Interrupts

Chapter 6

Interrupts concepts

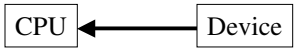
- Programs runs synchronously
  - CPU: fetch-execute cycle (clocked operation).
  - Activities performed according to order of instructions in program
- What is an interrupt?
  - Special event requiring CPU to stop normal program execution and perform some service related to that event.
  - Examples: I/O completion, time-out, illegal opcodes, arithmetic overflow, divide-by-0, etc.
- Interrupts allow for asynchronous execution of service routines
  - Example : Classroom questions.

Why are Interrupts Used ?

1. Coordinate the CPU with activities of I/O devices
   - I/O devices under control of their own circuitry
   - Data for I/O not under the control of CPU (fire alarm)
   - Data transfer require CPU and I/O device to synchronize
     - Device Response times: at least one order of magnitude slower than instruction execution
     - CPU response time: want immediate action, preventing CPU from being tied up
2. Remind the CPU to perform routine tasks.
3. Provide a graceful way to handle software/hardware errors

Interrupt Mechanism

- Originally appeared as synchronization mechanism between CPU - I/O
  - End of I/O: device informs data transfer successful.
  - Timer: a given amount of time elapsed
  - Parity error in memory (memory checking circuits).
  - Wrong Opcode: Control Unit detects non existing opcode
  - Requires a wired connection to the CPU



1. Hardware Interrupt : signal issued by circuitry (event)
2. CPU interrupt response: temporarily branches to execute services provided by Interrupt Service Routine (ISR) (executing program delayed for a while).
   - Hardware event triggers software response (internal CPU circuitry)

Event-Driven programming

- Interrupted processing: doesn't "know" it was interrupted
  - Processor:
    - temporarily suspends current thread of control
    - runs ISR
    - resumes suspended thread of control

- Polling: CPU asks devices whether there is anything to do.
  - sequential programming: next instruction determined by control transfer instructions.

- Interrupts: device tells CPU it is time to do something … NOW.
  - event-driven programming.
  - external hardware spontaneously cause control transfer (interruption in default program sequence).

Other sources of Interrupts : Resets, Exceptions, Traps

- Same circuitry has since been reused with other purposes
  - Make easier development of low level applications

Resets :

- ISR must preserve the interrupted programs' state
- Meant to re-establish initial conditions
  - ISR changes registers, flip-flops, I/O interfaces to initial values
  - Control transferred to a monitor or operating system.

- HC12 Resets
  - Power On Reset
  - COP Reset
  - Clock Monitor Reset

Other sources of Interrupts : Resets, Exceptions, Traps

Traps (System Calls)

- Request of service to Operating System/Monitor
- Standard mechanism to run dynamically loaded service routines
- Interrupt circuitry activated as result of the execution of an instruction
- Synchronous in nature because :
  - occur as result of execution of an instruction.
  - precise instant when interrupt is going to occur can be identified
- Also called Software Interrupts.
- HC12 Traps : `SWI`

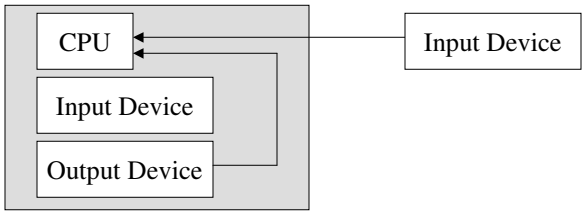
Other sources of Interrupts : Resets, Exceptions, Traps

Exceptions

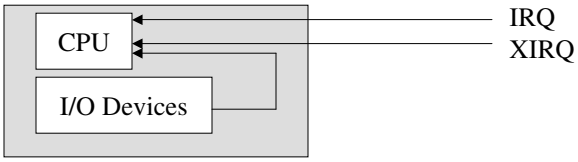
- Extension to the hardware error interrupts
- Used to signal software errors (e.g. Divide overflow)
- Provide a graceful way to exit or handle the error
- They are still “synchronous”
- They are implemented using the Hardware Interrupt circuitry
  - HC12 Exceptions: Unimplemented instruction trap

HW Interrupt Sources

- External : Circuitry is outside the system
- Internal : Circuitry is integrated (inside) the system



- HC12 Interrupt Sources
  - External : Only two : IRQ and XIRQ
  - Internal : Many, to be covered in coming lectures.



