Notes - Unit 7

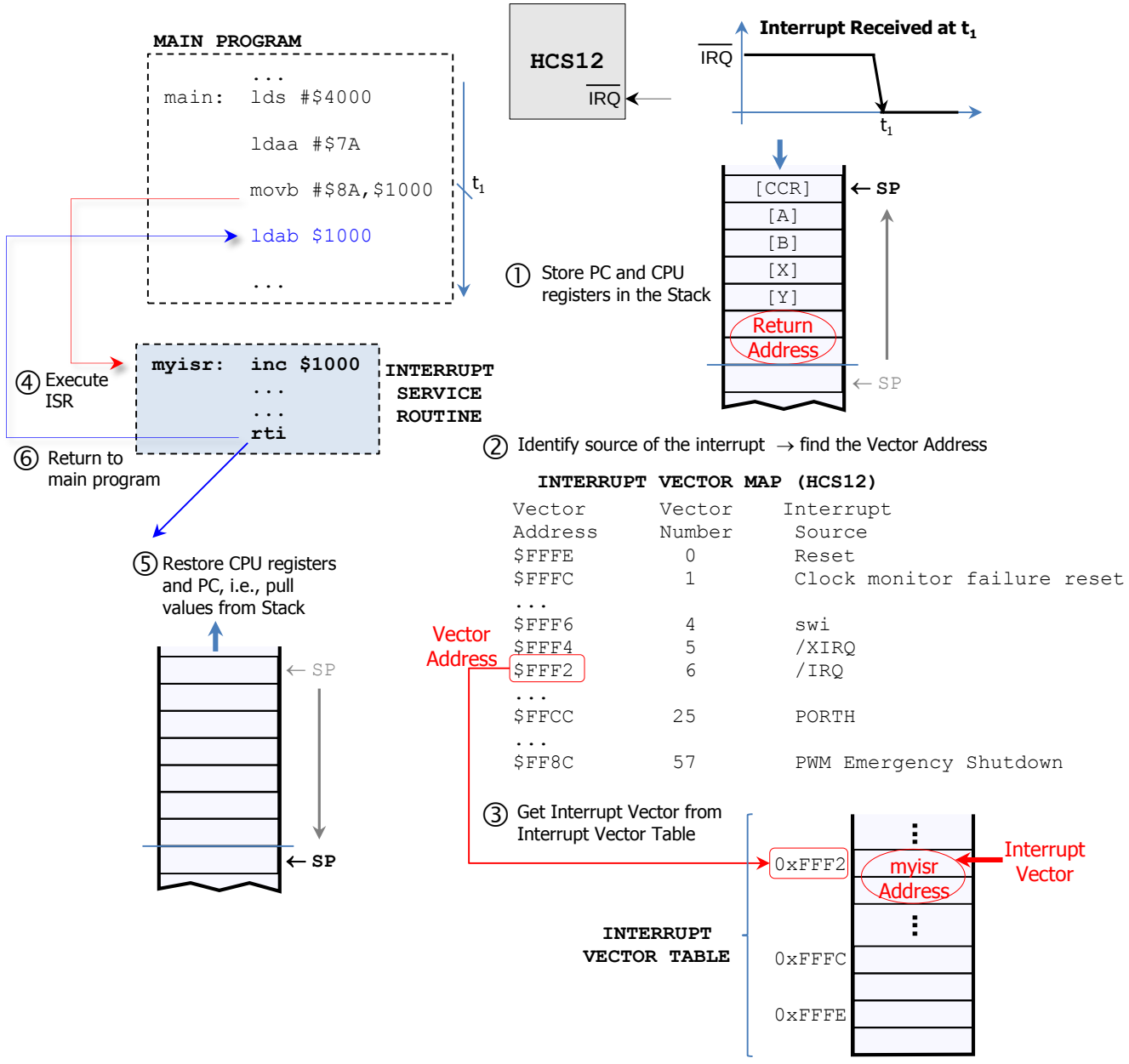
Interrupts and Exceptions

- **Interrupt**: Event that requires the CPU to stop normal program execution and perform some service (called *Interrupt Service*) related to the event. It can be generated externally (outside the chip), or internally (inside the chip).

Interrupt Service

The figure below depicts how an Interrupt is serviced. The example uses the HCS12 with the external interrupt `/IRQ`. At a time t_1, an event (falling edge) occurs on the `/IRQ` pin of the HCS12. This is how the processor attends the Interrupt:

1. Store the PC value (Return Address) and CPU Registers (`Y,X,B,A,CCR`) on the Stack.
2. Identify the source of the Interrupt. In the HCS12, this is akin to get the *Vector Address*. The subroutine that executes the service related to the interrupt is called the *Interrupt Service Routine*. The starting address of the *Interrupt Service Routine* is stored in a particular memory location in the HCS12, this location is known as the *Vector Address*.
3. Get the *Interrupt Vector*: This is the starting address of the *Interrupt Service Routine* (ISR). There is an Interrupt Vector for each type of interrupt. Interrupt Vectors are stored in a table called *Interrupt Vector Table*.
4. Execute the *Interrupt Service Routine* (ISR).
5. Pull CPU Registers (`CCR,A,B,X,Y`) and PC from the Stack.
6. Return to main program, specifically to the Return Address.



