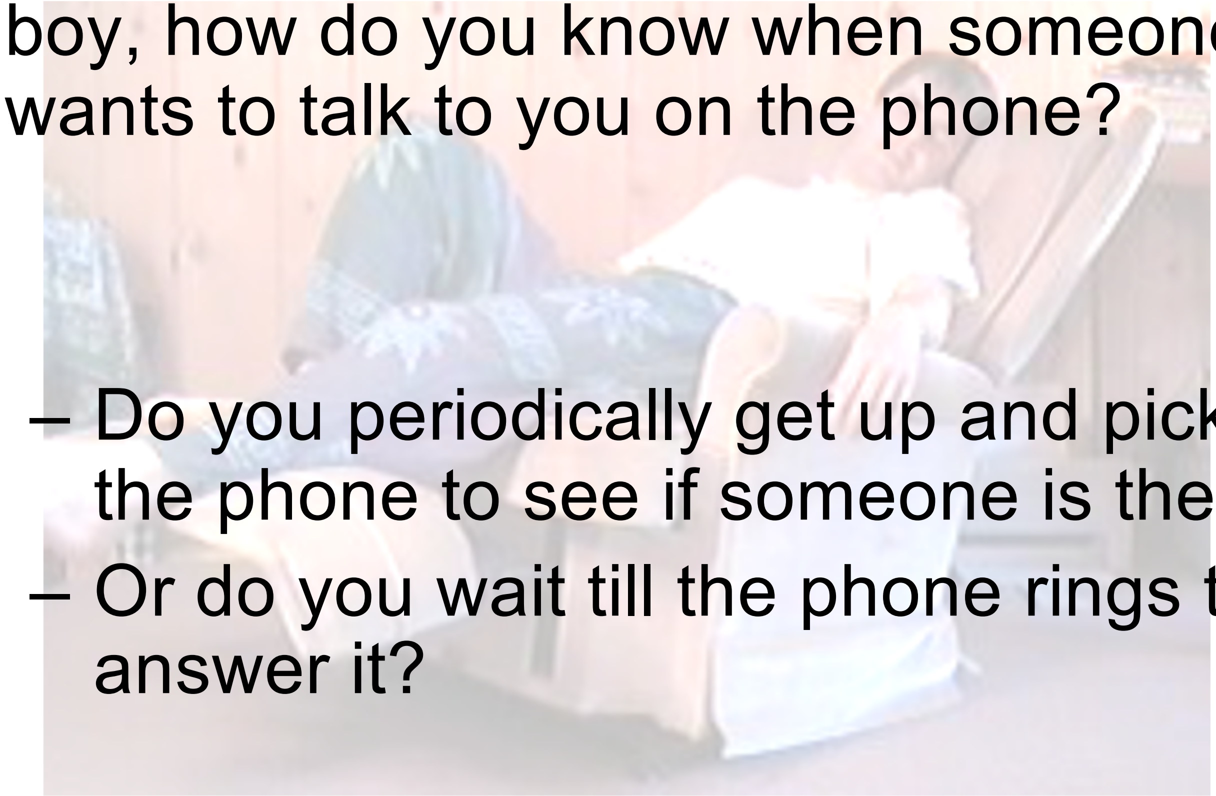
Question??



- When you are at home sitting on your lazy boy, how do you know when someone wants to talk to you on the phone?
  - Do you periodically get up and pick up the phone to see if someone is there?
  - Or do you wait till the phone rings to answer it?

Answer?

- The first scenario shows a person doing what is known as polling.

- The second case illustrates an interrupt-driven person.

Polling

- AKA “busy waiting”; looping program
- Continues checking status register until a particular state exists
- “Are we there yet? Are we there yet? Are we there yet?”
- What happens if something occurs at other devices while the processor is busy waiting??

Interrupt

- Device sends a special signal to CPU when data arrives.
- “Wake me up when we get there.”
- Responds to hardware interrupt signal by interrupting current processing.
- Now CPU can perform tasks before and after interrupt instead of just polling!! Good!

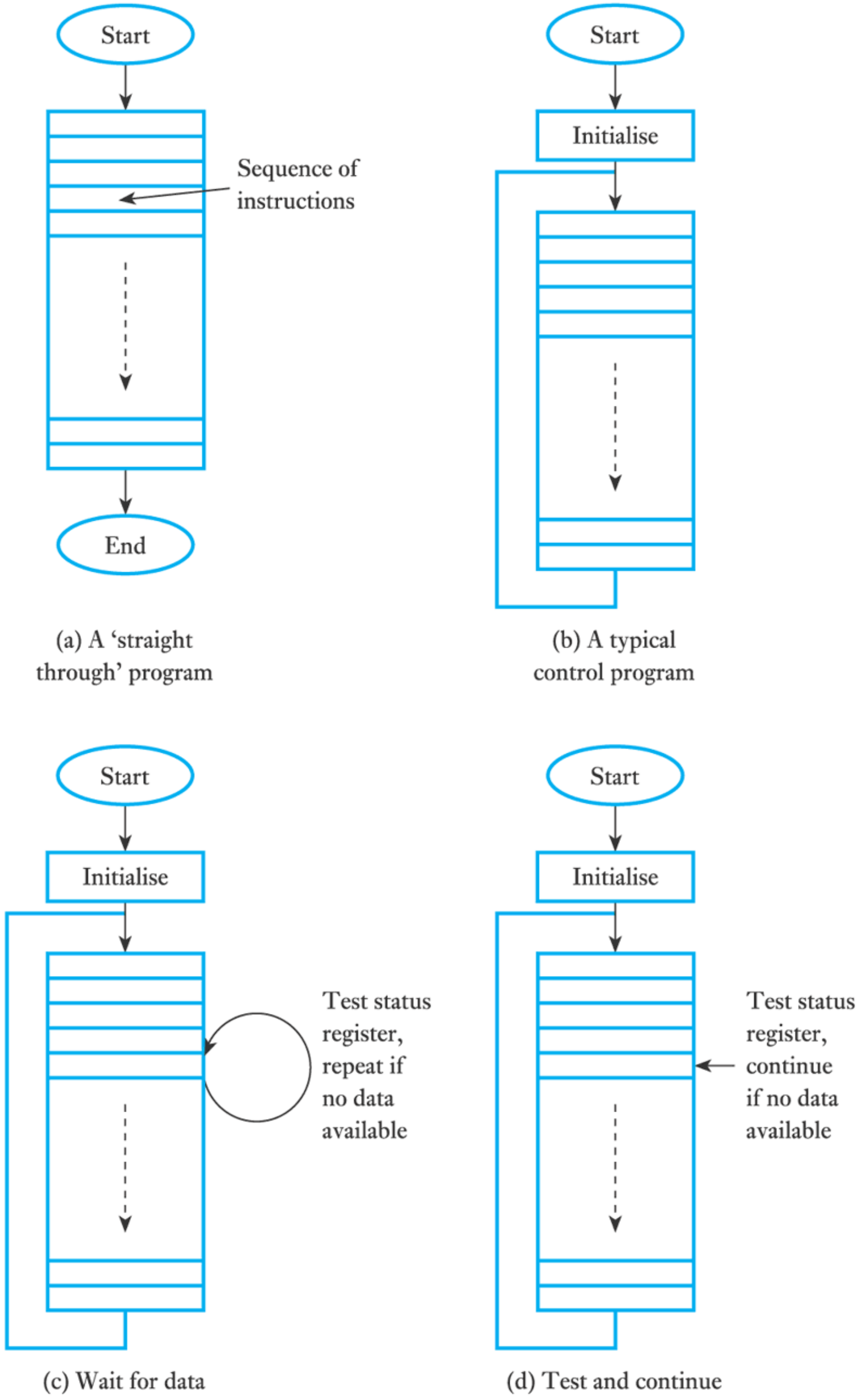
Brief Background

- Univac 1103/1103A (1953-56) – first recognized CPU with interrupts. Current instruction was stored in memory and program counter loaded with a fixed address.

