

Laboratório 1 – Modos de Funcionamento de Baixo Consumo

Os microcontroladores MSP430 foram projetados especificamente para aplicações de ultrabaixo consumo. Neste laboratório vamos avaliar os diversos modos de funcionamento dos microcontroladores da família MSP430 no modo normal e nos modos de baixo consumo. Em geral, a família MSP430 possui 6 modos de funcionamento: o modo de funcionamento ativo (AM) e 5 modos de baixo consumo (LPM0 – LPM4). Cada um dos modos de funcionamento caracteriza-se pelos diversos sinais de relógio disponíveis e ativos, que são, depois, utilizados para sincronizar os diversos módulos periféricos.

Introdução

Os microcontroladores MSP430 foram concebidos para terem o mais baixo consumo possível. Deste modo, os MSP430 estão entre os microcontroladores do mercado com o mais baixo consumo, sendo particularmente adequados para serem alimentados a partir de uma pilha ou bateria. Dado que uma bateria fornece corrente, daí a sua capacidade ser medida em Ampére hora (Ah) (normalmente em mAh), mantendo a sua tensão de saída constante, pelo menos até começar a ficar descarregada, o consumo destes dispositivos avalia-se medindo a corrente elétrica que consomem em funcionamento normal. Quanto mais baixa a corrente de funcionamento, menor o seu consumo, e mais tempo demorará a bateria que alimenta o sistema a ficar descarregada.

De modo a consumir o mínimo de corrente elétrica, e a poupar bateria, estes tipos de dispositivos possuem diversos modos de funcionamento, devendo passar o máximo tempo no modo funcionamento com o menor consumo. A escolha do modo de funcionamento depende dos módulos que precisam de estar ativos durante a operação do sistema. A estratégia passa por colocar o sistema no modo de funcionamento ativo sempre que é necessário o CPU realizar algum tipo de processamento, e passar logo que possível para o modo mais baixo de consumo compatível com os módulos periféricos que precisam de ficar ativos.

O que distingue os modos de funcionamento são os diversos sinais de relógio disponíveis para alimentar cada um dos módulos periféricos.

Nas secções seguintes vamos analisar cada um dos modos de funcionamento e medir o consumo de corrente de cada um deles, começando por fazer uma introdução ao módulo de relógio do MSP430G2 designado por Basic Clock Module+ (BCM+), ou módulo de relógio básico melhorado.

Módulo de Relógio do MSP430

Os sinais de relógio presentes no MSP430 são o MCLK (Master clock) que é usado pelo CPU, o SMCLK (Sub-main clock) que é selecionado por software para ser usado pelos módulos

periféricos, e ainda o ACLK (Auxiliary clock) que é igualmente selecionado por software e pode ser usado pelos módulos periféricos.

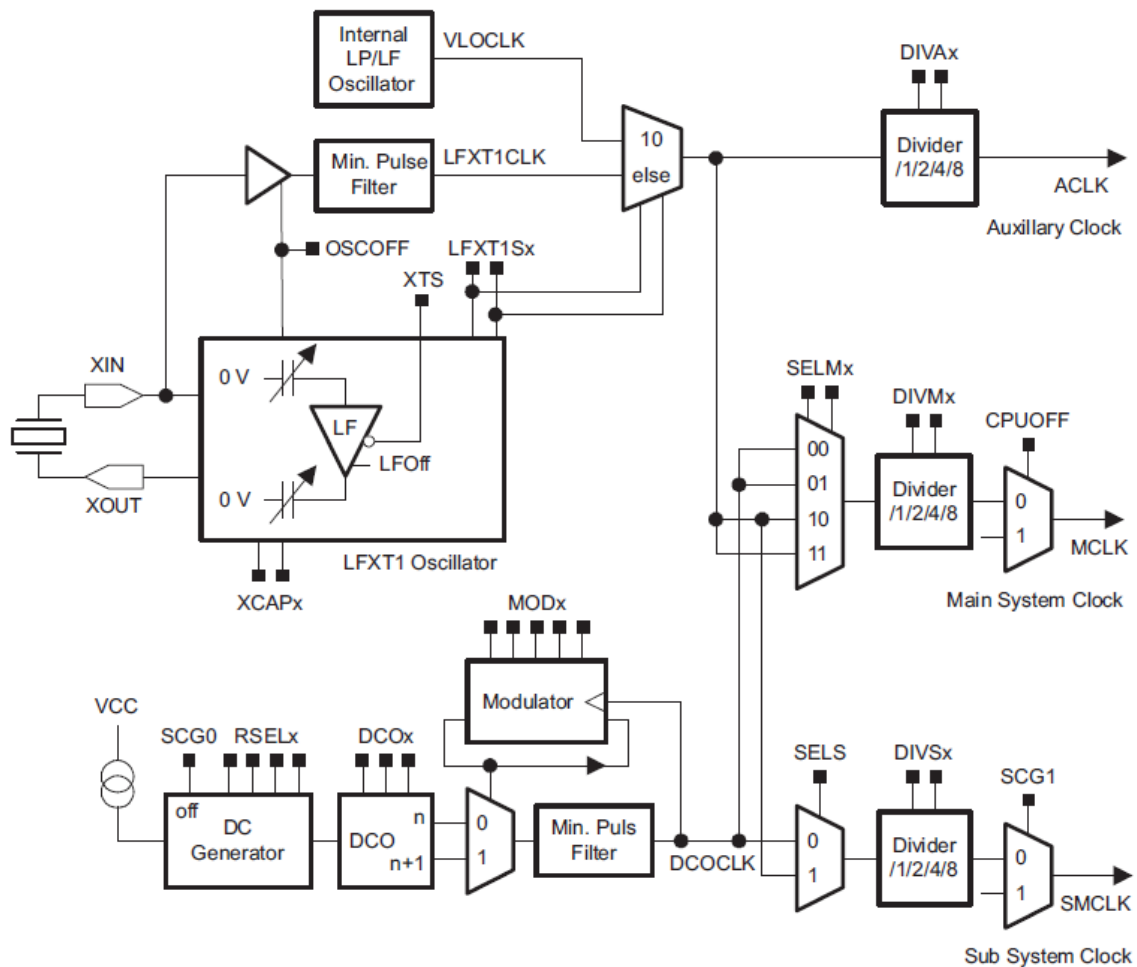


Figura: Sistema de relógio básico, BCS+, do MSP430 (adaptado de [slau144j]).

Cada um destes sinais de relógio pode ser obtido a partir de diversos osciladores que geram um sinal do tipo oscilante, frequentemente com uma forma de onda sinusoidal ou quadrada. Os sinais de relógio podem ser gerados a partir dos seguintes módulos osciladores:

- **LFXT1CLK - Low-frequency/high-frequency crystal oscillator**, ou seja, a partir de um cristal oscilador de baixa ou, em alternativa, de alta frequência (o tipo de cristal de baixa/alta frequência é configurado à parte). Para o cristal de baixa frequência usa-se normalmente um cristal de relógio com uma frequência de 32768 Hz. A frequência deste cristal pode andar entre os 400 kHz e os 16 MHz;
- **XT2CLK – High-frequency crystal oscillator (Optional)**, alguns membros da família MSP430 possuem ainda como gerador opcional de sinal de relógio XT2CLK, que pode usar como oscilador um cristal, um ressonador ou ainda um sinal de relógio externo com frequência entre 400 Hz e 16 MHz;
- **DCOCLK – Digitally controlled oscillator**, trata-se de um oscilador interno, baseado num circuito eletrónico, cuja frequência é controlada digitalmente;

- VLOCLK - *Internal very low power, low frequency oscillator*, tratando-se, como o próprio nome indica, de um oscilador de muito baixo consumo e de baixa frequência, tipicamente de cerca de 12 kHz, que pode ser usado em aplicações onde se pretende um consumo energético mínimo e onde não seja necessária uma grande precisão para o valor da frequência.

Os sinais de relógio MCLK, SMCLK e ACLK não podem ser obtidos arbitrariamente a partir de qualquer fonte geradora de sinal: LFXT1CLK, XT2CLK, DCOCLK e VLOCLK (não confundir os sinais de relógio com as fontes geradoras de sinais de relógio ou osciladores!) A tabela seguinte mostra a correspondência entre os diversos sinais de relógio e as fontes geradoras alternativas que podem ser usadas para produzir os diversos sinais de relógio.

Sinal de Relógio	Gerador do sinal de relógio				Utilização
	LFXT1CLK	VLOCLK	XT2CLK	DCOCLK	
MCLK	Sim	Sim	Sim	Sim*	CPU/Sistema
SMCLK	Sim	Sim	Sim	Sim*	Módulos periféricos
ACLK	Sim*	Sim			Módulos periféricos

Tabela: Correspondência entre cada um dos sinais de relógio presentes no MSP430 e a origem da oscilação utilizada para a sua geração. O campo assinalado com * corresponde à fonte osciladora usada por defeito para gerar o sinal de relógio.

Nem todos os MSP430 têm disponível o cristal XT2CLK como gerador de sinal de relógio, como é o caso do MSP430G2553.

1. Modo de Funcionamento Ativo

No modo de funcionamento ativo todos os sinais de relógio do microcontrolador, que são fornecidos ao CPU e aos periféricos, estão ativos, nomeadamente o MCLK, o SMCLK e o ACLK.

