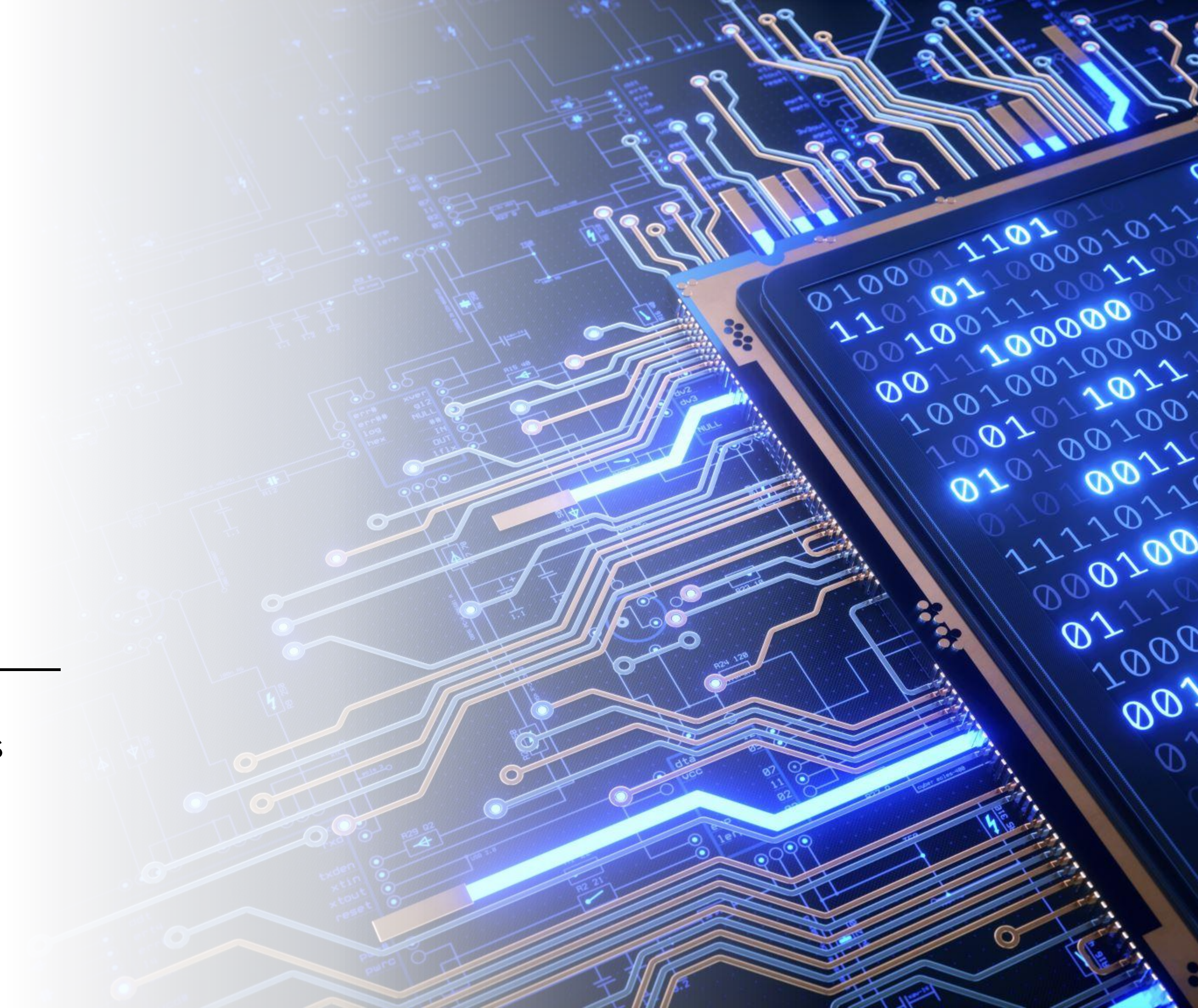
Interrupt Handling

Handling Asynchronous Events



Events

- Embedded systems, just like operating systems or a web browser, must handle events
- Events can
  - come from outside the system, such as from a switch being pressed, a mouse being clicked, etc. or
  - come from an internal operation, such as a counter hitting a certain value

