Interrupts and Stack Intro

EE3376

Intro to Stack

- Last In – First Out LIFO structure with hardware assist
  - Dedicated SP pointing to TOS
  - One push instruction and one emulated pop instruction
  - All other addressing modes work as well
- Must be initialized in ASM but handled under the covers for C
  - The first address loaded in SP is never used.
- Used
  - for local (temporary) variables
  - for saving snapshot of registers to use temporarily
  - for saving the PC and SR during interrupts
- Generally just a great way to save things for short period
  - Can easily de-allocate memory for other use

Intro to Stack - Push

3.4.6.35 PUSH

PUSH[.W]	Push word onto stack
PUSH.B	Push byte onto stack
Syntax	`PUSH src or PUSH.W src` `PUSH.B src`
Operation	SP - 2 → SP src → @SP
Description	The stack pointer is decremented by two, then the source operand is moved to the RAM word addressed by the stack pointer (TOS).
Status Bits	Status bits are not affected.
Mode Bits	OSCOFF, CPUOFF, and GIE are not affected.

Example	The contents of the status register and R8 are saved on the stack.
	`PUSH SR ; save status register`
	`PUSH R8 ; save R8`

Example	The contents of the peripheral TCDAT is saved on the stack.
	`PUSH.B &TCDAT ; save data from 8-bit peripheral module,`
	`; address TCDAT, onto stack`

NOTE: System Stack Pointer

The System stack pointer (SP) is always decremented by two, independent of the byte suffix.

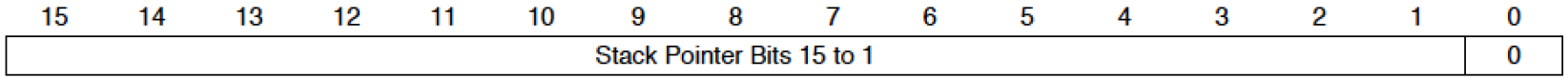
Intro to Stack - Example

3.2.2 Stack Pointer (SP)

The stack pointer (SP/R1) is used by the CPU to store the return addresses of subroutine calls and interrupts. It uses a predecrement, postincrement scheme. In addition, the SP can be used by software with all instructions and addressing modes. Figure 3-3 shows the SP. The SP is initialized into RAM by the user, and is aligned to even addresses.

Figure 3-4 shows stack usage.

Figure 3-3. Stack Counter




```
MOV   2(SP),R6    ; Item I2 -> R6
MOV   R7,0(SP)    ; Overwrite TOS with R7
PUSH  #0123h      ; Put 0123h onto TOS
POP   R8          ; R8 = 0123h
```

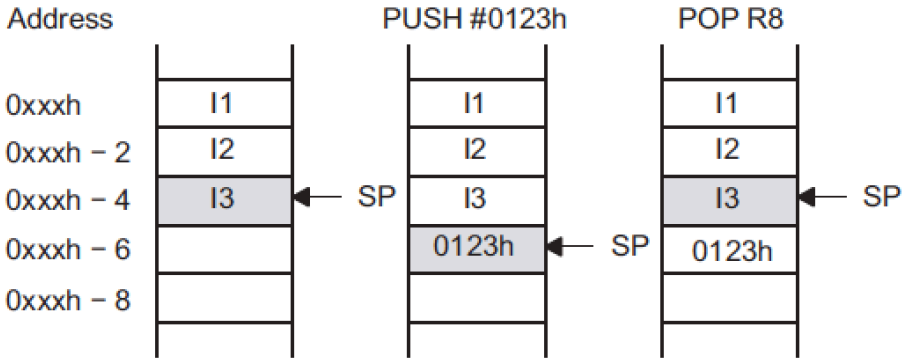


Figure 3-4. Stack Usage

Intro to Stack - Pop

***POP[.W]**	Pop word from stack to destination
***POP.B**	Pop byte from stack to destination
Syntax	`POP dst` `POP.B dst`
Operation	@SP → temp SP + 2 → SP temp → dst
Emulation	`MOV @SP+,dst or MOV.W @SP+,dst` `MOV.B @SP+,dst`
Description	The stack location pointed to by the stack pointer (TOS) is moved to the destination. The stack pointer is incremented by two afterwards.
Status Bits	Status bits are not affected.

