**Biblioteca do Projeto Arapuca - PUKA**

**Serial**

10/10/2020

Arduino Nano (Mega328P) tem uma única porta serial que é a USART0.

Não usar as funções seriais (classe “Serial.xxx”), pois comprometem a interrupção de recepção da porta serial 0.

Surge o erro: multiple definition of `\_\_vector\_xx'

Interrupções: USART\_RX\_vect e USART\_TX\_vect

Funções para a Serial 0

|  |  |  |
| --- | --- | --- |
|  |  |  |
| void | ser\_nrf\_param | (void) |
| void | ser\_wq\_blk | (long adr, word qtd) |
|  |  |  |
|  |  |  |
| void | ser\_cmdo\_letras; | (byte nr, byte x) |
| byte | ser\_pfn\_eeprom | (byte n) |
| byte | ser\_pfn\_dt | (byte n) |
| void | ser\_eeprom\_dump | (void) |
|  | void ser\_dump(byte \*vt, byte qtd) |  |
| void | ser\_eeprom\_config | (void) |
| void | ser\_eeprom\_dir | (void) |
|  |  |  |
|  |  |  |
| void | ser\_data\_hora | (char \*vet) |
|  |  |  |
| void | ser\_float | (float fx, byte prec) |
|  |  |  |
| void | ser\_dec32 | (long dt) |
| void | ser\_dec32u | (long dt) |
| void | ser\_dec32nz | (long dt) |
| void | ser\_dec32unz | (long dt) |
| void | ser\_hex32 | (long dt) |
|  |  |  |
| void | ser\_bcd16 | (word dt) |
| void | ser\_dec16 | (int dt) |
| void | ser\_dec16u | (word dt) |
| void | ser\_dec16nz | (int dt) |
| void | ser\_dec16unz | (word dt) |
| void | ser\_hex16 | (word dt) |
|  |  |  |
| void | ser\_bcd8 | (byte dt) |
| void | ser\_dec8 | (byte dt) |
| void | ser\_dec8u | (byte dt) |
| void | ser\_dec8nz | (byte dt) |
| void | ser\_dec8unz | (byte dt) |
| void | ser\_hex8 | (byte dt) |
|  |  |  |
| void | ser\_frase | (byte i) |
| void | ser\_str\_esc | (byte \*msg, byte qtd) |
| void | ser\_str | (byte \*msg) |
| void | ser\_char | (byte dt) |
| void | ser\_crlf | (byte qtd) |
| void | ser\_spc | (byte qtd) |
|  |  |  |
| void | seri\_fila\_prn | (void) |
| byte | seri\_num16u | (word \*nr) |
| byte | seri\_num16 | (int \*nr) |
| byte | seri\_bcd8 | (byte \*nr) |
| byte | seri\_num8 | (byte \*nr) |
| byte | seri\_str | (byte \*vt, byte qtd) |
| byte | seri\_letra | (char \*cha) |
| byte | seri\_cmdo\_sn | (void) |
| byte | seri\_cmdo\_x | (void) |
| byte | seri\_cmdo | (char \*argc, char \*argv, char limite) |
| void | seri\_come\_crlf | (void) |
| void | seri\_come\_spc | (void) |
| byte | seri\_xereta | (char \*cha) |
| void | seri\_config | (void) |
| char | seri\_vazia | (void) |
| char | seri\_poe | (char cha) |
| void | seri\_cheia | (void) |
| char | seri\_tira | (char \*cha) |
|  |  |  |
| - | ISR | (USART\_RX\_vect) |
|  |  |  |
| void | ser\_tx\_int\_EN | (void) |
| void | ser\_tx\_int\_en | (void) |
| void | ser\_rx\_int\_EN | (void) |
| void | ser\_rx\_int\_en | (void) |
|  |  |  |
| void | ser\_tx\_EN | (void) |
| void | ser\_tx\_en | (void) |
| void | ser\_rx\_EN | (void) |
| void | ser\_rx\_en | (void) |
|  |  |  |
| void | ser\_config\_16MHz | (char br) |
| void | ser\_config\_1MHz | (char br) |
|  |  |  |
|  |  |  |
|  |  |  |

* void **ser\_nrf\_param** (void)

Imprime o canal do rádio, a potência de o baudrate bps.

76 -18dBm 1Mbps

* byte **ser\_wq\_temp** (long adr)

Imprimir do dados de temperatura até encontrar 0xFF.

Contador temp Vbat

0003473408 000 000 053 048 054 047 049 048 047 050 049 058 050 051 000 255

0003473424 000 001 048 054 047 049 048 047 050 049 058 050 051 058 048 055

0003473440 000 002 068 072 005 255 255 255 255 255 255 255 255 255 255 255

0003473456 000 003 255 255 255 255 255 255 255 255 255 255 255 255 255 255

* byte **ser\_wq\_blk** (long adr, word qtd)

Imprimir em decimal um bloco de memória Flash.

Endereço e depois 16 bytes.

Endereço ------------------- dados -------------------------------------

0003473408 000 000 053 048 054 047 049 048 047 050 049 058 050 051 000 255

0003473424 000 001 048 054 047 049 048 047 050 049 058 050 051 058 048 055

0003473440 000 002 068 072 005 255 255 255 255 255 255 255 255 255 255 255

0003473456 000 003 255 255 255 255 255 255 255 255 255 255 255 255 255 255

* void **ser\_cmdo\_letras** (byte x)

Imprimir as 3 letras que caracterizam um comando.

* byte **ser\_pfn\_eeprom** (byte n)

Imprimir em hexadecimal os dados da cópia da EEPROM.

Imprime no mesmo formato de “ser\_pfn\_dt”.

* byte **ser\_pfn\_dt** (byte n)

Imprimir em hexadecimal os dados da fermentação n até chegar ao fim da memória ou encontrar um registrado como todos os dados iguais a 0xFF.

Retorna TRUE se chegou ao fim da memória

Retorna FALSE se encontrou um registro todo igual a 0xFF

Fermentacao 5

00050000: 00 00 05 74 65 73 74 65 31 00 FF FF FF FF FF FF

00050010: 00 01 30 31 2F 30 31 2F 30 30 3A 30 30 3A 30 35

00050020: 00 02 44 48 05 FF FF FF FF FF FF FF FF FF FF FF

00050030: 00 03 FF FF FF FF FF FF FF FF FF FF FF FF FF FF

00050040: 00 04 00 30 00 33 00 46 00 E8 00 00 03 FF FF FF

00050050: 00 05 00 30 00 33 00 46 00 EC 00 00 03 FF FF FF

00050060: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF

* void **ser\_data\_hora** (char \*vet)

Imprimir data e hora na porta serial: dd/mm/aa hh:mm:ss.

O vetor vet é cópia dos registardores do DS3231.

* void **ser\_float** (float f, byte prec)

No Arduino, double e float têm a mesma precisão

Escreve na serial float **f** com **prec** casas após a vírgula e com sinal.

Formato = + xxx xxx xxx , ddd ddd ddd ddd (usar char msg[24])

Limite da parte inteira = 9 dígitos. Limite da parte fracionária = 12 dígitos.

Caso ultrapasse esses os limites imprime ###, ###

O máximo é 999.999.999,999999. Se ultrapassar o máximo, escreve ###,###.

Na verdade, o máximo é 999.999.999,999967. Exemplos:

999.999.999,0 🡪 imprime 999.999.936,000000 (por causa da precisão da representação)

876.543.210,123456789 🡪 imprime 876543232,000000 (por causa da precisão da representação)

* void **ser\_dec32** (long dt)

Escrever na serial 32 bits em decimal, com sinal e com zeros à esquerda.

* void **ser\_dec32u** (long dt)

Escrever na serial 32 bits em decimal, sem sinal e com zeros à esquerda.

* void **ser\_dec32nz** (long dt)

Escrever na serial 32 bits em decimal, com sinal e sem zeros à esquerda.

* void **ser\_dec32unz** (long dt)

Escrever na serial 32 bits em decimal, sem sinal e sem zeros à esquerda.

* void **ser\_hex32** (long dt)

Escrever na serial 32 bits em hexadecimal.

* void **ser\_bcd16** (word dt)

Escrever na serial 16 bits em BCD, são 4 dígitos BCD sem sinal e com zeros à esquerda.

* void **ser\_dec16** (word dt)

Escrever na serial 16 bits em decimal, com sinal e com zeros à esquerda.

* void **ser\_dec16u** (word dt)

Escrever na serial 16 bits em decimal, sem sinal e com zeros à esquerda.

* void **ser\_dec16nz** (word dt)

Escrever na serial 16 bits em decimal, com sinal e sem zeros à esquerda.

* void **ser\_dec16unz** (word dt)

Escrever na serial 16 bits em decimal, sem sinal e sem zeros à esquerda.

* void **ser\_hex16** (word dt)

Escrever na serial 16 bits em hexadecimal.

* void **ser\_bcd8** (byte dt)

Escrever na serial 8 bits em BCD, são dois dígitos BCD sem sinal e com zeros à esquerda.

* void **ser\_dec8** (byte dt)

Escrever na serial 8 bits em decimal, com sinal e com zeros à esquerda.

* void **ser\_dec8u** (byte dt)

Escrever na serial 8 bits em decimal, sem sinal e com zeros à esquerda.

* void **ser\_dec8nz** (byte dt)

Escrever na serial 8 bits em decimal, com sinal e sem zeros à esquerda.

* void **ser\_dec8unz** (byte dt)

Escrever na serial 8 bits em decimal, sem sinal e sem zeros à esquerda.

* void **ser\_hex8** (byte dt)

Escrever na serial 8 bits em hexadecimal.

* void **ser\_frase** (byte i)

Imprimir a frase “i” que está armazenada na memória de programa.

* void **ser\_str\_esc** (byte \*msg, bye qtd)

Envia pela serial qtd letras da string apontada por msg. Imprime os caracteres de scape.

LF (0xA)🡪\n CR (0xD)🡪\r NULL🡪 \0

* void **ser\_str** (byte \*msg)

Envia pela serial a string apontada por msg.

* void **ser\_char** (byte cha)

Envia pela serial o caractere cha. Só retorna após a serial completar o envio.

* void **ser\_crlf** (byte qtd)

Escrever na serial qtd de pares CR (0xD = ‘\r’) e LF (0xA = ‘\n’).

* void **ser\_spc** (byte qtd)

Escrever na serial qtd brancos (0x20).

* void **seri\_fila\_prn** (void)

Imprime toda da fila de entrada: seri\_fila, seri\_pin e seri\_pout.

Pin=escreve e depois avança;

Pout=avança e depois lê;

Fila vazia se pin+1=pout;

Fila cheia se pin=pout.

* void **seri\_num16** u(int \*nr)

Retirar um número de 16 bits (sem sinal) da fila SERI e o retorna no ponteiro nr.

Monta ao número lendo os dígitos na base 10 (0, 1, ..., 65.535).

Consome CR e LF que estiverem antes do número.

Retorna TRUE --> retirou um número.

Retorna FALSE --> Não retirou número, não alterou a fila.

* void **seri\_num16** (int \*nr)

Retirar um número de 16 bits da fila SERI e o retorna no ponteiro nr.

Monta ao número lendo os dígitos na base 10 (-32.768, ..., 0, 1, ..., 32767).

Consome CR e LF que estiverem antes do número.

Retorna TRUE --> retirou um número.

Retorna FALSE --> Não retirou número, não alterou a fila.

* void **seri\_bcd8** (int \*nr)

Na certeza de o próximo elemento da fila é um número.

Retirar um número de 8 bits da fila SERI, considera de até 2 dígitos BCD.

Monta ao número lendo os dígitos na base 10 (0, 1, ..., 99).

Retorna TRUE --> retirou um número.

Retorna FALSE --> não retirou número.

* void **seri\_num8** (int \*nr)

Retirar um número de 8 bits da fila SERI, considera sem sinal.

Monta ao número lendo os dígitos na base 10 (0, 1, ..., 255).

Consome CR e LF que estiverem antes do número.

Retorna TRUE --> retirou um número.

Retorna FALSE --> Não retirou número, não alterou a fila.

* byte **seri\_str** (byte \*vt, byte qtd)

Retirar uma string da fila SERI e armazenar em vt dentro do limite qtd.

Consome CR e LF que existirem antes.

Os caracteres , . ; Branco CR LF marcam fim da string.

A função consome apenas , . ; branco. Não consome o CR LF.

Retorna quantidade de letras armazenadas na string. Se retornar 0, significa que nada foi lido.

* void **seri\_letra** (char \*cha)

Retirar uma letra da fila SERI. Só não aceita números 0x30 até 0x39

Consome CR e LF que estiverem antes da letra.

Retorna TRUE --> retirou uma letra.

Retorna FALSE --> Não retirou uma letra, não alterou a fila

* char **seri\_cmdo\_sn** (void){

Função para pedir confirmação de um comando perigoso. Apaga toda a fila de entrada.

Imprime a mensagem “(S/N)” e espera usuário teclar algo

Se teclar S ou s, retorna TRUE.

Se teclar algo diferente, retorna FALSE.

* char **seri\_cmdo\_x** (void){

Função para verificar se usuário pressionou “X” ou “x”.

Se teclar X ou x, retorna TRUE.

Se teclar algo diferente, retorna FALSE.

* char **seri\_cmdo** (char \*argc, char \*argv, char limite){

Receber uma linha de comando da fila de entrada (seri). A linha termina com CR ou LF.

Ao digitar, os argumentos devem ser separados por "branco".

