Chapter 6: Interrupts and Resets

- Interrupts and resets are among the most useful mechanisms in a CPU.
- I/O operations are performed more efficiently, errors are handled more smoothly, and CPU utilization is improved.

Interrupt

- A special event that requires the CPU to stop normal program execution and perform some services related to the event.

- An ***external*** interrupt is generated when external circuitry asserts an interrupt signal to the CPU. (Example: the phone rings while you are studying.)

- An ***internal*** interrupt can be generated by the hardware circuitry or subsuystems (e.g., timers, I/O interface) inside the CPU or caused by software errors (e.g., illegal opcodes, overflow, divide by zero).

- Applicatrions of interrupts:
  - Coordinating I/O activities and preventing CPU from being tied up during the data transfer process. (Without the interrupt capability, the CPU would need to check the status of the I/O devices continuously or periodically.) For example, burning a DVD.
  - Providing a way to force the CPU to divert from normal program execution and take immediate action when an emergency event, e.g., power failure, occurs.
  - Providing a graceful way to exit from an operation that generates a software error so that the error can be corrected.
  - Reminding the CPU to perform routine tasks, such as keeping the time of day, periodic data acquisition, and task switching in a multiraking operating system.

- An interrupt request is said to be ***pending*** when it is active but not yet serviced by the CPU.

- Interrupts that cannot be ignored and require immedate actions by the CPU are called ***nonmaskable*** interrupts.

- Interrupts that can be ignored by the CPU are called ***maskable*** interrupts.
  - A maskable interrupt must be *enabled* by setting an enable bit before it can interrupt the CPU. Once a maskable interrupt is enabled, the CPU will then respond to it.

  - CPU normally provides two-level of interrupt enabling capabaility for better flexibility: a ***global*** and ***local*** interrupt masking capability.
  - When no interrupts is needed, all of the interrupts can be disabled by clearing the global interrupt enable bit (or flag).
  - A user can selectively enable certain interrupts and, at the same time, disable other undesirable interrupts by setting the enable bit of each interrupt source in addition to setting the global interrupt enable bit.

- Beause a CPU can only service one interrupt at a time, interrupt ***Priority*** is added, which is the order in which the CPU will service multiple pending interrupts when all of them occur at the same time.
  - Higher priority interrupts are being serviced before lower priority ones.
  - Interrupt prorities are fixed and not programmable by users in most MCU.

Interrupt Service Routines

- CPU provides services to an interrupt by executing a subroutine called the ***interrupt service routine*** (ISR). (Example: The steps you do if the phone rings.)

- A complete ISR cycle includes:
  1. Saving the PC value and CPU status (including the CPU status register and some other registers) in the stack.
  2. Identifying the cause (or source) of interrupt. (That is, where does the interrupt come from?)
  3. Resolving the starting address of the assoicate ISR.
  4. Executing the ISR.
  5. Restoring the CPU status and the PC value from the stack.
  6. Restarting the interrupted program.

- For all maskable (hardware) interrupts, the CPU starts to provide services after it has completed the execution of the current instruction.

- For some nonmaskable interrupts, the CPU may start the services intermedaitely without having completed the current instruction.

- Many software interrupts (e.g., divided by zero) are caused by an error in an instruction execution that prevents the instruction from being completed. The service to this type of interrupt is simply to output an error message and abort the program.

Interrupt Vector Table

- Interrupt **vector** refers to the ***starting address*** of the associated ISR.

- All interrupt vectors (addresses) are stored in an ***Interrupt Vector Table***, which is a section of memory loctions reserved for storing the start addresses of all associated ISRs. (Tell the CPU where to find the ISR.)

- Each type of interrupts share one inerrupt vecor (address). Each interrupt vector is located at a fixed location in the interrupt vector table.

- There are three methods for the CPU to determine the interrupt vector (i.e., the starting address of the ISR) before providing services:

  1. Predefined locations (Intel 8051 approach): the starting address of the ISR is predefined when the MCU is designed. The processor uses a table to store all the ISR.

  2. ***Fetching the vector from a predefined memory location*** (HCS12 approach): the interrupt vector of each interrupt source is stored at a predefined memory location in the interrupt vector table, where the CPU can get it directly.

  3. Executing an interrupt acknowledge cycle to fetch a *vector number* in order to locate the interrupt vector (Intel 68000 and x86 families): during the interrupt knowledge cycle, the CPU performs a read bus cycle, and the external I/O device that requested the interrupt places an **interrupt vector number** on the data bus to identity itself. The CPU figures out the starting address of the ISR by using this number. The CPU needs to perform a read cycle in order to obtain it. This method is not used by MCU due to this latency.

Interrupt Programming

After resolving the type of an interrupt, the HCS12 goes to the interrupt vector table to look at the content in the interrupt vector address for that interrupt. The content contains the (memory) address of the ISR of that interrupt. (This step tells the CPU where to find the ISR of the interrupt.)

The interrupt vectors in the table and the memory location and size of the table are fixed in the architecture of the MCU in use and cannot be changed by a user. However, the actual (memory) address of an ISR can be changed by the user by manually storing the actual address of the ISR into the associated interrupt vector in the table .

Interrupt programming deals with the procuedure of setting up the ISR of each enabled interrupt. (Example: the steps you perform if the phone rings while you are studying.)

Step 1. ***Initializing the interrupt vector*** by using **ORG**.

```
org     $xxxx       ; xxxx is the address of that interrupt in the vector table
dc.w    irq_ISR     ; store the starting address of the ISR of IRQ interrupt
```

Step 2. ***Writing the interrupt service routine***.

