The Stack, Subroutines, Interrupts, and Resets: Ch3 Spasov

A. The stack

B. Subroutines

C. Interrupts

D. Resets

The stack is a special area in memory used by the CPU to store register information or general data information during program execution.

- The stack has a top and a bottom.
- The SP register is the special register that controls the address in the stack.
- The stack used is a LIFO structure that uses push and pull (pop) operations.
push operations:
- PSHA, PSHB, PSHX, and PSHY

pull operations
- PULA, PULB, PULX, and PULY

Sometimes we push the register data onto the stack when we need to load the registers with other values to perform certain operations...

---

**Stack as a Storage Buffer**

- Listing 3.1
- SUM OF SQUARES
- Demonstrate push and pull operations
- Program calculates $x^2 + y^2$.
- It gets $x$ and $y$ as 8-bit numbers from addresses $\$1031$ and $\$1032$.
- It puts the 8-bit result in ACCA.

```assembly
ORG $E000 ; start address of program
BEGIN
  LDS #$FF ; You must define the stack first
  LDAA $1031 ; Get first data, x
  TAB ;and square it
  MUL
  ADCA #$00 ; round it to 8-bit result
  PSHA ; and save it
  LDAA $1032 ; Get second data, y
  TAB ; and square it
  MUL
  ADCA #$00 ; round it to 8-bit result
  PULB ; retrieve first result
  ABA ; and add them
HERE BRA HERE ; For now, stop program
```
Using Subroutines

- A subroutine allows us to reuse code. It is a section of a program that may be used one or more times.
- The main program calls subroutines to perform certain steps.
- A data input/output process for subroutines is called parameter passing.
  - If there are not enough registers for parameter passing, a calling program can pass more parameters using stack.
- Ex: Modify the program from the previous slide to add many squares.

Subroutine Example

* SUM OF SQUARES: Demonstrate subroutines
* Program squares all the numbers in addresses $C000 to $C07F
* and puts the sum of the squares as a 16-bit result in IX.

ORG $E000 ;start address

* Start MAIN
MAIN LDS #$CFFF ;You must define
* ;the stack first squaring loop
LDX #$C000 ;init. data block pointer
LOOP1 JSR SQUARE ;square it and point to next data
PSHA ;save square
CPX #$C080 ;squared all data?
BNE LOOP1 ;get more if not
* ;summing loop
LDX #$0000 ;sum = 0
LDAA #$80 ;i=$80
LOOP2 PULB ;get square(i)
ABX ;sum = sum + square(i)
DECA ;i=i-1
BNE LOOP2 ;repeat until i == 0
HERE BRA HERE ;stop program
* End of MAIN
*
**Subroutine Example**

* SUBROUTINE SQUARE
* calculates the square of an 8-bit number
* as a rounded 8-bit normalized result
* data pointer (IX) increments
* calling registers:
  * IX = address of data to be squared
* return registers:
  * ACCA = 8-bit square
  * IX = IX+1
  * ACCB is affected
  * others unaffected

```
SQUARE
   LDAA $0,X
   TAB
   MUL
   ADCA #$00 ;round it to 8-bit result
   INX
   RTS
```

---

**Process of Calling Subroutines**

[Diagram showing the process of calling subroutines]
Example: Subroutine Using the Stack

* Listing 3.6
* Demonstrate a convention for using subroutines
* using a modification of Subroutine SQUARE
* (See Listing 3.2). Program squares all the numbers
* in addresses $C000 to $C07F
* Start MAIN

MAIN
LDS #$FF ;You must define the stack first
; squaring loop
LDX #$C000 ;init. data block pointer
; IX is the calling register
LOOP1
JSR SQUARE2 ;square it and note that
; original IX value returned
INX ;point to next data
CPX #$C080 ;squared all data?
BNE LOOP1 ;get more if not
HERE BRA HERE ;stop program
* End of MAIN

Example: Subroutine Using the Stack

* SUBROUTINE SQUARE2
* calculates the square of an 8-bit number as a rounded 8-bit result
* data pointer (IX) increments
* calling registers: IX = address of data to be squared
* return registers: IX = address of 8-bit square
* CCR affected; others unaffected
SQUARE2
PSHA ;preserve registers
PSHB ;note that following instructions modify
; ACCA and ACCB
LDAA $0,X ;get data to square
TAB ;copy it to ACCB
MUL ;square it
ADCA #$00 ;round it to 8-bit result
STAA $0,X ;store result
PULB ;restore registers
PULA ;contain their original values
RTS ;return, note that ACCA and ACCB contain their original values
Example: Subroutine Using the Stack

- The number of pushes must be equal with the number of pulls

What are interrupts?