MASKABILITY

- **Maskable Interrupts**: The CPU has the option of disabling some or all these interrupts. When the interrupt is disabled, the CPU will ignore it. When the interrupt is enabled, the CPU must service this interrupt. The current instruction is completed before servicing the interrupt.
  To allow more flexibility, a microprocessor usually provides a global and local interrupt masking capability. We can disable all the maskable interrupts by configuring the global mask bit. We can also selectively enable certain interrupts while disabling others interrupts. This is achieved by providing each interrupt source an enable bit. Then, we only enable the interrupts that we want to be serviced.
- **Non-maskable Interrupts**: The CPU cannot ignore these interrupts and must service them. In some cases, the CPU may start interrupt service without completing the current instruction.

- ✓ **Interrupt Vector Table**: In the HCS12 processor, this is located in memory positions from `$FF8C` to `$FFFF`.
- ✓ **Interrupt Vectors**: They are programmable in the HCS12. In other processors (e.g., Intel 8051), they are fixed. In other processors (e.g., Intel x86), we first read an index that allows the processor to read the corresponding Interrupt Vector.
- ✓ **Interrupt Priority**: When there are multiple sources, several interrupts might be pending at the same time. The CPU is required to prioritize all interrupt sources. An interrupt with higher priority always receives service before interrupts with lower priorities. The HCS12 prioritize interrupts in hardware. In processors that do not prioritize interrupts in hardware, software can be written to handle priority of interrupts. In most processors, interrupt priorities are not programmable.

INTERRUPT PROGRAMMING

1. **Initialize Interrupt Vector Table**: Provide the starting address of the ISRs that the program needs to service.
2. **Write Interrupt Service Routine**: The Service Routine might not return to the interrupted program in some cases (e.g.: software error such as a divide by zero). In the HCS12, the `rti` instruction is used to return to the interrupted program.

- ✓ **Enable interrupts**: Globally and locally. Before the HCS12 starts to service an interrupt, it disables the global mask bit (bit I of CCR) by setting it to 1. When the HCS12 returns from an ISR, it enables the global mask bit by setting it to 0.

RESETS

- Before a computer can operate properly, the initial values of some CPU registers and I/O control registers must be established. A reset mechanism allows to establish these initial values.
- Typically, there are at least two types of reset in a microprocessor:
  - ✓ Power-on-Reset (POR): It establishes initial values of registers and initializes all I/O interface chips when power to the microprocessor is turned on.
  - ✓ Manual reset: Similar to the Power-on-Reset, but it is triggered by the user at any time. It allows the computer to get out of most error conditions. The processor restarts the main program after a reset.
- **Reset Service Routine**: Subroutine that executes the service related to the Reset. The Reset Service Routine is stored in the read-only memory of all microprocessors so that it is always ready for execution. At the end of the service routine, control is transferred to the monitor program or the operating system. The starting address of the Reset Service Routine is called the *Reset Vector*.
- **Reset Vector**: It is a fixed value or it is stored at a fixed location (in the HCS12, the Reset Vector is stored at `$FFFE`).
- Resets are non-maskable.
- Resets exhibit several properties that resemble non-maskable Interrupts. For example, the Interrupt Vector Table of the HCS12 includes the Reset Vectors. There is also a *Reset Service Routine*. However, the Reset Service does not save any registers in the Stack, as a Reset initializes those register values. Also, after the Reset is serviced, the program restarts (unlike an interrupt, where the ISR usually returns to the interrupted program).

HCS12 EXCEPTIONS (INTERRUPTS AND RESETS)

MASKABLE INTERRUPTS

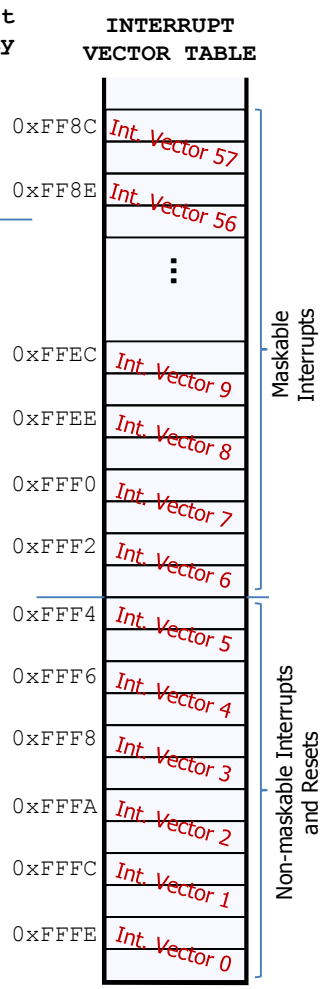
- We can mention: The `/IRQ` pin (PE1 pin in the MC9S12DG256), all peripheral function interrupts (e.g., `PORTH, PORTP`).
- `I` bit of `CCR`: Global mask of all maskable interrupts. If `I=1`, all maskable interrupts are disabled. If `I=0`, all maskable interrupts are enabled. At power on, or after a manual reset, the bit I is set to 1 (maskable interrupts disabled).
  - ✓ **Important**: Before servicing an interrupt, I is set o 1, disabling all other maskable interrupts during the ISR. When terminating an ISR, the I bit is restored (usually set to 0 to enable further maskable interrupts).
  - ✓ `cli` → Sets I to 0 (enables maskable interrupts) `sei` → sets I to 1 (disables all maskable interrupts)

