INFORMATION TECHNOLOGY
SRH HOCHSCHULE, HEIDELBERG

IMPLEMENTATION OF HAIR-DRYER USING MSP430 MICROCONTROLLER

EMBEDDED SYSTEMS

Submitted By:

Rianna Dsilva

Matriculation No: 11010632

Department of Information Technology

PROJECT SPECIFICATION AND WORKING

The TI MSP430 family of low-power microcontrollers, consists of several devices that feature different sets of peripherals targeted for various applications. The architecture, combined with extensive low-power modes, is optimized to achieve extended battery life in portable measurement applications. The device features a powerful 16-bit RISC CPU, 16-bit registers, and constant generators that contribute to maximum code efficiency. The digitally controlled oscillator (DCO) allows the device to wake up from low-power modes to active mode in less than 10 µs.

This project gives an in-detail description of how the MSP430FR4133 is used to implement a Hair-dryer. The MSP430FR4133 Launchpad is initialized to pre-defined conditions and waits for an interrupt ie; for button to be pressed. The Timers, LCD, GPIO are first initialized. Both Button1.2 and Button2.6 are hardware interrupts. The LED remains OFF in the initial state and the LCD does not display anything.

1. Now if Button1.2 is pressed on the MSP, this in-turn turns ON the motor of Hair-dryer to run at Low speed.

Button1.2 pressed -> Mode1

RED LED ON

2. If Button2.6 is pressed on the MSP, this turns ON the motor of Hair-dryer to run at High speed.

Button2.6 pressed -> Mode2

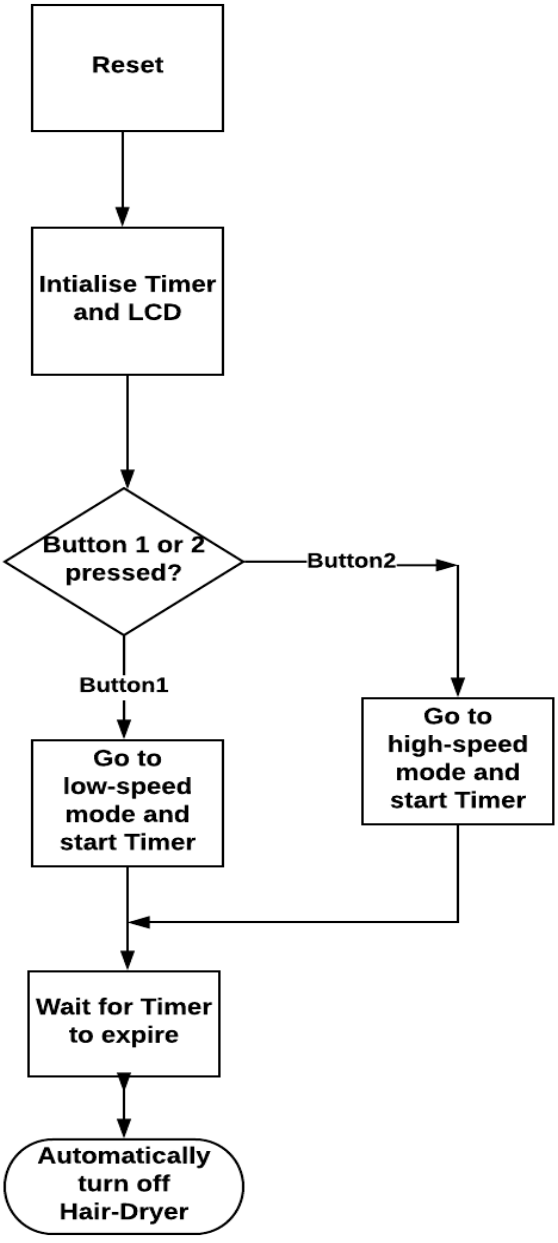
GREEN LED ON

The MSP430 recognizes the button being pressed by servicing the Interrupt service vector[ISR]. Now let us assume a 2-3 seconds timer (in real-time this could be dependent on the user's requirements typically 5-15 minutes), after which the Hair-dryer motor turns OFF automatically. This is indicated using a yellow toggling timer.

Mode1 lets the LCD display: HAIR DRYER ON LOW SPEED

Mode2 lets the LCD display: HAIR DRYER ON HIGH SPEED

IMPLEMENTATION FLOWCHART



WORKING

- The watchdog timer resets the MSP430 in order to erase any pre-existing programs running on the MSP.
- The timers, LCD, GPIO are initialized.
- The programs runs and waits for interrupt to be serviced.
- If no interrupt, the CPU goes to low power mode 3 (LPM3)
- If an interrupt occurs, the CPU services this interrupt. Here, the interrupt is defined as button being pressed.
- If Button1 is pressed, this indicates the motor of hair-dryer to run at low speed. This is indicated by the LCD display.
- If Button2 is pressed, this indicates the motor of hair-dryer to run at high speed. This is indicated by the LCD display.
- Alternatively, the mode of speed can be varied by continuously changing the buttons being pressed.
- If we want to turn off the system, simply stop pressing any of the buttons, the timer then elapses and turns off the system.
- LCD displays this condition as OFF state.

