1. **Know what the utilities mentioned in the tutorial do, or what kind of information they provide. Given a scenario, you must be able to choose the correct tool.**
    a. msp430-elf-objdump.exe
        i. Disassembler, display information from object files
    b. msp430-elf-readelf.exe
        i. Display information about the contents of ELF format files
    c. MSP430Flasher.exe
        i. Loads binary files into memory
    d. naken_util.exe
        i. Disassembler
    e. GNU Utilities graph from tutorial: strip Remove symbols from an object file.

Utility	Description
as	Assembler
elfedit	Edit ELF files
gdb	Debugger
gprof	Profiler
ld	Linker
objcopy	Copy object files, possibly making changes
objdump	Dump information about object files
nm	List symbols from object files
readelf	Display content of ELF files
strings	List printable strings
size	List total and section sizes

    f. ELF sections and what they mean:

Table 1. ELF Linking View: Common Sections.

Sections	Description
.interp	Path name of program interpreter
.text	Code (executable instructions) of a program; Typically stored in read-only memory.
.data	Initialized read/write data (global, static)
.bss	Uninitialized read/write data (global, static) Often it is initialized by the start-up code
.const/.rodata	Read-only data; typically stored in Flash memory
.init	Executable instructions for process initialization
.fini	Executable instructions for process termination
.ptl	Holds the procedure linkage table
.re.[x]	Relocation information for section [x]
.dynamic	Dynamic linking information
.symtab, .dynsym	Symbols (static/dynamic)
.strtab, .dynstr	String table
.stack	Stack

2. **Know how to figure out the instruction size by looking at the disassembly code.**

a. ? not sure

3. **What do different memory segments (.data, .text, .bss, .stack, etc.) signify?**
    a. .text -- Used for program code.
    b. .bss -- Used for uninitialized objects (global variables).
    c. .data -- Used for initialized non-const objects (global variables).
    d. .const -- Used for initialized const objects (string constants, variables declared const).
    e. .cinit -- Used to initialize C global variables at startup.
    f. .stack -- Used for the function call stack.
    g. .sysmem - Used for the dynamic memory allocation pool.

4. **Know how to work with (set direction, turn on/off, etc.) with the LEDs and switches in both assembly AND C.**
    a. Assembly:

```
bis.b   #0x01, &P1DIR           ; Set P1.0 as output, 0'b0000 0001
bis.b   #0x80, &P4DIR           ; Set P4.7 as output, 0'b1000 0000
bic.b   #0x01, &P1OUT           ; Turn P1.0 off.
bic.b   #0x80, &P4OUT           ; Turn P4.7 off.
; Setting Switch 2's data (i/o).
bic.b   #0x02, &P1DIR           ; Set P1.1 as input for SW2
bis.b   #0x02, &P1REN           ; Enable Pull-Up resister at P1.1
bis.b   #0x02, &P1OUT           ; required for proper IO set up
; Setting Switch 1's data (i/o).
bic.b   #0x02, &P2DIR           ; Set P2.1 as input for SW1
bis.b   #0x02, &P2REN           ; Enable Pull-up resistor at P2.1
bis.b   #0x02, &P2OUT           ; Required for proper IO setup.

xor.b   #0x80, &P4OUT           ; Toggle P4.7
bic.b   #0x01, &P1OUT           ; Turn P1.0 off.
bis.b  #001h, &P1OUT          ; Turn on LED1
```

    b. C:

```
#define RED 0x01            // Red LED Pin
#define GREEN 0x80          // Green LED Pin
P1DIR |= RED;               // P1.0 is output direction for REDLED
P1REN |= BIT1;              // Enable the pull-up resistor at P1.1
P1OUT &= ~RED;              // LED is off at start
```

```
P4DIR |= GREEN;            // P4.7 is output direction for GREENLED
P2REN |= BIT1;             // Enable the pull-up resistor at P2.1
P4OUT &= ~GREEN;           // LED is off at start
P4OUT ^= GREEN;            // Toggle green LED
P1OUT ^= RED;              // Toggle the red LED
P1OUT |= RED;              // Turn on RED LED.
P4OUT |= GREEN;            // Turn on green LED

#define SW1 ((P2IN&BIT1)== 0)
#define SW2 ((P1IN&BIT1)== 0)
P2DIR &= ~BIT1;                  // Configuring Switch 1
P2REN |= BIT1;
P2OUT |= BIT1;

P1DIR &= ~BIT1;                  // Configuring Switch 2
P1REN |= BIT1;
P1OUT |= BIT1;

if (SW1)                     // If switch 1 is pressed
```

