**Logbook**

**MAN4HEALTH**

****

*Construção e parametrização dos nós de medida IoT*

**Julho 2022**

Índice

[1. Construção nó IOT com ATMega328 3](#_Toc110244814)

[1.1. Programação do ATMega328 (bare metal) com Arduino IDE 5](#_Toc110244815)

[1.1.1. Comunicação série: 8](#_Toc110244816)

[1.1.2. Observação importante 8](#_Toc110244817)

[1.2. Firmware 8](#_Toc110244818)

[1.2.1. Watchdog timer 9](#_Toc110244819)

[Listagem: ModosWDTATmega328\_ex1 10](#_Toc110244820)

[Listagem: ModosWDTATmega328\_ex2 11](#_Toc110244821)

[Listagem: ModosWDTATmega328\_ex2 12](#_Toc110244822)

[1.2.2. Modos de sleep 13](#_Toc110244823)

[1.2.3. Referências 20](#_Toc110244824)

[1.2.4. Conversão A/D 21](#_Toc110244825)

[1.2.5. Parametrização da Gateway Dragino DSLO8 25](#_Toc110244826)

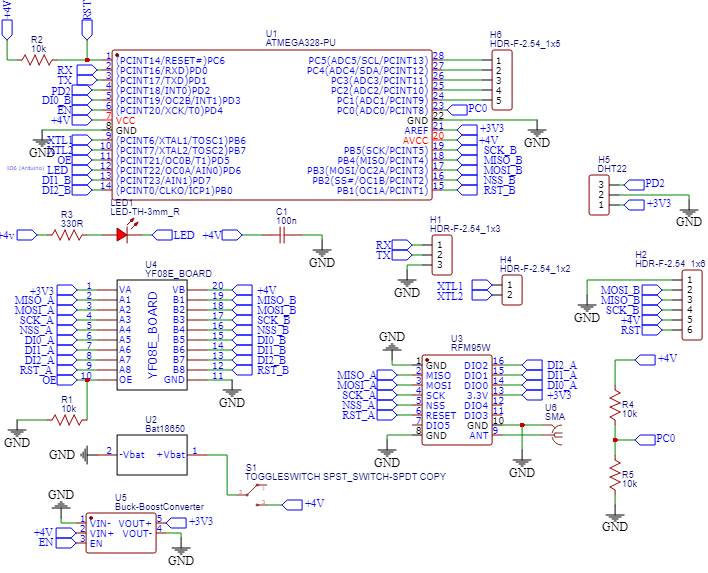
[1.2.6. Alteração da biblioteca LMIC (acabei por não fazer isso) 29](#_Toc110244827)

[2. Modelo de dados 30](#_Toc110244828)

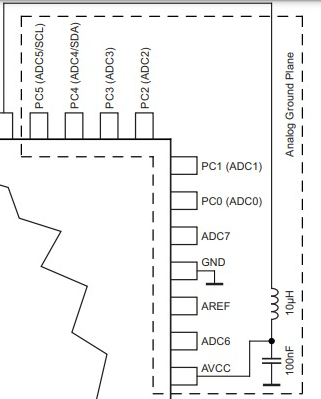
[3. Operação da estação meteorológica SenseCAP 33](#_Toc110244829)

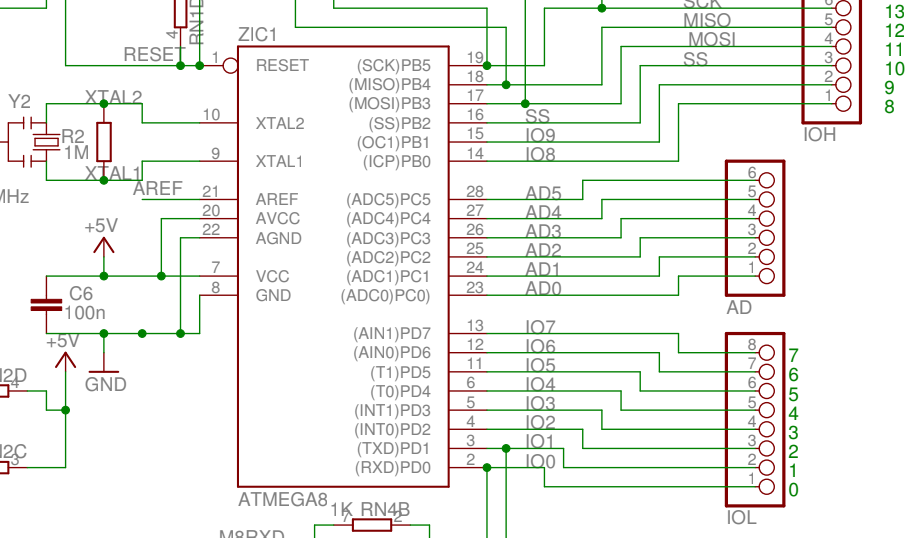
# Construção nó IOT com ATMega328

Nesta secção documenta-se o nó IoT feito em torno do microcontrolador ATMega328. O esquemático apresenta-se em baixo.

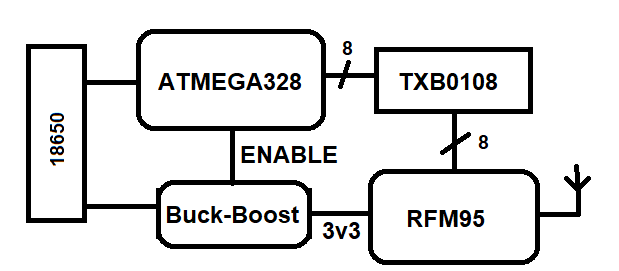


Nota: Em relação ao AVCC, as folhas de dados aconselham a utilização de uma rede RC como se mostra em baixo. No entanto, para já, considera-se apenas que VCC e AVCC são idênticos.



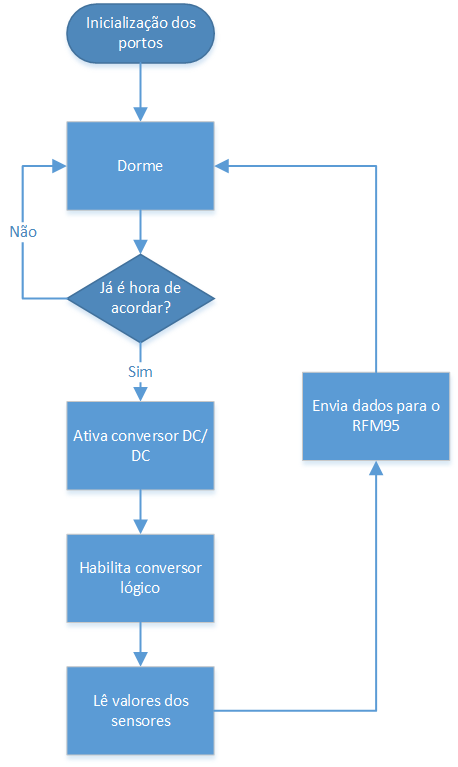


Em termos de funcionamento, este pode ser descrito com base no seguinte diagrama de blocos:



O microcontrolador corre com base no oscilador interno de 8MHz, sem *bootloader* e alimentado diretamente por uma bateria de iões de lítio do tipo 18650. A tensão nominal dessa bateria anda em torno dos 4V e será regulada para 3V3 para alimentar o RFM95. De modo a adaptar os valores lógicos do microcontrolador com os do transceiver LoRa, é usado um conversor de nível lógico bidirecional, TXS0108E, baseado no integrado TCB0108 da *Texas Instruments*. A forma como o ATMega328 é parametrizada e programada será descrito na seção que se segue.

Do ponto de vista operacional, o firmware desempenha um conjunto de tarefas que se resume no seguinte fluxograma:



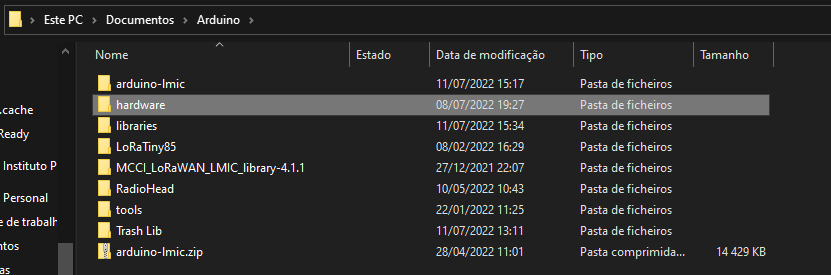
O firmware, assim como todos os detalhes operacionais, serão descritos adiante.

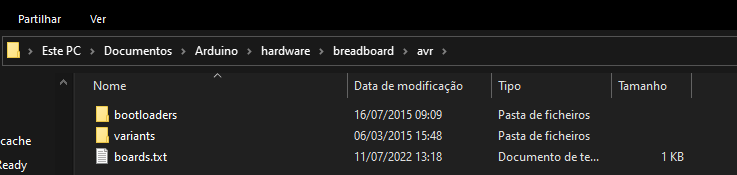
## Programação do ATMega328 (bare metal) com Arduino IDE

Esta secção foi adaptada deste [site](https://www.hackster.io/techmirtz/arduino-without-external-clock-crystal-on-atmega328-d4fcc4) e pretende descrever a forma como o ATMega328 é parametrizado no Arduino IDE e programado recorrendo ao USBASP V2.0. Neste caso, o microcontrolador não utilizará o cristal externo, mas sim o oscilador RC interno. Em particular, a frequência de relógio do microcontrolador neste caso será igual a 8 MHz.

Para isso, começa-se por instalar a biblioteca para programação do microcontrolador seguindo as etapas enumeradas a seguir:

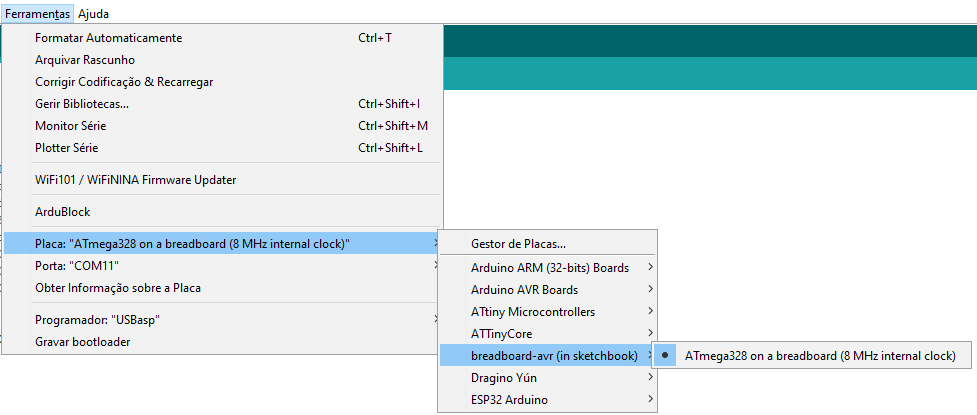
* Descarregar, dependendo do IDE que se está a utilizar, o arquivo de configuração de hardware: [breadboard-1-6-x.zip](https://www.arduino.cc/en/uploads/Tutorial/breadboard-1-6-x.zip) (instalei esta), [Breadboard1-5-x.zip](https://www.arduino.cc/en/uploads/Tutorial/breadboard-1-6-x.zip) ou [Breadboard1-0-x.zip](file:///C:\Users\jpcoe\Seafile\Minha%20Biblioteca\Logbooks\IoT\Breadboard1-0-x.zip);
* Criar uma subpasta "hardware" na pasta do Arduino (em Documentos) e mova a pasta breadboard-1-6-x para essa pasta.





* Reiniciar o software Arduino.

Depois desta etapa, é possível observar esse novo hardware disponível no Arduino IDE:



Para ensaiar toda a cadeia de programação, despois da biblioteca de hardware instalado, realizar as ligações entre o programador [USBASP](https://comink.pt/index.php?id_product=2504&controller=product) e o microcontrolador conforme se mostra a seguir:

#define LED 6

void setup() {

pinMode(LED, OUTPUT);

}

void loop() {

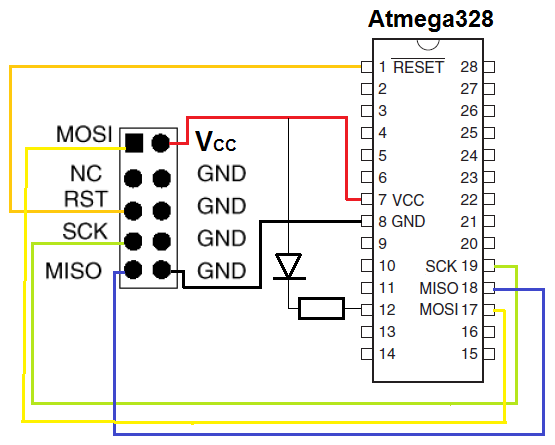
digitalWrite(LED, HIGH);

delay(1000);

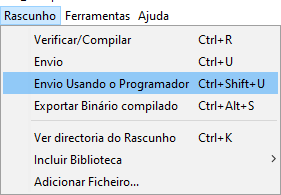
digitalWrite(LED, LOW);

delay(1000);