Interrupt Properties : Maskability

- Interrupt Maskability
  - Maskable interrupts can be ignored by the CPU
    - Enabled before interrupting the CPU (setting an associated enable flag).
  - NonMaskable interrupts cannot be ignored by the CPU

- Interrupt request *pending*: active but not yet serviced
  - May or may not be serviced (depending on whether or not it is enabled)

Global Interrupt masking

- Disables all maskable interrupt sources
  - Usually set by a flag : Interrupt Flag
- On the HC12

7	6	5	4	3	2	1	0
S	X	H	I	N	Z	V	C

Figure 2.8 Condition code register

- **I** bit (CCR) globally enables/disables all maskable interrupts
  - `SEI` Sets I bit (Disables all maskable interrupts)
  - `CLI` Clears I bit (Enables all maskable interrupts)
- Nonmaskable interrupts: not affected.

Local Interrupt masking

- Selective enabling/disabling of individual devices.
- Each interrupt source: an individual enable bit.
- HC12 Example : IRQ & the Interrupt Control Register (INTCR)

address: $1E	7	6	5	4	3	2	1	0
read: write:	IRQE	IRQEN	DLY	0	0	0	0	0
value after reset:	0	1	1	0	0	0	0	0

IRQE -- IRQ edge sensitive only bit
IRQE can be written once in normal mode.
1 = IRQ on falling edge 0 = IRQ on low level

IRQEN -- IRQ enable bit
IRQEN bit can be written any time in all modes.
1 = IRQ enabled 0 = IRQ disabled

DLY -- Oscillator startup delay on exit from stop mode

Figure 6.2 Interrupt control register (INTCR)

Interrupt Properties : When does the interrupt occur ?

- Level–sensitive: external interrupt generated as long as pin is high/low
  - Pros: multiple interrupt sources can be tied to this pin.
  - Cons: source must ensure line becomes inactive before IRQ's ISR is complete if only one interrupt request pending.
- Edge-sensitive: external interrupt generated when pin changes (either high-low or low-high)
  - Pros: no need for interrupt source to control duration of IRQ pulse.
  - Cons: not suitable for noisy environment (falling edge caused by noise will be recognized as an interrupt).

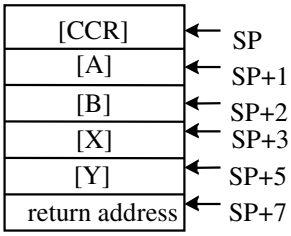
HC12 Interrupt Processing Sequence

- When does the MCU recognize interrupt requests?
- When it completes the execution of the current instruction
  - Why?
  - Instructions are atomic
  - Exception: fuzzy logic instructions (HC12 recognizes interrupt immediately). Why?
- For some nonmaskable interrupts, the CPU may start service without completing the current instruction
  - e.g. Unimplemented Instruction and `SWI`
  - Splitting hairs… can still rely on "instructions are atomic"

Interrupt Processing Sequence

On the HC12, the interrupt service cycle includes:

1. Save the PC value in the stack
2. Save CPU status (CCR and other registers) in the stack
   - All CPU registers are saved in the following order



3. Globally disable other maskable interrupts (Set the I mask)

Interrupt Processing Sequence

4. Identify the cause of interrupt
5. Resolve the starting address of the corresponding ISR
6. Execute the ISR
7. `RTI` instruction used to terminate ISRs.
   - Restore CPU status and PC from the stack (continue w/ interrupted program)
   - HC12 continues with the interrupted program unless another interrupt pending .
     - In that case, `RTI` "keeps" original stack and directly invokes pending interrupt ISR
     - When second ISR issues RTI, control return to the original interrupted program.

Resolving start address of ISR

Vector : start address of ISR

1. Predefined (ie. Address of ISR is fixed) (8051 approach)

2. Non/auto-vectored Interrupts: vector fetched from a predefined memory location (68HC12)
   - Interrupt vector table: table (array) of interrupt vectors, one for each supported interrupt source.

3. Vectored Interrupts: CPU executes an interrupt acknowledge cycle to fetch a vector number from external circuitry to locate the interrupt vector (68000 and x86 families)

HC12 Interrupt Sources

WARNING : Use HC12DP256 Manual

Vector Address	Interrupt Source
$FFFE	Reset (Power on)
$FFFC	Clock monitor reset
$FFFA	COP failure reset
$FFF8	Unimplemented instruction trap
$FFF6	SWI
$FFF4	!XIRQ
$FFF2	!IRQ
$FFF0	Real time interrupt
$FFEE..$FFE0	Timer channel 0 … 7
… and so on ($FF80).	