An interrupt is the automatic transfer of software execution in response to hardware that is asynchronous with the current software execution.

The hardware can be either:
1. An external I/O device (i.e., printer)
2. An internal event (i.e., periodic timer, opcode fault)

Interrupts allow us to have multiple background threads

A thread is defined as the path of action of software as it executes.
**Process** is a program in execution

![Image]

The entity that can be assigned to and executed on a processor.

The following are associated with a process:

1. A virtual address space that holds process image, files, I/O resources.
2. Protected access to processors, other process files, I/O resources.
3. Process has an execution state (running, ready...)
4. And a dispatching priority and is the entity that is scheduled and dispatched by OS

Within a process, there may be one or more threads:
- A thread execution state
- A saved thread context (when not running)
- An execution stack
- Access to the memory and resources of its process, shared with all other threads in that process

---

**Interrupt driven I/O**

- Here the processor issues an I/O command to a module and continues to do some other useful work.
- I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor, e.g., disk controller.
- Processor then executes data transfer, sleeps before resuming its former processing.

Adv: Interrupt driven I/O is more efficient than programmed I/O, **because** it eliminates needless waiting!!!

But, it still consumes a lot of processor time, **because** every word of data that goes from memory to I/O module must pass through processor.
Typical Program Execution

- Without interrupts the program executes continuously
A reset is a special type of interrupt
Unlike an interrupt, it does not return to the interrupted program
A reset stops execution of the application program to do a special function, such as reinitializing the registers and memory
The 68HC11 has four possible types of reset

- External RESET^ pin
- Power-on reset (POR)
- Computer operating properly (COP)
- Clock monitor reset

Types 1 and 2 are external resets and types 3 and 4 are fault tolerant resets or internal resets.

Other differences between resets and interrupts are:
- Reset exception is immediately recognized asynchronously to the clock
- MCU stays in the reset state as long as reset signal is active
- An interrupt is only sampled by the CPU at the end of instr. seq.

Q: How does MCU knows the source of RESET?

Interrupt

- An interrupt suspends the execution of an application to do something else – it is triggered by an interrupt request
- An interrupt service routine (ISR) is initiated
  - RTI instruction has to be at the end of the ISR to load registers
Interrupt Vector

- All resets and interrupts use vectors
- A vector indicates the start address of reset or interrupt routines
- A vector address is a 2-byte memory location that stores a vector
- Typical Motorola processors use a single area of memory to store the vectors => known as the vector table

- We can divide the interrupt sources into four groups:
  - interrupts from on-chip resources
  - external interrupts
  - software interrupts
  - and reset exceptions
CPU Interrupt Processing

- The interrupt process consists of hardware that detects an event and signals the CPU.
- The CPU changes the program flow to the Interrupt Service Routine.
- The process in the side diagram is called exception processing because the CPU will execute the next instruction except when one of the interrupts is pending.
- An interrupt is said to be pending when it is enabled, its event has been detected, and it has not been serviced.

Pending Interrupt Detection

- Most interrupts are detected by the on-chip resource.
- Only IRQ and XIRQ are detected directly by the CPU.
- The CPU checks the interrupt signals from all sources after it had finished executing each instruction.
- The interrupt signals from on-chip resources are implemented either as event flags that must be cleared explicitly or as automatic event flags.
- IRQ and XIRQ are active-low signals, so these sources are serviced as long as the interrupt signal is low.
Saving Context

- Once a pending interrupt is detected, CPU11, 12 saves the context by pushing all the CPU registers onto the stack.
- All registers must be saved because the interrupt is asynchronous to the program flow and we cannot predict when it will occur.
- At the end of ISR the RTI instruction restores the context of the interrupted program.

