

Tutorial Introduction

PURPOSE:

- To explain MCU processing of reset and interrupt events

OBJECTIVES:

- Describe the differences between resets and interrupts.
- Identify different sources of resets and interrupts.
- Describe the MCU reset recovery process.
- Identify the steps to configure and service an interrupt event.
- Describe MCU exception processing.

CONTENT:

- 20 pages
- 3 questions

LEARNING TIME:

- 25 minutes

PREREQUISITE:

- The 68HC08 CPU training module and a basic understanding of reset and interrupt events

Welcome to this tutorial on resets and interrupts. The tutorial describes the different sources of reset and interrupt events and provides detailed training on 68HC08 MCU exception processing. Please note that on subsequent pages, you will find reference buttons in the upper right of the content window that access additional content.

Upon completion of this tutorial, you'll be able to describe the differences between resets and interrupts, identify different sources of reset and interrupt events, and describe MCU exception processing.

The recommended prerequisite for this tutorial is the 68HC08 CPU training module. It is also assumed that you have a basic knowledge of reset and interrupt events. Click the Forward arrow when you're ready to begin the tutorial.

Resets and Interrupts Overview

- **Reset sources:**
 - **External - power on, reset pin driven low**
 - **Internal - COP, LVI, illegal opcode, illegal address**
- **Resets initialize the MCU to startup condition.**
- **Interrupt sources:**
 - **Hardware**
 - **Software**
- **Interrupts vector the program counter to a service routine.**

Resets and interrupts are responses to exceptional events during program execution.

Resets can be caused by a signal on the external reset pin or by an internal reset signal. Internal reset signals can be generated by the Computer Operating Properly Module or the Low Voltage Inhibit Module. Other internal reset sources include an illegal opcode and an illegal address.

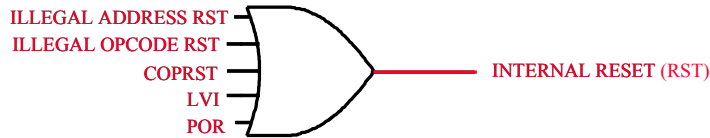
A reset stops MCU program execution. Once the reset is processed, the MCU immediately returns to a known startup condition and starts program execution from a user-defined memory location.

There are two types of interrupts, hardware interrupts and software interrupts. An interrupt doesn't stop the MCU or the operation of the instruction being executed. It vectors the program counter to an interrupt service routine. When an interrupt event occurs, the MCU first completes the current instruction and then changes the sequence of program execution to respond to the event.

An interrupt is similar to a reset in that it causes the MCU to fetch a new address for the program counter and sets the interrupt mask bit, or I-bit. Unlike a reset, interrupts suspend normal program execution only temporarily so the the processor can service the interrupt. After the interrupt is serviced, the processor is returned to where it left off executing normal program code.

Let's take a detailed look at resets and interrupts, beginning with resets.

Internal Reset Sources



First, let's review the different sources of internal resets.

An illegal address reset is generated when the CPU attempts to fetch an instruction from an address that isn't defined in the memory map. This type of reset provides additional system protection, returning the CPU to a known state when an invalid address is used.

An illegal opcode reset is generated when the CPU decodes an instruction that is not defined in the opcode set. Excessive noise or code that tries to execute incorrectly from data space may cause this event, returning the CPU to a known state.


The Computer Operating Properly (COP) reset is generated by an overflow of the COP counter, indicating that the COP timer was not serviced on-time and therefore expired. This is another system protection feature that will return the CPU to a known state in a runaway code scenario. For more information on the COP, see the COP tutorial.

The Low Voltage Inhibit (LVI) reset can optionally be generated when V_{DD} has dropped below a selected trip point. This feature protects against incorrect MCU operation during brown-outs or low-voltage power conditions. For the LVI reset, the reset line remains low for 4096 CGMXCLK clock cycles after V_{DD} is restored, allowing the clock to stabilize.