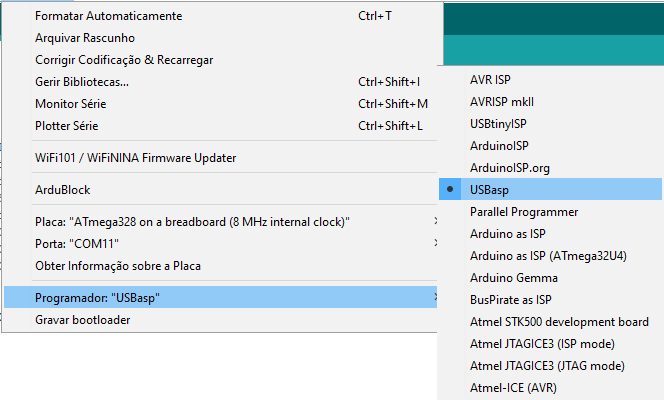
}



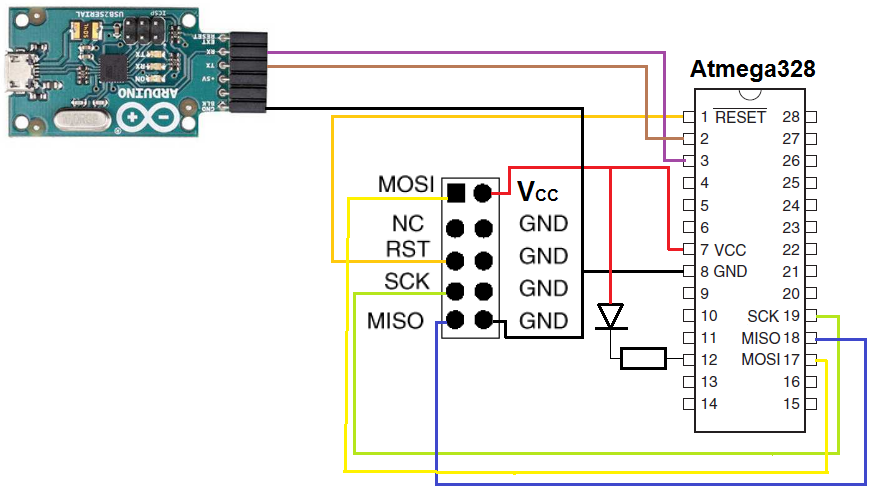
Depois destas ligações realizadas, e do programa estar escrito no Arduino IDE, a programação deverá ser realizada acedendo ao menu Rascunho/Envio Usando o Programador:



Garantindo que a opção correta do programador se encontra selecionada:



### Comunicação série:



#define LED 6

void setup() {

Serial.begin(9600);

pinMode(LED, OUTPUT);

}

void loop() {

digitalWrite(LED, HIGH);

delay(1000);

Serial.println("Desliga");

digitalWrite(LED, LOW);

delay(1000);

Serial.println("Liga");

}

### Observação importante

Para realizar estes ensaios utilizei um microcontrolador DIL que vinha numa placa Arduino Uno. Removi-o da placa e coloquei-o numa *breadboard*. Realizei as ligações conforme se descreveu anteriormente. Quando tentei programar o microcontrolador, verifiquei que o programador não conseguia transferir o programa nem sequer comunicar com o microcontrolador. Só depois me recordei que este se encontrava com os *fuses* definidos para funcionar com o oscilador de cristal. Deste modo, para conseguir programar o microcontrolador a primeira vez tive que acrescentar o cristal de 20MHz e dois condensadores de 22pF para desacoplamento. Outra alternativa seria programar o microcontrolador antes de o tirar da placa de desenvolvimento usando o conetor ICSP.

## Firmware

Nesta seção apresentam-se os detalhes para o desenvolvimento do firmware que será usado no microcontrolador. Começa-se por fazer um apanhado da forma como o microcontrolador é colocado em suspensão (sleep) e o papel que o watchdog timer tem nesse processo. A seguir, explora-se a livraria LMIC no sentido de a adaptar para envio dos dados, via LoRaWAN, usando ABP que é o que a Gateway LoRa suporta. Finalmente, agrega-se toda essa informação no sentido de se produzir uma primeira versão do firmware a ser embebido no microcontrolador.

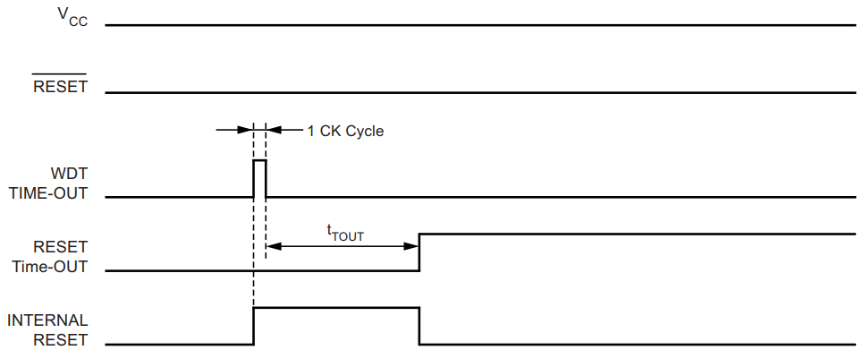
### Watchdog timer

Possui um oscilador à parte com uma frequência de 125kHz e apresenta três modos de operação:

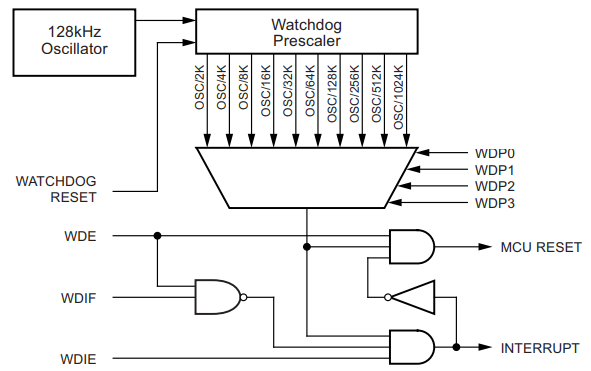
* Interrupção
* *Reset* do sistema
* Ambos

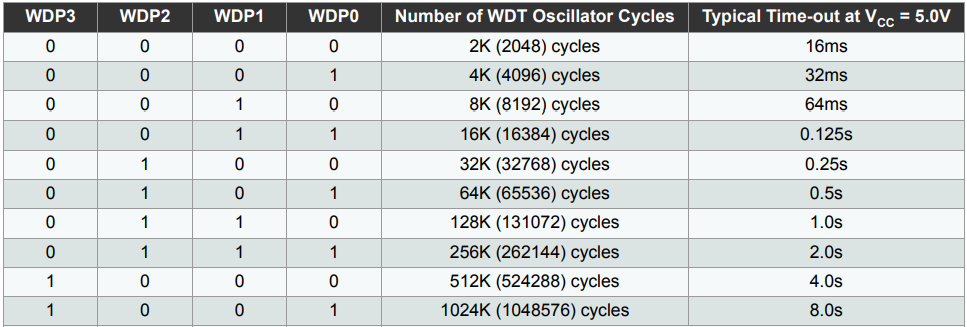
O watchdog gera uma interrupção ou reinicializa o sistema quando o contador atingir um determinado período de *time-out*. O período de *time-out* pode ser selecionado num intervalo entre 16 ms e 8 segundos.

Durante a operação normal, é necessário que seja feito um reset ao watchdog para reiniciar o contador antes que o *time-out* seja atingido. Se o sistema não reiniciar o contador, uma interrupção ou reset do sistema é gerado. Quando o watchdog expirar, ele gerará um pulso de reinicialização curto de duração de um ciclo de *clock*. No flanco descendente deste pulso, o temporizador de atraso começa a contar o período de tempo limite tTOUT.



No modo de interrupção, o watchdog gera uma interrupção quando o temporizador expira. Esta interrupção pode ser usada para despertar o dispositivo do modo de *sleep* assim como um temporizador geral do sistema.





No sketch que se segue o watchdog é parametrizado para reiniciar o microcontrolador a cada 4 segundos. Como o contador do watchdog nunca é reiniciado antes dos 4 segundos, a *string* “Inicio” é apresentada no Serial Monitor a cada 4 segundos.

void setup() {

Serial.begin(9600);

Serial.println("Inicio");

// Deabilita interrupções

cli();

//reset watchdog

// wdt\_reset(); // Opção 1 mas precisa de #include <avr/wdt.h>

MCUSR &= ~(1 << WDRF); // Opção 2

//Estabelece interrupções watchdog

// WDCE: Watchdog Change Enable - necessário para alterar os valores do watchdog (colocado auttomaticamente a zero ao fim de 4xclk

// WDE: Watchdog System Reset Enable - Na realidade, este bit é controlado por WDRF. Se WDRF = 1 então WDE = 1;

WDTCSR = (1<<WDCE)|(1<<WDE);

//Inicia watchdog timer com prescaler (4 segundos) WDP3:WDP0 = "1000"

// WDIW: Watchdog Interrupt Enable - Se WDE = 0 o watchdog está em modo interrupção e se WDE=1 está em modo interrução + reset

WDTCSR = (1<<WDIE)|(1<<WDE)|(1<<WDP3);

//Habilita interrupções globais

sei(); // sem esta linha o comportamento do MCU é estranho… o porto série não envia a string completa (apenas “In”) e reinicia automaticamente a cada 8-20 segundos.

}

void loop() {}

### Listagem: ModosWDTATmega328\_ex1

Usando este código, medi o tempo necessário para enviar 10 mensagens de “Início” o que resultou em perto de 40 segundo. Efetivamente, cerca de 4 segundos por mensagem. O sketch anterior foi adaptado no sentido de incluir uma rotina de atendimento à interrupção gerada pelo watchdog:

ISR(WDT\_vect)

{

Serial.println("Interrupção");

}

void setup() {

Serial.begin(2000000);

Serial.println("Inicio");

// Deabilita interrupções

cli();

//reset watchdog

// wdt\_reset(); // Opção 1 mas precisa de #include <avr/wdt.h>

MCUSR &= ~(1 << WDRF); // Opção 2

//Estabelece interrupções watchdog

// WDCE: Watchdog Change Enable - necessário para alterar os valores do watchdog (colocado auttomaticamente a zero ao fim de 4xclk

// WDE: Watchdog System Reset Enable - Na realidade, este bit é controlado por WDRF. Se WDRF = 1 então WDE = 1;

WDTCSR = (1<<WDCE)|(1<<WDE);

//Inicia watchdog timer com prescaler (4 segundos) WDP3:WDP0 = "1000"

// WDIW: Watchdog Interrupt Enable - Se WDE = 0 o watchdog está em modo interrupção e se WDE=1 está em modo interrução + reset

WDTCSR = (1<<WDIE)|(1<<WDE)|(1<<WDP3);

//Habilita interrupções globais

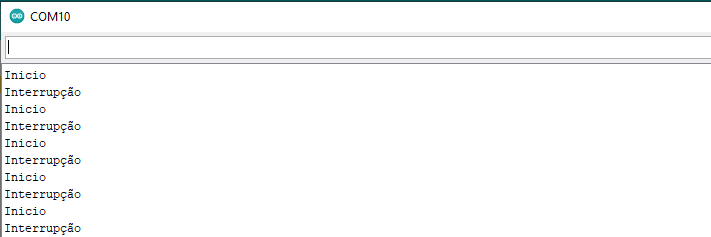
sei(); // sem esta linha o comportamento do MCU é estranho… o porto série não envia a string completa (apenas “In”) e reinicia automaticamente a cada 8-20 segundos.

}

void loop() {}

### Listagem: ModosWDTATmega328\_ex2

Agora no serial monitor aparecem as *strings* “Inicio” e “Interrupção” alternadamente. Voltei a medir o tempo necessário para observar dez mensagens consecutivas.



O tempo determinado foi em torno de 40 segundo. Daqui se pode concluir que, quando no modo Reset+Interrupção, o MCU executa cada um dos dois de forma alternada (e não simultânea) conforme se pode observar durante a execução do programa (para além de que se fosse simultânea, as 10 mensagens deveriam ter lugar em metade do tempo).

O sketch que se segue mostra a forma de estabelecer que o watchdog apenas ativa a interrupção:

ISR(WDT\_vect)

{

Serial.println("Interrupção");

}

void setup() {

Serial.begin(2000000);

Serial.println("Inicio");

// Desabilita interrupções

cli();

//reset watchdog

// wdt\_reset(); // Opção 1 mas precisa de #include <avr/wdt.h>

MCUSR &= ~(1 << WDRF); // Opção 2

//Estabelece interrupções watchdog

// WDCE: Watchdog Change Enable - necessário para alterar os valores do watchdog (colocado automaticamente a zero ao fim de 4xclk

// WDE: Watchdog System Reset Enable - Na realidade, este bit é controlado por WDRF. Se WDRF = 1 então WDE = 1;

WDTCSR = (1<<WDCE)|(1<<WDE);

//Inicia watchdog timer com prescaler (4 segundos) WDP3:WDP0 = "1000"

// WDIW: Watchdog Interrupt Enable - Se WDE = 0 o watchdog está em modo interrupção e se WDE=1 está em modo interrução + reset

WDTCSR = (1<<WDIE)|(1<<WDP3); // apenas no modo interrupção

//Habilita interrupções globais

sei(); // sem esta linha o comportamento do MCU é estranho… o porto série não envia a string completa (apenas “In”) e reinicia automaticamente a cada 8-20 segundos.

}

void loop() {}

### Listagem: ModosWDTATmega328\_ex2

O resultado do sketch anterior pode ser observado na figura que se segue:



Efetivamente, apenas a *string* “Inicio” apenas aparece uma única vez o que garante que o MCU nunca mais foi reiniciado. Vamos alterar agora o sketch anterior de modo que, ao fim de 4 interrupções, o watchdog é desabilitado.

static int nInt = 0;

ISR(WDT\_vect)

{

Serial.println("Interrupção");

if(nInt==3) WDTCSR &= ~(1<<WDIE);

nInt++;

}

void setup() {

Serial.begin(2000000);

Serial.println("Inicio");

// Desabilita interrupções

cli();

//reset watchdog

// wdt\_reset(); // Opção 1 mas precisa de #include <avr/wdt.h>

MCUSR &= ~(1 << WDRF); // Opção 2

//Estabelece interrupções watchdog

// WDCE: Watchdog Change Enable - necessário para alterar os valores do watchdog (colocado automaticamente a zero ao fim de 4xclk

// WDE: Watchdog System Reset Enable - Na realidade, este bit é controlado por WDRF. Se WDRF = 1 então WDE = 1;

WDTCSR = (1<<WDCE)|(1<<WDE);

// Inicia watchdog timer com prescaler (4 segundos) WDP3:WDP0 = "1000"

// WDIW: Watchdog Interrupt Enable - Se WDE = 0 o watchdog está em modo interrupção e se WDE=1 está em modo interrução + reset

WDTCSR = (1<<WDIE)|(1<<WDP3);

//Habilita interrupções globais

sei(); // sem esta linha o comportamento do MCU é estranho… o porto série não envia a string completa (apenas “In”) e reinicia automaticamente a cada 8-20 segundos.