- Other notables:
  - IBM (1954) – first to use interrupt masking
  - NBS DYSEAC (1954) – first with I/O interrupts

Programmed controlled input/output

- (From IS1:) polling of I/O devices:


Start
Sequence of instructions
End
Start
Initialise
Start
Initialise
Test status register, repeat if no data available
Start
Initialise
Test status register, continue if no data available

(a) A 'straight through' program

(b) A typical control program

(c) Wait for data

(d) Test and continue

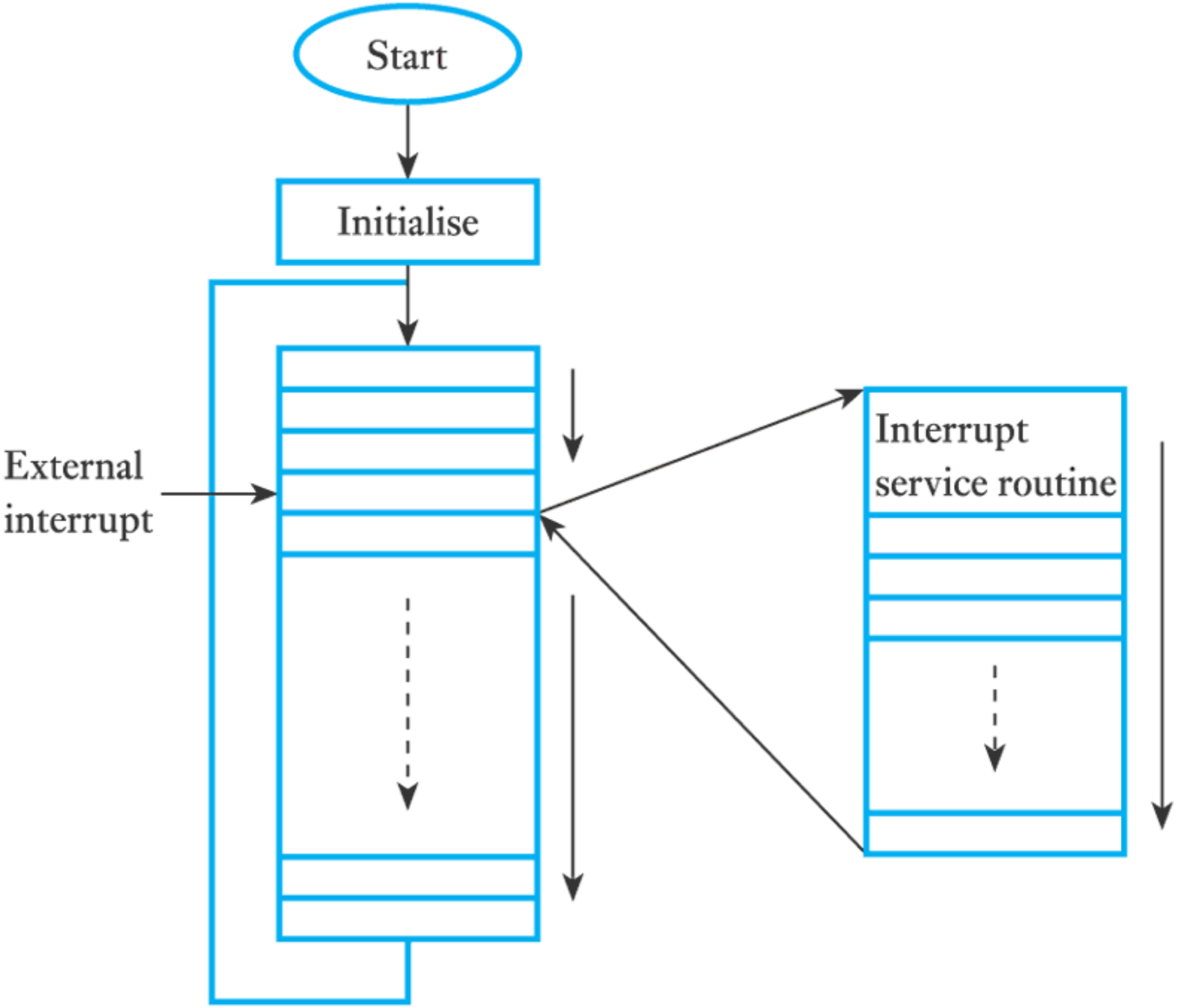
Polling – simple, but inflexible…

- In Lab 3 (“Tärning”) we may have wanted the text on the display “Rolling…” to have an animation as well…
- However, this would have made the program more complex….
- What if we somehow could have parallel execution, (similar to threads in Java)?
- There IS such a concept – Interrupts!
  - *(also DMA, but not described in this course…)*

