

RFID

Identificação por rádio frequência (*Radio Frequency IDentification*)

Há mais de uma forma de comunicação possível: SPI (*Serial Peripheral Interface*), I2C (*Inter-Integrated Circuit*) e por meio de UART (*Universal Asynchronous Receiver/Transmitter*).

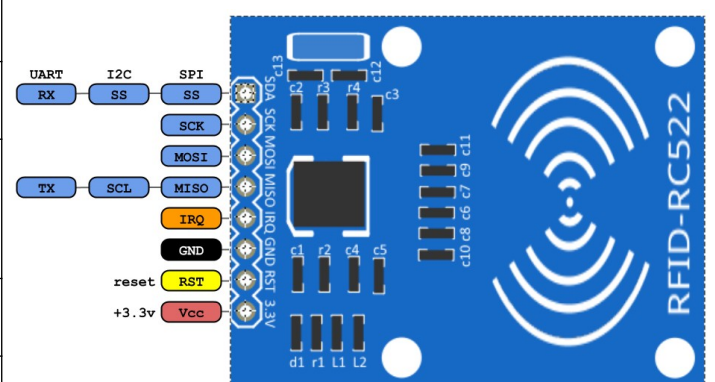
Tanto o SPI quanto o I2C são síncronos, isto é, dependem de um sinal de sincronismo (*clock*). A UART pode ser síncrona ou assíncrona; quando atuando em modo síncrono é chamada de USART (*Universal Synchronous Asynchronous Receiver Transmitter*).

SPI e I2C usam os termos ‘mestre’ e ‘escravo’. Mestre (*master*) é quem inicia e controla a comunicação, enquanto escravo (*slave*) é quem responde aos comandos do mestre. Vários dispositivos podem ser conectados utilizando-se esta configuração: até 127 em I2C, usando-se dois fios; e, usando 4 fios, tantos quantos forem possíveis gerar sinais de seleção (SS) com o SPI. Chamamos isto de barramento.

Quando se usa o SPI, a comunicação do *master* para o *slave* é identificada como MOSI (*Master Output Slave Input* – podendo estar marcado também como SDO ou DO ou Dout ou SO), e quando vai do *slave* para o *master* como MISO (*Master Input Slave Output* – podendo estar marcado como SDI ou DI ou SI ou Din). Ainda, são usados os sinais de sincronismo (ou *clock*, marcado como SCK ou SCLK ou CLK) e o de seleção de dispositivo *slave* (SS ou *slave select*, podendo estar marcado como CS ou nSS ou nCS).

Quando se usa o I2C, a comunicação será controlada por meio do endereço (em geral identificado por um número hexadecimal) do dispositivo escravo. O *master* escreverá no barramento o endereço do dispositivo com o qual ele quer se comunicar e o *slave* correspondente responderá. Para isso, são usados os sinais de controle, identificados como: SDA (*Serial Data*), para troca de dados, e SCL (*Serial Clock*), para sincronismo. O SDA funciona de forma bidirecional: é uma saída do *master* e entrada para o *slave* no momento inicial e de envio de comandos, e, depois, funciona como uma entrada do *master* e saída no *slave* para envio de dados do *slave* ao *master*.

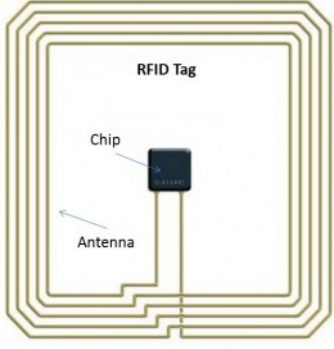
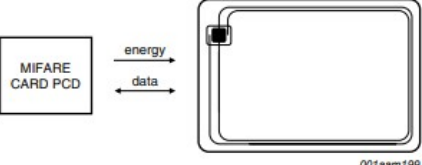
O cartão leitor de ‘tags’ (etiquetas) que iremos utilizar está representado a seguir, com breve explicação do significado de seus pinos de conexão.