}

void loop() {}

### Modos de sleep

O ATmega328P suporta diferentes modos de sleep. Os modos de sleep permitem parar ou desligar módulos não utilizados no microcontrolador, o que reduz significativamente o consumo de energia. Este microcontrolador possui *Brown-out detector* (BOD) que avalia a tensão de alimentação. Efetivamente, muitos microcontroladores possuem um circuito de proteção que deteta quando a tensão de alimentação fica abaixo de um determinado limiar e coloca-o no estado de reinicialização para garantir a inicialização adequada quando a tensão retornar.

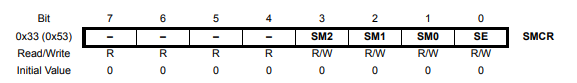
De acordo com as folhas de dados, existem seis tipos diferentes de modos de suspensão sumariados na seguinte tabela:



1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.
3. For INT1 and INT0, only level interrupt

Estes vários modos de suspensão podem ser gerados de forma simples com recurso à biblioteca **LowPower**. Detalhes sobre esta biblioteca serão apresentados adiante.

Para entrar em qualquer modo de suspensão, é preciso habilitar o bit de sleep (SE) no registro de controlo do modo de sleep (SMCR).



Na prática, este bit pode ser ativado diretamente. O sketch que se apresenta a seguir ilustra um de dois procedimentos possíveis (de entre outros que possam ser possíveis). É de notar que existem outras formas que se encontram disponíveis que evitam ter que consultar o mapa de registos. Por exemplo, o uso da bibliotecas <avr/sleep.h> é uma delas da qual se falará adiante.

void setup() {

Serial.begin(9600);

Serial.println(SMCR);

SMCR |=0x01; // Ativa SE (versão 1)

SMCR |= 1<<SE; // Ativa SE (versão 2)

Serial.println(SMCR);

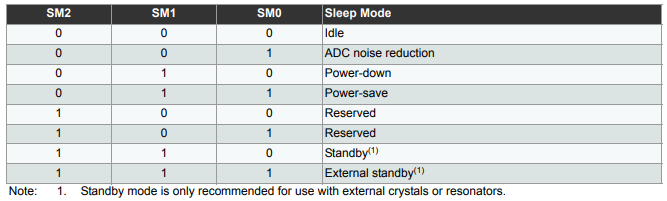
}

void loop() {

// put your main code here, to run repeatedly:

}

Os bits SM2:SM0 definem um dos cinco diferentes modos de sleep de acordo com a seguinte tabela:



De acordo com a atual topologia em que se utiliza o oscilador RC interno do microcontrolador, os últimos dois modos estão fora de questão pelo que restam os primeiros quatro.

* O modo de **power-down** salva o conteúdo do registro, mas congela o oscilador, desabilitando todas as outras funções do chip até à próxima interrupção ou reinicialização do hardware. De acordo com as folhas de dados, o consumo do microcontrolador no modo power-down é de **1uA @ 3V**
* No modo **power-save**, o temporizador assíncrono continua a funcionar enquanto o restante do dispositivo está dormindo.
* O modo de **redução de ruído ADC** suspende o CPU e todos os módulos de E/S, exceto o temporizador assíncrono e ADC. Isto para minimizar o ruído de comutação durante as conversões ADC.
* No modo **iddle** (espera), o oscilador de cristal/ressonador está a operar enquanto o resto do dispositivo está inativo. Isto permite um arranque muito rápido combinado com um baixo consumo de energia.

De acordo com os ensaios apresentados [aqui](http://gammon.com.au/power), a corrente elétrica consumida pelo ATMega328 em cada um dos modos de operação é[[1]](#footnote-1):

* NORMAL: 15.15 mA
* SLEEP\_MODE\_IDLE: 15 mA
* SLEEP\_MODE\_ADC: 6.5 mA
* SLEEP\_MODE\_PWR\_SAVE: 1.62 mA
* SLEEP\_MODE\_EXT\_STANDBY: 1.62 mA
* SLEEP\_MODE\_STANDBY : 0.84 mA
* SLEEP\_MODE\_PWR\_DOWN : 0.36 mA

O gráfico comparativo dos vários modos de suspensão encontra-se a seguir.

Para colocar o microcontrolador num destes modos de suspensão é necessário parametrizar o registo **SMCR** e utilizar a instrução (opcode) **sleep**. A execução de instruções Assembly dentro de programas escritos em C/C++, ainda que possível, é inconveniente. Deste modo, a bibliotecas <avr/sleep.h> permite agilizar a parametrização e utilização dos vários modos de suspensão. Em particular, esta biblioteca disponibiliza as seguintes funções:

void sleep\_enable (void)

void sleep\_disable (void)

void sleep\_cpu (void)

void sleep\_mode (void)

void sleep\_bod\_disable (void)

Exemplo de utilização ( a partir da documentação da biblioteca disponível [aqui](https://www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html)):

#include <[avr/interrupt.h](https://www.nongnu.org/avr-libc/user-manual/interrupt_8h.html)>

#include <[avr/sleep.h](https://www.nongnu.org/avr-libc/user-manual/sleep_8h.html)>

...

set\_sleep\_mode(<mode>);

[cli](https://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html#ga68c330e94fe121eba993e5a5973c3162)();

if (some\_condition)

{

[sleep\_enable](https://www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html#ga475174a7aa4eda03dfa7a4483e400a9e)();

[sleep\_bod\_disable](https://www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html#gabf889562cc5ea768ee80cfc8a5bb0312)();

[sei](https://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html#gaad5ebd34cb344c26ac87594f79b06b73)();

[sleep\_cpu](https://www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html#ga157b2578d95309c197b739f812938d94)();

[sleep\_disable](https://www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html#gaeae22433a78fd8d50f915fb68c416efd)();

}

[sei](https://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html#gaad5ebd34cb344c26ac87594f79b06b73)();

Para começar a explorar esta biblioteca, comecei por criar um exemplo genérico usando o Arduino Sketch. A ideia é confirmar que o Arduino entra em modo de POWER\_DOWN. Para isso, comecei por fazer a parametrização do registo, fazer a chamada à instrução sleep e mandar acender um LED. Estava à espera que, com base neste sketch, o LED nunca acendesse. No entanto, para meu espanto, o LED acendia. Acrescentando um Serial.print no loop() para debug verifiquei que este nunca era executado. O que significa que o MCU entrava em suspensão mas, por alguma razão, ainda executava o programa algumas instruções do setup(). Não executava tudo o que existia no setup() dado que o último print não era apresentado na consola.

#include <avr/sleep.h>

void setup() {

Serial.begin(2000000);

Serial.println("Inicio");

cli();

//-------------------------------------------------------------------------

// SLEEP CONFIG

//-------------------------------------------------------------------------

set\_sleep\_mode(SLEEP\_MODE\_PWR\_DOWN);

sleep\_enable();//Enabling sleep mode;

sleep\_bod\_disable();

sleep\_cpu();

pinMode(LED\_BUILTIN, OUTPUT);

digitalWrite(LED\_BUILTIN,LOW);

delay(1000);

digitalWrite(LED\_BUILTIN,HIGH);

sei();

Serial.println("Fim");

}

void loop() {

Serial.println("Ciclo");

delay(1000);

}

É de notar que, alterando a posição da instrução pinMode(), o programa passou a comporta-se como esperado. Em particular, colocando-a antes da parametrização do sleep(). O código que se segue tem o comportamento esperado….

#include <avr/sleep.h>

void setup() {

Serial.begin(2000000);

Serial.println("Inicio");

pinMode(LED\_BUILTIN, OUTPUT);

cli();

//-------------------------------------------------------------------------

// SLEEP CONFIG

//-------------------------------------------------------------------------

set\_sleep\_mode(SLEEP\_MODE\_PWR\_DOWN);

sleep\_enable();//Enabling sleep mode;

sleep\_bod\_disable();

//-------------------------------------------------------------------------

sleep\_cpu(); // Coloca MCU em Power Down

digitalWrite(LED\_BUILTIN,LOW);

delay(1000);

digitalWrite(LED\_BUILTIN,HIGH);

Serial.println("Fim");

sei();

}

void loop() {

Serial.println("Ciclo");

delay(1000);

}

A seguir, um sketch que utiliza o watchdog timer para acordar o MCU de oito em oito segundos e imprime na consola série a palavra ciclo. Funcionou de acordo com o especificado.

#include <avr/sleep.h>

ISR(WDT\_vect){}

void setup() {

Serial.begin(2000000);

Serial.println("Inicio");

pinMode(LED\_BUILTIN, OUTPUT);

cli();

//-------------------------------------------------------------------------

// WATCHDOG CONFIG

//-------------------------------------------------------------------------

// Começa por fazer o reset ao watchdog

MCUSR &= ~(1 << WDRF);

// Estabelece interrupções do watchdog

// WDCE: Watchdog Change Enable - necessário para alterar os valores do

// watchdog (colocado automaticamente a

// zero ao fim de 4xclk

// WDE: Watchdog System Reset Enable - Na realidade, este bit é controlado

// por WDRF. Se WDRF = 1 então WDE = 1

WDTCSR = (1<<WDCE)|(1<<WDE);

// Inicia watchdog timer com prescaler (8 segundos) WDP3:WDP0 = "1001"

// WDIW: Watchdog Interrupt Enable - Se WDE = 0 o watchdog está em modo

// interrupção e se WDE=1 está em modo

// interrução + reset

WDTCSR = (1<<WDIE)|(1<<WDP3)|(1<<WDP0);

//-------------------------------------------------------------------------

// SLEEP CONFIG

//-------------------------------------------------------------------------

set\_sleep\_mode(SLEEP\_MODE\_PWR\_DOWN);

sleep\_enable();//Enabling sleep mode;

//-------------------------------------------------------------------------

sei();

}

void loop() {

sleep\_bod\_disable()[[2]](#footnote-2); // ler nota no texto

sleep\_cpu();

Serial.println("Ciclo"); // De 8 em 8 segundos é enviada a palavra Ciclo

}

A desabilitação do BOD é controlada pelo bit 6, BODS (BOD Sleep) no registro de controle MCUCR. Ativar este bit desativa o BOD enquanto que atribuir-lhe um zero mantém o BOD ativo. A configuração padrão mantém o BOD ativo, ou seja, o BODS é definido como zero.

Inicialmente, o MCUCR = 0x00.

MCUCR = \_BV(BODS) | \_BV(BODSE);

MCUCR = \_BV(BODS);

The BODS bit is automatically cleared after three clock cycles. O que significa que o sleep tem que ser chamado imediatamente a seguir à desativação do BOD.

Para economizar energia, o ADC deve ser desabilitado antes de entrar em **qualquer** modo de suspensão. Quando o ADC é desligado e ligado novamente, a próxima conversão será uma **conversão estendida**. Não existe nas folhas de dados mais nenhuma referência a esse tipo de conversão. Pelo que vi online, a ideia é realizar entre uma a três conversões A/D e desprezá-las antes de realizar a conversão real do que se pretende.

Para desabilitar a ADC basta incluir ANTES de entrar em sleep:

// desabilita ADC

ADCSRA = 0;

A primeira conversão, após o ADC ser ativado (ou seja, o bit ADEN em ADCSRA ativado) requer 25 ciclos de relógio da ADC para inicializar o circuito.

ADCSRA &= ~\_BV(ADEN);

the choice by the analogRead library of a prescaler of 128

ADCSRA &= ~(bit (ADPS0) | bit (ADPS1) | bit (ADPS2)); // clear prescaler bits

ADCSRA |= bit (ADPS0) | bit (ADPS1) | bit (ADPS2); // 128

Este assunto será explorado em detalhe na próxima seção. Nesta ainda, falta falar nas bibliotecas LowPower e <avr/power.h>.

#### <avr/power.h>

Esta biblioteca (ou conjunto de macros) pode ser utilizada em microcontroladores AVR que possuam o registo PRR (Power Reduction Register). Este registo permite desativar alguns periféricos que não sejam utilizados de modo a reduzir a pegada energética do MCU. É de notar que a utilização deste registo apenas faz sentido no caso do MCU não se encontrar em Power\_DOWN dado que, para esse caso, os periféricos são todos desligados automaticamente.

Por exemplo, se o modo sleep for IDDLE, é possível desligar os periféricos através da execução da função:

power\_all\_disable ();

De acordo com [Gammon](http://gammon.com.au/power), “You must use the PRR after setting ADCSRA to zero, otherwise the ADC is "frozen" in an active state.”

#### Low-Power

Esta biblioteca encontra-se disponível no repositório <https://github.com/rocketscream/Low-Power> e define um conjunto de funções que agilizam a parametrização dos vários modos de suspensão possíveis.

A seguir alguns exemplos:

// Enter power down state with ADC and BOD module disabled.

// Wake up when wake up pin is low.

LowPower.powerDown(SLEEP\_FOREVER, ADC\_OFF, BOD\_OFF);

// Enter power down state for 8 s with ADC and BOD module disabled

LowPower.powerDown(SLEEP\_8S, ADC\_OFF, BOD\_OFF);

### Referências

<http://gammon.com.au/power>

<https://blog.duk.io/sleeping-atmega328-on-1ua-with-timer-wakeup/>

Nanopower timer que pode ser usado externamente para acordar o MCU: <https://www.ti.com/lit/ds/symlink/tpl5010.pdf?ts=1657845868984>

### Conversão A/D

#### Pinos analógicos

Os pinos analógicos podem ser usados de forma idêntica aos pinos digitais. Por exemplo, o seguinte código define o porto analógico 0 como saída e define-o com o valor lógico ALTO:

pinMode(A0, OUTPUT);

digitalWrite(A0, HIGH);

Os pinos analógicos também possuem resistores pull-up, que funcionam de forma idêntica aos resistores pull-up nos pinos digitais. Eles são habilitados emitindo um comando como

pinMode(A0, INPUT\_PULLUP); // set pull-up on analog pin 0

**Note-se que, ativar um pull-up afetará os valores relatados por analogRead().**

O comando analogRead não funcionará corretamente se um pino tiver sido definido anteriormente para uma saída, portanto, se esse for o caso, defina-o de volta para uma entrada antes de usar analogRead. Da mesma forma, se o pino foi definido como HIGH como uma saída, o resistor de pull-up será definido, quando comutado de volta para uma entrada. As folhas de dados também advertem contra a troca de pinos analógicos em proximidade temporal próxima para fazer leituras A/D (analogRead) em outros pinos analógicos. Isso pode causar ruído elétrico e introduzir jitter no sistema analógico. Pode ser desejável, depois de manipular os pinos analógicos (no modo digital), adicionar um pequeno atraso antes de usar analogRead() para ler outros pinos analógicos.