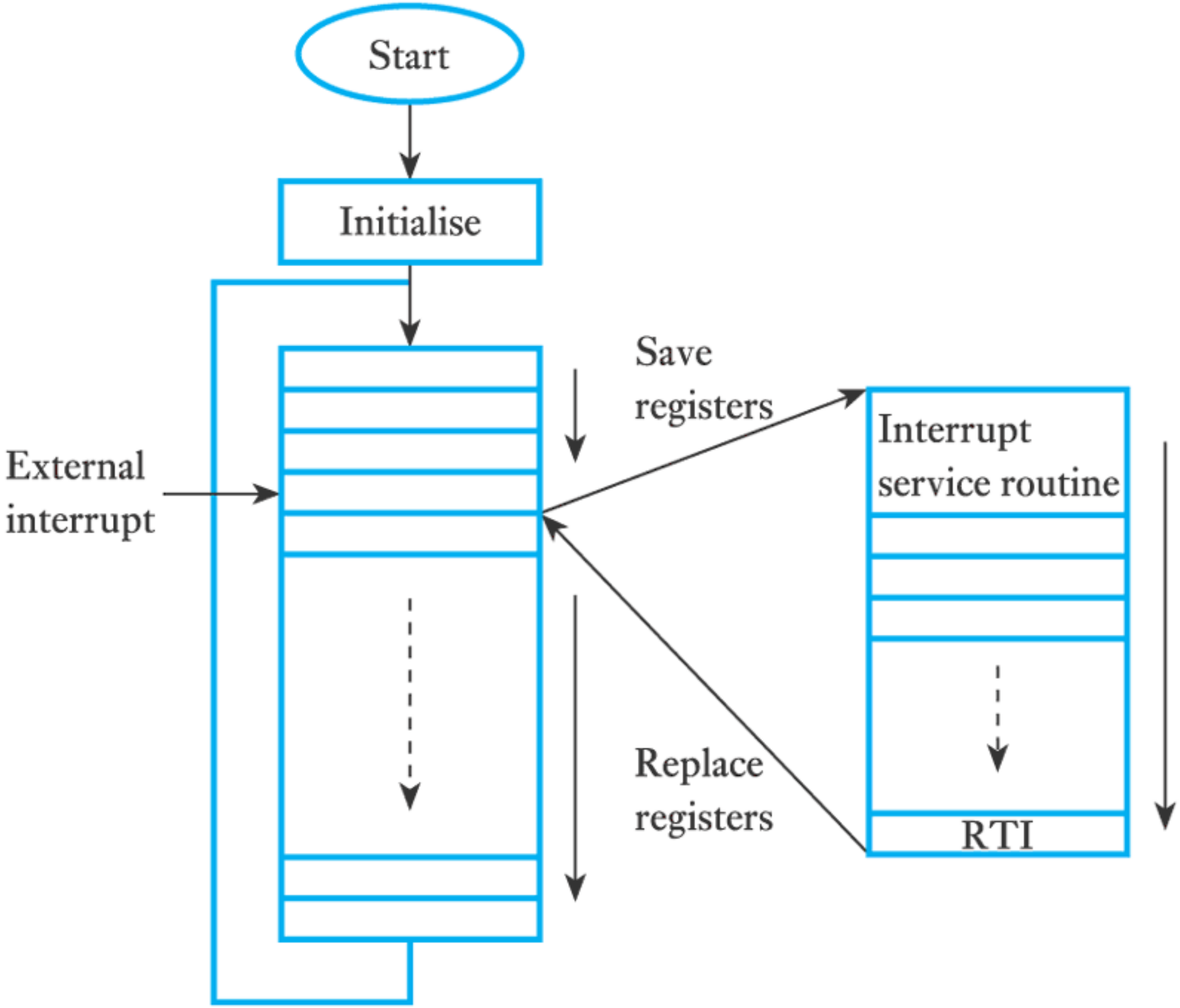
Alternative: Interrupts

Video 27G

- the interrupt mechanism



– a more detailed view of the interrupt mechanism



When an Interrupt Occurs

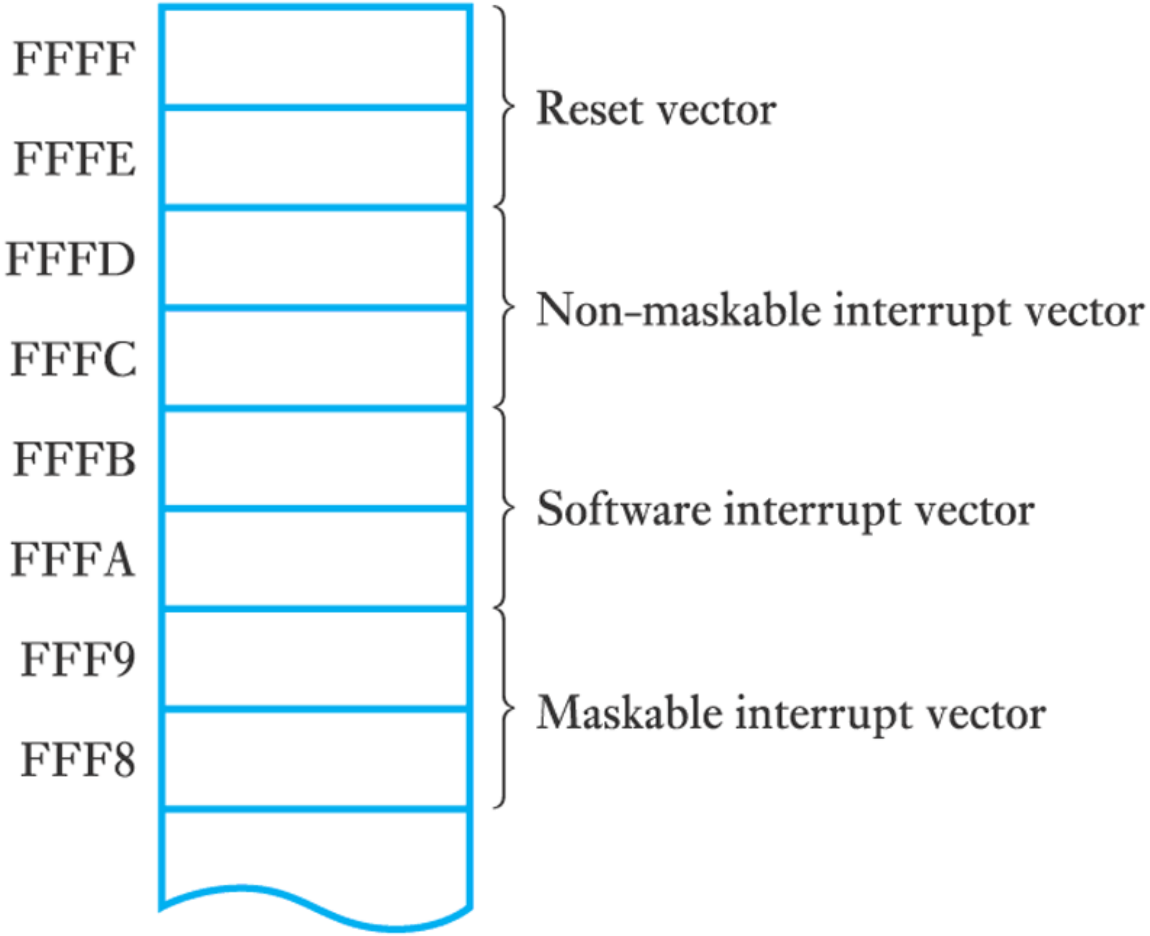
- Finish the current instruction
- Save minimal state information on stack
- Transfer to the interrupt handler, also known as the interrupt service routine (ISR)

  But there is more to it than this…How do we know which device interrupted?
- And what happens if two (or more) devices request an interrupt at the same time?

Interrupt Vector Table

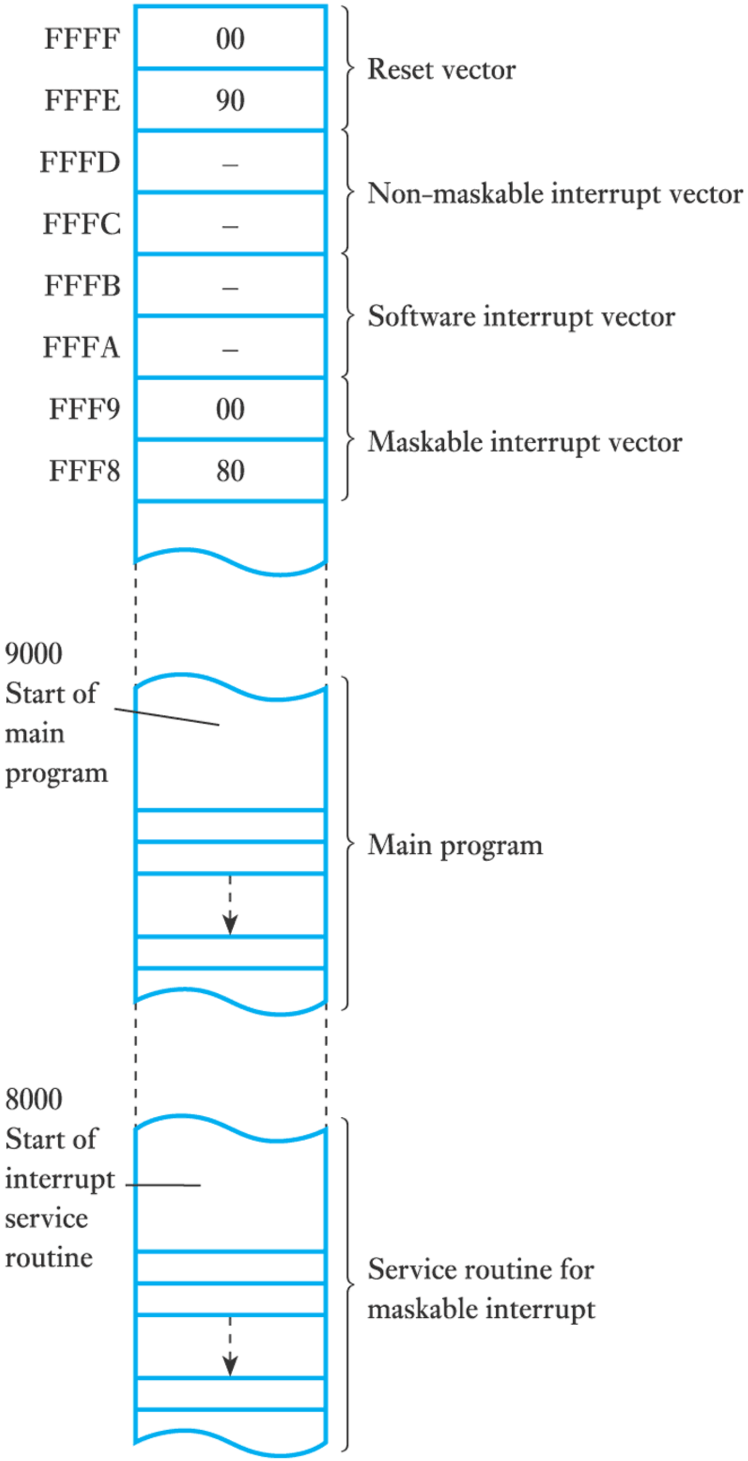
- When an interrupt occurs, control of the program moves to the interrupt handling routine
  - Event similar to subroutine
  - How do we know where the handler routine is??
- The address of the handler is provided by the interrupt vector table - IVT
  - IVT has one entry for each type of interrupt
  - Each entry is indexed by interrupt type, and includes a pointer to the handler

- **Vectors**
  - – the vectors of a "typical" 8-bit processor…


FFFF
FFFE
FFFD
FFFC
FFFB
FFFA
FFF9
FFF8
Reset vector
Non-maskable interrupt vector
Software interrupt vector
Maskable interrupt vector