Depois de separados, cada argumento é terminado em '\0'.

Retorna a quantidade de argumentos recebidos.

argc[] = posição de início dos comandos.

argv[] = vetor com todos os comandos.

limite = quantidade máxima de caracteres a serem recebidos.

* void **seri\_come\_crlf** (void)

Avança sobre a fila de forma a consumir CR (0xD = ‘\r’) ou LF (0xA = ‘\n’).

* void **seri\_come\_spc** (void)

Avança sobre a fila de forma a consumir os espaços (0x20).

* void **seri\_xereta** (char \*cha)

Informa em cha o próximo byte da fila. Não altera o estado dos ponteiros.

Não pode ser usada dentro de interrupções.

* void **seri\_config** (void)

Inicializar fila de recepção serial.

* byte **seri\_vazia** (void);

Verificar se a fila seri está vazia.

TRUE = vazia e

FALSE = tem alguma coisa na fila

* char **seri\_poe** (char cha)

Colocar um byte na fila de recepção. Ela é chamada pela ISR serial de recepção.

Retorna TRUE se conseguiu.

Retorna FALSE se fila cheia, mas antes chama função seri\_cheia().

* void **seri\_cheia** (void)

Imprime mensagem de erro na porta serial para indicar que a fila serial RX está cheia.

* char **seri\_tira** (char \*cha)

Chamada de fora da interrupção. Desabilita temporariamente todas as interrupções.

Coloca no ponteiro \*char o byte retirado da fila de transmissão e retorna TRUE.

Se retornar FALSE é porque a fila está vazia.

* **ISR (USART\_RX\_vect)**

ISR: Interrupção por dado recebido. Coloca dado recebido na fila de entrada.

* **void ser\_tx\_int\_EN (void)**

Habilita a interrupção por transmissão serial.

* **void ser\_tx\_int\_en (void)**

Desabilita a interrupção por transmissão serial.

* **void ser\_rx\_int\_EN (void)**

Habilita a interrupção por recepção serial.

* **void ser\_rx\_int\_en (void)**

Desabilita a interrupção por recepção serial.

* **void ser\_tx\_EN (void)**

Habilita a transmissão serial.

* **void ser\_tx\_en (void)**

Desabilita a transmissão serial.

* **void ser\_rx\_EN (void)**

Habilita a recepção serial.

* **void ser\_rx\_en (void)**

Desabilita a recepção serial.

* **void ser\_config\_16MHz (char BR)**

Faz a configuração da porta serial para operar na taxa BR, considerando relógio de CPU em 16 MHz.

Usa dobrador de velocidade.

Habilita transmissão e recepção serial.

Não habilita as interrupções.

Opções: BR\_9600, BR\_19200, BR\_28800, BR\_38400, BR\_57600.

* **void ser\_config\_1MHz (char BR)**

Faz a configuração da porta serial para operar na taxa BR, considerando relógio de CPU em 1 MHz.

Usa dobrador de velocidade.

Habilita transmissão e recepção serial.

Não habilita as interrupções.

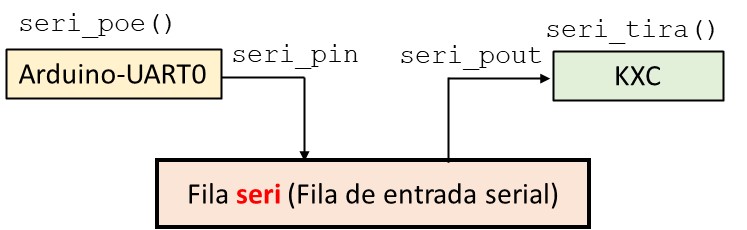
Opções: BR\_9600, BR\_19200, BR\_28800, BR\_38400, BR\_57600.

*Cálculo para os Baud Rate usados*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Baud Rate** | **16 MHz (U2X0 = 1)** | |  | **1 MHz (U2X0 = 1)** | |
| **UBRR** | **Erro** |  | **UBRR** | **Erro** |
| 9.600 | 207 | 0,2% |  | 12 | 0,6% |
| 19.200 | 103 | 0,2% |  | 6 | 7,0% |
| 28.800 | 68 | 0,6% |  | 3 | 8,5% |
| 38.400 | 51 | 0,2% |  | 2 | 8,5% |
| 57.600 | 34 | 0,8% |  | 1 | 8,5% |
| 115.200 | 16 | 2,1% |  | 0 | 8,5% |

*Gabarito dos registradores de configuração da porta serial USART0*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **UCSR0A** | RXC**0** | TXC**0** | UDRE**0** | FE**0** | DOR**0** | UPE**0** | U2X**0** | MPCM**0** |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **UCSR0B** | RXCIE**0** | TXCIE**0** | UDRIE**0** | RXEN**0** | TXEN**0** | UCSZ**0**2 | RXB8**0** | TXB8**0** |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| **UCSR0C** | UMSEL**0**1 | UMSEL**0**0 | UPM**0**1 | UPM**0**0 | USBS**0** | UCSZ**0**1 | UCSZ**0**0 | UCPOL**0** |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

****

Fila circular para recepção

Fila TX

volatile char seri\_fila [SERI\_FILA\_TAM]; //Espaço para a fila serial de recepção

volatile byte seri\_pin, seri\_pout; //Ponteiros para usar a fila

Fila Vazia 🡪 sero\_pout+1 = seri\_pin e

Fila Cheia 🡪 sero\_pout = seri\_pin e

**strings**

19/10/2020

Gerar e operar strings para facilitar a impressão na porta serial etc.

Funções

|  |  |  |
| --- | --- | --- |
| word | str\_conta | (char \*vt) |
| long | str\_2\_num | (byte \*vt) |
| byte | eh\_hexa | (byte letra) |
| byte | eh\_dec | (byte letra) |
| void | str\_maiusc | (byte \*vt) |
|  |  |  |
| void | str\_data\_hora | (char \*dh, char \*msg) |
|  |  |  |
| byte | str\_copia\_zf | (byte \*ft, byte \*dest) |
| byte | str\_copia | (byte \*ft, byte \*dest) |
| byte | str\_tam | (byte \*ft) |
| void | str\_rmvz\_u | (char \*msg) |
| void | str\_rmvz\_s | (char \*msg) |
| void | str\_float | (float f, byte prec, char \*msg) |
|  |  |  |
| void | str\_dec32 | (long c, char \*msg) |
| void | str\_dec32u | (long c, char \*msg) |
| void | str\_hex32 | (long c, char \*msg) |
|  |  |  |
| void | str\_bcd16 | (word c, char \*msg) |
| void | str\_dec16 | (int c, char \*msg) |
| void | str\_dec16u | (word c, char \*msg) |
| void | str\_hex16 | (word c, char \*msg) |
|  |  |  |
| void | str\_bcd8 | (byte c, char \*msg) |
| void | str\_dec8 | (byte c, char \*msg) |
| void | str\_dec8u | (char c, char \*msg) |
| void | str\_hex8 | (byte c, char \*msg) |
|  |  |  |
| void | str\_spc | (char qtd, char \*msg) |
| void | str\_crlf | (char qtd, char \*msg) |
| void | str\_cr | (char qtd, char \*msg) |
| void | str\_lf | (char qtd, char \*msg) |
| byte | asc\_nib | (byte asc) |
|  |  |  |

* word **str\_conta** (byte \*vt)

Contar o tamanho de uma string terminada em zero.

* long **str\_2\_num** (byte \*vt)

Converter para número uma string terminada em zero. Retorna long.

Aceita Hexadecimal **0x**ABCD ou decimal com ou sem sinal (0098, +123, -456, 0x12Bcf).

Não aceita octal ou binário.

Se houver caractere inválido, retorna ZERO.

* byte **eh\_hexa** (byte letra)

Verificar se uma letra (um caractere) é um Hexa válido.

TRUE --> caso seja um hexa válido (0, 1, ..., 9, A, ..., F).

False --> caso não seja um Hexa válido.

* byte **eh\_dec** (byte letra)

Verificar se uma letra (um caractere) é um dígito decimal válido.

TRUE --> caso seja um hexa válido (0, 1, ..., 9).

False --> caso não seja um Hexa válido.

* void **str\_maiusc** (byte \*vt)

Converter para maiúsculas somente as letras de uma string terminada em zero.

* void **str\_data\_hora** (char \*dh, char \*msg)

Escreve em msg a data e hora para depois ser impressa. dd/mm/aa hh/mm/ss.

É preciso de 18 posições de msg.

O vetor dh deve estar no formato dos registradores de DS3231.

* byte **str\_copia\_zf** (byte \*ft, byte \*dest)

Faz cópia de uma string e retorna tamanho string copiada. Inclui o Zero Final.

* byte **str\_copia** (byte \*ft, byte \*dest)

Faz cópia de uma string e retorna tamanho string copiada. NÃO inclui o Zero Final.

* byte **str\_tam** (byte \*ft)

Retorna tamanho da string, não conta o zero final.

* void **str\_rmvz\_u** (char \*msg)

Remove os zeros à esquerda da string de número sem sinal que está em msg.

* void **str\_rmvz\_s** (char \*msg)

Remove os zeros à esquerda da string de número com sinal que está em msg.

* void **str\_float** (float f, byte prec, char \*msg)

No Arduino, double e float têm a mesma precisão

Escreve em msg o float fx com prec casas após a vírgula e apresenta o sinal.

Formato = + xxx xxx xxx , ddd ddd ddd ddd (usar char msg[24])

Limite da parte inteira = 9 dígitos.

Limite da parte fracionária = 12 dígitos.

Caso ultrapasse os limites imprime ###, ###

O máximo é 999.999.999,999999. Se ultrapassar o máximo, escreve ###,###.

Na verdade, o máximo é 999.999.999,999967. Exemplos:

999.999.999,0 🡪 imprime 999.999.936,000000 (por causa da precisão da representação)

876.543.210,123456789 🡪 imprime 876543232,000000 (por causa da precisão da representação)

* void **str\_dec32** (long c, char \*msg)

Escreve em msg o (long) decimal 32 bits com sinal e com zeros à esquerda.

Usar char msg[12], pois +4 294 967 295 \0 - 12 posições.

* void **str\_dec32u** (long c, char \*msg)

Escreve em msg o (unsigned long) decimal 32 bits sem sinal e com zeros à esquerda.

Usar char msg[12], pois +4 294 967 295 \0 - 12 posições.

* void **str\_hex32** (long c, char \*msg)

Escreve em msg o (long) hexadecimal de 32 bits. Usar char msg[9].

* void **str\_bcd16** (word c, char \*msg)

Escreve em msg o (word) BCD sem sinal e com zeros à esquerda.

Usar char msg[5], pois 1 234 \0 - 5 posições.

* void **str\_dec16** (int c, char \*msg)

Escreve em msg o (int) decimal 16 bits com sinal e com zeros à esquerda.

Usar char msg[7], pois +67 295 \0 - 7 posições.

* void **str\_dec16u** (word c, char \*msg)

Escreve em msg o (word) decimal 16 bits sem sinal e com zeros à esquerda.

Usar char msg[7], pois +67 295 \0 - 7 posições.

* void **str\_hex16** (word c, char \*msg)

Escreve em msg o (word) hexadecimal de 16 bits. Usar char msg[5].

* void **str\_bcd8** (char c, char \*msg)

Escreve em msg o (char) BCD sem sinal e com zeros à esquerda.

Usar char msg[3], pois 12 \0 - 3 posições.

* void **str\_dec8** (char c, char \*msg)

Escreve em msg o (char) decimal 8 bits com sinal e com zeros à esquerda.

Usar char msg[5], pois +123 \0 - 5 posições.

* void **str\_dec8u** (byte c, char \*msg)

Escreve em msg o (byte) decimal 8 bits sem sinal e com zeros à esquerda.

Usar char msg[5], pois +295 \0 - 5 posições.

* void **str\_hex8** (byte c, char \*msg)

Escreve em msg o (byte) hexadecimal de 8 bits. Usar char msg[3].

* void **str\_spc**(char qtd, char \*msg)

Escrever em msg uma qtd de espaços = 0x20 (Espaço em Branco). Prever msg com tamanho adequado.

* void **str\_crlf**(char qtd, char \*msg)

Escrever em msg uma qtd de pares CR (‘\r’=0xD) e LF (‘\n’=0xA). Prever msg com tamanho adequado.

* void **str\_cr**(char qtd, char \*msg)

Escrever em msg uma qtd de CR (‘\r’=0xD). Prever msg com tamanho adequado.

* void **str\_lf**(char qtd, char \*msg)

Escrever em msg uma qtd de LF (‘\n’=0xA). Prever msg com tamanho adequado.

* byte **asc\_nib** (byte asc)

Converter ASCII em nibble.

ASCII = 0x30, 0x31, ..., 0x49, 0x41, ..., 0x46. (0,1,2, ..., F)

**Timers**

19/10/2020

Usa os Timer 1 (principal) e Timer 2 (gerar atrasos de 50 micro seg)

Gerar e operar strings para facilitar a impressão na porta serial etc.

Funções

|  |  |  |
| --- | --- | --- |
|  |  |  |
| void | espera\_100ms | (word dur) |
| void | espera\_50us | (word dur) |
|  | ISR | (TIMER1\_OVF\_vect) |
| void | t1\_config | (void) |
| void | clk\_1MHz | (void) |

* void **espera\_100ms** (word dur)

Esperar “dur”\*100ms. Tem a imprecisão de +/- 100 ms.