- This ISR should be as short as possible.
- For some interrupts, the ISR may only output a message to indicate that something unusual has occurred.
- Use **RTI** (the return-from-interrupt instruction) to return to the interrupted program. (Subroutine uses RTS).
- The ISR may or may not return to the interrupted program, depending on the cause of the interrupt.
  - If the interrupt is caused by a software error such as division-by-zero or overflow, the interrupted program is unlikely to generate correct results. Thus, the ISR should return to the monitor program or the operating system.

```
Irq_ISR     ldx     #msg        ; begin of the interrupt service routine
            jsr     putchar     ; call puts to output a string pointed by X to monitor
            rti                 ; return from interrupt

msg         fcc     “There is an error.”

putchar     …                   ; this is the routine that outputs the error message
```

Step 3. ***Enabling the interrupt to be serviced***, in the main program, by clearing the global interrupt mask (i.e., use the CLI instruction) and setting the local interrupt enable bit in the I/O control register of the associated interrupt source.

Overhead of Interrupts

The total overhead of the HCS12 interrupt is at least 17 to 20 E-cycles, which amounts to 1 μs for the 24MHz E-clock. The overhead includes:

- Saving the CPU status and other registers to the stack (at least 9 E-clock cycles).
- Execution time of instructions in the ISR.
- Execution time of RTI to restore all the CPU registers from the stack (8 to 11 E-clock cycles).

Resets

The reset mechanism establishes ***initial values*** (or conditions) of some CPU registers, flip-flops, and control registers in I/O interface chips must be established in order for the CPU to function properly.

- **Power-on reset** establishes the initial conditions of registers and flip-flops, and to initialize all I/O control registers.

- **Manual reset** without power-down allows the computer to get out of most error conditions (if hardware does not fail). The computer will reboot itself after a reset.

- Reset is ***nonmaskable*** and has the ***highest priority***. No registers are saved by resets.

- At the end of the service routine, control is transferred back to the monitor program or operating system.

HCS12 Interrupts and Resets

1. ***Maskable interrupts*** include the $\overline{\text{IRQ}}$ pin interrupt and all peripheral function interrupts.

2. ***Nonmaskable interrupts*** include the $\overline{\text{XIRQ}}$ pin interrupt, the SWI instruction interrupt, and the unimplemented opcode trap.

3. ***Resets*** include the power-on reset, the $\overline{\text{RESET}}$ pin manual reset, the computer operate properly (COP) reset (or the so-called watch-dog reset), and the clock monitor reset.

1. Maskable Interrupts

- Different numbers and types of peripheral (I/O) functions require different numbers of maskable interrupts.

- The ***I*** flag in CCR is the global mask of all maskable interrupts. (By default, **I=1**, which disables all maskable interrupts.) The CLI instruction is used to clear I and, in turn, to enable global interrupt.

- All maskable interrupts have a ***local enable bit*** that allow them to be selectively enabled. They are disabled by default.

- The priorities and vector addresses (ranging from $FF8C to $FFFF) of all HCS12 interrupts and resets are listed in the following tables.

- Higher vector addresses have higher priorities. (Reset has the highest priority.)

- The priorities of reset and nonmaskable interrupts (those in $FFF4-$FFFF) are unchangeable.

- The maskable interrupts are found from $FF8C to $FFF3. The (external) IRQ pin interrupt has the highest priority among all maskable interrupts.

Interrupt Vector Locations

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
\$FFFE, \$FFFF	Reset	None	None	–
\$FFFC, \$FFFD	Clock Monitor fail reset	None	PLLCTL (CME, SCME)	–
\$FFFA, \$FFFB	COP failure reset	None	COP rate select	–
\$FFF8, \$FFF9	Unimplemented instruction trap	None	None	–
\$FFF6, \$FFF7	SWI	None	None	–
\$FFF4, \$FFF5	XIRQ	X-Bit	None	–
\$FFF2, \$FFF3	IRQ	I-Bit	IRQCR (IRQEN)	\$F2
\$FFF0, \$FFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	\$F0
\$FFEE, \$FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE (C0I)	\$EE
\$FFEC, \$FFED	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	\$EC
\$FFEA, \$FFEB	Enhanced Capture Timer channel 2	I-Bit	TIE (C2I)	\$EA
\$FFE8, \$FFE9	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	\$E8
\$FFE6, \$FFE7	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	\$E6
\$FFE4, \$FFE5	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	\$E4
\$FFE2, \$FFE3	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	\$E2
\$FFE0, \$FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	\$E0
\$FFDE, \$FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2 (TOF)	\$DE
\$FFDC, \$FFDD	Pulse accumulator A overflow	I-Bit	PACTL (PAOVI)	\$DC
\$FFDA, \$FFDB	Pulse accumulator input edge	I-Bit	PACTL (PAI)	\$DA
\$FFD8, \$FFD9	SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	\$D8
\$FFD6, \$FFD7	SCI0	I-Bit	SC0CR2 (TIE, TCIE, RIE, ILIE)	\$D6
\$FFD4, \$FFD5	SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	\$D4
\$FFD2, \$FFD3	ATD0	I-Bit	ATD0CTL2 (ASCIE)	\$D2
\$FFD0, \$FFD1	ATD1	I-Bit	ATD1CTL2 (ASCIE)	\$D0
\$FFCE, \$FFCF	Port J	I-Bit	PTJIF (PTJIE)	\$CE
\$FFCC, \$FFCD	Port H	I-Bit	PTHIF(PTHIE)	\$CC
\$FFCA, \$FFCB	Modulus Down Counter underflow	I-Bit	MCCTL(MCZI)	\$CA

