC–00CFh			
<reserved11>		00D0–00D3h			
<reserved12>		00D4–00D7h			
<reserved13>		00D8–00DBh			
<reserved14>		00DC–00DFh			
<reserved15>		00E0–00E3h			
<reserved16>		00E4–00E7h			
<reserved17>		00E8h–00EBh			
<reserved18>		00EC–00EFh			
<reserved19>		00F0–00F3h			
<reserved20>		00F4–00F7h			
<reserved21>		00F8–00FBh			
<reserved22>		00FC–00FFh			

Notice that the vector address reserved for NMI is mapped into the Event Interrupt vector address space. This should not cause NMI, which is a non-maskable Exception Interrupt, to be confused with the maskable Event Interrupts. The NMI vector address is mapped into this space because NMI shares certain characteristics with the External Interrupts. Both NMI and External Interrupts are generated by a signal on an external XA pin that is then fed into the XA interrupt controller.

2.6.4. Software Interrupts

Software Interrupts act just like event interrupts, except that they are caused by software writing to an interrupt request bit in an SFR. The standard XA implementation of the software interrupt mechanism provides 7 interrupts that are associated with 2 SFRs. One SFR, the Software Interrupt Request register (**SWR**), contains 7 request bits – one for each software interrupt. The second SFR is the Software Interrupt Enable register (**SWE**), containing one enable bit for each software interrupt.

SWR (42Ah) – bit addressable

—	**SWR7**	**SWR6**	**SWR5**	**SWR4**	**SWR3**	**SWR2**	**SWR1**

Software Interrupt Request

SWE (47Ah) – **NOT** bit addressable

—	**SWE7**	**SWE6**	**SWE5**	**SWE4**	**SWE3**	**SWE2**	**SWE1**

Software Interrupt Enable

Software interrupts have fixed interrupt priorities, one each at priorities 1– 7. These are shown in the table below. Software interrupts are available in the XA Family Interrupt Structure but may not be present on all XA derivatives. Consult the data sheet for a specific XA derivative for details on the availability of software interrupts.

Software Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable Bit	Interrupt Priority
Software interrupt 1	SWR1	0100–0103	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010B	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010F	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0118–011B	SWE7	(fixed at 7)

Example of Software Interrupt:

```
        OR.B    SWE, #01        ; enable Software Interrupt 1 indirectly
                                ; since SWE not bit addressable!
        SETB    SWR1            ; generate Software Interrupt 1
```

; branch to SWI1 Interrupt Service Routine if and only if the current Execution Priority (**IM3–IM0**) = 0
; if the Execution Priority of the running code is > 0, then SWI1 will NOT occur, but will remain pending

Example of ISR for a Software Interrupt:

```
SWI1:   .
        CLR     SWR1            ;  clear Software Interrupt 1
        .
        RETI
```

Notice that Software Interrupt 1 has a fixed priority of 1. This means that the **IM3–IM0** value would need to be 0 (Execution Priority of the current executing code equal 0) for this priority 1 interrupt to occur. Any **IM3–IM0** value > 0 would block the Software Interrupt 1 from occurring. The SWR1 bit must be cleared by the user before exiting the ISR or the Software Interrupt will re-occur.

It is also important to address the Software Interrupt Request bits by their bit addressable names and not by their bit position in **SWR** since they are shifted (i.e., **SWR1** is SWR.0 not SWR.1). Thus it is correct to use these instructions:

```
SETB   SWR1          ; generate Software Interrupt 1
CLR    SWR1          ; clear Software Interrupt 1
```

but incorrect to use these instructions:

```
SETB   SWR.1         ; actually would generate Software Interrupt 2
CLR    SWR.1         ; actually would clear Software Interrupt 2
```

Since the Software Interrupt Enable register is NOT bit addressable it is wise to enable Software Interrupts as shown below (paying close attention to the actual bit position of the desired enable bit):

```
OR.B        SWE, #01H      ; enable Software Interrupt 1 indirectly
OR.B        SWE, #02H      ; enable Software Interrupt 2 indirectly
OR.B        SWE, #04H      ; enable Software Interrupt 3 indirectly
OR.B        SWE, #08H      ; enable Software Interrupt 4 indirectly
OR.B        SWE, #10H      ; enable Software Interrupt 5 indirectly
OR.B        SWE, #20H      ; enable Software Interrupt 6 indirectly
OR.B        SWE, #40H      ; enable Software Interrupt 7 indirectly
```

Using the “OR” instruction allows the individual Software Interrupt to be enabled without affecting the setting of the enable bits for any other Software Interrupts.