Set Interrupt Masks

- After context save, CPU sets the appropriate masks:
  - if the interrupt source is one controlled by the I mask, then I will be set automatically.
  - if the interrupt source is the XIRQ, then both X and I masks are set.
- This is done to prevent nested interrupts before the system is ready.
Vectors

- Next step is to get the interrupt service routine
- An interrupt vector is two fixed memory locations that contain the address of the ISR for a given interrupt
- The CPU 11 interrupt vectors are stored at addresses shown in slide 22
- So, for example, to load the interrupt vector for a service routine you could do:

```
ORG $vector
FDB ISRname ; form double byte
```

Multiple Interrupts and Priority

- When a system has more than one interrupt source enabled, it must be able to handle the condition when more than one interrupt is pending at the same time
- Since the CPU can only handle a single interrupt at a time, a priority is assigned to each source
- The CPU then services the highest priority pending interrupt first
- If the ISR does not reset the I bit, only nonmaskable interrupts or resets can interrupt this service routine
- The ISR could clear I bit to allow itself to be interrupted
- Execution of the RTI restores the original condition of the I bit then the lower-priority interrupt can be serviced
Software Interrupts

- A software interrupt is an instruction that initiates the interrupt process
- In the CPU11 and CPU12 the software interrupt instruction is SWI
- Software interrupts should be used exclusively for debugging tools such as software breakpoints in debug monitors and emulators

- SWI is a single byte opcode instruction
- When a breakpoint is reached in a debug monitor the monitor program substitutes the instruction at that address with an SWI
- When the monitor is finished the SWI opcode is replaced with the original opcode
- Another use for SWI is as a subroutine that preserves the registers automatically

**SWI**

Operation:
- $PC \downarrow (PC) - \$0001
- $(PCL), SP \downarrow (SP) - \$0001
- $(PCH), SP \downarrow (SP) - \$0001
- $(LY), SP \downarrow (SP) - \$0001
- $(YH), SP \downarrow (SP) - \$0001
- $(XL), SP \downarrow (SP) - \$0001
- $(XH), SP \downarrow (SP) - \$0001
- $(ACCA), SP \downarrow (SP) - \$0001
- $(ACCB), SP \downarrow (SP) - \$0001
- $(CCR), SP \downarrow (SP) - \$0001

I $\downarrow 1$, PC $\downarrow$ (SWI vector)

**Description:** The program counter is incremented by one. The program counter, index registers Y and X, and accumulators A and B are pushed onto the stack. The CCR is then pushed onto the stack. The stack pointer is decremented by one after each byte of data is stored on the stack. The I bit in the CCR is then set. The program counter is loaded with the address stored at the SWI vector, and instruction execution resumes at this location. This instruction is not maskable by the I bit.
Software Interrupts

- Another instructions that are used in the interrupt process are WRI and STOP
- These instructions are useful when the program can be in a wait mode while waiting for an interrupt
  - in the wait mode the 68HC11 can reduce its power consumption
**WAI** instruction

* When WAI is executed, it:
  1. Stacks the registers
  2. Waits for an unmasked interrupt

**Note:** This reduces power consumption although the clock is still ticking.

- Only an unmasked interrupt will wake up the controller.
- Some of the I/O subsystems in the controller may still be active and cause the interrupt that terminates the WAI.

---

**WAI**

**Wait for Interrupt**

**Operation:**

- \( PC \downarrow (PC) \) \(-\) $0001
- \( (PCL), SP \downarrow (SP) \) \(-\) $0001
- \( (PCH), SP \downarrow (SP) \) \(-\) $0001
- \( (IYL), SP \downarrow (SP) \) \(-\) $0001
- \( (IYH), SP \downarrow (SP) \) \(-\) $0001
- \( (IXL), SP \downarrow (SP) \) \(-\) $0001
- \( (IXH), SP \downarrow (SP) \) \(-\) $0001
- \( (ACCA), SP \downarrow (SP) \) \(-\) $0001
- \( (ACCB), SP \downarrow (SP) \) \(-\) $0001
- \( (CCR), SP \downarrow (SP) \) \(-\) $0001

**Description:** The program counter is incremented by one. The program counter, index registers Y and X, and accumulators A and B are pushed onto the stack. The CCR is then pushed onto the stack. The stack pointer is decremented by one after each byte of data is stored on the stack.

