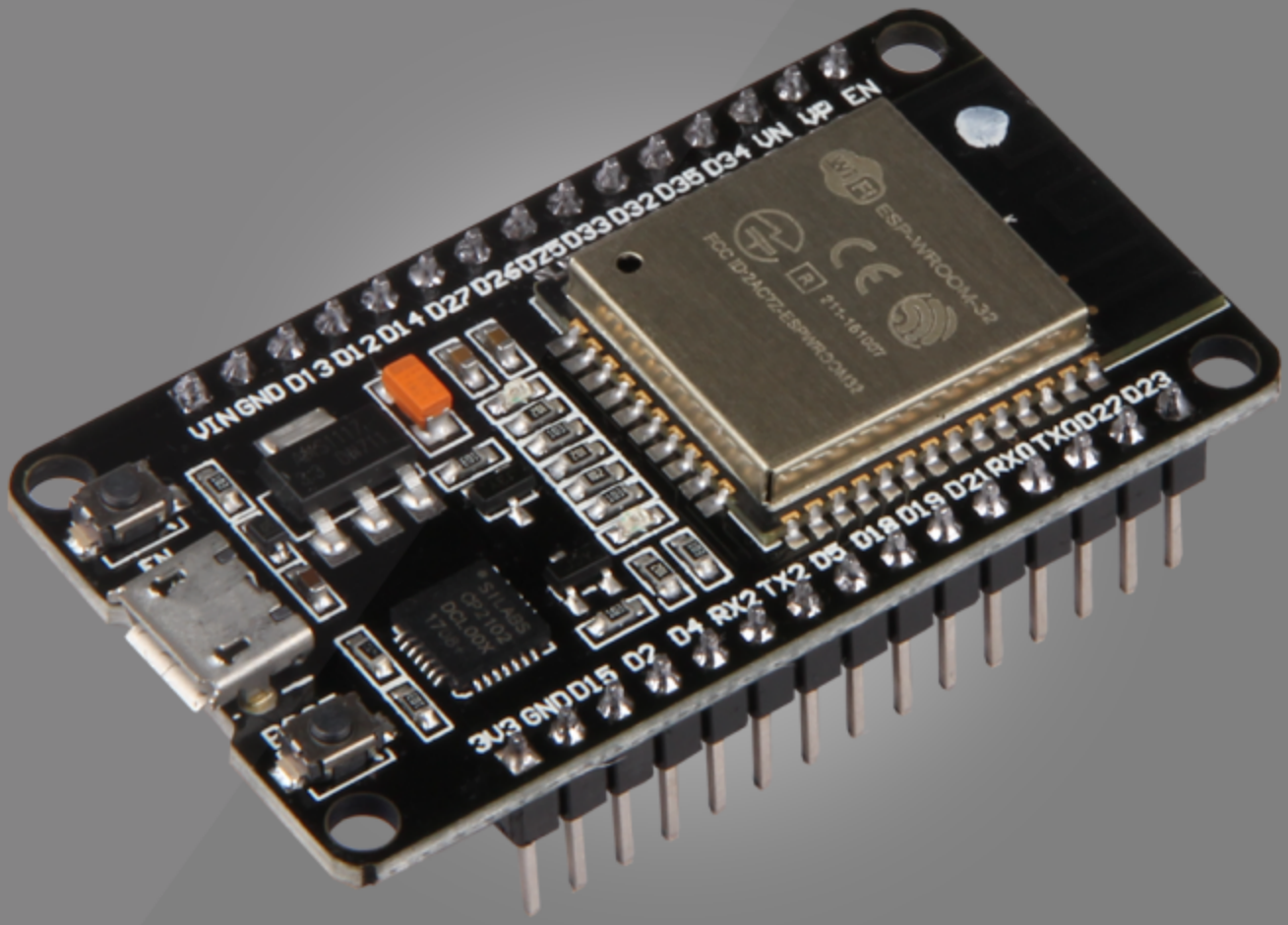
Parte 3

Coleção ESP32 do Embarcados

Aplicaçõies low power com ESP32



EMBARCADOS

Olá,

Obrigado por baixar o nosso ebook: **Coleção ESP32 do Embarcados** - Parte 3. Esse ebook traz uma coleção de textos já publicados no Embarcados.

Fizemos um compilado de textos que consideramos importantes para os primeiros passos com a plataforma. Espero que você aproveite esse material e lhe ajude em sua jornada.

Continuamos com a missão de publicar textos novos diariamente no site.

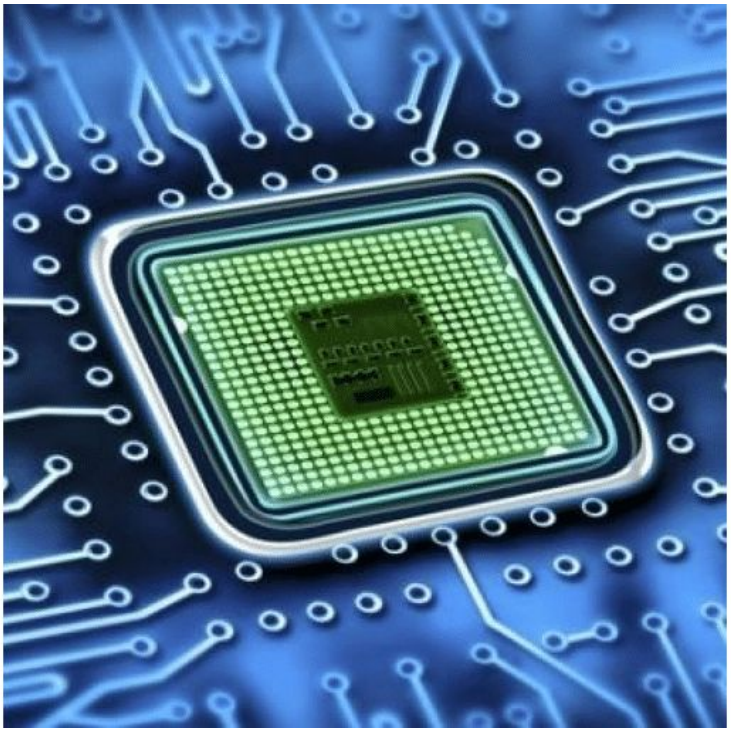
Um grande abraço.

Equipe Embarcados.

Sumário

Conhecendo o co-processador ULP (Ultra Low Power) do ESP32

Autor: José Morais



O ESP32, SoC da Espressif, além de todo potencial que é mostrado pelos mais diversos autores, conta com um terceiro processador, que ainda não foi dito aqui e em muitos lugares. Este terceiro processador é chamado de ULP (Ultra Low Power), feito especificamente para operação em Low Power, já que o consumo é de aproximadamente

150 uA, ou onde precisamos de co-processador para efetuar algum processamento paralelo ao sistema principal, como ler estados de pinos (digitais e analógicos), efetuar operações aritméticas/lógicas e até comunicações com outros sistemas embarcados enquanto o sistema principal faz algum processamento pesado ou de tempo crítico. O ULP é programado no ambiente de desenvolvimento chamado ESP-IDF, que é o local padrão de desenvolvimento do ESP32 e conta com todas ferramentas e features disponíveis para esse microcontrolador.

Por que e quando devo utilizar o ULP?

O ULP é indicado para operações e projetos Low Power ou ajudar o sistema principal em alguma tarefa (co-processamento). Vamos justificar estes 2 usos básicos:

- **Low Power:** Normalmente em dispositivos portáteis, o poder de processamento é baixo ou depende de algum evento externo para que ocorra alguma ação, por exemplo um sensor chegar em um valor específico. Por conta disso, não é indicado o uso de todo o potencial do microcontrolador, aí entramos com os métodos de economia do microcontrolador (Sleeps).

O ULP entraria em ação neste caso para, por exemplo, fazer Polling do sensor, já que seu consumo (150 uA) é extremamente mais baixo que o sistema principal (30-50 mA). Quando o ULP verificar que o sensor gerou um evento necessário para a ação do sistema, acordará o sistema principal para que seja feito o serviço pesado, como envio de dados ao banco de dados.

- **Co-processamento:** Imagine que seu microcontrolador está trabalhando quase no seu limite. Por exemplo, fazendo leitura de dezenas de sensores e controlando diversos atuadores; isso tudo sem que haja perda de sincronismo por conta de um passo longo a ser realizado para tomada de decisão, como uma operação

matemática complexa ou tratamento dos sensores e atuadores. Com esse cenário, é comum adicionar outros microcontroladores para ajudar a efetuar algumas tarefas específicas e livrar o sistema principal para focar na tarefa crítica.

O ULP entraria nesse cenário justamente efetuando alguma dessas 3 ações. Podemos atribuí-lo tanto a leitura dos sensores quanto o controle dos atuadores e até as contas matemáticas para tomada de decisões. Ele seria visto como um outro microcontrolador do sistema, mas este "mora" dentro do próprio ESP32, o que ajuda extremamente na velocidade de processamento e evita comunicações lentas entre sistemas separados. Poderíamos deixar o controle dos atuadores por conta total do ULP, assim livramos o sistema principal dessa tarefa e deixariamos a leitura dos sensores melhores, podendo-se adicionar um Polling mais profundo.

Faremos esses 2 projetos de forma simples, separadamente, para mostrar os conceitos de aplicação do incrível ULP:

- **Low Power:** Deixaremos o microcontrolador em Deep Sleep para economia de energia, enquanto o ULP ficará lendo um sensor analógico. Quando for detectado o nível de tensão definido, efetuará alguma ação como acordar o microcontrolador para uma tarefa pesada ou algo do tipo;
- **Co-processamento:** Faremos alguma tarefa pesada de tempo crítico com o sistema principal em que qualquer desvio de processamento prejudicará essa tarefa designada. O ULP ficará lendo sensores para detectar se há alguma mudança que precise ser feita na tarefa do sistema principal e, assim, livramos o sistema principal de fazer a checagem de sensores, visto que iria interferir no tempo de processamento e atrapalhar o Flow code.

Não será ensinado aqui como instalar a ESP-IDF + ULP, entretanto nos links abaixo há todas as instruções para instalação da ESP-IDF e do ULP na IDF.

Instalação da ESP-IDF

http://esp-idf.readthedocs.io/en/latest/get-started/index.html

Instalação do ULP na IDF

http://esp-idf.readthedocs.io/en/latest/api-guides/ulp.html

Características do ULP

O ULP pode ser pensado como um microcontrolador dentro de outro. Há algumas restrições de acesso às memórias e periféricos, mas vamos adotá-lo como um microcontrolador independente dentro do ESP32 para facilitar a didática. Ele é focado em Low Power e, por causa disso, seu consumo é extremamente baixo. Mesmo sabendo desse foco, podemos utilizar para o que bem entendermos, como no co-processamento.

- Consumo máximo (100% duty cycle): 150 uA;
- Clock: 8,5 MHz +- 7%;
- Memória (RTC_SLOW_MEM): 8 KB;
- 4 Registradores de 16 bits para uso geral (R0-R3);
- 1 Registrador de 8 bits para contagens e loops (STAGE_CNT);
- 26 Mnemônicos.

Aqui me cabe fazer algumas observações importantes para que não haja confusões na hora de você testar na prática.

- O consumo máximo é obtido quando o ULP fica 100% do tempo ligado, entretanto, é comum de utilizá-lo como se fosse um LED piscando com PWM, por isso foi dito com 100% duty cycle. O método utilizado para baixar ainda mais seu consumo é criar paradigmas para que seja feito algum processamento como ler um sensor, voltar a dormir e acordar algum tempo predefinido depois para ler novamente o sensor ou efetuar sua tarefa.

- O clock ainda é uma variável para muitos, já que foi dito nos datasheets que seu clock é de 8 MHz, entretanto isso não é uma verdade absoluta. O clock do ULP está numa grande faixa de variação de chip para chip, que varia inclusive pela temperatura em que o microcontrolador se encontra. A faixa de variação do Clock é de 7,905 MHz até 9,095 MHz.

Por conta dessa grande variação de chip para chip e ainda a diferença pela temperatura, para nossa sorte, os desenvolvedores do ESP32 criaram uma função na IDF que retorna o clock aproximado e, com isso, conseguimos utilizar esse clock calculado para efetuar tarefas de tempo crítico ou comunicações seriais de alta frequência, onde o tempo é crucial. Ainda sim, essa função nem sempre retorna o mesmo valor, fazendo com que tenhamos que fazer alguma espécie de benchmark para cálculo real do clock, como algum PWM e osciloscópio, analisador lógico ou frequencímetro.