The primary purpose of the software interrupt mechanism is to provide an organized way in which portions of the event interrupt routines may be executed at a lower priority level than the one at which the service routine began. An example of this would be an event Interrupt Service Routine that has been given a very high priority in order to respond quickly to some critical external event. This ISR has a relatively small portion of code that must be executed immediately, and a larger portion of follow-up or “clean-up” code that does not need to be completed right away (but does not need to wait until the main software loop). Overall system performance may be improved if the lower priority portion of the ISR is actually executed at a lower priority level, allowing other more important interrupts to be serviced.

If the high priority ISR simply lowers its execution priority at the point where it enters the follow-up code, by writing a lower value to the IM bits in the PSW, a situation called “priority inversion” could occur. Priority inversion describes a case where code at a lower priority is executing while a higher priority routine is kept waiting. An example of how this could occur by writing to the IM bits follows, and is illustrated in Figure 2.

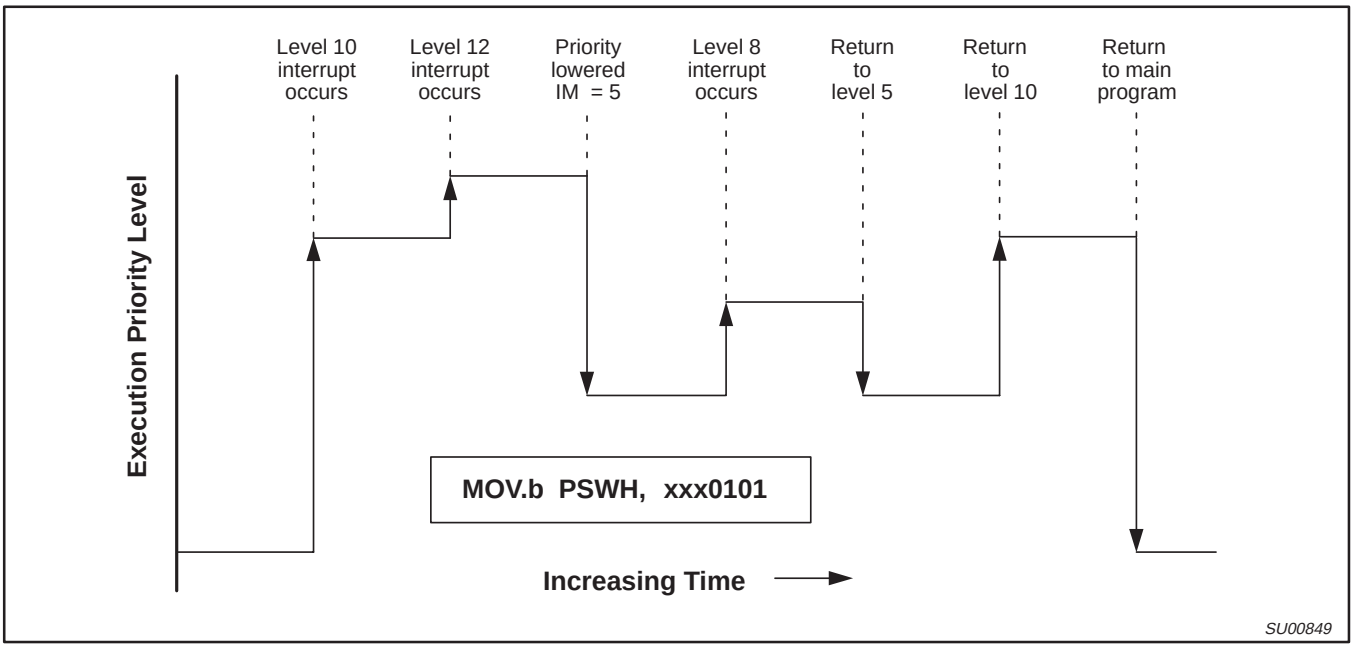

Figure 2. Priority Inversion (No Software Interrupts)

Suppose code is executing at level 0 and is interrupted by an event interrupt that runs at level 10. This is again interrupted by a level 12 interrupt. The level 12 ISR completes a time-critical portion of its code and wants to lower the priority of the remainder of its code (the non-time critical portion) in order to allow more important interrupts to occur. So, it writes to the IM bits, setting the execution priority to 5. The ISR continues executing at level 5 until a level 8 event interrupt occurs. The level 8 ISR runs to completion and returns to the level 5 ISR, which also runs to completion. When the level 5 ISR completes, the previously interrupted level 10 ISR is reactivated and eventually completes.