INTERRUPT PRIORITY

- The Interrupt Vector Map (see Table 5.1 in Device User Guide) in the figure shows the default priority in the HCS12. The higher the address, the higher the priority (or the higher the vector number, the higher the priority)
- The priorities of the Resets and non-maskable interrupts are nonprogrammable. For the maskable interrupts, we can program one priority to be the highest.
- `HPRIO` register (`$001F`): It sets the priority of maskable interrupts. We write the lower byte of the Vector Address of the Interrupt we want to prioritize.

	Vector Address	Vector Number	Interrupt Source	HPRIO value to set to highest priority
Non-askable Interrupts and Resets	$FFFE	0	Reset	---
	$FFFC	1	Clock monitor failure reset	---
	$FFFA	2	COP failure reset	---
	$FFF8	3	Unimplemented instruction trap	---
	$FFF6	4	swi	---
	$FFF4	5	/XIRQ	---
Maskable Interrupts	$FFF2	6	/IRQ	$F2
	$FFF0	7	Real-time interrupt	$F0
	$FFEE	8	Enhanced capture timer channel 0	$EE
	$FFEC	9	Enhanced capture timer channel 1	$EC
	...		...	...
	$FFE0	15	Enhanced capture timer channel 7	$E0
	$FFDE	16	Enhanced capture timer overflow	$DE
	...		...	...
	$FFD2	22	ATD0	$D2
	$FFD0	23	ATD1	$D0
	$FFCE	24	PORT J	$CE
	$FFCC	25	PORT H	$CC
	...		...	...
	$FFB8	35	Flash	$B8
	$FFB6	36	CAN0 wake-up	$B6
	$FFB4	37	CAN0 errors	$B4
	$FFB2	38	CAN0 receive	$B2
	$FFB0	39	CAN0 transmit	$B0
	...		...	...
	$FF96	52	CAN4 wake-up	$96
	$FF94	53	CAN4 errors	$94
	$FF92	54	CAN4 receive	$92
	$FF90	55	CAN4 transmit	$90
	$FF8E	56	Port P Interrupt	$8E
	$FF8C	57	PWM Emergency Shutdown	$8C

PRIORITY



HPRIO ($001F):

	7	6	5	4	3	2	1	0
Read:	PSEL7	PSEL6	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	0
Write:								

- ✓ The interrupt produced by the /IRQ pin is set to the highest priority by default.
- ✓ For example, if we want to elevate the interrupt produced by PORTH to the highest priority, we first identify its vector address (0xFFCC). Then we set HPRIO = 0xCC (lower byte of the Vector Address).
- ✓ For example, if we want to elevate the Real Time Interrupt to the highest priority, we first identify its vector address (0xFFF0). Then we set HPRIO = 0xF0 (lower byte of the Vector Address).

/IRQ PIN INTERRUPT

- /IRQ pin: External maskable interrupt signal. We can configure the triggering method as well as the Local Enable. This is allowed by the INTCR ($001E) register (/IRQ Interrupt Control Register):

INTCR ($001E):

	7	6	5	4	3	2	1	0
Read:	IRQE	IRQEN	0	0	0	0	0	0
Write:								

- IRQE bit of INTCR: Selects the triggering method. If IRQE=1 → A falling edge on the /IRQ pin causes the interrupt (edge sensitive interrupt). If IRQE=0 → A low level causes the interrupt (low sensitive interrupt).

  Edge sensitive interrupt: Here, the user does not have to worry about how long the /IRQ is low. This approach might not be appropriate for a noisy environment as any noise spike could cause a falling edge.
  Low sensitive interrupt: This is useful if we want to connect multiple external interrupt sources to this pin. The user of this method must make sure that the /IRQ signal is de-asserted (goes high) before the processor exits the ISR.