Primeiro se sincroniza com a flag\_100ms e depois inicia a contagem do tempo.

* void **espera\_50us** (word dur)

Esperar “dur”\*50 micro seg. Erra 10% para mais. Usa o Timer 2.

* **ISR**(TIMER1\_OVF\_vect)

Rotina para atender a interrupção ICF1.

Controla o pisca-pisca dos leds. Ver explicação mais abaixo.

Acontece a cada 20 ms, preparada para clock de 1 MHz.

Atualiza as variáveis e flags

* + cont\_frac\_seg 🡪 contador de 5, pois 5 x 20 ms = 100 ms.
  + flag\_100ms 🡪 vai a 1 a cada 100 ms.
  + cont\_seg 🡪 contador de 10, para medir 1 seg.
  + flag\_seg 🡪 vai a 1 a cada segundo.
  + cont\_min 🡪 contador de 60, para medir 1 minuto.
  + flag\_min 🡪 vai a 1 a cada minuto.
* void **t1\_config** (void)

Configurar Timer 1, CLK = 1 MHz, Fast PWM e modo invertido.

Preparado para contagens de 20 ms.

Fast PWM para acionar os leds.

* void **clk\_1MHz** (void)

Configura o clock da CPU para 1 MHz.

**Configuração do Timer 1 para interrupções de 20 ms (50 Hz), com clock de 1 MHz**

CLK = 1 MHz 🡪 1.000.000 x 0,02 = 20.000 🡪 ICR1 = 20.000-1 = 19.999

#define TIMER1\_20ms 19999

Gabarito para configurar os registradores do TC1, Modo 14 (Fast PWM e ICR1=TOPO)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **TCCR1A** | COM1A1 | COM1A0 | COM1B1 | COM1B0 | COM1C1 | COM1C0 | WGM11 | WGM10 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| **TCCR1B** | IC1C1 | ICES1 | - | WGM13 | WGM12 | CS12 | CS11 | CS10 |
| 0 | 0 |  | 1 | 1 | 0 | 0 | 1 |
| **TCCR1C** | FOC1A | FOC1B | FOC1C | - | - | - | - | - |
| 0 | 0 | 0 |  |  |  |  |  |
| **TIMSK1** | - | - | ICIE1 | - | OCIE1C | OCIE1B | OCIE1A | TOIE1 |
|  |  | 0 |  | 0 | 0 | 0 | 1 |
| **TIRF1** | - | - | ICF1 | - | OCF1C | OCF1B | OCF1A | TOV1 |
|  |  |  |  |  |  |  |  |

CS12 CS11 CS10 = 0 0 1 🡪 Seleciona CLK sem divisor, logo seleciona 1 MHz.

WGM13 WGM12 WGM11 WGM10 = 1 1 1 0 🡪 Modo 14, Fast PWM com Topo = ICR1 (ICR1 = 19.999).

TOIE = 1 🡪 interrupção por overflow do contador, interrompe a cada 20 ms.

COM1A1 COM1A0 = 1 1 🡪 PWM Modo invertido: OC1A = 1 se TCNT1 = OCR1A e OC1A = 0 se TCNT1 = 0.

COM1B1 COM1B0 = 1 1 🡪 PWM Modo invertido: OC1B = 1 se TCNT1 = OCR1B e OC1B = 0 se TCNT1 = 0.

**Sobre o ADC**

Temos uma interrupção a cada 20 ms, ou seja, 50 Hz. Incrementar variável adc\_cont;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adc\_cont |  |  | Adc\_cont |  |
| 0 | Config Hall\_Prox, Disp |  | 13 | 4-Ler e Somar ADC, calcular |
| 1 | Ler e jogar fora, Disp |  | 14 | Config Hall VBAT, Disp |
| 2 | Ler e jogar fora, Disp |  | 15 | Ler e jogar fora, Disp |
| 3 | 1-Ler e Somar ADC, Disp |  | 16 | Ler e jogar fora, Disp |
| 4 | 2-Ler e Somar ADC, Disp |  | 17 | 1-Ler e Somar ADC, Disp |
| 5 | 3-Ler e Somar ADC, Disp |  | 18 | 2-Ler e Somar ADC, Disp |
| 6 | 4-Ler e Somar ADC, calcular |  | 19 | 3-Ler e Somar ADC, Disp |
| 7 | Config Hall\_Dist, Disp |  | 20 | 4-Ler e Somar ADC, calcular |
| 8 | Ler e jogar fora, Disp |  | 21 | Nada |
| 9 | Ler e jogar fora, Disp |  | 22 | Nada |
| 10 | 1-Ler e Somar ADC, Disp |  | 23 | Nada |
| 11 | 2-Ler e Somar ADC, Disp |  | 24 | Nada |
| 12 | 3-Ler e Somar ADC, Disp |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Sobre os leds**

O modo PWM foi adotado para acender os leds. Assim, o led nunc

No modo direto, com potência = 0, ainda surge um pulso com duração de 1 µs (conferir).

Foi escolhido o modo invertido para evitar esse

OC1A = PB1 = Pino 9 🡪 Led Azul.

OC1B = PB2 = Pino 10 🡪 Led Vermelho.

Para acender o led, programa-se PWM com carga de 50% 🡪 0,5\*TIMER1\_20ms.

Então o led nunca fica verdadeiramente aceso, fica 10 ms aceso e 10 ms apagado.

#define TIMER1\_MAX TIMER1\_20ms

#define POT\_AZ (word)(0.5\*TIMER1\_MAX) //Azul acende

#define POT\_az (word) (TIMER1\_MAX+1) //Azul apaga

#define POT\_VM (word)(0.5\*TIMER1\_MAX) //Vermelho acende

#define POT\_vm (word)(TIMER1\_MAX+1) //Vermelho apaga

Os valores são independentes para se poder escolher o ciclo de carga de cada led.

OCR1A = POT\_AZ; 🡪 acende led azul e OCR1A = POT\_az; 🡪 apaga led azul

OCR1B = POT\_VM; 🡪 acende led vermelho e OCR1B = POT\_vm; 🡪 apaga led vermelho

Notar que POT\_vm = POT\_az = TIMER1\_MAX+1 🡪 1 valor além da contagem, com isso o contador nunca chega a este valor, ou seja, nunca temos TCNT1 = OCR1A (ou OCR1B). O OC1A (OC1B) nunca vai a 1.

Para gerar os diversos padrões de pisca-pisca foi usado um vetor de 10 posições, um para cada led.

#define LED\_PISCA 20 🡪 20 contagens, logo 20 \* 20 ms = 0,4 seg, freq. atualização dos leds.

volatile word cont\_leds; 🡪 contar intervalo LED\_PER para mudar estado leds (0,4 seg).

volatile byte led\_az\_vet[10]; 🡪 Vetor para indicar estado led azul.

volatile byte led\_vm\_vet[10]; 🡪 Vetor para indicar estado led vermelho.

volatile byte led\_az\_ix; 🡪 Indexador para vetor led azul.

volatile byte led\_vm\_ix; 🡪 Indexador para vetor led vermelho.

O contador cont\_leds é incrementado a cada interrupção do Timer 1, ou seja, a cada 20 ms. Quando este contador chega o limite indicado por LED\_PISCA, os indexadores led\_az\_ix e led\_vm\_ix são incrementados e passam a indicar o novo estado dos leds. Eles contam 0, 1, ..., 9, 0, 1, ...

Para criar um padrão de pisca-pisca, basta preencher os vetores com 0 e 1. Note que o “1” implica em 20ms aceso e 20 ms apagado (50 Hz, o olho não percebe)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Indexador | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| led\_AZ() ou led\_VM() | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| led\_az() ou led\_vm() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Intervalo pisca | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Indexador | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Pisca rápido | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| Pisca lento | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| Pisca muito lento | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Intervalo pisca | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s |

Padrões com os dois leds

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PISCA\_0 | Indexador | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Azul | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Vermelho | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| Intervalo pisca | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s |
| Intervalo total | 4 segundos | | | | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PISCA\_1 | Indexador | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Azul | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| Vermelho | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Intervalo pisca | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s |
| Intervalo total | 4 segundos | | | | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PISCA\_2 | Indexador | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Azul | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| Vermelho | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Intervalo pisca | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s | 0,4s |
| Intervalo total | 4 segundos | | | | | | | | | |

**GPIO**

02/12/2020 - Funções para GPIO

Pinos usados como GPIO:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pino Nano | Nome | GPIO | Função | Sentido | Valor inicial |
|  | CE-WQ | PB0 | CE da Flash W25Q64 | OUT | 1 |
|  | Led1 | PB1 | (OCR1A) Led Azul | PWM | 0% |
|  | Led2 | PB2 | (OCR1B) Led Vermelho | PWM | 0% |
|  | CE-NRF | PC2 | CE do Rádio NRF24L01 | OUT | 1 |
|  | CSN | PC3 | CSN do Rádio NRF24L01 | OUT | 1 |
|  | VIB | PD3 | Ligar vibrador | OUT | 0 |
|  | SW | PD4 | Entrada da chave | IN | - |
|  | MT1 | PD5 | Ligar motor | OUT | 1 |
|  | MT0 | PD6 | Ligar motor | OUT | 1 |
|  | G2 | PD7 | Ligar energia da placa (Gate de Q2) | OUT | 1 |

#define ABERTA 1

#define FECHADA 0

Funções

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| void | gpio\_scp1\_config | (void) |
| void | gpio\_SCP1 | (void) |
| void | gpio\_scp1 | (void) |
| void | gpio\_Scp1 | (void) |
| void | gpio\_scp2\_config | (void) |
| void | gpio\_SCP2 | (void) |
| void | gpio\_scp2 | (void) |
| void | gpio\_Scp2 | (void) |
|  |  |  |
|  |  |  |
| void | gpio\_config | (void) |
| void | kxc\_off | (void) |
|  |  |  |
| void | leds\_pisca | (byte pisca) |
| void | led\_AZ | (void) |
| void | led\_az | (void) |
| void | led\_Az | (void) |
| void | led\_VM | (void) |
| void | led\_vm | (void) |
| void | led\_Vm | (void) |
|  |  |  |
| byte | sw\_mon | (void) |
| byte | sw\_estado | (void) |
|  |  |  |
| void | motor\_desce | (void) |
| void | motor\_sobe | (void) |
| void | motor\_PARA | (void) |
| void | motor\_para | (void) |
|  |  |  |
| void | adc\_config | (void) |
| char | adc\_hall\_1 | (char \*dt) |
| char | adc\_hall\_2 | (char \*dt) |
|  |  |  |
| void | vib\_on | (void) |
| void | vib\_off | (void) |

* void **gpio\_scp1\_config** (void)
* void **gpio\_scp1** (void) scp1=0
* void **gpio\_SCP1** (void) scp1=1
* void **gpio\_Scp1** (void) scp1 invertido

Usa C2=SW=D4(PD4) 🡪 chave SW para de funcionar. É lento por causa do capacitor (transição = 10 useg).

* void **gpio\_scp2\_config** (void)
* void **gpio\_scp2** (void) scp1=0
* void **gpio\_SCP2** (void) scp1=1
* void **gpio\_Scp2** (void) scp1 invertido

Usa RC=CE-WQ=D8(PB0) 🡪 Memo Flash para de funcionar. É rápido (transição = 3 useg).

* void **gpio\_config** (void)

Configurar todos o GPIO e coloca as saídas nos valores como os definidos na tabela no início deste tópico.

* void **kxc\_off** (void)

Desligar o Cachacinha. Faz G2=0. Corta a energia pelo transistor Q2.

* void **leds\_pisca** (void)

Programa um padrão de pisca-pisca para os leds. Ver no tópico do timer.

Temos PISCA\_1, PISCA\_2, etc.

* void **led\_AZ** (void)

Acende o led Azul. Programa o vetor led\_az\_vet[i]=1;

* void **led\_az** (void)

Apaga o led Azul. Programa o vetor led\_az\_vet[i]=0;

* void **led\_Az** (void)

Inverte o estado do led Azul. Programa o vetor led\_az\_vet[i]=^1;

* void **led\_VM** (void)

Acende o led Vermelho. Programa o vetor led\_vm\_vet[i]=1;

* void **led\_vm** (void)

Apaga o led Vermelho. Programa o vetor led\_vm\_vet[i]=0;

* void **led\_Vm** (void)

Inverte o estado do led Vermelho. Programa o vetor led\_vm\_vet[i]=^1;

* byte **sw\_mon** (void)

Monitorar acionamento de SW. Retorna

TRUE 🡪 Passou de Aberta para Fechada.

FALSE 🡪 Demais casos.

* byte **sw\_estado** (void)

Retorna estado atual da chave. Retorna ABERTA ou FECHADA

* void **motor\_desce**(void)

Ligar motor de forma que mesa se aproxime do corpo do motor. Faz MT0=1 e MT1=0

* void **motor\_sobe** (void)

Ligar motor de forma que mesa se afaste do corpo do motor. Faz MT0=0 e MT1=1.

* void **motor\_PARA** (void)

Parar o motor. Faz MT1=MT0=1.

* void **motor\_para** (void)

Parar o motor. Faz MT1=MT0=0.

* void **adc\_config** (void)

Configurar o ADC para ler sensores Hall. Alinhado pela esquerda, basta ler ADCH.

* char **adc\_hall\_1** (char \*dt)

Sensor Hall mais próximo do corpo do motor.