Para testar o modo de funcionamento ativo vamos colocar o microcontrolador a piscar um LED continuamente, usando para isso um atraso (*delay*) gerado a partir de um ciclo `for`, onde o microcontrolador está exclusivamente a incrementar uma variável de controlo até esta atingir um valor limite, comutando depois o estado do LED e recomeçando a contagem. Este programa é simplesmente:

```
#include <msp430.h>

/**
 * main.c
 */
void main(void){

    WDTCTL = (WDTPW | WDTHOLD); // stop watchdog timer

    P1DIR |= (BIT0);
```

```

int i;
while (1) {
    for (i = 0; i < 15000; i++); // empty cycle -> just a delay

    P1OUT ^= (BIT0); // toggle LED bit
}
}

```

Listagem: Piscar um LED no Modo Ativo (AM) contínuo.

A forma usual de obter o consumo de uma aplicação baseada em microcontrolador é medir a queda de tensão numa resistência de monitorização, cujo valor é conhecido, que é colocada em série com sistema, e de onde se infere a corrente através da lei de Ohm.

A figura abaixo ilustra uma possível montagem para medir a corrente consumida pelo microcontrolador recorrendo a um osciloscópio:

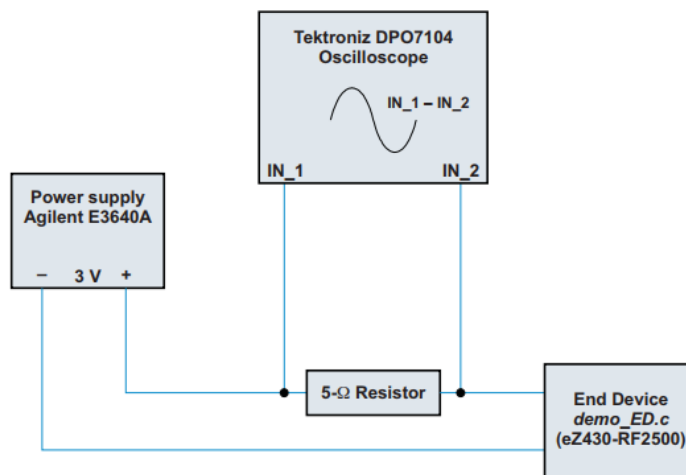


Figura: Diagrama de blocos do sistema usado na medição da corrente consumida por um dispositivo [slaa378d].

Medição do Consumo

Para medir o consumo de potência, energia e corrente podemos usar uma placa de desenvolvimento que suporte a tecnologia EnergyTrace. Nem todas as placas de desenvolvimento da Texas Instruments, possuem esta tecnologia, como é o caso da placa LaunchPad MSP-EXP430G2. No entanto podemos programar e medir o seu consumo usando outra placa de desenvolvimento, como é o exemplo da placa MSP-EXP430FR5969, que possua a EnergyTrace Technology [slaa603].

Vamos seguir o procedimento descrito no documento [slaa603] que usa a placa LaunchPad MSP-EXP430FR5969. A tecnologia EnergyTrace, da Texas Instruments, possui diversas versões: EnergyTrace, EnergyTrace+, EnergyTrace++, todas disponíveis na placa MSP-EXP430FR5969. No entanto, para a sua utilização em conjunto com a placa MSP-EXP430G2 só podemos usar a tecnologia EnergyTrace com menos opções, mas é suficiente para o pretendido.

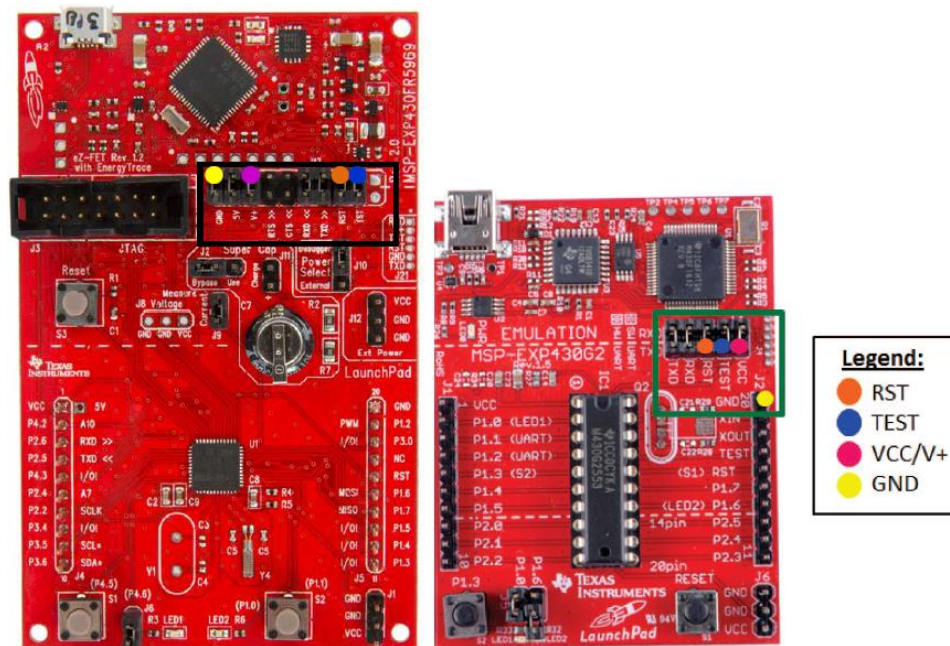


Figura: Conexões entre as placas MSP-EXP430FR5969 (à esquerda) e MSP-EXP430G2 (à direita) para usar o EnergyTrace [slaa603].

Para medir os consumos do MSP430G2553 a executar o programa, seguir os seguintes passos:

1. Ativar a ferramenta EnergyTrace seguindo: **View->Other->EnergyTrace** e seleccionar **EnergyTrace Technology**:

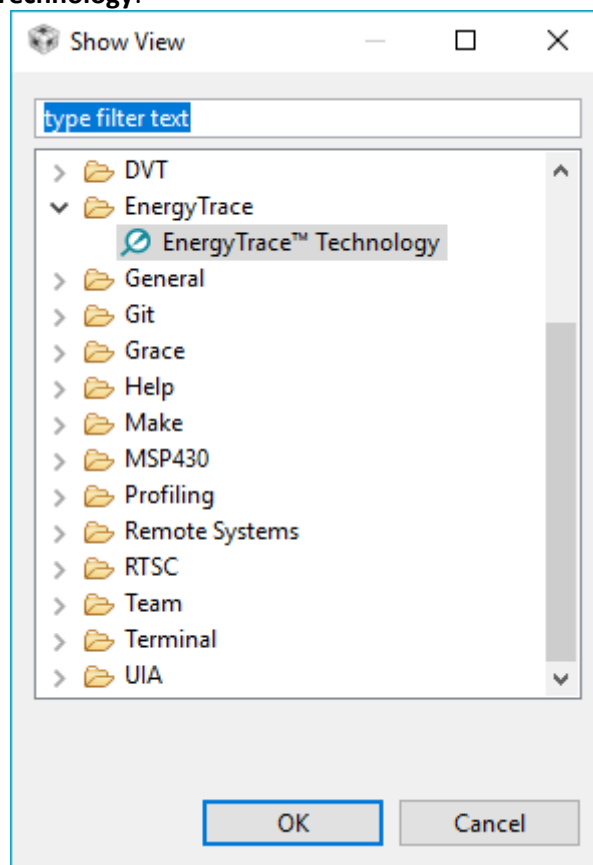




Figura: Ativar a ferramenta *EnergyTrace Technology* em **View->Other->EnergyTrace**.

2. Entrar no modo *Debug* do projeto carregando com o botão direito do rato no projeto e seleccionando *Debug As, CCS Debug Session*. Alternativamente clicar no icon de *Debug*: .
3. No caso de não se pretender obter o perfil de consumo de todo o programa podemos colocar um *breakpoint* na linha do programa a partir da qual se pretende estudar o perfil de consumo, normalmente depois da inicialização do software e do hardware e antes do ciclo principal da função **main**.
4. Com a execução do código em pausa, ir ao separador do *EnergyTrace* e ajustar o tempo durante o qual se pretende obter o perfil de consumo usando o menu *Set measurement duration*. Por defeito o tempo é de 10 segundos:



5. Executar o código e aguardar que o tempo de captura passe clicando no icon  do menu *Debug*. As janelas da *EnergyTrace Technology* são atualizadas com os dados em tempo-real.
6. A janela de *EnergyTrace Technology* mostra o tempo de execução considerado e os valores médio, máximo e mínimo para a potência, tensão e corrente consumidos.