Store address of ISR here : *Install Vector*

Interrupt Properties : Priority

- What happens if two interrupts occur at the same time ?
  - CPU assigns priority to each potential source

- First, priority …
  - HC12 Priority : Assigned in order of their position in the vector table
    - Increasing priority at higher address
      - Reset : highest priority
      - Timer : lower priority

HC12 Interrupt Priority Control

Highest Priority Interrupt Register (HPRIO)

- Programs one to be the highest priority.
  - Load vector address into HPRIO register
  - Relative priorities of the others remain the same.
- Priority of resets and non-maskable interrupts: not programmable.

	7	6	5	4	3	2	1	0
read:	1	1	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	0
write:								
value after reset:	1	1	1	1	0	0	1	0

Figure 6.1 Highest priority I interrupt register

- e.g. To make the timer channel 0 the highest priority

```
LDAA    #$EE
STAA    HPRIO
```

Interrupt Properties : Priority

- What if an interrupt occurs during the execution of the previous ISR?
  - Answer: interrupt processing sequence
- **Scenario 1**: ISR for a low-priority interrupt is running when a higher-priority interrupt occurs.
- **Scenario 2**: ISR for a high-priority interrupt is running when a lower-priority interrupt occurs.
- **Scenario 3**: ISR is running when a reset occurs.

Steps of Interrupt Programming

Step 1. Initialize the interrupt vector table (Install the vector)

```
org $xxxx
FDB isr
```

Step 2. Write the interrupt service routine

```
isr:  …
      rti
```

Step 3. Enable the interrupt (if maskable)

- Globally enable maskable interrupts.
- Locally enable maskable interrupts

Developing Interrupt Software

- Loading vectors into reserved vector table addresses works fine for final systems or in simulator.

- When developing programs using evaluation board:
  - Boards that provide a debug monitor (e.g. Mon12) need their vector table in ROM
  - Allows RESET vectors to be installed
  - But how do we install our own ISRs?

Developing Interrupt Software (MON12 or NoICE)

When using the (Axiom) MON12 debug monitor …

ROM Vector Table			Monitor RAM Vector Table	
$FFFE		Reset		Not available
$FFFC		Clock Monitor	$3FFC	0000
$FFFA		COP Failure	$3FFA	0000
$FFF8		Unimpl Opcode	$3FF8	4002
$FFF6		XIRQ	$3FF6	0000
…			…	

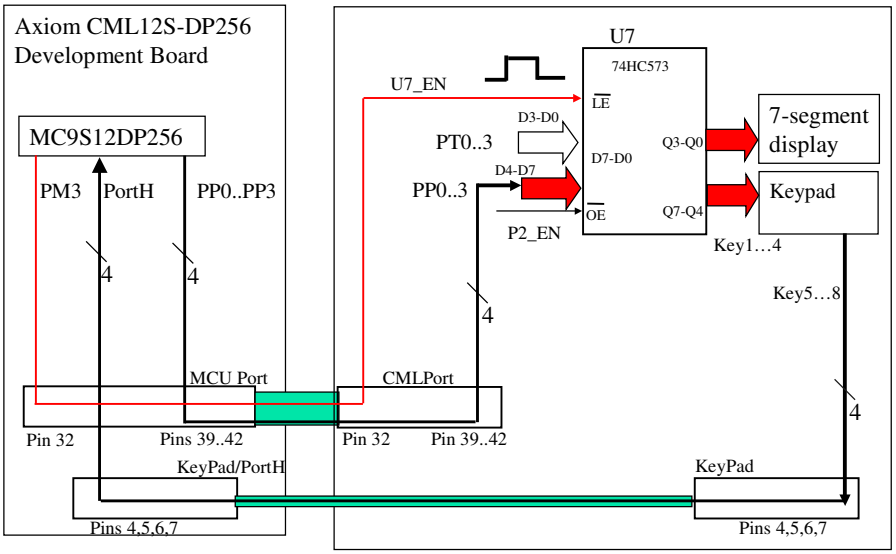
- Monitor duplicates supported interrupts sources in *RAM* Vector Table
  - All others go to ROM vector table (e.g., Reset)
- ISRs required to be located in $4000 - $7FFF
- See vectors_dp256.c (MON12) vectors_dp256_NoICE.c (NoICE)