TIMING DIAGRAM OF THE SYSTEM

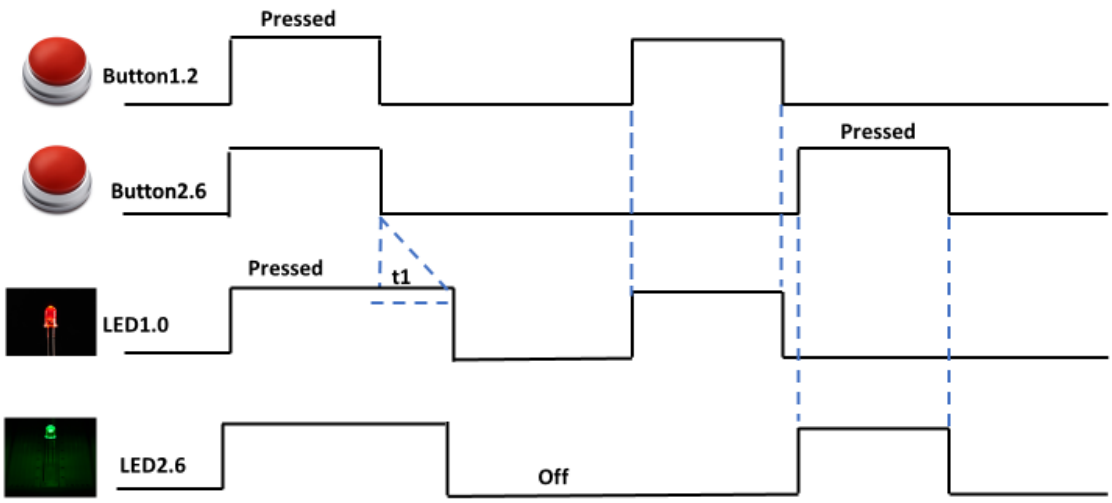
Given below the timing diagram of the working of the designed system. Let us consider various cases of implementation:

Case 1: When only Button1.2 is pressed, this turns on the RED LED of the MSP immediately without any delay and timer starts counting at this point. Once the timer elapses and the overflow occurs, it turns off the LED.

Case 2: When only Button2.6 is pressed, this turns on the GREEN LED of the MSP immediately without any delay and timer starts to count. Once the timer elapses and the overflow occurs, it turns off the LED.

Case 3: When both buttons are pressed, then both LED'S are turned ON. Now, if we press both buttons to turn it off again, then the LED turns off but the LCS first displays LOW SPEED -> OFF -> HIGH SPEED -> OFF. There is a certain delay to turn OFF the motor or hair-dryer when both buttons are pressed at the same time.

This delay here is represented as t1.



INITIAIZING THE MSP

When starting up the MSP430 and running the system, there are several elements that must be initialized.

- The Stack Pointer must be initialized but the system does this automatically.
- Since the Watchdog Timer defaults to "ON", it must be configured. During development and debugging we usually turn it off.
- GPIO pins [general purpose bit I/O]: It is recommended that all GPIO pins on the device are configured. This is done so as to minimize power dissipation.
- Clock options also need to be initialized unless we use the default options set by the system.

GENERAL PURPOSE INPUT-OUTPUT [GPIO] INITILIZATION

The MSP430 provides one or more 8-bit I/O ports. The number of ports is often correlated to the number of pins on the device – more pins, more I/O. This code snippet used in the program initializes P1.0

```
//Initialize GPIO
//Button 1.2

    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );  // Red LED
(LED1)
    GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setAsOutputPin( GPIO_PORT_P4, GPIO_PIN0 );  // Green LED
(LED2)
    GPIO_setOutputLowOnPin( GPIO_PORT_P4, GPIO_PIN0 );
    GPIO_setAsInputPinWithPullUpResistor( GPIO_PORT_P1, GPIO_PIN2 );
    GPIO_selectInterruptEdge ( GPIO_PORT_P1, GPIO_PIN2,
GPIO_LOW_TO_HIGH_TRANSITION );
    GPIO_clearInterrupt ( GPIO_PORT_P1, GPIO_PIN2);
    GPIO_enableInterrupt ( GPIO_PORT_P1, GPIO_PIN2 );
```

IMPLEMENTING TIMERS

What are Timers?

A **timer** is a specialized type of clock which is used to measure time intervals. A timer that counts from zero upwards for measuring time elapsed is often called a **stopwatch**. It is a device that counts down from a specified time interval and used to generate a time delay.

A **counter** is a device that stores (and sometimes displays) the number of times a particular event or process occurs, with respect to a clock signal. It is used to count the events happening outside the microcontroller.

Timers in FR4133: The FR4133 family contains 4 timers given below:

1. Watchdog Timer
2. Timer A
3. Timer B
4. Real Time clock

Implementation of Timer:

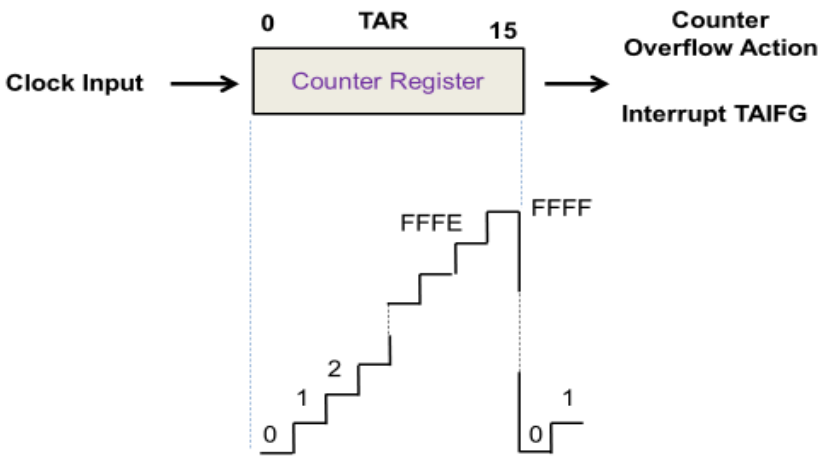


Figure: Working of a 16-bit timer/counter

A counter is the fundamental hardware element found inside a timer. The other essential element is a clock input. The counter is incremented each time a clock pulse is applied to its clock input. Therefore, a 16-bit timer will count from zero (0x0000) up to 64K (0xFFFF).

When the counter reaches it reaches its maximum value, it overflows – that is, it returns to zero and starts counting upward again. Most timer peripherals can generate an interrupt when this overflow event occurs; on TIMER_A, the interrupt flag bit for this event is called TAIFG (TIMER_A Interrupt Flag).

For our system, we require the usage of two timers.

1. Watchdog Timer:

Watchdog Timers provide a system failsafe. If the counter ever rolls over (back to zero) this resets the processor.

By default, the MSP430 always enables the watchdog timer at reset. Hence it is very important to disable this timer, in order to prevent it from continually re-setting the system.

The code snippet given below is used in the program to prevent the system from being reset.

```
WDT_A_hold(WDT_A_BASE);
```

2. Timer_A:

Timer_A is a 16-bit timer/counter. It has multiple Capture-Compare registers. It can generate PWM and other complex waveforms. It can also directly trigger the GPIO, DMA, ADC etc. We can configure the timer as per our requirements. Here, Timer_A runs in Up mode and we use a slower clock ACLK. The Timer_A counts from 0000h to CCR0 and then resets. The TACCR0 CCIFG flag interrupt flag is set when the timer counts to the TACCR0 value. The TAIFG interrupt flag is set when the timer counts from TACCR0 to zero.

Why UP Mode? Up mode allows better control the timer's frequency – that is, we can control the time period for when the counter resets back to zero.

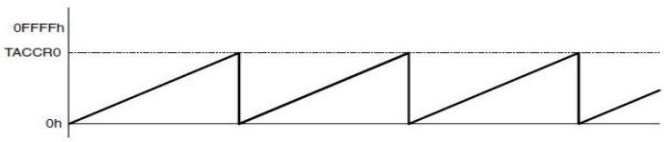


Figure: Timer_A in UP Mode

Timer Calculation:

We can set the timer's time period simply by changing the Divider to a lower or higher value. For example, ACLK = 32KHz

Time period = FFFFh * 1/32k = 2.0479 seconds.

Capture-Compare feature of Timer_A:

Capture: When a capture input signal occurs, a snapshot of the Counter Register is captured; that is, it is copied into a capture register (CCR for Capture and Compare Register). This is ideal since we get the timer counter value captured with no latency and very little power used (timer remains in low-power mode).

Compare: In this mode, whenever a match between the Counter and Compare occurs, a compare action is triggered. The compare actions include generating an interrupt, signaling another peripheral (e.g. triggering an ADC conversion), or changing the state of an external pin. This can also be used to generate PWM signals.