Sensor Hall vazio retorna 0x80. Uma variação de 0x17 acima ou abaixo desse valor é a indicação de que o sensor Hall foi acionado. Em \*dt é entregue o valor lido.

Se diferença em valor absoluto entre leitura atual e 0x80 for maior que 0x17, retorna TRUE;

Do contrário, retorna FALSE.

* char **adc\_hall\_2** (char \*dt)

Sensor Hall mais afastado do corpo do motor.

Sensor Hall vazio retorna 0x80. Uma variação de 0x17 acima ou abaixo desse valor é a indicação de que o sensor Hall foi acionado. Em \*dt é entregue o valor lido.

Se diferença em valor absoluto entre leitura atual e 0x80 for maior que 0x17, retorna TRUE;

Do contrário, retorna FALSE.

* void **vib\_on** (void)

Ligar vibrador. Faz VIB=1.

* void **vib\_off** (void)

Desligar vibrador. Faz VIB=0.

ADC: modo 8 bits, CLK/2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **ADMUX** | REFS1 | REFS0 | ADLAR | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0/1 |
| **ADCSRA** | ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **ADCSRB** | - | ACME | - | - | MUX5 | ADST2 | ADST1 | ADST0 |
| - | 0 | - | - | 0 | 0 | 0 | 0 |

HALL

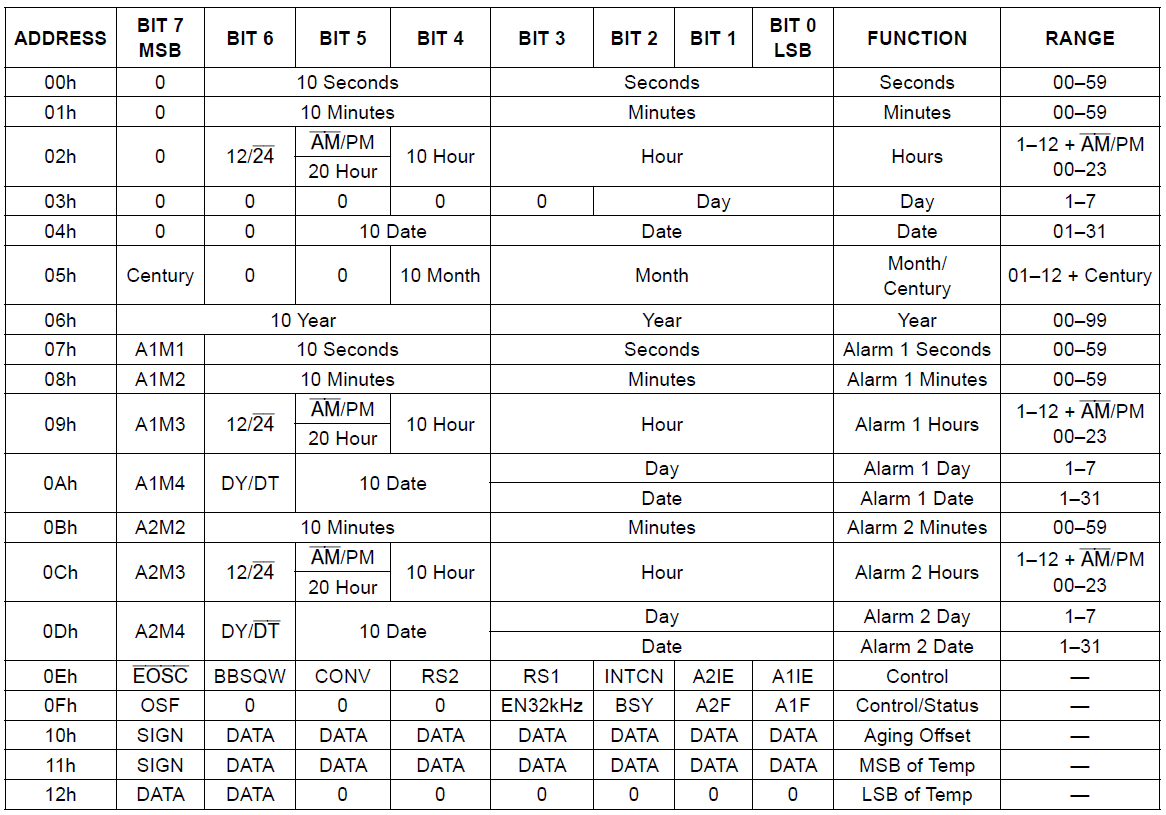
REFS1,0 = 01 🡪 Ref = AVcc ADLAR = 1 🡪 8 bits (ADCH) MUX3,2,1,0 = 011x 🡪 Hall1/Hall2 ADPS2,1,0 = 000 🡪 ADCclk = CLK/2 ADLAR = 1 🡪 8 bits, usar ADCH

BATERIA (VB)

REFS1,0 = 10 🡪 Ref = 1,1 V ADLAR = 1 🡪 8 bits (ADCH) MUX3,2,1,0 = 011x 🡪 Hall1/Hall2 ADPS2,1,0 = 000 🡪 ADCclk = CLK/2 ADLAR = 1 🡪 8 bits, usar ADCH

**RTC**

02/12/2020 - Funções para acessar o RTC DS3231



Funções

|  |  |  |
| --- | --- | --- |
|  |  |  |
| byte | rtc\_temp | (byte \*tp) |
|  |  |  |
| byte | rtc\_kxc\_on\_hora | (byte minu) |
| byte | rtc\_kxc\_on\_minuto | (byte seg) |
| byte | rtc\_alm\_hora | (byte qual, byte minu) |
| byte | rtc\_alm\_min | (byte seg) |
| void | rtc\_alm\_masc | (void) |
| void | rtc\_ALM | (byte qual) |
| void | rtc\_alm | (byte qual) |
| void | rtc\_alm\_flag | (byte qual) |
| void | rtc\_INTCN | (void) |
| void | rtc\_intcn | (void) |
| byte | rtc\_wr\_data\_hora | (void) |
|  |  |  |
| void | rtc\_wr | (byte reg, byte dado) |
| byte | rtc\_rd | (byte reg) |
| void | rtc\_wr\_blk | (byte reg, byte \*vt, byte qtd) |
| void | rtc\_rd\_blk | (byte reg, byte \*vt, byte qtd) |
|  |  |  |
|  |  |  |
|  |  |  |

* byte **rtc\_temp** (byte \*tp)

Dispara uma conversão de temperatura.

Retorna FALSE de aconteceu timeout na espera do BUSY.

Valor em tp está em binário, complemento a 2.

* byte **rtc\_kxc\_on\_minuto** (byte seg)

Faz todos os preparativos para ligar o KXC a cada minuto.

Quanto contador de segundos do RTC fica igual a seg (em decimal) o KXC é ligado.

* byte **rtc\_alm\_hora** (byte qual, byte minu)

Programa alarme para acontecer a cada hora. Alarma quando contador de minutos fica igual a minu.

O argumento qual indica qual alarme (1 ou 2).

Retorna TRUE se tudo certo, FALSE se minu>59.

* byte **rtc\_alm\_min** (byte seg)

Programa alarme para acontecer a cada minuto. Alarma quando contador de segundos fica igual a seg.

Só o alarme 1 tem este recurso.

Retorna TRUE se tudo certo, FALSE se seg>59.

* void **rtc\_alm\_masc** (void)

Mascara todos os alarmes do RTC.

Alarme 1: A1M1=A1M2=A1M3=A1M4=1 e Alarme 2: A2M2=A2M3=A2M4=1

* void **rtc\_ALM** (byte qual)

Habilitar um dos dois alarmes. O argumento qual indica qual alarme (1 ou 2). A1IE=1 ou A2IE=1;

* void **rtc\_alm** (byte qual)

Desabilitar um dos dois alarmes. O argumento qual indica qual alarme (1 ou 2). A1IE1=0 ou A2IE=0;

* void **rtc\_alm\_flag** (byte qual)

Apagar a flag de alarme. O argumento qual indica qual alarme (1 ou 2). A1F ou A2F=1;

* void **rtc\_INTCN** (void)

Habilitar um modo de interrupção do DS3231. Faz pino INT/SQW = 0 (G1 de Q1) quando o alarme acontece.

É preciso para que o DS3231 ligue o KXC. Faz INTCN = 1.

* void **rtc\_intcn** (void)

Desabilitar um modo de interrupção do DS3231. Faz pino INT/SQW = onda quadrada.

Faz INTCN = 0 e neste caso DS3231 não liga o KXC.

Onda quadrada irá ligar o KXC cada vez que for para nível baixo. É bom ,não usar.

* byte **rtc\_wr\_data\_hora** (void)

Acerta a data ou a hora. Retira data ou hora da fila SERI e escreve no RTC.

É preciso ter a certeza de que o próximo elemento da fila é um número

dd/mm/aa --> formato da data

hh:mm:ss --> formato da hora

A distinção entre data e hora é feita pelo “/” e pelo “:”.

Retorna TRUE se conseguiu acertar, FALSE caso contrário.

* void **rtc\_wr** (byte reg, byte dado)

Escreve o dado no registrador de endereço “reg”.

* byte **rtc\_rd** (byte reg)

Retorna o valor lido no registrador de endereço “reg”.

* void **rtc\_wr\_blk** (byte reg, byte \*vt, byte qtd)

Escreve um bloco de dados a partir do endereço “reg”. Os dados estão no vetor “vt” e “qtd” indica a quantidade a ser escrita.

* void **rtc\_rd\_blk** (byte reg, byte \*vt, byte qtd)

Ler um bloco de dados a partir do endereço “reg”. Os dados lidos são escritos no vetor “vt” e “qtd” indica a quantidade a ser lida.

**MPU 6050**

08/12/2020 - Funções para acessa MPU que está no endereço 0x69 (pino A1 = VCC)

Funções

|  |  |  |  |
| --- | --- | --- | --- |
| Uso |  |  |  |
|  |  |  |  |
|  | byte | mpu\_pos | (int \*acel) |
|  | word | mpu\_ang\_dec | (int \*acel) |
|  | byte | mpu\_ang | (int \*acel) |
|  | byte | mpu\_ang\_y | (int \*acel, int \*vert) |
|  |  |  |  |
|  | byte | mpu\_estavel | (byte vz) |
|  | byte | mpu\_agito | (void) |
|  | int | mpu\_rd\_esc\_acel | (void) |
|  | int | mpu\_rd\_esc\_giro | (void) |
|  |  |  |  |
| S | void | mpu\_media | (int \*vetor, int qtd) |
|  | void | mpu\_acel\_media | (int \*vet, int qtd) |
|  | void | mpu\_rd\_acel | (int \*vt) |
|  | void | mpu\_giro\_media | (int \*vet, int qtd) |
|  | void | mpu\_rd\_giro | (int \*vt) |
|  | void | mpu\_rd\_temp | (int \*vt) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | void | mpu\_rd\_ac\_tp\_gi | (int \*vt) |
|  | void | mpu\_config | (void) |
|  | void | mpu\_esc\_acel | (byte acel) |
|  | void | mpu\_esc\_giro | (byte giro) |
|  | void | mpu\_who\_am\_i | (void) |
|  | void | mpu\_acorda | (void) |
|  | void | mpu\_dorme | (void) |
|  |  |  |  |

mpu\_rd\_ac\_tp\_gi() gasta um pouco mais de 20 ms, com twi\_55k().

* byte **mpu\_pos** (int \*acel)

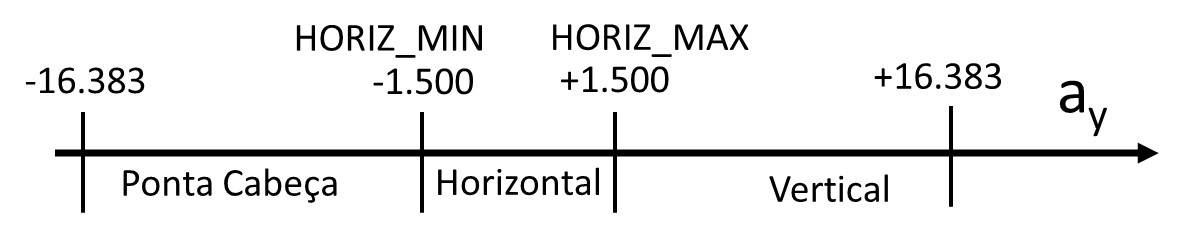
Retorna posição do KXC. São 3 posições possíveis:

VERT\_OK 🡪 vertical, posição de operação;

HORIZ 🡪 horizontal, deitado sobre uma superfície

VERT\_INV 🡪 Ponta cabeça.

Usa a seguinte faixa para determinar a posição



* void **mpu\_ang\_dec** (int \*acel)

Calcular o ângulo de inclinação do KXC com precisão de décimos de graus. Usa os 3 eixos.

Lê posição vertical diretamente na EEPROM.

Temos leituras ax, ay e az e a referência vertical vx, vy e vz. Usa a equação abaixo.

* void **mpu\_ang** (int \*acel)

Calcular o ângulo de inclinação do KXC. Usa os 3 eixos.

Lê posição vertical diretamente na EEPROM.

Temos leituras ax, ay e az e a referência vertical vx, vy e vz. Usa a equação abaixo.

* void **mpu\_ang\_y** (int \*acel, int \*vert)

Calcular o ângulo de inclinação do KXC. Usa apenas o eixo ay.

Temos leituras ax, ay e az e a referência vertical vx, vy e vz. Usa a equação abaixo.