Power-On reset (POR) is an internal reset caused by a positive transition from 0 on the V_{DD} pin. All internal clocks to the CPU and MCU Modules are held inactive for 4096 CGMXCLK clock cycles to allow the oscillator to stabilize. During this time, the RST pin is driven low. The POR reset can only be activated when V_{DD} drops to 0 volts. The POR is not a brown-out detector, low-voltage detector, or glitch detector.

SIM Reset Status Register (SRSR)

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	POR	PIN	COP	ILOP	ILAD	0	LVI	0
Write:								
POR:	1	0	0	0	0	0	0	0

 = Unimplemented

POR — Power-On Reset Flag

1 = Power-on reset since last read of SRSR

0 = Read of SRSR since last power-on reset

The SIM reset status register, SRSR, contains six flags that show the source of the last reset.

If the power-on reset bit, POR, is set, the last reset was caused by the POR circuit.

If the external reset pin bit, PIN, is set, the last reset was caused by an external device pulling the reset pin low.

If the Computer Operating Properly bit (COP) is set, the last reset was caused by the COP counter timing out before it was serviced.

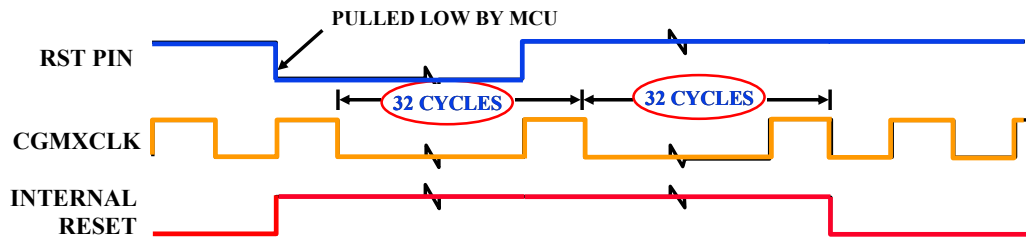
If the illegal opcode bit (ILOP) is set, the last reset was caused by an illegal opcode.

If the illegal address bit (ILAD bit) is set, the last reset was caused by an opcode fetch from an illegal address.

If the low voltage inhibit bit (LVI) is set, the last reset was caused by the LVI detecting a voltage below the selected trip point.

The flags in the SIM reset status register are cleared when the register is read. If the SIM reset status register has not been read after multiple reset sources, it may have multiple sources indicated. The register can help you to debug code problems. For example, if your application is unintentionally being reset, the reset source can be determined and corrective action taken. In the event of an in-application reset, the register helps you determine which action to take. The application may require additional set-up and initialization for a POR reset that is not required with a COP reset. Other reset sources may require unique diagnostics or servicing.

Internal Reset Timing



Now that we've identified the different sources of internal resets, let's look at internal reset timing.

All internal reset sources, except POR, pull the RST pin low for 32 CGMXCLK clock cycles to allow resetting of external devices.

The MCU is held in reset for an additional 32 CGMXCLK clock cycles after releasing the RST pin to allow external devices to stabilize. The RST pin is then tested to check if the RST pin is still being pulled low. If so, it indicates that an external reset has occurred. If not, it indicates that an internal reset occurred and the appropriate internal reset bit is set in the SIM reset status register.

Interrupt Processing Overview

Hardware Interrupt

- **Initiated by hardware pin or Module**
- **Uses an interrupt vector and a service routine**
- **Can be masked**

Software Interrupt (SWI)

- **Executed as part of the instruction flow**
- **Processed like a hardware interrupt**
- **Can't be masked**

Next, we'll discuss interrupt events.

Recall that interrupt events can be generated by either hardware or software sources.

A hardware interrupt is generated by internal or external hardware conditions and can be initiated by a hardware pin or Module. When a hardware interrupt occurs, the program context is stacked, the I-bit in the condition code register is set, and the interrupt vector is fetched.

Hardware interrupts can be masked. This means that hardware interrupts can be recognized only when the I-bit is cleared.

A software interrupt occurs as a result of the SWI instruction. A software interrupt is always executed as part of the instruction flow. The important difference between software and hardware interrupts is that software interrupts can't be masked. This means that the value of the I-bit has no effect on software interrupts. Otherwise, a software interrupt is processed the same way as a hardware interrupt.