It can be seen in this example that lower priority ISR code executed and completed while higher priority code was kept waiting on the stack. This is priority inversion.

In those cases where it is desirable to alter the priority level of part of an ISR, a software interrupt may be used to accomplish this without risk of priority inversion. The ISR must first be split into 2 pieces: the high priority portion, and the lower priority portion. The high priority portion remains associated with the original interrupt vector. The lower priority portion is associated with the interrupt vector for a software interrupt, in this case Software Interrupt 5. At the completion of the high priority portion of the ISR, the code sets the request bit for software interrupt 5, and then returns. The remainder of the ISR, now actually the ISR for software interrupt 5, executes when it becomes the highest priority pending interrupt.

The diagram in Figure 3 shows the same sequence of events as in the example of priority inversion, except using software interrupt 5 as just described. Note that the code now executes in the correct order (higher priority first).

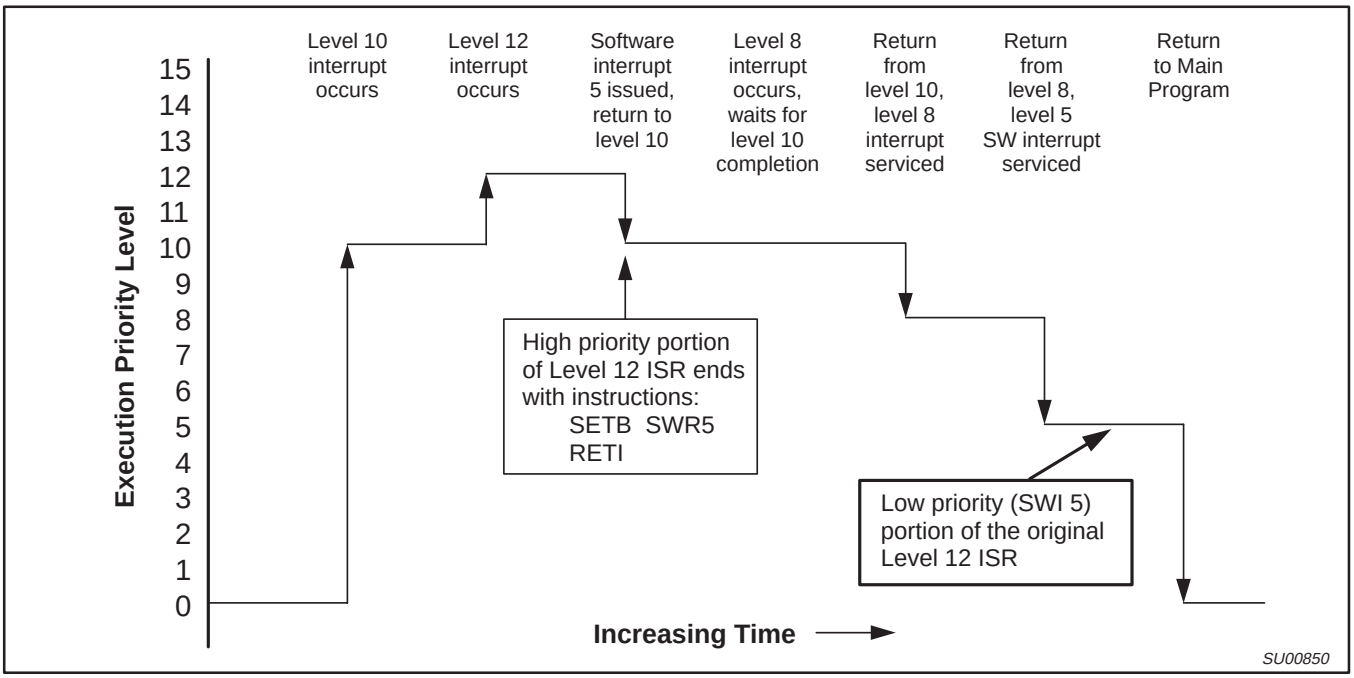

Figure 3. Correct Priority Execution with Software Interrupts

3. XA-G3 INTERRUPT STRUCTURE

This section covers only the interrupts that are implemented on the XA-G3. Recall that not all interrupt options covered in the XA Family Interrupt Structure are available in this first XA derivative product. Our focus here will be on the actual XA-G3 Interrupts and any differences from the XA Family Interrupts. For details on XA-G3 interrupts that are identical to the XA Family Interrupts please refer to the appropriate section in the “XA Family Interrupt Structure”.

3.1. XA-G3 Interrupts

The XA-G3 defines four types of interrupts:

- Exception Interrupts – These are system level errors and other very important occurrences that include Stack overflow, Divide by 0, Breakpoint, Trace, User Mode RETI and Reset.
- Trap Interrupts – These are TRAP instructions, generally used to call system services in a multi-tasking system.
- Event Interrupts – These are peripheral interrupts from devices such as UARTs, timers, and external interrupt inputs.