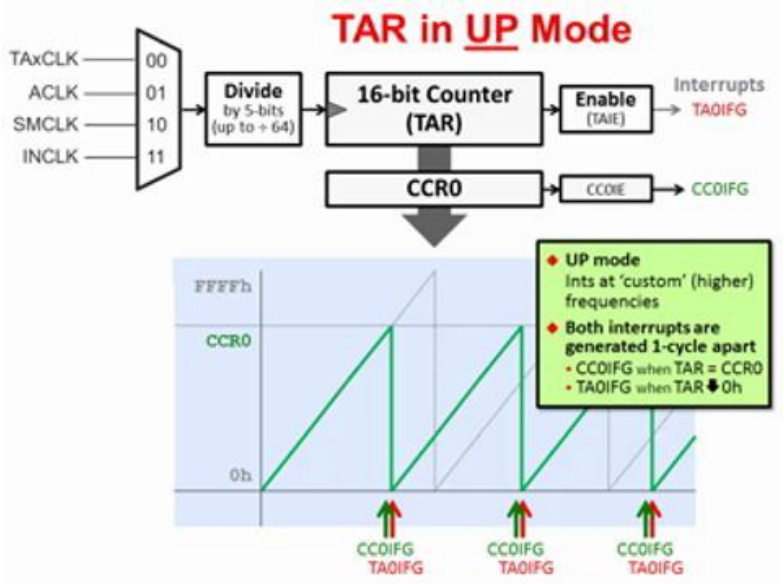


Figure reference: https://training.ti.com/msp430-workshop-series-6-12-timers

The code snippet below gives the timer initialization used in the program.

```
        Timer_A_initUpModeParam initUpParam = { 0 };
        initUpParam.clockSource =                   TIMER_A_CLOCKSOURCE_ACLK;       //
Use ACLK (slower clock)
        initUpParam.clockSourceDivider =            TIMER_A_CLOCKSOURCE_DIVIDER_1;  //
Input clock = ACLK / 3 = 32KHz
        initUpParam.timerPeriod =                   0xFFFF;                         //
Half the time
        initUpParam.timerInterruptEnable_TAIE =     TIMER_A_TAIE_INTERRUPT_ENABLE;  //
Enable TAR -> 0 interrupt
        initUpParam.captureCompareInterruptEnable_CCR0_CCIE =
TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE; //Enable compare interrupt
        initUpParam.timerClear =                    TIMER_A_DO_CLEAR;               //
Clear TAR & clock divider
        initUpParam.startTimer =                    false;                          //
Don't start the timer, yet

        Timer_A_initUpMode( TIMER_A1_BASE, &initUpParam );
        Timer_A_clearTimerInterrupt( TIMER_A1_BASE );
        Timer_A_clearCaptureCompareInterrupt(TIMER_A1_BASE,
TIMER_A_CAPTURECOMPARE_REGISTER_0); //Clear CCROIFG

        Timer_A_startCounter(
                TIMER_A1_BASE,
                TIMER_A_UP_MODE
```

It is also necessary to service the CCRO timer interrupts. Here, the timer ISR is given in the below snippet. We check the working on the timers, by toggling a yellow LED to indicate when the timer starts or overflows.

```
#pragma vector=TIMER1_A0_VECTOR
__interrupt void CCR0_ISR (void)
{
   GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN5 );
// Toggle EXTERNAL (YELLOW) LED on/off
    mytime1 = mytime1 ^ 1;
}
```

MSP430 CLOCKS

SMCLK is an internal clock generally taken from DCO (Digitally Controlled Oscillator) and is in the Megahertz range. The DCO clock is temperature dependent and may change depending upon temperature.

ACLK is an internal clock taken either from VLO or an external 32KHz clock crystal connected to the MSP430.**ACLK** when taken from an external clock **crystal is usually stable** and does not change with temperature. The ACLK frequency is usually in Kilo hertz range (32KHz).

TACLK and INCLK are external clocks that can be connected to MSP430 through external pins.

TASSELx bits are used to select the required clock to the 16-bit Timer register and **IDx** bits are used to select clock divider which are present in the **TACTL** register. The 16 bit Timer A register(**TAR**) starts counting on the positive edge of the clock. TAR can be cleared by setting the **TACLR** bit to 1.

OTHER PROGRAM FUNCTIONS

Other important aspects to be considered to efficiently run the program are:

1. LCD Initialization and linking the LCD.c and LCD.h files in the project folder.

```
LCD_init();
LCD_E_clearMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12,
LCD_HEART);
```

2. While loop: The while loop runs indefinitely, unless an interrupt needs to be serviced.

```
while(1){
        if(mode1) {
                displayScrollText("HAIR DRYER ON LOW SPEED");
        if(mytime1)
                   displayScrollText("HAIR DRYER OFF");
        }
        if(mode2){
                displayScrollText("HAIR DRYER ON HIGH SPEED");
        if(mytime1)
                      displayScrollText("HAIR DRYER OFF");
           }
      }
  return 0;
```

IMPLEMENTATING INTERRUPTS

There are two methods by which events can be recognized by the processor. One is called “Polling” and the other, “Interrupts”. Polling uses 100% CPU load whereas Interrupts makes use of >0.1% CPU load. Since the goal is to execute this program is low power mode, we make use of interrupts.

To execute interrupts, we require;

- The #pragma sets up the interrupt vector.
- The __interrupt keyword tells the compiler to code this function as an interrupt service routine (ISR).
- To enable Interrupts

From the program we notice that, it ismain() thread that begins running once the processor has been started. The compiler’s initialization routine calls main() when its work is done. With the main() thread started, since it is coded with a while(1) loop, it will keep running forever. That is, unless a hardware interrupt occurs. When an enabled interrupt is received by the CPU, it preempts the main() thread and runs the associated ISR routine – for example, ISR1. In other words, the CPU stops running main() temporarily and runs ISR1; when ISR1 completes execution, the CPU goes back to running main().

The key DriverLib function which enables the external interrupt is: GPIO_enableInterrupt()

Also, the following function is used to enable interrupts globally in the program. __bis_SR_register(GIE);

If we choose to run the code in LPM, it is recommended to switch to:

```
__bis_SR_register(GIE + LPM3_bits );

__no_operation();

//exit(LPM3)
```

The snippet below used in our program explains using an external hardware interrupt such as pushing button Pin 1.2 to perform a certain function. Here the Interrupt function asks the program to turn on the RED LED at Pin 1.0 and to display a text on the LCD screen. This defined in the while() loop.

There are two items in the program which help the compiler to produce better, more optimized, code. While these intrinsic functions are not specific to interrupt processing, they are useful in creating optimized ISR's.

• The __even_in_range() intrinsic function provides the compiler a bounded range to evaluate. In other words, this function tells the compiler to only worry about even results that are lower or equal to 10.

• Likewise the _never_executed() intrinsic tells the compiler that, in this case, "default" will never occur.