- IRQEN bit of INTCR: Local interrupt enable. If IRQEN=1 → The /IRQ pin interrupt is enabled. If IRQEN=0 → The /IRQ pin interrupt is disabled. After a reset, this bit is set to 1 (/IRQ interrupt enabled)

Examples:

- **ASM Code:** `unit7a.asm`. This Assembly program increases an 8-bit count on `PORTB` each time the `/IRQ` ISR is serviced. The count starts at zero. The `/IRQ` interrupt is configured for falling edge.
  - ✓ Initialize Interrupt Vector Table: This is done by `ORG $FFF2` followed by `dc.w IRQISR`. `IRQISR` is the label that represents the starting address for the ISR of the `/IRQ` interrupt.
  - ✓ Write Interrupt Service Routine: Here, the ISR is called IRQISR. It consists of 3 instructions. The first two increments the count and writes it on PORTB. The last instruction `rti` returns to the main program.
  - ✓ Enable Interrupts: The `cli` instruction enables the Global Mask Bit (I in CCR) by clearing it. Locally, we configure the register `INTCR` (`$001E`) to enable `/IRQ` and making it respond to a falling edge: `movb #$C0, INTCR`.

```
            INCLUDE 'derivative.inc'
ROMStart    EQU  $4000
; variable/data section
            ORG RAMStart
count ds.b 1
; code section
            ORG   ROMStart
Entry:
_Startup:   LDS   #RAMEnd+1       ; initialize the stack pointer
            CLI                   ; bit I of CCR: Global Mask of all maskable interrupts
            ; I/O Configuration: Dragon12-Light Board
            movb #$FF, DDRB          ; set Port B to be all output
            movb #$00, DDRH          ; set Port H to be all input

showDIPSW:  ; Setting the IRQ interrupt vector to address IRQISR
            ; movw #IRQISR, $FFF2  ; Not allowed by CodeWarrior: use ORG and dc.w instead (see below)
            movb #$C0, INTCR ; Local enable for /IRQ. Falling edge triggers /IRQ interrupt

            clr count
            movb count, PORTB
forever:    nop
            bra forever; wait indefinitely for IRQ pin interrupt

; *************************
; Interrupt Service Routine
; *************************
IRQISR:     inc count
            movb count, PORTB
            rti
;*******************************************
;* Interrupt Vectors                       *
;*******************************************
         ORG $FFF2
         DC.W IRQISR      ; The IRQ Vector is the address 'IRQISR'
```

- **C Code**: `unit7b.c`. The following C program does the same as the previous ASM Code:

  - ✓ Initialize Interrupt Vector Table: This is taken care of by the following code, where the starting address of the ISR (called irqISR) will be stored at `0xFFF2` (which is the vector address of the `/IRQ` interrupt)

```
#pragma CODE_SEG DEFAULT /* change code section to DEFAULT (for Small Memory Model, this is $C000) */
// Interrupt Vector Table
typedef void (*near tIsrFunc)(void); // keyword in HCS12 so that the following is in nonbanked memory
const tIsrFunc _vect[] @0xFFF2 = { // 0xFFF2 is the address to store the IRQ interrupt vector
   /* Interrupt table */
   irqISR // Real Time Interrupt
};
```

  - ✓ Write Interrupt Service Routine: The ISR is called irqISR and it is function in C. The `interrupt` keyword identifies it as an ISR. As this function cannot have return values, the variables affected need to be global (`count` is a global variable). This function increments the count and writes it on PORTB, and then returns to the main program:

```
/* Interrupt Service Routine for /IRQ pin */
 interrupt void irqISR(void) // 'Interrupt' keyword: tells the C compiler that this function is an ISR
 {
  count = count + 1;
  PORTB = count;  // the rti instruction is automatically included at the end
 }
```

  - ✓ Enable Interrupts: The `EnableInterrupts` function enables the Global Mask Bit (I in CCR) by clearing it. Locally, we configure the register `INTCR` to enable `/IRQ` and to configure it as a falling edge: `INITCR=0xC0`.

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */

/* Program: Increases an 8-bit count on the LEDs every time a falling edge on IRQ pin is received. */
/* Global variables */
unsigned char count;

/* On CODE_SEG: This is a 'pragma' (directive) that specifies where the function segment is allocated.
                It affects function declarations and definitions
   #pragma CODE_SEG NON_BANKED --> Functions after this directive are stored in the NON_BANKED area
                                   (non-expanded memory). In particular, we are storing the ISRs there
   #pragma CODE_SEG DEFAULT --> Places the functions in the default code section
*/

/* List of functions */
#pragma CODE_SEG NON_BANKED

/* Interrupt Service Routine for /IRQ pin */
 interrupt void irqISR(void) // 'Interrupt' keyword: tells the C compiler that this function is an ISR
 {
  count = count + 1;
  PORTB = count;
  // the rti instruction is automatically included at the end
 }

 #pragma CODE_SEG DEFAULT /* change code section to DEFAULT (for Small Memory Model, this is $C000) */