**TWI (I2C)**

08/12/2020 - Funções para acessar TWI

|  |  |  |
| --- | --- | --- |
|  |  |  |
| void | twi\_25k | (void) |
| void | twi\_6k | (void) |
| void | twi\_55k | (void) |
|  |  |  |
| void | twi\_start | (byte ix) |
| void | twi\_start\_rep | (int ix) |
| void | twi\_stop | (void) |
| byte | twi\_er\_check | (byte ee, byte ix) |
| void | twi\_er | (byte eer, byte ix) |
| void | twi\_et | (byte eet, byte ix) |
| void | twi\_dado\_er | (byte dado, byte ix) |
| byte | twi\_dado\_et\_ack | (byte ix) |
| byte | twi\_dado\_et\_nack | (byte ix) |
| byte | espera\_twint | (byte ix) |
|  |  |  |
|  |  |  |

* void **twi\_25k** (void)

Com relógio de 1 MHz, resulta em 25 kHz. Vira 400 kHz com relógio de 16 MHz.

* void **twi\_6k** (void)

Com relógio de 1 MHz, resulta em 6,25 kHz. Vira 100 kHz com relógio de 16 MHz.

* void **twi\_55k** (void)

Essa é maior velocidade com relógio de 1 MHz.

* void **twi\_start** (byte ix)

Gerar um START no TWI.

* void **twi\_start\_rep** (int ix)

Gerar um START Repetido no TWI.

* void **twi\_stop** (void)

Gerar um STOP.

* byte **twi\_er\_check** (byte ee, byte ix)

Enviar endereço de Escrita do Escravo Receptor (ER) e esperar ACK.

Retorna TRUE se escravo gerou ACK.

Retorna FALSE se gerou NACK (não gera msg de erro).

ee = endereço do escravo.

* void **twi\_er** (byte eer, byte ix)

Enviar endereço de Escrita do Escravo (ER) e esperar ACK.

eer = endereço do escravo receptor.

* void **twi\_et** (byte eet, byte ix)

Enviar endereço de Leitura do Escravo (ET) e esperar ACK.

eet = endereço do escravo transmissor.

* void **twi\_dado\_er** (byte dado, byte ix)

Enviar dado para escravo previamente endereçado.

* byte **twi\_dado\_et\_ack** (byte ix)

Receber dado e gerar ACK.

* byte **twi\_dado\_et\_nack** (byte ix)

Receber dado e gerar NACK.

* byte **espera\_twint** (byte ix)

Esperar TWINT.

* void **twi\_erro** (int cod, int ix)

ERROS no TWI (I2C).

1 = Erro ao gerar START.

2 = Erro ao gerar START Repetido.

3 = Erro Escravo Receptor endereçado (ER) não enviou ACK.

4 = Erro Escravo Transmissor endereçado (ET) não enviou ACK.

5 = Erro Escravo Receptor (ER) não enviou ACK após envio do dado.

6 = Erro ao receber um dado do Escravo Transmissor (ET) e gerar um ACK.

7 = Erro ao receber um dado do Escravo Transmissor (ET) e gerar um NACK.

8 = Erro ao esperar TWINT - Timeout esperando TWINT ir para 1.

**nRF24L01+**

Pinos de controle: CE-NRF 🡪 PC2 (A2) CSN 🡪 PC3 (A3)

|  |  |  |
| --- | --- | --- |
|  |  |  |
| byte | nrf\_mrt | (byte modo) |
|  |  |  |
| void | nrf\_ser\_buf | (void) |
|  |  |  |
| byte | nrf\_tx\_prep | (char \* nome) |
| byte | nrf\_rx\_prep | (void) |
|  |  |  |
| byte | nrf\_canal | (byte canal) |
| byte | nrf\_pot | (byte pot) |
| byte | nrf\_vel | (byte vel) |
|  |  |  |
| byte | nrf\_rx\_int\_hab | (void) |
| byte | nrf\_rx\_int\_desab | (void) |
| byte | nrf\_tx\_int\_hab | (void) |
| byte | nrf\_tx\_int\_desab | (void) |
| byte | nrf\_mrt\_int\_hab | (void) |
| byte | nrf\_mrt\_int\_desab | (void) |
| void | nrf\_config | (void) |
| void | nrf\_dpl\_hab | (void) |
| void | nrf\_ser\_status | (void) |
| void | nrf\_ser\_regs | (void) |
| void | nrf\_ser\_addr | (void) |
|  |  |  |
| byte | nrf\_power\_up | (void) |
| byte | nrf\_power\_down | (void) |
| byte | nrf\_modo\_tx | (void) |
| byte | nrf\_modo\_rx | (void) |
|  |  |  |
|  |  |  |
| byte | nrf\_rx\_dado | (byte \*pipe) |
| void | nrf\_rx\_hab\_pipe | (byte pipe) |
|  |  |  |
| void | nrf\_flush\_tx | (void) |
| void | nrf\_flush\_rx | (void) |
|  |  |  |
| byte | nrf\_wr\_addr | (byte pipe, byte \*vet) |
| byte | nrf\_rd\_addr | (byte pipe, byte \*vet) |
| byte | nrf\_wr\_canal | (byte canal) |
| byte | nrf\_rd\_canal | (void) |
|  |  |  |
| byte | nrf\_wr\_dpl | (byte \*vet, byte qtd) |
| byte | nrf\_wr\_ack\_dpl | (byte \*vet, byte pipe, byte qtd) |
| byte | nrf\_wr\_payload | (byte \*vet, byte qtd) |
| byte | nrf\_rd\_payload | (byte \*vet, byte qtd) |
|  |  |  |
| byte | nrf\_rd\_reg\_mb | (byte reg, byte \*vt, byte qtd) |
| byte | nrf\_wr\_reg\_mb | (byte reg, byte \*vt, byte qtd) |
| byte | nrf\_wr\_reg | (byte reg, byte dado) |
| byte | nrf\_rd\_dpl\_qrd | (void) |
| byte | nrf\_rd\_reg\_blk | (byte reg, byte \*vt, byte qtd) |
| byte | nrf\_rd\_status | (void) |
| byte | nrf\_rd\_reg | (byte reg) |
|  |  |  |
| void | nrf\_ce | (void) |
| void | nrf\_CE | (void) |
| void | nrf\_csn | (void) |
| void | nrf\_CSN | (void) |
|  |  |  |
| byte | spi\_transf | (byte dado) |
| byte | spi\_config | (byte clk) |
|  |  |  |
|  |  |  |
|  |  |  |

* byte **nrf\_mtr** (byte modo)

Rotina para tratar o MRT. Ainda tenta reenviar NRF\_MRT\_MAX vezes, se não conseguir, faz o flush() e retorna.

Tem duas opções: modo == QUIETO ==> nada imprime

modo == NÃO\_QUIETO ==> imprime R (a cada tentativa de retransmissão) ou F (flush)

Retorna TRUE se conseguiu retransmitir com sucesso.

Retorna FALSE se fez o flush() e com nrf\_tx\_flag=TRUE .

* void **nrf\_ser\_buf** (void)

Imprimir o buffer do rádio. Para ajudar no debug. Usa ser\_char\_velho().

* byte **nrf\_tx\_prep** (char \*nome)

Faz todas as preparações para o rádio (FFK0) funcionar como transmissor para o rádio “nome”.

Acerta nome, dynamic payload, etc.

* byte **nrf\_rx\_prep** (void)

Faz todas as preparações para o rádio funcionar como receptor, sendo o FFK0 o transmissor.

Acerta nome, dynamic payload, etc.

* byte **nrf\_canal** (byte canal)

Especificar o canal a ser usado. Canal = 0, 1, 2, ..., 127. Retorna o Status.

* byte **nrf\_pot** (byte pot)

Especificar potência do rádio, pot = 1, 2, 3, ou 4. Decrementa (pot=0,1,2,3) e depois separa os 2 bits menos significativos. Retorna o Status.

* byte **nrf\_vel** (byte vel)

Especificar velocidade do rádio, vel = 1, 2, ou 3. Decrementa (vel=0,1,2) e depois separa os 2 bits menos significativos. Retorna o Status

* void **nrf\_rx\_int\_hab** (void)

Habilitar a interrupção por dado recebido. Faz CONFIG.MASKRX\_DR=0 e EIMSK.INT0=1 (ATmega).

Assim, interrompe (INT0) se STATUS.RX\_DR=1. Retorna STATUS.

* void **nrf\_rx\_int\_desab** (void)

Desabilitar a interrupção por dado recebido. Faz CONFIG.MASK\_RX\_DR=1. Assim, não interrompe (INT0) se STATUS.RX\_DR=1. Retorna STATUS.

* void **nrf\_tx\_int\_hab** (void)

Habilitar a interrupção por dado transmitido. Faz CONFIG.MASK\_TX\_DS=0 e EIMSK.INT0=1 (ATmega).

Assim, interrompe (INT0) se STATUS.TX\_DS=1. Retorna STATUS.

* void **nrf\_rx\_int\_desab** (void)

Desabilitar a interrupção por dado transmitido. Faz CONFIG.MASK\_TX\_DS=1. Assim, não interrompe (INT0) se STATUS.TX\_DS=1. Retorna STATUS.

* void **nrf\_mrt\_int\_hab** (void)

Habilitar a interrupção por dado transmitido. Faz CONFIG.MASK\_MAX\_RT=0. Assim, interrompe (INT0) se STATUS.MAX\_RT=1. Retorna STATUS.

* void **nrf\_mrt\_int\_desab** (void)

Desabilitar a interrupção por atingir o máximo de retransmissões. Faz CONFIG.MASK\_MAX\_RT=1 e EIMSK.INT0=1 (ATmega).

Assim, não interrompe (INT0) se STATUS.MAX\_RT=1. Retorna STATUS.

* void **nrf\_config** (void)

Configuração inicial do NRF24L01. Canal 0x4C. Configura até 15 retransmissões e 1,5 mseg entre cada uma. Habilita auto ack em todas as pipes.

Endereço com 5 bytes e todas as pipes com 32 bytes de payload. Habilita CRC de 2 bytes.

|  |  |  |
| --- | --- | --- |
| **04**  **SETUP\_RETR** | ARD | ARC |
| 5 | 15 |

ARD = 5 🡪 1.500 µseg entre as retransmissões ARC = 15 🡪 15 retransmissões

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **06**  **RF\_SETUP** | CONT\_WAVE | 0 | RF\_DR\_LOW | PLL\_LOCK | RF\_DR\_HIGH | RF\_PWR | X |
| 0 | 0 | 0 | 0 | 0 | 0 | X |

CONT\_WAVE=0 🡪 Carrier contínuo RF\_DR\_LOW |RF\_DR\_HIGH=00 🡪 1Mbps RF\_PWR=0 🡪 -18dBm

|  |  |  |
| --- | --- | --- |
| **05**  **RF\_CH** | - | RF\_CH |
| 0 | 0x4C |

RF\_CH = 0x4C 🡪 Escolha do canal

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **01**  **EN\_AA** | 0 | 0 | ENAA\_P5 | ENAA\_P4 | ENAA\_P3 | ENAA\_P2 | ENAA\_P1 | ENAA\_P0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

EN\_AA = 0x3F 🡪 Habilita autorresposta em todos os pipes

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **03**  **SETUP\_AW** | 0 | 0 | 0 | 0 | 0 | 0 | AW |
| 0 | 0 | 0 | 0 | 0 | 0 | 3 |

SETUP\_AW = 3 🡪 endereço de 5 bytes

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **00**  **CONFIG** | 0 | MASK\_RX\_DR | MASK\_TX\_DS | MASK\_MAX\_RT | EN\_CRC | CRCO | PWR\_UP | PRIM\_RX |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

MASK\_RX\_DR=MASK\_TX\_DS=MASK\_MAX\_RT=0 🡪 Todas interrupções habilitadas

EN\_CRC=1🡪Hab CRC CRCO=1🡪CRC de 2bytes PWR\_UP=0 🡪 Power Down PRIM\_RX=0 🡪 Modo TX

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **07**  **STATUS** | 0 | RX\_DR | TX\_DS | MAX\_RT | RX\_P\_NO | TX\_FULL |
| 0 | 1 | 1 | 1 | 0 | 0 |

RX\_DR=1 🡪 apagar Data Ready TX\_DS=1 🡪 apagar Data Sent MAX\_RT=1 🡪 apagar indicador de que atingiu máximo de retransmissões

* void **nrf\_dpl\_hab** (void)

Habilitar o Dynamic Payload nos pipes 0 e 1. No receptor, usar a função comando R\_RX\_PL\_WID para ler a quantidade de dados no payload. Se maior que 32, descartar com comando Flush\_RX.

TX ou RX

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1C**  **DYNPD** | 0 | 0 | DPL\_P5 | DPL\_P4 | DPL\_P3 | DPL\_P2 | DPL\_P1 | DPL\_P0 |
| 0 | 0 | - | - | - | - | **1** | **1** |
| **1D**  **FEATURE** | 0 | 0 | 0 | 0 | 0 | EN\_DPL | EN\_ACK\_PAY | EN\_DYN\_ACK |
| 0 | 0 | 0 | 0 | 0 | **1** | - | - |

* void **nrf\_ser\_regs** (void)

Imprime todos os registradores da NRF. Destaca os bits do registrador de STATUS.

* void **nrf\_ser\_status** (void)

Imprime e interpreta o registrador de status.

* void **nrf\_ser\_regs** (void)

Imprime todos os registradores da NRF. Destaca os bits do registrador de STATUS.

* void **nrf\_ser\_addr** (void)

Imprime todos os endereços das pipes.