- A memória dedicada do ULP é de 8 KB (RTC_SLOW_MEM), onde fica seu código e variáveis, entretanto, é óbvio que podemos comunicar-se com o sistema principal, já que essa memória é compartilhada entre os 3 processadores. Podemos fazer envio de variáveis para lá e também a obtenção delas, onde há uma imensa quantidade de memória disponível e, com isso, criar métodos para expandir a memória do ULP através da memória principal.

Podemos tanto acessar variáveis criadas no sistema principal (Main Core programado em C/C++) como também acessar as variáveis criadas no ULP (programado em Assembly) pelo sistema principal. Essa capacidade de comunicação entre os 2 sistemas permite uma incrível e gigantesca gama de aplicações.

- Os Mnemônicos, ou Instruction set disponíveis para uso no ULP, estão disponíveis tanto no Datasheet quanto nesta página web.

Entendendo o funcionamento do ULP

Vamos começar com um fluxograma de como é o funcionamento do sistema inteiro até que o ULP entre em execução. O sistema principal deve definir (reservar) a memória de uso e também seu Entry Point antes que o ULP entre em execução, ou seja, ele é dependente do sistema principal para ser iniciado.

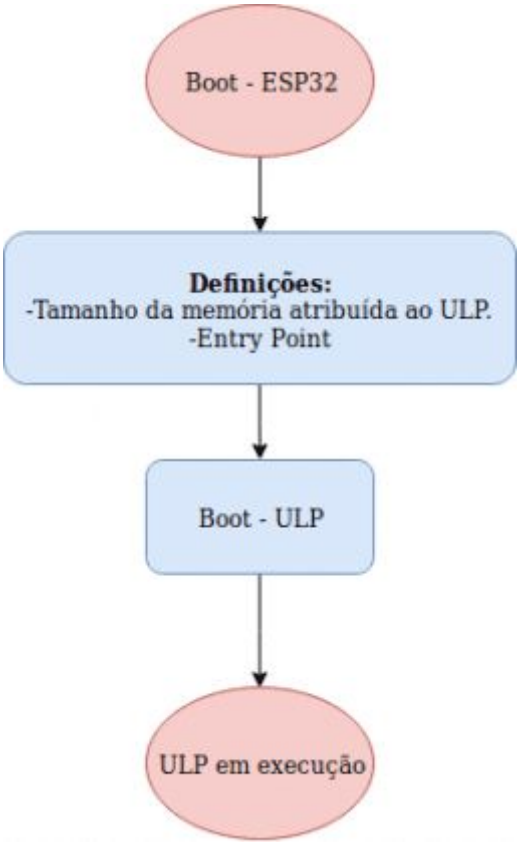


Figura 1 - Fluxograma de boot do ULP.

Ainda podemos utilizá-lo (para uma economia de energia maior) similarmente a um LED com PWM. Isso ocorre com a junção de 3 itens:

1. **Timer do ULP:** Responsável por acordá-lo com o tempo pré-definido. O período deste pode ser definido e alterado a qualquer momento, tanto pelo sistema principal quanto pelo próprio ULP. Após otTimer acordar o ULP, o periférico entra

em uma espécie de “espera” e fica inativo até que o comando HALT seja executado pelo ULP, onde todo o processo é reiniciado e o timer volta a contar para acordar o ULP novamente.

2. **Execução do código:** Após ser acordado, começará a execução normal de seu código.
3. **Mnemônico HALT:** Responsável por fazê-lo dormir.

Vejamos um gráfico do funcionamento do método citado acima, que é bastante utilizado para operações e projetos de Low Power:

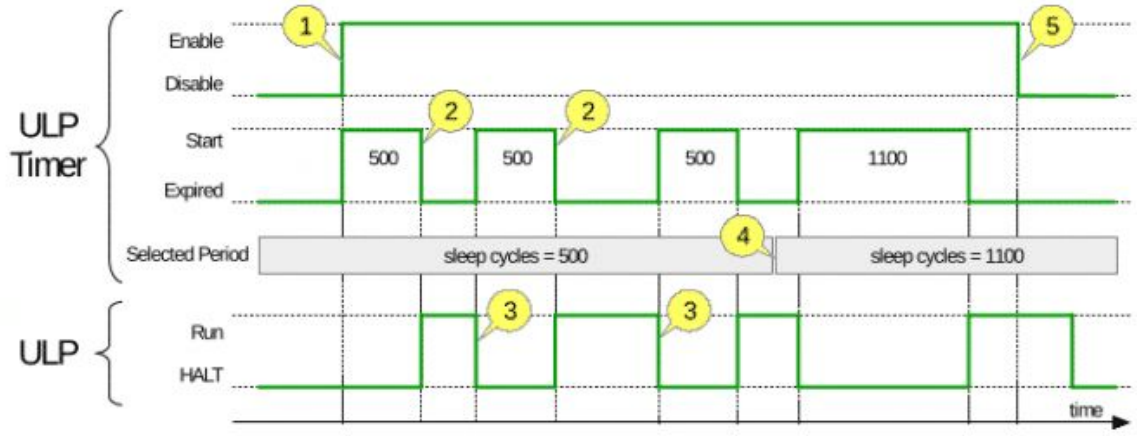


Figura 2 - Ciclos de Sleep <-> Wakeup do ULP.

1. Timer habilitado. Como pode ser visto, o Sleep cycles é de 500, logo, após 500 ciclos do Timer, o ULP acordará automaticamente;
2. 500 ciclos expirado e ULP entra em execução do seu código;
3. Ao fim de seu código, o comando HALT é executado, colocando o ULP para dormir novamente. O Timer também é reativado e retorna a contagem dos ciclos;
4. Sleep cycles mudado para 1100, a frequência de execução do código será menor que antes;

5. Timer desabilitado. Caso você execute o HALT após o Timer estar desabilitado, o ULP não acordará novamente.

Já para projetos onde é usado para ajudar no processamento principal, normalmente, não é utilizado essa técnica para baixar o consumo. Então nós iremos apenas ativar o ULP e mantê-lo funcionando o tempo todo ou desejado.

No próximo post desta série sobre ULP vamos começar a programá-lo focando em operações e projetos de Low Power, deixaremos o sistema principal em Deep Sleep enquanto o ULP lê um sensor analógico para efetuar alguma ação, como acordar o ESP32 do Deep Sleep. Os projetos feitos serão simples apenas para demonstração básica do funcionamento e não dificultar o entendimento geral.

Usando o ULP do ESP32 em projetos Low Power

Autor:José Morais

Vamos dar início à programação do ULP do ESP32 e finalmente testar esse item incrível e relativamente raro entre microcontroladores. A programação do sistema principal é feita em C/C++, entretanto, o ULP é programado em Assembly e, por conta disso, afasta muitas pessoas que não têm um contato tão grande com programação de baixa abstração. Mas vamos em frente que tudo será devidamente explicado e comentado.

O código basicamente irá colocar o ESP32 em Deep Sleep, enquanto o ULP lê um sensor analógico (LDR) para detectar sombras no local. Caso perceba uma sombra, efetuará uma

ação como acordar o ESP32 e incrementar uma variável persistente entre os ciclos de Sleep <-> Wakeup, que também é outro item importante, já que a Flash tem ciclo de escrita baixo.

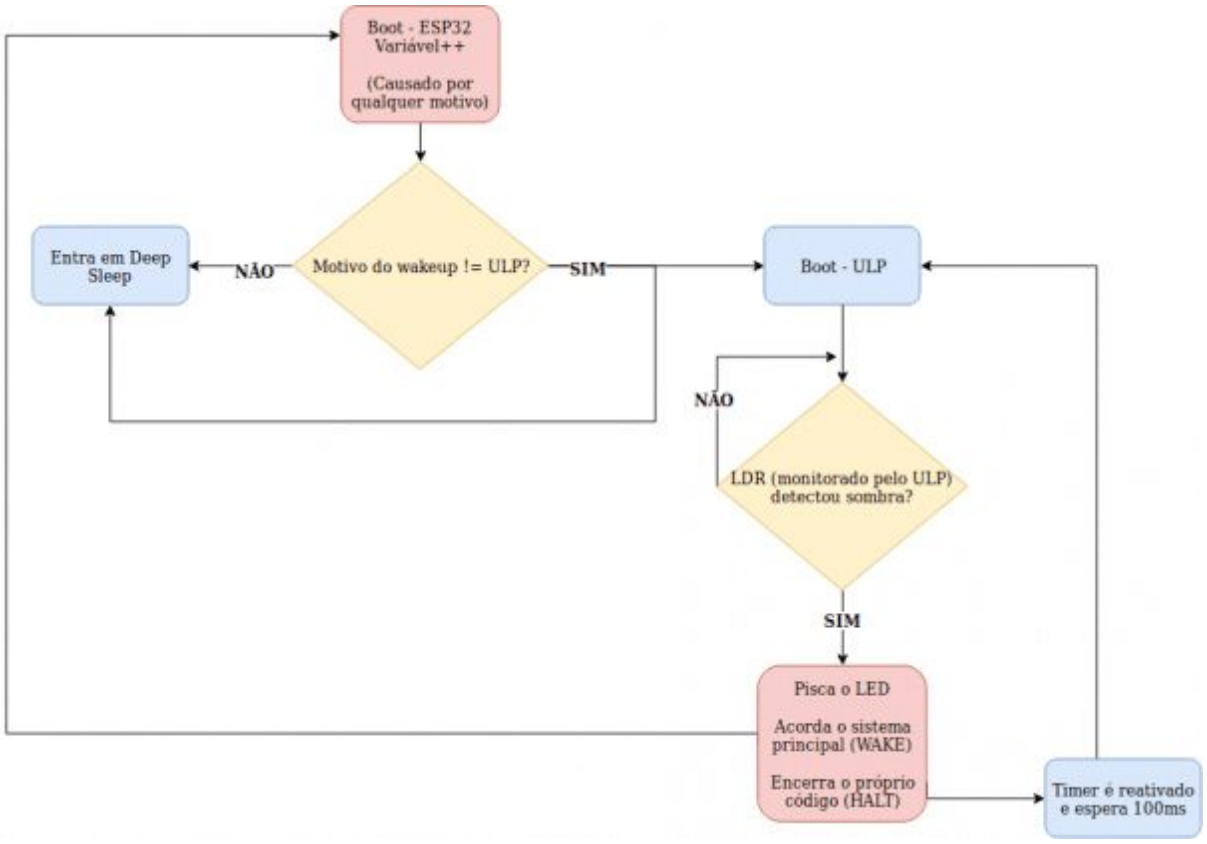


Figura 1 - Fluxograma do código.