– setting the reset and interrupt vectors

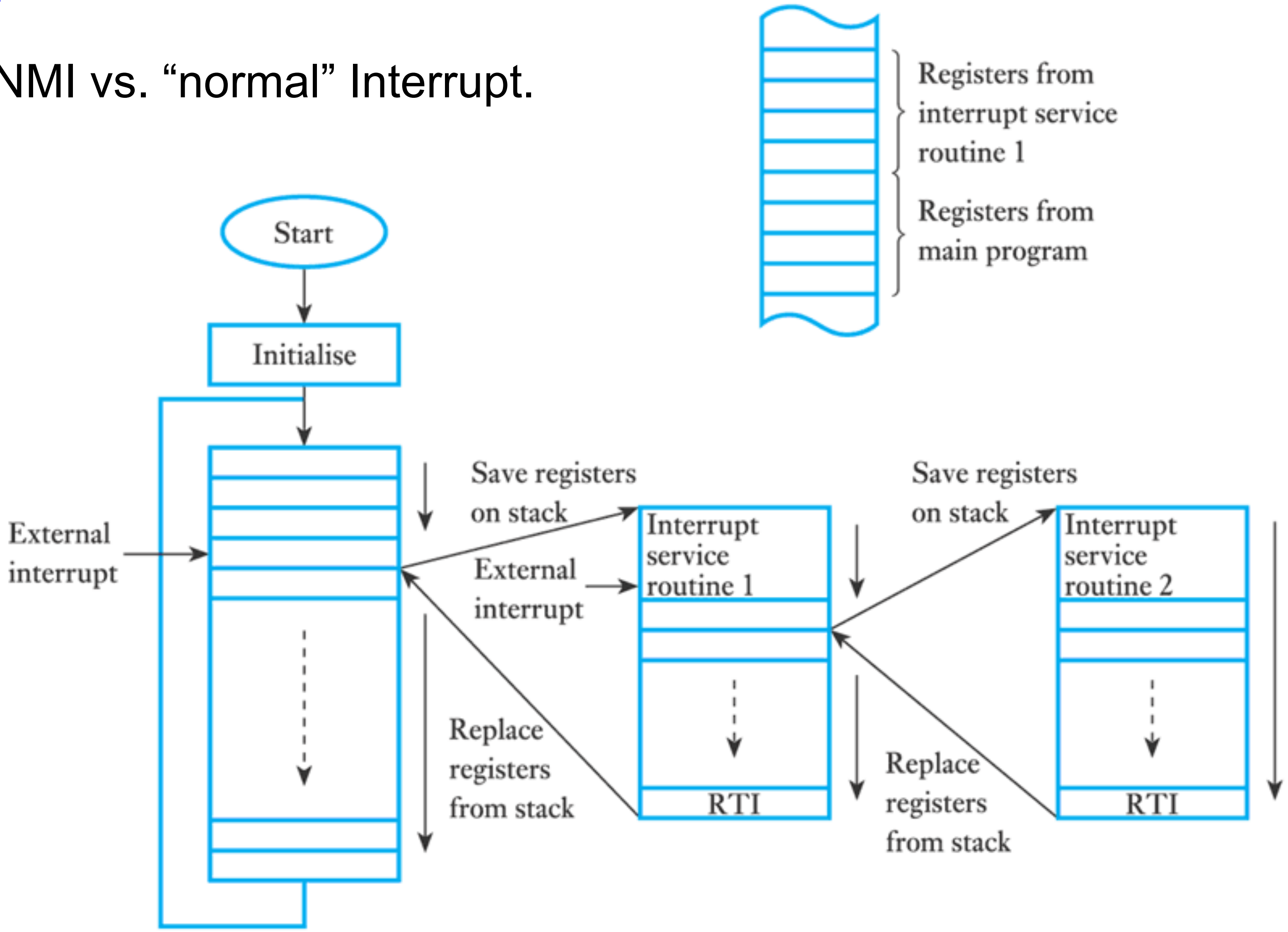
- The example is taken from an older processor (6502), but the concept is the same in our AVR processor



FFFF
00
FFFE
90
Reset vector
FFFD
–
FFFC
–
Non-maskable interrupt vector
FFFB
–
FFFA
–
Software interrupt vector
FFF9
00
FFF8
80
Maskable interrupt vector
9000
Start of main program
Main program
8000
Start of interrupt service routine
Service routine for maskable interrupt

– multiple interrupt handling with a stack

– E.g. NMI vs. “normal” Interrupt.


Registers from interrupt service routine 1
Registers from main program
Start
Initialise
External interrupt
Save registers on stack
External interrupt
Interrupt service routine 1
RTI
Replace registers from stack
Save registers on stack
Interrupt service routine 2
RTI
Replace registers from stack

Types of interrupts

- Ignorable interrupts (or Maskable)
  - – Most often used
  - – Good for using when computer needs to do something more important
  - – When the interrupt mask is set, interrupts are hidden and therefore are ignored.
- Non-ignorable interrupts (Non-maskable)
  - NMI's take precedence and interrupt any task
  - Example: RESET…..

Interrupt Sources

- Interrupts are generally classified as
  - internal or external
  - software or hardware
- An external interrupt is triggered by a device originating off-chip
- An internal interrupt is triggered by an on-chip component

Interrupt Sources

- Hardware interrupts occur due to a change in state of some hardware
- Software interrupts are triggered by the execution of a machine instruction
  - *Typical example: Timer reaches a specific value*

Interrupt Handler

- An interrupt handler (or interrupt service routine) is a function ending with the special return from interrupt instruction (RETI)
- Interrupt handlers are not explicitly called; their address is placed into the processor's program counter by the interrupt hardware

Interrupt Vectors (in AVR)

- The interrupt handler for interrupt k is located at address 2(k-1) in program memory
  - Address $0000 is the reset interrupt
  - Address $0002 is external interrupt 0
  - Address $0004 is external interrupt 1
- Because there is room for only one or two instructions, each interrupt handler begins with a jump to another location in program memory where the rest of the code is found
  - *"JMP handler"* is a 32-bit instruction, hence each handler is afforded 2 words of space in this low memory area

Interrupt Vector Table

- The 43 instructions at address $0000 through $0055 comprise the interrupt vector table
    - This is why our program starts at $0056…
- These jump instructions vector the processor to the actual service routine code
    - A long JMP is used so the code can be at any address in program memory
- An interrupt handler that does nothing could simply have an RETI instruction in the table
    - *…safeguard – should not be needed!*

IS8.20

Atmega 32U4 Interrupt table

Separate vectors in the table

Priority in accordance with position in table.

Lower address have higher priority.

Reset has top priority.

9.1 Interrupt Vectors in ATmega16U4/ATmega32U4

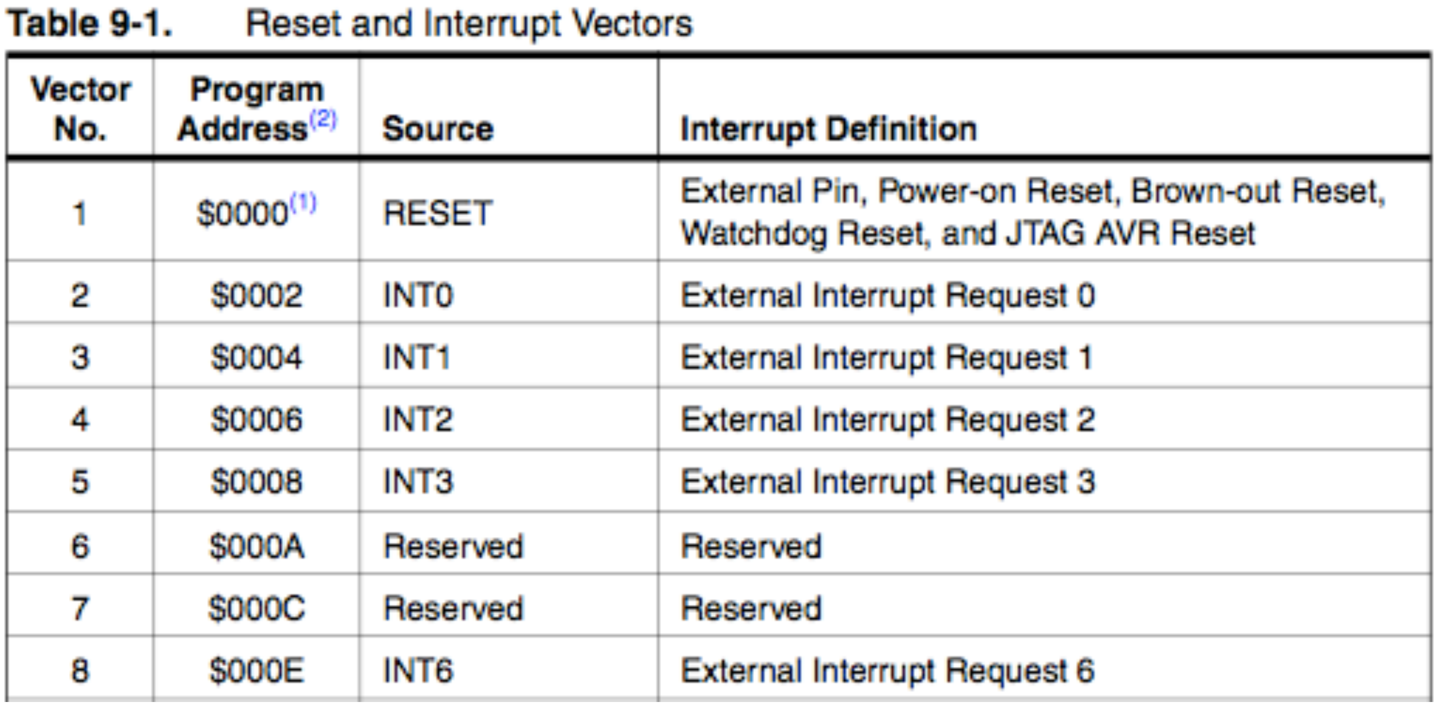
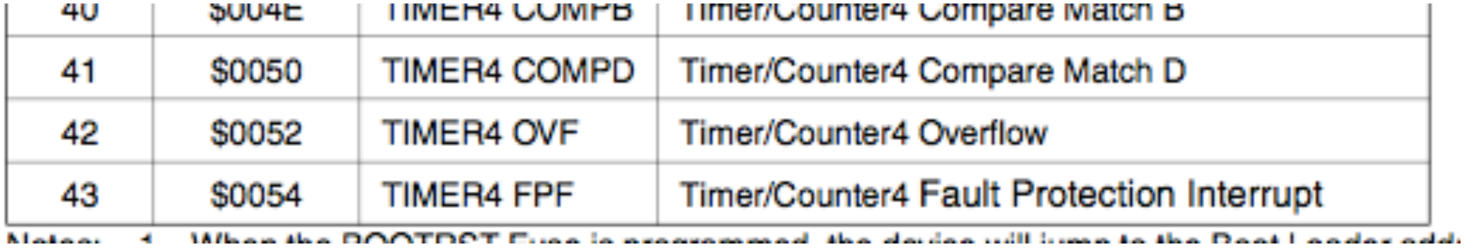