SDA – Pode ser uma entrada (quando se usa a comunicação SPI), como transmissão de dados seriais quando está sendo usada a comunicação I2C, ou como entrada de dados quando está sendo usada a UART	
SCK - Usado para sincronizar os pulsos de <i>clock</i> do barramento SPI	
MOSI – <i>Master Out/ Slave In</i> para o módulo RC522, quando usando-se o SPI.	
MISO - / SCL / TX - <i>Master In/ Slave Out</i> para o módulo RC522, quando usando-se o SPI. É usado como sincronismo (relógio, SCL) quando se está usando o I2C, ou como saída de dados quando usando-se UART.	
IRQ – Este pino gera um sinal (<i>interrupt request</i>) para alertar o controlador (Arduino) quando uma ‘tag’ estiver próxima.	
GND – Energia, terra, 0V.	
RST – É o <i>reset</i> , ou reinicialização. Se for colocado em nível baixo (0V, GND) o RC522 fica inativo.	
VCC – Energia, +3,3 V. ATENÇÃO: é 3,3Volts, não conecte no +5V!	

Para facilitar o controle do cartão, utilizamos de uma biblioteca de códigos. As bibliotecas disponíveis costumam ser voltadas à comunicação SPI, então, usaremos uma destas, por ser mais comum.

Porém, para que a comunicação funcione, temos mais um elemento: a ‘tag’. Foi usado o nome *tag* (etiqueta) em função de que uma das aplicações pensada foi justamente a de colocar etiquetas sensíveis a rádio frequência em produtos diversos (embora a *tag* possa estar presente em cartões e outras apresentações).

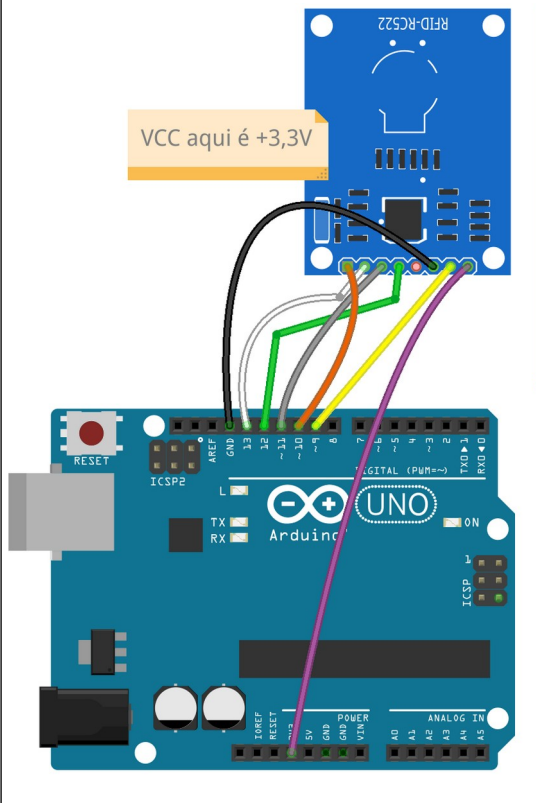
Este tipo de dispositivo faz parte de uma grande ‘família’ conhecida como NFC (*Near Field Communication* – comunicação por campo próximo; seu celular ou seu cartão de crédito podem ter esta característica), que inclui o subgrupo PICC (*Proximity Inductive Coupling Card*, cartão de acoplamento indutivo de proximidade)¹.

<p>A etiqueta possui uma antena. A antena recebe (e é capaz de transmitir) sinais na faixa de rádio frequência (uma frequência comum em <i>tags</i> RFID é a de 13,56MHz).</p> <p>Para funcionar, o <i>chip</i> interno precisa de energia. Ela é obtida por meio do próprio sinal de rádio frequência: ele é retificado e convertido em corrente contínua para fornecer energia ao <i>chip</i>. Assim, o <i>chip</i> não precisa de alimentação externa vinda de pilhas, baterias ou fontes – enquanto estiver próximo a um campo de rádio frequência ele terá energia.</p> <p>Quando recebe energia, o <i>chip</i> gera um sinal de rádio frequência e insere na comunicação um código de identificação (ID), o qual foi previamente cadastrado – podendo, ou não, ser atualizável.</p>	 <p>Diagrama de uma etiqueta RFID. No topo, há uma antena formada por várias espiras concêntricas. Abaixo da antena, há um chip retangular. O chip está conectado à antena por uma linha de conexão.</p>
<p>A comunicação PCD (<i>Proximity Coupling Device</i>, acoplamento de dispositivo por proximidade) segue padrões internacionais (ex.: ISO/IEC 14443). Algumas <i>tags</i> RFID poderão oferecer a possibilidade de gravação de dados e mesmo de mudança de código de identificação</p> <p>Ver: https://www.mouser.com/datasheet/2/302/MF1S503x-89574.pdf e https://nfc-tools.github.io/resources/standards/iso14443A/</p>	 <p>Diagrama de comunicação entre um cartão MIFARE CARD PCD e uma antena. O cartão está à esquerda, e a antena está à direita. Há duas setas entre eles: uma seta verde apontando da antena para o cartão, rotulada 'energy', e uma seta vermelha apontando do cartão para a antena, rotulada 'data'.</p>

Para aumentar o alcance (para até uns 10 metros), algumas *tags* poderão possuir bateria interna.