#### Leitura de valores analógicos

analogReference(opcão)

Configura a tensão de referência de acordo com as seguintes opções:

* DEFAULT: o valor no pino AVCC
* INTERNAL: referência interna de 1.1 volts
* EXTERNAL: o valor em AREF.

Após alterar a referência analógica, as primeiras leituras de analogRead() podem não ser precisas.

Se for usada a referência externa do pino AREF, deve definir-se a referência analógica como EXTERNAL antes de chamar analogRead(). Caso contrário, gera-se um curto-circuito na tensão de referência ativa (gerada internamente) e o pino AREF, possivelmente danificando o microcontrolador.

#### Conversão A/D: versão #1

#include <avr/sleep.h>

#include <avr/sleep.h>

#define ordemFiltro 25

//-------------------------------------------------------------------------

// FUNÇÕES

//-------------------------------------------------------------------------

float leituraAnalogica(const byte port)

{

ADCSRA |= \_BV(ADEN); // Ativa ADC

float valorFiltrado = 0;

for (int i=0;i<ordemFiltro;i++) {

valorFiltrado += (float)analogRead(port);

delayMicroseconds(50);

}

ADCSRA &= ~\_BV(ADEN); // Desativa ADC

// Filtro de média...

return(valorFiltrado/(float)ordemFiltro);

}

//-------------------------------------------------------------------------

// VETORES DE INTERRUPÇÃO

ISR(WDT\_vect){}

//-------------------------------------------------------------------------

void setup() {

Serial.begin(2000000);

cli();

//-------------------------------------------------------------------------

// WATCHDOG CONFIG

//-------------------------------------------------------------------------

// Começa por fazer o reset ao watchdog

MCUSR &= ~(1 << WDRF);

// Estabelece interrupções do watchdog

// WDCE: Watchdog Change Enable - necessário para alterar os valores do

// watchdog (colocado automaticamente a

// zero ao fim de 4xclk

// WDE: Watchdog System Reset Enable - Na realidade, este bit é controlado

// por WDRF. Se WDRF = 1 então WDE = 1

WDTCSR = (1<<WDCE)|(1<<WDE);

// Inicia watchdog timer com prescaler (8 segundos) WDP3:WDP0 = "1001"

// WDIW: Watchdog Interrupt Enable - Se WDE = 0 o watchdog está em modo

// interrupção e se WDE=1 está em modo

// interrução + reset

// WDTCSR = (1<<WDIE)|(1<<WDP3)|(1<<WDP0); //(8 segundos)

WDTCSR = (1<<WDIE)|(1<<WDP2)|(1<<WDP1); //(1 segundo)

//-------------------------------------------------------------------------

// Habilita interrupções

sei();

}

void loop() {

float sensorValue=leituraAnalogica(0);

Serial.println(sensorValue);

//-------------------------------------------------------------------------

// SLEEP CONFIG

//-------------------------------------------------------------------------

set\_sleep\_mode(SLEEP\_MODE\_PWR\_DOWN);

sleep\_enable();//Enabling sleep mode;

sleep\_bod\_disable(); // Desativa BOD

sleep\_cpu(); // Dorme

sleep\_disable ();

}

#### Conversão A/D: versão #2

Esta versão recorre à suspensão do MCU durante a fase de conversão A/D de modo a reduzir o ruído:

#include <avr/sleep.h>

#define ordemFiltro 25

//-------------------------------------------------------------------------

// VETORES DE INTERRUPÇÃO

//-------------------------------------------------------------------------

EMPTY\_INTERRUPT (WDT\_vect);

EMPTY\_INTERRUPT (ADC\_vect);

//-------------------------------------------------------------------------

//-------------------------------------------------------------------------

// FUNÇÕES

//-------------------------------------------------------------------------

//-------------------------------------------------------------------------

// Leitura da ADC em modo sleep (crédito para Nick Gammon)

//-------------------------------------------------------------------------

float getReading (const byte port)

{

ADCSRA = bit (ADEN) | bit (ADIF); // enable ADC, turn off any pending interrupt

ADCSRA |= bit (ADPS0) | bit (ADPS1) | bit (ADPS2); // prescaler of 128

ADMUX = bit (REFS0) | (port & 0x07); // AVcc

// REFS1 REFS0 Voltage Reference Selection

// 0 0 AREF, internal VREF turned off

// 0 1 AVCC with external capacitor at AREF pin (100nF é o que é usado no Arduino Uno)

// 1 0 Reserved

// 1 1 Internal 1.1V voltage reference with external capacitor at AREF pin

cli ();

set\_sleep\_mode (SLEEP\_MODE\_ADC); // sleep during sample

sleep\_enable();

// start the conversion

ADCSRA |= bit (ADSC) | bit (ADIE);

sei ();

sleep\_cpu ();

sleep\_disable ();

// awake again, reading should be done, but better make sure

// maybe the timer interrupt fired

while (bit\_is\_set (ADCSRA, ADSC)) { }

ADCSRA &= ~\_BV(ADEN); // Desativa ADC

return (ADC);

} // end of getReading

//-------------------------------------------------------------------------

// leituraAnalogica()

//-------------------------------------------------------------------------

float leituraAnalogica(const byte port)

{

float valorFiltrado = 0;

for (int i=0;i<ordemFiltro;i++) {

valorFiltrado += getReading(port);

delayMicroseconds(50);

}

// Filtro de média...

return(valorFiltrado/(float)ordemFiltro);

}

//-------------------------------------------------------------------------

void setup() {

Serial.begin(2000000);

cli();

//-------------------------------------------------------------------------

// WATCHDOG CONFIG

//-------------------------------------------------------------------------

// Começa por fazer o reset ao watchdog

MCUSR &= ~(1 << WDRF);

// Estabelece interrupções do watchdog

// WDCE: Watchdog Change Enable - necessário para alterar os valores do

// watchdog (colocado automaticamente a

// zero ao fim de 4xclk

// WDE: Watchdog System Reset Enable - Na realidade, este bit é controlado

// por WDRF. Se WDRF = 1 então WDE = 1

WDTCSR = (1<<WDCE)|(1<<WDE);

// Inicia watchdog timer com prescaler (8 segundos) WDP3:WDP0 = "1001"

// WDIW: Watchdog Interrupt Enable - Se WDE = 0 o watchdog está em modo

// interrupção e se WDE=1 está em modo

// interrução + reset

// WDTCSR = (1<<WDIE)|(1<<WDP3)|(1<<WDP0); //(8 segundos)

WDTCSR = (1<<WDIE)|(1<<WDP2)|(1<<WDP1); //(1 segundo)

//-------------------------------------------------------------------------

// Habilita interrupções

sei();

}

//-------------------------------------------------------------------------

void loop() {

float sensorValue=leituraAnalogica(0);

Serial.println(sensorValue);

//-------------------------------------------------------------------------

// SLEEP CONFIG

//-------------------------------------------------------------------------

set\_sleep\_mode(SLEEP\_MODE\_PWR\_DOWN);

sleep\_enable();//Enabling sleep mode;

sleep\_bod\_disable(); // Desativa BOD

sleep\_cpu(); // Dorme

sleep\_disable ();

}

#### Comparação das duas versões

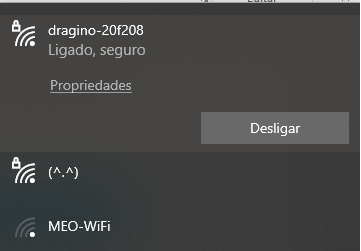
Foi feito um ensaio onde 430 amostras foram adquiridas usando ambos os sketches anteriores. Os resultados apurados foram os seguintes:

|  |  |  |
| --- | --- | --- |
|  | **Com ADC\_SLEEP** | **Usando analogRead()** |
| Média | 512.1589931 | 511.7408 |
| Desvio Padrão | 0.236403813 | 0.26517 |
| Espaço ocupado | 3380 bytes | 3364 bytes |

Dos resultados obtidos pode concluir-se que a versão com ADC\_SLEEP resulta num menor valor de desvio padrão pelo que se pode considerar que os valores registados pela ADC se encontram mais estáveis. Aparentemente a pegada do código é superior em 16 bytes.

### Parametrização da Gateway Dragino DSLO8

Ligar à rede WiFi criada pela gateway:

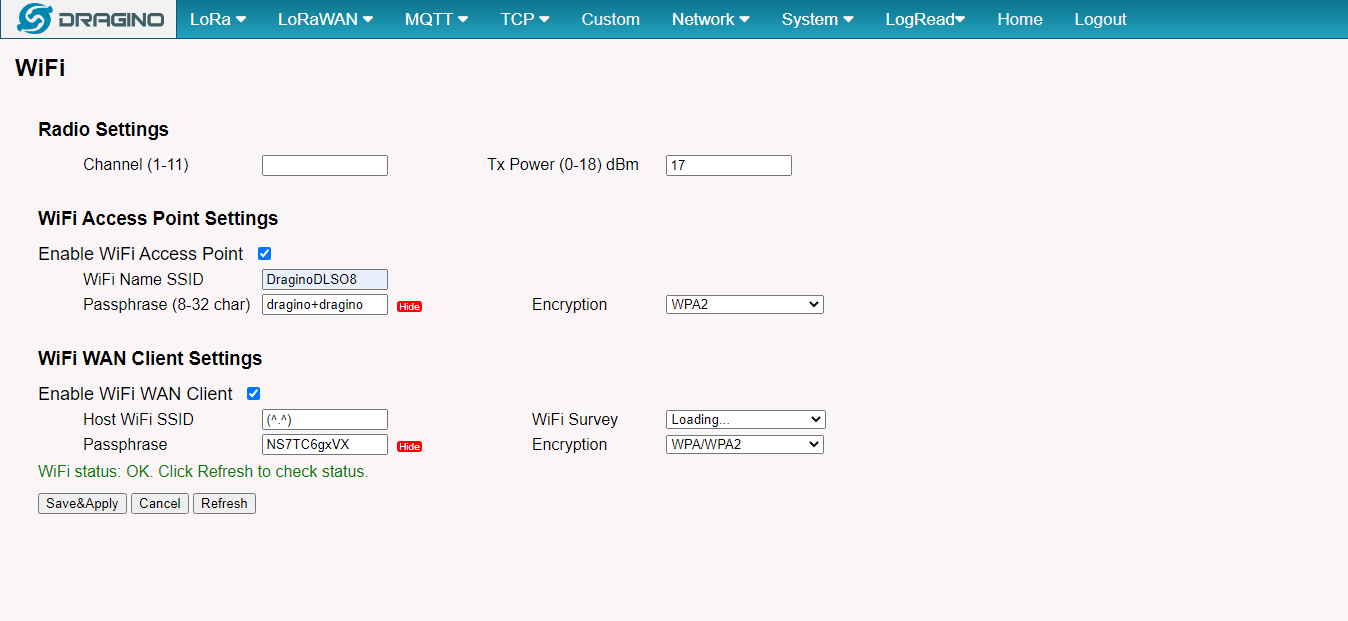


A password, por defeito, é **dragino+dragino**

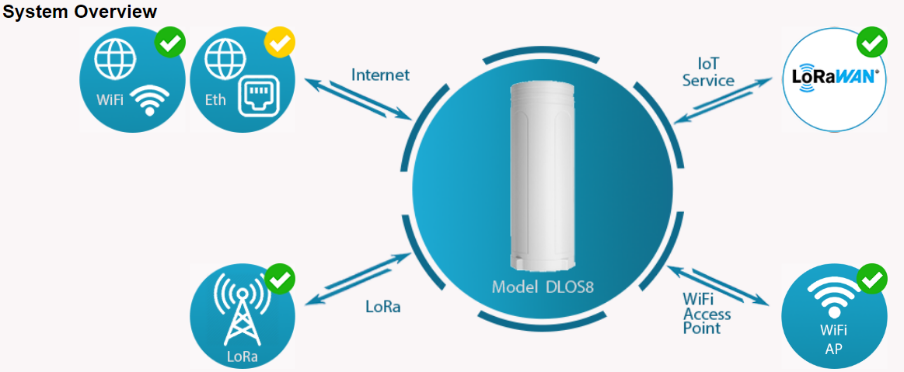
Acedendo ao endereço 10.130.1.1 aparece um popup para preencher com as credenciais que, por defeito, são:

Nome do utilizador: **root**

Palavra passe: **dragino**Começar por parametrizar a rede para acesso via WiFi:



Depois de configurar a rede obtém-se o seguinte *front end*:



#### Envio dos dados via MQTT usando APB decoder

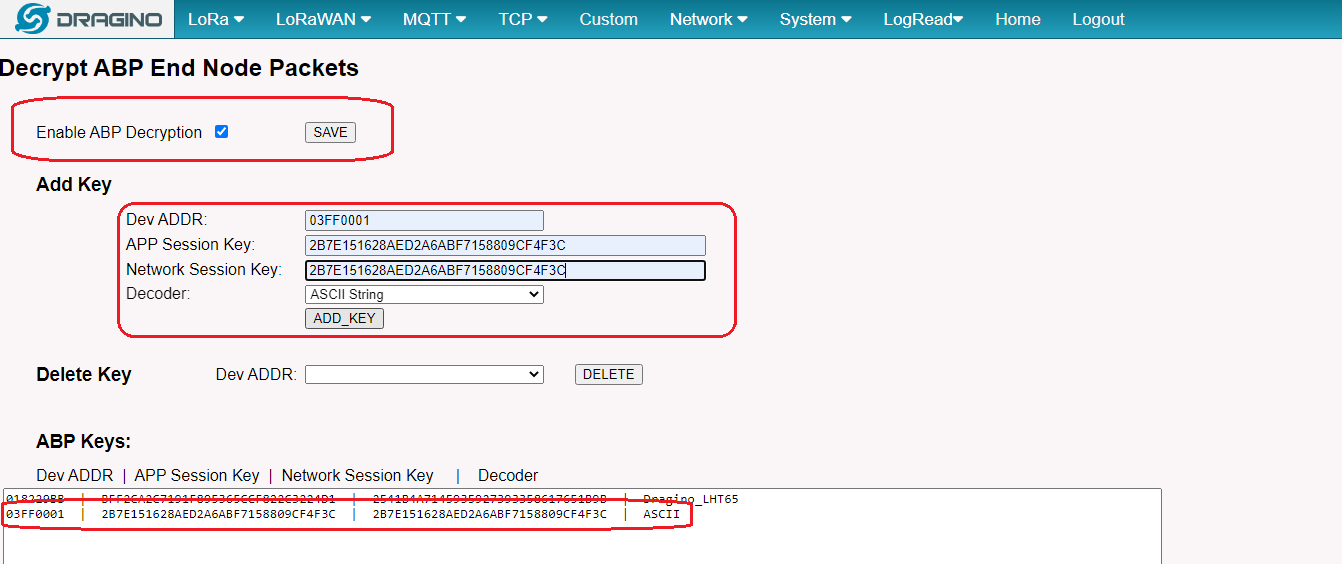
Neste ponto, considera-se que o nó IoT é identificado com o número 0x03FFF001 e com as restantes credenciais ABP.