Let's take a closer look at hardware interrupts sources.

Hardware Interrupt Sources

- **IRQ pin**
- **I/O port pins**
- **Timer Interface Module (TIM)**
- **SCI/SPI ports**

The IRQ pin can be used to trigger external hardware interrupts. Depending on the MCU configuration, an IRQ interrupt is generated by a logic low or a high-to-low transition on the IRQ pin. This type of interrupt can be used to monitor external systems or events.

In most 68HC08 MCUs, you can also generate an interrupt using additional I/O port pins. This is referred to as keyboard interrupts (KBI) and is commonly used to interface with key pad inputs. These inputs have programmable pullups that generate an interrupt when pulled low.

For example, a 16 key input pad is commonly organized as 4 rows by 4 columns. The four rows can be connected to the KBI inputs. When a key is pushed, the corresponding row input is pulled low, and an interrupt will be generated without glue logic. The interrupt service routine would debounce the key and determine which key was pressed by scanning the columns.

The 16-bit timer of the the Timer Interface Module (TIM) can generate several different interrupts depending on the particular model. The output compare, input capture, and timer overflow functions can generate interrupts. Some models also have a real-time interrupt feature. These types of interrupts can be used to process events based on a time reference.

For MCUs equipped with a Serial Communications Interface Module (SCI) or a Serial Peripheral Interface Module (SPI), the serial ports can generate a variety of interrupts depending on the model. SCI and SPI interrupts include receive register full, transmit register empty, and transmission complete. These types of interrupts can be used to process serial communications events. Other peripherals, such as the Controller Area Network (CAN) and the Universal Serial Bus (USB), can also generate interrupts.

Interrupt Sources

Source	Vector Address	Flag	Mask	INT Reg Flag	Priority
TimeBase	\$FFDC - \$FFDD	TBIF	TBIE	IF16	16
ADC Conv. Complete	\$FFDE - \$FFDF	COCO	AIEN	IF15	15
Keyboard Pin	\$FFE0 - \$FFE1	KEYF	IMASKK	IF14	14
SCI Trans. Complete	\$FFE2 - \$FFE3	TC	TCIE	IF13	13
SCI Transmitter Empty		SCTE	SCTIE		
SCI Input Idle	\$FFE4 - \$FFE5	IDLE	ILIE	IF12	12
SCI Receiver Full		SCRIF	SCRIFIE		
SCI Receiver Overrun	\$FFE6 - \$FFE7	OR	ORIE	IF11	11
SCI Noise Flag		NF	NEIE		
SCI Framing Error		FE	FEIE		
SCI Parity Error		PE	PEIE		
SPI Transmitter Empty	\$FFE8 - \$FFE9	SPTIE	SPTIE	IF10	10
SPI Receiver Full	\$FFEA - \$FFEB	SPRF	SPRIE	IF9	9
SPI Overflow		OVRF	ERRIE		
SPI Mode Fault		MODF	ERRIE		
TIM2 Overflow	\$FFEC - \$FFED	TOF	TOIE	IF8	8
TIM2 Channel 1	\$FFEE - \$FFEF	CH1F	CH1IE	IF7	7
TIM2 Channel 0	\$FFF0 - \$FFF1	CH0F	CH0IE	IF6	6
TIM1 Overflow	\$FFF2 - \$FFF3	TOF	TOIE	IF5	5
TIM1 Channel 1	\$FFF4 - \$FFF5	CH1F	CH1IE	IF4	4
TIM1 Channel 0	\$FFF6 - \$FFF7	CH0F	CH0IE	IF3	3
PLL	\$FFF8 - \$FFF9	PLLIF	PLLIE	IF2	2
IRQ	\$FFFA - \$FFFB	IRQF	IMASK1	IF1	1
SWI	\$FFFC - \$FFFD	None	None	None	0
Reset	\$FFFD - \$FFFF	None	None	None	0

Interrupt sources are serviced using a vector address and a priority. The table summarizes the 16 different interrupt sources associated with the 68HC908GP32 MCU. The 68HC08 architecture can handle up to 128 different reset and interrupt sources. For more information about interrupt implementations in specific 68HC08 derivatives, check the device technical data book.