Código em C++:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_io_reg.h"
#include "soc/sens_reg.h"
#include "soc/rtc_cntl_reg.h"
```

```
.bss//Declaracao de variaveis aqui


.text

     .global main
     main://Inicio do codigo (Entry point)
          WRITE_RTC_REG(RTC_GPIO_OUT_W1TC_REG,
RTC_GPIO_OUT_DATA_W1TC_S+12, 1, 1)//LED OFF (GPIO2 = LOW)


     loop:

          move r0, 0     //R0 = 0
          move r1, 0     //R1 = 0
          stage_rst //stage_cnt = 0

          //Aqui sera feito um laco FOR() para 4 leituras do ADC e
depois, tiramos uma media
          1:
               stage_inc 1   //stage_cnt++
               adc r1, 0, 7  //efetua a leitura ADC do GPIO34 e
guarda no R1
               add r0, r0, r1//R0 = R0 + R1 (guarda o total das
leituras)
               wait 65000
               wait 12000    //delay de 10ms
          jumps 1b, 4, lt         //retorna a label 1 enquanto
stage_cnt < 4

          rsh r0, r0, 2      //divide o total das leituras (4) por 4
          jumpr wkup, 1600, ge    //se valor do ADC >= 1600, ativa o
LED para indicar o evento, acorda o sistema principal e encerra o
ULP
```

```
                    //entretanto, o Timer0 foi ativado para
100ms, entao apos 100ms do HALT, ULP iniciara novamente

    jump loop//retorn ao loop


    wkup:

        WRITE_RTC_REG(RTC_GPIO_OUT_W1TS_REG,
RTC_GPIO_OUT_DATA_W1TS_S+12, 1, 1)//LED ON  (GPIO2 = 1)
        stage_rst
        1:
            stage_inc 1
            wait 32000
        jumps 1b, 125, lt  //delay de 500ms
        WRITE_RTC_REG(RTC_GPIO_OUT_W1TC_REG,
RTC_GPIO_OUT_DATA_W1TC_S+12, 1, 1)//LED OFF (GPIO2 = 0)

        wake //Acorda o sistema principal
        halt //Encerra o codigo do ULP, mas iniciara novamente apos
100ms
```

Testando esse código, podemos observar que mesmo com o sistema principal em Deep Sleep, o ULP continua em execução normal, que no caso é fazendo a leitura do sensor analógico (LDR) e, caso o valor (luminosidade) chegue ao valor definido, irá piscar um LED indicativo e acordará o sistema principal, que também mostrará no Serial Monitor a quantidade de vezes que ocorreu o Wakeup. Em teoria, removendo todos os componentes extras do circuito, como sensores e atuadores, o consumo com o sistema principal em Deep Sleep enquanto o ULP está em execução deve ser <=150 uA. Imagine quantas aplicações podemos dar a esse pequeno guerreiro para projetos de Low Power?

Praticamente um microcontrolador dentro de outro, porém de baixo consumo energético!

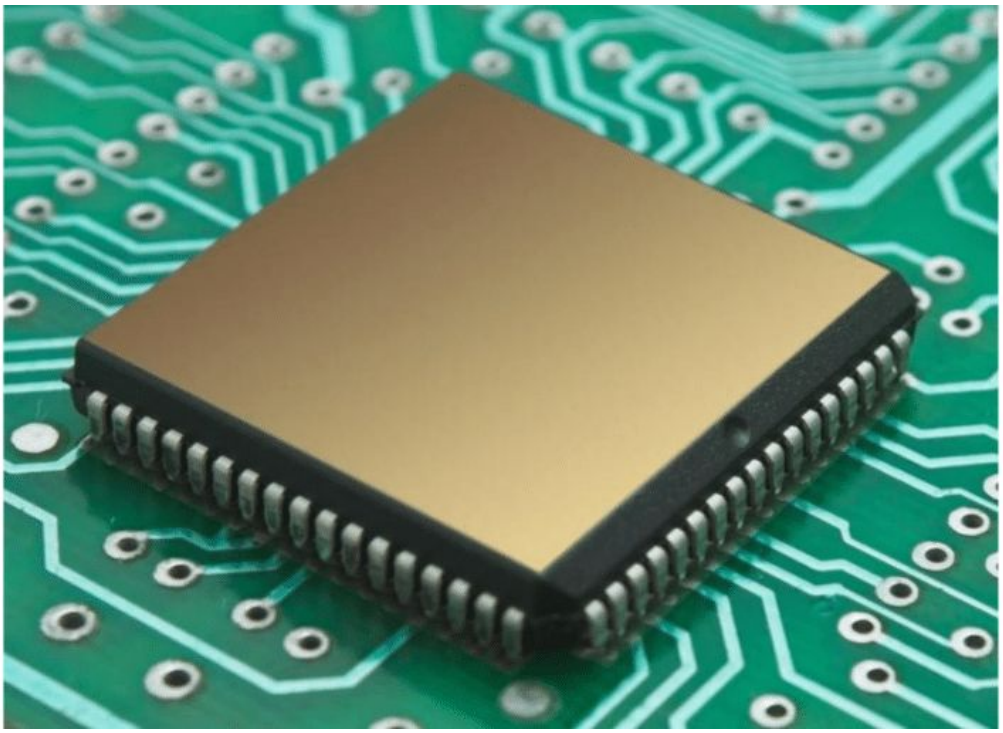
Utilizamos uma variável na memória RAM do RTC Domain, logo, não perdemos informação durante ou após o Deep Sleep, sendo uma ótima maneira de manter dados entre ciclos de Sleep <-> Wakeup sem precisar se preocupar com ciclos baixos de escrita na Flash.

No próximo post dessa série sobre o ULP do ESP32, vamos colocá-lo para ajudar o sistema principal no processamento, ficando encarregado de ler os sensores e ativar atuadores, já que essa tarefa interfere no bom funcionamento do Flow code.

Publicado originalmente no Embarcados, no dia 11/04/2018: link para o artigo original, sob a Licença Creative Commons Atribuição-CompartilhaIgual 4.0 Internacional.

Utilizando o ULP do ESP32 para co-processamento

Autor:José Morais



Agora que já sabemos programá-lo, vamos colocá-lo para ajudar no co-processamento do sistema principal, onde reside os 2 núcleos principais do ESP32, cada um rodando a 240 MHz. A ideia deste post é tentar simular alguma tarefa pesada para o microcontrolador, em que qualquer desvio de código acarreta em uma visível falha que pode ser relevante ou não.

Um método bem simples de estressar o microcontrolador é um loop infinito sem delay efetuando alguma tarefa. Pensando nisso, faremos um PWM via software que é

extremamente pior do que um PWM por hardware como Timers, que não são interferidos pelo código em si, como desvio por interrupções ou delays.

O PWM por software sofrerá com qualquer desvio de programação, inclusive interrupções externas, e é isso que será mostrado.

Vamos primeiramente entender a situação que nosso microcontrolador enfrentará:

- O sistema principal irá gerar um PWM de 4 MHz continuamente, onde não pode haver flutuações na frequência;
- Há um sensor conectado no microcontrolador para ativar um atuador sonoro em caso de pânico no sistema. Isso é feito através de uma interrupção externa, onde o código é desviado para uma rotina de interrupção (ISR) que ativa a buzina. Esse é um dos métodos mais eficazes e velozes para tratar eventos com microcontroladores, entretanto, vamos verificar como o sistema se comportou mais a frente...

Agora que já sabemos como o sistema precisa se comportar, vamos primeiramente testar o método de utilizar uma interrupção no mesmo núcleo que gera o PWM de 4 MHz. Veja o fluxograma que apenas idealiza como o processo é executado:

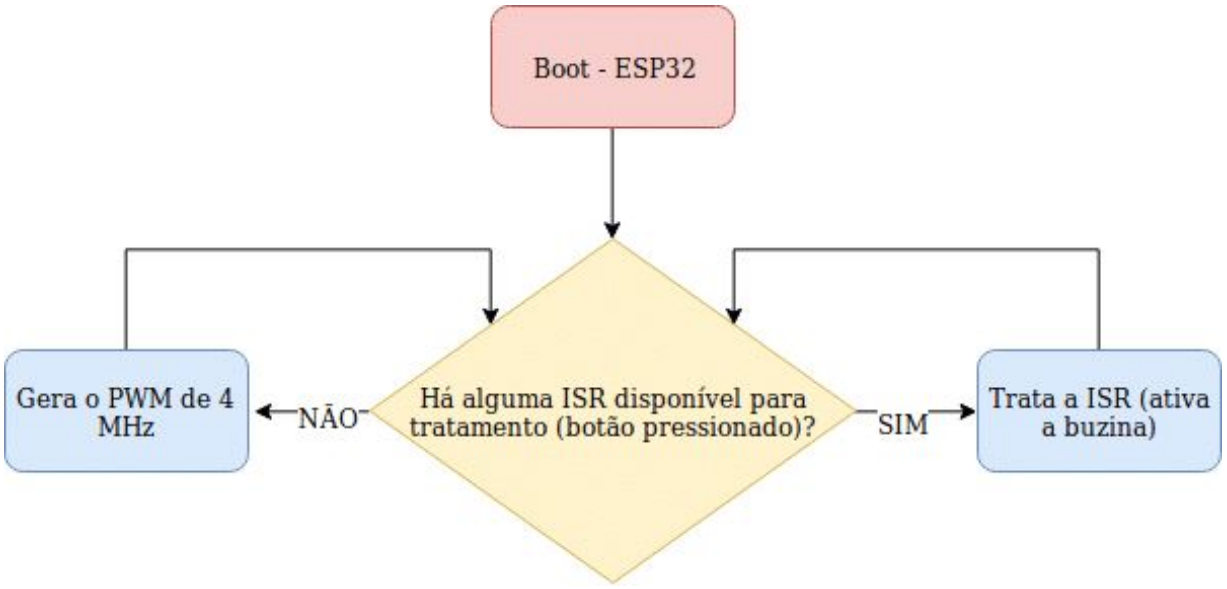


Figura 1 - Fluxograma do código com interrupção.

Código C++:

```
void atd()
{
	REG_WRITE(GPIO_OUT_W1TS_REG, BIT2);//Ativa o atuador
}

extern "C" void app_main()
{

	pinMode(13, OUTPUT);//Pino do PWM 4MHz
	pinMode(02, OUTPUT);//Pino do atuador
	attachInterrupt(0, atd, FALLING);//Interrupcao do botao de panico


	while (1)//PWM de 4MHz
	{
		REG_WRITE(GPIO_OUT_W1TS_REG, BIT13);

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
```

```
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
           REG_WRITE(GPIO_OUT_W1TC_REG, BIT13);

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
      }

}
```

Com um analisador lógico podemos visualizar os pinos do microcontrolador numa linha do tempo e ver como o sistema se comportou quando o botão de pânico foi pressionado.

Observações:

- Canal 0: PWM de 4 MHz;
- Canal 1: Atuador (buzina) ativo quando em nível lógico ALTO;
- Canal 2: Botão de pânico pressionado quando em nível lógico BAIXO.

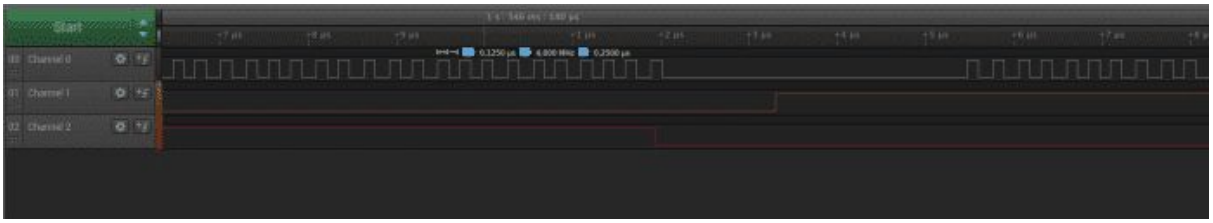


Figura 2 - Analisador lógico no código com interrupção.

É possível mais que claramente ver a ineficiência de tratamento da interrupção e gerência do PWM de alta frequência ao mesmo tempo. Mesmo o ESP32 trabalhando em 240 MHz, seu tratamento convencional de interrupções não é tão bom se comparado com outras arquiteturas de microcontroladores, como AVR, sendo possível ver que desde o botão ser pressionado e o atuador ligar, passaram-se aproximadamente 1,3 us, o que é relativamente lento. Todo o processo de desvio da interrupção até o retorno do PWM durou aproximadamente 3,5 us e isso não é tolerável no projeto. Apesar dos métodos convencionais serem lentos, é possível atribuir, via Assembly, interrupções de baixa latência diretamente na arquitetura da XTensa, mas não vamos tão a fundo por um problema que pode ser resolvido mais facilmente com algum dos outros 2 processadores.

Você pode estar se perguntando por que não atribuí a interrupção ao outro núcleo, já que o ESP32 conta com 2 núcleos principais e podemos deixar as tarefas separadas por núcleo, mas o outro núcleo também está ocupado com outra tarefa, então sobrou o ULP.

Vamos então programá-lo para ajudar o processamento do sistema principal. O ULP ficará encarregado de tratar todos sensores e atuadores do nosso sistema, que nesse caso é apenas um de cada, mas já é suficiente para analisar como é eficiente no co-processamento, visto que quanto mais sensores, mais o sistema principal seria prejudicado.

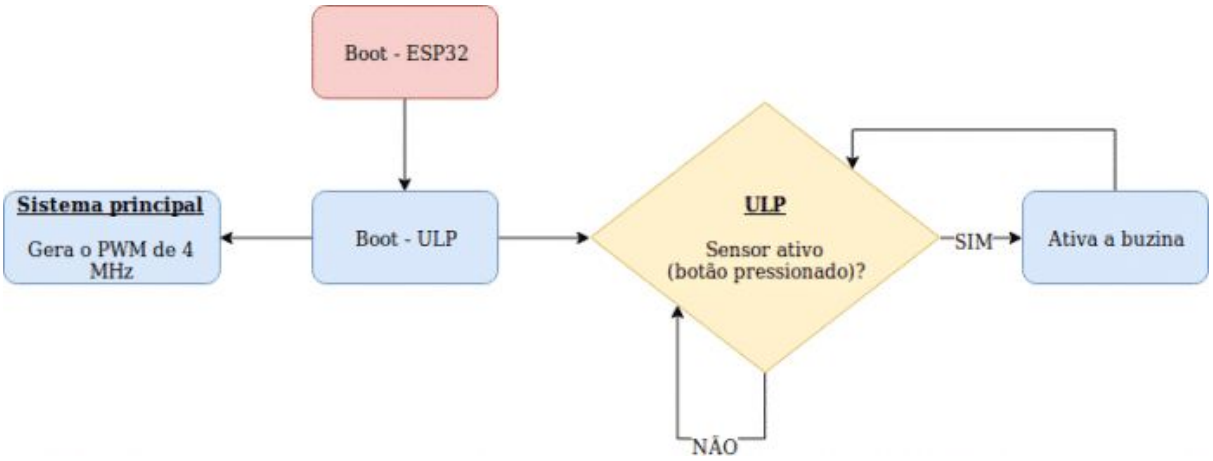


Figura 3 - Fluxograma do código com ULP

Código C++:

```
extern "C"
{
#include <driver/gpio.h>
#include <driver/rtc_io.h>
#include <ulp/ulp.c>
#include <ulp_main.h>
}

extern const uint8_t ulp_main_bin_start[]
asm("_binary_ulp_main_bin_start");//Inicio do binario
extern const uint8_t ulp_main_bin_end[] asm("_binary_ulp_main_bin_end");//Fim
do binario


void initULP()
{
      //Configura o GPIO0 como entrada no RTC Domain (ULP reside no RTC Domain)
      rtc_gpio_init(GPIO_NUM_0);
      rtc_gpio_set_direction(GPIO_NUM_0, RTC_GPIO_MODE_INPUT_ONLY);


      //Configura o GPIO2 como saida no RTC Domain (ULP reside no RTC Domain)
      rtc_gpio_init(GPIO_NUM_2);
```

```
	rtc_gpio_set_direction(GPIO_NUM_2, RTC_GPIO_MODE_OUTPUT_ONLY);



	ulp_load_binary(0, ulp_main_bin_start, (ulp_main_bin_end -
ulp_main_bin_start) / sizeof(uint32_t));//Carrega o binario na RTC_SLOW_MEM
	ulp_run((&ulp_main - RTC_SLOW_MEM) / sizeof(uint32_t));//Inicializa o ULP
}


extern "C" void app_main()
{
	initULP();//Funcao que inicializa o ULP

	//Configura o GPIO13 como saida
	gpio_pad_select_gpio(13);
	gpio_set_direction(GPIO_NUM_13, GPIO_MODE_OUTPUT);


	while (1)//PWM 4 MHz
	{
		REG_WRITE(GPIO_OUT_W1TS_REG, BIT13);//GPIO13 = HIGH

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
		REG_WRITE(GPIO_OUT_W1TC_REG, BIT13);//GPIO13 = LOW

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
	}
```

```
}
```

Código ASM:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_io_reg.h"
#include "soc/sens_reg.h"
#include "soc/rtc_cntl_reg.h"


.bss//Declaracao de variaveis aqui


.text

      .global main
      main://Inicio do codigo (Entry point)
            WRITE_RTC_REG(RTC_GPIO_OUT_W1TC_REG, RTC_GPIO_OUT_DATA_W1TC_S+12, 1,
1)//Desliga o atuador (GPIO2 = LOW)



      loop:

            READ_RTC_REG(RTC_GPIO_IN_REG, RTC_GPIO_IN_NEXT_S+11, 1)//Le o estado
do GPIO0 e guarda no R0
            jumpr on, 1, lt//Se o botao for pressionado (0), ativa o atuador

            WRITE_RTC_REG(RTC_GPIO_OUT_W1TC_REG, RTC_GPIO_OUT_DATA_W1TC_S+12, 1,
1)//Caso o botao nao esteja pressionado, mantem o atuador desligado


      jump loop//retorna ao loop


      on://ativa o atuador e retorna ao loop
            WRITE_RTC_REG(RTC_GPIO_OUT_W1TS_REG, RTC_GPIO_OUT_DATA_W1TS_S+12, 1,
1)//Ativa o atuador (GPIO2 = HIGH)
```

```
    jump loop
```

Vamos analisar novamente pelo analisador lógico, como o sistema se comportou quando o botão de pânico foi pressionado.

Observações:

- Canal 0: PWM de 4 MHz;
- Canal 1: Atuador (buzina) ativo quando em nível lógico ALTO;
- Canal 2: Botão de pânico pressionado quando em nível lógico BAIXO.

Figura 4 - Analisador lógico no código do ULP.

Observe que mesmo durante o evento (pressionar do botão e atuador ativar) o PWM continuou perfeitamente como o esperado (4 MHz), mostrando a eficiência e importância de usar outro núcleo/microcontrolador para ajudar no processamento.

Os co-processadores têm uma trajetória relativamente importante para computação atual, sendo um dos mais famosos a Float Point Unit (FPU), que é um co-processador para efetuar cálculos de ponto flutuante presente na maioria dos dispositivos atuais, inclusive no ESP32. Os co-processadores livram o processador central de alguma tarefa, tornando o sistema, em geral, mais rápido.

O ULP pode ter poucos Mnemônicos (Instruction set limitado), mas se torna importantíssimo em projetos específicos, como nos 2 citados neste artigo. O simples fato de conseguir ler pinos digitais/analógicos e controlar pinos já o torna um aliado interessante para você aprender e utilizar.

Publicado originalmente no Embarcados, no dia 13/04/2018: link para o artigo original, sob a Licença Creative Commons Atribuição-CompartilhaIgual 4.0 Internacional.

Idealização de um projeto IoT portátil

Autor: José Morais

Você já deve ter se questionado como baterias de relógios duram tanto tempo, e que em muitos casos duram vários anos. Então, vamos tentar chegar a este “patamar incrível” com IoT ou até sistemas embarcados? Para isso, serão aplicados desde métodos de baixo consumo em microcontroladores chamados de Sleep Modes, até estratégias de programação que podem diminuir extraordinariamente o consumo do seu projeto portátil.

Nesta série de artigos sobre projeto IoT portátil vamos abordar os conceitos básicos para se ter uma grande economia energética pela parte lógica, o microcontrolador.

Um pouco mais sobre dispositivos portáteis

Se você já se fez a pergunta acima, provavelmente acha fascinante a ideia de algo durar vários anos sem precisar trocar a pilha ou bateria. Isso é muito importante principalmente para o cliente e também para os desenvolvedores, visto que caso seu dispositivo seja um “comilão de baterias”, terá uma má reputação e, consequentemente, as vendas podem ser inferiores do que um dispositivo igual mas com maior economia.

Figura 1 - Bateria SR626 para relógio.

“Como essa bateria minúscula pode durar anos?”. Certamente quem faz projetos portáteis precisa pensar em cada detalhe na questão energética, qualquer descuido pode

arruinar o tempo de vida da sua bateria, sendo necessário a troca inconveniente ou recarregá-la.

Vamos abordar alguns parâmetros para cálculo de autonomia de baterias em nossos projetos:

- T: Tempo (Horas)
- C1: Capacidade da bateria
- C2: Consumo médio do circuito

$$T = \frac{C1}{C2}$$

$$C2 = \frac{C1}{T}$$

$$C1 = T * C2$$

Como citado sobre os relógios, vamos pegar alguns dados aleatórios apenas para verificar as fórmulas:

- Bateria SR626: 28mAh
- Consumo do sistema: ?
- Tempo: 3 Anos

$$C2 = \frac{28}{24 * 365 * 3}$$

$$C2 = \sim 1uA$$

Podemos observar que o consumo médio do relógio para durar aproximadamente 3 anos é 1uA, que é bem pequeno se você comparar com um LED 5mm por exemplo, que é ~15mA. A potência também deve ser levada em consideração, visto que se podemos trabalhar com uma tensão mais baixa e mantendo a corrente, o consumo será menor (P = V * I).

Baixo consumo em microcontroladores

Normalmente em um projeto de IoT ou em um sistema embarcado de forma geral, temos muitos itens além do microcontrolador (MCU) e, mais especificamente para IoT, onde são amplamente utilizadas **comunicações Wireless**, o transmissor é provavelmente o culpado com maior consumo do sistema, juntamente com motores ou similares se for o caso.

Serão abordados nesta série métodos para minimizar este maior consumo do nosso sistema, podendo, assim, elevar a duração da bateria em patamares semelhantes aos do exemplo do relógio visto acima.

Na grande maioria dos microcontroladores existem modos de economia chamados “Sleep Modes” ou similares. Esses modos desligam setores específicos do microcontrolador ocasionando no menor consumo e, assim, aumentando o tempo que a bateria aguentaria.

Com o foco é em IoT, será utilizado o ESP32 WROOM, que é um microcontrolador relativamente potente. Veja algumas de suas características:

- Processador: Dual-Core (Ix6 32-bit) que trabalha de 2 até 240MHz;
- Conexões Wireless: Wi-Fi e Bluetooth (BLE);
- Conexões Wired: SPI, I2C, I2S, SDIO, UART, CAN, ETHERNET...;

- Memória RAM: 520KB;
- Memória FLASH: 4MB.

Um item muito interessante no ESP32 em relação a alguns MCU's desse porte, como PIC32 ou SMT32, é a existência de um terceiro processador chamado ULP (Ultra Low Power coprocessor). Este pequeno "processador" pode ser usado em Deep Sleep para não precisar acordar o MCU para colher dados de sensores ou efetuar comunicações Wired, ocasionando em um consumo médio muito menor.

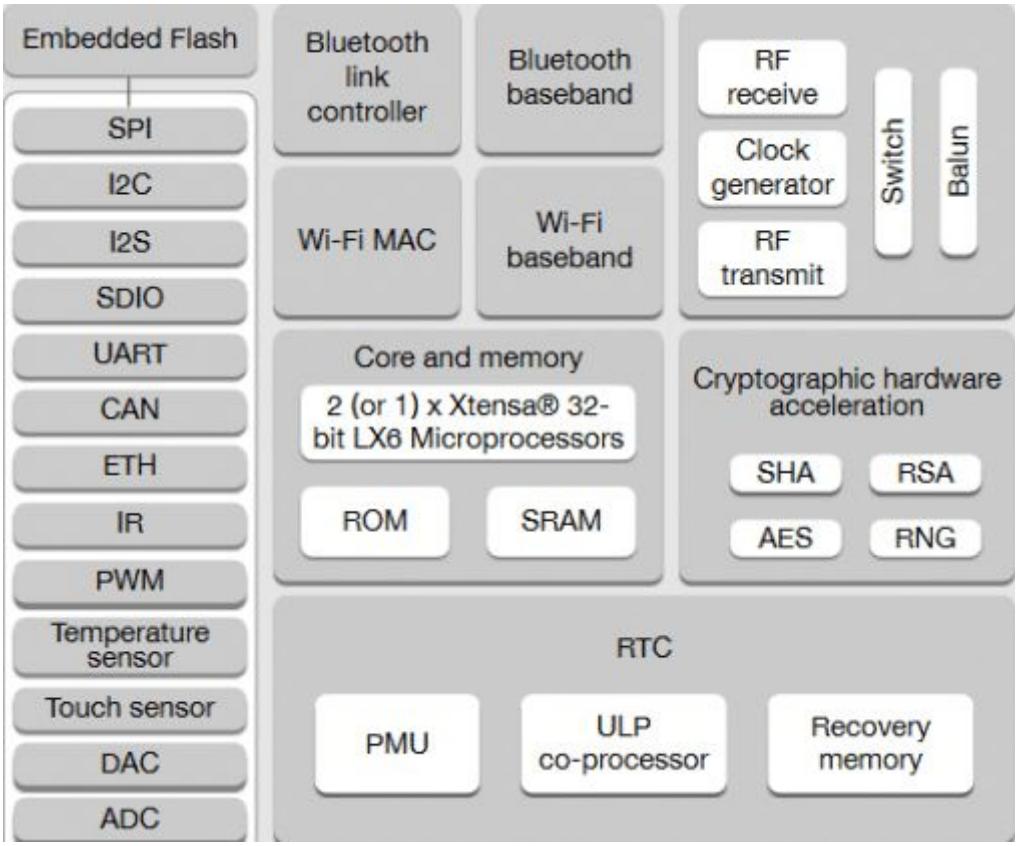


Figura 2 - Seções do ESP32.