Handling Events

- An event can happen at almost any time
- Only two choices for handling:
  - Poll, which means repeatedly checking for the event, or
  - Interrupts
    - MCU stops whatever it is doing
    - Calls code to handle the event
    - Resumes the previous task

Some Terminology

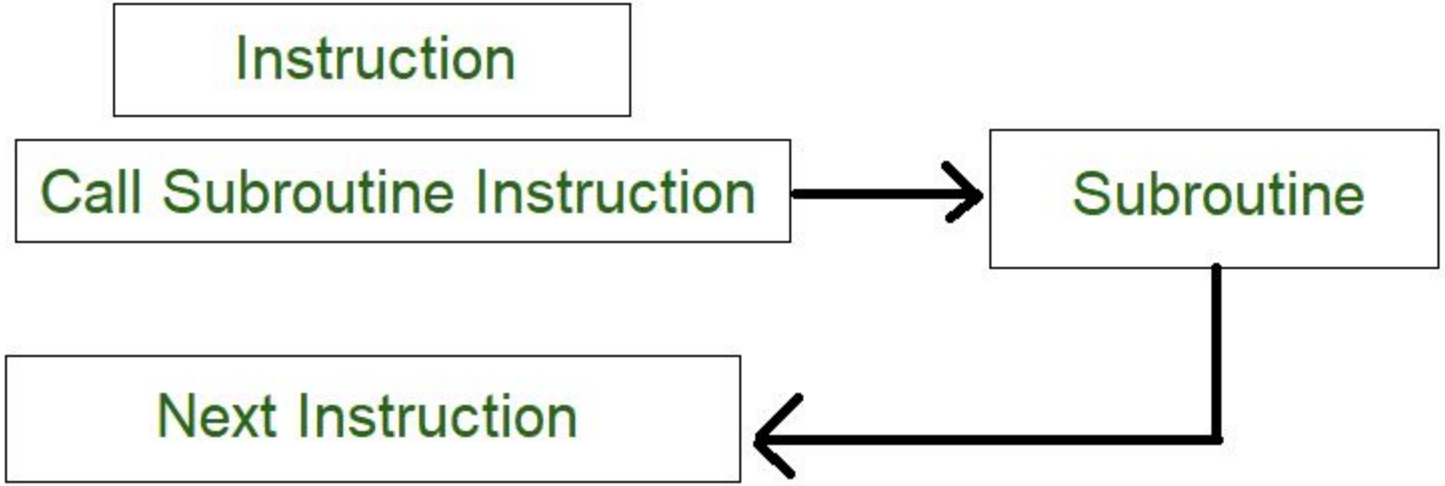
- Program Counter (PC): pointer in memory to the next instruction to be executed
- Stack: A block in memory that is used for
  - temporary variables
  - passing variables to subroutines
  - return values for subroutines and interrupts
- Stack Pointer (SP): pointer to the last item put on the stack

What is a Stack?

- An array or list with associated functions so that the last element added is the first element removed
  - Last In First Out (LIFO)
- Think in terms of a stack of plates
  - Adding a plate means adding it to the top of the stack
  - Removing a plate comes from the top, so that last added is the first removed
- Important terms:
  - Push: add an element to the stack
  - Pop: remove an element from the stack
  - Peek: look at the top element without removing it
- Example in C:
  - https://www.geeksforgeeks.org/stack-data-structure-introduction-program/

Calling a Subroutine

- A set of instructions that are used repeatedly in a program can be referred to as Subroutine.