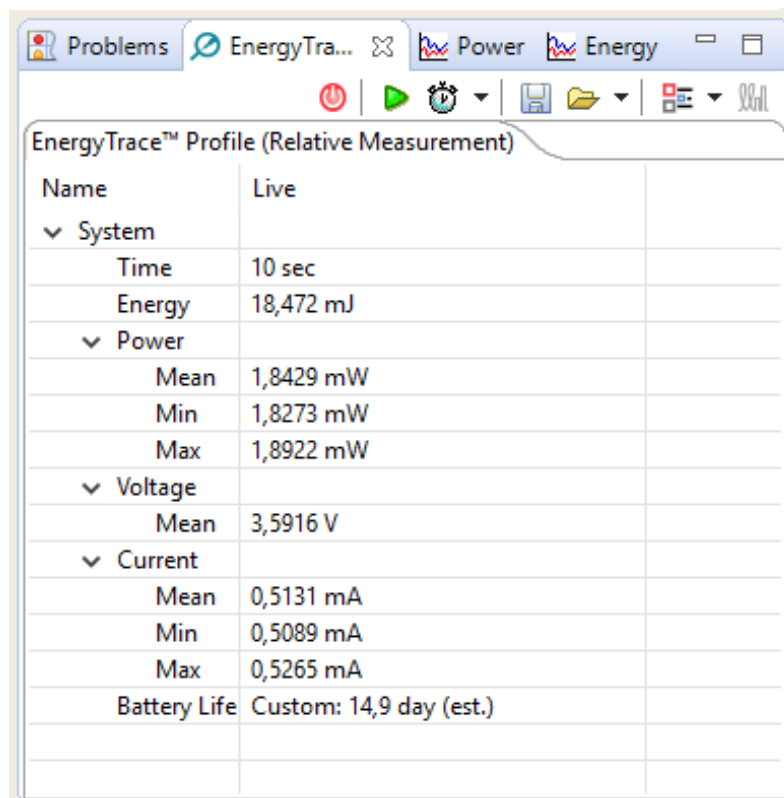


Figura: Perfil de consumo durante 10s com o *jumper* P1.0 em J5 removido.

Clicando nos separadores de *Power* e *Energy*, a ferramenta mostra um gráfico com a evolução temporal da potência consumida e da energia despendida, respetivamente.

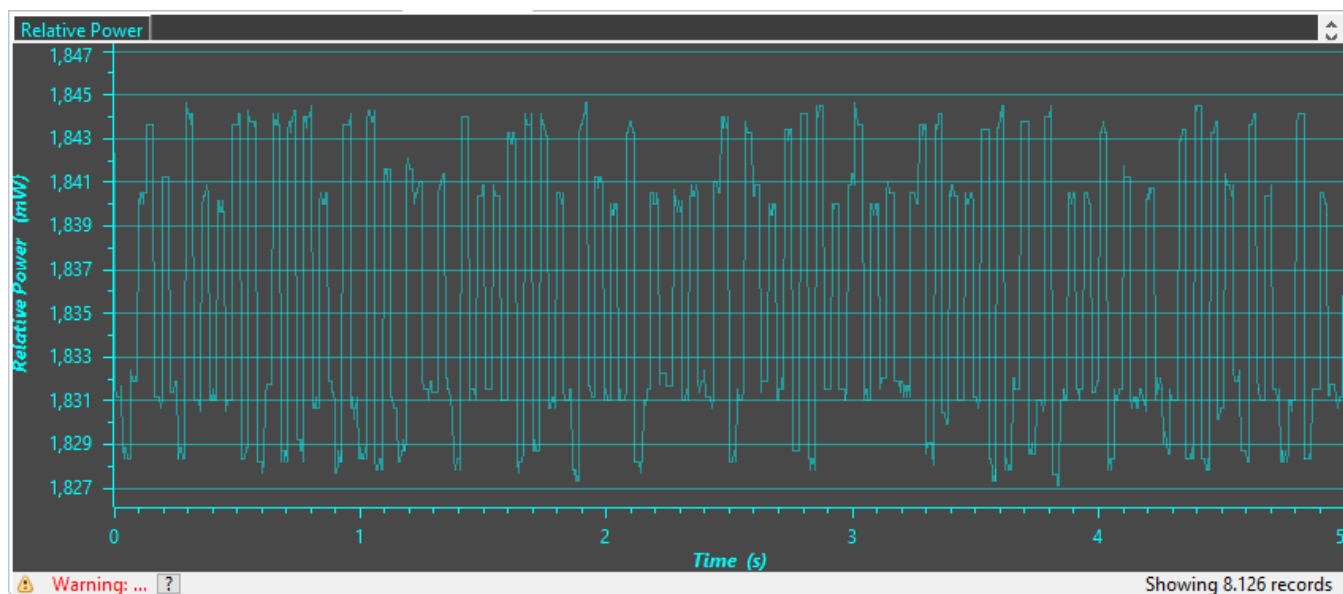


Figura: Perfil de consumo de potência durante 5s com o *jumper* de P1.0 em J5 removido.

A partir do gráfico, e dos dados acima, vemos que o consumo de potência oscila entre 1,827mW e 1,845mW. O consumo de potência oscila entre um intervalo relativamente pequeno, porque não se está a tirar partido dos modos de baixo consumo, pelo que a energia é consumida a um ritmo constante, conforme mostra o gráfico da mesma:

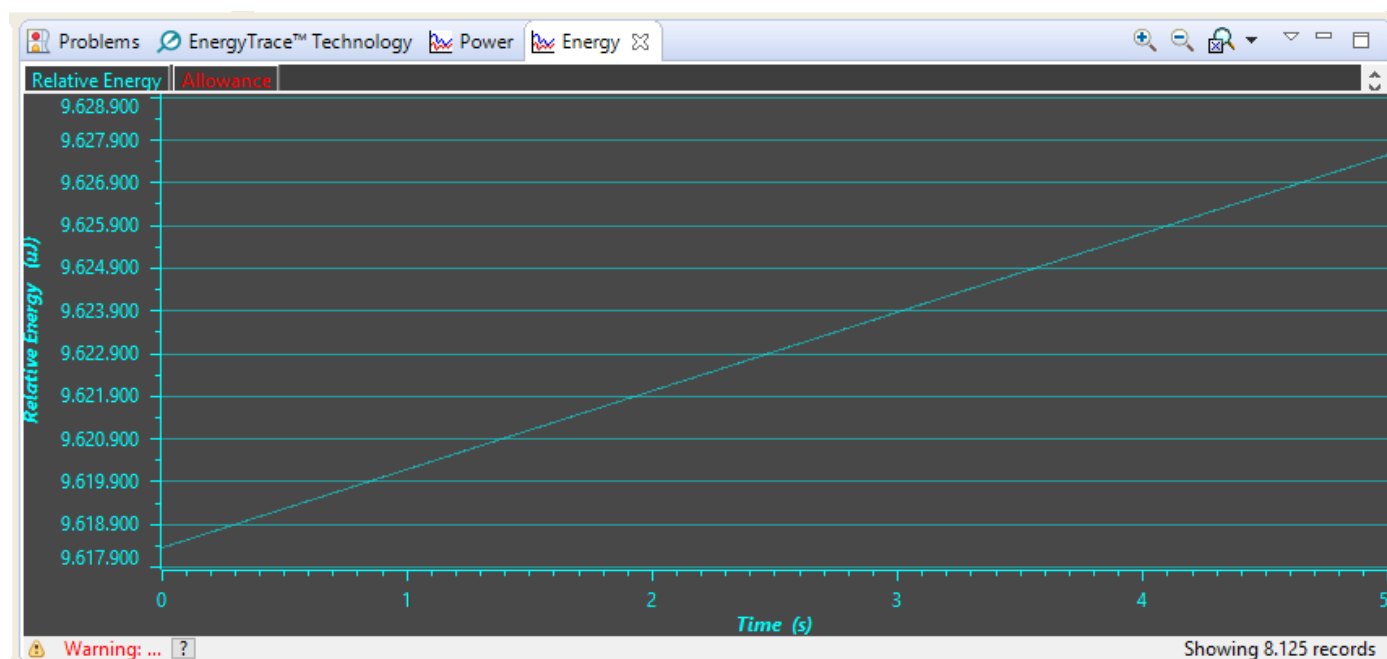


Figura: Perfil de consumo de energia durante 5s com o *jumper* de P1.0 em J5 removido.

Considerando uma bateria (pilha) CR2032, que tem uma tensão nominal de 3V e possui uma capacidade de 220mAh, este programa pode ser executado continuamente durante 14,4 dias. Se colocarmos o *jumper* de P1.0 em J5, com o LED efetivamente a piscar este tempo reduz-se para 3,3 dias, donde constatamos que a maior parte da corrente é consumida pelo LED.



Figura: Uma bateria/pilha comum com a referência CR2032.

O tipo de bateria a considerar para os cálculos de tempo de duração pode ser selecionado no separador **Preferences**, conforme ilustrado na figura seguinte. Neste menu também se pode customizar uma bateria em termos da sua tensão e capacidade para corresponder ao que iremos efetivamente usar.