$FFC8, $FFC9	Pulse Accumulator B Overflow	I-Bit	PBCTL(PBOVI)	$C8
$FFC6, $FFC7	CRG PLL lock	I-Bit	CRGINT(LOCKIE)	$C6
$FFC4, $FFC5	CRG Self Clock Mode	I-Bit	CRGINT (SCMIE)	$C4
$FFC2, $FFC3	BDLC	I-Bit	DLCBCR1(IE)	$C2
$FFC0, $FFC1	IIC Bus	I-Bit	IBCR (IBIE)	$C0
$FFBE, $FFBF	SPI1	I-Bit	SP1CR1 (SPIE, SPTIE)	$BE
$FFBC, $FFBD	SPI2	I-Bit	SP2CR1 (SPIE, SPTIE)	$BC
$FFBA, $FFBB	EEPROM	I-Bit	EECTL(CCIE, CBEIE)	$BA
$FFB8, $FFB9	FLASH	I-Bit	FCTL(CCIE, CBEIE)	$B8
$FFB6, $FFB7	CAN0 wake-up	I-Bit	CAN0RIER (WUPIE)	$B6
$FFB4, $FFB5	CAN0 errors	I-Bit	CAN0RIER (CSCIE, OVRIE)	$B4
$FFB2, $FFB3	CAN0 receive	I-Bit	CAN0RIER (RXFIE)	$B2
$FFB0, $FFB1	CAN0 transmit	I-Bit	CAN0TIER (TXEIE2-TXEIE0)	$B0
$FFAE, $FFAF	CAN1 wake-up	I-Bit	CAN1RIER (WUPIE)	$AE
$FFAC, $FFAD	CAN1 errors	I-Bit	CAN1RIER (CSCIE, OVRIE)	$AC
$FFAA, $FFAB	CAN1 receive	I-Bit	CAN1RIER (RXFIE)	$AA
$FFA8, $FFA9	CAN1 transmit	I-Bit	CAN1TIER (TXEIE2-TXEIE0)	$A8
$FFA6, $FFA7	CAN2 wake-up	I-Bit	CAN2RIER (WUPIE)	$A6
$FFA4, $FFA5	CAN2 errors	I-Bit	CAN2RIER (CSCIE, OVRIE)	$A4
$FFA2, $FFA3	CAN2 receive	I-Bit	CAN2RIER (RXFIE)	$A2
$FFA0, $FFA1	CAN2 transmit	I-Bit	CAN2TIER (TXEIE2-TXEIE0)	$A0
$FF9E, $FF9F	CAN3 wake-up	I-Bit	CAN3RIER (WUPIE)	$9E
$FF9C, $FF9D	CAN3 errors	I-Bit	CAN3RIER (TXEIE2-TXEIE0)	$9C
$FF9A, $FF9B	CAN3 receive	I-Bit	CAN3RIER (RXFIE)	$9A
$FF98, $FF99	CAN3 transmit	I-Bit	CAN3TIER (TXEIE2-TXEIE0)	$98
$FF96, $FF97	CAN4 wake-up	I-Bit	CAN4RIER (WUPIE)	$96
$FF94, $FF95	CAN4 errors	I-Bit	CAN4RIER (CSCIE, OVRIE)	$94
$FF92, $FF93	CAN4 receive	I-Bit	CAN4RIER (RXFIE)	$92
$FF90, $FF91	CAN4 transmit	I-Bit	CAN4TIER (TXEIE2-TXEIE0)	$90
$FF8E, $FF8F	Port P Interrupt	I-Bit	PTPIF (PTPIE)	$8E
$FF8C, $FF8D	PWM Emergency Shutdown	I-Bit	PWMSDN (PWMIE)	$8C
$FF80 to $FF8B	Reserved			

- One of the maskable interrupts can be raised to the highest priority so that it can receive quicker service from the CPU. This is achieved by programming **bits 1~5** of the HPRIO register (at memory location **$001F**).

7	6	5	4	3	2	1	0	
PSEL7	PSEL6	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	0	$001F

Figure 6.1 Highest priority I interrupt register

- To raise a maskable interrupt source to the highest priority, simply write the assocated **HPRIO Value to Elevate** of that interrupt in the above tables to the HPRIO register. (For example, to raise the Timer Channel 5 interrupt to the highest priority, write the value of $E4 to HPRIO.)

$\overline{\text{IRQ}}$ Pin Interrupt

- The ***only external maskable*** interrupt signal for the HCS12. It is a physical pin on the MCU.
- The $\overline{\text{IRQ}}$ pin interrupt can be edge-triggered or level-triggered, selected by setting the IRQE bit of the interrupt control register (**INTCR**) at memory location $001E. (1 = responds only to falling edge; 0 = responds only to low level.)
- The $\overline{\text{IRQ}}$ pin interrupt has a local enable bit, IRQEN, in bit 6 of the INTCR. (1 = interrupt enabled; 0 = interrupt disabled.)

	7	6	5	4	3	2	1	0
	IRQE	IRQEN	0	0	0	0	0	0
reset:	0	1	0	0	0	0	0	0

IRQE -- $\overline{\text{IRQ}}$ edge sensitive only bit
IRQE can be written once in normal mode. In special modes, it can be written any time, but the first write is ignored.
1 = $\overline{\text{IRQ}}$ pin responds only to falling edge
0 = IRQ pin responds to low level.
IRQEN -- $\overline{\text{IRQ}}$ enable bit
IRQEN bit can be written any time in all modes. The $\overline{\text{IRQ}}$ pin has an internal pullup.
1 = $\overline{\text{IRQ}}$ pin interrupt enabled
0 = $\overline{\text{IRQ}}$ pin interrupt disabled

Figure 6.2 Interrupt control register (IRQCR)

- $\overline{\text{IRQ}}$ *Level-Sensitive (Active Low)* IRQE=0
  - Pros: allows multiple external interrupt sources to be tied to this pin. An interrupt request is detected whenever one of the interrupt sources is low.
  - Cons: need to remember to set $\overline{\text{IRQ}}$ pin back to 1 before the CPU exits the ISR if there are no other pending interrupts connected to the $\overline{\text{IRQ}}$ pin.