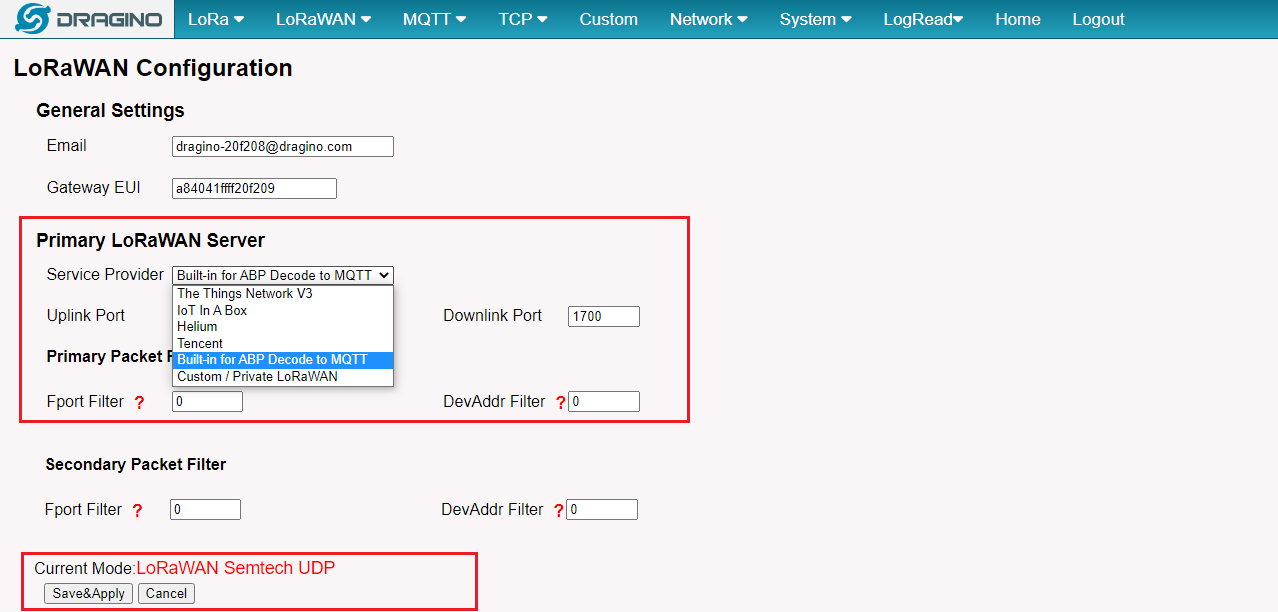
NWKSKEY[16] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6, 0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C }

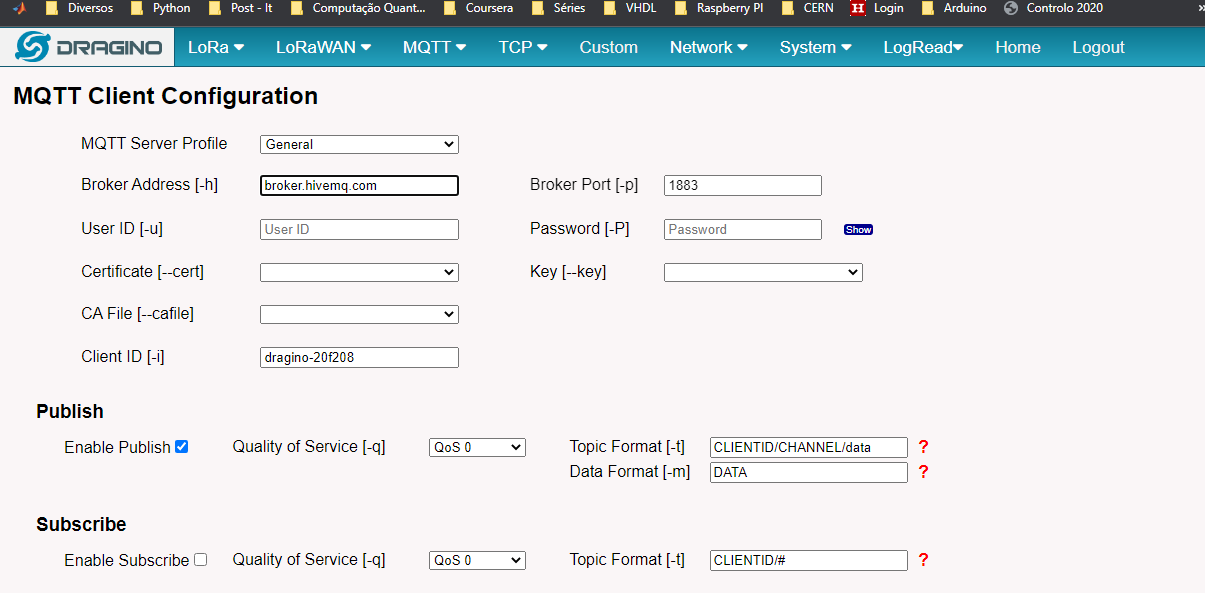
APPSKEY[16] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6, 0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C };

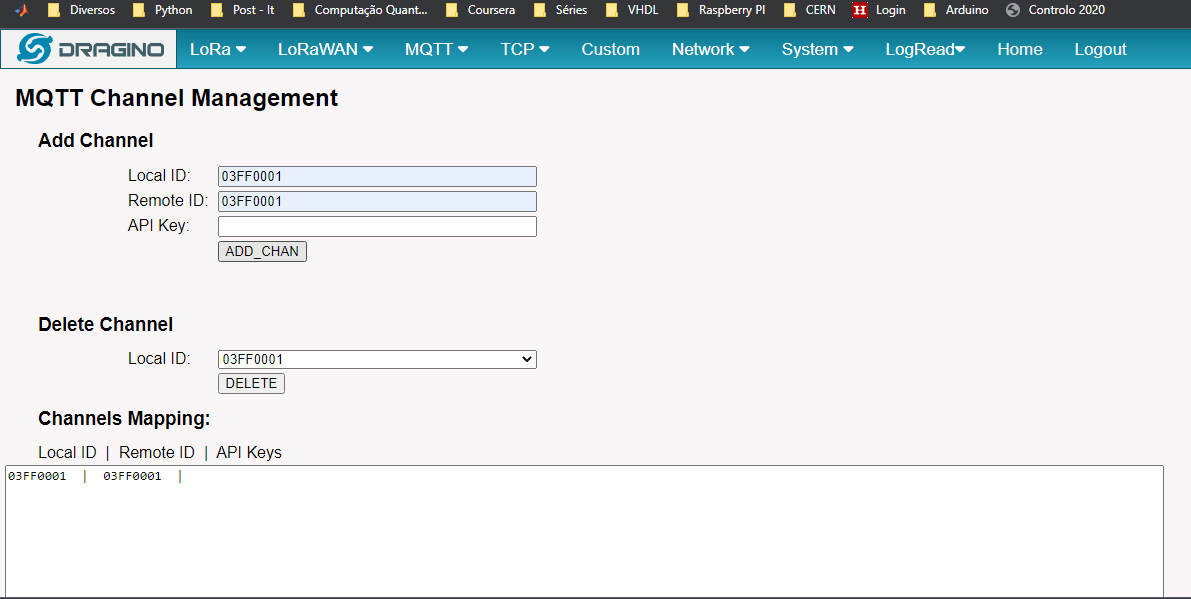
DEVADDR = 0x03FF0001;

Na gateway é necessário realizar a seguinte parametrização:









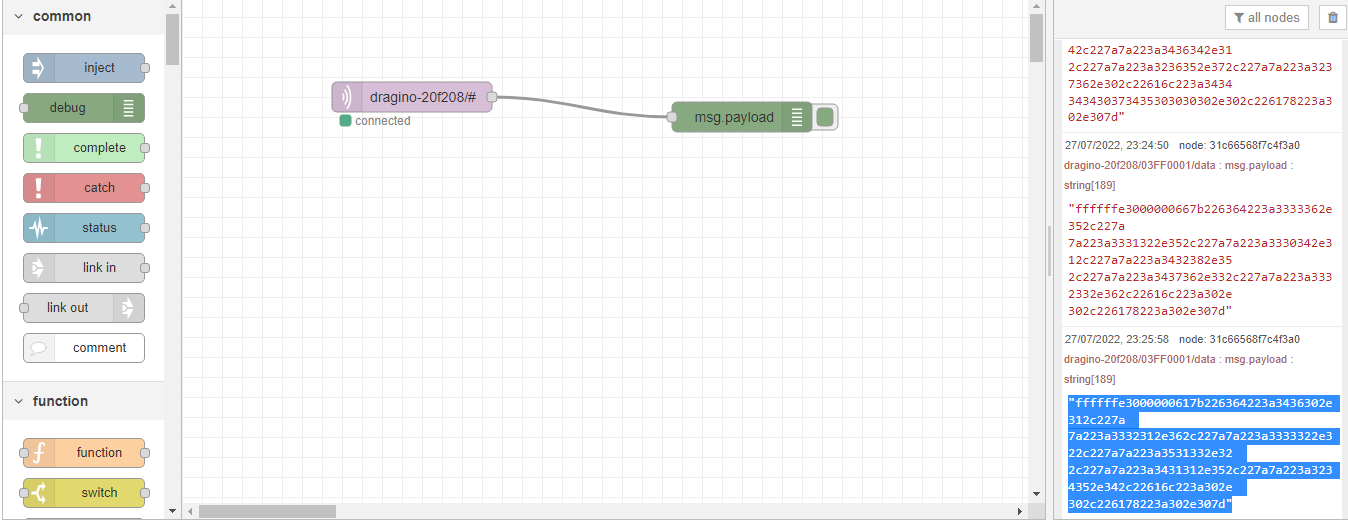
Depois de parametrizar a gateway, passa-se aos ensaios de validação onde o nó referido no início deste documento envia, via gateway, a mensagem “Hello world” para um broker MQTT alojado em *hivemq*. Do outro lado, um *flow* feito em *NodeRed*, subscreve o tópico onde o nó irá publicar. Os valores obtidos serão apresentados via *debug window*.

#### Firmware colocado no nó IoT.

Ver GitHub no diretório Firmware.

#### Flow NodeRED para subscrição de tópico.

O **NodeRED** encontra-se a ser executado num *container* **Docker**. Em baixo segue o *flow* criado para o efeito.



[{"id":"31680187a12acca2","type":"tab","label":"Flow 25","disabled":false,"info":""},{"id":"fbef968ba15bb6d7","type":"mqtt in","z":"31680187a12acca2","name":"","topic":"dragino-20f208/#","qos":"2","datatype":"auto","broker":"37a32862a16ac898","nl":false,"rap":true,"rh":0,"x":230,"y":100,"wires":[["31c66568f7c4f3a0"]]},{"id":"31c66568f7c4f3a0","type":"debug","z":"31680187a12acca2","name":"","active":true,"tosidebar":true,"console":false,"tostatus":false,"complete":"false","statusVal":"","statusType":"auto","x":560,"y":120,"wires":[]},{"id":"37a32862a16ac898","type":"mqtt-broker","name":"","broker":"broker.hivemq.com","port":"1883","clientid":"","usetls":false,"protocolVersion":"4","keepalive":"60","cleansession":true,"birthTopic":"","birthQos":"0","birthPayload":"","birthMsg":{},"closeTopic":"","closeQos":"0","closePayload":"","closeMsg":{},"willTopic":"","willQos":"0","willPayload":"","willMsg":{},"sessionExpiry":""}]

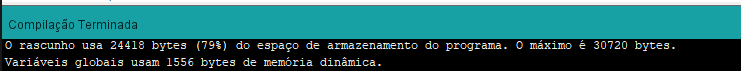
ffffffe3000000617b226364223a3436302e312c227a 7a223a3332312e362c227a7a223a3333322e322c227a7a223a3531332e32 2c227a7a223a3431312e352c227a7a223a3234352e342c22616c223a302e 302c226178223a302e307d

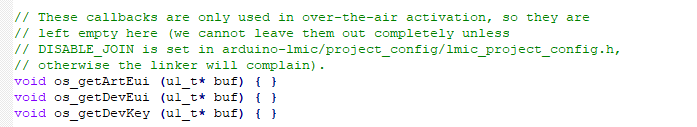
em ASCII

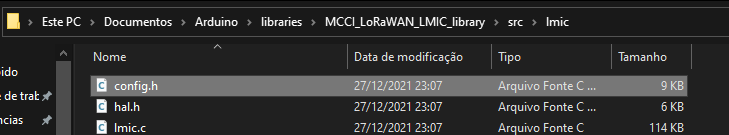
ÿÿÿãa{"cd":460.1,"zz":321.6,"zz":332.2,"zz":513.2,"zz":411.5,"zz":245.4,"al":0.0,"ax":0.0}

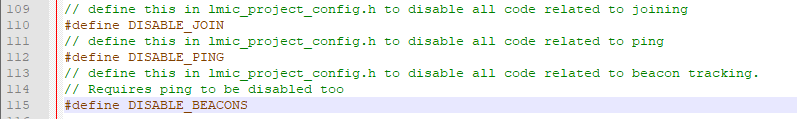
### Alteração da biblioteca LMIC (acabei por não fazer isso)