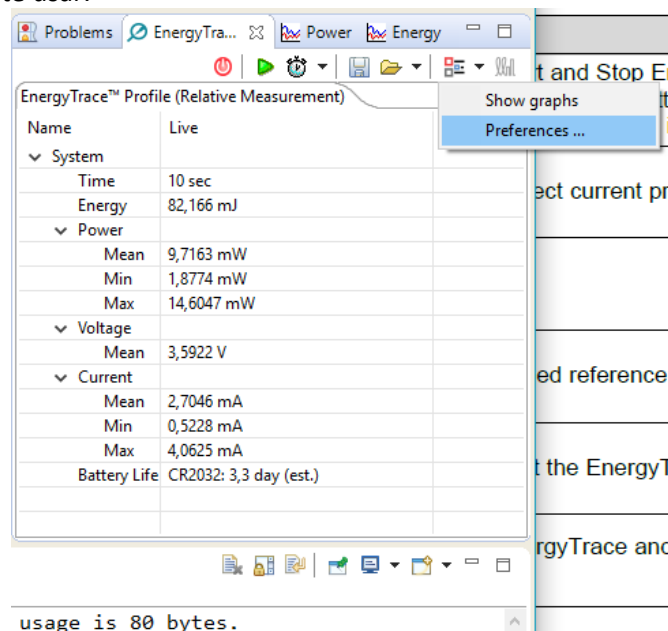


Figura: O perfil de consumo da figura corresponde a uma análise de 10s com o *jumper* inserido e o LED a piscar efetivamente.

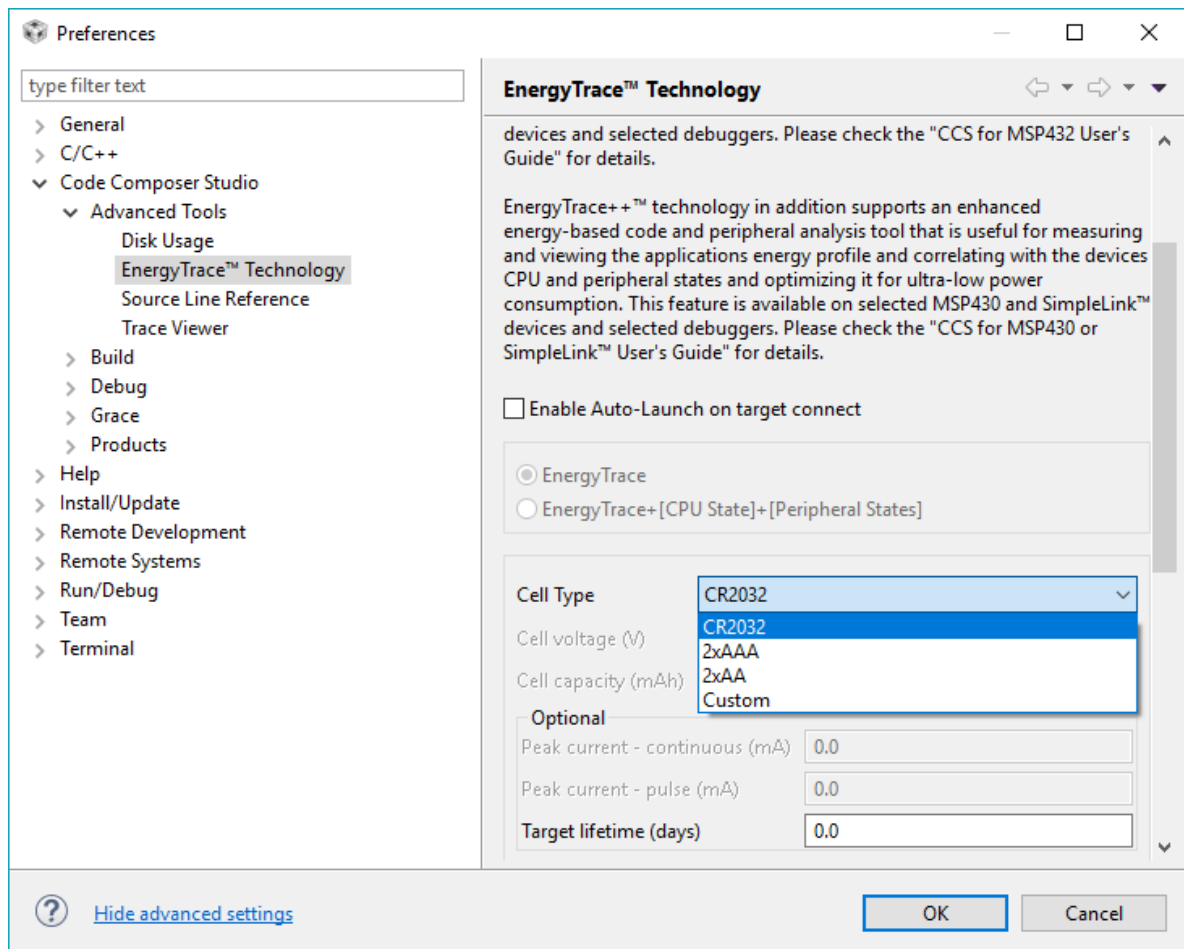


Figura: Painel das configurações de defeito da ferramenta *EnergyTrace Technology*. De notar que, como estamos a usar a placa MSP-EXP430G2 só temos como opção o modo simples *EnergyTrace*.

Questões:

Medir os valores da potência, tensão e corrente para a execução do programa em piscar LED em modo ativo.

Como se pode inferir, o programa acima é extremamente ineficiente em termos de consumo energético, dado que estamos a piscar um LED a uma frequência baixa, de apenas alguns Hertz (já iremos ver quanto), e no restante tempo, que na verdade corresponde á maior parte do tempo, o microcontrolador não está a executar nenhuma tarefa, no entanto continua no modo ativo.

Por defeito, não tendo o cristal LFXT1CLK instalado na placa do LaunchPad MSP430G2, a frequência de relógio do MCLK, e por consequência do SMCLK, que são gerados a partir do DCOCLK, é aproximadamente igual a **1.1 MHz** (ver [slau144j], pág. 275).

Questões:

Usando um osciloscópio, ou um contador de frequência, medir a frequência do relógio do CPU. Para isso deve colocar o sinal de relógio SMCLK no pino P1.0 do microcontrolador MSP430G2553.

Programando para Baixo Consumo

Vamos começar a preocupar-nos em usar os recursos à disposição para otimizar o consumo do sistema e o primeiro passo a seguir é configurar os pinos de modo a que estes não drenem corrente.

Os pinos que não são usados devem de ser configurados para funcionarem como pinos de I/O (entrada/saída), e, de modo a evitar que fiquem com um valor de tensão flutuante, devem de ser configurados preferencialmente como sendo de saída, de modo a consumirem o mínimo de corrente e não devem de ser conectados na placa de PCB. No caso de serem configurados como pinos de entrada deve-se de ativar a resistência de *pull-up* ou de *pull-down* de modo a que a tensão de entrada não seja flutuante [slau144j].

Vamos começar por introduzir esta alteração no programa anterior e ver se é possível observar alguma mudança no consumo.

O programa seguinte começa por configurar os portos 1 e 2 colocando nos seus *buffers* de saída o valor 0 (registo PxOUT), depois todos os seus pinos são configurados como sendo de saída (registo PxDIR) e finalmente é selecionada a função de I/O para os portos (registo PxSEL).

Neste programa também foi seguido o conselho dado pelo *ULP Advisor* onde a variável de controle do ciclo é comparada com 0, o que acarreta menos instruções em assembler.

```
#include <msp430.h>

/**
 * main.c
 */
void main(void)
{
    int i;
    // Stop watchdog timer to prevent time out reset
    WDTCTL = (WDTPW | WDTHOLD); // stop watchdog timer

    // Port 1 configuration:
    P1OUT = (0x00); // set output values
    P1DIR = (0xFF); // set P1.x as output
    P1SEL = (0x00); // select I/O function
    P1SEL2 = (0x00); // select I/O function

    // Port 2 configuration:
    P2OUT = (0x00); // set output values
    P2DIR = (0xFF); // set P1.x as output
    P2SEL = (0x00); // select I/O function
    P2SEL2 = (0x00); // select I/O function
}
```

```

while (1)
{
    for(i = 15000; i > 0; i--)
    { // Delay
    }
    P1OUT ^= (BIT0);
}
}

```

Listagem: Programa para acende/apaga LED alternadamente usando unicamente o modo ativo, com a configuração inicial dos ports P1 e P2.

Questões: Comparar o assembler gerado para o ciclo for do programa anterior, onde a variável de controlo é inicializada com o valor 15000 e comparada com zero em cada iteração do ciclo, com a primeira implementação deste programa, onde a variável de controlo do ciclo era inicializada com zero e comparada com 15000.

Meça os valores do consumo de potência, corrente e energia seguindo o procedimento descrito anteriormente. É possível observar alguma alteração nos valores da corrente consumida pelo dispositivo? Porquê?

2. Preparação para Utilização dos Modos de Baixo Consumo

Como é de suspeitar, a forma como estamos a realizar o atraso (*delay*) para ligar/desligar o LED é extremamente ineficiente em termos energéticos, para além de ser muito difícil controlar o seu tempo de duração. Se tomarmos atenção à pista que o CCS nos dá, como mostra a caixa a amarelo na figura seguinte, este recomenda a utilização de um módulo temporizador para gerar o atraso. Esta recomendação é feita por uma outra ferramenta integrada no IDE do CCS designada por *Ultra-Low Power Advice*, ou abreviadamente por ULP.