- Software Interrupts – These are equivalent to hardware event interrupts, but are requested only under software control and have fixed priority levels.

Exception interrupts, trap interrupts, and software interrupts are generally standard for XA derivatives and are detailed in the XA Family Interrupt Structure. Event Interrupts tend to be different on various XA derivatives and will be explained in detail for the XA-G3.

The XA-G3 supports 38 vectored interrupt sources. These include 9 maskable Event Interrupts (for the various XA-G3 peripherals), 7 Software Interrupts, 6 Exception Interrupts and 16 Traps.

The complete interrupt vector list for the XA-G3, including all 4 interrupt types, is shown in the following tables. The tables include the address of the vector for each interrupt, the related priority register bits (if any), and the arbitration ranking for that interrupt source. The arbitration ranking determines the order in which interrupts are processed if more than one interrupt of the same priority occurs simultaneously.

3.2. XA-G3 Interrupt Vectors

3.2.1. Exception Interrupts

Exceptions – Non-Maskable

Description	Vector Address	Arbitration Ranking	Service Precedence
Reset	0000–0003h	0 (High)	7
Breakpoint	0004–0007h	1	0
Trace	0008–000Bh	1	1
Stack Overflow	000C–000Fh	1	2
Divide-by-0	0010–0013h	1	3
User RETI	0014–0017h	1	4

3.2.2. Trap Interrupts

Traps – Non-Maskable

Description	Vector Address	Arbitration Ranking
Trap 0	0040–0043h	1
Trap 1	0044–0047h	1
Trap 2	0048–004Bh	1
Trap 3	004C–004Fh	1
Trap 4	0050–0053h	1
Trap 5	0054–0057h	1
Trap 6	0058–005Bh	1
Trap 7	005C–005Fh	1
Trap 8	0060–0063h	1
Trap 9	0064–0067h	1
Trap 10	0068–006Bh	1
Trap 11	006C–006Fh	1
Trap 12	0070–0073h	1
Trap 13	0074–0077h	1
Trap 14	0078–007Bh	1
Trap 15	007C–007Fh	1

3.2.3. Event Interrupts

Event Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable bit	Interrupt Priority	Arbitration Ranking
External interrupt 0	IE0	0080–0083h	EX0	IPA0.2–0	2
Timer 0 interrupt	TF0	0084–0087h	ET0	IPA0.6–4	3
External interrupt 1	IE1	0088–008Bh	EX1	IPA1.2–0	4
Timer 1 interrupt	TF1	008C–008Fh	ET1	IPA1.6–4	5
Timer 2 interrupt	TF2(EXF2)	0090–0093h	ET2	IPA2.2–0	6
Serial port 0 Rx	RI.0	00A0–00A3h	ERI0	IPA4.2–0	7
Serial port 0 Tx	TI.0	00A4–00A7h	ETI0	IPA4.6–4	8
Serial port 1 Rx	RI.1	00A8–00ABh	ERI1	IPA5.2–0	9
Serial port 1 Tx	TI.1	00AC–00AFh	ETI1	IPA5.6–4	10