Each interrupt source has its own unique vector address. This can eliminate the need of software polling within a service routine to determine the correct source of the interrupt, resulting in faster interrupt servicing.

Each source also has its own unique interrupt status register flags that can be polled. You can disable an interrupt source by resetting the source's unique interrupt bit.

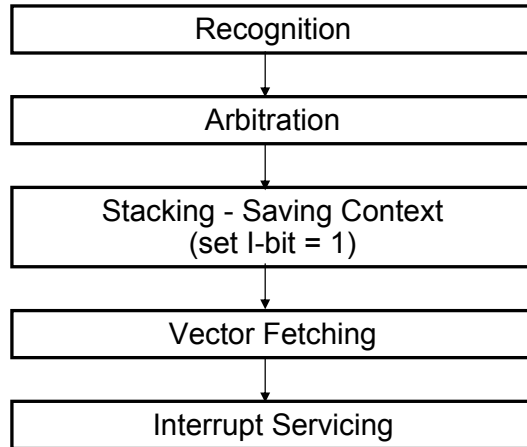
Each interrupt type has a pre-defined priority associated with it. The Timebase Module (TBM) interrupt has the lowest priority and the external IRQ has the highest priority. When multiple interrupts occur, the CPU looks at the priority of the events and services the event with the highest priority first, with the others pending. Notice that a reset event has priority over all interrupt events.

Recall that the global interrupt mask I-bit enables or disables all interrupt processing with the exception of the software interrupt, SWI. Interrupt events generated by on-chip peripheral Modules can be masked and are recognized only if the I-bit is cleared. These peripherals typically have local masks to enable or disable specific types of interrupts.

For example, the Timer Interface Module, TIM, has a separate interrupt mask and enable bit for overflow and for each timer channel.

Resets can't be masked, but some internal Modules can be disabled so that they can't generate a reset. The COP and LVI Modules are examples of potential reset sources that can be disabled out of reset.

Context Switching



Next, we'll cover how the SIM Module uses this information to service interrupts.

Determining which type of handling is required is called exception processing. Exception processing is handled through discrete tasks sometimes called "context switching". Exception processing is different for resets and interrupts. However, the processing tasks are the same.

First, the SIM Module recognizes the events and performs arbitration. The highest priority event is processed first. Before the vector is fetched, the I-bit is set to 1 to prevent further interrupt events and the current CPU context is saved on the stack. Finally, the interrupt service routine or exception handler is executed.

Let's take a closer look at the exception processing tasks, beginning with recognition.

Recognition


- **Resets**
 - **Recognized and acted on immediately**
- **Interrupts**
 - **Recognized during last cycle of current instruction**
 - **Acted on after last cycle of the current instruction**

During the recognition phase, all resets are recognized and acted upon immediately once asserted.

Interrupts are recognized during the last cycle of instruction execution. The timing of interrupt recognition depends on when the interrupt occurs. If an interrupt occurs before the last cycle of the current instruction, it will be recognized during the last cycle and then acted on. If an interrupt occurs during the last cycle of the current instruction, it won't be recognized until the last cycle of the next instruction.

Arbitration

Source	Priority
TimeBase	16
ADC Conv. Complete	15
Keyboard Pin	14
SCI Trans. Complete	13
SCI Transmitter Empty	
SCI Input Idle	12
SCI Receiver Full	
SCI Receiver Overrun	11
SCI Noise Flag	
SCI Framing Error	
SCI Parity Error	
SPI Transmitter Empty	10
SPI Mode Fault	9
SPI Overflow	
SPI Mode Fault	
TIM2 Overflow	8
TIM2 Channel 1	7
TIM2 Channel 0	6
TIM1 Overflow	5
TIM1 Channel 1	4
TIM1 Channel 0	3
PLL	2
IRQ	1
SWI	0
Reset	0