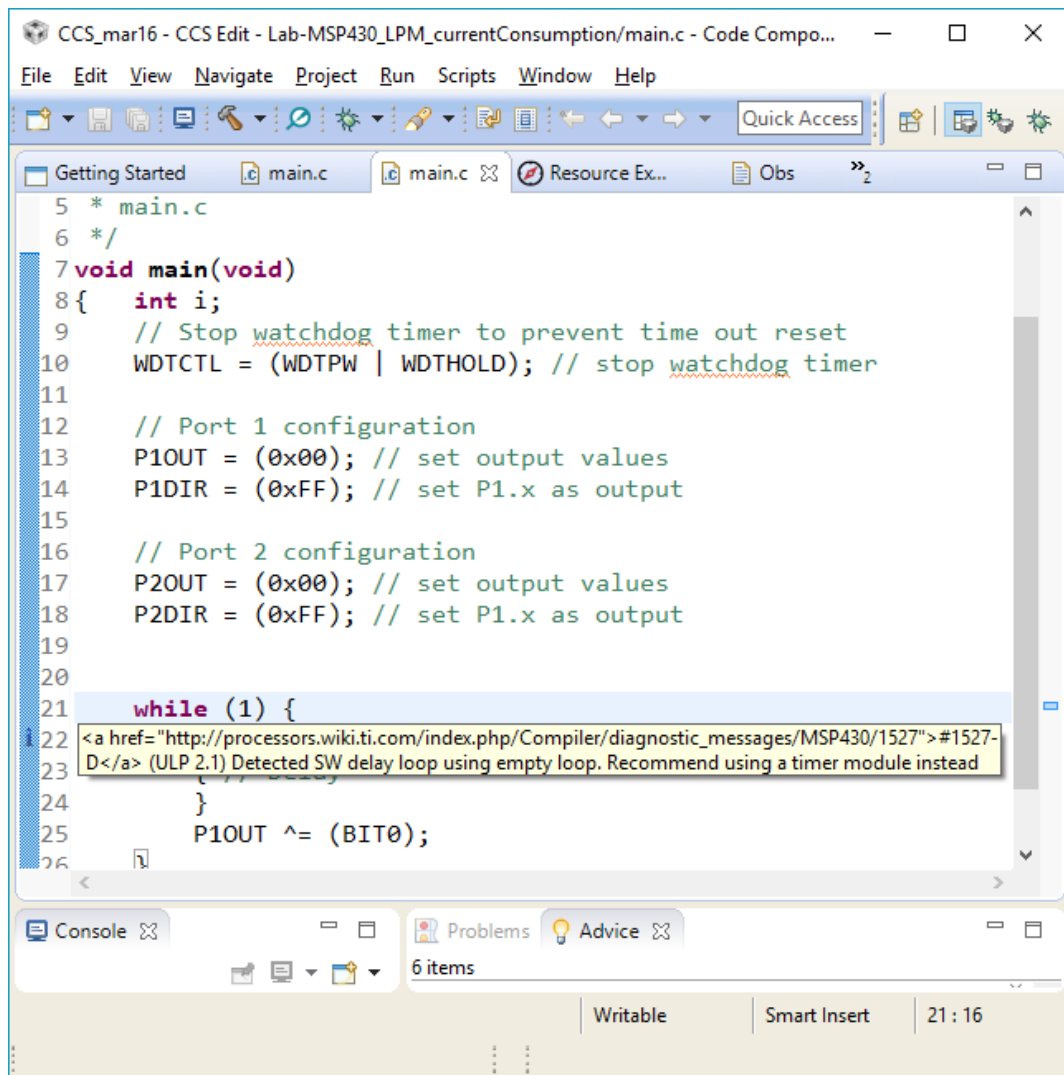


Figura: Janela de codificação do CCS com informação sobre (in)eficiência do código dada pelo ULP (*Ultra-Low Power Advisor*).

Após a compilação do programa o *ULP Advisor*, também deteta a ausência e recomenda a utilização de modo de baixo consumo, como assinalado na figura seguinte.

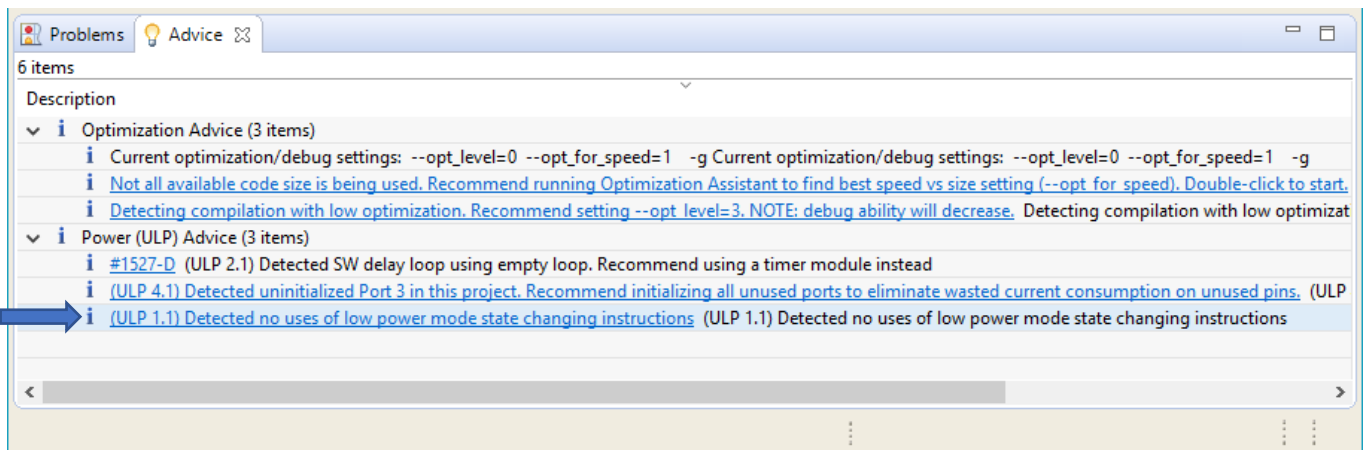


Figura: O *ULP Advisor* fornece conselhos após a compilação e análise do código. Aqui alerta para o facto de não estarmos a usar nenhum *low power mode* (modo de baixo consumo), entre outras sugestões que podemos seguir para reduzir o consumo. (A sugestão ULP 4.1 é proceder à inicialização do porto P3, no entanto este porto só está presente nos dispositivos com 28 e 32 pinos.)

Assim, a estratégia que vamos adotar passa por utilizar um módulo temporizador, que gera uma interrupção a intervalos regulares com uma duração bem definida, interrompendo o CPU que, entretanto, foi colocado e está em modo de baixo consumo. Após a interrupção, e quando no modo ativo, o CPU executa as diversas tarefas (neste caso alterna apenas o estado do pino associado ao LED) e assim que terminar volta a ser colocado no modo de baixo consumo.

Os passos a seguir pelo programa são:

1. Configuração do hardware.
 - a. Neste caso todos pinos são configurados como sendo de entrada/saída (**PxSEL = 0x00**), e colocados no modo de saída (**PxDIR = 0xFF**), e apresentando o valor digital 0 à saída (**PxOUT = 0x00**).
 - b. Configuração do temporizador para gerar interrupções a intervalos regulares.
2. Colocar o microcontrolador num modo de baixo consumo (LPM0, LPM1, LPM2, LPM3, LPM4) para minimizar o consumo de energia.
 - a. O modo de baixo consumo a escolher depende de quais os módulos periféricos que devem ficar ativos durante o mesmo.
3. Sempre que o microcontrolador é despertado, e passa do modo de baixo consumo para o modo ativo, executa as tarefas pretendidas e retorna ao modo de baixo consumo, aguardando a próxima interrupção.
 - a. Neste caso, a única tarefa a executar pelo CPU será alternar o estado de um pino de saída, o pino P1.0 entre 0 e 1.

Antes de passarmos a recorrer á utilização dos modos de baixo consumo, para colocar o processador a dormir quando este não está a mudar o estado do pino associado ao LED, que é a única coisa que o CPU faz nos programas anteriores, vamos organizar o código do programa anterior de forma a ser mais fácil perceber o que cada parte faz realmente e a introduzir alterações, dado que a quantidade de código vai aumentando à medida que se vão

acrescentando mais funcionalidades, acabando por tornar-se ininteligível se não estiver devidamente organizado.

No programa anterior os portos foram configurados diretamente na função **main**, o que, a acrescentar à configuração de outros módulos no futuro, torna difícil perceber o objetivo de cada parte desta função.

De modo a tornar o código mais legível, fácil de navegar e de alterar vamos organizar os programas seguindo sempre a mesma estrutura, onde o projeto é dividido em diversos módulos de modo a organizar as partes de código com um dado objetivo num mesmo ficheiro.