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
    switch( __even_in_range( P1IV, P1IV_P1IFG7 )) {
    case P1IV_NONE:   break;// None
    case P1IV_P1IFG0:       // Pin 0
        __no_operation();
        break;
    case P1IV_P1IFG1:       // Pin 1
        __no_operation();
        break;
    case P1IV_P1IFG2:       // Pin 2 (button 1)
        flag1 = flag1 ^ 1;
        GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        break;
    case P1IV_P1IFG3:       // Pin 3
        __no_operation();
        break;
    case P1IV_P1IFG4:       // Pin 4
        __no_operation();
        break;
    case P1IV_P1IFG5:       // Pin 5
        __no_operation();
        break;
    case P1IV_P1IFG6:       // Pin 6
        __no_operation();
        break;
    case P1IV_P1IFG7:       // Pin 7
        __no_operation();
        break;
    default:   _never_executed();
    }
}
```

PULSE WIDTH MODULATION

PWM, or pulse-width modulation, is commonly used to control the amount of energy going into a system. For example, by making the pulse widths longer, more energy is supplied to the system. In the case of the MSP430, any timer can generate a PWM waveform by configuring the CCR registers appropriately. We can set PWM_PERIOD to a value of our choosing.

```
P1DIR |= BIT0;
P1SEL0 |= BIT0;
TA0CCR0 = PWM_PERIOD-1; //PWM_PERIOD = any value
TA0CCR1 = PWM_PERIOD-1;
TA0CCTL1 = OUTMOD_4;
TA0CTL = TASSEL_2 + MC_1;
```

The duty cycle (‘on’ time) can be set by selecting the Output Mode and varying the CCRx value. Given in the below figure the various Output Modes that can be set;

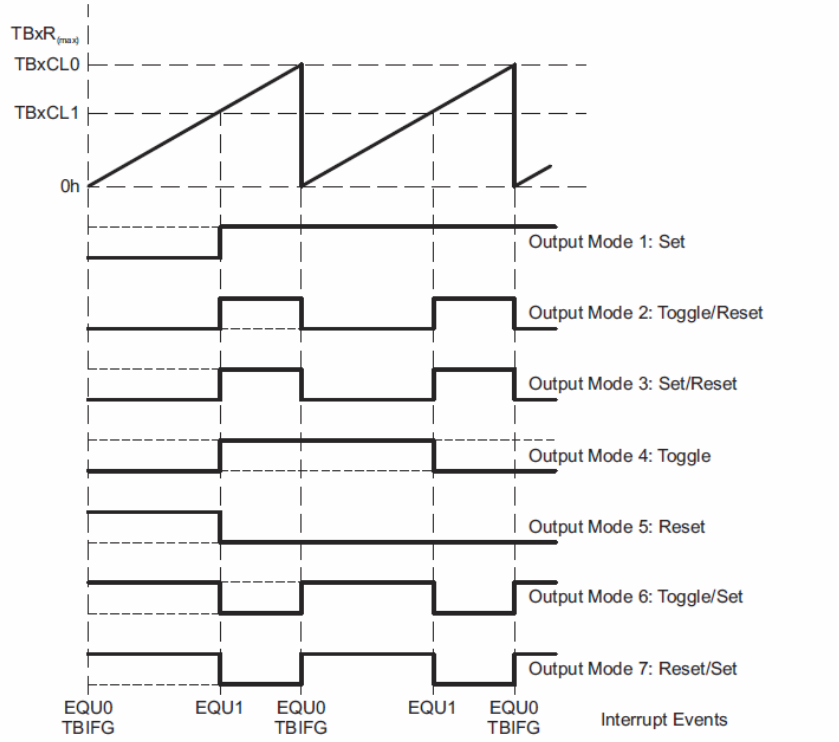


Figure reference: TI MSP430 training document

RUNNING THE DEVICE IN LOW POWER MODE

The MSP430 supports the sleep/wake/sleep profile by providing a variety of low-power modes (LPM). The MSP430 CPU can reside in: Reset, Active, or one of many Low-Power Modes (LPM).

Entering Low Power Mode 3:

LPM3 **_low_power_mode_3();** or **_bis_SR_register(GIE + LPM3_bits);**