 // Interrupt Vector Table
 typedef void (*near tIsrFunc)(void); // keyword in HCS12 so that the following is in nonbanked memory
 const tIsrFunc _vect[] @0xFFF2 = { // 0xFFF2 is the address to store the IRQ interrupt vector
    /* Interrupt table */
    irqISR // Real Time Interrupt
 };


void main(void) {
     EnableInterrupts; // asm("cli") // Enables all maskable interrupts
     INTCR = 0xC0;

  count = 0;
  DDRB = 0xFF;
  PORTB = count;

  for (;;); // infinite loop that waits for an interrupt
}
```

PORTH INTERRUPT

- PORTH interrupt: External edge-sensitive maskable interrupt. We can configure the triggering method as well as the Local Enable. This is allowed by the PIEH ($0026) register (PORTH Interrupt Enable Register) and the PPSH ($0025) register (PORTH Polarity Select Register). In addition, the register PIFH ($0027) register (PORTH Interrupt Flag Register) allows us to monitor whether there has been an active edge in a particular PORTH pin.

- PIEH: This register enables or disables on a per pin basis the edge sensitive external interrupt associated with PORTH. To determine which bit caused the interrupt, the user needs to read the PIFH register.

PIEH ($0026):

	7	6	5	4	3	2	1	0
Read: / Write:	PIEH7	PIEH6	PIEH5	PIEH4	PIEH3	PIEH2	PIEH1	PIEH0

Example: PIEH = 0x81. Bits 7 and bit 0 of PORTH will cause an interrupt.

- PPSH: This register selects the polarity on a per pin basis of an interrupt source. PORTH is only edge-sensitive. We can configure individual bits of PORTH to respond to either a falling edge or a rising edge. This is done by writing a 0 (falling edge) or 1 (rising edge) on the corresponding PPSH bits. If one or more edges are detected in the corresponding bits configured at PPSH, a '1' is written into the corresponding bit of the PIFH register.

PPSH ($0025):

	7	6	5	4	3	2	1	0
Read: / Write:	PPSH7	PPSH6	PPSH5	PPSH4	PPSH3	PPSH2	PPSH1	PPSH0

Example: PPSH = 0x42. Bits 6 and 2 are configured to respond to a rising edge. The rest as falling edge.

- `PIFH`: If an edge is detected on a `PORTH` pin and if the edge matches the one configured in the corresponding `PPSH` bit, the corresponding bit in `PIFH` is set to 1. An interrupt occurs if the corresponding enable bit in `PIEH` is 1 (enabled).
  In the corresponding Interrupt Service Routine, we need to clear the corresponding `PIFH` bit by writing a '1' on that bit. Otherwise, the interrupt will be activated forever.

PIFH (`$0027`):

	7	6	5	4	3	2	1	0
Read: / Write:	PIFH7	PIFH6	PIFH5	PIFH4	PIFH3	PIFH2	PIFH1	PIFH0

  Example: Using bit 3 of `PORTH` to issue an interrupt when a falling edge is detected.
  1. We write `0x08` on `PIEH` (enabling the interrupt on pin 3 of `PORTH`).
  2. We write `0xF7` on `PPSH` (configuring pin 3 of `PORTH` to respond to falling edges).
  3. In an interrupt in `PORTH` occurs, the state of `PIFH` will be `0x80`. Before finishing the interrupt service, we must clear the corresponding bit on `PIFH` by writing `0x08` on it (this will make `PIFH = 0x00`)

Example:

- **C Code**: `unit7c.c`. This C program increases an 8-bit count on `PORTB` each time the `PORTH` ISR is serviced. The count starts at zero. We are only using bit0 of `PORTH` to be the source of the interrupt when a falling edge is detected.

  ✓ Initialize Interrupt Vector Table: This is taken care of by the following code, where the starting address of the ISR (called porthISR) will be stored at `0xFFCC` (which is the vector address of the `PORTH` interrupt)

```
#pragma CODE_SEG DEFAULT /* change code section to DEFAULT (for Small Memory Model, this is $C000) */
// Interrupt Vector Table
typedef void (*near tIsrFunc)(void);
const tIsrFunc _vect[] @0xFFCC = { // 0xFFCC is the address to store the PORTH interrupt vector
    /* Interrupt table */
    porthISR // PortH Interrupt
 };
```

  ✓ Write Interrupt Service Routine: The ISR is called porthISR and it is function in C. The `interrupt` keyword identifies it as an ISR. As this function cannot have return values, the variables affected need to be global (`count` is a global variable). This function increments the count and writes it on PORTB, and then returns to the main program:

```
/* Interrupt Service Routine for /IRQ pin */
interrupt void porthISR(void)
{ // the Interrupt occurs when PORTH0 has a falling edge. Bit 0 PIFH is set to 1 if this happens
   count = count + 1;
   PORTB = count;
   PIFH = 0x01; // clears flag of bit 0 (activated because of the falling edge)
               // When an edge is detected in PORTH, the corresponding bit in PIFH is set to 1, and
               // then the interrupt occurs. If this bit is not cleared, the ISR will run indefinitely
}
```

  ✓ Enable Interrupts: The `EnableInterrupts` function enables the Global Mask Bit (I in CCR) by clearing it. Locally, we configure the register `PIEH` to enable `PORTH` interrupt (on bit 0): `PIEH = 0x01`. We also configure the register `PPSH` to detect falling edges: `PPSH=0x00`.