O ESP32, em modo normal de transmissão (Tx) ou recepção (Rx), consome:

- **Wi-Fi Ativo (Tx):** 180 - 240mA;

- **Wi-Fi Ativo (Rx):** 95 - 100mA.

Modem Sleep: A parte de transmissão e recepção de dados (Antena WiFi/Bluetooth) é desligada, o consumo varia de acordo com o clock do MCU, que pode ser selecionado (2, 80, 160, 240MHz).

- **Consumo:** 2 - 50mA.

Light Sleep: Apenas o core principal, Digital Domain, está em "pausa", mantendo dados nos registradores e RAM.

- **Consumo:** 800uA.

Deep Sleep: Todo o Digital Domain, incluindo os processadores, são desligados perdendo todos os dados armazenados na RAM e registradores.

- **Consumo:** 10 - 150uA.
- **Obs:** 150uA se aplica ao usar o ULP, caso não, o consumo tende a ~10uA.

Hibernação: São desligados praticamente todos setores do RTC Domain, incluindo as memórias RTC, exceto o RTC Timer e RTC GPIO's, que fazem o MCU acordar.

- **Consumo:** 5uA.

No próximo post da série começaremos a calcular os Sleep's Modes em nosso projeto de IoT portátil, que incluirá desde as contas teóricas para autonomia, até otimizar a programação (Code-Flow) para aumentar o tempo de duração da bateria.

Aplicando conceitos de economia em microcontroladores

Autor: José Morais

Agora que já sabemos calcular a autonomia e o que é sleep em microcontroladores, vamos começar a idealizar e praticar nosso projeto portátil usando conceitos de economia em microcontroladores. A série pretende mostrar os conceitos aplicados para aumentar a duração de baterias, logo, pode ser aplicada em qualquer projeto. Então o projeto que será feito aqui é algo básico apenas para demonstração prática.

Vamos criar um sensor de temperatura e umidade que irá enviar os dados para uma planilha excel online (será nosso banco de dados) e assim vamos conseguir desenhar o gráfico de temperatura/umidade para o determinado local automaticamente.

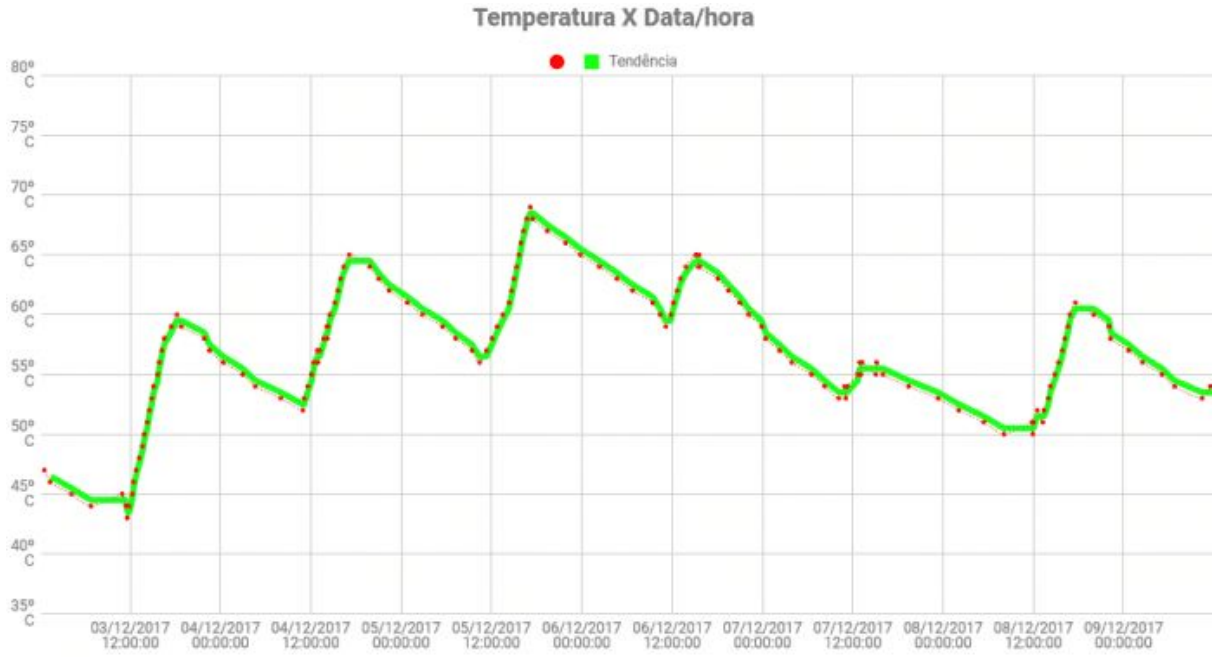


Figura 1: Exemplo de gráfico de temperatura/umidade.

Especulando a duração da bateria

Entendendo o caso: Nosso projeto será apenas um logger de temperatura/umidade que irá ler os dados do sensor a cada 5 minutos. Entre esse tempo o microcontrolador entrará em modo de economia (sleep) para aumentar o tempo de duração da bateria.

O projeto irá consumir uma determinada corrente por um determinado tempo, ou seja, precisamos de outra fórmula para encontrar a corrente (consumo) média do nosso projeto, visto que a fórmula apresentada na primeira parte é uma conta simples.

Ao usar os diversos modos de economia presente no microcontrolador, a corrente média será dada pela média ponderada dos modos (consumo X tempo).

$$Cm = \frac{C1*P1+C2*P2+...Cn*Pn}{P1+P2+...Pn}$$

Para encontrar o Consumo médio (Cm), basta fazer o consumo (Cn) do modo de operação pelo tempo do modo de operação (Pn). Em nossa primeira especulação que irá dar um "norte" ao projeto, serão dois consumos diferentes, que são: Ativo e Deep Sleep.

Consumo médio em transmissão WiFi: 150mA

Consumo médio Deep Sleep: 15uA

Tempo em transmissão: 1 segundo

Tempo em Deep Sleep: 300 segundos

$$Cm = \frac{0.15*1+0.000015*300}{1+300} \; Cm \approx 513uA$$

Obs: Os valores inseridos foram um pouco maiores, admitindo erros do sistema e pensando no "pior caso", visto que ainda iremos calcular o tempo real de processamento, que dará mais precisão à autonomia real, podendo ser menor ou maior.

Agora que temos o consumo médio do projeto, podemos aplicar o consumo para dimensionar a bateria corretamente. Nesse projeto não temos horas mínimas de funcionamento, logo, pretendemos descobrir apenas quanto tempo a bateria já escolhida irá aguentar (para efetuar trocas ou etc). Para isso vamos usar a fórmula indicada na primeira parte da série.

Capacidade da bateria (18650): 3000 mAh

Consumo médio do projeto: 513 uA

$$T = \frac{3}{0.000513}\ T \approx 5848 Horas$$

Para conseguirmos comparar esse tempo, vamos fazer a conta de duração em que não foram aplicados os conceitos apresentados nesta série. Veja na conta acima onde aplicamos Deep Sleep e compare com os valores abaixo, onde não é aplicado nenhum método de baixo consumo (sleep).

Capacidade da bateria (18650): 3Ah

Consumo médio do projeto: 90mA

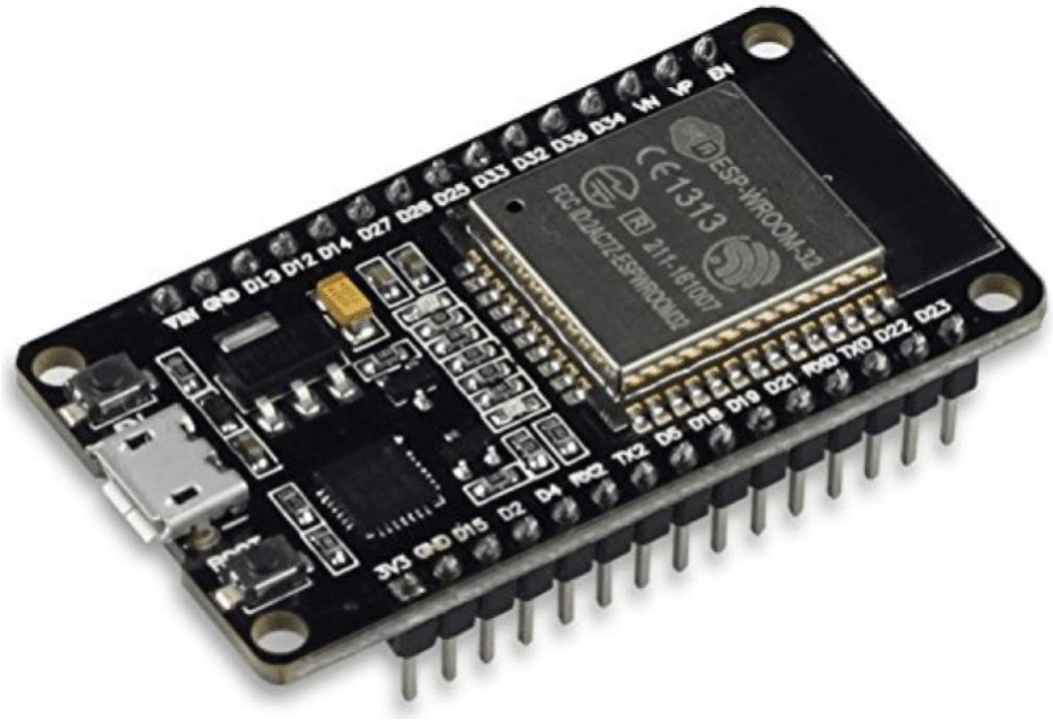
$$T = \frac{3}{0.09}\ T \approx 33 Horas$$

Podemos observar como os modos de sleep aumentaram incrivelmente a duração de bateria, que nesse caso teve um aumento de ~175x, chegando em ~243 dias! Agora você pode estar se perguntando: “Posso aumentar mais a economia?” Sim e será mostrado mais a frente.

Na próxima parte desta série vamos finalmente “botar a mão na massa” programando o ESP32 e montando nosso protótipo para refazer as contas e chegar no consumo real do circuito.

Construindo o projeto de IoT portátil

Autor: José Morais



Finalmente chegamos na parte de começar a dar vida ao projeto IoT portátil. Até aqui você já aprendeu a dimensionar a bateria pro seu projeto e também a calcular o consumo médio. Então vamos prototipar o projeto!

As referências sobre instalação do ESP32 na Arduino IDE e também como usar o Google planilhas para enviar dados do microcontrolador para planilha são importantes.

Já foi citado como o projeto funcionará, mas vamos relembrar. Será um Datalogger que enviará a temperatura e umidade do local para um banco de dados online, usaremos o Google planilhas pois dispensa um servidor para host, gratuito, permite compartilhar os

dados facilmente com qualquer pessoa, aplicar fórmulas, desenhar gráficos e muito mais ao estilo Excel. Vamos programar logo e depois as explicações.