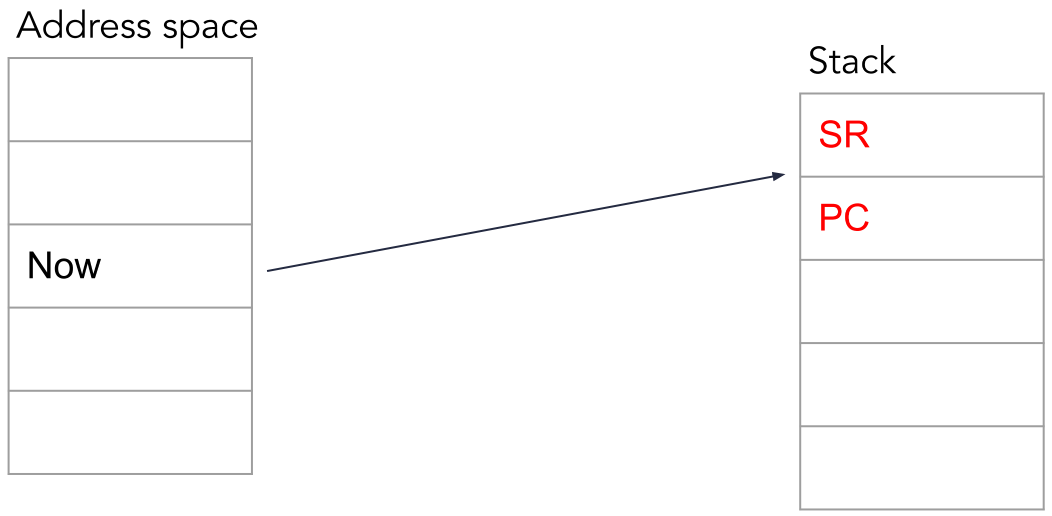


https://www.geeksforgeeks.org/subroutine-subroutine-nesting-and-stack-memory/

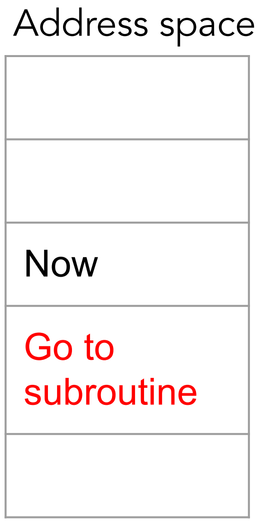
Calling a Subroutine

1. PC and SR are pushed on the stack
2. PC is loaded with location of subroutine
3. Arguments to sub are pushed onto stack
4. Subroutine is executed, pulling arguments from stack
5. Subroutine pushes return values on to stack (if not void)
6. Main program pops return values
7. PC and SR are popped from stack
8. Execution continues