3.2.4. Software Interrupts

Software Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable Bit	Interrupt Priority
Software interrupt 1	SWR1	0100–0103h	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107h	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010Bh	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010Fh	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113h	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117h	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0118–011Bh	SWE7	(fixed at 7)

Arbitration ranking is only relevant when more than one interrupt (from the same category) is triggered at the same time. For example, 2 exceptions or 2 event interrupts at the same time would use the arbitration ranking to determine which interrupt source was serviced first. The interrupt with the lower arbitration ranking will be serviced first, and thus has a higher priority. Since this simultaneous triggering is not possible for Traps or Software Interrupts, these two interrupt categories do not require an arbitration ranking.

Since Exceptions and Traps are non-maskable they will always occur immediately and therefore do not require an Interrupt Priority. Exceptions and Traps may be considered to have “infinite” priority.

3.3. XA-G3 Event Interrupts

The 9 maskable Event Interrupts on the XA-G3 share a global interrupt enable bit (the **EA** bit in the **IEL** register) and each also has a separate individual interrupt enable bit (in the **IEH** or **IEL** registers). Notice that the power-up reset value for **EA** and each of the separate interrupt enable bits is 0. This effectively disables each of the maskable interrupts in two different places. For a maskable event interrupt to occur the global **EA** bit must be set to "1" and the individual interrupt enable bit in the **IEH/IEL** register must be set for that particular interrupt source. The interrupt enable bits were listed in the previous table of Event Interrupts along with their associated interrupt. As shown below the interrupt enable bits are all bit addressable.

IEH (427h) – bit addressable

—	—	—	—	ETI1	ERI1	ETI0	ERI0

Interrupt Enable High Byte

IEL (426h) – bit addressable

EA	—	—	ET2	ET1	EX1	ET0	EX0

Interrupt Enable Low Byte

In the XA-G3 each event interrupt can be set to occur at 1 of 8 priority levels via bits in the Interrupt Priority (IPA) registers, **IPA0–IPA5**. The value 0 in the IPA field gives the interrupt priority 0, in effect, disabling the interrupt. Since the **IPA0–IPA5** registers all have power-up reset values of 0, each of the event interrupts starts with a priority of 0 and is thus disabled.

The XA-G3 differs slightly from the XA Family Interrupt Structure in the way that interrupt priority levels are set via the IPA registers. Since only 3 of the 4 IPA register bits are implemented in the XA-G3, only 8 of the 16 possible priority levels are available for each of the event interrupts. The value 0000h in one nibble of the **IPA0–IPA5** register gives the interrupt priority 0, a value of 0001h gives the interrupt a priority of 9, the value 0010h gives priority 10, etc. The value 0111h in one nibble of the **IPA0–IPA5** register gives the interrupt priority 15. Since the MSB or 4th bit in each nibble of the **IPA0–IPA5** register is not implemented in the XA-G3, writing the value 0001h or 1001h to the IPA register will yield the same results. The interrupt in question will be set to a priority level of 9 in both cases. However, since the 4th bit in each nibble of the **IPA0–IPA5** register is not implemented it can not be read back if written. If 1001h is written to either nibble of the **IPA0–IPA5** register and then read back, the value returned will be 0001h.

On the XA-G3 the user may want to write any non-zero IPA value with the upper bit always set. This provides both a reminder of the true interrupt priority and software compatibility with future XA derivatives.