5. **Make sure you know how to manipulate different ports in BOTH C and Assembly, and what are the functions of different registers (PxDIR, PxOUT, PxIN, etc.)?**
    a. Assembly:

```
bis.b   #0x01, &P1DIR          ; Set P1.0 as output, 0'b0000 0001
bis.b   #0x80, &P4DIR          ; Set P4.7 as output, 0'b1000 0000
bic.b   #0x01, &P1OUT          ; Turn P1.0 off.
bic.b   #0x80, &P4OUT          ; Turn P4.7 off.
```

    b. C:

```
#define RED 0x01
#define GREEN 0x80        // LED 1 = RED, LED 2 = GREEN
P1DIR |= RED;             // P1.0 is output direction for 0x01, or red.
P1REN |= BIT1;            // Enable the pull-up resistor at P1.1
P1OUT |= BIT1;            // LED is on.
P4DIR |= GREEN;           // P4.7 is output for 0x80, or green.
P2REN |= BIT1;            // Enable pull-up resistor at P2.1
P2OUT |= BIT1;            // LED is on
```

    c. Functions of different registers

i. **PxIN** - input register, reading it returns the logical values on the pins (determined by the external signals). These registers are read-only and bit value of 0 indicates that the corresponding input is low and bit value of 1 indicates that the input is high.
ii. **PxOUT** - output register, writing it sends the value to the corresponding port pin when the pin is configured for the I/O function with output direction. Bit value of 0 will produce low output voltage and bit value of 1 will produce high output voltage.
iii. **PxDIR** - direction register, configures the direction of the corresponding I/O pins (e.g., P2DIR=0xFC = 111111100b configures bits 1 and 0 of port P2 as input pins and all other pins are outputs).
iv. **PxSEL** - selection register, setting bits in this register allows the user to change the port pin function from the standard digital I/O to its corresponding special function. MSP430 interfaces external world predominantly through parallel ports and their default operation is a standard digital input/output (PxSEL=0x00). However, some of the pins have an alternative special function – e.g., they can act as an analog input channel (A0) to the analog-to-digital converter or serial data output of a serial communication interface (TDO). The reference manual specifies special functions for each port pin. They are highly device-specific – developers have to consult the reference manual for the microcontroller they are using.
v. **PxREN** – enables the pull-up or pull-down resistor configuration (e.g, P2REN = 0x02 enables the pull-up resistor on P2.1 that is connected in a series with switch SW1 on the MSP-EXP430F5529LP board.)
vi. Ports P1 and P2 also have ability to serve as sources of interrupts and several registers are associated with this function. These are:
    1. **PxIE** – Port x Interrupt Enable register for enabling/disabling interrupts (x=1, 2),
    2. **PxIFG** – Port x Interrupt Flag register for tracking pending requests,
vii. **PxIES** – Port x Interrupt Edge Select register for selecting type of event that triggers an interrupt – rising edge at the port input (0 -> 1) or falling edge (1 -> 0);
viii. **PxIV** – Port x Interrupt Vector Word. All interrupts associated with a single port share a single interrupt service routine. The highest priority enabled pending interrupt request generates a number in the PxIV register. This number can be used by the code in the corresponding

interrupt service routine to speed up interrupt processing.

6. **How to set a bit to 0 or 1 without affecting other bits (bit masking) in BOTH C and assembly**
    a. Set a specific bit: Use the or operator: **l=**
    b. Clear a specific bit: Use the and operator: **&=**
    c. Toggling a bit: use the XOR operator: **^=**

```
P4OUT ^= GREEN;          // toggle the green LED
P4OUT l= GREEN;          // Green LED is on,
P1OUT &= ~RED;           // Red LED is off
```

d. Assembly:

    e. Set bits in destination: BIC (.B) src, dst ; scr .or .dst->dst
    f. Clear bits in destination: BIC (.B) src, dst ; not.src and dst -> dst
    g. Toggle bits in destination: XOR (.B) src, dst ; src xor dst -> dst

```
xor.b   #0x80, &P4OUT          ; Toggle P4.7
bic.b   #0x01, &P1OUT          ; Turn P1.0 off.
bis.b   #001h, &P1OUT          ; Turn on LED1
```