```
void main(void) {
     EnableInterrupts; // asm("cli") // Enables all maskable interrupts

     DDRH = 0x00;
     DDRB = 0xFF;

     // Setting bit 0 of PORTH as the Interrupt source
     PIEH = 0x01; // Port H Interrupt Enable Register
     PPSH = 0x00; // Falling edge on the bit 0 of PORTH. If PPSH = 0x01 -> Rising edge

  count = 0;
  PORTB = count;

  for (;;); // infinite loop that waits for an interrupt

}
```

NON-MASKABLE INTERRUPTS

- These interrupts are always serviced. The global mask `I` has no effect on these interrupts. So, a non-maskable interrupt can be serviced during the ISR of a maskable interrupt.

/XIRQ PIN INTERRUPT

- This non-maskable interrupt is activated if a low-level is detected on the `/XIRQ` pin (PE0 pin in the MC9S12DG256).
- During a reset, both the X and I bits in the CCR are set to 1. `I=1` disables all maskable interrupts. `X=1` disables the nonmaskable interrupt `/XIRQ`. After a reset, X and I remain set to 1.
- Likewise, when a non-maskable interrupt is serviced, both the X and I bits are set to 1. This prevents other interrupts from being recognized during the ISR. The `rti` instruction at the end of the ISR restores X and I to their pre-interrupt state.
- **Enabling /XIRQ Interrupt**: After a reset, note that X is disabled (`X=1`). We start with `/XIRQ` disabled because enabling the `/XIRQ` interrupt before a system is fully powered and stable can lead to spurious interrupts. We can then enable the `/XIRQ` interrupt by setting X to 0 (e.g.: `andcc #$BF`). Once `/XIRQ` has been enabled, we CANNOT disable it. Hence, `/XIRQ` is considered a non-maskable interrupt.

Example:

- **ASM Code:** `unit7d.asm`. This Assembly program increases an 8-bit count on `PORTB` every 250 ms. Each time the `/XIRQ` interrupt occurs, the `/XIRQ` ISR will set the count to $FF. The count starts at zero.
  - ✓ Initialize Interrupt Vector Table: This is done by `ORG $FFF4` followed by `dc.w isrXIRQ`. `isrXIRQ` is the label that represents the starting address for the ISR of the `/XIRQ` interrupt.
  - ✓ Write Interrupt Service Routine: Here, the ISR is called isrXIRQ. It sets the count to $FF and writes it on `PORTB`. The last instruction `rti` returns to the main program.
  - ✓ Enable Interrupt: Even though it is a non-maskable interrupt, we need to enable it: `andcc #$BF (X←0)`. Note that we cannot disable it afterwards.

```
            INCLUDE 'derivative.inc'
ROMStart    EQU  $4000
            ORG RAMStart
count ds.b 1
            ORG   ROMStart
Entry:
_Startup:   LDS   #RAMEnd+1       ; initialize the stack pointer
            movb #$FF, DDRB          ; set Port B to be all output
            movb #$00, DDRH          ; set Port H to be all input

            andcc #$BF; X <- 0. ; /XIRQ enabled. Once enabled, it cannot be disabled
            clr count

            ; Loop that increases a count from $00 to $FF every 250 ms.
forever:    inc count
            ldaa #$FA; A <- 250
            bsr asm_mydelay; Subroutine that executes a delay of A=250 ms

            movb count, PORTB; Display 'count' on the LEDs
            bra forever;

; SUBROUTINE asm_mydelay: Input: Contents of Register A. Outputs: none
; ----------------------
asm_mydelay: cmpa #$00; If A=0, we finish immediately
             beq next

; 1 ms loop with #400 on X for the 4 MHz bus speed.
oloop:       ldx #2400  ; #2400 (by default, it seems that the bus speed is set to 24 MHz)
iloop:       psha      ; 2 cycles
             pula      ; 3 cycles
             nop       ; 1 cycle
             nop       ; 1 cycle
             dbne X,iloop; ;3 cycles
             dbne A, oloop; 3 cycles
next:        rts

; Interrupt Service Routine:
; -------------------------
isrXIRQ:    movb #$FF, count
            movb count, PORTB
            rti