Table 9-1. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	Reserved	Reserved
7	$000C	Reserved	Reserved
8	$000E	INT6	External Interrupt Request 6



40	$004E	TIMER4 COMPB	Timer/Counter4 Compare Match B
41	$0050	TIMER4 COMPD	Timer/Counter4 Compare Match D
42	$0052	TIMER4 OVF	Timer/Counter4 Overflow
43	$0054	TIMER4 FPF	Timer/Counter4 Fault Protection Interrupt

Notes: 1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader addr

Interrupt Enabling

- Each potential interrupt source can be individually enabled or disabled
  - The reset interrupt is the one exception; it cannot be disabled
- The global interrupt flag must be set (enabled) in SREG, for interrupts to occur
  - Again, the reset interrupt will occur regardless *(i.e. it is "Non-Maskable"!)*

Interrupt Actions

- If
  - global interrupts are enabled
  - AND a specific interrupt is enabled
  - AND the interrupt condition is present
- Then the interrupt will occur
- What actually happens?
  - At the completion of the current instruction,
    - the current PC is pushed on the stack
    - global interrupts are disabled
    - the proper interrupt vector address is placed in PC

Return From Interrupt

- The RETI instruction will
  - pop the address from the top of the stack into the PC
  - set the global interrupt flag, re-enabling interrupts
- This causes the next instruction of the previously interrupted program to be executed
  - At least one instruction will be executed before another interrupt can occur

Stack

- Since interrupts require stack access, it is essential that the reset routine initialize the stack before enabling interrupts
- Interrupt service routines should use the stack for temporary storage so register values can be preserved

When to Use Interrupts

- Interrupts should be used for infrequent events (1000's of clock cycles) processor can do something else in the meantime
  - keyboard strokes
  - 1ms clock ticks
  - incoming packet from network
  - analog to digital conversion
  - Example: start a disk transfer,
    disk interrupts when transfer is finished
- When event is rare but critical
  - Example: building on fire -- save and shut down!

When to Use Interrupts – response time

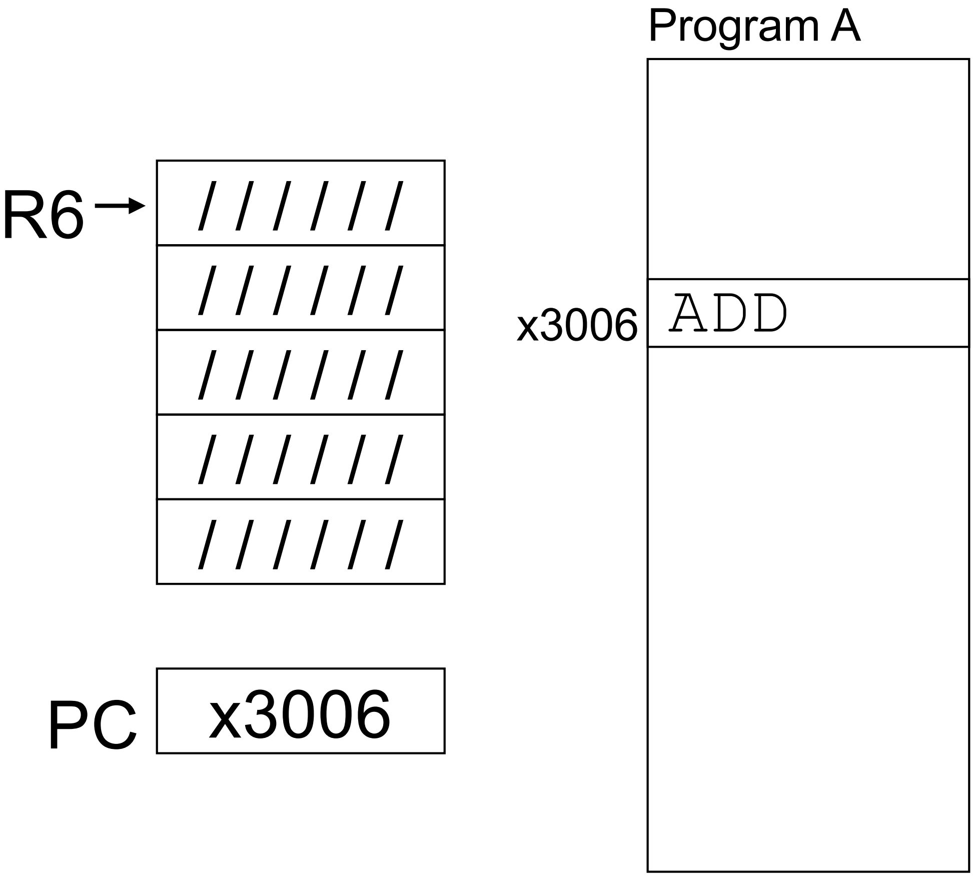
- uC response time to interrupts is very fast
  - AVR: 4 cycles max
  - 80386: 59 cycles min, 104 cycles max;

- If response time is really critical, a "tight" *polling loop* is used - polling can be faster than interrupts......,
- **but other processing is on hold!**

How does Processor Handle It?