1. PC and SR are pushed on the stack

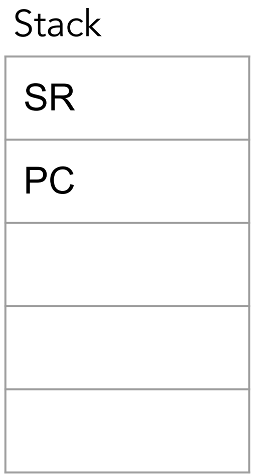


2. PC is loaded with location of subroutine

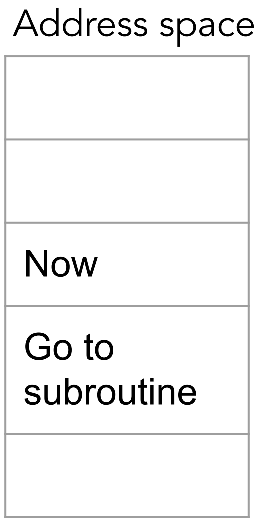


PC = subroutine address

```
Subroutine (var1,var2 ){
//do stuff
}
```

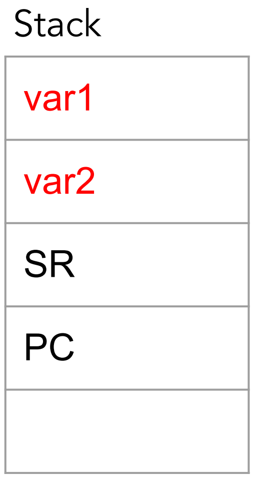


3. Arguments to sub are pushed onto stack

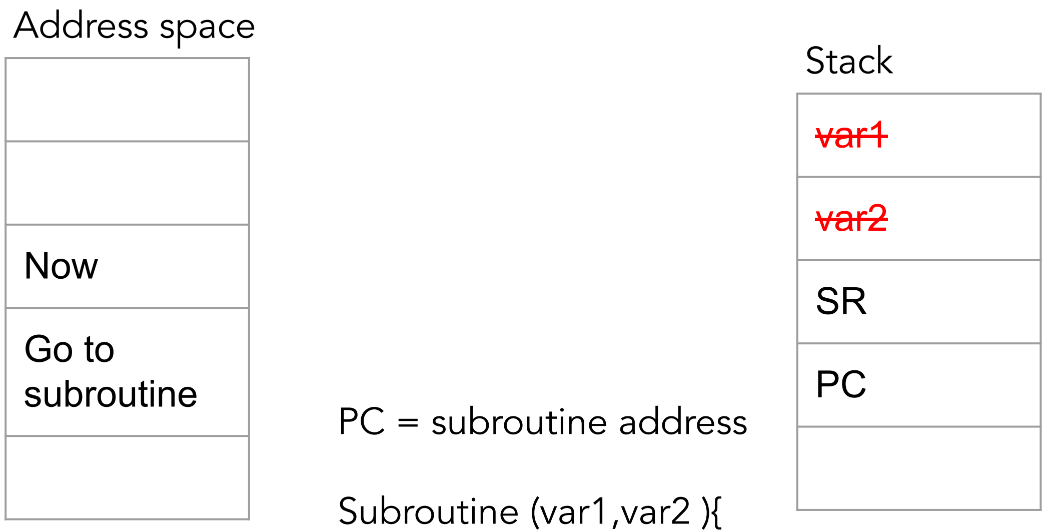


PC = subroutine address

```
Subroutine (var1,var2 ){
//do stuff
}
```



4. Subroutine is executed, pulling arguments from stack

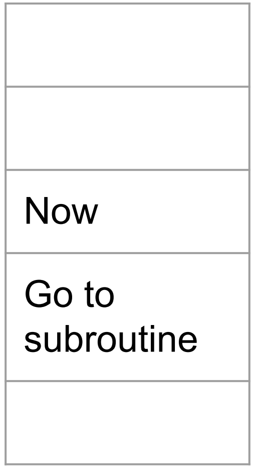


PC = subroutine address

```
Subroutine (var1,var2 ){
//do stuff
}
```

5. Subroutine pushes return values on to stack (if not void)

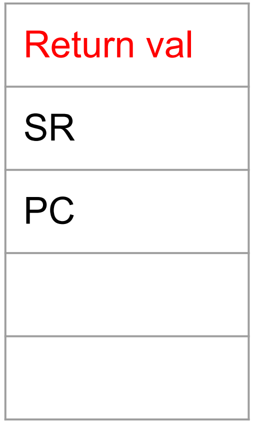
Address space



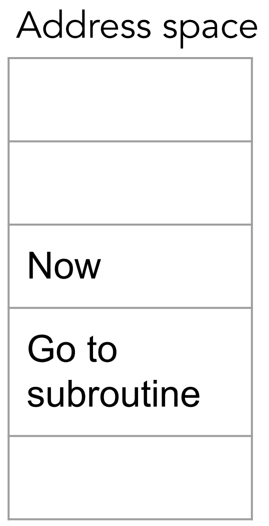
PC = subroutine address

```
Subroutine (var1,var2 ){
//do stuff
}
```

Stack

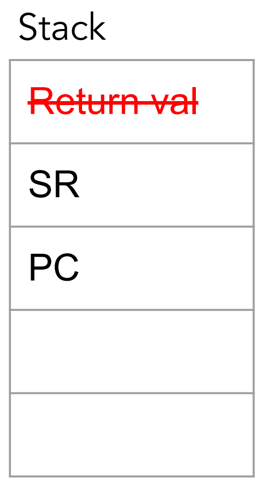


6. Main program pops return values



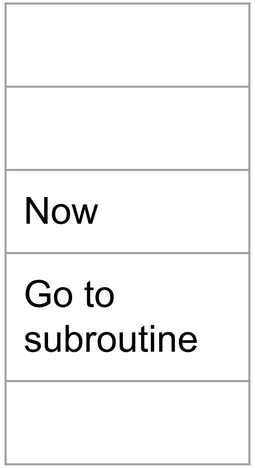
PC = subroutine address

```
Subroutine (var1,var2 ){
//do stuff
}
```