; Interrupt Vectors
            ORG $FFF4
            DC.W isrXIRQ      ; The XIRQ Interrupt Vector is the address 'isrXIRQ'
```

UNIMPLEMENTED CODE TRAP

- The OPCODE specifies the operation to be performed and the addressing modes used to access the operand(s). The OPCODE in the HCS12 consists of 1 or 2 bytes. Here we refer to Table A-2 in the HCS12 CPU Reference Manual Rev. 4.0.
- The first byte is specified in Page 1 of Table A-2. If the first byte is $18, it means that the OPCODE requires 2 bytes. The second byte is specified in Page 2 of Table A-2.
- For the second byte, we should expect 256 combinations. However, in the HCS12, only 54 out of 256 are actually used. The other 202 are unused numbers, and are not recognized as proper OPCODES (greyed out values in Table A-2)
- So, if an instruction used 2 bytes of OPCODE, and if the 2nd byte is an unused number, an interrupt is issued. This is called the Unimplemented Code Trap. All 202 unused 2nd bytes will cause an interrupt that share the same interrupt vector address: `$FFF8`. The HCS12 uses the next address after an unimplemented page 2 OPCODE as a return address.
- Note that the `X` bit does not have any effect on this interrupt. It is possible to activate this interrupt while in an ISR of a maskable or a non-maskable interrupt (including `/XIRQ`), though not recommended.

SOFTWARE INTERRUPT INSTRUCTION (SWI)

- Execution of this instruction causes an interrupt without an interrupt request signal. Execution of SWI disables all maskable interrupts (`I←0`). The value of I is restored when exiting the service routine (`rti` instruction).
- `swi` is commonly used for Debug operations. Breakpoints are inserted in a Program by inserting `swi` after a particular instruction. The `swi` ISR then displays information about instruction execution. The return address points to the next address after the OPCODE.
- Note that the X bit does not have any effect on this interrupt. It is possible to activate this interrupt while in an ISR of a maskable or a non-maskable interrupt (including `/XIRQ`), though not recommended.
- **CodeWarrior**: The Debugger uses '`swi`' to stop program execution at any time, and to execute the instructions line by line. We cannot use '`swi`' in the CodeWarrior Debugger, otherwise there would be no way to have a break in Debug Mode.

RESETS

The X, I bits have no effect on resets. Each reset has a separate vector. There are four possible sources of reset:

- **Power-On-Reset (POR):** The HCS12 incorporate circuitry to detect a possible transition in the V_{DD} supply and initialize the device, generally by asserting the reset signal to the internal circuits. The signal is typically released after a delay that allows the device clock generator to stabilize.
- **External Reset** (/RESET manual pin)**:** The Microcontroller Unit (MCU) distinguishes between internal and external resets by sensing how quickly the signal on the /RESET pin rises to logic level 1 after it has been asserted (set to 0).
  When any of the four reset conditions in reached, an internal circuitry drives the /RESET signal low for a number or cycles, then releases. A number of cycles later, the MCU samples the state of the signal applied to the /RESET pin. If the signal is still low, an external reset (manual) has occurred. If the signal is high, reset is assumed to have been initiated internally by either the COP (Computer Operating Properly) system or the Clock Monitor.
- **COP (Computer Operating Properly) Reset:** HCS12: The MCU includes a COP system to help protect against software failures. When the COP is enabled, software must write a particular code sequence ($55 followed by $AA) to the `ARMCOP` register to keep the COP from timing out and generate a reset. This sequence must be completed prior to the COP timeout period to avoid a reset. `COPCTL` register: Controls functioning and timeout period configuration.
- **Clock monitor reset:** The Clock Monitor circuit uses an internal RC circuit to determine whether clock frequency is above a predetermined limit. If clock frequency falls below the limit when the clock monitor is enabled, a reset occurs. To enable/disable the Clock Monitor, one must use the bit 7 (CME) of the `PLLCTL` (CRG PLL Control Register) register.

EXAMPLE: INITIALIZING AN INTERRUPT VECTOR TABLE FOR THE HCS12

HCS12 interrupts start at `$FF8C`. You need to enable the interrupts and write the corresponding ISRs. If only some interrupts are used, you can list all interrupts and then write a single ISR (only one instruction: `rti`) for the unused interrupts.

```
ORG $FF8C
dc.w isrPWM    ; PWM Emergency Shutdown
...
dc.w isrPORTH  ; PORTH Interrupt
dc.w isrPORTJ  ; PORTJ Interrupt
dc.w isrATD1   ; ATD1 Interrupt
dc.w isrATD0   ; ATD0 Interrupt
...
dc.w isrRTI    ; Real-Time Interrupt
dc.w isrIRQ    ; /IRQ interrupt
dc.w isrXIRQ   ; /XIRQ interrupt
dc.w isrSWI    ; Software interrupt
dc.w isrTRAP   ; Unimplemented instruction trap
dc.w isrCOP    ; COP Failure reset
dc.w isrCLKMON ; Clock Monitor
dc.w Entry  ; Reset. Entry: ASM initial address
```