Example: *Developing* Interrupt Software

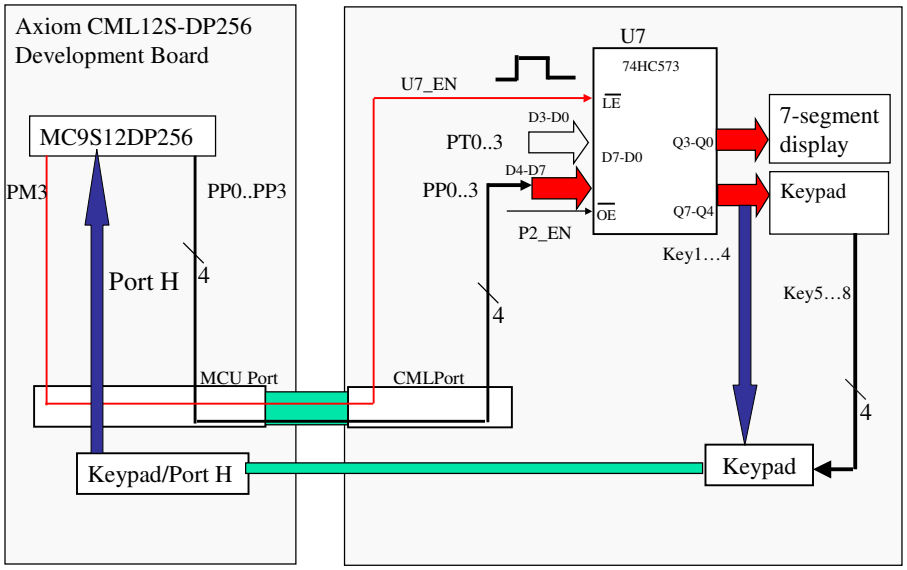
Install a Trap handler that increments a count of the number of Unimplemented Opcodes encountered.

```
                org $1000
1000 00         numInterrupts   db      0
    org $FFF8           ; Install ISR On the simulator
;   org $3FF8           ; Install ISR On the Axiom board
FFF8 400C       FDB     TRAP_ISR
                org $4000
4000 79 003C    main: clr     COPCTL ; Disable watchdog
4003 CF 3DFF          lds     #$3DFF ; Only in simulator
4006 10EF             cli            ; Enable interrupts
4008 41               db $41         ; COMA
4009 18 55            db $18, $55    ; No such instr!
400B 3F               swi
400C            TRAP_ISR:
400C 72 1000          inc     numInterrupts
400F 0B               rti
4010                  END
```

Keypad on the Axiom Project Board



Interrupt-Driven Keypad on the Axiom Project Board



Port H Key Wakeup

- When used as a general purpose I/O port :
  - Port H Input/Output Register (PTH)
  - Port H Data Direction Register (DDRH)
- When used as an interrupt source :
  - Port H Interrupt Flag Register (**PIFH**)
    - Bit is set when an active edge detected on associated pin
    - Also generates interrupt, if enabled
    - Read: To determine which pin is set
    - Write: To clear the pin (to acknowledge the interrupt)
  - Port H Interrupt Enable Register (**PIEH**)
    - Enables interrupts on associated pin
  - Port H Polarity Select Register (**PPSH**)
    - Sets the active edge of a bit as falling (0) or rising (1)
    - Also activates associated pull-up/down device, if enabled
  - Port H Pull Device Enable Register (**PERH**)
    - Enable(1) or disable (0) pull-up/down devices

Simple Keypad Driver

- Program echos all keystrokes on the serial channel
  - Runs forever
  - Interrupt-driven : Port H interrupts signal key press
  - Actual key press read using same method as in poll
    - Scan row (PP0..3) → Read the column (PH4..7)

```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Enable interrupts
  for (;;)    {
    while (char == $FF) {}
    print char;
    char = $FF
  }
```

```
char db $FF
```

```
KISR

find row,col
key=toKey(row,col)
ascii=convert(key)
char = ASCII

acknowledge int
```

Example : Interrupt Driven Keypad

```
        org $1000
char    db    $FF

        org $3FCC
fdb KISR

        org $4000
ldd   #BAUD19K
jsr   setbaud
bset  DDRP, #$0F  ; P0-3 as outputs (Key1..4=rows)
bclr  DDRH, #$F0  ; H4-7 as inputs (columns)
movb  #$0F, PTP   ; Set scan row(s)
movb  #$FF, PIFH  ; Clear all previous interrupt flags
movb  #$F0, PPSH  ; Rising edge
movb  #$00, PERH  ; Disable pull downs (enabled by Axiom)
movb  #$F0, PIEH  ; Local enable on columns inputs
cli               ; Global enable interrupts
```