- $\overline{\text{IRQ}}$ *Edge-Sensitive (Falling Edge)* IRQE=1
  - Pros: no need to worry about the assertion time duration of the $\overline{\text{IRQ}}$ signal. (Reduce timing or synchronization problem of signals.)
  - Cons: not suitable for noisy environment because falling edges caused by noise spikes can be recognized as interrupt request on the $\overline{\text{IRQ}}$ pin.

- *When does the CPU recognize interrupt requests?*
  - First enable the $\overline{\text{IRQ}}$ pin interrupt. Then, an $\overline{\text{IRQ}}$ interrupt request can be recognized at any time after the I flag (bit 4) in the CCR is cleared.
  - CPU responds to the interrupt request only after it has completed the execution of the current instruction.
  - Interrupt latency varies according to the number of cycles required to complete the current instruction.

- *Stack Order on Entry of an Interrupt*
  - Before the CPU starts to service an interrupt, it will set the I flag (bit 4) in the CCR to disable other maskable interrupts.
  - When the CPU begins to service an interrupt, the instruction queue is refilled, a return address is calculated, and then the return address and the contents of all CPU registers (except SP) are saved in the stack.

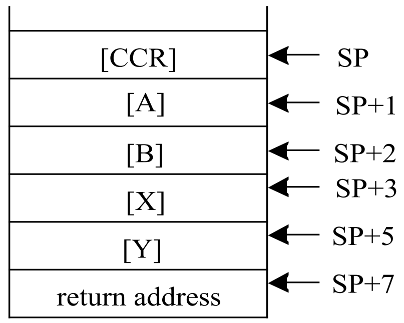

Figure 6.3 Stack order on entry to interrupts

- *Operating Sequence upon execution of the RTI Instruction*
  - Used to terminate ISRs.
  - Restore the CCR, A, B, X, and Y registers and the return address from the stack.
  - Clears the I flag (bit 4) in the CCR to enable further maskable interrupts.
  - CPU will continue to execute the interrupted program unless there is another pending interrupt.

2. Nonmaskable Interrupts

There are three nonmaskable interrupts: $\overline{\text{XIRQ}}$ pin, SWI instruction, and unimplemented instruction opcode trap.

$\overline{\text{XIRQ}}$ **Pin Interrupt**

- The $\overline{\text{XIRQ}}$ pin interrupt is disabled during a system reset and upon entering the ISR for an eariler $\overline{\text{XIRQ}}$ interrupt.
    - During system reset, both the I (bit 4) and X (bit 6) bits in the CCR are set to 1, disabling maskable interrupts and interrupt requests made by asserting the $\overline{\text{XIRQ}}$ pin to 0.
    - After minimal system initialization, software can clear the X flag (bit 6) of the CCR by using the **ANDCC #%1011 1111** instruction in order to re-enable the $\overline{\text{XIRQ}}$ interrupt.
    - Software cannot reset the X flag from 0 to 1 once the X flag has been cleared. Hence the interrupt requests made via the $\overline{\text{XIRQ}}$ pin become nonmaskable.

- When a nonmaskable interrupt is recognized, both the X and I flags are set after CPU registers have been saved.

- The execution of the RTI instruction at the end of the $\overline{\text{XIRQ}}$ service routine will restore the X and I flags to the pre-interrupt request state.

Unimplemented Opcode Trap

- HCS12 uses up to 16-bit (2 pages) to encode the opcode. All 256 combinations in page 1 opcode map are used, but only 54 of the 256 combinations in page 2 are used.
- If the CPU attempts to execute one of the 202 unused opcodes, an umimplemented opcode trap occurs.
- These unimplemented opcodes are essentially interrupts that share the same vector address $FFF8-$FFF9.

Software Interrupt Instruction (SWI)

- Execution of SWI causes an interrupt without an interrupt request signal.
- Cannot be inhibited by the global mask bits in the CCR.
- SWI is commonly used in the debug monitor to implement ***breakpoints*** and to transfer control from a user program back to the D-Bug12 monitor.

3. Resets

There are four possible sources of resets: power-on reset (POR), external reset ($\overline{\text{RESET}}$ pin), COP reset, and clock monitor reset.

Power-On Reset

- HCS12 has internal circuitry to detect a positive transition in the V_{DD} (power) supply and initialize the MCU by asserting the reset signal internally.
- The reset signal is released after a time delay for the clock signal to stabilize.

External Reset ($\overline{\text{RESET}}$ **pin**)

- Power-on reset and external reset share the same reset vector.
- HCS12 distinguish between internal and external resets by sensing how quickly the signal on the $\overline{\text{RESET}}$ pin rises to 1 after it has been asserted.
- Applictions: The on-chip EEPROM may be corrupted if the power supply drops below the required level. The common solution is to pull the $\overline{\text{RESET}}$ pin low to prevent instruction execution when the power supply drops too low.
    - A low-voltage inhibit circuit, such as the Motorola MC34064, can be used to protect against the EEPROM corruption.