Vamos testar!

Monte o *hardware* a seguir:

 <p>Fotografia da montagem do hardware. Um módulo RFID-RC522 (azul) está conectado a um Arduino Uno (verde) por meio de fios coloridos. O módulo RFID está no topo, e o Arduino está na base. Os fios conectam os pinos do módulo ao Arduino.</p>	<p>RST no pino 9 do Arduino</p> <p>SDA no pino 10 do Arduino</p> <p>MOSI no pino 11 do Arduino</p> <p>MISO no pino 12 do Arduino</p> <p>SCK no pino 13 do Arduino</p> <p>GND no GND do Arduino</p> <p>VCC no 3,3V do Arduino !!!</p>	<p><u>Atenção para a ligação do VCC: é em 3,3V, não em 5V!</u></p> <p>RST no pino 9 do Arduino</p> <p>SDA no pino 10 do Arduino</p> <p>MOSI no pino 11 do Arduino</p> <p>MISO no pino 12 do Arduino</p> <p>SCK no pino 13 do Arduino</p> <p>GND no GND do Arduino</p> <p>VCC no 3,3V do Arduino !!!</p> <p>(IRQ fica desconectado)</p>
---	--	---

¹ Ver: <http://www.nfc-research.at/index.php?id=40.html>

Depois, na Ide do Arduino adicione a biblioteca (Sketch / Incluir biblioteca) para trabalho com RFID (RC522), ou adicione o arquivo ‘zip’ (Sketch / Incluir biblioteca / Adicionar biblioteca zip), se você tiver um – em nosso exemplo será usada a ‘RFID MASTER’ disponível em: <https://github.com/miguelbalboa/rfid>. Em seguida, abra os exemplos (Arquivo/ Exemplos/ MFRC522) e procure por **DumpInfo**. Compile e envie ao Arduino. Ligue o monitor serial e aproxime uma *tag* no leitor. Você verá algo assim (os valores mudarão em função de sua tag e de haver, ou não, dados gravados nela):

```
Card UID: 2A A1 F0 1E
Card SAK: 08
PICC type: MIFARE 1KB
Sector Block 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 AccessBits
15 63 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
62 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
14 59 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
58 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
57 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
56 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
13 55 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
54 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
53 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
52 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
12 51 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
49 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
11 47 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
45 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
44 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
10 43 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
42 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
9 39 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
38 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
37 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
36 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
8 35 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
34 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
33 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
7 31 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
29 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
6 27 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
26 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
25 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
24 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
5 23 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
4 19 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
17 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
3 15 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
13 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
2 11 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
1 7 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
0 3 00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF [ 0 0 1 ]
2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 [ 0 0 0 ]
0 2A A1 F0 1E 65 08 04 00 62 63 64 65 66 67 68 69 [ 0 0 0 ]
```

Para o meu caso a identificação da *tag* é **2A A1 F0 1E** (são sempre 4 ou 7 bytes). E, ela possui **1KB**, de memória (16 setores x 4 blocos x 16 bytes = 1024 bytes => 1k - mas, há *tags* com 4k²). O conteúdo exibido acima corresponde ao conteúdo armazenado na memória da *tag*. Cada *tag* é de um tipo.


Tente com as suas *tags*. Registre as identificações nas ‘*tags*’, pois vamos usá-las depois.

Agora, abra novamente os exemplos e selecione **ReadNUID**. Compile, envie ao Arduino e abra o monitor serial. Em síntese, agora você verá somente algumas características da *tag* – não verá, por exemplo, o conteúdo da memória.