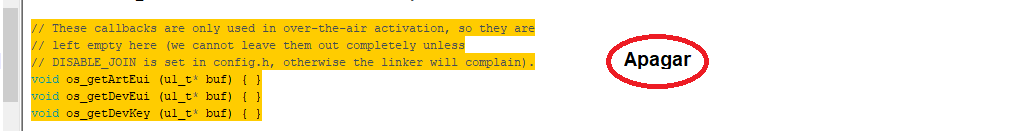
#### Versão final

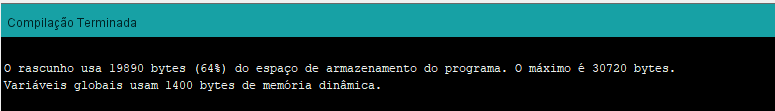










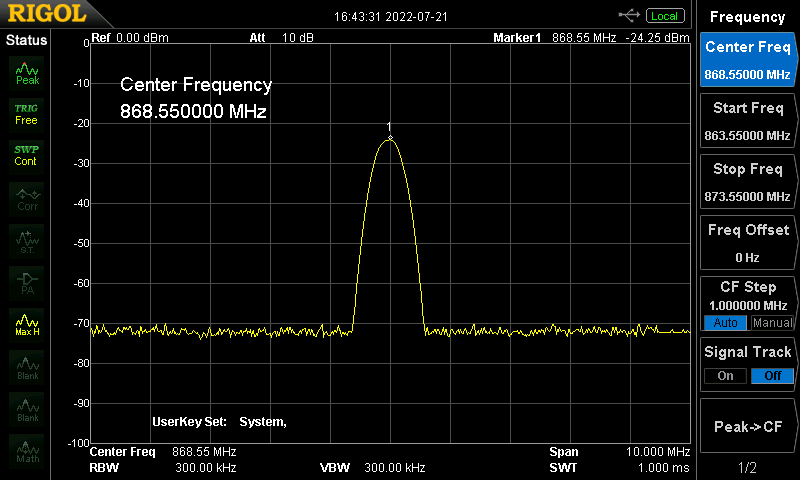


15% de poupança de memória

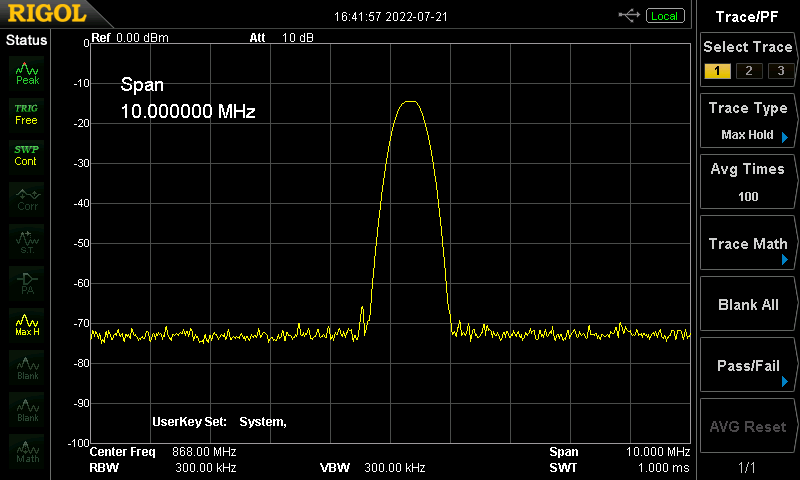
# Pegada RF associada as envios

Inicialmente pensei usar a biblioteca LMIC. No entanto essa biblioteca tem vários problemas. O primeiro é o enorme *footprint* de código que gera. Quase 70% da memória do ATMega328 é ocupada com o sketch básico. A segunda, e mais grave, é a possibilidade de alterar o código que existe para fazer aquilo que queremos. Aparentemente, a biblioteca LMIC cria uma espécie de sistema operativo com uma máquina de estados e não é trivial fazer as alterações que precisamos nomeadamente no que diz respeito ao modo sleep. Por isso mesmo, fiz a adaptação de um conjunto de funções, na forma de uma nova biblioteca, que foram desenhadas para funcionar com o ATTiny85 para operarem segundo o protocolo LoRaWAN via ABP que é o que precisamos.

A nova biblioteca funciona corretamente como se pode verificar com a primeira versão do firmware disponível no GitHub. A figura que se segue mostra a pegada RF associada a um envio do nó IoT programado com esse sketch.



No entanto, após algum tempo a funcionar verifiquei que ele envia sempre na mesma frequência o que é contraproducente por várias razões. Principalmente quando dezenas de nós estão a funcionar simultaneamente usando todos o mesmo canal. A figura que se segue mostra o espetro gerado (sobreposição) após algum tempo com o nó a funcionar.

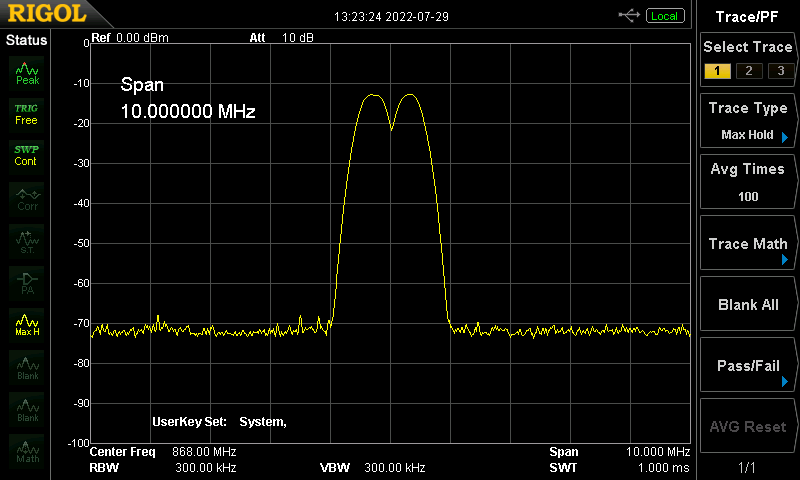


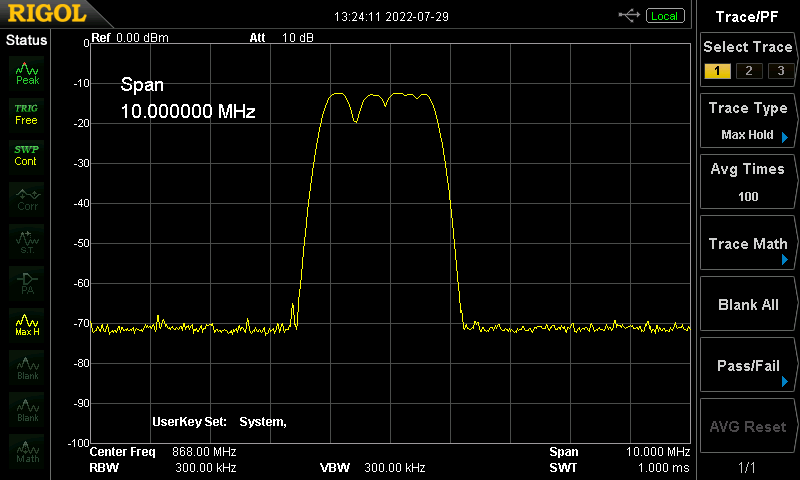
Esse facto também pode ser comprovado através do acesso à gateway LoRa e observar os metadados associados aos envios:

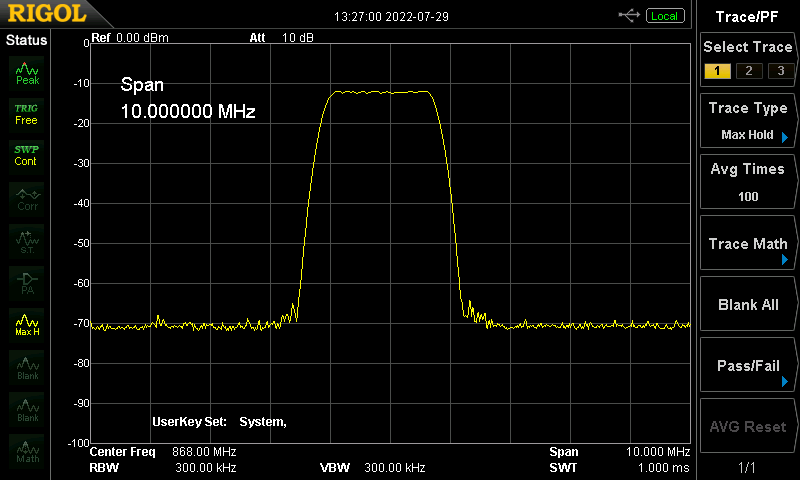


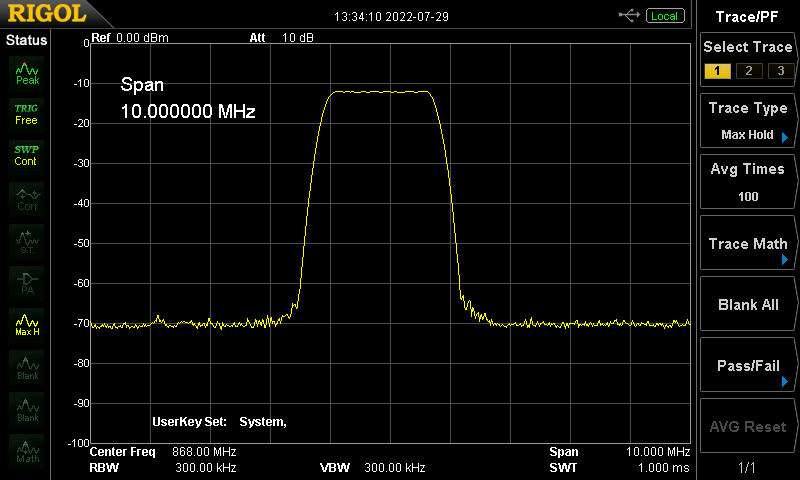
O envio é semrpe feito à frequência de 868.1 MHz com fator de espalhamento 7.

De acordo com o protocolo LoRaWAN, os nós devem comutar automaticamente entre 8 canais de frequência durante o processo de envio. A escolha deve ser feita aleatoriamente sem reposição. Para ilustrtar o que deve ser esperado, a seguinte sequencia de espectro mostra o sinal gerado pelo transceiver quando a biblioteca LMIC é utilizada.









Como se pode verificar, existe não um único canal, mas uma faixa de canais que é utilizada durante o processo de envio. Deste modo, a biblioteca que estou a utilizar neste momento tem que ser capaz de realizar esta operação.

## Canais de Frequência

Acedendo à gateway, verifica-se que esta se encontra programada para ouvir os seguintes canais LoRa:

* Lora MAC, 125kHz, all SF, 868.1 MHz
* Lora MAC, 125kHz, all SF, 868.3 MHz
* Lora MAC, 125kHz, all SF, 868.5 MHz
* Lora MAC, 125kHz, all SF, 867.1 MHz
* Lora MAC, 125kHz, all SF, 867.3 MHz
* Lora MAC, 125kHz, all SF, 867.5 MHz
* Lora MAC, 125kHz, all SF, 867.7 MHz
* Lora MAC, 125kHz, all SF, 867.9 MHz
* Lora MAC, 250kHz, SF7, 868.3 MHz

De acordo com o site <https://www.thethingsnetwork.org/docs/lorawan/frequency-plans/>, os planos de frequência associados à TTN ( e logo ao protocolo LoRaWAN) para a faixa de frequência europeia (EU863-870) são:

Uplink:

* **868.1** - SF7BW125 to SF12BW125
* **868.3** - SF7BW125 to SF12BW125 and SF7BW250
* **868.5** - SF7BW125 to SF12BW125
* **867.1** - SF7BW125 to SF12BW125
* **867.3** - SF7BW125 to SF12BW125
* **867.5** - SF7BW125 to SF12BW125
* **867.7** - SF7BW125 to SF12BW125
* **867.9** - SF7BW125 to SF12BW125

Downlink:

* Uplink channels 1-9 (RX1)
* **869.525** - SF9BW125 (RX2)[[3]](#footnote-3)

Confirma-se que são os mesmos. Assim, e de modo a contornar o problema desta biblioteca, vou desenvolver uma função que seja capaz de ir de encontro ao protocolo LoRaWAN.

## Escolha aleatória sem reposição

Função que retorna um número inteiro entre 0 e 7 sem reposição[[4]](#footnote-4).

int arraylist[]={0,1,2,3,4,5,6,7};

int maxval = 8;

byte returnRandomRFChannel()

{

// random(min,max) gera um número aleatório com distribuição

// uniforme no intervalo [min,max[

int val = random(0,maxval);

int swapval=arraylist[val];

arraylist[val]=arraylist[maxval-1];

arraylist[maxval-1] = swapval;

maxval--;

if (maxval==0) {

maxval = 8;

for (int i=0;i<maxval;i++) arraylist[i]=i;

}

return(swapval);

}

void setup() {

Serial.begin(2000000);

randomSeed(analogRead(A1));

}

void loop() {

Serial.println(returnRandomRFChannel());

delay(10);

}

### Análise estatística dos resultados

#### Com reset do vetor inicial







Removendo a média e recalculando a autocorrelação:





#### Sem reset do vetor inicial

int arraylist[]={0,1,2,3,4,5,6,7};

int maxval = 8;

byte returnRandomRFChannel()

{

// random(min,max) gera um número aleatório com distribuição

// uniforme no intervalo [min,max[

int val = random(0,maxval);

int swapval=arraylist[val];

arraylist[val]=arraylist[maxval-1];

arraylist[maxval-1] = swapval;

maxval--;

if (maxval==0) maxval = 8; // Reset do contador

return(swapval);

}

void setup() {

// put your setup code here, to run once:

Serial.begin(2000000);

randomSeed(analogRead(A1));

}

void loop() {

// put your main code here, to run repeatedly:

Serial.println(returnRandomRFChannel());

delay(10);

}











#### Codificação em Matlab

arraylist=0:7;

maxval=8;

for i=1:3495,

[arraylist,maxval,valor(i)]=returnRandomRFChannel(arraylist,maxval);

end

%%

function [arraylist,maxval,swapval] = returnRandomRFChannel(arraylist,maxval)

val = randi(maxval,1,1);

swapval=arraylist(val);

arraylist(val)=arraylist(maxval);

arraylist(maxval) = swapval;

maxval=maxval-1;

if (maxval==0)

maxval = 8;

end

end









Confirma-se alguma autocorrelação linear entre as 5 amostras consecutivas.