Example	The contents of R7 and the status register are restored from the stack.
	`POP R7 ; Restore R7`
	`POP SR ; Restore status register`

Example	The contents of RAM byte LEO is restored from the stack. `POP.B LEO ; The low byte of the stack is moved to LEO.`

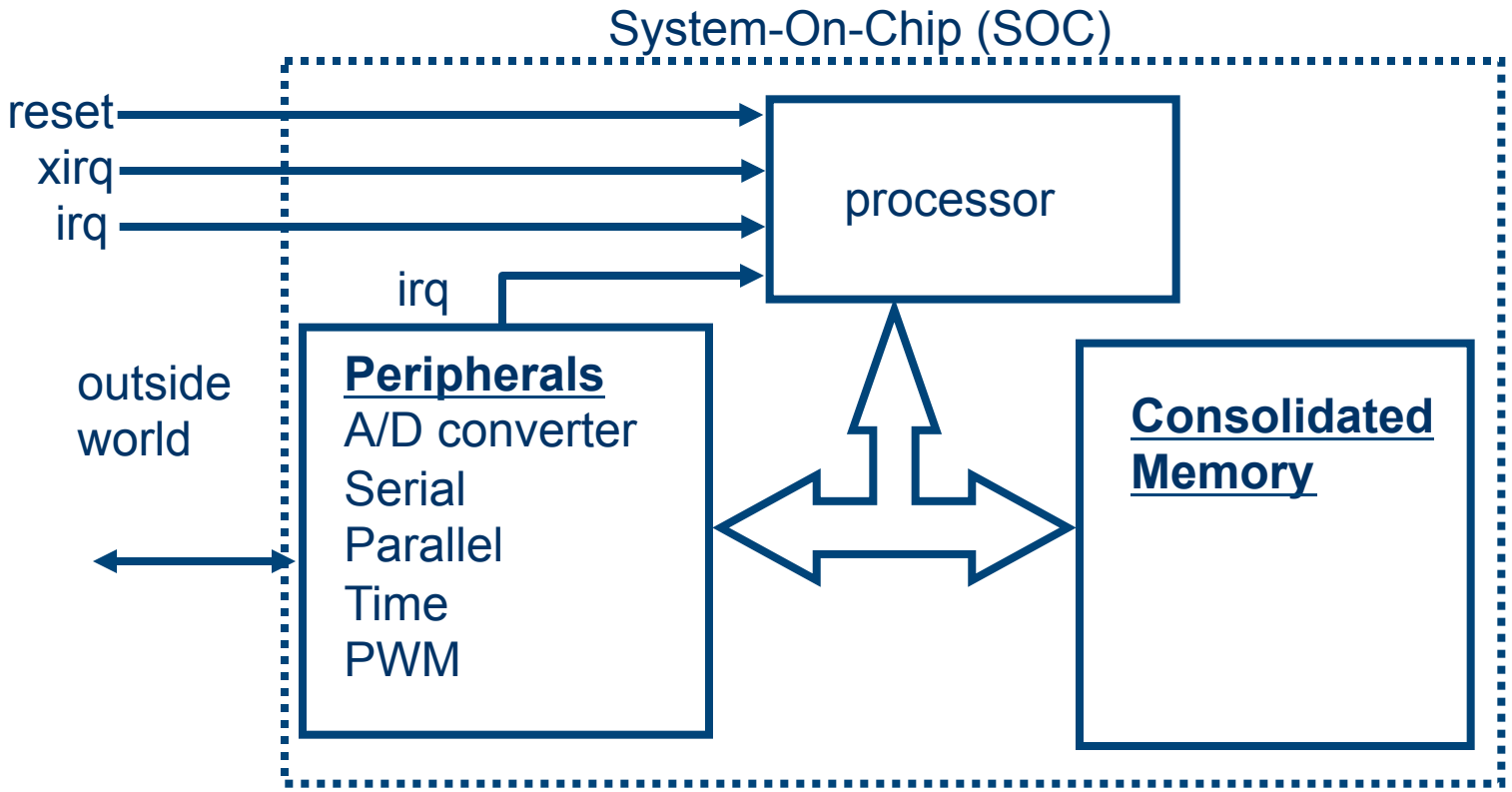
Example of Stack usage

```
; Subroutine to give delay of R12*0.1s
; Parameter is passed in R12 and destroyed
; R4 used for loop counter, stacked and restored
;-----------------------------------------------------------------------
DelayTenths:
    push.w  R4                      ; Stack R4: will be overwritten
    jmp     LoopTest                ; Start with test in case R12 = 0
OuterLoop:
    mov.w   #DELAYLOOPS,R4          ; Initialize loop counter
DelayLoop:                          ; [clock cycles in brackets]
    dec.w   R4                      ; Decrement loop counter  [1]
    jnz     DelayLoop               ; Repeat loop if not zero [2]
    dec.w   R12                     ; Decrement number of 0.1s delays
LoopTest:
    cmp.w   #0,R12                  ; Finished number of 0.1s delays?
    jnz     OuterLoop               ; No: go around delay loop again
    pop.w   R4                      ; Yes: restore R4 before returning
    ret                             ; Return to caller
```