Você verá algo como mostrado a seguir:

```
A new card has been detected.  
The NUID tag is:  
In hex:  2A A1 F0 1E  
In dec:  42 161 240 30  
PICC type: MIFARE 1KB  
A new card has been detected.  
The NUID tag is:  
In hex:  37 00 38 33  
In dec:  55 0 56 51  
PICC type: MIFARE 1KB  
A new card has been detected.  
The NUID tag is:  
In hex:  F7 97 2F 33  
In dec:  247 151 47 51  
PICC type: MIFARE 1KB  
A new card has been detected.  
The NUID tag is:  
In hex:  2A A1 F0 1E  
In dec:  42 161 240 30  
PICC type: MIFARE Ultralight or Ultralight C  
Your tag is not of type MIFARE Classic.
```

Há vários tipos de *tags*. A da figura que segue não é no padrão usado pelo exemplo *ReadNUID* (e foi reportada como ‘*Your tag is not of type MIFARE Classic*’, na cópia de tela acima). Ela pode ser lida por meio do programa exemplo *DumpInfo*, anteriormente utilizado:

Conteúdo – notar que o ID possui 7 bytes:	Aspecto da <i>tag</i> :
<pre>Card UID: 04 1C 67 F2 DE 64 80 Card SAK: 00 PICC type: MIFARE Ultralight or Ultralight C Page 0 1 2 3 0 04 1C 67 F7 1 F2 DE 64 80 2 C8 48 00 00 3 E1 10 12 00 4 01 03 A0 0C 5 34 03 00 FE 6 00 00 00 00 7 00 00 00 00 8 00 00 00 00 9 00 00 00 00 10 00 00 00 00 11 00 00 00 00 12 00 00 00 00 13 00 00 00 00 14 00 00 00 00 15 00 00 00 00</pre>	

Vamos usar a memória da *tag*!

Escolha uma *tag*; abra e execute o exemplo **DumpInfo**. Guarde uma cópia dos dados exibidos (você pode salvar a tela ou copiá-los e guardar em um editor de textos; a ideia é verificar as diferenças em função dos próximos experimentos).

Depois, nos exemplos, abra o arquivo **rfid_write_personal_data**. Após compilar e enviar o código ao Arduino, vamos interagir com o monitor serial (deixe a *tag* sobre o leitor).

Vamos gravar dados na *tag*. No exemplo de programa em questão, forneça o sobrenome ("*Type Family name, ending with #*") e o nome ("*Type first name, ending with #*"), conforme solicitado.

Agora, nos exemplos, abra o arquivo ***rfid_read_personal_data***. Após compilar e enviar o código ao Arduino, vamos interagir com o monitor serial (deixe a *tag* sobre o leitor) para ler seu conteúdo de memória:

```
Read personal data on a MIFARE PICC:
Read personal data on a MIFARE PICC:
**Card Detected:**
Card UID: 37 00 38 33
Card SAK: 08
PICC type: MIFARE 1KB
Name:
JOSE SIMAO
SIMAO
**End Reading**
```

Por fim, volte ao ***DumpInfo***. Leia novamente o conteúdo da *tag* e compare com aquele que você havia armazenado. Em meu caso, ficou como exibido a seguir (note a diferença):

Antes da gravação:																Depois da gravação:													
1	7	00	00	00	00	00	00	FF	07	80	69	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	[0 0 1]
	6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	[0 0 0]
	5	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	[0 0 0]
	4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	[0 0 0]
0	3	00	00	00	00	00	00	FF	07	80	69	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	[0 0 1]
	2	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	[0 0 0]
	1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	[0 0 0]
	0	F7	97	2F	33	7C	08	04	00	62	63	64	65	66	67	68	69												[0 0 0]
1	7	00	00	00	00	00	00	FF	07	80	69	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	[0 0 1]
	6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	[0 0 0]
	5	20	20	20	20	41	4F	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	[0 0 0]
	4	0D	0A	55	46	50	52	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	[0 0 0]
0	3	00	00	00	00	00	00	FF	07	80	69	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	[0 0 1]
	2	20	53	49	4D	41	4F	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	[0 0 0]
	1	4A	4F	53	45	20	53	49	4D	41	4F	0D	0A	4A	4F	53	45												[0 0 0]
	0	F7	97	2F	33	7C	08	04	00	62	63	64	65	66	67	68	69												[0 0 0]

O que eu gravei na *tag*:

```
Read personal data on a MIFARE PICC:
Read personal data on a MIFARE PICC:
**Card Detected:**
Card UID: F7 97 2F 33
Card SAK: 08
PICC type: MIFARE 1KB
Name:
UFPR JOSE SIMAO
JOSE
**End Reading**
```

Se você quiser, pode utilizar outras formas de comunicação serial (creio ser mais simples usar outra placa, mas a RC522 é mais barata e mais comum) ao invés da SPI:

- Para trabalhar com I2C a biblioteca deverá ser outra. Tente esta - <https://github.com/arozcan/MFRC522-I2C-Library>. O endereço do dispositivo, para a placa RC522, será 0x3C. Mas, para funcionar você deverá realizar uma pequena modificação na placa: <https://europe1.discourse-cdn.com/arduino/original/4X/5/d/b/5db77e1e84b1fb9977963604ad67d22511a6fc59.jpeg>
- Para trabalhar com UART - <https://github.com/zodier/MFRC522-UART-Arduino> (esta eu não testei)
- Veja mais informações no fórum do Arduino em <https://forum.arduino.cc/t/rc522-rfid-rc522-switching-spi-to-uart-interface-or-i2c-possible/425741/17>

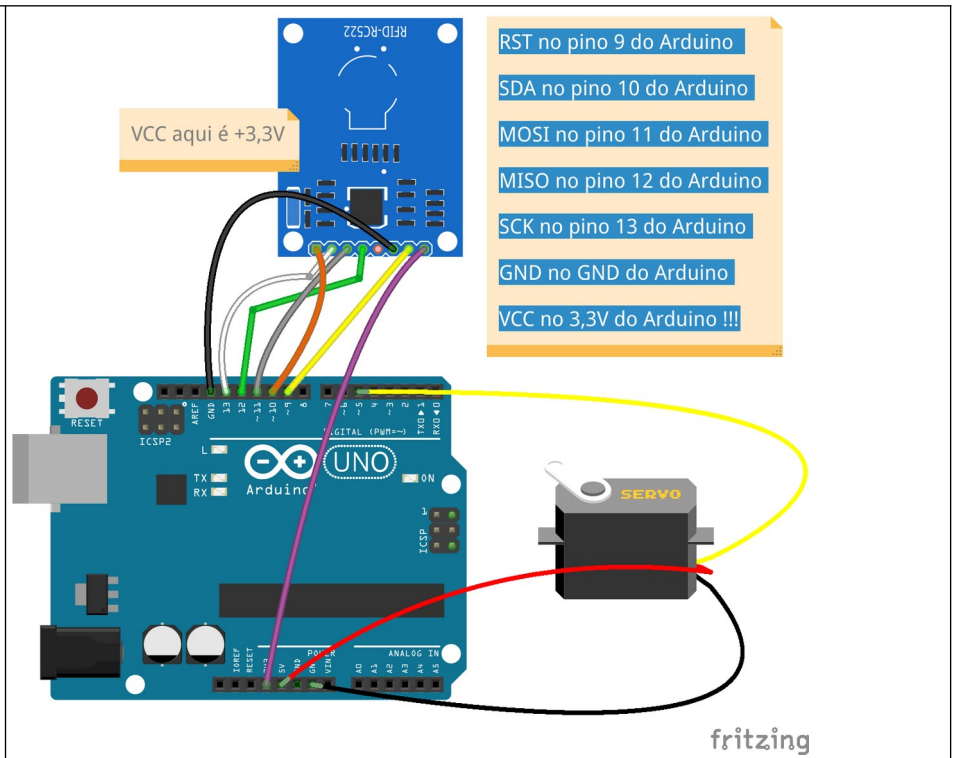
Agora, vamos terminar as experiências com RFID controlando um servo motor, em função do valor lido em uma 'tag'!

Monte o seguinte *hardware* (somente foi acrescentado um servo, já visto):

Foi usado o mesmo *hardware* utilizado para trabalho com as *tags*, acrescentando-se somente um servo motor.

O servo motor foi ligado de forma a usar o GND, VCC em +5V e o pino de controle do ângulo de rotação, cujo sinal de controle foi conectado no pino digital 5 do Arduino.

Cuidado – não ‘misture’ o VCC do RFID (3,3V) com o do servo motor (5V), sob pena de queimar seu Arduino.



Nosso objetivo é: ao ser detectada uma *tag* de código conhecido, fazer com que o servo motor se movimente 90° (por exemplo, para abrir uma cancela), por alguns segundos (depois ele retorna a 0° e ‘fecha’ a cancela). Obviamente, ao invés de controlarmos um servo, o comando poderia ser ligar um led, um relé, enviar uma mensagem para um *display*, emitir um som... a escolha é sua (e, sendo o caso de mudar, avalie as novas conexões e seu registro no código - #define, pinMode - e necessidades de outros código/ outras bibliotecas).