As regras principais são:

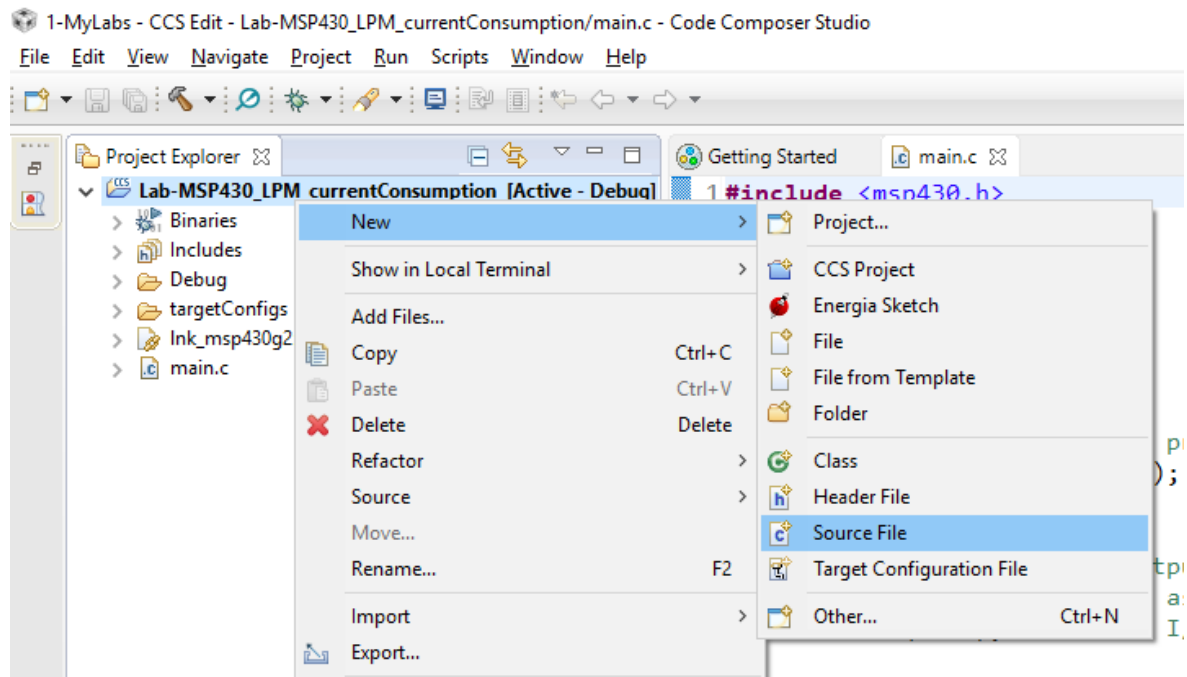
- O programa deve ser dividido em funções agrupadas por módulos, consoante o seu propósito. A designação dos módulos deve indicar qual o seu objetivo e funcionalidades. Exemplos de módulos são: **System.h** e **System.c**.
- O nome das funções dentro de cada módulo deve começar com o nome do módulo. Por exemplo: ex. **System_PortsInitialization()**.
- As funções globais devem de ser colocadas em ficheiros com extensão **.h** (de *header*) de modo a poderem ser incluídas e chamadas a partir de outros módulos.

Antes de aplicar as estratégias que temos avançado para conseguirmos um menor consumo de energia vamos reescrever o programa acima organizando o código por módulos.

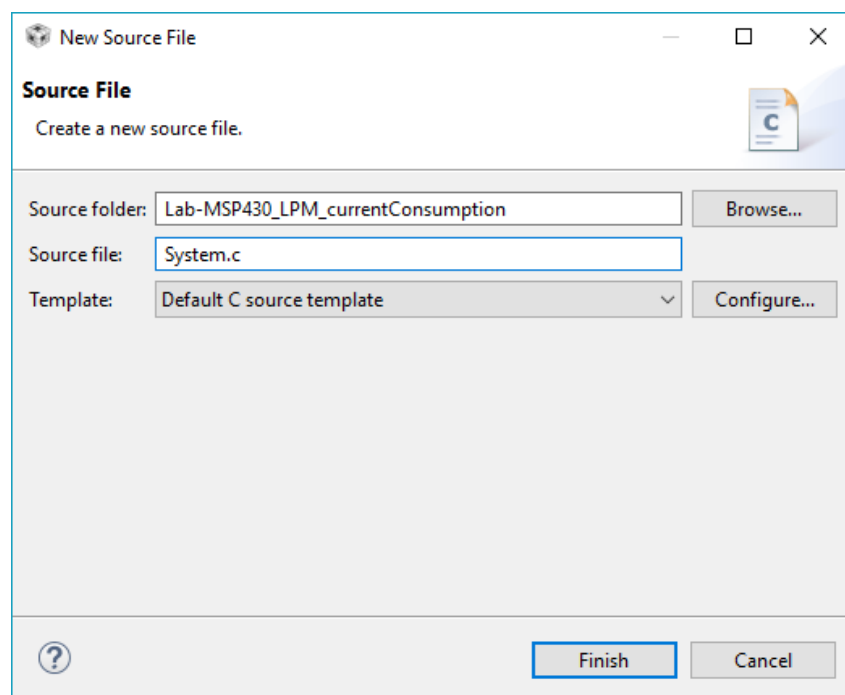
Todos as aplicações devem de conter o módulo designado por **main.c**, cujo nome indica diretamente que é onde está definida a função **main()**.

Todos as aplicações devem de conter um módulo designado por **System.c** onde são feitas todas as configurações iniciais do hardware e do software, antes do código da aplicação principal começar a ser executado. Para já só vai conter as configurações iniciais dos portos.

Vamos começar por criar o ficheiro **System.c** recorrendo ao auxílio do CCS clicando com o botão direito do rato no nome do projeto e seguindo: **New->Source File**



e dando o nome **System.c** ao ficheiro:



Este ficheiro vai conter uma função por cada módulo do microcontrolador configurado inicialmente. Para já temos apenas a função **void System_PortsInitialization(void)** onde é feito um mapa de cada porto, e onde é colocado o objetivo/função de cada pino do porto. O microcontrolador MSP430G2553 apenas possui os portos P1 e P2 cujos pinos são configurados como mostra a listagem seguinte:

```

/*
 * System.c
 *
 * Created on: 12/08/2017
 * Author: JoaoC
 */

#include <msp430.h>
#include <System.h>

void System_HWInitialization(void);
void System_PortsInitialization(void);

void System_HWInitialization(void)
{
    System_PortsInitialization();
}

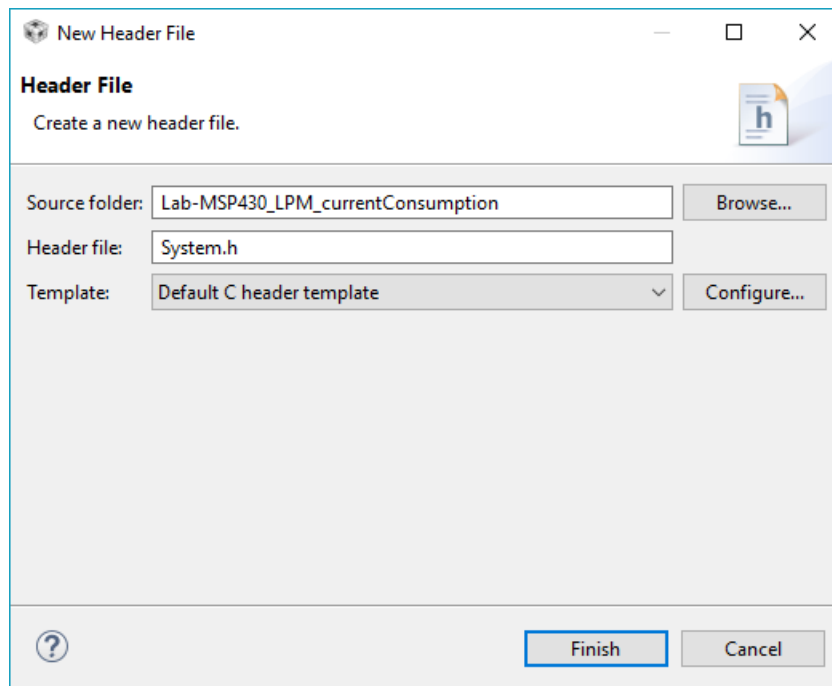
void System_PortsInitialization(void)
{
    // Port1 initialization
    // P1.0 - LED Red
    // P1.1 - n/c
    // P1.2 - n/c
    // P1.3 - n/c
    // P1.4 - n/c
    // P1.5 - n/c
    // P1.6 - n/c
    // P1.7 - n/c
    P1OUT = (0x00);
    P1DIR = (0xFF);
    P1SEL = (0x00);
    P1SEL2 = (0x00);

    // Port2 initialization
    // P2.0 - LED Red
    // P2.1 - n/c
    // P2.2 - n/c
    // P2.3 - n/c
    // P2.4 - n/c
    // P2.5 - n/c
    // P2.6 - n/c
    // P2.7 - n/c
    P2OUT = (0x00);
    P2DIR = (0xFF);
    P2SEL = (0x00);
    P2SEL2 = (0x00);
}

```

Todas as funções de configuração são chamadas a partir duma função global que tem a definição **void System_HWInitialization(void)**. Esta função reúne todas as chamadas de funções de configuração para cada um dos módulos, e por isso é a única função global que tem de ser chamada no início do programa, a partir da função `main`. Por isso, e para já, esta é a única função deste módulo que tem de ser tornada pública. Para isso deve ser criado um ficheiro com extensão `.h` onde deve constar a declaração desta função que pode ser incluído no módulo `main.c`.