Example of local variables

```
; Subroutine to give delay of R12*0.1s
; Parameter is passed in R12 and destroyed
; Space for two loop counters is created on stack, after which
;   0(SP) is innermost (little) loop, 2(SP) is big loop counter
;-------------------------------------------------------------------
; Iterations of delay loop for about 0.1s (6 cycles/iteration):
BIGLOOPS     EQU     130
LITTLELOOPS  EQU     100
;-------------------------------------------------------------------
DelayTenths:
     sub.w    #4,SP                ; Allocate 2 words (4 bytes) on stack
     jmp      LoopTest             ; Start with test in case R12 = 0
OuterLoop:
     mov.w    #BIGLOOPS,2(SP)      ; Initialize big loop counter
BigLoop:
     mov.w    #LITTLELOOPS,0(SP)   ; Initialize little loop counter
LittleLoop:                        ; [clock cycles in brackets]
     dec.w    0(SP)                ; Decrement little loop counter [4]
     jnz      LittleLoop           ; Repeat loop if not zero       [2]
     dec.w    2(SP)                ; Decrement big loop counter    [4]
     jnz      BigLoop              ; Repeat loop if not zero       [2]
     dec.w    R12                  ; Decrement number of 0.1s delays
LoopTest:
     cmp.w    #0,R12               ; Finished number of 0.1s delays?
     jnz      OuterLoop            ; No: go around delay loop again
     add.w    #4,SP                ; Yes: finished, release space on stack
     ret                           ; Return to caller
```

Interrupts


System-On-Chip (SOC)
reset
xirq
irq
processor
irq
outside
world
Peripherals
A/D converter
Serial
Parallel
Time
PWM
Consolidated
Memory

Interrupt Overview

- Interrupts (a.k.a. exceptions) are requests to the CPU
- Typical example is hardware requesting service
- Avoids wasteful polling – CPU is unproductive during polling.
- Some sources of interrupts are intentional
  - from internal sources (i.e. Timer or ADC sample completion)
  - from external sources (i.e. NMI pin)
  - from reset assertion (highest priority)
- Some sources of interrupts are not necessarily intentional
  - from instructions (i.e. illegal instruction - trap)
  - from a variety of illegal hardware conditions (bad clock or V_{dd})
  - Watch Dog Timeout
- When they will occur is unknown to the CPU – asynchronous
- Different priority levels for interrupts
- Most interrupts are mask-able and need to be armed.

Interrupt Service Routine (Handler)

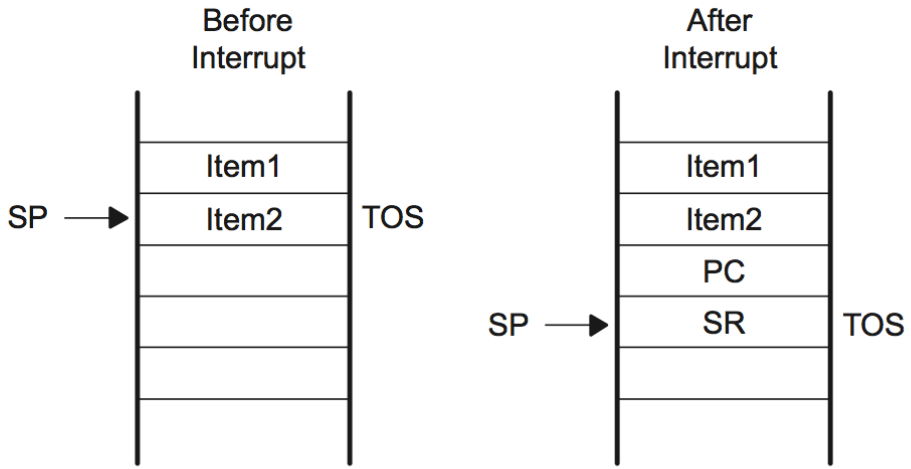
- Interrupt cause the ISR to be executed when …
  - the interrupt is armed (interrupt specific arm bit is set – P1IE)
  - interrupts in general are enabled (GIE is set in SR)
  - and the interrupt signal is asserted (either internally or externally)

- For each type of interrupt, there is an entry in the interrupt vector

- An Interrupt Service Routine (ISR) is like a special subroutine.
  - current instruction in main program is completed
  - typically need to clear the interrupt source with an acknowledge
  - execute an rti instruction (not an rts for subroutine)
  - all registers are restored after execution
  - main program resumes as if nothing happened

Interrupt Activities

2. The PC, which points to the next instruction, is pushed onto the stack.
3. The SR is pushed onto the stack.
4. The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.
5. The interrupt request flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.
6. The SR is cleared. This terminates any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.
7. The content of the interrupt vector is loaded into the PC: the program continues with the interrupt service routine at that address.



RETI instruction

2.2.3.2 Return From Interrupt

The interrupt handling routine terminates with the instruction:

RETI (return from an interrupt service routine)

The return from the interrupt takes 5 cycles (CPU) or 3 cycles (CPUx) to execute the following actions and is illustrated in Figure 2-7.

1. The SR with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings used during the interrupt service routine.
2. The PC pops from the stack and begins execution at the point where it was interrupted.