```
#pragma CODE_SEG DEFAULT
typedef void (*near tIsrFunc)(void);
 const tIsrFunc _vect[] @0xFF8C = {
    isrPWM, // PWM Emergency Shutdown
    ...
    isrPORTH,  // PORTH Interrupt
    isrPORTJ,  // PORTJ Interrupt
    isrATD1,   // ATD1 Interrupt
    isrATD0,   // ATD0 Interrupt
    ...
    isrRTI,    // Real-Time Interrupt
    isrIRQ,    // IRQ interrupt
    isrXIRQ,   // /XIRQ interrupt
    isrSWI,    // Software interrupt
    isrTRAP,   // Unimplemented instruction trap
    isrCOP,    // COP Failure reset
    isrCLKMON, // Clock Monitor
    isrReset,  // Reset
 };
```

REAL TIME INTERRUPT

CLOCK AND RESET GENERATION (CRG) BLOCK

- This block generates the COP reset, the Clock Monitor Reset, the Real Time Interrupt, and the system clocks.
- In the Dragon12-Light Board, the crystal frequency is 8 MHz. The Oscillator Clock (OSCCLK) generated by the CRG Block has been set to have the same frequency as the crystal frequency. Thus, OSCCLK = 8 MHz.
- The CRG Block contains a Phase-Locked-Loop (PLL) circuit whose frequency is given by:

$$PLLCLK = 2 \times OSCCLK \times \frac{(SYNR + 1)}{(REFDV + 1)}$$

  where `SYNR` and `REFDV` are I/O registers.
- The System Clock SYSCLK (also called Core clock) generated by the CRG Block can be selected (via `CLKSEL` register) to be either PLLCLK or OSCCLK. Usually, PLLCLK is preferred.
- The E-clock (or *bus clock*) is equal to SYSCLK/2.
- Given the OSCCLK frequency (8 MHz in the Dragon12-Light Board), we can modify the Bus Clock (up to 24 MHz). By default, if the Dbug-12 Monitor is used, the Bus Clock is 4 MHz. If the Serial Monitor is used (CodeWarrior), the Bus Clock is 24 MHz).

Example: We want to modify our Bus Speed (or *bus clock*) from 4 MHz to 24 MHz, we must do:

- ✓ E-clock = bus speed = SYSCLK/2 = 24 MHz → SYSCLK 48 MHz.
- ✓ For 24 MHz bus speed, we need SYSCLK=PLLCLK = 48 MHz. Since OSCCLK is 8 MHz, we have:

$$48 = 2 \times 8 \times \frac{(SYNR + 1)}{(REFDV + 1)}$$

- ✓ There are many solutions. For example SYNR=2, REFDV=0. To program it, we need to do:
  a. Set SYNR to 2: `movb #$02, SYNR`
  b. Set REFVD to 0: `movb #$00, REFDV`
  c. Make SYSCLK=PLLCLK: `movb #$80, CLKSEL`; bit 7 of CLKSEL selects between PLLCLK and OSCCLK
  d. Disable clock monitor, enable PLL, set automatic bandwidth control, disable RTI and COP in pseudo-stop: `movb #$60, PLLCTL`
  e. Wait until PLL locks into the target frequency (i.e., when CRGFLG(3) is 1): `wait: brclr CRGFLG, $08, wait`

REAL TIME INTERRUPT (RTI)

- This is an important maskable interrupt. It generates a periodic interrupt. The frequency of the interrupt is given by:

$$OSCCLK / RTICTL$$

  Where `RTICTL` is the CRG RTI control register. The `RTICTL` value is the divider (see Table 6.4 in the textbook)
- Global Mask: We need to enable it, i.e., make `I=0`.
- Local Enable: This is configured in `CRGINT` (CRG Interrupt Enable) register, bit 7 (called RTIE). If RTIE=1, RTI is enabled; if RTIE=0, RTI is disabled. Also, if `RTICTL(2..0)=000`, RTI is disabled.
- `CRGFLG` (CRG Flag) Register: The bit 7 is set to 1 at the end of an RTI period. If this bit is 1 and if RTIE=1, an interrupt is generated. In the ISR of RTI, we must clear this bit by writing 1: `movb #$80, CRGFLG`, otherwise the interrupt will still be active after exiting the ISR.

Example:

- **ASM Code:** `unit7d.asm`. This Assembly program increases an 8-bit count on `PORTB` every 250 ms. RTI generates an interrupt at a frequency of 7.63 Hz (about every 130 ms). Each time the RTI interrupt occurs, we flash the RGB LED as yellow and checks whether `PORTH=$8E`. If it is, we do not exit the ISR (the count will be paused). If not, we exit the interrupt. The count starts at zero.
  - ✓ Initialize Interrupt Vector Table: We use `isrRTI` as the label that represents the ISR starting address:
    ```
    ORG $FFF0
    dc.w isrRTI
    ```
  - ✓ Write Interrupt Service Routine: The ISR is called `isrRTI`. We clear the bit 7 of `CRGFLG` register here:
    `movb #$80, CRGFLG`.
  - ✓ Enable Interrupts: The `cli` instruction enables the Global Mask Bit (I in CCR) by clearing it. Locally, we configure the register `CRGINT` to enable `RTI` (`movb #$80, CRGINT`). We also configure `RTICTL=$7F` so that it generates an interrupt every 130 ms, i.e. with a frequency of: $8\,MHz / 16 \times 2^{16} = 7.629\,Hz$.