The MPU then enters a wait state for an integer number of MPU E-clock cycles. While in the wait state, the address/data bus repeatedly runs read bus cycles to the address where the CCR contents were stacked. The MPU leaves the wait state when it senses any interrupt that has not been masked.

Upon leaving the wait state, the MPU sets the I bit in the CCR, fetches the vector (address) corresponding to the interrupt sensed, and instruction execution is resumed at this location.
PIOC configured as shown in Figure 9.13
and all other required initialization has been done.

```
ORG $100

LDY #PTR ;initialize data pointer
CLI ;enable interrupts

REPEAT
  WAI ;wait for falling STRA
  LDAA 0,Y ;send out data when it occurs
  STAA PORTB,X
  INY ;note, MCU also pulses STRB low for 2 E-cycles
  BRA REPEAT

RIRQ
  LDAA PIOC,X ;these two instructions clear STAF
  LDAA PORTCL,X
  RTI

PTR EQU $180
```

This is the interrupt handler routine for parallel I/O
Note it has the same vector ($FF2,F3) as IRQ

```
LDAA PIOC,X ;these two instructions clear STAF
LDAA PORTCL,X
RTI
```

**STOP**

**Stop Processing**

**STOP**

**Description:** If the S bit in the CCR is set, then the STOP instruction is disabled and operates like the NOP instruction. If the S bit in the CCR is clear, the STOP instruction causes all system clocks to halt, and the system is placed in a minimum-power standby mode. All CPU registers remain unchanged. I/O pins also remain unaffected.

Recovery from STOP may be accomplished by RESET, XIRQ, or an unmasked IIRQ. When recovering from STOP with XIRQ, if the X bit in the CCR is clear, execution will resume with the stacking operations for the XIRQ interrupt. If the X bit in the CCR is set, masking XIRQ interrupts, execution will resume with the opcode fetch for the instruction which follows the STOP instruction (continue).
The effect of executing STOP depends on the state of the CCR (Condition Code Register) bit.

1. If the CCR bit is set, the CPU treats STOP like a NOP instruction and continues on to execute any instruction following STOP.

2. If the CCR bit is reset, all internal clocks in the CPU halt, thus halting execution.
   - This puts the CPU in a state of lowest power consumption.
   - To wake up the controller, use RES 0, STOP.
   - If $x=1$, processing will continue with the instruction following STOP.
   - If $x=0$, the processor will perform the XIRQ operation.

![Diagram showing the effect of STOP instruction on program flow]
External Interrupts

- There are 2 external interrupt pins on the CPU11 and 12
- IRQ – is an active-low, maskable interrupt signal request
- XIRQ – is an active-low, pseudo-nonmaskable interrupt request
- By default, when you power up the MPU or RESET it, XIRQ is masked, that is:
  - The X bit in CCR is set
- To unmask XIRQ, you use the TAP instruction to clear bit X
- Once the program has cleared it, executing another TAP will not set X again
- The only way to set X is to RESET the 68HX11

```
TPA ;CCR -> A
ANDA #$BF; reset bit 6 (X)
TAP ;A -> CCR
```
68HC11 Interrupts

Many of the internal I/O devices can generate interrupt requests based on external events (e.g., STRA, input capture, SCI, SPI).

Most of the interrupt requests will temporarily set the I bit in CC during the interrupt program to prevent other interrupts.

- Priority fixed to change use
- A STRA interrupt
- Three input capture interrupts
- Fire output compare interrupts
- Three timer interrupts (timer overflow, RTI pulse accumulator)
- Two serial port interrupts (SCI/SP1)
Changing Priority

- HPRIO – highest priority interrupt register
  - may be read at any time but may only be written under special circumstances
- Interrupts obey a fixed hardware-priority circuit to resolve simultaneous requests
- However, an I-bit-related interrupt source may be elevated to the highest I-bit priority position in the resolution circuit
- The first six interrupts are not masked by the I bit in the CCR and have a fixed priority:
  - reset, clock monitor fail, COP fail, illegal opcode and XIRQ
- HPRIO may only be written while the I-bit related interrupts are inhibited (I=1)