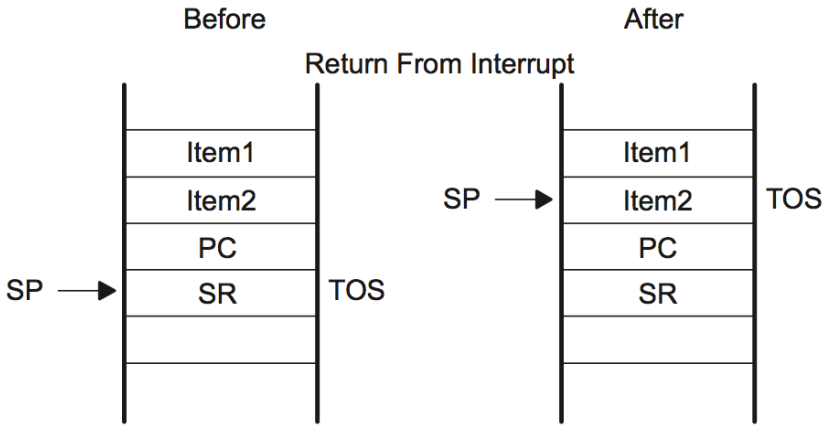


Figure 2-7. Return From Interrupt

Interrupt Vector Table

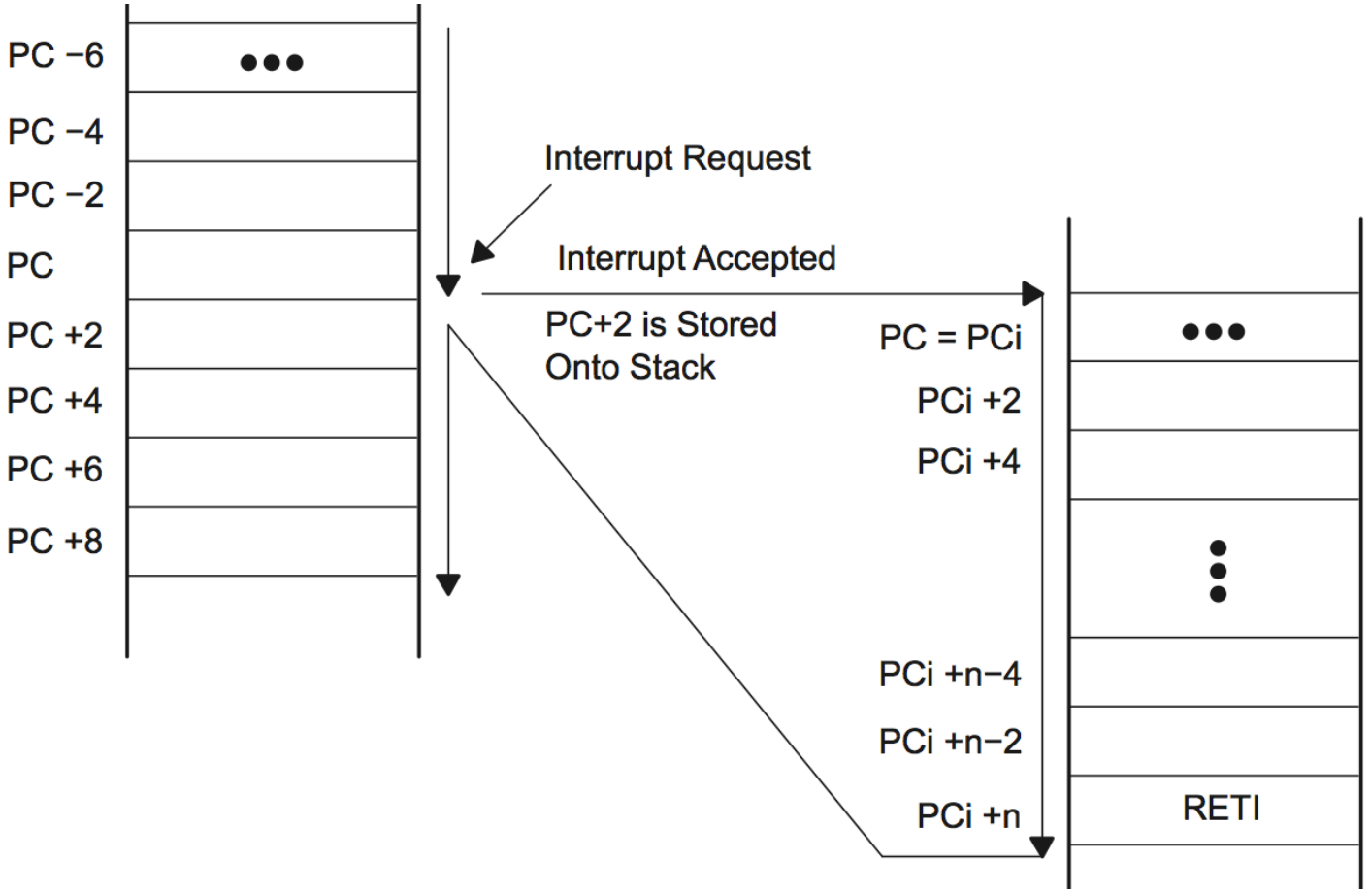
Table 5. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Timer+ Flash key violation PC out-of-range (1)	PORIFG RSTIFG WDTIFG KEYV (2)	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG (2) (3)	(non)-maskable (non)-maskable (non)-maskable	0FFFCh	30
Timer1_A3	TA1CCR0 CCIFG (4)	maskable	0FFFAh	29
Timer1_A3	TA1CCR2 TA1CCR1 CCIFG, TAIFG (2) (4)	maskable	0FFF8h	28
Comparator_A+	CAIFG (4)	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer0_A3	TA0CCR0 CCIFG (4)	maskable	0FFF2h	25
Timer0_A3	TA0CCR2 TA0CCR1 CCIFG, TAIFG (5) (4)	maskable	0FFF0h	24
USCI_A0/USCI_B0 receive USCI_B0 I2C status	UCA0RXIFG, UCB0RXIFG (2) (5)	maskable	0FFEEh	23
USCI_A0/USCI_B0 transmit USCI_B0 I2C receive/transmit	UCA0TXIFG, UCB0TXIFG (2) (6)	maskable	0FFECh	22
ADC10 (MSP430G2x53 only)	ADC10IFG (4)	maskable	0FFEAh	21
			0FFE8h	20
I/O Port P2 (up to eight flags)	P2IFG.0 to P2IFG.7 (2) (4)	maskable	0FFE6h	19
I/O Port P1 (up to eight flags)	P1IFG.0 to P1IFG.7 (2) (4)	maskable	0FFE4h	18