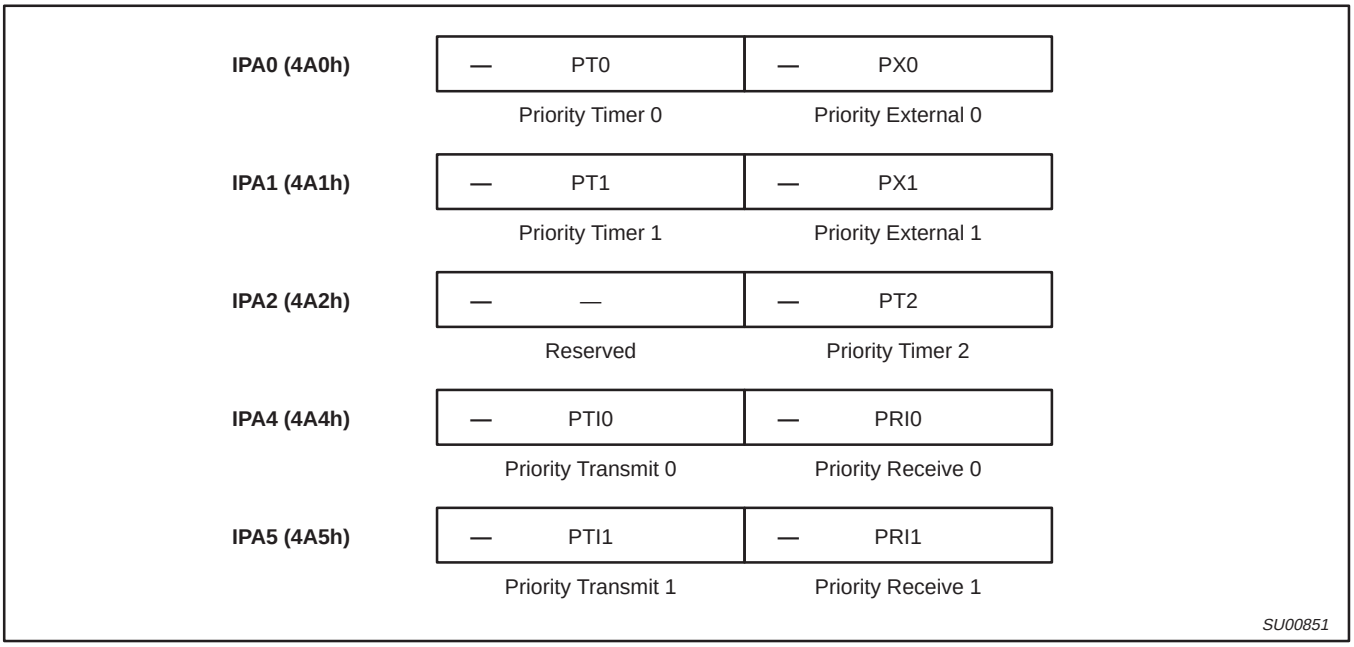

Figure 4. Interrupt Priority Registers IPA0–IPA5

Since event interrupts in the XA-G3 only support 8 of the 16 priority levels available in the XA Family Interrupt Structure, they can only have priorities of either 0 or 9–15. This means that Software Interrupts, with fixed priorities of 1–7, can not be granted higher priority than any of the XA-G3 Event Interrupts.

Event interrupts in the XA-G3 can be grouped into three basic types:

1. External Interrupts
2. Timer Interrupts
3. Serial Port Interrupts

Let's take a detailed look at each type of event interrupt.

3.3.1. External Interrupts

External interrupts available on the XA-G3 are External Interrupt 0 and External Interrupt 1. These external interrupts are controlled by bits in the **TCON** register as shown below:

TCON (410h) – bit addressable

TF1	**TR1**	**TF0**	**TR0**	**IE1**	**IT1**	**IE0**	**IT0**

Timer/Counter Control

External interrupts can be either falling edge triggered or low level triggered. This is controlled by the Interrupt Type Control bits **IT1/IT0**. If **IT1/IT0** is set to "1" then that interrupt will be set for falling edge trigger. If **IT1/IT0** is set to "0" then that interrupt will be set for low level trigger. When an external interrupt is detected it will set the Interrupt Edge Flag **IE1/IE0**. If the external interrupt is enabled the setting of this flag will generate an External Interrupt 1 or External Interrupt 0. The **IE1/IE0** flag will be cleared when the interrupt is processed or it can be cleared by software at any time.

3.3.2. Timer Interrupts

Timer interrupts available on the XA-G3 are Timer 0 interrupt, Timer 1 interrupt and Timer 2 interrupt. Timer 0 and Timer 1 interrupts are identical and are controlled by bits in the **TCON** register as shown below:

TCON (410h) – bit addressable

TF1	**TR1**	**TF0**	**TR0**	**IE1**	**IT1**	**IE0**	**IT0**

Timer/Counter Control

The timer is turned on by setting the Timer Run Control bit **TR1/TR0** to “1”. The timer is turned off by setting the Timer Run Control bit **TR1/TR0** to “0”. When the timer/counter overflows it will set the Timer Overflow Flag **TF1/TF0**. If the timer interrupt is enabled the setting of this flag will generate a Timer 0 Interrupt or a Timer 1 Interrupt. The **TF1/TF0** flag will be cleared when the interrupt is processed or it can be cleared by software at any time.