TABLE 2-2

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>Default Priority</th>
<th>HPRIO Promote to #1 priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ</td>
<td>1</td>
<td>0 1 1 (0) 0 (1)</td>
</tr>
<tr>
<td>RTI</td>
<td>2</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>Input Capture 1</td>
<td>3</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>Input Capture 2</td>
<td>4</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>Input Capture 3</td>
<td>5</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>Output Compare 1</td>
<td>6</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>Output Compare 2</td>
<td>7</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>Output Compare 3</td>
<td>8</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>Output Compare 4</td>
<td>9</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>Input Capture 4/Output Compare 5</td>
<td>10</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>Timer Overflow</td>
<td>11</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>Pulse Accumulator Overflow</td>
<td>12</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>Pulse Accumulator Input Edge</td>
<td>13</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>SPI Serial Transfer Complete</td>
<td>14</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>SCI Serial System</td>
<td>15</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>
**External Interrupt Design Approach**

We identify the busy-to-done state transition to cause interrupt.

1. Decide what status signals from the I/O device signify when the input device has new data and needs software to read them.
2. For an output device, we look for a signal that specifies when it is finished.

Next, we connect I/O status signals to a microcontroller input that can generate interrupts.

*There are four major components to interrupting a software system:
- **Ritual** (initialized)
- Main program
- ISR
- Interrupt vectors

Ritual

1. Is a program executed during start-up that initializes hardware & software.
2. It is executed once on start-up.
3. It is a good idea to disable interrupts at the beginning of the ritual so that interrupts are not requested before entire initialization is allowed to finish.
4. **sei** — disable
5. **set I/O port direction**
6. **set I/O port interrupt control reg**
7. **ei** — enable
Main program → ritual executed, generally performs tasks that are not time critical.

Interrupt handler:
- For polled interrupt, determine the source polling order will determine priority
- ISR must ack interrupt i.e clear flag
- Info is exchanged with main program via global memory
- Executes RTI to return control back

Interrupt vectors:
- General purpose → RAM
- Embedded systems → PROM

Interrupt vectors
- All resets and interrupts use vectors:
  - **A vector** indicates the start address of reset or interrupt routine.
  - **A vector address** is a 2-byte memory location that stores a vector.
  - **Vector table** is a single area of memory to store vectors.

Check App B Table B.2
How to Establish the Vector

1. Single chip standalone system
   - We can use assembly language to set the vectors
     - ORG $FF00
     - FDB RTIHAN - pointer to RTI

2. With 8811 EVB
   - Interrupt source 8811 vector Buffalo Jump Token
     - SCI $FF6 $00C4 - $00CE
     - RTI $FF0 $0088 - $0080
   - In each case a 3-byte value is stored in the jump location
**IRQ Interrupt Request**

1. IRQ is maskable by setting bit I in CCR
2. You can mask IRQ using SETI
3. You can program pin IRQ to be edge sensitive or level sensitive

Option Register #1039

<table>
<thead>
<tr>
<th>norest</th>
<th>IRQE</th>
<th>ELE</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>IRQE</td>
<td>ELE</td>
</tr>
<tr>
<td>set to 1</td>
<td>edge sensitive</td>
<td></td>
</tr>
</tbody>
</table>

*Note* Some bits in option Reg are time protected i.e. you can change the bit only within the first 4 or clock cycles after a reset. So only Reset Routine can make IRQ edge sensitive.

---

**Interrupt Polling Using Linked Lists**

The process of polling involves (software) at the beginning of ISR that checks one by one the list of possible devices that might have caused interrupt. Sometimes we might poll even if our interrupting device has its own dedicated vector - Robustness.

Interrupt list data structure used to implement polling

Example of a system with a potential interrupt source.

See program 4.38 page 445 for detail implementation:

- read status reg (eliminate irrelevant bits)
- determine if device is requesting
- execute the handler
6811 Assembly Structure for Interrupt Polling using Linked Lists

```
start  fdb 11STAF    place to start polling
Sreg  equ 0        Index to Status Register
Amask equ 2        and mask
Cmask equ 3        compare mask
DevHan equ 4       device handler
NextPt equ 6       next pointer
num fcb 3         number of devices
11STAF fdb $1002   address of PIOC
   fcb $ff        look at all the bits in PIOC
   fcb $C0        expect exact match with $C0
   fdb STAFHan    device handler
   fdb 11CA1      pointer to next device to poll

11CA1 fdb $2011   address of 6821 Port A Control/Status
   fcb $87        look at bits 7,2,1,0
   fcb $85        expect bits 7,2,1,0 to be 1,1,0,1
   fdb CA1Han     device handler for CA1
   fdb 11CB2      pointer to next device to poll
11CB2 fdb $2013   address of 6821 Port B Control/Status
   fcb $7C        look at bits 6,5,4,3,2
   fcb $4C        expect bits 6,5,4,3,2 to be 1,0,0,1,1
   fdb CB2Han     device handler for CB2
   fdb 0          no more

IrqHan idx start    Reg X points to linked list place to start
   ldab num      number of possible devices
next idx Sreg,x     Reg Y points to status reg
   ldae,y       read status
   anda Amask,x  clear bits that are indeterminate
   cmp Cmask,x    expected value if this device active
      bne Notyet skip if this device not requesting
   ldy DevHan,x   Reg Y points to device handler
      jsr ,y       call device handler, will return here
Notyet idx NextPt,x Reg X points to next entry
   decb          device counter
      bne next    check next device
      rti
```

ENGG4640/3640; Fall 2004; Prepared by Radu Muresan
Periodic Polling Plus Real Time Interrupts

A RTI is one that is requested on a fixed time basis (data acquisition & control system). Because software servicing must be performed at accurate time intervals, it is important to establish an accurate sampling rate.

Another application of RTIs is "periodic polling." If no device needs service, then interrupt returns usually used if I/O hardware cannot generate interrupts directly.

The RTI feature can be used to generate interrupts at a fixed rate.

→ Two bits (RTTI & RTTB) in PACTR determine interrupt rate.
### 6811 Registers Used to Configure the RTI

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDRA7</td>
<td>PAMOD</td>
</tr>
<tr>
<td>6811</td>
<td>RTI</td>
</tr>
<tr>
<td>$1026</td>
<td>PAEN</td>
</tr>
<tr>
<td></td>
<td>PEDGE</td>
</tr>
<tr>
<td></td>
<td>PACTR</td>
</tr>
<tr>
<td></td>
<td>default =</td>
</tr>
<tr>
<td></td>
<td>000000000</td>
</tr>
</tbody>
</table>

Pulse Accumulator Control Reg

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOI</td>
<td>PAOVI</td>
</tr>
<tr>
<td>$1024</td>
<td>RTI</td>
</tr>
<tr>
<td></td>
<td>PAII</td>
</tr>
<tr>
<td></td>
<td>PRO</td>
</tr>
<tr>
<td></td>
<td>TMSK2</td>
</tr>
<tr>
<td></td>
<td>default =</td>
</tr>
<tr>
<td></td>
<td>000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOF</td>
<td>PAOVF</td>
</tr>
<tr>
<td>$1025</td>
<td>RTIF</td>
</tr>
<tr>
<td></td>
<td>PAIF</td>
</tr>
<tr>
<td></td>
<td>TFLG2</td>
</tr>
<tr>
<td></td>
<td>default =</td>
</tr>
<tr>
<td></td>
<td>000000000</td>
</tr>
</tbody>
</table>
RITUAL

set disable interrupts during RITUAL
ldaa #3
sta $1026
ldaa #$40
sta $1024
cli
rts

RTIHAN

ldaa $1025 Polling for zeroes and ones expect=X1XX0000
anda #$4F ignore TOF, PAOVF, and PAIF
cmpa #$40 RTIF should equal 1
beq OK
swi Error

OK

ldaa #$40 RTIF is cleared by writing to TFLG2
sta $1025 with bit 6 set
* service occurs every 32.768ms or about 30.517Hz
rti

68HC11 Assembly Language Implementation of a Periodic Interrupt Using RTI
ENGG4640/3640; Fall 2004; Prepared by Radu Muresan

II. Dedicated edge-triggered requests

* Here each device has its own interrupt request line that is usually connected to a status signal in the I/O device.
* With vectored interrupts, these individual requests have a unique interrupt vector.
* A separate ISR for each device.
* The ISR does not need to determine which device requested the interrupt.

Interrupt service Routine?

ISR is the software module that is executed when the hardware requests an interrupt.

From previous discussion:
1. There may be one large ISR that handles all requests (polled interrupts).
2. There can be many ISRs, specific for each potential source of interrupt (vectored interrupts).