Interrupt Examples

- Servicing data transfers for data communication peripherals
  - UART buffer full and transmit complete
  - UART receive buffer full and transmit complete
  - IIC
- To perform periodic tasks using Timer interrupts
  - Blinking LED (once a second)
  - LCD display update (60 Hz)
- To handle unexpected software or hardware problems
  - missing clock
  - low Vdd
  - bad instruction
- Reset is a fundamental interrupt requesting to start over
  - Unmaskable, highest priority and no reti

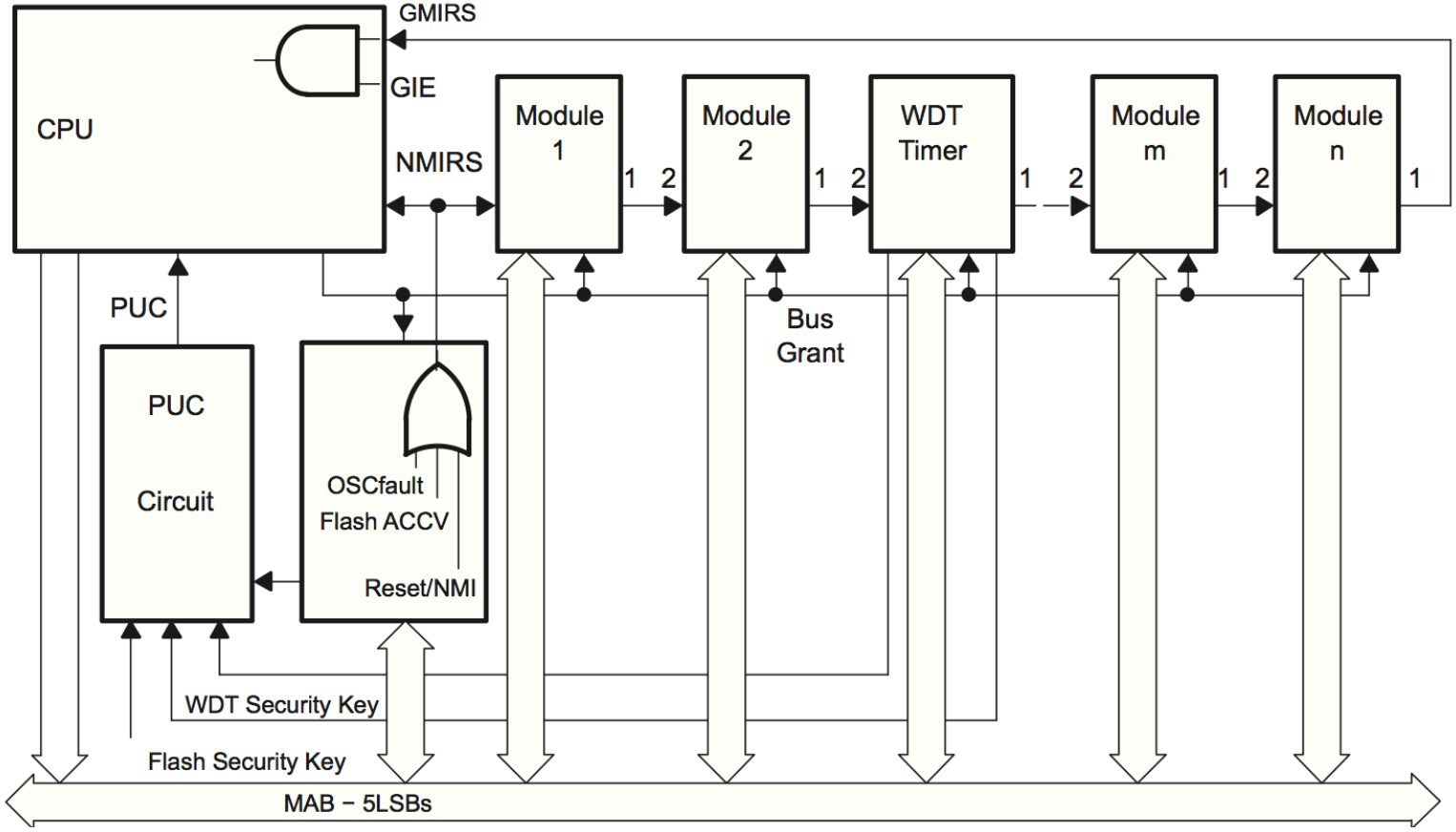
Interrupt flow



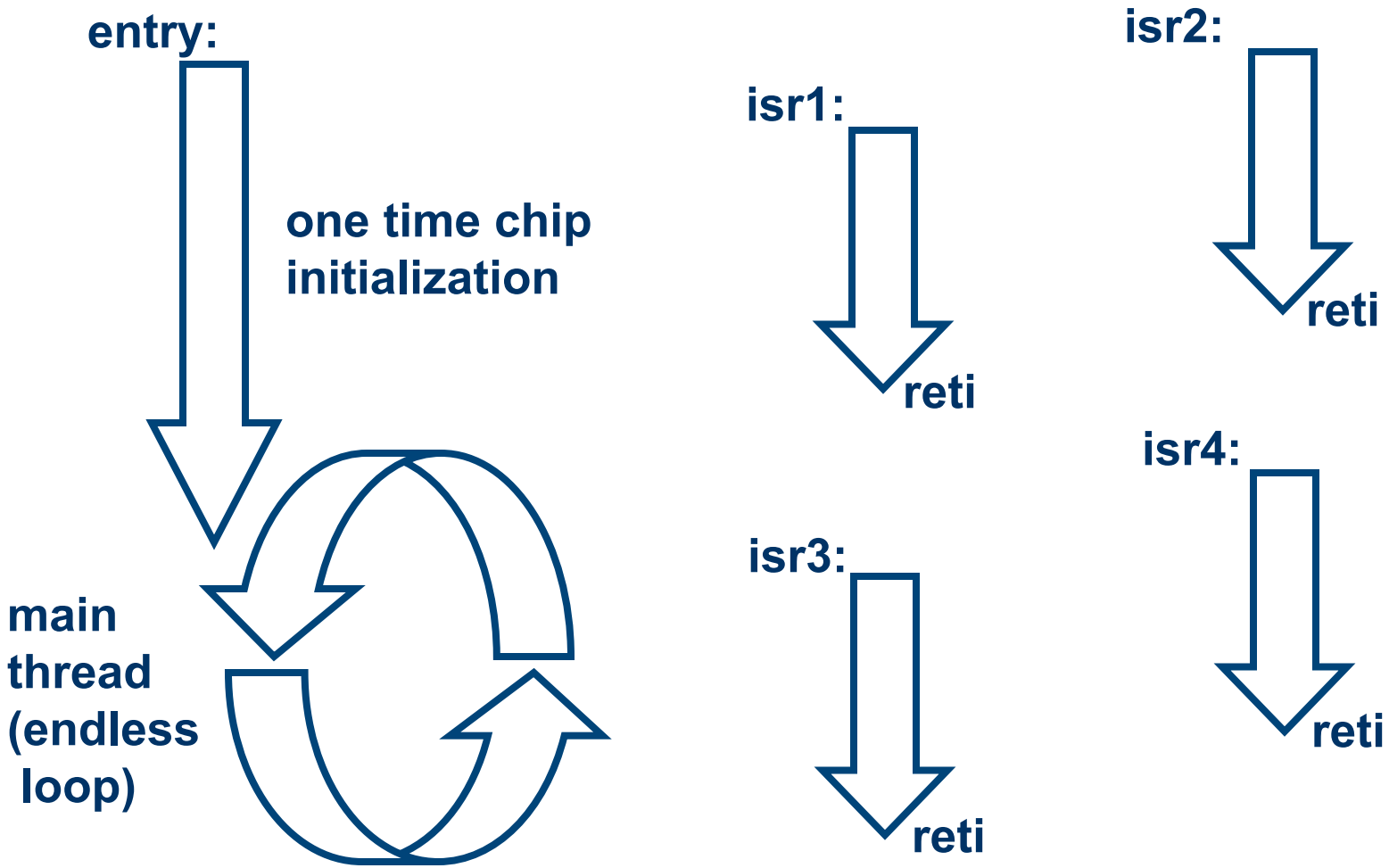
Interrupt Priorities

- For real time embedded systems, some interrupts have to be handled in a specific amount of time (real time constraint)
  - Analog has hard timing requirements
  - Anti-lock brakes in a car need sense of urgency
  - Playing Tetris – not so important – humans are slow
- These interrupts get priority and can interrupt current interrupts.
- Maskable interrupts also have individual bits that should set first
- GIE enables all maskable interrupts
  - This bit should be set last at the end of the configuration
- Individual bits control the (non)-maskable interrupts
  - non is in parenthesis because technically they can be masked
    - Just not affected globally with the GIE bit.