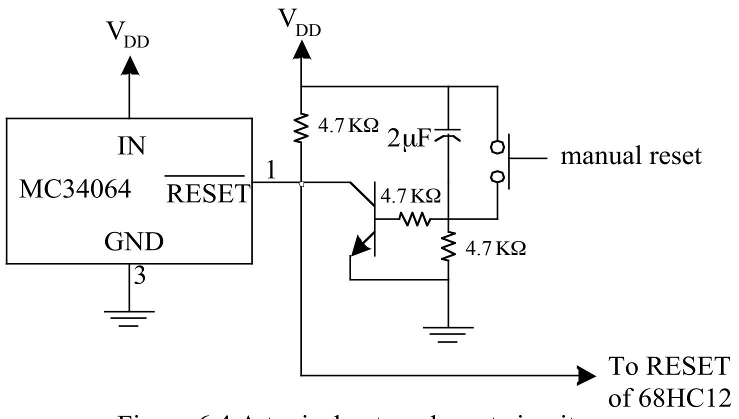

Figure 6.4 A typical external reset circuit

COP Reset

- The ***computer operate properly*** (COP) system, acting like a free-running watchdog timer, is designed to protect against software failures.
- If software was written correctly, the execution time can be predicted.
- When the COP is enabled, software must write \$55 and following by \$AA into the ARMCOP register (at memory location \$003F) to keep a watchdog timer from timing out.
- If software was not written properly, \$55 and \$AA may not be written to ARMCOP before the COP times out, and the COP will reset the software for you.
- The operation of the COP timer circuit and its timeout period are configured by the COPCTL register (at memory location \$0016).
- The COP system is driven by a constant frequency of $E/2^{14}$. The bits 2, 1, and 0 (CR2, CR1, and CR0, respectively) in the COPCTL register specify an additional division factor to calculate the COP timeout rate.

	7	6	5	4	3	2	1	0
	WCOP	RSBCK	0	0	0	CR2	CR1	CR0
reset:	0	1	0	0	0	0	0	0

WCOP: windowed COP mode bit
When set, a write to the ARMCOP register must occur in the last 25% of the selected period. A write during the first 75% of the selected period will reset the MCU.
0 = normal COP operation
1 = windowed COP operation
RSBCK: COP and RTI stop in active BDM mode bit
0 = allows the COP and RTI to keep running in active BDM mode
1 = stops the COP and RTI whenever the HCS12 is in active BDM mode
CR2:CR0: COP watchdog time rate select (number of OSCCLK cycles)
000: COP disabled
001: 2^{14}
010: 2^{16}
011: 2^{18}
100: 2^{20}
101: 2^{22}
110: 2^{23}
111: 2^{24}

Figure 6.17 CRG COP control register (COPCTL)

The following subroutine reset the COP.

```
cop_reset    ldaa    #$55         ; get first COP reset value
             staa    ARMCOP       ; store it
             ldaa    #$AA         ; get second COP reset value
             staa    ARMCOP       ; store it
             rts
```

Clock Monitor Reset

- The clock monitor reset circuit uses an internal RC circuit to determine whether the clock frequency is above a predetermined limit.
- If no OSCCLK clock edges are detected within this RC time delay, the clock monitor can optionally generate a system reset to indicate clock failure.
- The clock monitor function is enabled/disabled by the CME control bit in the COPCTL register.

Low Power Modes

- Power consumption is unavoidable in normal operation for an embedded system.
- Reduce the power consumption to minimum whenever a MCU is idle.

WAIT Mode

- The **WAI** instruction pushes all CPU registers (except the stack pointer) and the return address into the stack and enters a wait state to save power.
- During the wait state, CPU clocks are stopped (clock signals that drive the ALU and register file), but other clocks in the MCU (clock signals that drive peripheral functions) continue to run.
- CPU leaves the wait state when it senses maskable interrupts that are not masked, nonmaskable interrupts, or resets.
- Upon leaving the wait state, the CPU sets the appropriate interrupt mask bit(s), fetches the associated interrupt vector(s), and continues instruction execution at the location the vector (address) points.

STOP Mode

- When the **S** flag in the CCR register is set to 0 and the **STOP** instruction is executed, HCS12 saves all CPU registers (except SP) in the stack, stops all system clocks, and puts the MCU in a standby mode.
- Standby operation minimizes system power consumption.
- The contents of CPU registers and the states of I/O pins remain unchanged.
- Standby mode is ended by the assertion of $\overline{\text{RESET}}$, $\overline{\text{XIRQ}}$, or $\overline{\text{IRQ}}$ signals.
- If it is the $\overline{\text{XIRQ}}$ signal that ends the stop mode and the **X** flag is 0, instruction execution resumes with an interrupt vector fetched for the $\overline{\text{XIRQ}}$ interrupt.
- If X=1 (i.e., $\overline{\text{XIRQ}}$ disabled), a 2-cycle recovery sequence is used to adjust the instruction queue, and execution continues with the next instruction after the STOP instruction.

Real-Time Interrupt

- Main function is to generate periodic interrupt to the MCU, acting like a timer.

- The real-time interrupt (RTI) is enabled by the CRGINT register and the interrupt interval is selected by the RTICTL register with the user-selected interrupt periods.

- The real-time interrupt uses XTAL = 8 MHz clock.

	7	6	5	4	3	2	1	0
	RTIE	0	0	LOCKIE	0	0	SCMIE	0
reset:	0	0	0	0	0	0	0	0

RTIE: real time interrupt enable bit
0 = interrupt requests from RTI are disabled.
1 = interrupt requests from RTI are enabled.
LOCKIE: lock interrupt enable bit
0 = LOCK interrupt requests are disabled.
1 = LOCK interrupt requests are enabled.
SCMIE: self clock mode interrupt enable bit
0 = SCM interrupt requests are disabled
1 = Interrupt will be requested whenever the SCMIF bit is set

Figure 6.11 The CRG interrupt enable register(CRGINT)

	7	6	5	4	3	2	1	0
	0	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0
reset:	0	1	0	0	0	0	0	0

Figure 6.16 CRG RTI control register (RTICTL)

Table 6.4 RTI interrupt period (in units of OSCCLK cycle)