* byte **nrf\_power\_up** (void)

CONFIG.PWR\_UP=1 (registrador CONFIG). Faz o power up do NRF

Retorna o status

* byte **nrf\_power\_down** (void)

CONFIG.PWR\_UP=0 (registrador CONFIG). Faz o power down do NRF

Retorna o status

* byte **nrf\_modo\_tx** (void)

CONFIG.PRIM\_RX=0 (registrador CONFIG). Coloca no modo de transmissão.

Retorna o status

* byte **nrf\_modo\_rx** (void)

CONFIG.PWR\_UP=0 (registrador CONFIG). Coloca no modo de recepção.

Retorna o status

* byte **nrf\_rx\_dado** (byte \*pipe)

Verificar se alguma pipe recebeu dado. Em \*pipe é retornado o múmero da pipe que tem dado.

Retorna TRUE ou FALSE de acordo com o caso.

* void **nrf\_rx\_hab\_pipe** (byte pipe)

Habilitar uma pipe para receber dados. Ativa o bit correspondente no registrador.

STATUS: RX\_DR = TX\_DS = MAX\_RT = 0.

CONFIG: PRIM\_RX=1, CRCO=1, EM\_CRC 🡪 modo recepção, com CRC de 2 bytes.

* byte **nrf\_flush\_tx** (void)

Descarta pipe de transmissão. Retorna o STATUS.

* byte **nrf\_flush\_rx** (void)

Descarta pipe de recepção. Retorna o STATUS.

* byte **nrf\_wr\_addr** (byte pipe, byte \*vet)

Programa endereço de uma pipe. Consulta o registrador SETUP\_AW para saber o tamanho do endereço. Retorna o STATUS.

Usa as constantes: NRF\_RX\_ADDR\_P0, NRF\_RX\_ADDR\_P1, ..., NRF\_RX\_ADDR\_P5, NRF\_TX\_ADDR.

* byte **nrf\_rd\_addr** (byte pipe, byte \*vet)

Ler o endereço de uma pipe. Consulta o registrador SETUP\_AW para saber o tamanho do endereço. Retorna o STATUS.

Usa as constantes: NRF\_RX\_ADDR\_P0, NRF\_RX\_ADDR\_P1, ..., NRF\_RX\_ADDR\_P5, NRF\_TX\_ADDR.

* byte **nrf\_wr\_canal** (byte canal)

Programa um novo canal no registrador RF\_CH. Retorna o STATUS.

* byte **nrf\_rd\_canal** (void)

Retorna canal programado no registrador RF\_CH.

* byte **nrf\_wr\_dpl** (byte \*vet, byte qtd)

Para usar com Dynamic Payload. Escreve um certo payload do tamanho indicado (qtd) e faz CE=1 e CE=0 para transmitir.

Retorna condição de STATUS como estava antes de executar a carga.

* byte **nrf\_wr\_ack\_dpl** (byte \*vet, byte pipe, byte qtd)

Carrega o Payload ACK para o pipe indicado. Escreve um certo payload do tamanho indicado (qtd).

Retorna condição de STATUS como estava antes de executar a carga.

* byte **nrf\_wr\_payload** (byte \*vet, byte qtd)

Escreve um certo payload, completa com ZEROS até chegar a 32 bytes e faz CE=1 e CE=0 para transmitir.

Retorna condição de STATUS como estava antes de executar a carga.

* byte **nrf\_rd\_payload** (byte \*vet, byte qtd)

Lê o payload. Retorna condição de STATUS como estava antes de executar a leitura.

* byte **nrf\_rd\_reg\_mb** (byte reg, byte \*vet, byte qtd)

Ler um registrador Multi Byte. Retorna condição de STATUS.

* byte **nrf\_wr\_reg\_mb** (byte reg, byte \*vet, byte qtd)

Escrever num registrador Multi Byte. Retorna condição de STATUS.

* byte **nrf\_wr\_reg** (byte reg, byte dado)

Escrever num registrador. Retorna condição de STATUS.

* byte **nrf\_rd\_dpl\_qtd** (void)

Usado quando se está no modo Dynamic Payload. Usa o comando R\_RX\_PL\_WID para ver quantos bytes chegaram.

Se a quantidade que chegou for maior que 32, é porque deu erro. Então descarta toda a recepção com a função nrf\_flush\_rx().

* byte **nrf\_rd\_reg\_blk** (byte reg, byte \*vet, byte qtd)

Ler uma sequência de registradores, a partir de “reg”. Retorna condição de STATUS.

* byte **nrf\_rd\_status** (void)

Retorna o conteúdo do registrador de STATUS.

* byte **nrf\_rd\_reg** (byte reg)

Retorna o conteúdo de um registrador.

* void **nrf\_ce** (void) 🡪 NRF\_CE = LOW.
* void **nrf\_CE** (void) 🡪 NRF\_CE = HIGH.
* void **nrf\_csn** (void) 🡪 NRF\_CSN = LOW.
* void **nrf\_CSN** (void) 🡪 NRF\_CSN = HIGH.
* byte **spi\_transf** (byte dado)

Envia e recebe um dado pela porta SPI.

* void **spi\_config** (byte clk)

Configura porta SPI. Opções de clk: SPI\_125k, SPI\_250k e SPI\_500k.

Interrupções do Rádio:

MASK\_RX\_DR e RX\_DR 🡪 quando chega dado na FIFO

MASK\_RX\_DS e RX\_DS 🡪 dado enviado e chegou ACK

MASK\_MAX\_RT e MAX\_RT 🡪 ultrapassou máximo de retransmissões

**Registradores - nRF24L01+**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **00**  **CONFIG** | 0 | MASK\_RX\_DR | MASK\_TX\_DS | MASK\_MAX\_RT | EM\_CRC | CRCO | PWR\_UP | PRIM\_RX |
| 0 |  |  |  |  |  |  |  |
| **01**  **EN\_AA** | 0 | 0 | ENAA\_P5 | ENAA\_P4 | ENAA\_P3 | ENAA\_P2 | ENAA\_P1 | ENAA\_P0 |
| 0 | 0 |  |  |  |  |  |  |
| **02**  **EN\_RXADDR** | 0 | 0 | ERX\_P5 | ERX\_P4 | ERX\_P3 | ERX\_P2 | ERX\_P1 | ERX\_P0 |
| 0 | 0 |  |  |  |  |  |  |
| **03**  **SETUP\_AW** | 0 | 0 | 0 | 0 | 0 | 0 | AW | |
| 0 | 0 | 0 | 0 | 0 | 0 |  | |
| **04**  **SETUP\_RETR** | ARD | | | | ARC | | | |
|  | | | |  | | | |
| **05**  **RF\_CH** | - | RF\_CH | | | | | | |
| 0 |  | | | | | | |
| **06**  **RF\_SETUP** | CONT\_WAVE | 0 | RF\_DR\_LOW | PLL\_LOCK | RF\_DR\_HIGH | RF\_PWR | | X |
|  | 0 |  |  | 0 |  | | X |
| **07**  **STATUS** | 0 | RX\_DR | TX\_DS | MAX\_RT | RX\_P\_NO | | | TX\_FULL |
| 0 |  |  |  |  | | |  |
| **08**  **OBSERVE\_TX** | PLOS\_CNT | | | | ARC\_CNT | | | |
|  | | | |  | | | |
| **09**  **RPD** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | RPD |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| **0A**  **RX\_ADDR\_P0** | RX\_ADDR\_P0 | | | | | | | |
| 5 bytes = 40 bits | | | | | | | |
| **0B**  **RX\_ADDR\_P1** | RX\_ADDR\_P1 | | | | | | | |
| 5 bytes = 40 bits | | | | | | | |
| **0C**  **RX\_ADDR\_P2** | RX\_ADDR\_P2 | | | | | | | |
| 1 byte = 8 bits | | | | | | | |
| **0D**  **RX\_ADDR\_P3** | RX\_ADDR\_P3 | | | | | | | |
| 1 byte = 8 bits | | | | | | | |
| **0E**  **RX\_ADDR\_P4** | RX\_ADDR\_P4 | | | | | | | |
| 1 byte = 8 bits | | | | | | | |
| **0F**  **RX\_ADDR\_P5** | RX\_ADDR\_P5 | | | | | | | |
| 1 byte = 8 bits | | | | | | | |
| **10**  **TX\_ADR** | TX\_ADDR | | | | | | | |
| 5 bytes = 40 bits | | | | | | | |
| **11**  **RX\_PW\_P0** | 0 | 0 | RX\_PW\_P0 | | | | | |
| 0 | 0 |  | | | | | |
| **12**  **RX\_PW\_P1** | 0 | 0 | RX\_PW\_P1 | | | | | |
| 0 | 0 |  | | | | | |
| **13**  **RX\_PW\_P2** | 0 | 0 | RX\_PW\_P2 | | | | | |
| 0 | 0 |  | | | | | |
| **14**  **RX\_PW\_P3** | 0 | 0 | RX\_RW\_P3 | | | | | |
| 0 | 0 |  | | | | | |
| **15**  **RX\_PW\_P4** | 0 | 0 | RX\_RW\_P4 | | | | | |
| 0 | 0 |  | | | | | |
| **16**  **RX\_PW\_P5** | 0 | 0 | RX\_RW\_P5 | | | | | |
| 0 | 0 |  | | | | | |
| **17**  **FIFO\_STATUS** | 0 | TX\_REUSE | TX\_FULL | TX\_EMPTY | 0 | 0 | RX\_FULL | RX\_EMPTY |
| 0 | 0 |  |  | 0 | 0 |  |  |
| **1C**  **DYNPD** | 0 | 0 | DPL\_P5 | DPL\_P4 | DPL\_P3 | DPL\_P2 | DPL\_P1 | DPL\_P0 |
| 0 | 0 |  |  |  |  |  |  |
| **1D**  **FEATURE** | 0 | 0 | 0 | 0 | 0 | EM\_DPL | EM\_ACK\_PAY | EM\_DYN\_ACK |
| 0 | 0 | 0 | 0 | 0 |  |  |  |
| **ACK\_PLD** | ACK\_PLD | | | | | | | |
| 32 bytes = 256 bits | | | | | | | |
| **TX\_PLD** | TX\_PLD | | | | | | | |
| 32 bytes = 256 bits | | | | | | | |
| **RX\_PLD** | RX\_PLD | | | | | | | |
| 32 bytes = 256 bits | | | | | | | |

Dúvidas sobre o rádio:

1) Pag 59 e 60, Tabela de registradores, RX\_ADDR\_P0, RX\_ADDR\_P1, TX\_ADDR.

O que signifiva “**LSByte is written first**”? Significa que para escrever “KXC10” deveríamos enviar “01CXK”?

2) Pag 40, fig 13 e pag 41, fig 14

Sugestão: Tentar programas os pipes para os seguintes endereços:

PIPE0 = “KXC10”, PIPE1 = “KXC11”, PIPE2 = “KXC12”, PIPE3 = “KXC13”, PIPE4 = “KXC14”, PIPE5 = “KXC15”,

O transmissor envia duas mensagens pelo PIPE0, dá uma pausa e depois se configura para enviar pelo PIPE1 e assim por diante até o PIPE5. Depois volta a repetir tudo.

Fazer inicialmente para PIPE0 e PIPE1. Se funcionar, adicionar mais PIPES

Talvez seja o caso de se inverter a ordem de envio dos endereços.

3) Na tabela de registradores, ver o registrador ACK\_PLD. É o payload que pode ser enviar junto com a mensagem de ACK. Como será que se consegue usá-lo.

Sugestão: com um programa muito simples tentar enviar para transmissor alguma coisa junto com o ACK. Por exemplo, enviar os números (ASCII) 0, 1, 2 e 3.

O transmissor tenta ler o payload que veio com o ACK, e toma as ações:

0 = apaga os dois leds; 1 = acende led vermelho; 2 = acende led azul e 3 = acende ambos os leds.

Assim, dá para conferir de forma simples se o transmissor está conseguindo receber o payload.

4) Pag 56, fig 30 e na Tabela de registradores, ver o registrador DYNPD:

Será é ele que permite o uso dos PIPEs para construir o payload? Os dois RX\_FIFO e TX\_FIFO consomem os PIPEs ou são formados por memória extra?

**W25Q32**

Pino de controle: CE-WQ 🡪 PB0 (D8)

Ensaio com a memória Flash de 4 MB (00xx xxxx xxxx xxxx xxxx xxxx) (0 🡪 4.194.303).

São 16 K páginas de 256 bytes (00xx xxxx xxxx xxxx 0000 0000) (0 🡪 16.385).

São 1K setores de 4 KB (00xx xxxx xxxx 0000 0000 0000) (0 🡪 1.023).

São 128 setores de 32 KB (00xx xxxx x000 0000 0000 0000) (0 🡪 127).

São 64 setores de 64 KB (00xx xxxx 0000 0000 0000 0000) (0 🡪 63).

Ao entrar neste modo deve surgir as informações abaixo.