MSP430 Priorities

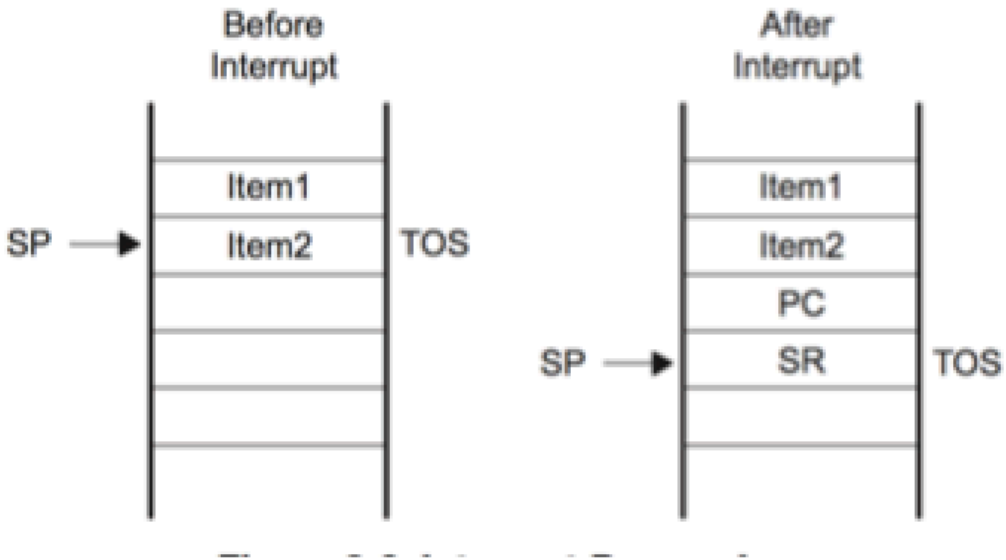


Interrupt Flow



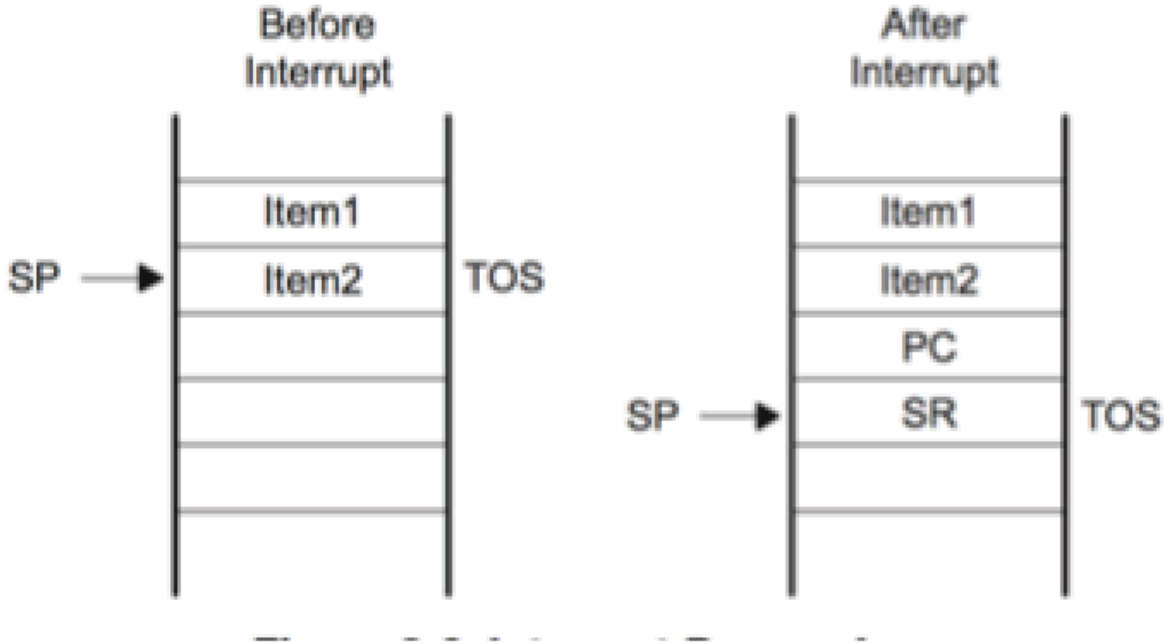
Stack Pushing Prior to Interrupt

- Processors can take two approaches for “saving context”
  - Save everything – safer – all registers stacked - Motorola
  - Save minimum – let user save anything that could change
    - Faster - MSP430 uses the second approach so beware.