RTR[3:0]	RTR[6:4]							
	000 (off)	001 (2^{10})	010 (2^{11})	011 (2^{12})	100 (2^{13})	101 (2^{14})	110 (2^{15})	111 (2^{16})
0000 (÷1)	off*	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0001(÷2)	off*	2×2^{10}	2×2^{11}	2×2^{12}	2×2^{13}	2×2^{14}	2×2^{15}	2×2^{16}
0010 (÷3)	off*	3×2^{10}	3×2^{11}	3×2^{12}	3×2^{13}	3×2^{14}	3×2^{15}	3×2^{16}
0011 (÷4)	off*	4×2^{10}	4×2^{11}	4×2^{12}	4×2^{13}	4×2^{14}	4×2^{15}	4×2^{16}
0100 (÷5)	off*	5×2^{10}	5×2^{11}	5×2^{12}	5×2^{13}	5×2^{14}	5×2^{15}	5×2^{16}
0101 (÷6)	off*	6×2^{10}	6×2^{11}	6×2^{12}	6×2^{13}	6×2^{14}	6×2^{15}	6×2^{16}
0110 (÷7)	off*	7×2^{10}	7×2^{11}	7×2^{12}	7×2^{13}	7×2^{14}	7×2^{15}	7×2^{16}
0111 (÷8)	off*	8×2^{10}	8×2^{11}	8×2^{12}	8×2^{13}	8×2^{14}	8×2^{15}	8×2^{16}
1000 (÷9)	off*	9×2^{10}	9×2^{11}	9×2^{12}	9×2^{13}	9×2^{14}	9×2^{15}	9×2^{16}
1001 (÷10)	off*	10×2^{10}	10×2^{11}	10×2^{12}	10×2^{13}	10×2^{14}	10×2^{15}	10×2^{16}
1010 (÷11)	off*	11×2^{10}	11×2^{11}	11×2^{12}	11×2^{13}	11×2^{14}	11×2^{15}	11×2^{16}
1011 (÷12)	off*	12×2^{10}	12×2^{11}	12×2^{12}	12×2^{13}	12×2^{14}	12×2^{15}	12×2^{16}
1100 (÷13)	off*	13×2^{10}	13×2^{11}	13×2^{12}	13×2^{13}	13×2^{14}	13×2^{15}	13×2^{16}
1101 (÷14)	off*	14×2^{10}	14×2^{11}	14×2^{12}	14×2^{13}	14×2^{14}	14×2^{15}	14×2^{16}
1110 (÷15)	off*	15×2^{10}	15×2^{11}	15×2^{12}	15×2^{13}	15×2^{14}	15×2^{15}	15×2^{16}
1111 (÷16)	off*	16×2^{10}	16×2^{11}	16×2^{12}	16×2^{13}	16×2^{14}	16×2^{15}	16×2^{16}

Include the following statements in the main program to set up the real-time interrupt:

```
        movb  #$40, RTICTL       ; set RTI to about 1 ms
        bset  CRGINT, $80        ; enable RTI locally
        cli                      ; enable interrupt globally
```

Include the following subrotuine in the program to set up the ISR:

```
rti_ISR movb  #$80,CRGFLG        ; clear the RTIF flag to restart the RTI timer
        …                        ; put the ISR instructions here
        …                        ;
        rti                      ; end of the ISR
```

Finally, include the following statements at the end of the program to store the address of "rti_ISR" in the correct location of the interrupt vector table:

```
        org   $FFF0              ; RTI vetor stored here
        dc.w  rti_ISR
```

Clock and Reset Generation Block (CRG)

- CRG generates the clock (square waveform) signals required by the HCS12 instruction execution and all peripheral operations.

- Crystal oscillators, which generate sinusoidal waveform, are often used to generate clock signals. A circuit is used to square up the sinusoidal waveform.

- The CRG block also has a PLL circuit that can increase the frequency of the incoming clock signal, and stabilizes the frequency of its output signal.

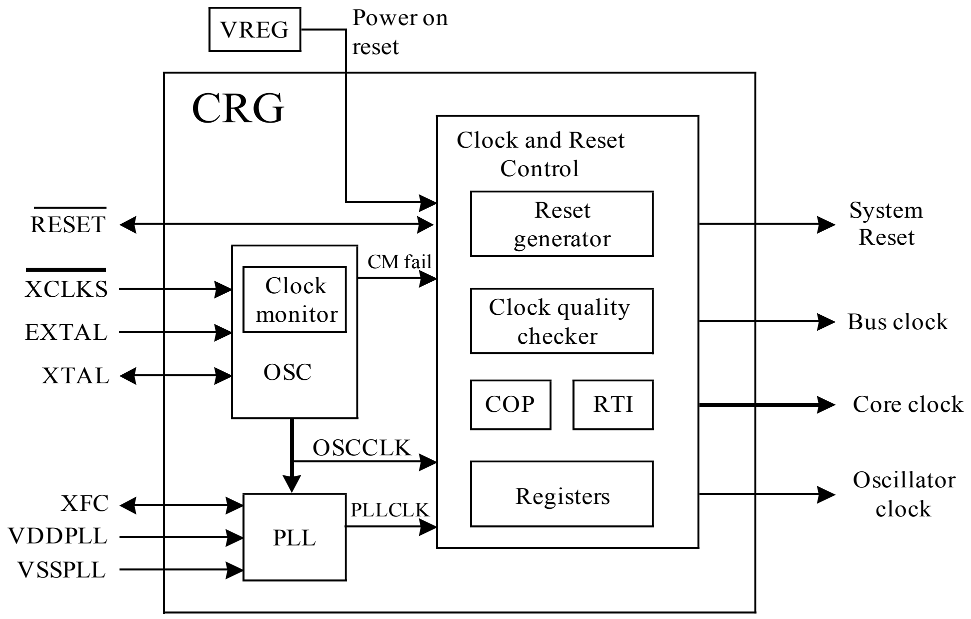

Figure 6.4 Block diagram of CRG

Choice of Clock Source

- User can choose between using the external clock (square waveform) or crystal oscillator (sinusoidal waveform) to produce the clock signal in the CRG.
- The $\overline{\text{XCLKS}}$ signal must be grounded to select the external clock signal.
- If an external oscillator is used, the two terminals of the oscillator circuit are connected to the EXTAL and XTAL pins as it needs an on-chip oscillator circuitry to square it up. If an external clock source is used, XTAl is not used.
- The output from the OSC module may bypass or go through the PLL circuit.
- Either the OSCCLK or the PLLCLK can be chosen as the SYSCLK which will be divided by 2 to derive the bus clock to control the instruction execution and peripheral operation.