This function enables the interrupts globally as well as the LMP3 by setting the Status Register intrinsically (bis: bit set & bic: bit clear).

Exiting Low Power Mode 3: This automatically takes the program flow back to low power mode.

_low_power_mode_off_on_exit;

MSP430 is inherently low-power, but design has a big impact on power efficiency. Given below are some methods to run a device in Ultra low power mode:

- Use interrupts to control program flow
- Maximize the time in LPM3
- Replace software peripherals
- Configure unused pins
- Power manage external devices
- Write efficient codes

In order to run our device in low power, the program has implemented a couple of functions such as the usage of a slower clock [ACLK]. The low power clock and peripherals interrupt CPU only for processing. The unused pins are configured either by setting an input using the pull up/down resistor or are driven to Vcc or ground.

It is important to note that power dissipation increases with CPU clock speed (MCLK), Input voltage and temperature.

There are two main methods from the TI'S MSP430 & CCS to convert your device to a LPM Application:

1. ULP Advisor
2. Energy Trace

ULP ADVISOR:

Using the ULP Advisor in Code Composer Studio[CCS]:

1. The ULP Advisor analyses all the MSP430 C code line-by-line.
2. It checks against a thorough ULP checklist
3. Highlights area of improvement within code.

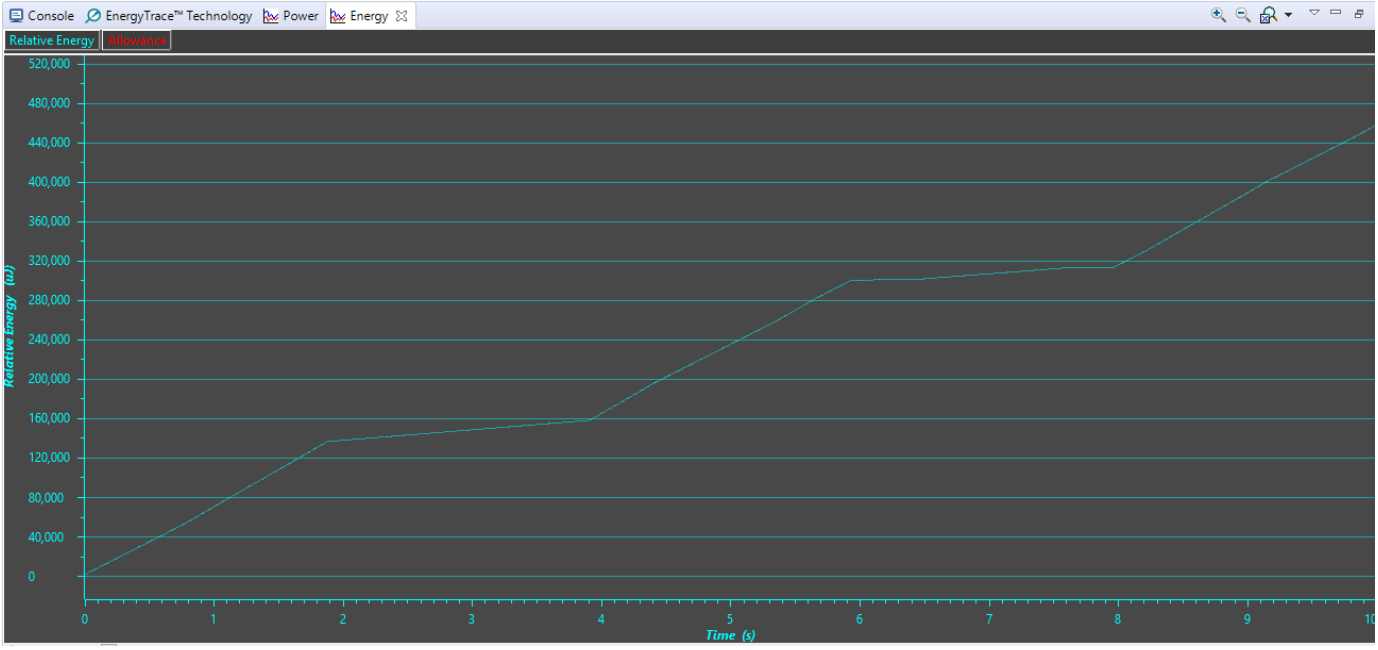
System Performance Comparison:

System Parameters	Enabling LMP3	Without Enabling LMP3
Time	10 seconds	10 seconds
Energy	287.769 mJ	456.102 mJ
Minimum – Maximum Power	0 mW – 75.8119 mW	0 mW – 75.0279 mW
Mean Power	28.8734 mW	45.6940 mW
Mean Voltage	3.2838 V	3.2834 V
Minimum - Maximum Current	0 mA – 23.0009 mA	0 mA – 22.8674 mA
Mean Current	8.7972 mA	13.6582 mA

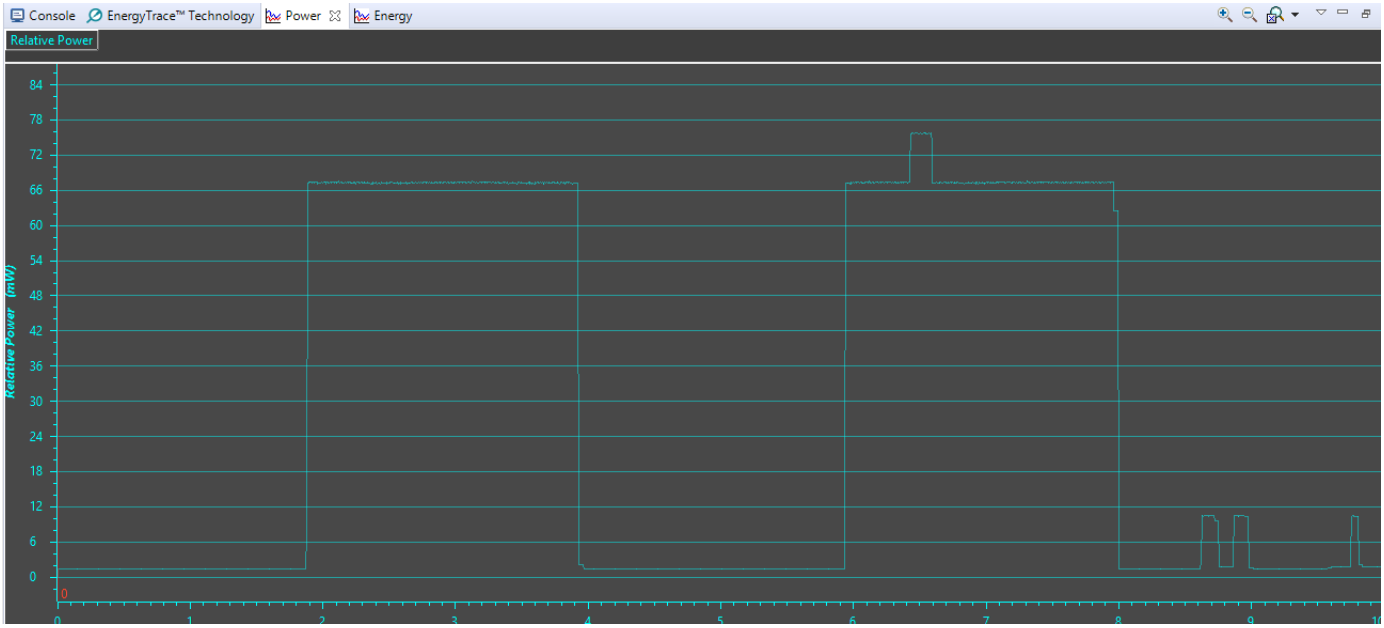
Comment: The ULP Advisor sends a warning message that all the unused pins must be initialized in order to eliminate power wastage. If not initialized, these unused pins have a floating-point value. This can be considered as future improvements of the designed system.