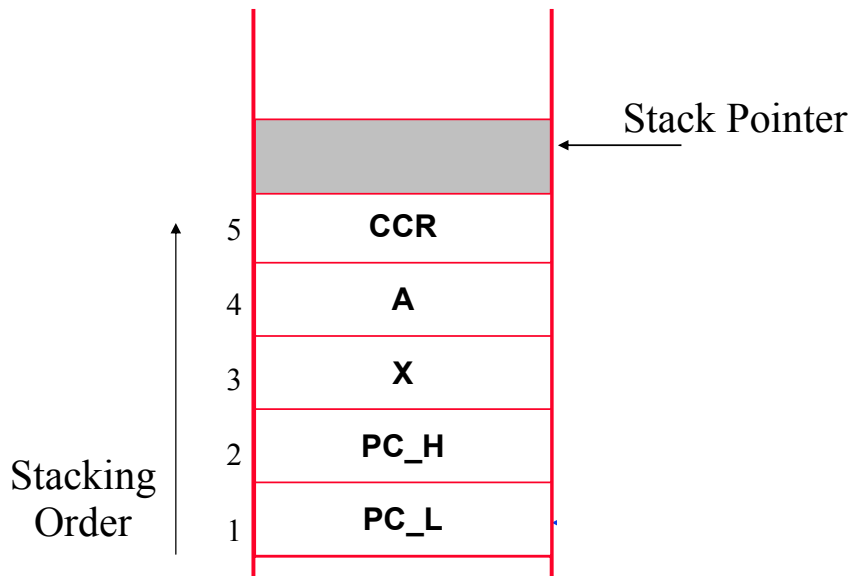
After recognition, the next phase is arbitration.

During the arbitration phase, the SIM Module determines the source of the exception and prioritizes the interrupt events for processing. All resets are considered equal and have higher priority over all interrupt sources. In the case of resets, no arbitration is required. Whichever reset occurs first will be acted on immediately and reflected in the SIM reset status register.

Interrupts are arbitrated using pre-defined priority levels associated with the different interrupt sources. The SIM provides the CPU with information about which sources generated the exception requests. The 68HC908GP32 has a total of 16 possible interrupt sources associated with the different on-chip Modules.

If the interrupt mask is cleared, the CPU will check all pending interrupts after every instruction. If more than one interrupt is pending when an instruction is complete, the highest priority interrupt will be serviced first. Once an interrupt is recognized as the highest priority interrupt, no new interrupt can take precedence regardless of its priority.

Stacking



Once the MCU has identified which interrupt to service, it sets the I-bit to prevent additional interrupt events while saving context. In the stacking phase, critical CPU information is saved to the stack. All CPU registers, including PC, X, A, and CCR are stacked. The H register is not stacked for 68HC05 compatibility reasons. Note that the stack pointer always points to the next available byte on the stack.

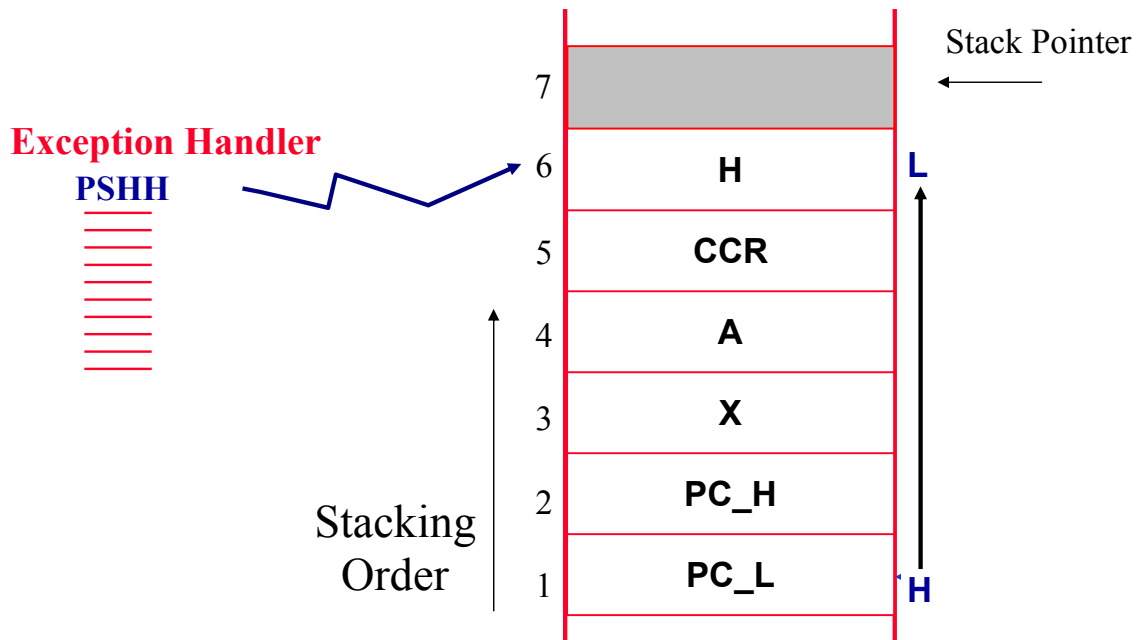
Because a reset event causes the CPU and stack to reset, stacking isn't performed for reset events.