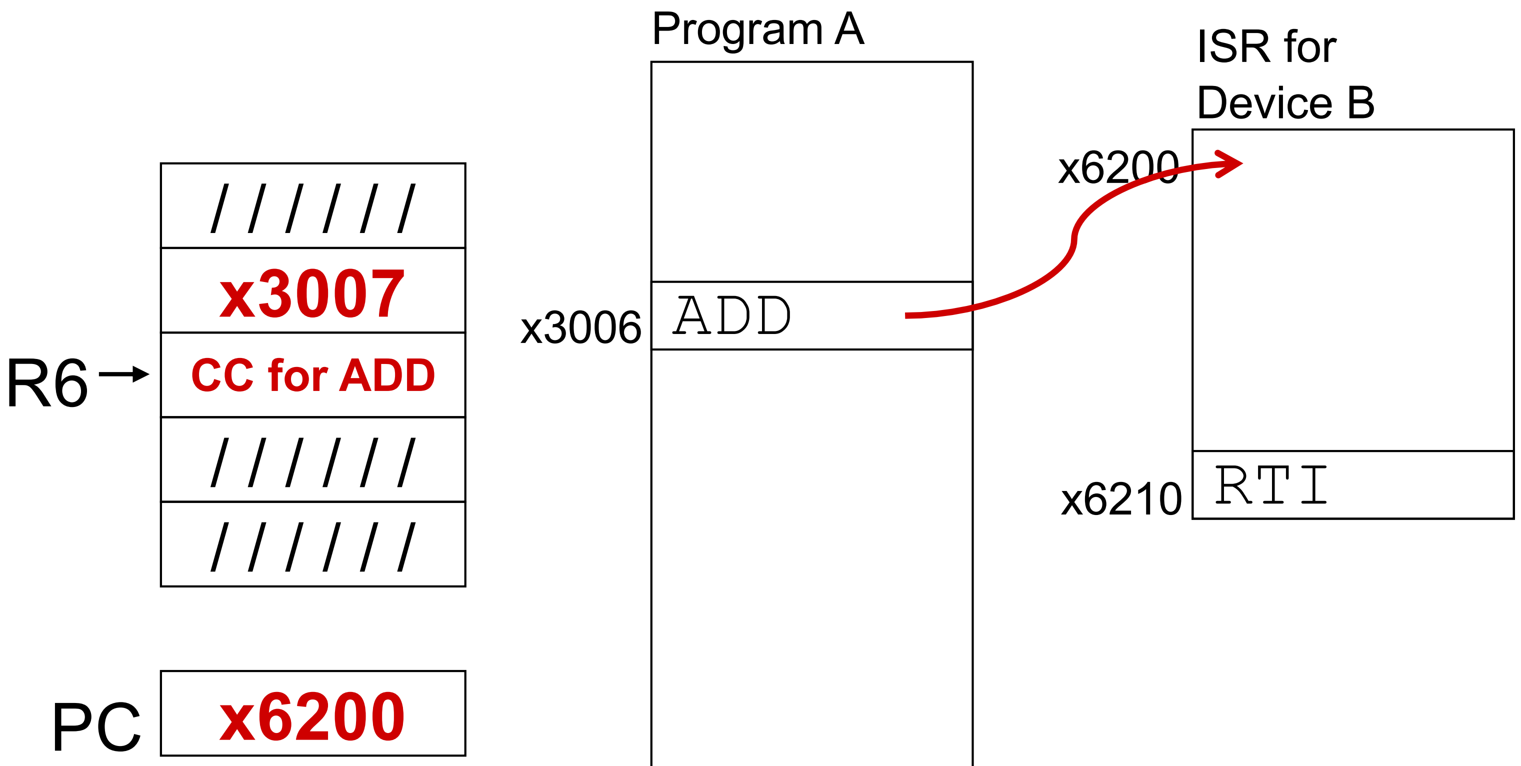
- Look at INT signal just before entering FETCH phase.
  - If INT=1, don’t fetch next instruction.
  - Instead:
    - save state (PC and condition codes) on stack
    - use vector to fetch ISR starting address; put in PC
- After service routine,
  RTI instruction restores condition codes and old PC
  - need a different return instruction,
    because RET only gets PC from stack, no condition codes
- Processor only checks between STORE and FETCH phases

Example (1)



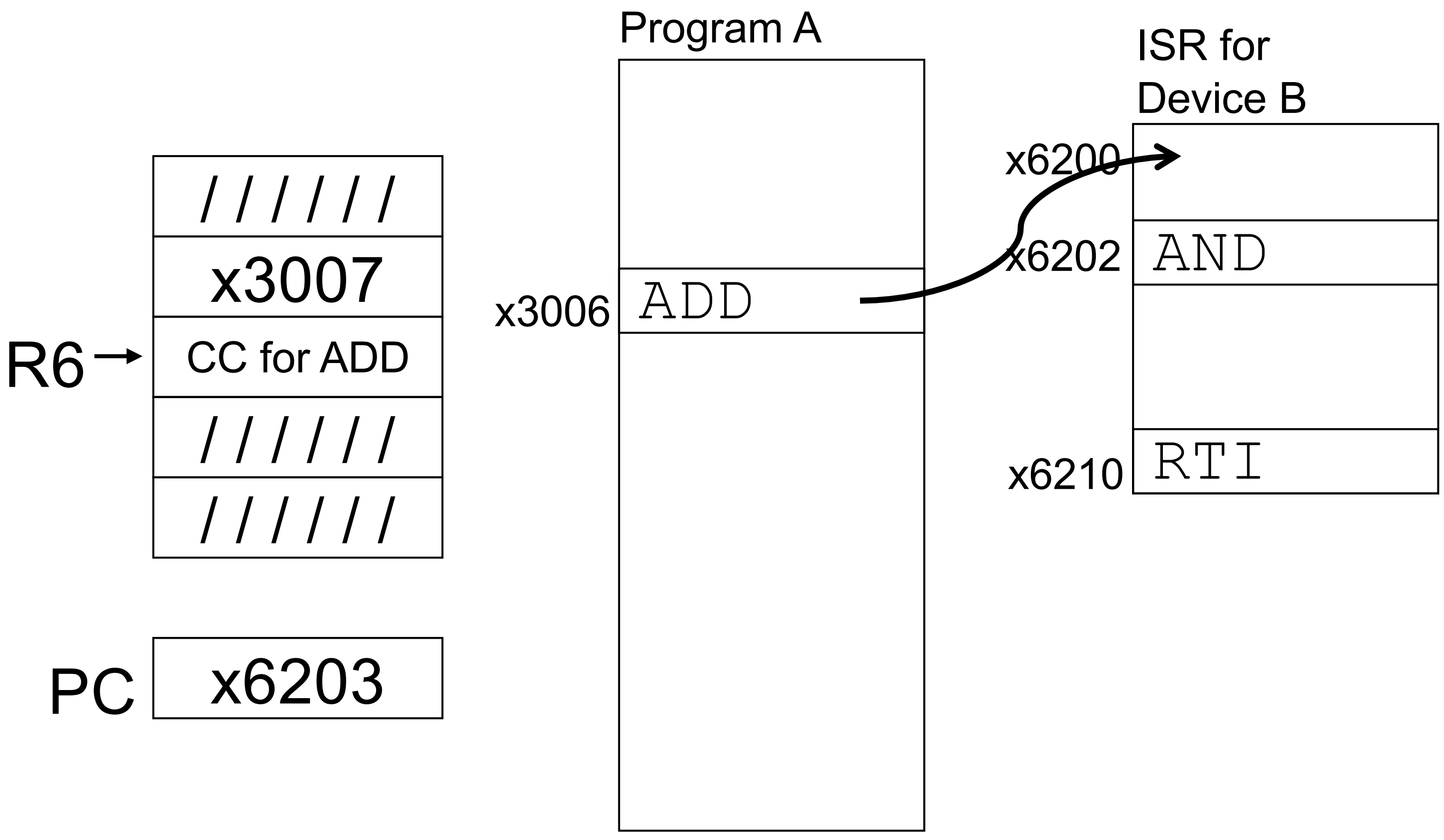
Executing ADD at location x3006 when Device B interrupts.

Example (2)



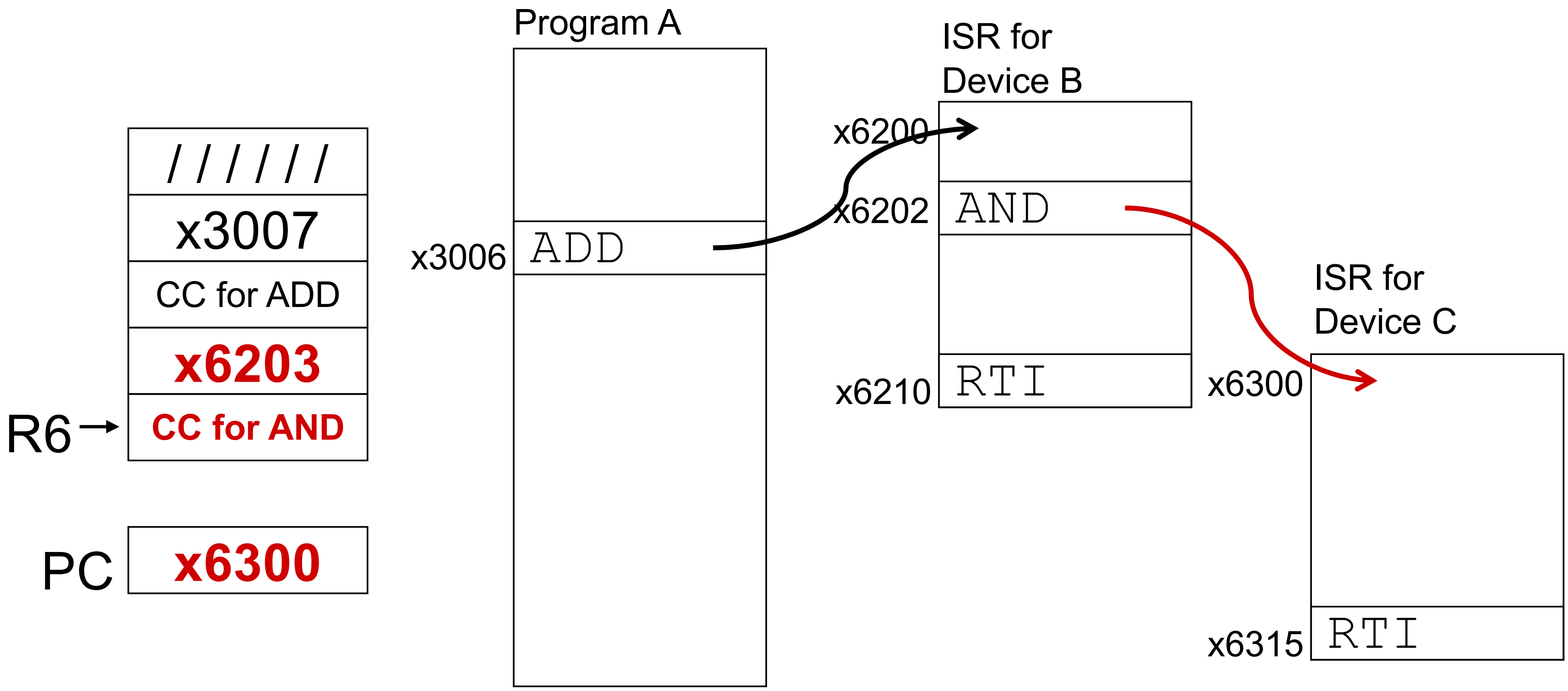
Push PC and Condition Codes (status reg) onto stack, then transfer to Device B service routine (at x6200).