Podemos recorrer à ajuda do CCS na criação do ficheiro `.h` clicando com o botão direito do rato sobre o nome do projeto seguindo: **New->Header File** e dando o nome `System.h` ao ficheiro.



As definições condicionais:

```
#ifndef SYSTEM_H_
#define SYSTEM_H_

#endif /* SYSTEM_H_ */
```

são automaticamente colocadas pelo CCS e permitem a repetição da inclusão deste ficheiro noutros módulos de código sem geração de conflitos devido à sua inclusão em mais do que um módulo.

Como só queremos expor a função **System_HWInitialize()** aos outros módulos, para já, só temos de a declarar no ficheiro **System.h** que fica:

```
/*
 * System.h
 *
 * Created on: 12/08/2017
 * Author: JoaoC
 */

#ifndef SYSTEM_H_
#define SYSTEM_H_

void System_HWInitialization(void);

#endif /* SYSTEM_H_ */
```

Listagem: Listagem do ficheiro **System.h**.

Tendo organizado o código no ficheiro **System.c** e **System.h** o módulo **main.c** pode ficar agora como:

```

#include <msp430.h>
#include "System.h"

/**
 * main.c
 */
void main(void)
{
    int i;
    // Stop watchdog timer to prevent time out reset
    WDTCTL = (WDTPW | WDTHOLD);    // stop watchdog timer

    System_HWInitialization();

    while (1)
    {
        for(i = 15000; i > 0; i--)
        { // Delay
        }
        P1OUT ^= (BIT0);
    }
}

```

Listagem: Listagem do ficheiro main.c recorrendo ao

De reparar que, para além da inclusão do ficheiro `msp430.h`, também é incluído o ficheiro `System.h`, entre aspas (""), por ser um ficheiro local e, em vez da configuração explícita dos portos na função `main`, como nas listagens anteriores, apenas é chamada a função `System_HWInitialization()`; definida em `System.c`.

Modo de Baixo Consumo 0 – LPM0 (Low-Power Mode 0)

Para utilizarmos os modos de baixo consumo: LPM0 a LPM3, temos de ter uma forma de interromper o processador do seu estado e despertá-lo de modo a que este execute as tarefas pretendidas. Para isso vamos usar um temporizador, o TimerA3 que entretanto vamos configurar juntando uma função para a sua configuração ao módulo `System.c`, que designaremos por `System_TimerA3Initialization()`.

Nesta função vamos configurar o temporizador/contador TimerA3 a ser alimentado pelo sinal de relógio SMCLK, cuja frequência é dividida por 2. O temporizador irá funcionar no modo de contagem *Up Mode*, isto é, o temporizador vai incrementado o conteúdo do registo TAR desde 0 até atingir o valor guardado no registo TACCR0, voltando a zero e recomeçando a contagem.

A figura seguinte mostra o modo de contagem em *Up Mode*:

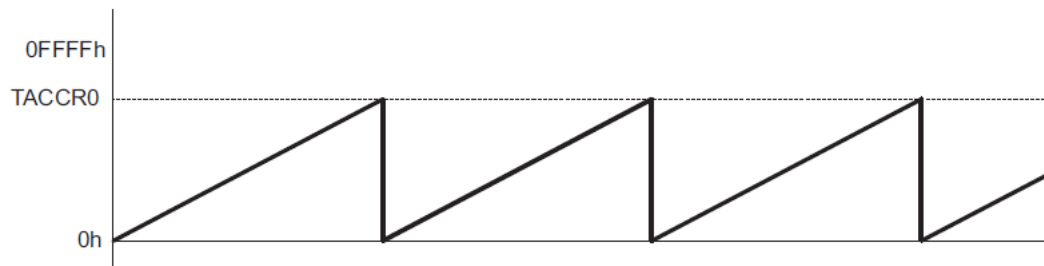


Figura: Modo de contagem em *Up Mode*. A linha do gráfico representa o conteúdo do registro TAR, que quando atinge o valor do registro TACCR0 volta a zero.

Quando o valor guardado no registro TACCR0 é atingido o bit de interrupção CCIFG do registro TACCR0 é ativado. A figura abaixo ilustra a ativação do indicador de interrupção CCIFG de TACCR0 quando se dá a transição do valor de CCR0-1 para CCR0. De notar que quando o registro TAR transita do valor CCR0 para 0 também é ativado o indicador de interrupção TAIFG:

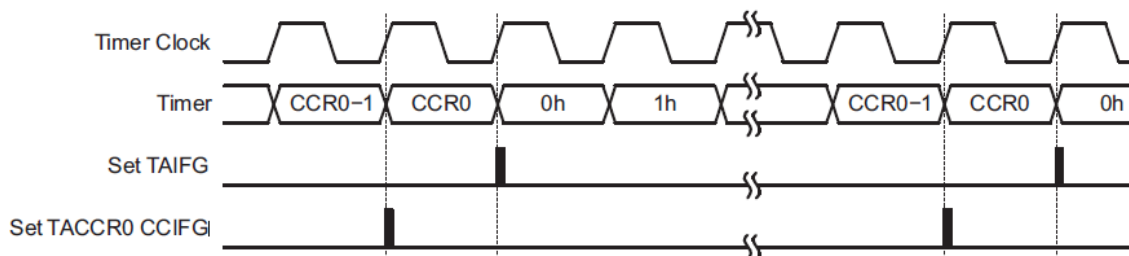


Figura: Ativação dos indicadores de interrupção do TACCR0 e do TAIFG.

O objetivo é que o LED pisque com uma frequência de 8Hz (deve apagar e acender 8 vezes por segundo), o que significa que temos de gerar interrupções com uma frequência de 16Hz. Ora usando o sinal de relógio SMCLK, que tem uma frequência de cerca de 1.1 MHz, temos de dividir este valor até obtermos a frequência pretendida. Se fizermos $IDx = 2$ passamos a ter uma frequência de 550 kHz para o contador. Para obtermos 16Hz temos de dividir este valor por: $550000/16 = 34375$. Se de cada vez que a contagem no registro TAR contar 34375 tiques de relógio gerarmos uma interrupção, iremos gerar 16 interrupções num segundo e alternando o estado do pino P1.0 associado ao LED estaremos a piscar o LED 8 vezes por segundo (que corresponde, assim, a uma frequência de 8Hz).

A listagem seguinte mostra a função para a configuração do TimerA3:

```
void System_TimerA3Initialization(void)
{
    // Set value for the CCR0
    TA0CCR0 = TICK_DIVISOR-1;

    // Configure from SMCLK | divisor 2 | Up Mode | clear
    TA0CTL = (TASSEL_2 | ID_2 | MC_1 | TACLR);

    // Enable interrupt on CCR0
    TACCTL0 = (CCIE);
}
```

Sendo TICK_DIVISOR um valor definido no início do programa que contém o divisor do contador do temporizador calculado anteriormente e definido como:

```
#define TICK_DIVISOR 34375
```

De realçar que a subtração de 1 ao valor de TICK_DIVISOR é um pormenor de importância prática quase nulo. A razão é que o contador começa em zero e, por isso, de modo a perfazer um número exato de contagens igual a TICK_DIVISOR temos de subtrair 1 ao valor colocado em TA0CCR0.

Vale a pena esclarecer a nomenclatura usada para nos referirmos aos registos associados aos temporizadores. Os temporizadores podem possuir, dois ou três blocos de captura/comparação: CCR0, CCR1 e CCR2. No caso do Timer_A no MSP430G2553, este possui três módulos de comparação e daí designação Timer_A3. Os MSP430 podem possuir mais do que um Timer_A. No caso do MSP430G2553, possui dois temporizadores do tipo A que são então designados por: Timer0_A3 e Timer1_A3. Daí os registos começarem por TA0x, dado estarmos a usar o Timer0_A3.

Configuração do BCM+

De modo a estarmos seguros da configuração do módulo de relógio do microcontrolador vamos prestar atenção á sua configuração.

Quando o sistema é iniciado os registos de configuração do BCM+ possuem o conteúdo da figura seguinte:

System_Clock			
>	1010 0101	DCOCTL	0x60 DCO Clock Frequency Control [Memory Mapped]
>	1010 0101	BCSCTL1	0x87 Basic Clock System Control 1 [Memory Mapped]
>	1010 0101	BCSCTL2	0x00 Basic Clock System Control 2 [Memory Mapped]
>	1010 0101	BCSCTL3	0x05 Basic Clock System Control 3 [Memory Mapped]

Figura: Valores de defeito dos registos de configuração do BCM+.

Podemos saber o conteúdo por defeito dos registos de duas formas: consultando as especificações do microcontrolador ou, no CCS, entrando no modo de **Debug**, seguindo o caminho **View->Registers** e procurando o conjunto de registos dos quais pretendemos saber o conteúdo. Se observarmos estes registos mesmo antes do programa começar a executar qualquer instrução teremos os seus valores de defeito.