Vector Fetching

Source	Vector Address
TimeBase	\$FFDC - \$FFDD
ADC Conv. Complete	\$FFDE - \$FFDF
Keyboard Pin	\$FFE0 - \$FFE1
SCI Trans. Complete	\$FFE2 - \$FFE3
SCI Transmitter Empty	
SCI Input Idle	\$FFE4 - \$FFE5
SCI Receiver Full	
SCI Receiver Overrun	\$FFE6 - \$FFE7
SCI Noise Flag	
SCI Framing Error	
SCI Parity Error	
SPI Transmitter Empty	\$FFE8 - \$FFE9
SPI Receiver Full	\$FFEA - \$FFEB
SPI Overflow	
SPI Mode Fault	
TIM2 Overflow	\$FFEC - \$FFED
TIM2 Channel 1	\$FFEE - \$FFEF
TIM2 Channel 0	\$FFF0 - \$FFF1
TIM1 Overflow	\$FFF2 - \$FFF3
TIM1 Channel 1	\$FFF4 - \$FFF5
TIM1 Channel 0	\$FFF6 - \$FFF7
PLL	\$FFF8 - \$FFF9
IRQ	\$FFFA - \$FFFB
SWI	\$FFFC - \$FFFD
Reset	\$FFFD - \$FFFF

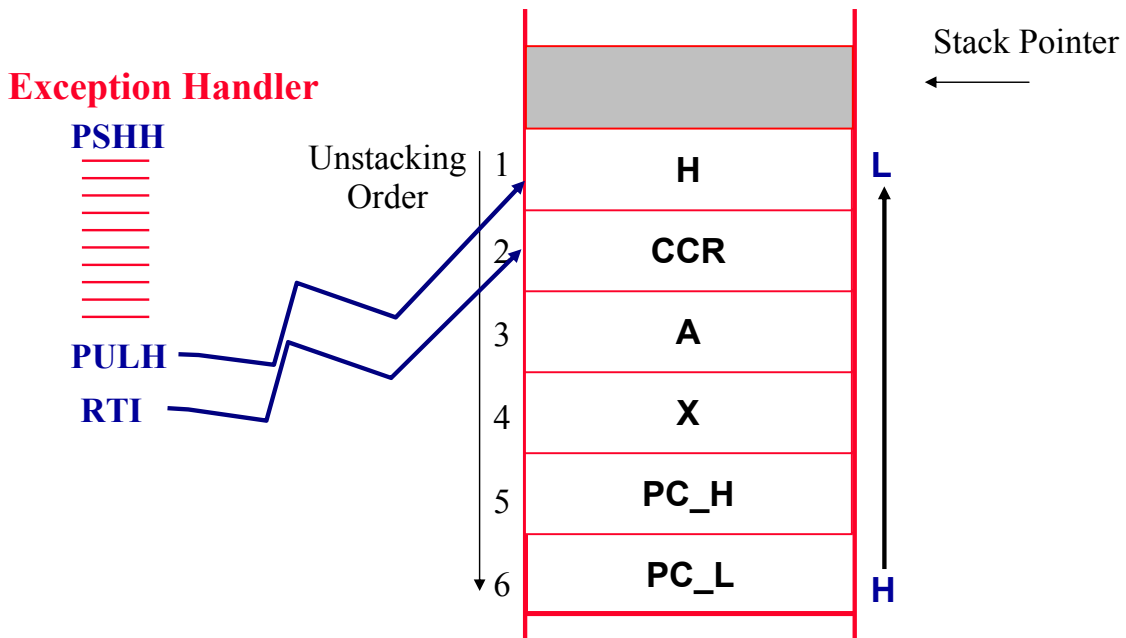
After all of the CPU registers are saved to the stack, the program counter is loaded with a user-defined vector address and begins processing. This 16-bit address is the start address for the interrupt service routine or exception handler.