Example (3)



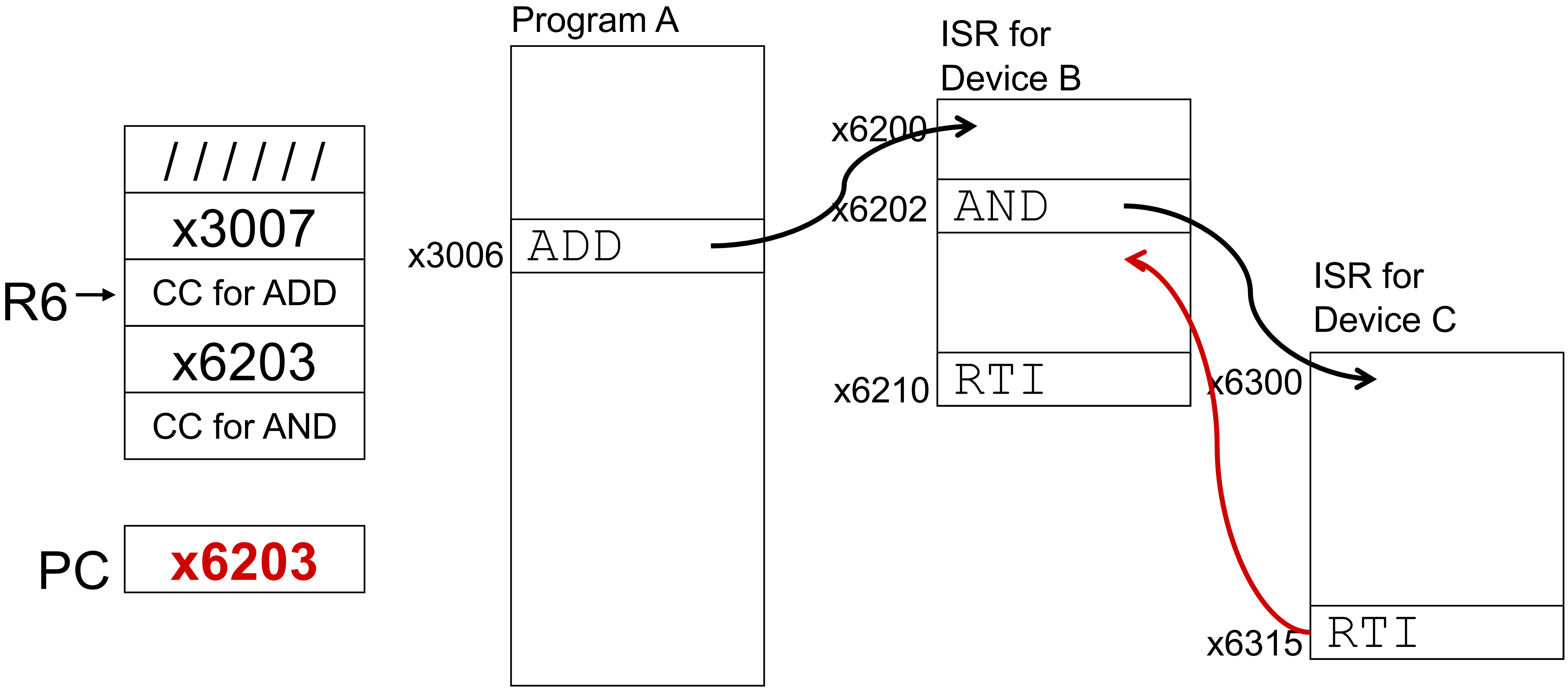
Executing AND at x6202 when Device C interrupts.

Example (4)



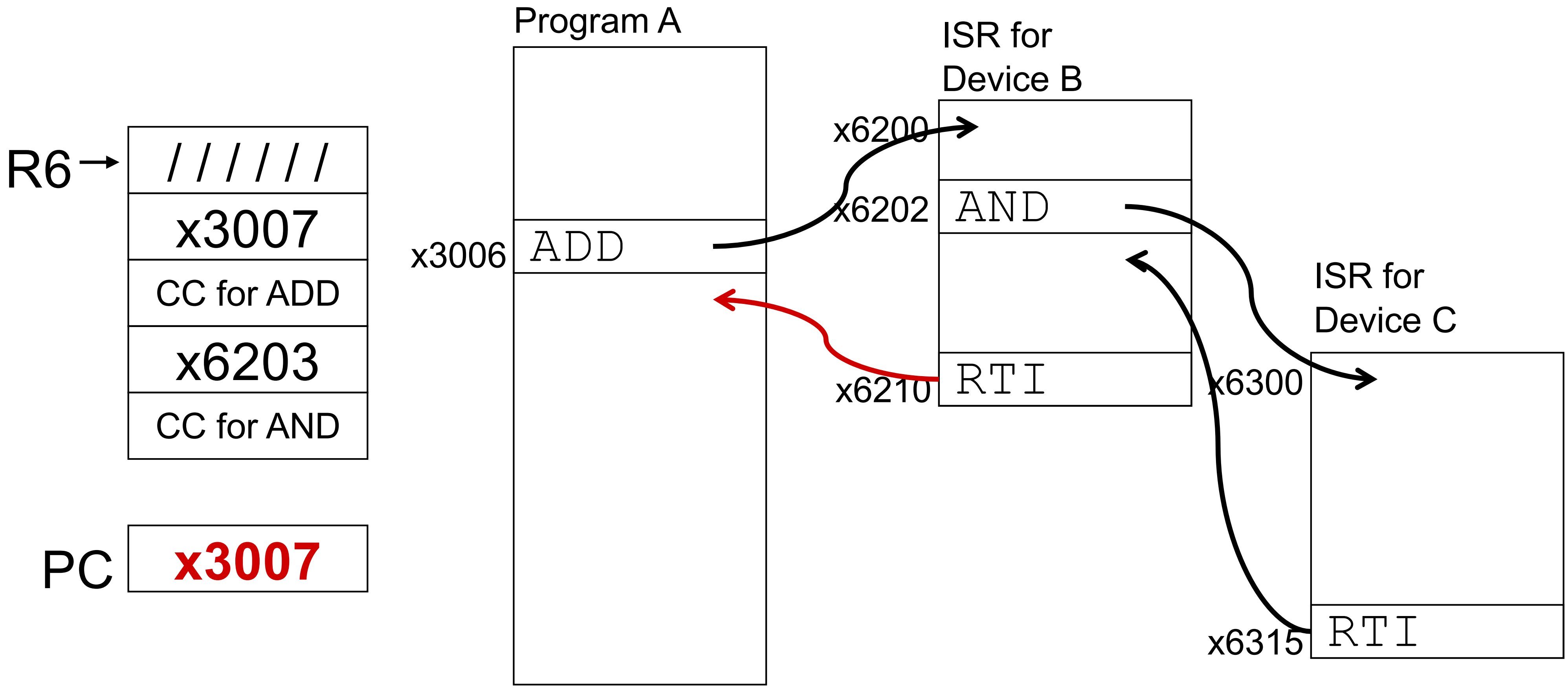
Push PC and condition codes onto stack, then transfer to Device C service routine (at x6300).