JEDEC: EF 40 16 SR1=10 SR2=00

Manuf ID: EF Dev ID: 15

Unique ID: D6 65 84 98 46 21 71 15

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| byte | wq\_dir\_cria | (byte cmdo, char \*nome) |
| byte | wq\_dir\_prox | (void) |
|  |  |  |
| byte | wq\_ser\_tam | (byte id) |
| void | wq\_ser\_id | (byte id) |
| word | wq\_rd\_word | (long adr) |
| byte | wq\_rd\_byte | (long adr) |
| void | wq\_wr\_word\_bcd | (long adr, word dado) |
| void | wq\_wr\_word | (long adr, word dado) |
| void | wq\_wr\_byte | (long adr, byte dado) |
| void | wq\_wr\_str | (long adr, byte \*vet) |
| void | wq\_rd\_str | (long adr, byte \*vet, word qtd) |
|  |  |  |
| void | wq\_wr\_blk | (long adr, byte \*vet, word qtd) |
| void | wq\_wr\_pag | (word pag, byte \*vet) |
| void | wq\_rd\_blk | (long adr, byte \*vet, long qtd) |
| void | wq\_rd\_pag | (word pag, byte \*vet) |
|  |  |  |
| void | wq\_erase\_chip | (void) |
| void | wq\_erase\_64k | (byte pag) |
| void | wq\_erase\_4k | (word sec) |
|  |  |  |
| byte | wq\_ocupado | (void) |
| void | wq\_wel | (void) |
| void | wq\_WEL | (void) |
| byte | wq\_rd\_sr1 | (void) |
| byte | wq\_rd\_sr2 | (void) |
|  |  |  |
| void | wq\_manuf\_dev\_id | (byte \*vet) |
| void | wq\_jedec\_id | (byte \*vet) |
| void | wq\_unique\_id | (byte \*vet) |
|  |  |  |
| void | wq\_ce | (void) |
| void | wq\_CE | (void) |
|  |  |  |

* byte **wq\_dir\_cria** (byte cmdo, char \*nome)

Criar uma posição de fermentação na FLASH.

Busca pela primeira página de 64K vazia e, por segurança, apaga essa página de 64K.

Escreve os Registradores 0, 1, 2, 3 e faz cópia de toda a EEPROM para a última página de 1 KB.

Retorna o número da pag de 64KB usada.

Retorna ZERO se a FLASH estiver cheia, ou seja, se não criou o diretório.

Sempre prepara para ‘D’ (densidade) ‘H’ (a cada hora).

* byte **wq\_dir\_prox** (void)

Busca por uma página de 64KB que tenha os primeiros 16 bytes iguais a 0xFF.

Isto caracteriza que a página não está sendo usada.

Retorna o número da pag de 64 KB.

* byte **wq\_ser\_tam** (byte id)

Retorna o tamanho da Flash em MegaBytes (MB).

Id = 0x15 🡪 W25Q32 🡪 4 MB

Id = 0x16 🡪 W25Q64 🡪 8 MB (? Confirmar)

Id = 0x17 🡪 W25Q128 🡪 16 MB

* byte **wq\_ser\_id** (byte id)

Imprime o nome da memória Flash

Id = 0x15 🡪 **W25Q32** 🡪 4 MB

Id = 0x16 🡪 **W25Q64** 🡪 8 MB (? Confirmar)

Id = 0x17 🡪 **W25Q128** 🡪 16 MB

* word **wq\_rd\_word** (long adr)

Ler uma word (16 bits) da FLASH. Big Endian.

* byte **wq\_rd\_byte** (long adr)

Ler um byte da FLASH.

* void **wq\_wr\_word\_bcd** (long adr, word dado)

Escrever uma word em BCD na Flash. O “dado” é convertido para BCD e depois escrito.

Big Endian.

* void **wq\_wr\_word** (long adr, word dado)

Escrever uma word na Flash. Big Endian.

* void **wq\_wr\_byte** (long adr, byte dado)

Escrever um byte na Flash.

* void **wq\_wr\_str** (long adr, byte \*vet)

Escrever na Flash uma string terminada em ZERO (‘\0’).

O ZERO também é escrito na Flash.

* void **wq\_rd\_str** (long adr, byte \*vet, word qtd)

Ler da Flash uma string terminada em ZERO (‘\0’). O ZERO também é lido.

O valor “qtd” indica o tamanho máximo da string, incluindo o zero.

Em outras palavras “qtd” é o tamanho de “vet”.

* void **wq\_wr\_blk** (long adr, byte \*vet, long qtd)

Escrever um bloco de dados na memória. Deve ficar dentro de uma página de 256 bytes.

Ativa WEL para poder escrever e ao final o desativa.

* void **wq\_wr\_pag** (word pag, byte \*vet)

Escrever uma página completa (256 bytes) da memória, ou seja, em \*vet devem estar 256 bytes disponíveis.

O endereço é dado por pag\*256.

Ativa WEL para poder escrever e ao final o desativa.

* void **wq\_rd\_blk** (long adr, byte \*vet, long qtd)

Ler um bloco de dados na memória. Não há limite para a quantidade de dados a ser lida.

* void **wq\_rd\_pag** (word pag, byte \*vet)

Ler uma página completa (256 bytes) da memória, ou seja, em \*vet deve ter 256 bytes disponíveis.

O endereço é dado por pag\*256.

* void **wq\_erase\_chip** (void)

Apagar toda a memória Flash. Demora!

* void **wq\_erase\_64k** (byte pag)

Apaga uma página de 64 KB.

Ativa WEL para poder apagar e a memória desativa automaticamente quando termina a operação.

* void **wq\_erase\_4k** (word sec)

Apaga (0xFF) um setor de 4 KB. O endereço de início é sec \* 4.096.

Ativa WEL para poder apagar e a memória desativa automaticamente quando termina a operação.

* void **wq\_ocupado** (void)

Verifica o bit de ocupado. É o bit 0 (BUSY) do registrador 1.

TRUE 🡪 ocupado FALSE 🡪 livre

* void **wq\_wel** (void)

Zera o bit 2 (WEL) do registrador 1.

WEL = 0 🡪 escritas estão bloqueadas.

* void **wq\_WEL** (void)

Ativa o bit 2 (WEL) do registrador 1.

WEL = 1 🡪 escritas estão liberadas.

* byte **wq\_rd\_sr1** (void)

Ler o registrador de status 1.

* byte **wq\_rd\_sr2** (void)

Ler o registrador de status 2.

* void **wq\_manuf\_dev\_id** (byte \*vet)

Ler o Manufacturer e o Device ID.

Vet[0] = Manufacturer ID (0xEF).

Vet[1] = Device ID.

* void **wq\_jedec\_id** (byte \*vet)

Ler o JEDEC para identificar a memória.

Vet[0] = Manufacturer ID (0xEF).

Vet[1] = Memory type (bits 15, …, 8)

Vet[2] = Capacity (bits 7, …, 0)

* void **wq\_unique\_id** (byte \*vet)

Ler os 8 bytes do Unique ID da memória. O vetor vet precisa ter espaço para 8 bytes.

* void **wq\_ce** (void)

Faz CE=LOW, ou seja, permite o acesso à memória. Marca o início de um acesso por SPI.

(noInterrupts()) Desabilita as interrupções para evitar conflito com o Rádio.

* void **wq\_CE** (void)

Faz CE=HIGH, ou seja, inibe o acesso à memória. Marca o fim de um acesso por SPI.

(interrupts()) Habilita as interrupções porque já terminou o acesso à Flash.

**DS18B20 + One Wire**

29/12/2020

Funções para acessar o termômetro DS18B20.

Funções para operação com protocolo ONE WIRE

Usa Timer 2 em 1 MHz para construir as funções básicas

Funções

|  |  |  |
| --- | --- | --- |
|  |  |  |
| word | ds\_temp\_3 | (void) |
| word | ds\_temp | (void) |
|  |  |  |
| void | espera\_100ms | (word dur) |
| void | espera\_100ms | (word dur) |
|  | ISR | (TIMER1\_OVF\_vect) |
| void | t1\_config | (void) |
| void | clk\_1MHz | (void) |

* word **ds\_temp\_3** (void)

Faz três leituras de temperatura do DS18B20, espaçadas por 1 segundo. Ordena as medidas e retorna a central. O total é de 4 leituras porque inicia com um leitura dummy do DS18B20 e depois faz a 3 leituras cujos resultados serão usados.

* word **ds\_temp** (void)

Retorna em décimos de graus a temperatura lida pelo DS18B20.

Já dispara conversão para a próxima leitura. Uma conversão demora 0,75 seg.

* void **espera\_100ms** (word dur)

Esperar “dur”\*100ms. Tem a imprecisão de +/- 100 ms.

* void **t1\_config** (void)

Configurar Timer 1, CLK = 1 MHz, Fast PWM e modo invertido.

Preparado para contagens de 20 ms.

Fast PWM para acionar os leds.

* ISR: TIMER1\_OVF\_vect

Ocorre a cada 20 mseg. Duração típica de 100 microseg.

Poucas vezes chega a 500 micro seg.

**Configuração do Timer 2 com clock de 1 MHz**

CLK = 1 MHz 🡪 1.000.000 x 0,02 = 20.000 🡪 ICR1 = 20.000-1 = 19.999

#define TIMER1\_20ms 19999

Gabarito para configurar os registradores do TC2, Modo Normal

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **TCCR2A** | COM2A1 | COM2A0 | COM2B1 | COM2B0 | - | - | WGM21 | WGM20 |
| 0 | 0 | 0 | 0 |  |  | 0 | 0 |
| **TCCR2B** | FOC2A | FOC2B | - | - | WGM22 | CS22 | CS21 | CS20 |
| 0 | 0 |  |  | 0 | 0 | 0 | 1 |
| **TIMSK2** | - | - | - | - | - | OCIE2B | OCIE2A | TOIE2 |
|  |  |  |  |  | 0 | 0 | 0 |
| **TIRF2** | - | - | - | - | - | OCF2B | OCF2A | TOV2 |
|  |  |  |  |  |  |  |  |
| **ASSR** | - | EXCLK | AS2 | TCN2UB | OCR2AUB | OCR2BUB | TCR2AUB | TCR2BUB |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **GTCCR** | TMS | - | - | - | - | - | PSRASY | PSRSYNC |
| 0 |  |  |  |  |  | 0 | 0 |

CS22 CS21 CS20 = 0 0 1 🡪 Seleciona CLK sem divisor, logo seleciona 1 MHz.

WGM22 WGM21 WGM20 = 0 0 0 🡪 Modo 0, Normal, Topo = 0xFF.

TOV2 = 1 🡪 quando acontece overflow

OCF2A e OCF2B 🡪 coincidência com os comparadores OCR2A e OCR2B.

TCNT2 🡪 é o contador.

**EEPROM**

26/02/2021

Arduino Nano (Mega328P) tem uma EEPROM de 1024 bytes.

#define EEPROM\_TAM 1024 //Tamanho da EEPROM interna do ATMega

Funções para a EEPROM

|  |  |  |
| --- | --- | --- |
|  |  |  |
| void | eeprom\_dir\_apaga | (byte x) |
| void | eeprom\_dir\_cria | (byte nr, char \*nome) |
| void | eeprom\_dir\_busca | (void) |
| void | eeprom\_dir\_ser | (void) |
| void | eeprom\_dir\_tudo\_ser | (void) |
| void | eeprom\_ser\_config | (void) |
|  |  |  |
| void | eeprom\_volta\_sim | (int modo) |
| void | eeprom\_volta\_nao | (void) |
|  |  |  |
| void | eeprom\_apaga | (void) |
| void | eeprom\_apaga\_config | (void) |
| void | eeprom\_apaga\_dir | (void) |
| void | eeprom\_wr\_ff | (word adri, word qtd) |
| void | eeprom\_wr\_seq | (word adr, byte \*vet, word qtd) |
| void | eeprom\_rd\_seq | (word adr, byte \*vet, word qtd) |
|  |  |  |
| long | eeprom\_rd\_word32 | (word adr) |
| word | eeprom\_rd\_word | (word adr) |
| void | eeprom\_wr\_word32 | (word adr, long dado) |
| void | eeprom\_rd\_vet\_int | (word adr, int \*vt, byte qtd) |
| void | eeprom\_wr\_word | (word adr, word dado) |
| byte | eeprom\_wr\_str | (word adr, byte \*vet) |
| byte | eeprom\_rd\_str | (word adr, byte \*vt, byte qtd) |
| void | eeprom\_wr\_byte | (word adr, byte dado) |
| byte | eeprom\_rd\_byte | (word adr) |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

* void **eeprom\_volta\_sim** (word md)

Gravar na EEPROM o código SS para que ao ligar, seja restaurado do modo anterior.

Modo anterior “md”.

* void **eeprom\_volta\_nao** (void)

Gravar na EEPROM o código NN para que ao ligar, NÃO seja restaurado do modo anterior.

* void **eeprom\_apaga** (void)

Gravar 0xFF em toda a EEPROM.

* void **eeprom\_apaga\_config** (void)

Gravar 0xFF na área de configuração (0,...,63). Está definido EE\_DIR\_INI=64

* void **eeprom\_apaga\_dir** (void)

Gravar 0xFF na área de diretórios (64,...,1023). Está definido EE\_DIR\_INI=64 e EEPROM\_TAM=1024

* void **eeprom\_wr\_ff** (word adri, word qtd){

Gravar qtd bytes 0xFF numa faixa de endereços. adri=endereço inicial.

* void **eeprom\_wr\_seq** (word adr, byte \*vet, word qtd)

Gravar qtd bytes do vetor vet a partir de adr.

* void **eeprom\_rd\_seq** (word adr, byte \*vet, word qtd)

Ler qtd bytes a partir do endereço adr e gravar no vetor vet.

* void **eeprom\_wr\_word** (word adr, word dado)

Gravar a word dado no endereço adr. É big endian, grava primeiro o MSB.