Executing Exception Handler



Notice the stacking order of the registers. The registers are stacked from the highest to the lowest address. We mentioned before that the H register is not automatically saved by the CPU. Therefore, it must be saved at the beginning of the exception handler by executing a PSHH instruction. This means that we will have to restore it prior to exiting the exception handler.

Restoring Old Context



Before we restore the old context, note the order for the unstacking operation.

We start by executing the PULH instruction to restore H.

Next, execute the RTI instruction to restore the remaining registers, starting with the CCR register. Note that when the CCR register is restored to its original cleared condition, the global interrupt mask bit is also restored, enabling interrupts.

When the old context is restored, normal Operation of the application program is then resumed. If additional interrupts are pending, the process will begin again.

Example: Unused Interrupts Trap

```
;* Using a "Trap" with a COP Watchdog
;* Unused Vectors

TRAP:

    bra TRAP      ; wait for a COP reset
    org $1FF8     ; Timer Vector
    fdb TRAP      ; Points to TRAP
    org $1FFC     ; Software Interrupt
    fdb TRAP      ; Points to TRAP
```

A more fault-tolerant method for handling unused interrupt vectors is to use a trap with the COP watchdog timer. With the COP Module enabled, this method uses a loop to trap unwanted interrupts. The COP will reset the part after it times out. This provides a way to reset the MCU when an unexpected or spurious interrupt condition occurs.

Note that in some 68HC08 MCUs, the user-defined vector values are used for security. Assigning identical address values to unused interrupts may not be acceptable. If this is a consideration, you can duplicate the trap code in different memory locations to further enhance security.

Question

Which of the following exceptions can't be masked? Click on your BEST choice.

- a) Software interrupts
- b) TIM overflow
- c) SCI parity error
- d) Internal resets
- e) b and c
- f) a and d

Let's complete this tutorial with a few questions to check your understanding of the material. Which of the following exceptions can't be masked? Click on your best choice.

Answer: Both software interrupts and internal resets can't be masked. All hardware interrupts can be masked using the I-bit, and most hardware interrupt sources also have a local interrupt mask.

Question

When does the interrupt service routine begin executing?
Click on your choice.

- a) Immediately
- b) In the next clock cycle
- c) After the current instruction is finished executing
- d) During last cycle of the current instruction

When does the interrupt service routine begin executing? Click on your choice.

Answer: Interrupt service processing always begins after the current instruction is finished executing. First the CPU context is stacked, and then the first instruction of the interrupt service routine is executed.

Question

If these five hardware interrupts occurred at the same time, which interrupt event would be serviced first? Click on your choice.

- a) SCI Receiver Full
- b) PLL
- c) IRQ
- d) TIM 2 Channel 0
- e) ADC Conversion Complete

If these five hardware interrupts occurred at the same time, which interrupt event would be serviced first? Click on your choice.

Answer: Since the IRQ interrupt has the highest priority of the five interrupt sources, it would be serviced first. The five interrupts would be serviced in this order: IRQ, PLL, TIM 2 Channel 0, SCI Receiver Full, ADC Conversion Complete.

Tutorial Completion

- Reset and Interrupt Sources
- Reset Recovery
- MCU Exception Processing
- Interrupt Servicing

In this tutorial, you've had an opportunity work with resets and interrupts. You've learned about sources of resets and interrupts and the differences between these types of exceptions. You've also learned about MCU exception processing, including reset recovery and interrupt servicing.