Example (5)



Execute RTI at x6315; pop CC and PC from stack.

Example (6)



Execute RTI at x6210; pop CC and PC from stack.
Continue Program A as if nothing happened.

Interrupts

Enabling interrupts

1. Global Interrupt Enable (GIE) bit must be set in the status register (SREG)

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

***Global Interrupt Enable (GIE)**:*
This bit must be set for interrupts to be enabled. To set GIE:

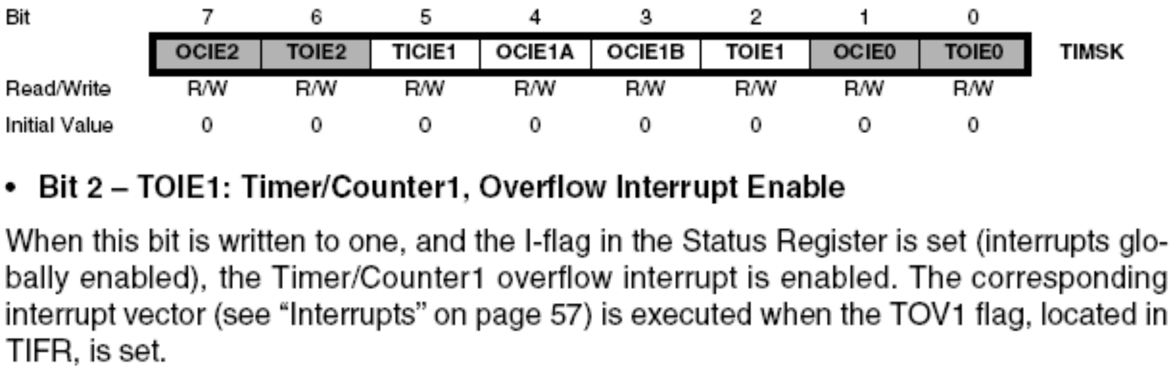
```
sei();      //global interrupt enable (cli();//disable…)
```

It is cleared by hardware after an interrupt has occurred, but may be set again by software [ISR(xxx_vec, ISR_NOBLOCK)] or manually with the `sei()` to allow nested interrupts.
It is set by the RETI instruction to enable subsequent interrupts. This is done automatically by the compiler.

Interrupts - example

Enabling interrupts

2. The individual interrupt enable bits must be set in the proper control register.
-For example, for timer counter 1, the timer overflow bit (TOV)



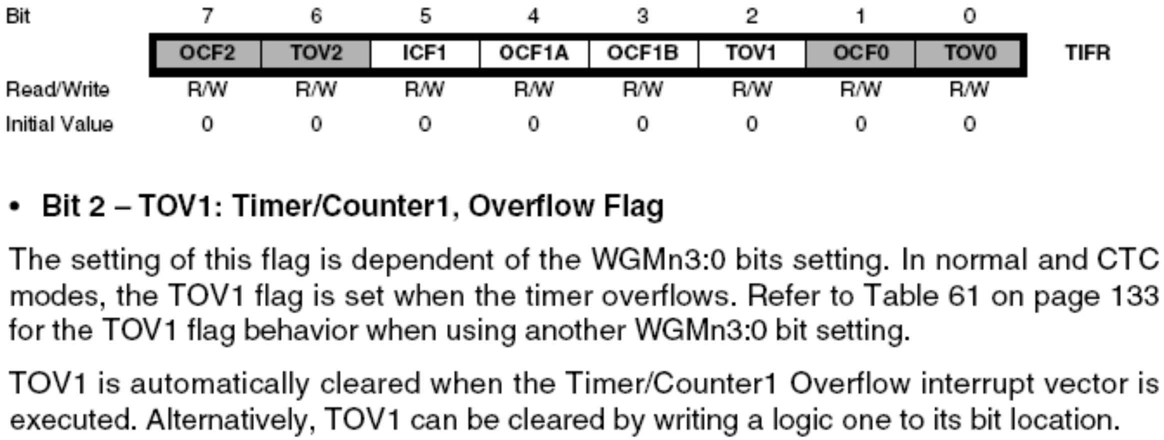
Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 overflow interrupt is enabled. The corresponding interrupt vector (see "Interrupts" on page 57) is executed when the TOV1 flag, located in TIFR, is set.

TOIE1 must be set to allow the TCNT1 over flow bit to cause an interrupt

The interrupt occurs when the TOV1 flag becomes set. Once you enter the ISR, it is reset automatically by hardware.



Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 2 – TOV1: Timer/Counter1, Overflow Flag**

The setting of this flag is dependent of the WGMn3:0 bits setting. In normal and CTC modes, the TOV1 flag is set when the timer overflows. Refer to Table 61 on page 133 for the TOV1 flag behavior when using another WGMn3:0 bit setting.

TOV1 is automatically cleared when the Timer/Counter1 Overflow interrupt vector is executed. Alternatively, TOV1 can be cleared by writing a logic one to its bit location.

Interrupts - ISR Signal Names in C

Things to note:

1 - The "avr/interrupt.h" files must be included to use interrupts.

2 - The ISR() function is a general interrupt service routine for all interrupts. This means that you can have several ISR() functions in an AVR C program. Examples:

```
ISR(SPI_STC_vect){};       //SPI serial transfer complete
ISR(TIMER0_COMP_vect){};
                           //TCNT0 compare match A
ISR(TIMER0_OVF_vect){};    //TCNT0 overflow
ISR(INT2_vect){};          //External Interrupt Request 2
```

These names come from:
http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

Interrupts - guidelines

- understand how often the interrupt occurs
- understand how much time it takes to service each interrupt
- make sure there is time to service all int. and to still get work done in the main loop
- **there's only 1 sec of compute time per second to get everything done!**
- Keep ISRs short and simple. (short time, not short code)
    - Do only what has to be done then RETI.
    - Long ISRs may preclude others from being run
    - Ask yourself, “Does this code really need to be here?”

Interrupts – possible problems

ISRs are never called by the main routine. Thus,

- nothing can be passed to them (there is no “passer”)
- they can return nothing (its not a real function call)

So, how can it effect the main program flow?

- use a global var? Yech! – use “static” to limit access to the C-file!
- use *volatile* type modifier

Compiler has a optimizer component

- 02 level optimization can optimize away some variables
- it does so when it sees that the variable cannot be changed within the scope of the code it is looking at
- variables changed by the ISR are outside the scope of main()
- thus, they get optimized away

Volatile tells the compiler that the variable is shared and may be subject to outside change elsewhere.

Interrupts - Example

```
volatile uint_8 tick; //keep tick out of regs!

ISR(TIMER1_OVF_vect){
  tick++; //increment my tick count
}

main(){
while(tick <= 60){
    bla, bla, bla...
}
```

- Without the volatile modifier, -02 optimization removes *tick* because nothing in while loop can ever change *tick*.

Useful links…

- http://www.scriptoriumdesigns.com/embedded/interrupts.php
- http://www.scriptoriumdesigns.com/embedded/interrupt_examples.php
- http://www.csee.umbc.edu/~tinoosh/cmpe311/notes/Interrupts.pdf
- http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html
- http://www.avrfreaks.net/forum/tut-newbies-guide-avr-interrupts?name=PNphpBB2&file=viewtopic&t=89843

Extra: One use of interrupts and timers… (1/2)

```
struct task {
        uint16_t SP;
};
struct tasks[2];
uint8_t runningtask = 0;

int main (){
        init_task();
        sei();
        task0();
        while (1) {}
};
```

```
void task0 {
        while (1){
        // do stuff
        }
};
void task1 {
        while (1){
        // do stuff
        }
};
```

Extra: One use of interrupts and timers … (2/2)

```
/* We set up a stack for
added task1 and add 3 bytes
to ”stack1” via p */

void init_task {
      uint16_t *p;
      tasks[1].SP =SP-200;
      p = tasks[1].SP;
      *p = 0x80; //8 bit
      p--;
      *p = &task1; // 16bit
      tasks[1].SP -=3;
      //…init timer too!
} // init_task
```

```
ISR (Timer1_OVF_vect){
      if (runningtask==0) {
            tasks[0].SP=SP;
            SP=tasks[1].SP;
            runningtask=1;
      } else {
            tasks[1].SP=SP;
            SP=tasks[0].SP;
            runningtask=0;
      };
} // ISR
```