Example : Interrupt Driven Keypad (ISR)

```
KISR:
   pshd
   ldaa PIEH    ; Disable Port H. Help with debouncing
   psha
   clr  PIEH

   ; Acknowledge interrupt (non-selective)
   movb #$F0, PIFH
   jsr findRowAndCol ; Returns Row in B, Col in A
   jsr translateKeyToASCII  ; Returns ASCII in A
   staa char

   pulb       ; Re-enablew PortH Interrupts
   stab PIEH
   puld
   rti
```

Example : Interrupt Driven Keypad (main continued)

```
again:
   ldaa #$FF

waiting:
   cmpa      char
   beq       waiting
   ldaa      char
   movb      #$FF, char

   tfr a,b                ; Transmit char over SCO
   jsr putChar_sc0

   bra       again
```

Programming Interrupts in C

- ISR must be declared as a special kind of subroutine that must be absolutely located.
  - ISRs cannot be called.

- pragmas: allow programmers to supply implementation-defined information to the compiler
  - Implementations should ignore information they do not understand
- ICC recognizes two interrupt-related pragmas

```
#pragma abs_address:0x3F8C
```
Statements following will be located at this absolute address.

```
#pragma interrupt_handler KISR()
```
Routine is identified as an ISR, not a callable subroutine.

vectors_dp256(_NoICE).c

- Must be added to project (containg other file(s) with main() & ISR)
- First : Identifies Addresses of all/any ISRs

```
#pragma nonpaged_function _start
extern void _start(void);      // Don't touch

extern void KISR(void);        // Example

#define NOICE_DUMMY_ENTRY(void (*)(void))0xF8CF
#define NOICE_XIRQ       (void (*)(void))0xF8C7
#define NOICE_SWI        (void (*)(void))0xF8C3
#define NOICE_TRAP       (void (*)(void))0xF8CB
#define NOICE_COP        (void (*)(void))0xF805
#define NOICE_CLM        (void (*)(void))0xF809
```

vectors_dp256(_NoICE).c

- Second, installs the addresses in the vector table.

```
//#pragma abs_address:0xFF80   // For Standalone
#pragma abs_address:0x3F8C // For monitor
void (*interrupt_vectors[])(void) =  {
   . . .
   NOICE_DUMMY_ENTRY,       /*CRG Lock*/
   NOICE_DUMMY_ENTRY,       /*PA B Overflow*/
   NOICE_DUMMY_ENTRY,       /*M.Down Counter Underflow*/
   KISR,                    /*Port H Interrupt*/
   NOICE_DUMMY_ENTRY,       /*Port J Interrupt*/
   NOICE_DUMMY_ENTRY,       /*ATD1*/
   NOICE_DUMMY_ENTRY,       /*ATD0*/
   …
   NOICE_COP,               /*COP failure reset*/
   NOICE_CLM,               /*Clock monitor fail reset*/
   _start,                  /*Reset*/
};
#pragma end_abs_address
```

C Example : Interrupt Driven Keypad

```
char getASCII( void );     // Gets key and converts to ASCII

#pragma interrupt_handler KISR()

void KISR(void) {
 char temp;

 PIEH = 0x00;          // Local disable

 temp = getASCII( );
 if (temp != 0x00) key = temp;

 PIFH = PIFH;          // Acknowledge (all) interrupts
 PIEH |= 0xF0;         // Local enable on columns inputs

}
```

C Example : Interrupt Driven Keypad (CKisr.c)

```
#include <hcs12dp256.h>
#include "stdio.h"
char key;
void main( void )  {
 char next;

 key = 0xFF;
 DDRP |= 0x0F;       // bitset PP0-3 as outputs (rows)
 DDRH &= 0x0F;       // bitclear PH4..7 as inputs (columns)
 PTP = 0x0F;         // Set scan row(s)
 PIFH = 0xFF;        // Clear previous interrupt flags
 PPSH = 0xF0;        // Rising Edge
 PERH = 0x00;        // Disable pulldowns
 PIEH |= 0xF0;       // Local enable on columns inputs
 asm( "cli" );
 for (;;){
  while (key == 0xFF );
  next = key;
  key = 0xFF;
  printf ( "%c", next );
 }
```