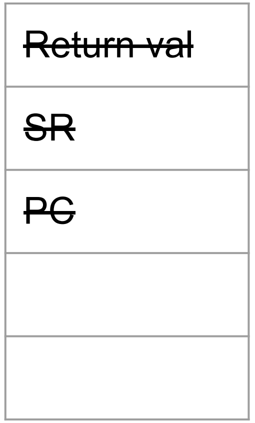


7. PC and SR are popped from stack

Address space



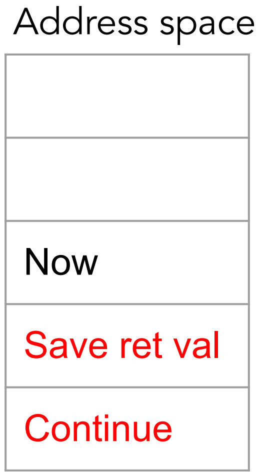
Stack



PC = subroutine address

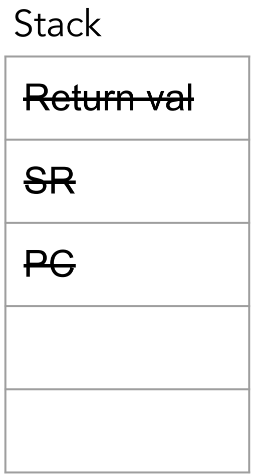
```
Subroutine (var1,var2 ){
//do stuff
}
```

8. Execution continues



PC = subroutine address

```
Subroutine (var1,var2 ){
//do stuff
}
```

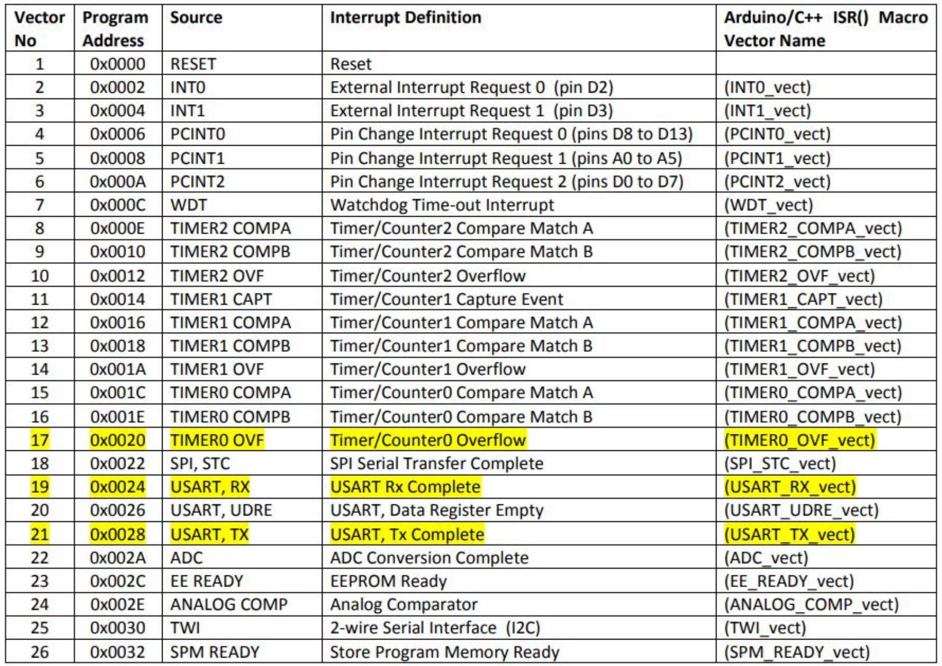


Interrupt Service Routine (ISR)

- Code block that is executed in response to an interrupt
- Process is very similar to subroutine calls
- A few differences between subroutines and ISRs
  - Interrupts can happen at *any* time, while subroutines are called explicitly
  - Variables cannot be passed to ISRs on the stack
  - Variables that would be used by an interrupt must be volatile
  - ISRs must not call functions such as Serial.print, which use interrupts themselves