Circuito

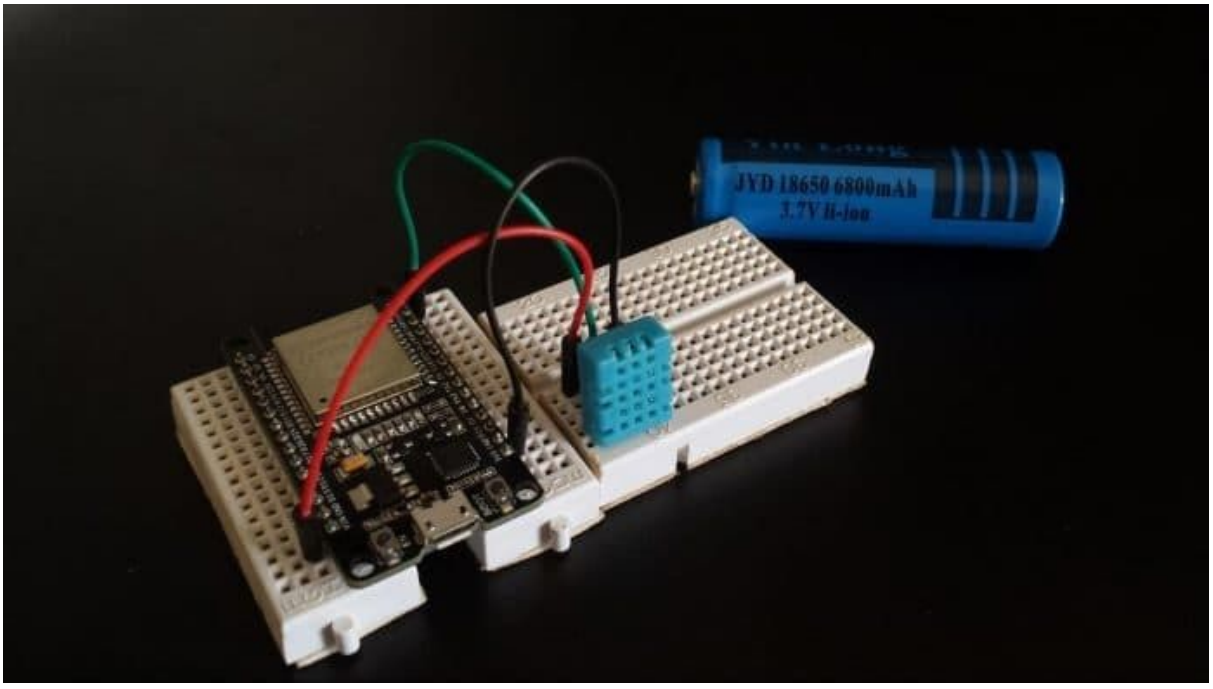


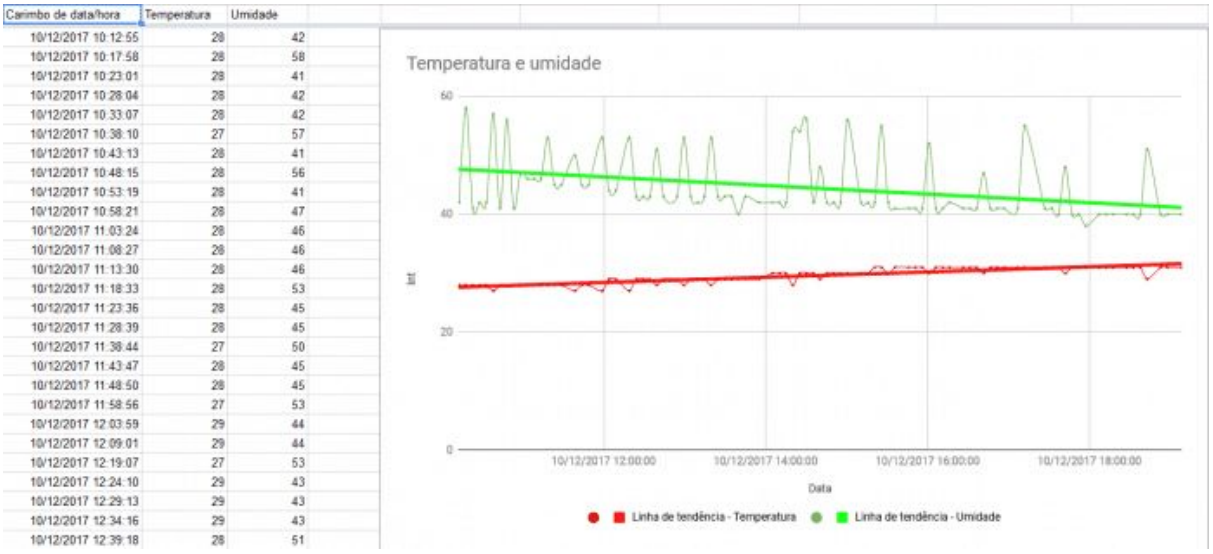
Figura 1 - Protótipo simples do projeto portátil com bateria 18650.

```
#include <DHT.h>
#include <WiFi.h>
#include <WiFiClientSecure.h>

DHT dht(23, DHT11);
WiFiClientSecure client;
float t = 0, u = 0;//Variaveis que armazenam a leitura ATUAL
```

```
RTC_DATA_ATTR float lt = 0;//Variavel alocada na RTC RAM da ultima temperatura lida (Last Temp)

//Variaveis alocadas na RTC RAM não são perdidas entre Deep Sleep, então
//usamos para guardar a ultima temperatura lida e fazer uma nova verificação
//no proximo Wake UP

void setup()
{
        pinMode(23, INPUT);//Pino de dados do DHT11
        dht.begin();//Inicializa o DHT11

        t = dht.readTemperature();//Atribui a temperatura atual na variavel "t"
        u = dht.readHumidity();//Atribui a umidade atual na variavel "u"


        if (lt != t)//Se a ultima temperatura lida for diferente da atual, irá enviar ao banco de dados
        {
            lt = t;//Iguala a ultima temp. com a temp. atual
            WiFi.mode(WIFI_STA);//
            WiFi.begin("SUA REDE", "SUA SENHA");//Conecta no WiFi

            for (int i = 0; i < 500; i++)
            {
                delay(10);
                if (WiFi.status() == WL_CONNECTED)//Se conseguir conectar no WiFi
                {
                    if (client.connect("docs.google.com", 443) ==
```

```
true)//Se conseguir conectar no servidor da Google
                        {
                            String        toSend        =        "GET
/forms/d/e/1FAIpXXXf6EKACSqEhAsXXXKb3qDBiNSh6MXn6ck44TBj7zRYH72SXXX/
formResponse?ifq";
                            toSend += "&entry.634150418=";   toSend
+= t;
                            toSend += "&entry.2106983911=";  toSend
+= u;
                            toSend += "&submit=Submit HTTP/1.1";

                            client.println(toSend);
                            client.println("Host: docs.google.com");
                            client.println();
                            client.stop();
                        }

                        break;//Encerra o loop FOR()
                    }
                }
            }


            ESP.deepSleep(300000000);//Dorme por 5 minutos

}

void loop()
{

}
```

Explicação



Carimbo de data/hora	Temperatura	Umidade
10/12/2017 10:12:55	28	42
10/12/2017 10:17:58	28	58
10/12/2017 10:23:01	28	41
10/12/2017 10:28:04	28	42
10/12/2017 10:33:07	28	42
10/12/2017 10:38:10	27	57
10/12/2017 10:43:13	28	41
10/12/2017 10:48:15	28	56
10/12/2017 10:53:19	28	41
10/12/2017 10:58:21	28	47
10/12/2017 11:03:24	28	46
10/12/2017 11:08:27	28	46
10/12/2017 11:13:30	28	46
10/12/2017 11:18:33	28	53
10/12/2017 11:23:36	28	45
10/12/2017 11:28:39	28	45
10/12/2017 11:38:44	27	50
10/12/2017 11:43:47	28	45
10/12/2017 11:48:50	28	45
10/12/2017 11:58:56	27	53
10/12/2017 12:03:59	29	44
10/12/2017 12:09:01	29	44
10/12/2017 12:19:07	27	53
10/12/2017 12:24:10	29	43
10/12/2017 12:29:13	29	43
10/12/2017 12:34:16	29	43
10/12/2017 12:39:18	28	51

Figura 2 - Dados e gráfico gerado pelo projeto com DHT11.

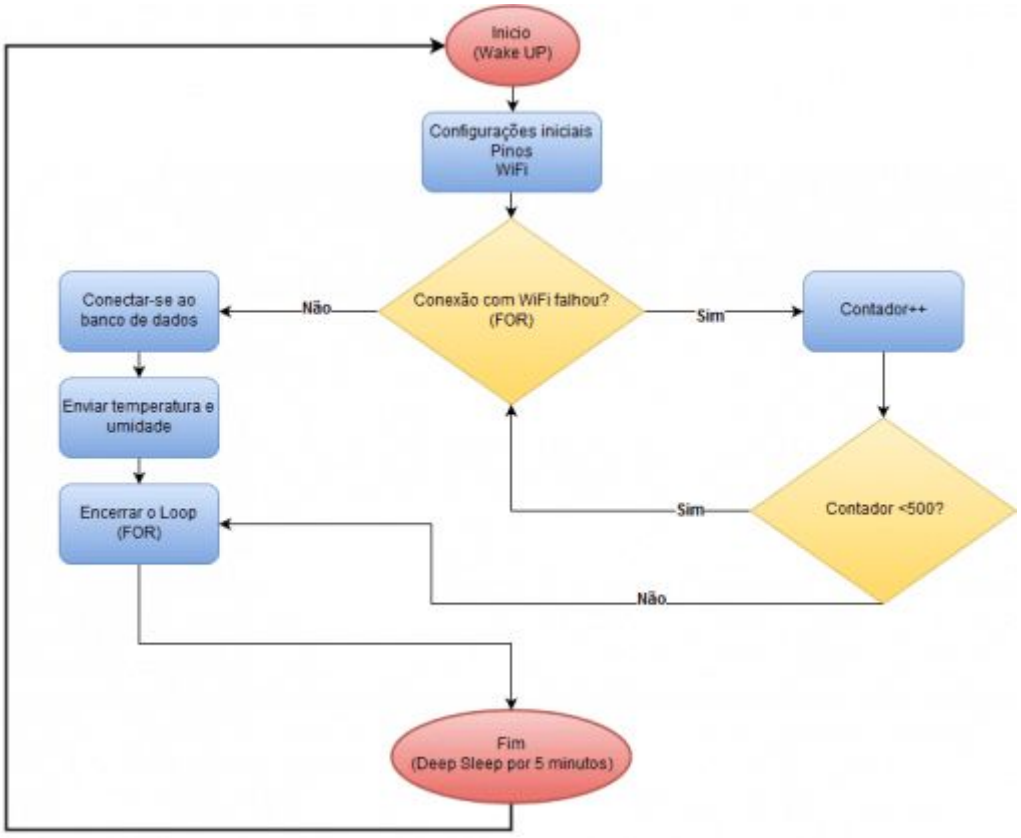


Figura 3 - Fluxograma do projeto.

Observações sobre os dados:

- **Intervalo entre leituras:** 5 minutos;
- **Total de leituras:** 92;
- **Tempo total de colheita:** 9 horas
- Apesar das leituras de umidade oscilam bastante por motivos desde protoboard até sensor ruim, com a linha de tendência linear foi possível notar a redução da umidade conforme o aumento da temperatura.

Como em todo projeto de sistemas embarcados, as condições do ambiente a ser monitorado podem influenciar no funcionamento, dando margem a melhorias de desempenho (e, nesse caso, consumo energético). Observando o ambiente cuja temperatura foi monitorada, nota-se que a temperatura do local demorou aproximadamente 2 horas para variar 1°C e, neste meio tempo, foram enviados 24

valores iguais de temperatura. Para este projeto, estes 24 valores iguais de temperatura significam 24 envios de um mesmo valor de temperatura, levando à conclusão que estes envios causaram grande consumo energético desnecessário (uma vez que o grande "vilão" do consumo no projeto é a transmissão de dados por Wi-Fi, conforme visto na parte 1 desta série). Logo, poderíamos enviar apenas valores diferentes de temperatura, diminuindo, assim, o uso do Wi-Fi e, por consequência, aumentando a economia da bateria, nesse caso para 24x.

Vamos fazer algumas contas para dar um norte a esta nova informação, que deixará o ESP32 dormindo por 2 horas:

- **Capacidade da bateria:** 3000mAh;
- **Consumo médio com 2 horas de sleep:** 36uA;
- **Obs:** A transmissão foi feita em média a cada 2 horas, portanto o ESP32 dormiu 2 horas e depois enviou.

Consumo médio com 2 horas de sleep:

$$Cm = \frac{0.15*1+0.000015*7200}{1+7200} \; Cm \approx 36uA$$

Tempo de duração da bateria com 2 horas de sleep:

Tempo de duração da bateria com 2 horas de sleep:

$$T = \frac{3}{0.000036} \; T = 83333Horas$$

Veja que para este caso, com 2 horas de intervalo entre transmissões de dados, a bateria duraria (teoricamente) 9 anos e 200 dias! Incrível não?! Mas nem tudo são flores, todas contas desse projeto até agora foram apenas para dar um norte, excluindo todos componentes extras do circuito. O tempo real de execução do código ainda será

calculado e o consumo provavelmente será bem maior por conta dos componentes extras no circuito.

Vamos começar a fazer alguns ajustes de tempo e consumo para que no próximo post seja calculado o tempo real de duração para nosso projeto.

Foi observado o tempo em que cada ação demora para ocorrer, separado por partes:

- **Conexão com Wi-Fi:** 2.5 segundos (por causa do Wi-Fi - SCAN);
- **Comunicação com a planilha:** 1.7 segundos;
- **Total:** 4.26 segundos.

Agora precisamos do real consumo do circuito, que anteriormente foi levado em consideração apenas o ESP32. Entretanto, há um componente a mais, o DHT11. De acordo com o Datasheet do DHT11, o consumo é:

- **Em medições:** 300uA;
- **Em Standby (sleep):** 60uA.