## Alteração da frequência e *spreading factor* na biblioteca

A função RFM\_Send\_Package() disponível em LoRaWAN.ino apresenta a forma como o transceiver RFM95 é parametrizado de modo a poder suporta alterações de frequência e fator de espalhamento. Esta função apresenta-se replicada a seguir:

void RFM\_Send\_Package(unsigned char \*RFM\_Tx\_Package, unsigned char Package\_Length)

{

unsigned char i;

unsigned char RFM\_Tx\_Location = 0x00;

//Set RFM in Standby mode wait on mode ready

RFM\_Write(0x01,0x81);

/\*

while (digitalRead(DIO5) == LOW)

{

}

\*/

delay(10);

//Switch DIO0 to TxDone

RFM\_Write(0x40,0x40);

//Set carrier frequency

// 868.100 MHz / 61.035 Hz = 14222987 = 0xD9068B

RFM\_Write(0x06,0xD9);

RFM\_Write(0x07,0x06);

RFM\_Write(0x08,0x8B);

//SF7 BW 125 kHz

RFM\_Write(0x1E,0x74); //SF7 CRC On

RFM\_Write(0x1D,0x72); //125 kHz 4/5 coding rate explicit header mode

RFM\_Write(0x26,0x04); //Low datarate optimization off AGC auto on

//Set IQ to normal values

RFM\_Write(0x33,0x27);

RFM\_Write(0x3B,0x1D);

//Set payload length to the right length

RFM\_Write(0x22,Package\_Length);

//Get location of Tx part of FiFo

//RFM\_Tx\_Location = RFM\_Read(0x0E);

//Set SPI pointer to start of Tx part in FiFo

//RFM\_Write(0x0D,RFM\_Tx\_Location);

RFM\_Write(0x0D,0x80); // hardcoded fifo location according RFM95 specs

//Write Payload to FiFo

for (i = 0;i < Package\_Length; i++)

{

RFM\_Write(0x00,\*RFM\_Tx\_Package);

RFM\_Tx\_Package++;

}

//Switch RFM to Tx

RFM\_Write(0x01,0x83);

//Wait for TxDone

while(digitalRead(DIO0) == LOW)

{

}

//Switch RFM to sleep

RFM\_Write(0x01,0x00);

}

Da análise desta função verifica-se que a alteração da frequência central do canal (portadora) é conseguida à custa da alteração dos valores de três registos 0x06, 0x07 e 0x08. Os cálculos dos valores a colocar nesses registos são calculados a partir de[[5]](#footnote-5):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Frequência central** | **Decimal** | **Hexadecimal** | **0x06** | **0x07** | **0x08** |
| **0** | 868.1 | 14222987 | D9068B | D9 | 06 | 8B |
| **1** | 868.3 | 14226264 | D91358 | D9 | 13 | 58 |
| **2** | 868.5 | 14229540 | D92024 | D9 | 20 | 24 |
| **3** | 867.1 | 14206603 | D8C68B | D8 | C6 | 8B |
| **4** | 867.3 | 14209880 | D8D358 | D8 | D3 | 58 |
| **5** | 867.5 | 14213156 | D8E024 | D8 | E0 | 24 |
| **6** | 867.7 | 14216433 | D8ECF1 | D8 | EC | F1 |
| **7** | 867.9 | 14219710 | D8F9BE | D8 | F9 | BE |

O seguinte exceto de código foi escrito para gerar os valores dos registos apresentados na tabela com base no índice aleatório gerado pela função da seção anterior.

byte RF0x06[]={0xD9,0xD9,0xD9,0xD8,0xD8,0xD8,0xD8,0xD8};

byte RF0x07[]={0x06,0x13,0x20,0xC6,0xD3,0xE0,0xEC,0xF9};

byte RF0x08[]={0x8B,0x58,0x24,0x8B,0x58,0x24,0xF1,0xBE};

int arraylist[]={0,1,2,3,4,5,6,7};

int maxval = 8;

byte returnRandomRFChannel()

{

// random(min,max) gera um número aleatório com distribuição

// uniforme no intervalo [min,max[

int val = random(0,maxval);

int swapval=arraylist[val];

arraylist[val]=arraylist[maxval-1];

arraylist[maxval-1] = swapval;

maxval--;

if (maxval==0) maxval = 8; // Reset do contador

return(swapval);

}

void setup() {

// put your setup code here, to run once:

Serial.begin(2000000);

randomSeed(analogRead(A1));

}

void loop() {

// put your main code here, to run repeatedly:

byte res = returnRandomRFChannel();

Serial.print(res);

Serial.print(" > ");

Serial.print(RF0x06[res],HEX);

Serial.print(RF0x07[res],HEX);

Serial.println(RF0x08[res],HEX);

delay(1000);

}

//Set carrier frequency

// 868.100 MHz / 61.035 Hz = 14222987 = 0xD9068B

RFM\_Write(0x06,0xD9);

RFM\_Write(0x07,0x06);

RFM\_Write(0x08,0x8B);

// Registos para alterar a frequência da portadora

static const byte RF0x06[]={0xD9,0xD9,0xD9,0xD8,0xD8,0xD8,0xD8,0xD8};

static const byte RF0x07[]={0x06,0x13,0x20,0xC6,0xD3,0xE0,0xEC,0xF9};

static const byte RF0x08[]={0x8B,0x58,0x24,0x8B,0x58,0x24,0xF1,0xBE};

// Registos para alterar o SF

byte RF0x1E = 0x74; //SF7 CRC On

byte RF0x1D = 0x72; //125 kHz 4/5 coding rate explicit header mode

byte RF0x26 = 0x04; //Low datarate | optimization off | AGC auto on

// Registo 0x1E

// | SF |x |CC| x

// |b7|bc|b5|b4|c0|d0|e1|e0

//

// Spreading factor (SF)

// 0110 - 6

// 0111 - 7

// 1000 - 8

// 1001 - 9

// 1010 - 10

// 1011 - 11

// 1100 - 12

//

// CRC

// 0 - Header indicates CRC off

// 1 - Header indicates CRC on

//

// Registo 0x1D

// | BW | CR |Hr

// |b2|b2|b1|b0|c2|c1|c0|d0

//

// Valores de BW com interesse prático: 125 kHz ou 250 kHz (apenas para 868.3)

// 0111 - 125 kHz

// 1000 - 250 kHz

//

// Coding Rate (normalmente 4/5)

// 001 - 4/5

// 010 - 4/6

// 011 - 4/7

// 100 - 4/8

//

// Header type

// 0 - Explicit header

// 1 - Implicit header

//

// 0111 0010 -> 0x72

//

// Registo 0x26

//| xxx |MN|G | xx

// b3|b2|b1|b0|c0|d0|e1|e0|

//

// MN - Mobile node

// 0 - Static node

// 1 - Mobile node

//

// G - LNA gain

// 0 - Set by reg LnaGain

// 1 - Set by iunternal AGC loop

// 0000 0100 - > 4

int arraylist[]={0,1,2,3,4,5,6,7};

int maxval = 8;

byte returnRandomRFChannel()

{

// random(min,max) gera um número aleatório com distribuição

// uniforme no intervalo [min,max[

int val = random(0,maxval);

int swapval=arraylist[val];

arraylist[val]=arraylist[maxval-1];

arraylist[maxval-1] = swapval;

maxval--;

if (maxval==0) maxval = 8; // Reset do contador

return(swapval);

}

void setup() {

// put your setup code here, to run once:

Serial.begin(2000000);

randomSeed(analogRead(A1));

}

void loop() {

// put your main code here, to run repeatedly:

byte res = returnRandomRFChannel();

Serial.print(res);

Serial.print(" > ");

Serial.print(RF0x06[res],HEX);

Serial.print(RF0x07[res],HEX);

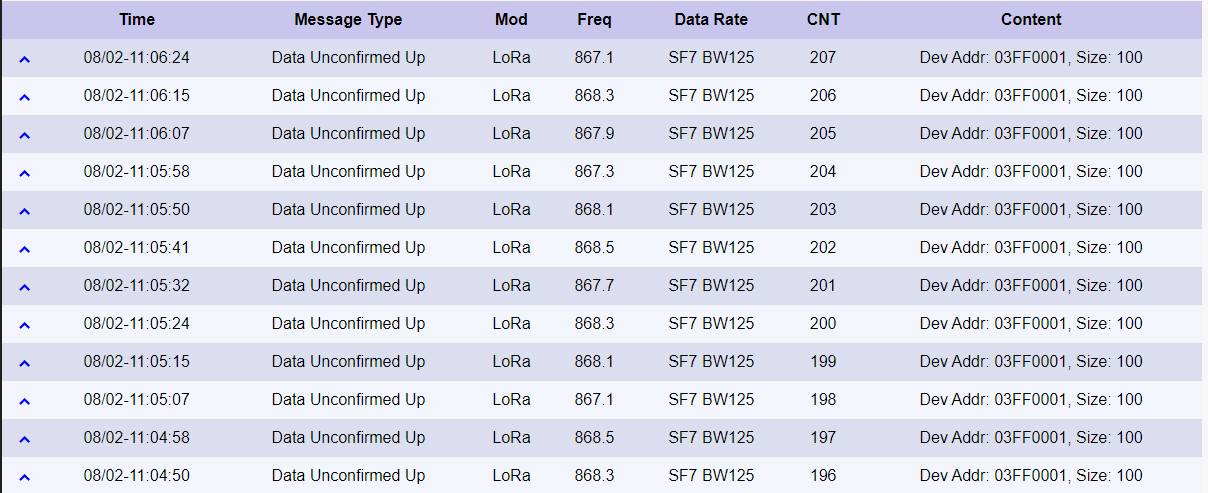
Serial.println(RF0x08[res],HEX);

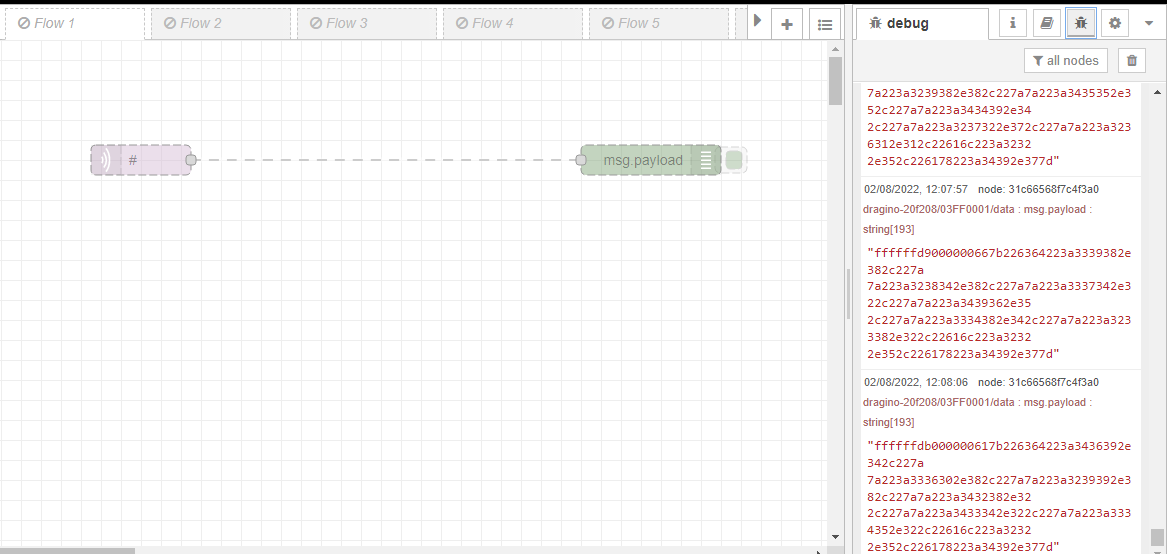
delay(1000);

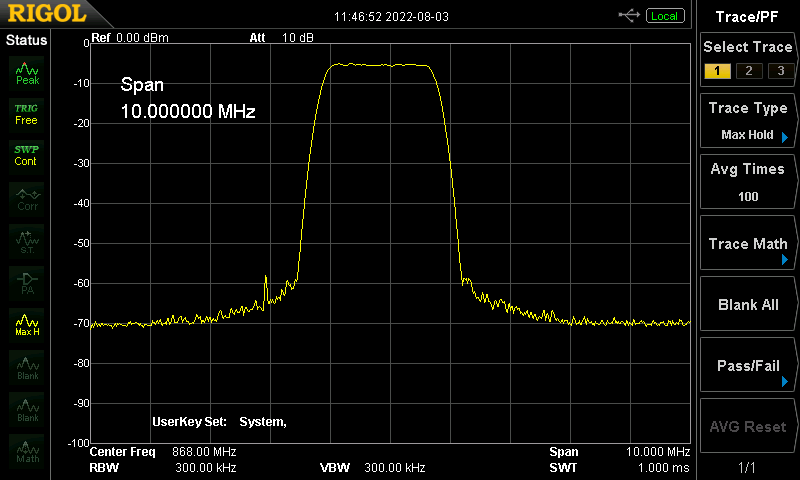
}

## Resultado final

Código disponível no github do projeto.