Timer 2 on the XA-G3 has additional functional modes over Timer 0 and 1 that will not be discussed here. Timer 2 interrupts are controlled by bits in the **T2CON** register as shown below:

T2CON (418h) – bit addressable

TF2	**EXF2**	**RCLK0**	**TCLK0**	**EXEN2**	**TR2**	**C/T2**	**CP/RL2**

Timer/Counter Control

Timer 2 is turned on by setting the Timer Run Control bit **TR2** to “1”. Timer 2 is turned off by setting the Timer Run Control bit **TR2** to “0”. When the timer/counter overflows it will set the Timer 2 Overflow Flag **TF2**. If the timer 2 interrupt is enabled, the setting of this flag will generate a Timer 2 Interrupt. The **TF2** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Timer 2 interrupt will reoccur. If **RCLK1/RCLK0** or **TCLK1/TCLK0** are set to “1”, then the Timer 2 overflow rate is being used as a baud rate clock source for UART0 or UART1. In this case the **TF2** flag will NOT be set when the timer/counter overflows.

If Timer 2 is enabled in external capture or reload mode, a negative transition on the T2EX pin will set the Timer 2 external flag **EXF2**. If the Timer 2 interrupt is enabled, the setting of the Timer 2 external flag **EXF2** can also generate a Timer 2 Interrupt. The **EXF2** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Timer 2 interrupt will reoccur.

3.3.3. Serial Port Interrupts

The two Serial Ports on the XA-G3 are identical and are called Serial Port 0 and Serial Port 1. Each Serial Port has two interrupts – one for the transmitter and one for the receiver. Notice that this is an enhancement over the Serial Port on the 8051 (which had only a single shared interrupt for both the transmitter and receiver). This gives the XA-G3 a total of four interrupts for the Serial Ports:

1. Serial Port 0 Rx
2. Serial Port 0 Tx
3. Serial Port 1 Rx
4. Serial Port 1 Tx

These Serial Port interrupts are controlled by bits in identical registers called **S0CON** and **S1CON**. To avoid confusion we will look only at **S0CON** as shown below:

S0CON (420h) – bit addressable

SM0_0	**SM1_0**	**SM2_0**	**REN_0**	**TB8_0**	**RB8_0**	**TI_0**	**RI_0**

Serial Port 0 Control

The Serial Port 0 receiver is enabled by setting the Receiver Enable bit **REN_0** to “1”. The Serial Port 0 receiver is disabled by setting the Receiver Enable bit **REN_0** to “0”. When a character is received by Serial Port 0 the Receive Interrupt Flag **RI_0** will be set. If the Serial Port 0 Rx interrupt is enabled the setting of this flag will generate a Serial Port 0 Rx Interrupt. The **RI_0** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Serial Port 0 Rx interrupt will reoccur.

When a character is transmitted by Serial Port 0 the Transmit Interrupt Flag **TI_0** will be set. If the Serial Port 0 Tx interrupt is enabled the setting of this flag will generate a Serial Port 0 Tx Interrupt. The **TI_0** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Serial Port 0 Tx interrupt will reoccur.

Serial Port 0 also has a Status Interrupt flag **STINT0** that is contained in the Serial Port 0 Extended Status Register (**S0STAT**). If the **STINT0** flag is set to “1” the extended status flags are enabled and any one of them can also generate a Serial Port 0 Rx Interrupt by setting the **RI_0** flag. These extended status flags include Framing Error, Overrun Error and Break Detect. Please refer to the XA-G3 data sheet for more details on these flags. The **RI_0** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Serial Port 0 Rx interrupt will reoccur.

As mentioned earlier the function of the Serial Port 1 Interrupts is identical to the Serial Port 0 Interrupts and therefore will not be covered here.

XA interrupts AN713

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381

Let's make things better.

Philips Semiconductors

PHILIPS