As medições do DHT11 podem demorar até ~25mS, o restante do tempo ele se encontra em Standby. Logo, vamos arrumar o último cálculo de consumo do projeto (Figura 4).

Consumo médio do projeto que se aproxima a realidade:

$$Cm = \frac{0.15*4.26+0.000075*7200}{4.26+7200}\ Cm \approx 164uA$$

Tempo de duração da bateria que se aproxima à realidade:

$$T = \frac{3}{0.000164}\ T \approx 18293 Horas$$

Veja que arrumando as contas com novos tempos e componentes, o consumo ficou 4.5x maior que os anteriores, que eram apenas simulações para nos guiar. Entretanto, a bateria ainda irá durar ~2 anos, que é um bom intervalo.

Observações importantes em relação aos tempos do ESP32

1. A conexão com Wi-Fi pode chegar até ~300mS, entretanto, como o ESP32 é reiniciado totalmente, é preciso configurar toda pilha Wi-Fi novamente. E o ESP32 faz o uso da LWIP, que não é muito otimizada em velocidade, e algumas mudanças, como IP fixo, podem melhorar esses tempo;
2. Se você precisa de uma inicialização do Deep Sleep mais rápida, há inúmeras configurações que permitem isso, por exemplo, a calibragem do RTC_SLOW_CLK que leva por padrão 1024 ciclos do XTAL principal (~2.5uS) e o tempo extra de inicialização da FLASH (~2mS);
3. Você pode notar alguma perda de "temporização" ao usar longos períodos de Deep Sleep, isso acontece pois o cristal do RTC é mais instável que o cristal principal. Você pode melhorar a precisão configurando uma calibragem maior (como citado na Obs 2), que é útil em Deep Sleep's de longo período.
4. A conexão com a planilha é relativamente lenta, pois a conexão ao servidor do Google planilhas é feita por SSL/TLS (HTTPS) e isso necessita de um processamento maior e maior número de trocas de mensagens entre cliente e servidor. Logo leva a um tempo de Wi-Fi ativo maior do que se comparado ao caso de uso de HTTP "simples" ou MQTT, por exemplo.

Nosso projeto finalmente está vivo e pronto, mas existe algo a mais que podemos fazer para melhorar o consumo e aumentar ainda mais a duração da bateria, caso seu projeto precise de extrema economia: **estratégias de programação**. Vamos melhorar a eficiência deste código apresentado, incluindo verificações de dados e um modo de economia a mais, o Modem Sleep.

Referências

1. Instalando ESP32 na Arduino IDE
2. Google planilhas para debug
3. Datasheet do DHT11

Estratégias de programação para portáteis

Autor: José Morais

No projeto montado anteriormente há algumas melhorias que podemos fazer. Nesse caso será otimizar o código a partir da verificação de dados e, assim, adicionar o **Modem Sleep.**

Vamos observar o código e pensar um pouco...É fácil perceber onde pode ser melhorado e será exatamente onde o consumo é maior, na transmissão de informações pela internet

(uso do WiFi). Ao adicionar uma verificação de dados, podemos ignorar dados irrelevantes e, assim, prolongar a vida da nossa bateria. Veja o que será feito:

- Adicionar a verificação da temperatura para enviar uma nova temperatura **apenas** se esta for diferente da última enviada, poupando, assim, envios desnecessários ao banco de dados e consumo desnecessário de energia com a transmissão WiFi;
- Na verificação de temperatura, a conexão com a internet não é necessária, então iremos ligar o WiFi **apenas** quando a verificação acima for verdadeira. Neste meio tempo o ESP32 ficará com o WiFi OFF (Modem Sleep).

Anteriormente, melhoramos o consumo aumentando o intervalo de sleep, entretanto a temperatura pode variar bem mais rapidamente dependendo do local. Logo, dormir por 2 horas não é interessante. Agora vamos melhorar o código e depois as explicações!

Código do projeto pode ser visto abaixo:

```
#include <DHT.h>
#include <WiFi.h>
#include <WiFiClientSecure.h>

DHT dht(23, DHT11);
WiFiClientSecure client;
float t = 0, u = 0;//Variaveis que armazenam a leitura ATUAL

RTC_DATA_ATTR float lt = 0;//Variavel alocada na RTC RAM da ultima
temperatura lida (Last Temp)

//Variaveis alocadas na RTC RAM não são perdidas entre Deep Sleep,
então
//usamos para guardar a ultima temperatura lida e fazer uma nova
verificação
//no proximo Wake UP
```

```
void setup()
{
          pinMode(23, INPUT);//Pino de dados do DHT11
          dht.begin();//Inicializa o DHT11

          t = dht.readTemperature();//Atribui a temperatura atual na
variavel "t"
          u = dht.readHumidity();//Atribui a umidade atual na variavel
"u"


          if (lt != t)//Se a ultima temperatura lida for diferente da
atual, irá enviar ao banco de dados
          {
               lt = t;//Iguala a ultima temp. com a temp. atual
               WiFi.mode(WIFI_STA);//
               WiFi.begin("SUA REDE", "SUA SENHA");//Conecta no WiFi

               for (int i = 0; i < 500; i++)
               {
                    delay(10);
                    if (WiFi.status() == WL_CONNECTED)//Se conseguir
conectar no WiFi
                    {
                         if (client.connect("docs.google.com", 443) ==
true)//Se conseguir conectar no servidor da Google
                         {
                              String toSend = "GET
/forms/d/e/1FAIpXXXf6EKACSqEhAsXXXKb3qDBiNSh6MXn6ck44TBj7zRYH72SXXX/for
mResponse?ifq";
                              toSend += "&entry.634150418="; toSend
+= t;
                              toSend += "&entry.2106983911="; toSend
+= u;
                              toSend += "&submit=Submit HTTP/1.1";
```

```
                    client.println(toSend);
                    client.println("Host: docs.google.com");
                    client.println();
                    client.stop();
                }

                break;//Encerra o loop FOR()
            }
        }
    }


    ESP.deepSleep(300000000);//Dorme por 5 minutos


}

void loop()
{

}
```

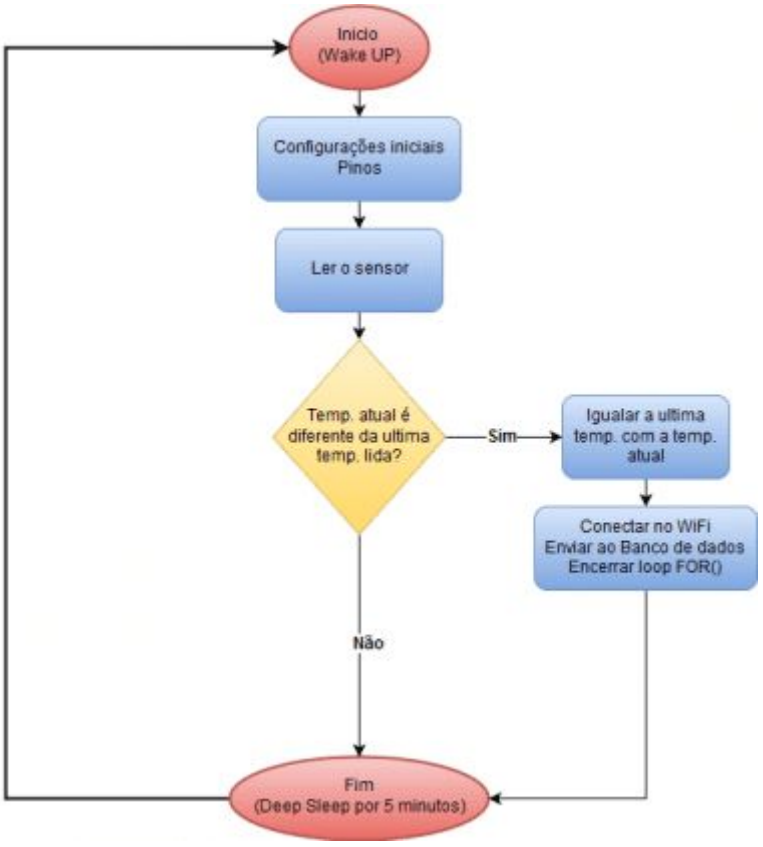


Figura 1 - Novo fluxograma do projeto IoT portátil.

O fluxograma está mais simplificado na parte de envio e conexão com WiFi (se comparado ao anterior), mas a ideia é apenas mostrar a condicional que verifica a última temperatura lida com a atual. Como vimos anteriormente, nos dados gerados pelo sensor, a temperatura se alterou 1°C a cada 2 horas, logo, percebemos que a "**seta não**" da condicional ocorrerá 24x mais que a "**seta sim**", uma economia bem grande já que o maior consumo está no WiFi.

Foi feita uma nova medição do tempo de processamento para melhorar os cálculos e chegar a um novo e último consumo médio do projeto:

- **Consumo médio em transmissão (WiFi):** 150 mA;
- **Consumo médio ao ler o sensor:** 50 mA (clock a 240 MHz);

- **Consumo médio em Deep Sleep:** 75 uA (15 uA + 60 uA do DHT11);
- **Tempo em transmissão (WiFi):** 4,6 segundos;
- **Tempo lendo o sensor:** 23 ms;
- **Tempo em Deep Sleep:** 5 minutos.

Entretanto, devemos lembrar que a transmissão dos dados é ~24x menos frequente que a leitura do sensor. Logo, vamos dividir alguma das variáveis relacionadas com a transmissão por 24, iremos escolher o consumo em Deep sleep.

Consumo real do projeto IoT Portable:

$$Cm = \frac{(\frac{0.15}{24}) * 4.26 + 0.05 * 0.023 + 0.000075 * 300}{4.26 + 0.023 + 300} \; Cm \approx 172uA$$

Duração em horas do projeto (765 dias):

$$T = \frac{3}{0.000172} \; T \approx 17442Horas$$

Você pode estar perguntando por que o consumo com as estratégias de programação deu um consumo maior (172uA) do que o último cálculo feito no artigo anterior (164uA). Isso se deve ao fato de que lá foi usado Deep sleep de 2 horas e, neste tempo, perdemos muitas leituras, inviabilizando vários tipos de projetos. Agora foi feito com Deep sleep de 5 minutos (24x mais frequente) e mesmo assim, podemos resumir que o consumo foi igual, mostrando como um código bem escrito e manipulado pode ser mais econômico do que usar sleep's frequentes ou com tempo longo.

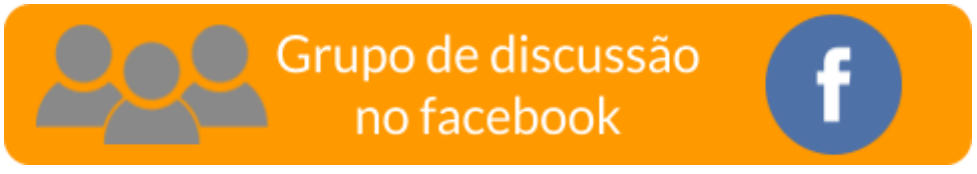
Observação importante: O consumo poderia ser facilmente reduzido aproximadamente 20-100x diminuindo a velocidade do clock, trocando por um sensor melhor e mais rápido e até usando o ULP.

Conclusão

Agora que nossa série chegou ao fim, podemos perceber a imensa importância de uma programação bem feita e principalmente os métodos de Sleep para o mundo de IoT, Sistemas embarcados e Wearables que fazem uso de baterias.

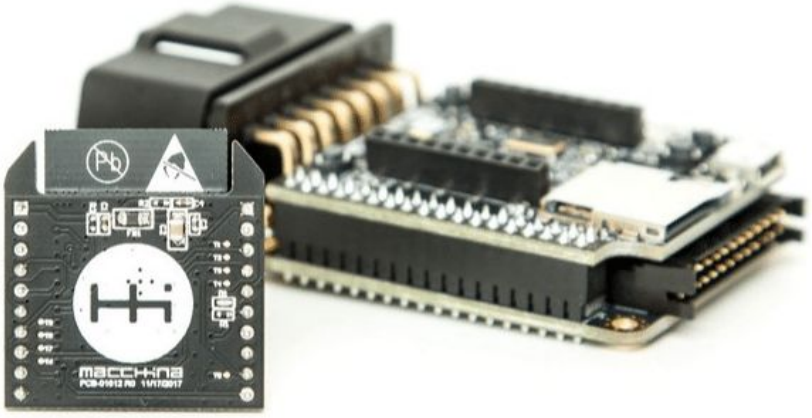
Aplicando os conceitos mostrados nesta série, você conseguirá criar inúmeros dispositivos portáteis que durem anos em uma barata e pequena bateria, o limite é sua imaginação.