Stack Pushing Prior to Interrupt

- PC is pushed on stack
- SR is pushed on stack
- SR is cleared (GIE and low power modes remembered)
- Jump to Interrupt Handler location described in Interrupt Vector
- GIE is turned off for during handler



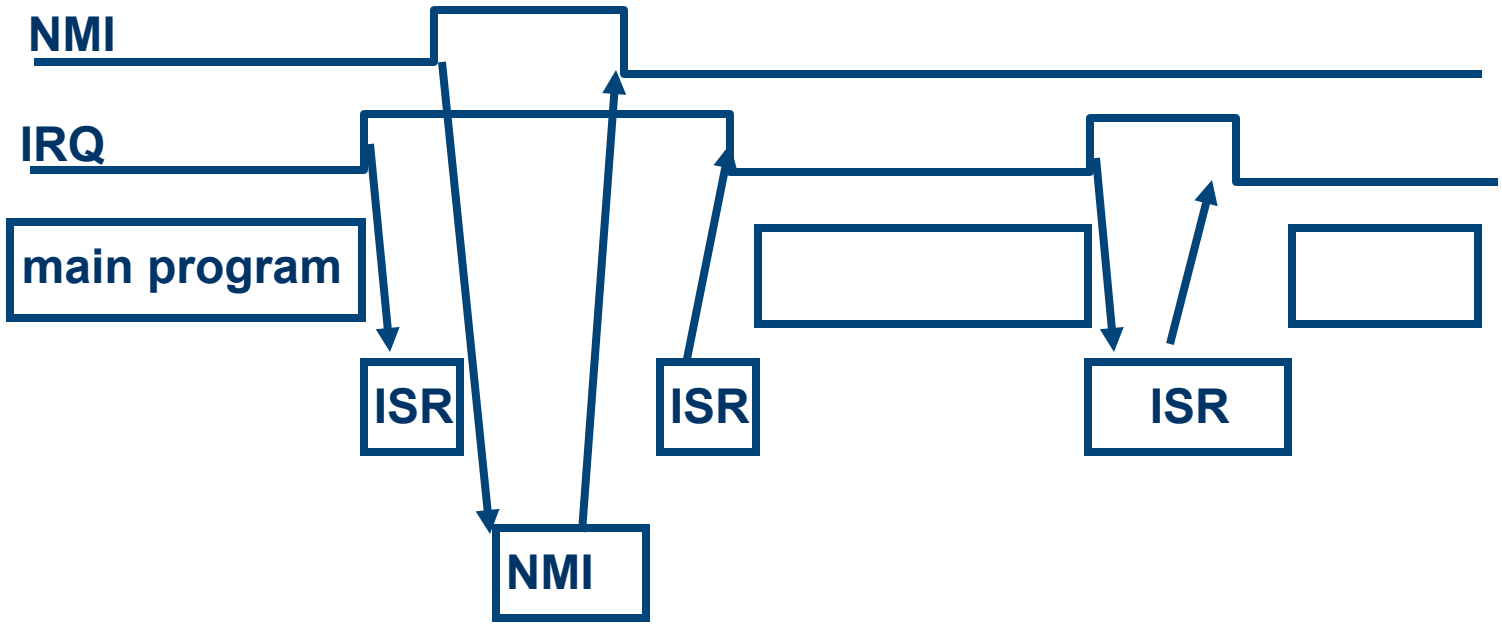
Interrupt Acknowledging

- Must clear the flag of any interrupt during the handler
  - Otherwise, you will repeat the interrupt
- Other processors usually require clearing the flag
- MSP430
  - No requirement for acknowledging for single source IRQs
  - For multi-source IRQs – Timer for instance – you will must clear manually

Interrupt Synchronization

- Different software threads are executing
  - main program is considered foreground thread
  - interrupts are background threads
- Global variables are used to communicate between them
  - Global variables have dedicated space in SRAM
- Different global data structures can be used
  - mail box
  - FIFO
  - linked list
- Local variables are not allowed for passing data.
  - can't use the stack because the threads are asynchronous

Nested Interrupt Flow



Three software threads with only one active at a time.

Interrupts are a hardware mechanism to switch between threads.

To get second maskable interrupt in interrupt, the first handler must enable GIE. NMI will always interrupt a maskable interrupt.

First-In First Out (FIFO) data structure

- FIFOs are useful structures for consumer-producer applications
- consumer-producer apps
  - data flow in one direction
  - order of data is important
  - allows producer to get X bytes ahead of consumer without problem.
- Some finite amount of memory is dedicated to FIFO
- Two pointers are used
  - get pointer – controlled by the consumer (main program)
  - put pointer – controlled by the producer (interrupt handler)
  - get pointer chases put pointer
- Pointers wrap around to the beginning if they extend boundary
- If get pointer = put pointer after a get, FIFO empty
- If get pointer = put pointer after a put, FIFO overflow

First-In First Out (FIFO) data structure