Interrupt Vector Table and Interrupt Priority

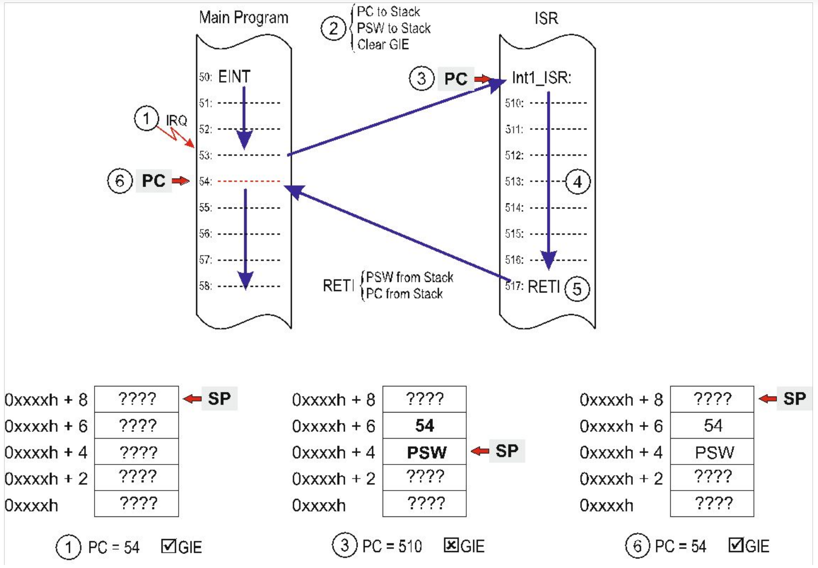
- Each interrupt has an address in a table that is used to point to the ISR for that interrupt
- The order that they appear in the table determines the priority
- Example: INT0 is higher priority than TIMER0 OVF, meaning that it would be processed first if both needed processing



Vector No	Program Address	Source	Interrupt Definition	Arduino/C++ ISR() Macro Vector Name
1	0x0000	RESET	Reset	
2	0x0002	INT0	External Interrupt Request 0 (pin D2)	(INT0_vect)
3	0x0004	INT1	External Interrupt Request 1 (pin D3)	(INT1_vect)
4	0x0006	PCINT0	Pin Change Interrupt Request 0 (pins D8 to D13)	(PCINT0_vect)
5	0x0008	PCINT1	Pin Change Interrupt Request 1 (pins A0 to A5)	(PCINT1_vect)
6	0x000A	PCINT2	Pin Change Interrupt Request 2 (pins D0 to D7)	(PCINT2_vect)
7	0x000C	WDT	Watchdog Time-out Interrupt	(WDT_vect)
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A	(TIMER2_COMPA_vect)
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B	(TIMER2_COMPB_vect)
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow	(TIMER2_OVF_vect)
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event	(TIMER1_CAPT_vect)
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A	(TIMER1_COMPA_vect)
13	0x0018	TIMER1 COMPB	Timer/Counter1 Compare Match B	(TIMER1_COMPB_vect)
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow	(TIMER1_OVF_vect)
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A	(TIMER0_COMPA_vect)
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B	(TIMER0_COMPB_vect)
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow	(TIMER0_OVF_vect)
18	0x0022	SPI, STC	SPI Serial Transfer Complete	(SPI_STC_vect)
19	0x0024	USART, RX	USART Rx Complete	(USART_RX_vect)
20	0x0026	USART, UDRE	USART, Data Register Empty	(USART_UDRE_vect)
21	0x0028	USART, TX	USART, Tx Complete	(USART_TX_vect)
22	0x002A	ADC	ADC Conversion Complete	(ADC_vect)
23	0x002C	EE READY	EEPROM Ready	(EE_READY_vect)
24	0x002E	ANALOG COMP	Analog Comparator	(ANALOG_COMP_vect)
25	0x0030	TWI	2-wire Serial Interface (I2C)	(TWI_vect)
26	0x0032	SPM READY	Store Program Memory Ready	(SPM_READY_vect)

Servicing an Interrupt

1. IRQ (interrupt request) occurs
2. PC and PSW (status register) pushed to stack
3. PC loaded with address of ISR
4. ISR executed
5. Return from interrupt (RETI) executed
6. Program continues



GIE = Global Interrupt Enable
PSW = Processor Status Register
Image: Jimenez p. 303