Interrupts in D-Bug12 EVB Mode

- Most users would use the EVB mode of the D-Bug12 monitor to develop applicartions on a demo board.

- However, D-Bug12 monitor' EVB mode does not allow the use of the on-chip flash memory, thus preventing the use of the default interrupt-vector table to hold interrupt vectors (ISR address).

- To allow users to develop interrupt-driven applications, the D-Bug12 monitor provides SRAM-based interrupt vector table, starting at $3E00 and having 64 entries of 2 bytes each.

- All entries in this table contain the initial value of $1000. Storing a different value (address) in any one of the entries causes execution of the ISR pointed to by the address when an associated interrupt occurs.

Table 6.3 Mnemonic names for D-Bug12 RAM interrupt-vector addresses

Interrupt Source	RAM Vector Address	Vector number	Interrupt Source	RAM Vector Address	Vector number
UserRsrv0x80	$3E00	0	UserIIC	$3E40	32
UserRsrv0x82	$3E02	1	UserDLC	$3E42	33
UserRsrv0x84	$3E04	2	UserSCME	$3E44	34
UserRsrv0x86	$3E06	3	UserCRG	$3E46	35
UserRsrv0x88	$3E08	4	UserPAccBOv	$3E48	36
UserRsrv0x8a	$3E0A	5	UserModDwnCtr	$3E4A	37
UserPWMShDn	$3E0C	6	UserPortH	$3E4C	38
UserPortP	$3E0E	7	UserPortJ	$3E4E	39
UserMSCAN4Tx	$3E10	8	UserAtoD1	$3E50	40
UserMSCAN4Rx	$3E12	9	UserAtoD0	$3E52	41
UserMSCAN4Errs	$3E14	10	UserSCI1	$3E54	42
UserMSCAN4Wake	$3E16	11	UserSCI0	$3E56	43
UserMSCAN3Tx	$3E18	12	UserSPI0	$3E58	44
UserMSCAN3Rx	$3E1A	13	UserPAccEdge	$3E5A	45
UserMSCAN3Errs	$3E1C	14	UserPAccOvf	$3E5C	46
UserMSCAN3Wake	$3E1E	15	UserTimerOvf	$3E5E	47
UserMSCAN2Tx	$3E20	16	UserTimerCh7	$3E60	48
UserMSCAN2Rx	$3E22	17	UserTimerCh6	$3E62	49
UserMSCAN2Errs	$3E24	18	UserTimerCh5	$3E64	50
UserMSCAN2Wake	$3E26	19	UserTimerCh4	$3E66	51
UserMSCAN1Tx	$3E28	20	UserTimerCh3	$3E68	52
UserMSCAN1Rx	$3E2A	21	UserTimerCh2	$3E6A	53
UserMSCAN1Errs	$3E2C	22	UserTimerCh1	$3E6C	54
UserMSCAN1Wake	$3E2E	23	UserTimerCh0	$3E6E	55
UserMSCAN0Tx	$3E30	24	UserRTI	$3E70	56
UserMSCAN0Rx	$3E32	25	UserIRQ	$3E72	57
UserMSCAN0Errs	$3E34	26	UserXIRQ	$3E74	58
UserMSCAN0Wake	$3E36	27	UserSWI	$3E76	59
UserFlash	$3E38	28	UserTrap	$3E78	60
UserEEPROM	$3E3A	29	N/A	$3E7A	-1
UserSPI2	$3E3C	30	N/A	$3E7C	-1
UserSPI1	$3E3E	31	N/A	$3E7E	-1

Note. Vector number is used by the SetUserVector function to set up the interrupt vector .

- If an unmasked interrupt occurs and a table enrty contains the default adress of $1000, program exection is returened to D-Bug12 with a message indicating the source of the inerrupt and the CPU registers at the point where the program was interrupted.

- Only exception to this is the SCI0 interrupt, which it has been used D-Bug12 for all of its communications.

Setting Up the Interrupt Vector

- Assume that the label (or name) of the $\overline{\text{IRQ}}$ interrupt service routine is "irq_ISR" and the service routine is inside the assembly program. The interrupt vector must be stored in SRAM by using

```
        movw  #irq_ISR,$3E72        ; store the vector at the designated address
```

or

```
        org   $3E72
        dc.w  irq_ISR               ; this stores the memory address of the ISR
```