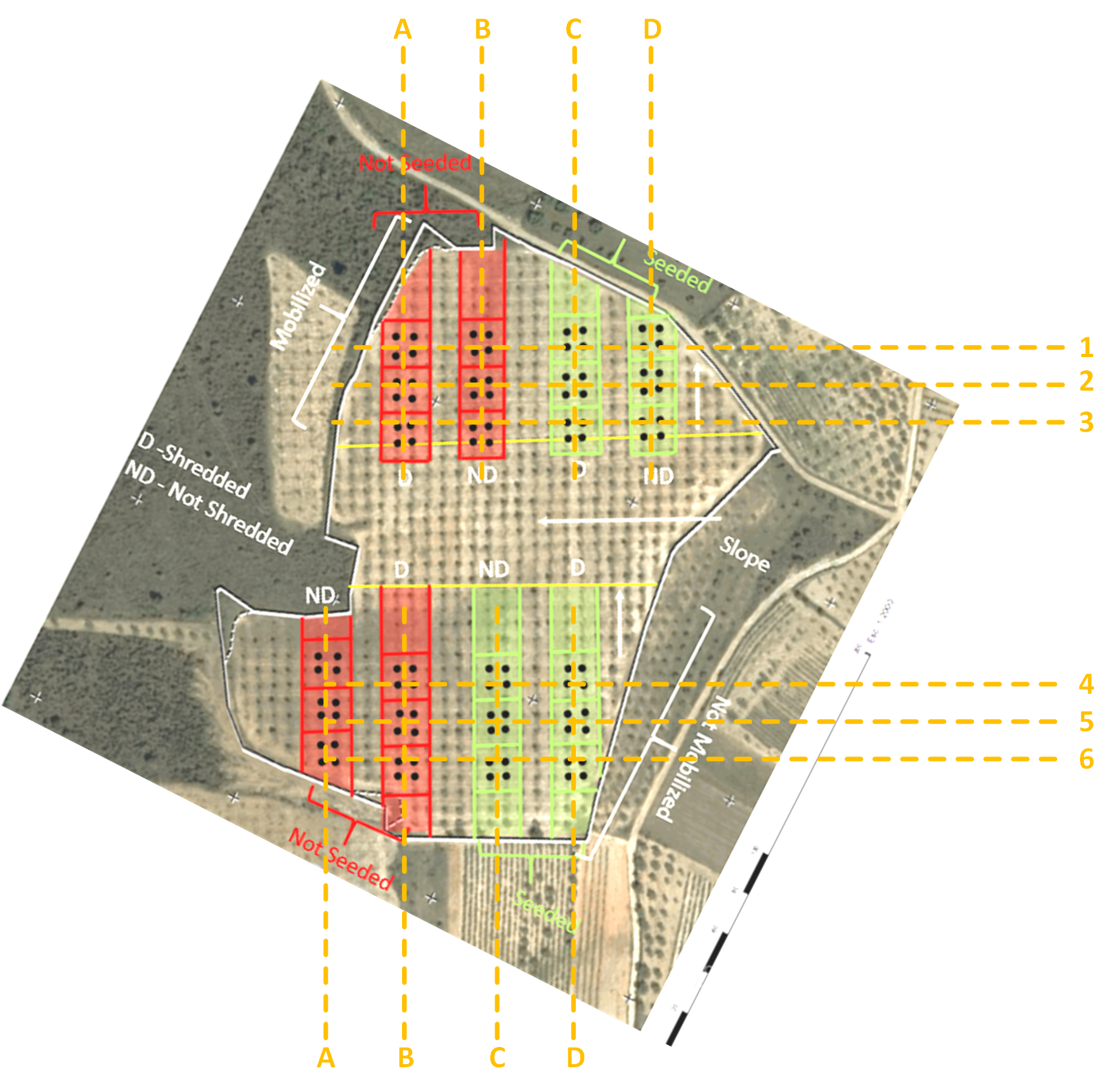


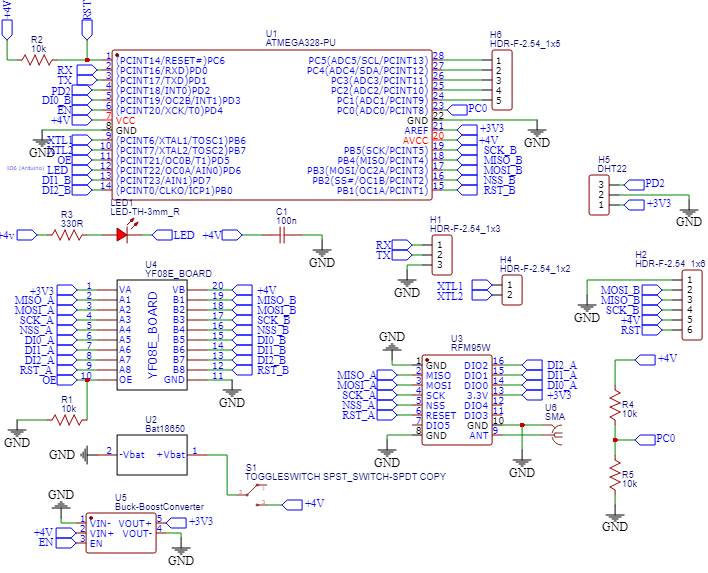




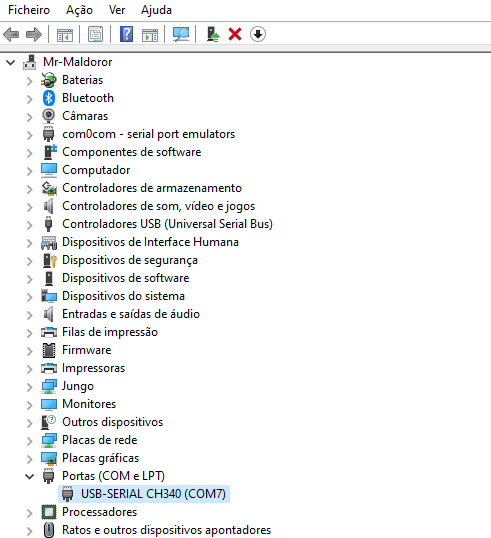
# Modelo de dados

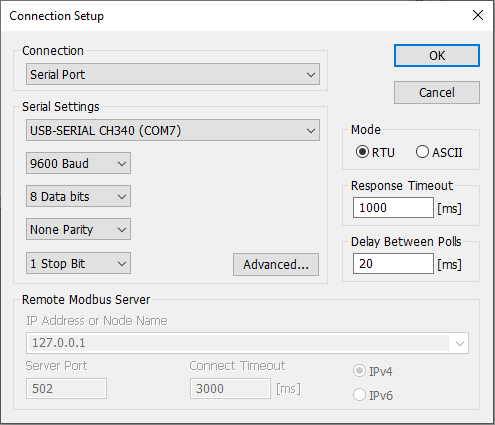
<https://github.com/jpcoelhoATipbDOTpt/MAN4HEALTH>

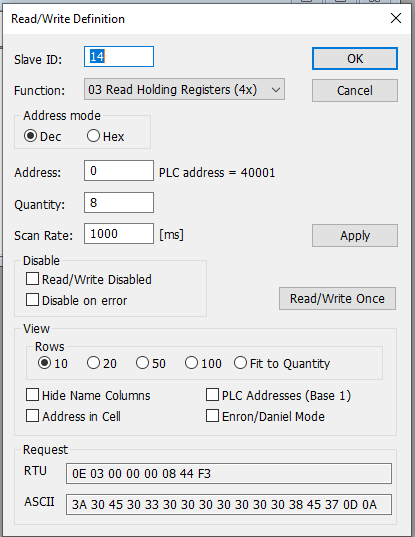


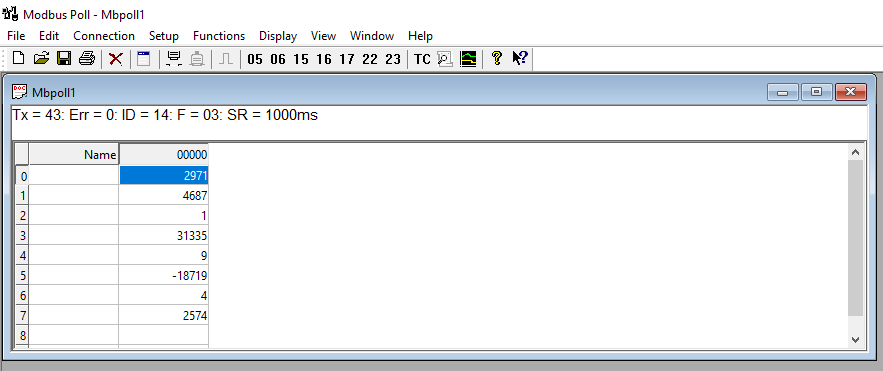


# Operação da estação meteorológica SenseCAP





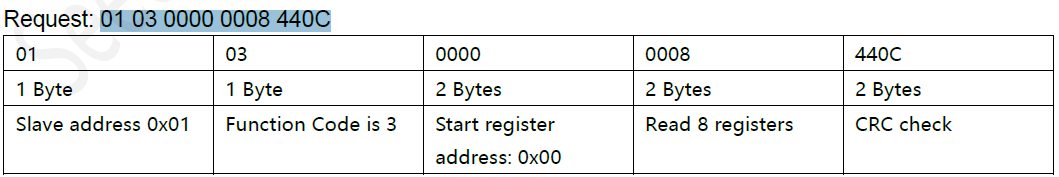


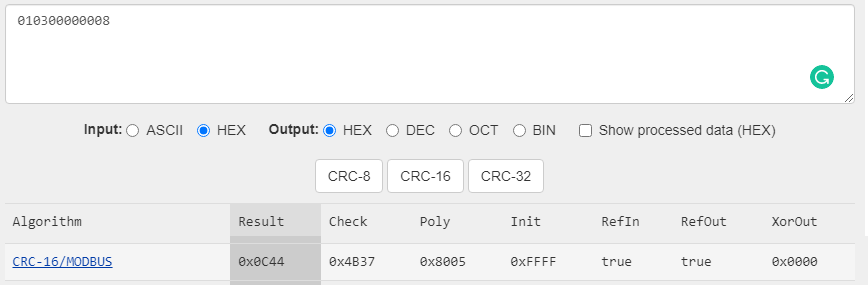


Usando o Serial Port Monitor

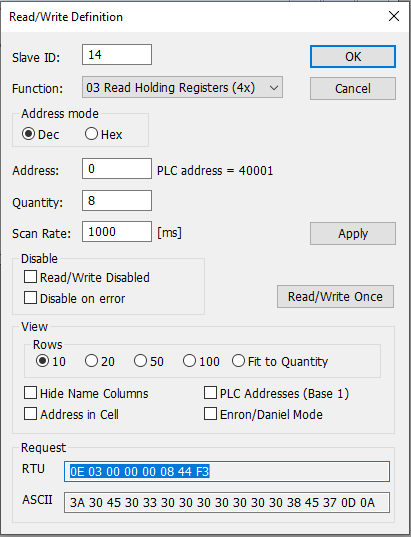
Calculo do CRC

<https://crccalc.com/>

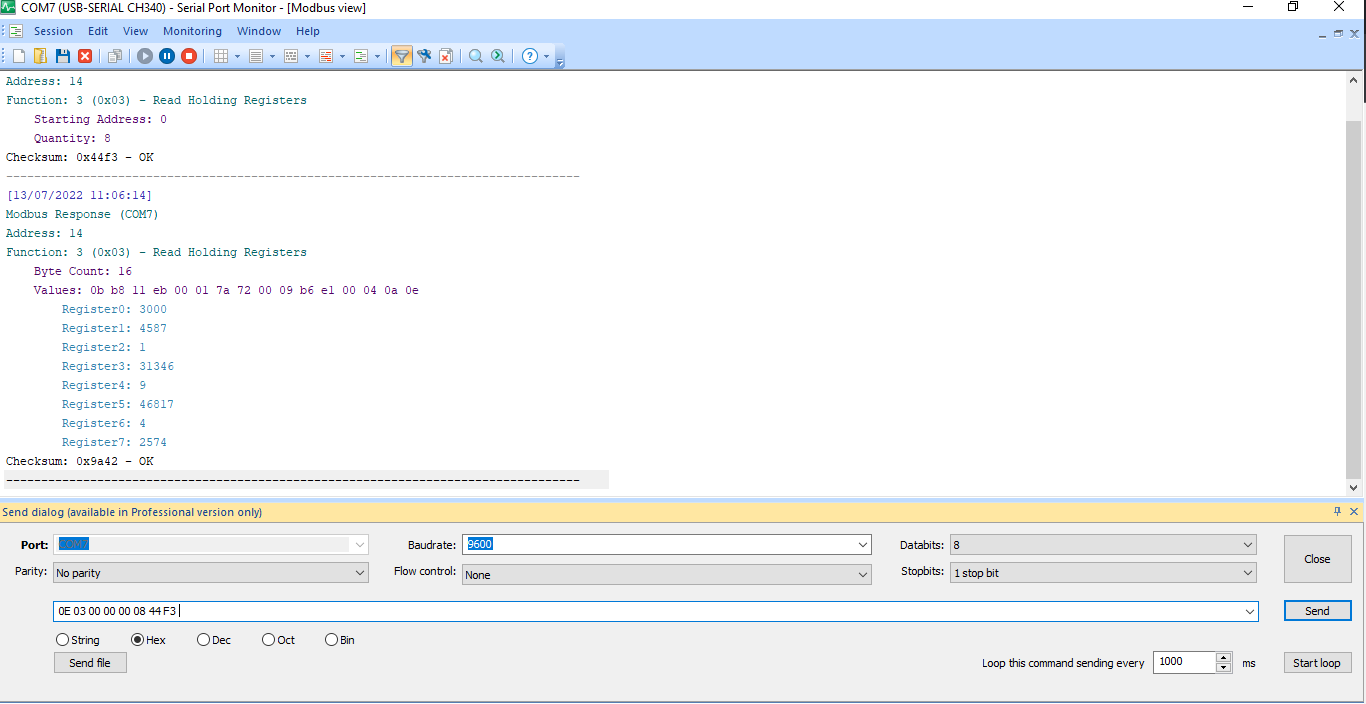




Notar a inversão dos bytes



0E 03 00 00 00 08 44 F3



1. Aparentemente, nestes ensaios é utilizado um ressonador externo para gerar o clock do microcontrolador. [↑](#footnote-ref-1)
2. If BOD is disabled in software, the BOD function is turned off immediately after entering the sleep mode. Upon wake-up from sleep, BOD is automatically enabled again. [↑](#footnote-ref-2)
3. The Things Network uses the non-standard SF9BW125 data rate for RX2 in Europe. If your devices use OTAA, this will be configured automatically when they join. If your devices use ABP, you will need to program this RX2 data rate into the devices in order to make them work with TTN. [↑](#footnote-ref-3)
4. Adaptado de <https://www.faqcode4u.com/faq/362441/unique-non-repeating-random-numbers-in-o1> [↑](#footnote-ref-4)
5. Register values must be modified only when device is in SLEEP or STAND-BY mode. [↑](#footnote-ref-5)