Os conceitos mostrados aqui são aplicáveis a qualquer microcontrolador, desde PIC's até ARM's, diferenciando basicamente os consumos e nomes de Sleep's.



Conheça o SuperB - Módulo ESP32 compatível com Xbee

Autor: Fábio Souza



O módulo SuperB, possui form factor XBee da DIGI e vem com o ESP32-WROOM-32, possibilitando adicionar Wi-Fi e Bluetooth aos hardwares compatíveis com Xbee já existentes.

O módulo foi lançado em uma campanha de financiamento coletivo no Crowd Supply, com o valor de $23.

O projeto é open source e possui as seguintes características:

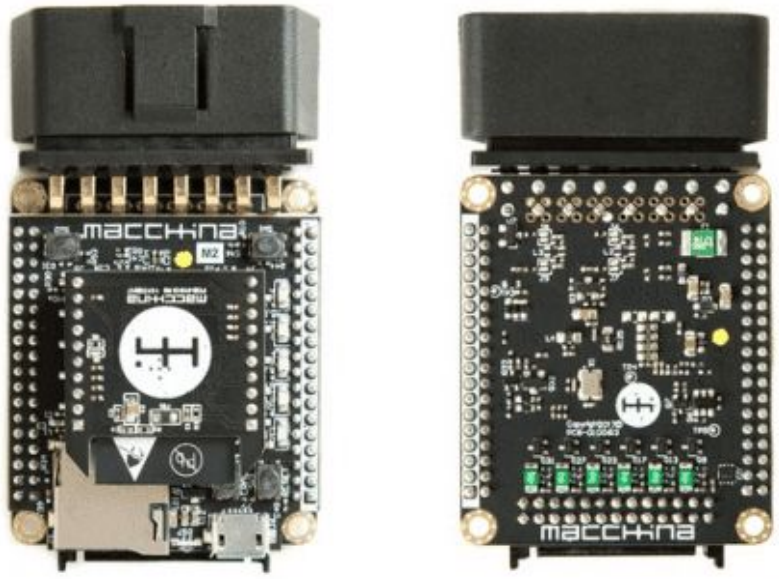
- Compatibilidade com o form-factor XBee (3.3 V, UART, SPI e GPIOs disponíveis nos headers)
- Baseado no ESP32-WROOM-32 que possui:
  - ESP32 dual core WiSoC @ 240 MHz
  - 4MB de flash
  - Conectividade– 802.11 b/g/n WiFi e Bluetooth Classic + LE
- Programável com muitas ferramentas, incluindo Arduino IDE
- OTA (Over-the-Air)
- Corrente em sleep menor que 5 µA
- Totalmente certificado com antenas integradas e pilhas de software

Vídeo da campanha do SuperB no Crowd Supply

O SuperB foi desenvolvido pela Macchina, a empresa por trás da plataforma de desenvolvimento M2, que é um conector OBD2 de código aberto e placa de desenvolvimento.

Os usuários do M2, solicitaram um módulo compatível com ESP32 para viabilizar os seus projetos, diagnósticos, etc.

O M2 também está disponível para compra junto a campanha do SuperB:



Características do M2

- Soquete padrão Xbee
- 12 V I/O - UART, SPI, I2C, Analógica, etc.

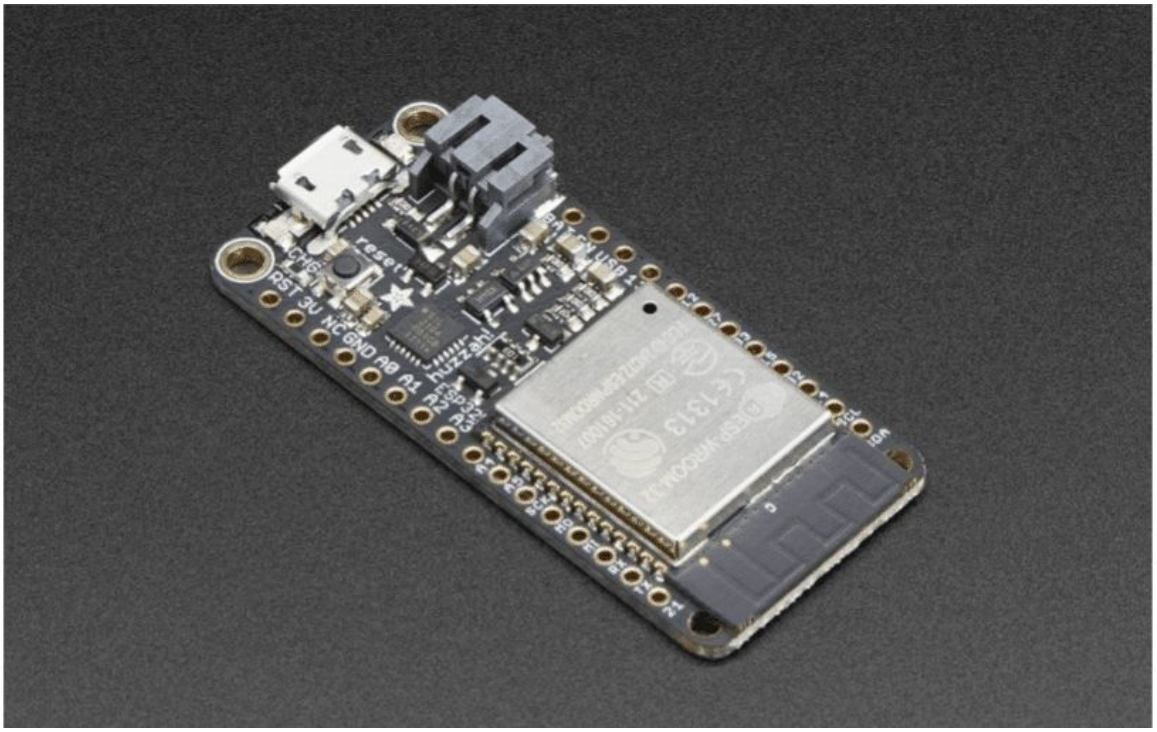
- 6x LEDs um RGB)
- 2x Chave táctil
- 2x transceptor CAN
- 2x transceptor LIN/K-LINE
- Single wire CAN
- J1850 VPW/PWM
- EEPROM
- MicroSD
- Micro-USB
- Programável via Arduino IDE, Simulink, C ++, etc.
- Dimensão: 56.4 mm x 40.6 mm x 15.7 mm

Vídeo de funcionamento do M2 com SuperB

Confira todos os detalhes da campanha e as recompensas no Crowd Supply. O envio está previsto para abril de 2019.

Adafruit HUZZAH32 – ESP32 Feather Board

Autor: Giovanni Cerqueira



A placa

A Adafruit Feather é uma linha de placas de desenvolvimento criada pela Adafruit cujos principais diferenciais são a flexibilidade, a portabilidade e a leveza. Hoje estarei apresentando o modelo HUZZAH32 Feather Board. Farei uma introdução sobre ela, explicando suas características, e realizei um projeto com ela.

Características da Adafruit HUZZAH32

A Adafruit Feather HUZZAH32 é baseada no módulo WROOM32, fabricado pela Espressif Systems. O módulo contém um chip ESP32 dual-core, 4 MB de SPI Flash e antena sintonizada. O ESP32, além de muito poderoso, possui suporte para Wi-Fi e Bluetooth, tanto Clássico quanto Low Energy (LE). O microcontrolador presente no módulo é o Tensilica LX6. Isso faz da Feather HUZZAH32 uma placa excelente para aplicações wireless e de IoT. A lista de especificações técnicas mais avançadas do ESP32 podem ser encontradas aqui.

Pinagem

Uma das vantagens mais expressivas do ESP32 em relação ao ESP8266 é a maior quantidade de GPIOs. A Feather32 possui 13 portas analógicas e 11 GPIOs. Temos também os pinos TX e RX para comunicação serial e pinos SCL e SDA para a utilização de dispositivos I2C e SPI, que não devem faltar.

Há várias maneiras de se fornecer energia à Feather. Além da entrada micro USB, temos o conector JST, através do qual é possível alimentá-la com uma bateria LiPo 3,7/4,2 V. Como se já não bastasse, ainda temos o pino BAT (ligado ao conector JST).

Importante: O ESP32 trabalha com 3,3V. Utilizá-lo com tensão superior pode danificar a placa.

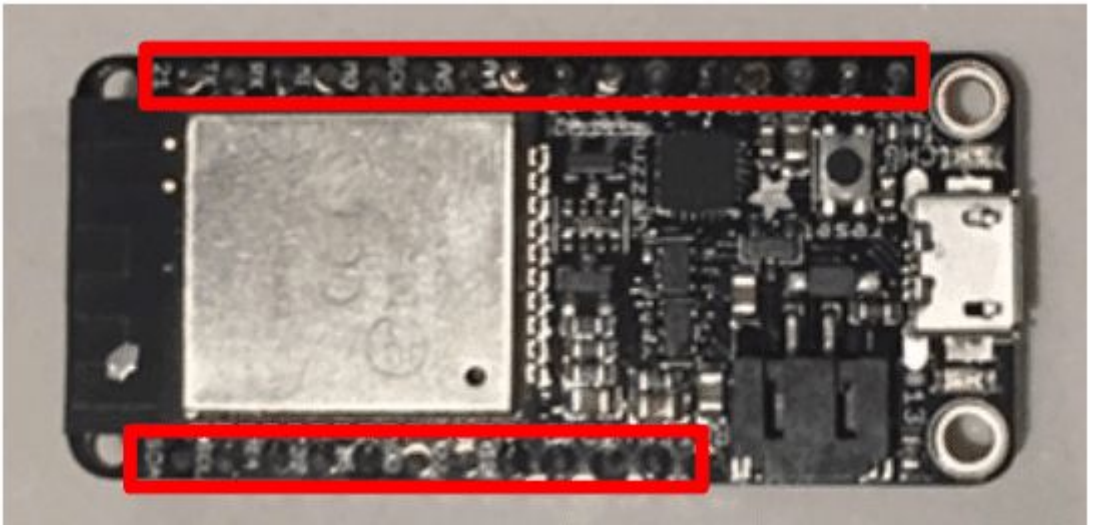


Figura 1 - Pinos da Placa Adafruit HUZZAH32.

Considerações Finais

Como o intuito deste artigo era apenas fazer a apresentação da placa, deixarei a parte da realização efetiva de projetos para um próximo artigo, já que a instalação da Adafruit HUZZAH32 e a sua configuração é um processo um tanto quanto complicado e laborioso. Portanto, este artigo vai ficando por aqui. Também gostaria de ressaltar que a HUZZAH32 é focada em desenvolvedores, possuindo defeitos e bugs que ainda estão sendo corrigidos pela comunidade. Portanto, não a recomendo para quem ainda está começando, sendo pouco amigável a makers de primeira viagem. Espero ter trazido informações relevantes a respeito da placa, transmitindo conhecimento útil. Em breve, estarei ensinando a programação e montagem de projetos com ela.

Considerações Finais

Os artigos são escritos pela comunidade, isso mesmo, **você pode contribuir com artigos para o Embarcados**. Veja como é fácil: Seja Colaborador.

Somos uma grande comunidade que conversa diariamente sobre assuntos relacionados à área, compartilhando conhecimento de forma online e offline.

Participamos de diversos eventos e também realizamos eventos com o foco em desenvolvimento de Sistemas Embarcados

Para aproximar os integrantes da comunidade temos 3 canais de comunicação:

- Comunidade Embarcados no Facebook
- Comunidade Embarcados no LinkedIn
- Comunidade Embarcados no Telegram

Convidamos você a compartilhar conhecimento sobre eletrônica e sistemas embarcados

Participe da comunidade Embarcados, compartilhe e aprenda muito!

Se ainda não é registrado no site, aproveite e faça seu cadastro para receber os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

Caso você tenha encontrado algum problema no material ou tenha alguma sugestão, por favor, entre em contato conosco. Sua opinião é muito importante para nós: contato@embarcados.com.br

Siga o Embarcados na Redes Sociais

 https://www.facebook.com/osembarcados/

 https://www.instagram.com/portalembarcados/

 https://www.youtube.com/embarcadostv/

 https://www.linkedin.com/company/embarcados/

 https://twitter.com/embarcados