* void **eeprom\_rd\_vet\_int** (word adr, int \*vet, byte qtd)

Ler uma certa quantidade de inteiros da EEPROM.

* word **eeprom\_rd\_word** (word adr)

Retorna a word do endereço adr. É big endian, lê primeiro o MSB.

* byte **eeprom\_wr\_str** (word adr, byte \*vt)

Gravar na EEPROM uma string terminada em ZERO.

* void **eeprom\_wr\_byte** (word adr, byte dado)

Gravar o byte dado no endereço adr.

* byte **eeprom\_rd\_str** (word adr, byte \*vt, byte qtd)

Ler da EEPROM uma string terminada em ZERO.

qtd = limite para leitura, incluindo o zero final, se houver

Retorna quantidade de bytes lidos incluindo o ZERO.

* byte **eeprom\_rd\_byte** (word adr)

Retorna o byte do endereço adr.

**Global**

Aqui está a lista das variáveis globais usadas pelo Cachacinha.

**Desenvolvimento da Biblioteca One Wire**

Aqui estão alguns detalhes do desenvolvimento da biblioteca One Wire. Eles serão úteis para adaptá-las para os demos casos. O KXC trabalha em 1 MHz e isso gerou alguns pequenos problemas.

1) Geração de atrasos

Primeiro se pensou em usar usado o Timer 2 no modo Normal com clk = 1 MHz. A função abaixo geraria o atraso, mas tem um lastro (overhead) de 20 µs e o argumento atz não deve ser menor que 7. Como clk da CPU muito baixo, ficou ruim para gerar pulsos estreito como de 10 µs.

void atz\_micro(byte atz){

TCNT2=0;

OCR2A=atz-1;

TIFR2 |= (1<<OCF2A);

while( (TIFR2&(1<<OCF2A)) == 0);

}

A solução foi fazer atrasos por laços de programa usando o assembly NOP.

asm("nop;");

2) Geração do sinal de Reset

Diagrama

Descrição gerada automaticamente

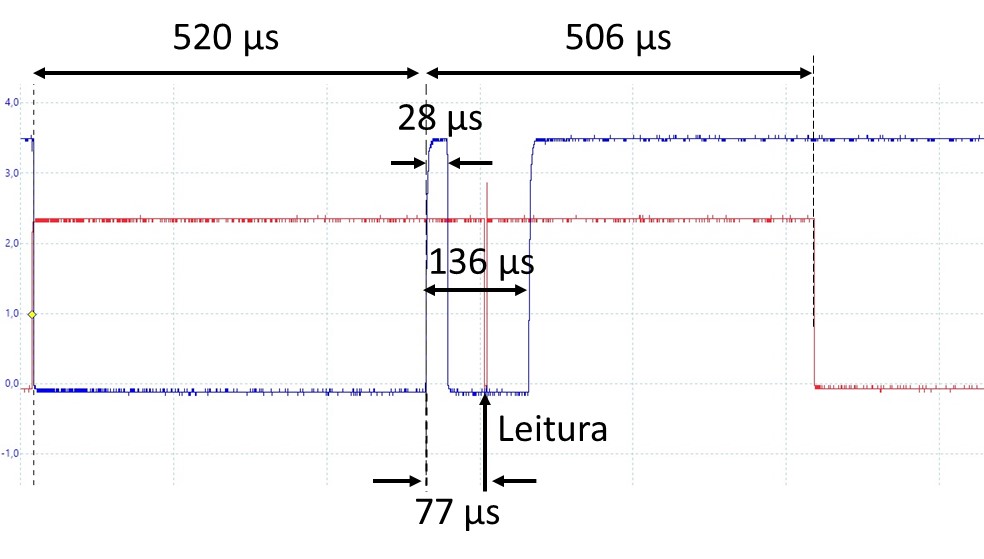


Fig OW1. Sinalização do Reset.

3) Escrita de bits 0 e 1

Diagrama, Esquemático

Descrição gerada automaticamente

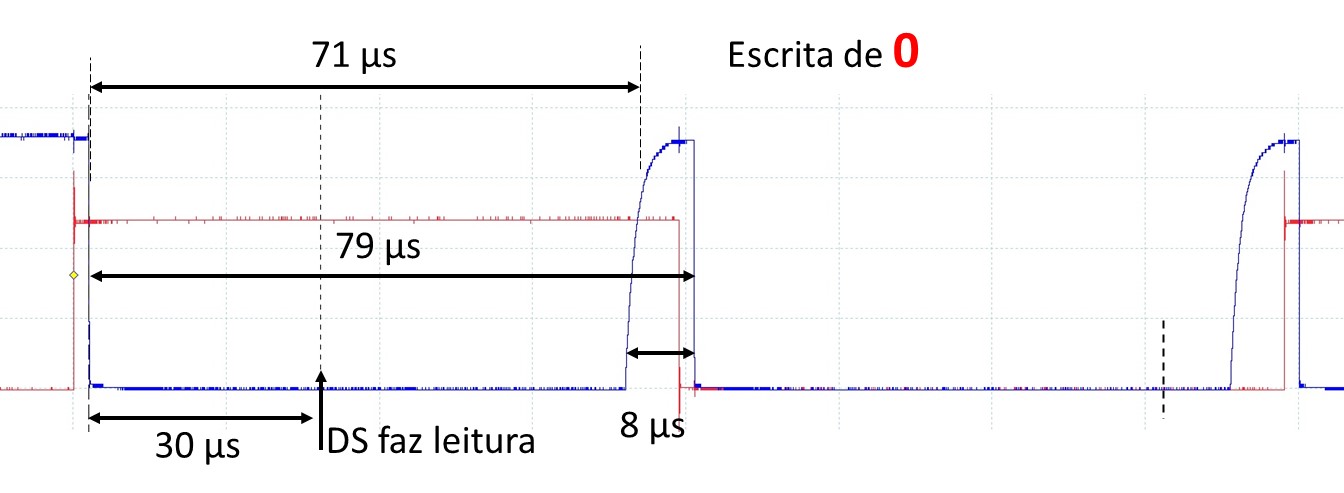


Fig OW2. Sinalização para escrita de ZERO.

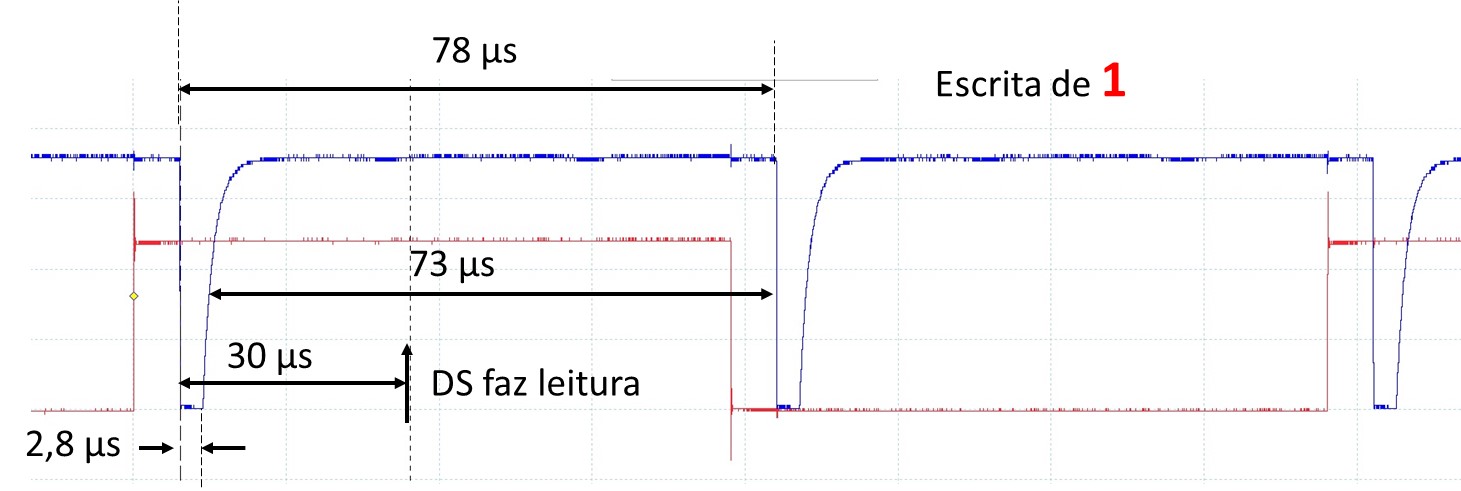


Fig OW3. Sinalização para escrita de UM.

4) Leitura de 0 e 1

Diagrama, Desenho técnico

Descrição gerada automaticamente

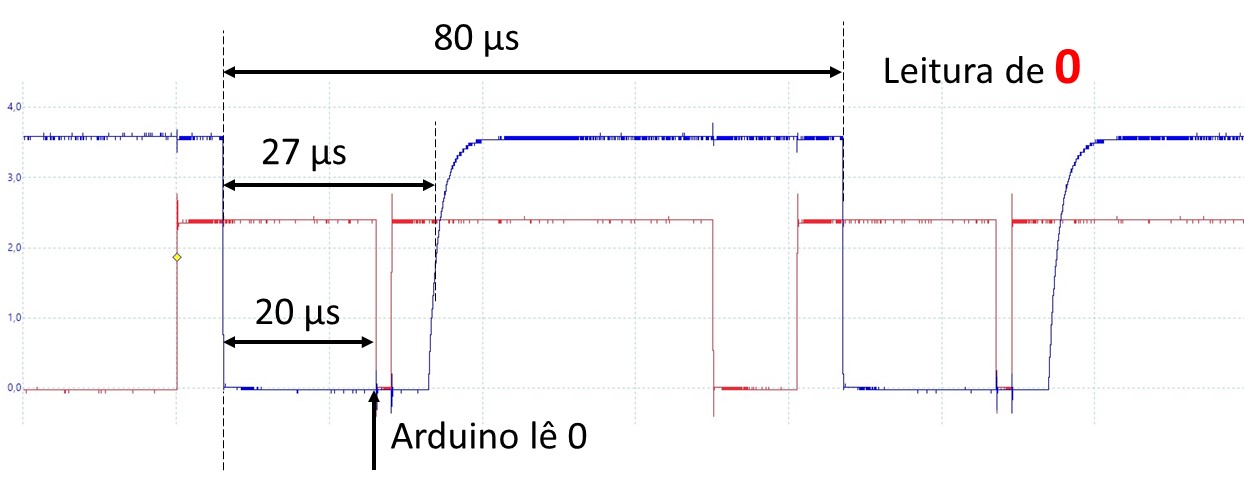


Fig OW4. Sinalização para a leitura de ZERO.

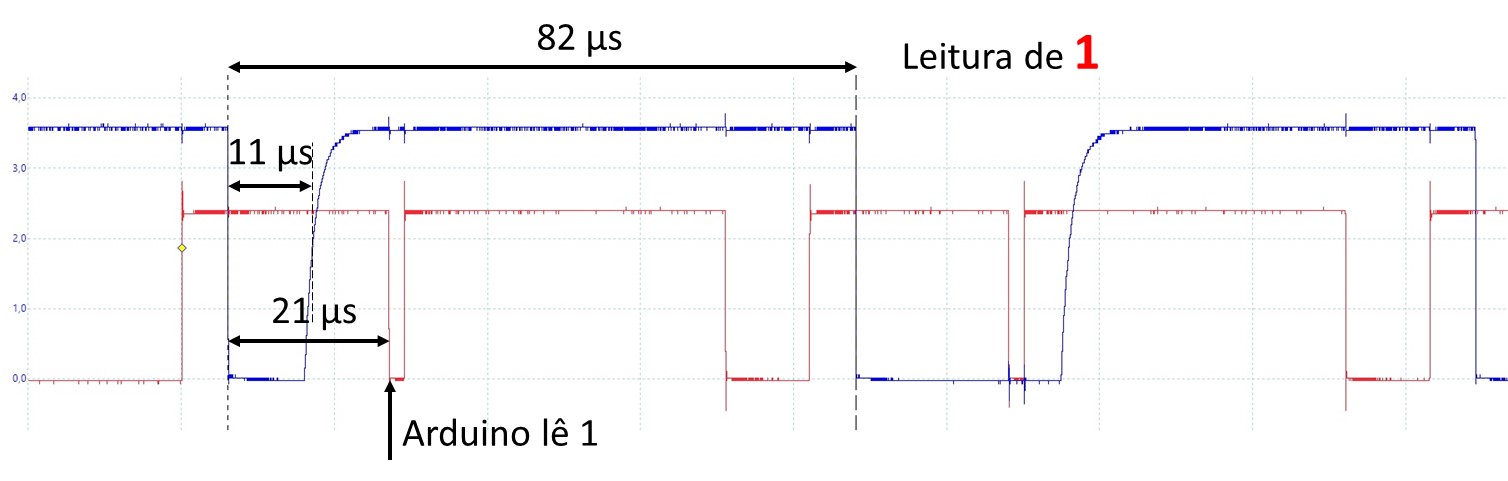


Fig OW4. Sinalização para a leitura de UM.

LASTRO

Ensaios para ver a melhor posição e o valor do lastro.

|  |  |
| --- | --- |
| Posição 0 – sem lastro | Posição 1 |
| Posição 2 | Posição 3 |
| Posição 4 |  |

Usada a taxa de 5 conversões por segundo. Cada ensaio durou, pelo menos, 3 minutos.

Densímetro de ponta cabeça para marcar o início de o fim de cada ensaio.