A partir do conteúdo dos registos podemos ver o estado de cada módulo. Ainda podemos ver os valores de cada campo do registo clicando no sinal > junto ao registo:

▼	System_Clock		
▼	DCOCTL	01100000b (Binary)	DCO Clock Frequency Control [Memory Mapped]
	DCO2	0	DCO Select Bit 2
	DCO1	1	DCO Select Bit 1
	DCO0	1	DCO Select Bit 0
	MOD4	0	Modulation Bit 4
	MOD3	0	Modulation Bit 3
	MOD2	0	Modulation Bit 2
	MOD1	0	Modulation Bit 1
	MOD0	0	Modulation Bit 0
▼	BCSCTL1	0x87	Basic Clock System Control 1 [Memory Mapped]
	XT2OFF	1	Enable XT2CLK
	XTS	0	LFXTCLK 0:Low Freq. / 1: High Freq.
	DIVA	00 - DIVA_0	ACLK Divider 0
	RSEL3	0	Range Select Bit 3
	RSEL2	1	Range Select Bit 2
	RSEL1	1	Range Select Bit 1
	RSEL0	1	Range Select Bit 0
▼	BCSCTL2	0x00	Basic Clock System Control 2 [Memory Mapped]
	SELM	00 - SELM_0	MCLK Source Select 0
	DIVM	00 - DIVM_0	MCLK Divider 0
	SELS	0	SMCLK Source Select 0:DCOCLK / 1:XT2CLK/LFXTCLK
	DIVS	00 - DIVS_0	SMCLK Divider 0
▼	BCSCTL3	0x05	Basic Clock System Control 3 [Memory Mapped]
	XT2S	00 - XT2S_0	Mode 0 for XT2
	LFXT1S	00 - LFXT1S_0	Mode 0 for LFXT1 (XTS = 0)
	XCAP	01 - XCAP_1	XIN/XOUT Cap 0
	XT2OF	0	High frequency oscillator 2 fault flag
	LFXT1OF	1	Low/high Frequency Oscillator Fault Flag

De onde vemos que DCOx = 003h e MODx = 000h que, conjuntamente com o campo RSELx = 007h do registo BCSCTL1, define a frequência para o DCO de aproximadamente 1.1 MHz.

Da análise do registo BCSCTL1 vemos que: XT2OFF = 1, pelo que o cristal de alta frequência (XT2) não está presente (o MSP430G2 não tem este cristal); que estamos no modo de baixa frequência (XTS = 0) e o divisor do sinal de relógio ACLK é igual a 1 (DIVA = 0).

Da análise do registo BCSCTL2 vemos que: o sinal de relógio MCLK é gerado a partir do DCOCLK (SELM = SELM_0 = 00) que é dividido por 1 (DIVM = DIVM_0 = 00); o SMCLK é gerado a partir de DCOCLK (SELS = 0) que é dividido por 1 (DIVS = DIVS_0 = 00).

Da análise de BCSCTL3 vemos que: o campo XT2S, que define o intervalo de frequências de XT2 é definido como XT2S = XT2S_0 = 00 por defeito. Por defeito o sinal de relógio para ACLK é definido como proveniente do cristal de baixa frequência pois LFXT1S = LFXTaS_0 que, como não foi instalado (soldado) na placa do LaunchPad dá alerta de falha através do campo LFXT1OF = 1.

Podemos observar os sinais de relógio SMCLK e ACLK num osciloscópio através dos pinos P1.0 e P1.4, respetivamente, conforme mostra o mapa de pinos para o MSP430G2x53 da figura ativando a função dos pinos para colocarem estes sinais na saída.

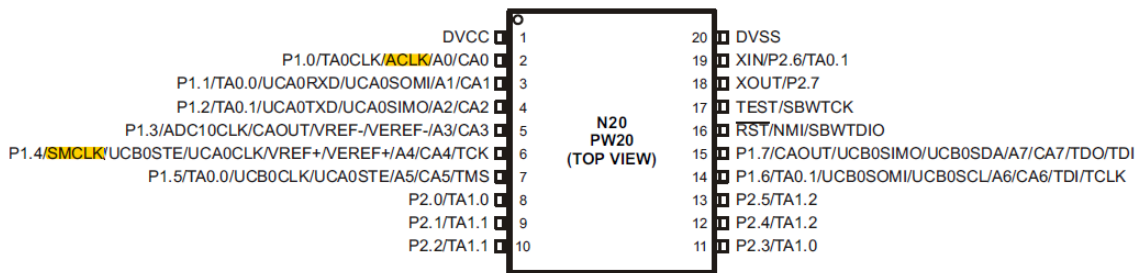


Figura: Mapa de pinos (*pinout*) dos microcontroladores MSP430G2x53 com encapsulamento PDIP.

Para isso temos de configurar os pinos P1.0 e P1.4 usando o registo P1SEL e P1SEL2 numa das suas funções alternativas, conforme a figura abaixo indica para o caso de P1.4, para colocar os sinais ACLCK e SMCLK na saída.

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS AND SIGNALS ⁽¹⁾					
			P1DIR.x	P1SEL.x	P1SEL2.x	ADC10AE.x INCH.x=1 ⁽²⁾	JTAG Mode	CAPD.y
P1.4/	4	P1.x (I/O)	I: 0; O: 1	0	0	0	0	0
SMCLK/		SMCLK	1	1	0	0	0	0
UCB0STE/		UCB0STE from USCI	1	1	0	0	0	0
UCA0CLK/		UCA0CLK from USCI	1	1	0	0	0	0
VREF+ ⁽²⁾ /		VREF+	X	X	X	1	0	0
VEREF+ ⁽²⁾ /		VEREF+	X	X	X	1	0	0
A4 ⁽²⁾ /		A4	X	X	X	1 (y = 4)	0	0
CA4		CA4	X	X	X	0	0	1 (y = 4)
TCK/	4	TCK	X	X	X	0	1	0
Pin Osc		Capacitive sensing	X	0	1	0	0	0

(1) X = don't care



(2) MSP430G2x53 devices only

Figura: Conteúdos dos registos de configuração para o pino P1.4 para as suas diferentes funções [SLAS735J]. Está assinalada a configuração para o pino funcionar como saída do sinal SMCLK.

Assim, P1.0 deixará de ter como função piscar o LED vermelho do LaunchPad – podemos usar, na sua vez, o LED verde associado ao pino P1.6.

Questão: Altere o programa de acordo com as indicações anteriores e observe os sinais de relógio ACLK e SMCLK a partir dos pinos P1.0 e P1.4.

Naturalmente constatamos que o pino P1.0 não apresenta nenhum sinal de saída relativo a ACLK. Isto porque ACLK só pode ser gerado a partir de LFXT1 ou de VLO, sendo configurado por defeito a partir de LFXT1 que, como não foi soldado nenhum cristal na placa LaunchPad, levanta o sinal OFIFG (*Oscillator Fault Interrupt Flag*) de interrupção de falha no oscilador presente no registo IFG1 (*Interrupt Flag Register 1*), conforme a figura abaixo ilustra.

▼  Special_Function		
>  IE1	0x00	Interrupt Enable 1 [Memory Mapped]
▼  IFG1	0x0E	Interrupt Flag 1 [Memory Mapped]
 NMIIFG	0	NMI Interrupt Flag
 RSTIFG	1	Reset Interrupt Flag
 PORIFG	1	Power On Interrupt Flag
 OFIFG	1	<u>Osc. Fault Interrupt Flag</u>
 WDTIFG	0	Watchdog Interrupt Flag

Questão:

1. Para evitar a interrupção de OFIFG crie um função no módulo System.c para inicializar o BCM+, configurando como fonte para ACLK o gerador de sinal de relógio VLO. Reinicie o sistema e verifique o sinal de relógio ACLK a partir do pino P1.0. Visualize o sinal de relógio ACLK num osciloscópio e meça a sua frequência (em vez do osciloscópio, pode-se usar um frequencímetro para obter uma medida mais precisa).
2. Configure o BCM+ com o seguintes critérios: configurar o DCO com os parâmetros de fábrica para uma frequência de 1MHz; MCLK e SMCLK devem ser gerados a partir do DCO. ACLK deve usar o cristal LFXT1 com uma frequência de 32768 kHz como referência, e configurar o valor correto para a capacidade do condensador associado ao cristal que está a ser utilizado.
 - a. Visualizar os sinais SMCLK e ACLK no osciloscópio ou frequencímetro.
 - b. Qual a corrente consumida neste modo usando o LPM0.
3. Repita o mesmo procedimento para medir a corrente consumida pelo sistema nos seguintes modos: LPM1, LPM2 e LPM3.

Bibliografia

SLAU144J – “MSP430x2xx Family User's Guide”, Texas Instruments, July 2013
(<http://www.ti.com/lit/ug/slau144j/slau144j.pdf>)

SLAA378D – Miguel Morales, Zachery Shivers, “Wireless Sensor Monitor Using the eZ430-RF2500,” Application Report, Texas Instruments, April 2011
(<http://www.ti.com/lit/an/sl原因378d/sl原因378d.pdf>)

SLAA603 - Brittany Finch, William Goh, “MSP430™ Advanced Power Optimizations: ULP Advisor™ Software and EnergyTrace™ Technology”, Application Report, Texas Instruments, June 2014 (<http://www.ti.com/lit/an/sl原因603/sl原因603.pdf>)

SLAS735J - MSP430G2x53, MSP430G2x13 Mixed Signal Microcontroller, Texas Instruments, May2013 (<http://www.ti.com/lit/ds/sl原因735j/sl原因735j.pdf>)