ENERGY TRACE – WITH LMP3

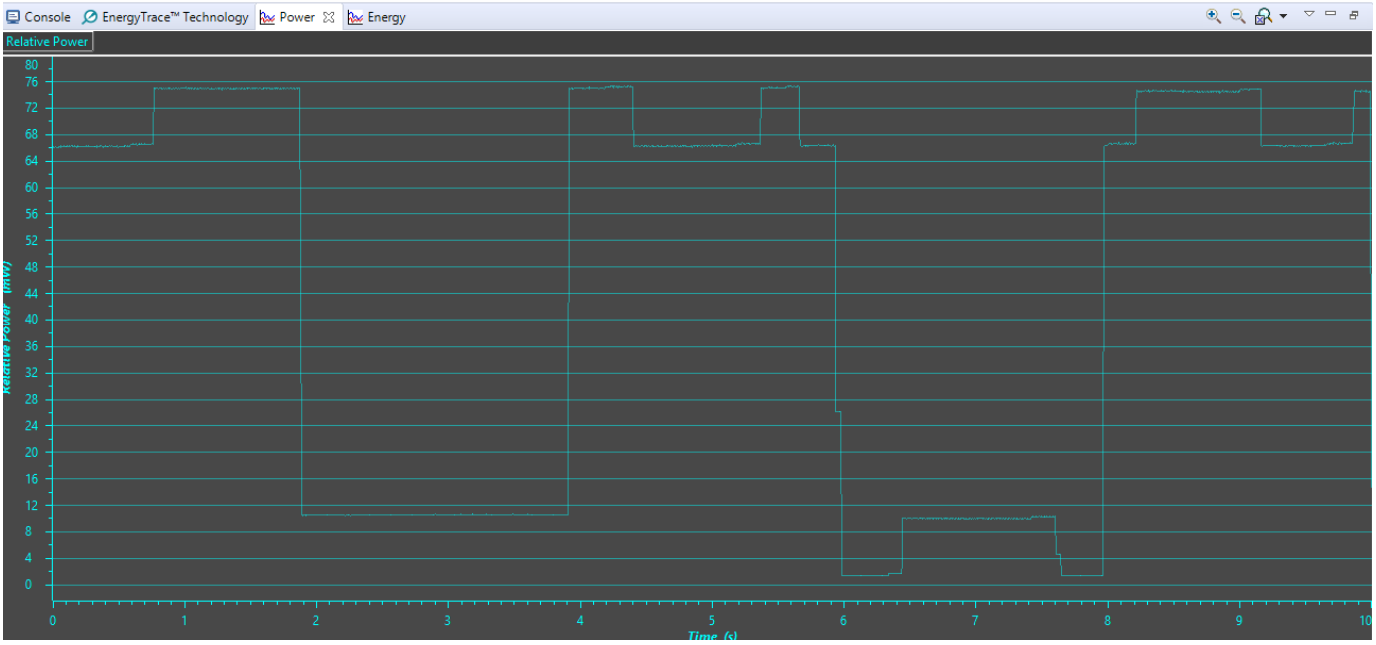
ENERGY TRACE – WITHOUT LMP3



RELATIVE POWER TRACE - LMP3



RELATIVE POWER TRACE – WITHOUT LMP3



CONCLUSION AND FUTURE WORK

This report explains how the MSP430FR4133 is used to design a simple, robust and efficient hair-dryer. It can be modified by making minute changes for other applications such as tachometer, dc motors, electric heaters and so on. The system makes use of interrupts, timers, clocks, LCD display, breadboard, LED'S, and the TI FR4133 Launchpad.

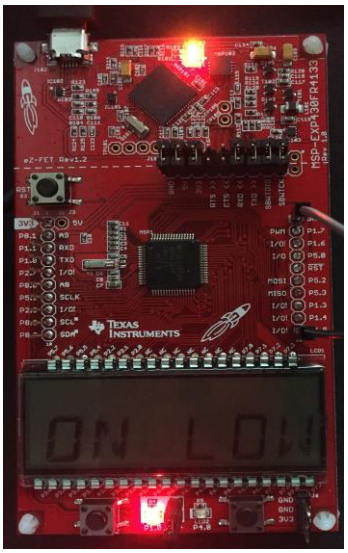
The system has been implemented in Low power mode to calculate its efficiency and measure the power dissipation. Pulse width modulation has also been implemented in this system. The Energy trace was performed multiple times to calculate variations and the average power dissipated by the system has been measured.

Some challenges during the implementation of the system are;

- Configuring the system to implement Pulse width modulation
- Optimizing the code to match low power requirements
- Implementing the right timers modes (up, continuous, up-down) and deciding which clock is suitable.

The program was successfully compiled and linked. The resulting 163 bytes of executable code was downloaded into RAM memory of MSP430 device using Code Composer Studio [CCS]. Flash/FRAM usage is 3050 bytes.

The program was successfully executed. The system was tested under multiple conditions of clocks and time-periods and the LED glow and LCD display functioned as required.



INSIDE A HAIR-DRYER - TRIVIA

A hair dryer needs only two parts to generate the blast of hot air that dries your hair:

1. Simple motor-driven fan
2. Heating element

Hair dryers use the motor-driven fan and the heating element to transform electric energy into convective heat.

The whole mechanism is really simple:

1. When you plug in the hair dryer and turn the switch to "on," current flows through the hair dryer.
2. The circuit first supplies power to the heating element. In most hair dryers, this is a bare, coiled wire, but in models that are more expensive there can be fancier materials in action, like a tourmaline-infused ceramic coating.
3. The current then makes the small electric motor spin, which turns the fan.
4. The airflow generated by the fan is directed down the barrel of the hairdryer, over and through the heating element.
5. As the air flows over and through the heated element, the generated heat warms the air by forced convection.
6. The hot air streams out the end of the barrel.

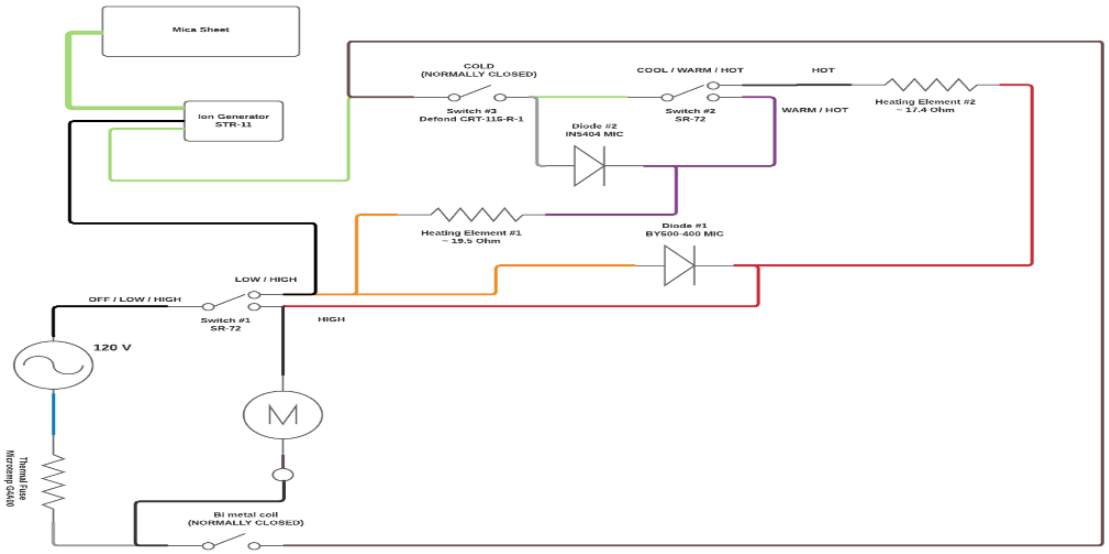


Fig: Circuit Diagram of Hair Dryer

REFERENCES

1. Texas Instruments Incorporated, “MSP430x2xx Family Data Sheet.” [Online document]
   HTTP: http://focus.ti.com/lit/ds/symlink/msp430f2012.pdf
   Cited: 2018 May 5th – Available

2. https://home.howstuffworks.com/hair-dryer1.htm
   Cited: 2018 May 20 – Available

3. HTTP: https://training.ti.com/msp430-workshop-series-6-12-timers
   Cited: 2018 May 12 – Available

4. MSP430 Microcontroller Basics – John Davies
   ISBN: 978-0-7506-8276-3

5. Texas Instruments Incorporated, “MSP430 Peripheral Driver Library” [Online Document]
   Cited: 2018 June 2nd – Available

6. Embedded Systems Design using the TI MSP430 Series – Chris Nagy
   ISBN: 0 7506 7623 X

7. P. Gaspar, A. Santo, B. Ribero - MSP430 microcontrollers essentials - A new approach for the embedded systems courses: Part 3 - Data acquisition and communications
   Education and Research Conference (EDERC), 2010 4th European Conference

8. S. Belgaonkar, E. Elavarasi, G. Singh - Smart Lighting and Control using MSP430 & Power Line Communication
   International Journal Of Computational Engineering Research, ISSN: 2250–3005.

9. M. Rosenblatt and H. Choset, "Designing and implementing hands-on robotics labs," Intelligent Systems and Their Applications, IEEE, 15(6): Nov.-Dec. 2000.