Saving the Context

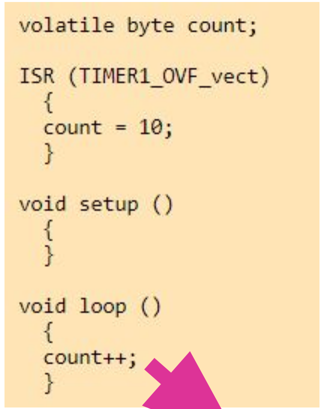
- The *software context* may be defined as the CPU environment as seen by each assembly instruction.
  - Typically, the context may be specified by the set of CPU registers, including
- When an interrupt occurs, the current context must be saved
  - This means any register values must be saved so that they can be restored after the ISR is executed
  - The ISR runs in a different context, in that the register values and PC is uses are different.
  - Moving from one context to another is referred to as *context switching*
- Context switching happens to us
  - Think about being in the middle of working a problem or having a conversation (with someone 6 ft away) and then getting a phone call

Sharing Variables with ISRs

- Shared variables should be declared as volatile
  - Tells the compiler that their value can change at any time
  - Compiler will not place copies of the variable data in general purpose registers and will be forced to read the data from SRAM
- Shared variables must be protected using with critical sections
  - Critical sections prevent multiple threads of execution from modifying resources (variables in this case)

The Need for Critical Sections

- C functions, even increments, take more than one assembly instruction
- In an 8 bit system, more than one instruction required just to load an int
- An ISR might occur in the middle of another instruction

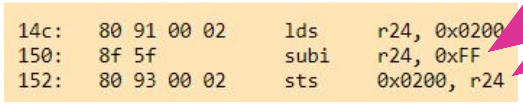


```
volatile byte count;

ISR (TIMER1_OVF_vect)
  {
  count = 10;
  }

void setup ()
  {
  }

void loop ()
  {
  count++;
  }
```



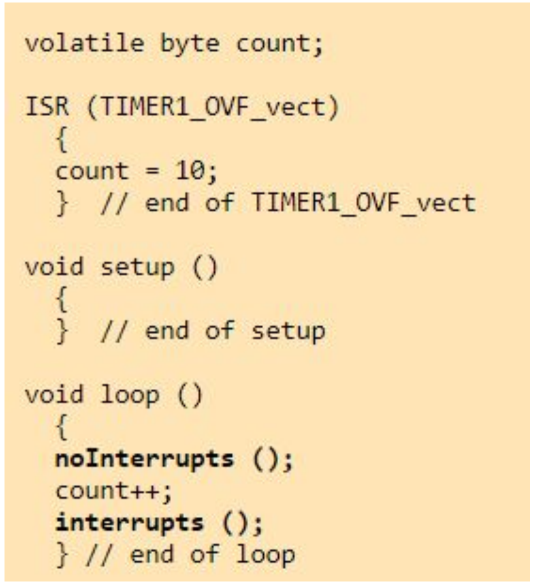
```
14c:   80 91 00 02     lds    r24, 0x0200
150:   8f 5f           subi   r24, 0xFF
152:   80 93 00 02     sts    0x0200, r24
```

The count variable might be changed by the ISR (TIMER1_OVF_vect), however this change is now lost, because the variable in the register was used instead

ISR could occur before either of these are executed

Implementing a Critical Section

- The solution in this case is to turn off all relevant interrupts when accessing a shared variable



```
volatile byte count;

ISR (TIMER1_OVF_vect)
  {
  count = 10;
  }  // end of TIMER1_OVF_vect

void setup ()
  {
  }  // end of setup

void loop ()
  {
  noInterrupts ();
  count++;
  interrupts ();
  } // end of loop
```

Interrupt Masking

- There are many times when we do not want an interrupt to be processed
  - We might be using polling to monitor a process
  - We might simply wish to ignore an event
  - We might be in the midst of processing another interrupt
    - It *is* possible to have interrupts occur during the processing of an interrupt
    - *Nested interrupts* occur when a higher priority interrupt is triggered while an interrupt is being processed

The Need for Speed

- ISRs take time to execute
  - ~2.625 microseconds of overhead – just saving and restoring context
  - External interrupts have additional overhead (~5.125 microseconds)
- The main program loop is halted during the ISR execution
- So
  - Do as little as possible as quickly as possible in ISRs

ISRs on the Arduino

- Library provides macros to make implementation easy
- The basic ISR macro handles saving and restoring of context and disabling global interrupts
- The macros take a parameter specifying which interrupt is to be processed

```
ISR(TIMER1_OVF_vect)
 {
      g_timerCount++;
}
```