Example: (*Port H Interrupt*) Write an assembly program that blinks a LED continuously and, at the same time, wait for a DIP switch to be toggled from 1 to 0. The downward movement of the DIP switch generates an inerrupt and causes a buzzer to sound momentorary. (Port B is connected to eight LEDs and Port T pin 5 is connected to a buzzer in Dragon12-plus2. Port H is connected to DIP switches and also can be used to generate one kind of maskasble interrupt.)

```
#include "reg9s12.h"                ; this file contain addresses of all I/O ports

        org     $1000
d1      dc.b    1
d2      dc.b    1
d3      dc.b    1
d4      dc.b    1

        org     $2000
        lds     #$2000              ; set up the stack

        movb    #$FF,DDRB           ; set PB as output for LEDs
        movb    #$02,DDRJ           ; set PJ1 as output (required by Dragon12+)
        bclr    PTJ,$02             ; set PJ 1 to enable LEDs
        movb    #$02,DDRT           ; set PT5 as output for buzzer
        movb    #$00,DDRH           ; set PH as input for DIP switches
        movb    #$FF,PIEH           ; enable port H interrupt
        movb    #$00,PPSH           ; enable the interrupt (low) level-triggered
        cli                         ; enable global interrupt

again:  bset    PortB,$10           ; turn on LED4
        jsr     delay               ;
        bclr    PortB,$10           ; turn off LED4
        jsr     delay
        bra     again               ; waits for releasing pushbutton

pth_ISR:
        ldaa    #5                  ; set the duration of the buzzer turn on/off
        staa    d4
over    bset    PTT,$20             ; turn on the buzzer
        jsr     delay
        bclr    PTT,$20             ; turn off the buzzer
        jsr     delay
        dec     d4
        bne     over

        movb    #$FF,PIFH           ; enable interrupt for next round of interrupt
        rti                         ; return to the main program

delay   psha
        ldaa    #100                ; set for 100ms time delay
        staa    d3
; E-Clock = 24MHz. (1/24MHz) x10x240x10 =1ms time delay.
loop3   ldaa    #10
        staa    d2
loop2   ldaa    #240
        staa    d1
loop1   nop                         ; loop1 contains 10 E-cycles
        nop                         ; 1 E-cycle
        nop
        dec     d1                  ; 4 E-cycles
        bne     loop1               ; 3 E-cycles
        dec     d2
        bne     loop2
        dec     d3
        bne     loop3
        pula
        rts

        org     $3E4C         ; vector location (in D-Bug12) for Port H interrupt
        dc.w    pth_ISR       ; pth_ISR = label (address of the inerrupt routine)

        end
```

Eample: (*Real-Time Interrupt*) Write an assembly program to sound the buzzer continuously and, at the same time, wait for a real-time interrupt. This interrupt casues a LED to blink every one second. (Port B is connected to eight LEDs and Port T pin 5 is connected to a buzzer in Dragon12-plus2.)

```
#include “reg9s12.h”

        org     $2000
d1      dc.b    1
d2      dc.b    1
d3      dc.b    1
temp    dc.b    1
count   dc.b    1

        org     $2000
        lds     #$2000
        clr     temp
        clr     count

        movb    #$10,DDRB           ; set PB4 as output for LED4
        movb    #$02,DDRJ           ; set PJ1 as output (required by Dragon12+)
        bclr    PTJ, $02            ; set PJ1=1 to enable LEDs
        movb    #$20,DDRT           ; set PT5 as output for buzzer

        bset    CRGINT,$80          ; enable real-time interrupt by clearing the RTI flag
        bset    RTICTL,$7F          ; set the longest time delay
                                    ; =16x2^16/(8MHz) = 0.131s
        cli                         ; eanble global interrupts

again   bset    PTT, $20            ; turn on buzzer
        jsr     delay
        bclr    PTT, $20            ; turn off buzzer
        jsr     delay
        bra     again

rti_ISR:        ; uses XTAL=8MHz. The longest delay in each RT interrupt = 0.131s
        inc     count               ; set up ISR to count for 1 sec delay
        ldaa    count               ;
        cmpa    #8                  ; 8 x 0.131 sec = 1 sec.
        bne     over
        ldaa    temp
        eora    #$10                ; toggle the content of temp
        staa    temp                ; → toggle LED4 every second
        staa    PortB
        clr     count
```

```
over    bset    CRGFLG, $80         ; clear the RTI flag for next round of interrupt
        rti                         ;

delay   psha                        ;
        ldaa    #100                ; set for 100ms delay
        staa    d3                  ;
; E-Clock = 24MHz. (1/24MHz) x10x240x10 =1ms time delay.
loop3   ldaa    #10
        staa    d2
loop2   ldaa    #240
        staa    d1
loop1   nop                         ; loop1 contains 10 E-cycles
        nop                         ; 1 E-cycle
        nop                         ;
        dec     d1                  ; 4 E-cycles
        bne     loop1               ; 3 E-cycles
        dec     d2                  ;
        bne     loop2
        dec     d3
        bne     loop3
        pula
        rts

        org     $3E70           ; vector location (in D-Bug12) for RTI interupt
        dc.w    rti_ISR         ; rti_ISR = label (address of the inerrupt routine)

        end
```

Example: Assume that the $\overline{\text{IRQ}}$ pin of the HCS12DG256 is connected to a 1-Hz square waveform and port B is connected to eight LEDs in Dragon12-plus2. Write an assembly program to configure port B for output, enable the $\overline{\text{IRQ}}$ interrupt, and set up the service routine for the $\overline{\text{IRQ}}$ interrupt. The service routine for the $\overline{\text{IRQ}}$ interrupt simply increments a counter and outputs the value to LEDs.

```
#include “reg9s12.h”

                org     $1000
count           ds.b    1                   ; reserve one byte for count

                org     $2000
                lds     #$2000              ; set up the stack pointer
                clr     count
                movb  #$FF,DDRB    ; set port B for output
                bset  DDRJ,$02     ; set PJ1 pin for output (required in Dragon12)
                bclr  PTJ,$02      ; enable LEDs to light (required in Dragon12)

                movb  count,PortB  ; display the count value on LEDs

                movb  #$C0,INTCR   ; enable IRQ pin interrupt; (falling) edge-trigger
                cli                ; enable global interrupt
loop            bra   loop         ; wait for IRQ pin interrupt forever

irq_ISR         inc   count        ; increment count
                movb  count, PortB ; and display count on LEDs
                rti

                org   $3E72        ; set up interrupt vector in SRAM
                dc.w  irq_ISR

                end
```

End of Chapter 6