Mnemonic		Description		V	N	Z	C
ADC(.B)†	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR†	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)†	dst	Clear destination	0 → dst	–	–	–	–
CLRC†		Clear C	0 → C	–	–	–	0
CLRN†		Clear N	0 → N	–	0	–	–
CLRZ†		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst − src	*	*	*	*
DADC(.B)†	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)†	dst	Decrement destination	dst − 1 → dst	*	*	*	*
DECD(.B)†	dst	Double-decrement destination	dst − 2 → dst	*	*	*	*
DINT†		Disable interrupts	0 → GIE	–	–	–	–
EINT†		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)†	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)†	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B)†	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		–	–	–	–
JEQ/JZ	label	Jump if equal/Jump if Z set		–	–	–	–
JGE	label	Jump if greater or equal		–	–	–	–
JL	label	Jump if less		–	–	–	–
JMP	label	Jump	PC + 2 x offset → PC	–	–	–	–
JN	label	Jump if N set		–	–	–	–
JNC/JLO	label	Jump if C not set/Jump if lower		–	–	–	–
JNE/JNZ	label	Jump if not equal/Jump if Z not set		–	–	–	–
MOV(.B)	src,dst	Move source to destination	src → dst	–	–	–	–
NOP†		No operation		–	–	–	–
POP(.B)†	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	–	–	–	–
PUSH(.B)	src	Push source onto stack	SP − 2 → SP, src → @SP	–	–	–	–
RET†		Return from subroutine	@SP → PC, SP + 2 → SP	–	–	–	–
RETI		Return from interrupt		*	*	*	*
RLA(.B)†	dst	Rotate left arithmetically		*	*	*	*
RLC(.B)†	dst	Rotate left through C		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)†	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC†		Set C	1 → C	–	–	–	1
SETN†		Set N	1 → N	–	1	–	–
SETZ†		Set Z	1 → C	–	–	1	–
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		–	–	–	–
SXT	dst	Extend sign		0	*	*	*
TST(.B)†	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

7. **How to work with the switches and LEDs? What values would they return if they are pressed? What about when they are not pressed?**
    a. From the schematic we see that if we want LED1 on, **we should provide a logical '1' at the output port of the microcontroller (port P1.0), and a logical '0' if we want LED1 to be off.** We could take several approaches to solving this problem. Figure 2 illustrates one such approach - after initializing the port P1.0 as output (P1DIR=00000001), setting P1.0 to logic '1', the program will spend all its time in an infinite loop (Figure 2).

b. If it has indeed been pressed (bit 1 of P2IN is 0) – i.e. The button pressed returns 0 and the button not pressed returns 1,=.

8. **How to calculate delays and how they are affected by different components (clock frequency, loop instruction, loop upper and lower limit, etc.)**
   a. delay of 16cc so the total delay is 65535*16cc/2^20 ~= 1s
   b. Worst-case settling time for the DCO when the DCO range bits have been changed is n x 32 x 32 x f_MCLK / f_FLL_reference . See UCS chapter in 5xx UG for optimization. 32 x 32 x 2.45 MHz / 32,768 Hz = 76600 = MCLK cycles for DCO to settle

9. **How would you clear an interrupt flag?**
   a. Switch 1 Press example (C):

```c
// Switch 1 Press
#pragma vector = PORT2_VECTOR
__interrupt void PORT2_ISR(void)
{
    WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
    P1OUT &= ~RED;                 // Turn Red LED off
    P4OUT &= ~GREEN;               // Turn Green LED off
    P2IFG &= ~BIT1;                // Clear interrupt flag.
}
```

   a. SW2_ISR (assembly):

```asm
;-------------------------------------------------------------------------------
; P4_7 (Green) / P1_1 (SW2) interrupt service routine (ISR)
;-------------------------------------------------------------------------------
SW2_ISR:
            bic.b   #0x02, &P1IFG       ; Clear interrupt flag
            bit.b   #00000010b, &P1IN   ; Check if S2 is pressed; (0000_0010 on P1IN)
            jnz     Exit2               ; If not zero, SW is not pressed; loop and check again
            xor.b   #0x80, &P4OUT       ; Toggle P4.7
Debounce:   mov.b   #2000, R7           ; Set to (2000 * 10 cc )
SWD20ms:    dec.w   R7                  ; Decrement R15
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            jnz     SWD20ms                     ; If R7 is 0, then the loop will break and move on.
            bit.b   #0x02, &P1IN        ; Verify S2 is still pressed
            jnz     Exit2               ; If not, wait for S2 press
            mov.b   #1, R7              ; Move 1 into R7
Exit2:      reti                        ; Return from interrupt
```