O trabalho com RFID exige uma sequência de comandos para controle da comunicação, motivo pelo qual estamos utilizando uma biblioteca. Como já temos códigos funcionais, para realizar nosso experimento vamos adaptar um dos códigos já utilizados (vamos usar o exemplo **ReadNUID.**) e incluir nosso acionamento do servo.

```
1 /*
2  * Adaptado de: https://github.com/miguelbalboa/rfid
3  * -----
4  * MFRC522      Arduino Uno
5  * Pino        Pino
6  * -----
7  * RST          9
8  * SDA(SS)      10
9  * MOSI         11
10 * MISO         12
11 * SCK          13
12 *
13 * ATENÇÃO - o módulo usa 3,3V e não 5V no VCC !!!
14 *
15 * servoMotor ligado no pino 5
16 */
17
18 //Bibliotecas de códigos
19 #include <SPI.h> //biblioteca de comunicação serial protocolo SPI
20 #include <MFRC522.h> //biblioteca de controle do cartão RC522
21 #include <Servo.h> //para usar o servo motor
22
23 //Constantes
24 #define pinoReset 9 //para acesso ao sinal RST do módulo
25 #define pinoSelect 10 //para acesso ao sinal de seleção do chip do módulo
26 #define pinoServo 5 //para acesso ao controle do servo
27 #define quantidadeTags 3 //quantas tags eu conheço?
28
29 //Variáveis
30 MFRC522::MIFARE_Key key; //para acesso aos dados da tag, código original
31 //IDs das 'quantidadeTags' conhecidas:
32 uint8_t tagsConhecidas[quantidadeTags][4] = {{0x4B, 0xB1, 0x3B, 0x09},{0xD7, 0xBC, 0xD1, 0x4E},{0x2A, 0xA1, 0xF0, 0x1E}};
33 uint8_t IDdaTagEmTeste[4]; //tag a ser lida
34
35 //Objetos
36 MFRC522 RC522(pinoSelect, pinoReset); //cria uma instância passando os valores dos pinos
37 Servo servoMotor; //para controle do servo motor
```


As linhas 19 a 21 definem as bibliotecas de códigos a serem utilizadas: para comunicação serial (SPI), comunicação com o módulo RFID (MFRC522) e com o servo motor (Servo). As linhas 24 a 27 definem algumas constantes para acesso ao módulo (pinos de *reset*, 9, e seleção do módulo, 10), para acesso ao controle do servo (pino 5) e para a quantidade de *tags* que iremos armazenar como sendo ‘conhecidas’.

Nas linhas 30, 32 e 33 foram declaradas variáveis para trabalho com informações da *tag* (*‘key’*, na linha 30), para armazenar Ids das *tags* conhecidas (linha 32) e da *tag* que está sob teste (linha 33). O tipo de variável é `uint8_t`, que significa que é uma variável que armazena números inteiros (`int`), sem sinal (`u`, de *unsigned*, sem sinal) de 8 *bits* de tamanho (ou seja, números ‘pequenos’, de apenas um *byte*).

A declaração da linha 32, `uint8_t tagsConhecidas[quantidadeTags][4]`, tem a seguinte explicação:

- trata-se do armazenamento de números inteiros, sem sinal (positivos), de 8 bits (`uint8_t`);
- o nome da variável é `tagsConhecidas`;
- a variável é matricial, formada por dois vetores: `[quantidadeTags]` e `[4]`;
- `‘[quantidadeTags]’` é uma constante, declarada na linha 27 (e que tem, neste exemplo, o valor 3); `‘[4]’` é também uma constante (valor 4);

A leitura a ser realizada é: temos um **conjunto** formado por 3 (`‘[quantidadeTags]’`) sequências de 4 *bytes* (`‘[4]’`). Isto permite que armazenemos o conjunto de IDs das *tags* de interesse (pois são sequências de 4 *bytes*, conforme vimos nos exemplos). Neste exemplo, os valores dos Ids são: {0x4B, 0xB1, 0x3B, 0x09}, {0xD7, 0xBC, 0xD1, 0x4E} e {0x2A, 0xA1, 0xF0, 0x1E}; note que são três conjuntos, identificados entre `‘{’` e `‘}’` (para formar um conjunto), e, que cada um deles possui uma sequência de 4 *bytes*, representados em seus valores hexadecimais (o que, na linguagem utilizada, nos obriga a iniciar a declaração com `‘0x’`, para que o compilador entenda que é um número hexadecimal).

Nas linhas 36 e 37 são declarados dois objetos (os quais, portanto, possuem várias funções incorporadas por meio das declarações de classes de suas bibliotecas): um para trabalharmos com o módulo RFID (`RC522`, linha 36, que recebe como parâmetros os números dos pinos de controle necessários, declarados nas constantes das linhas 25 e 24) e outro para trabalharmos com o servo motor (`servoMotor`, linha 37).

```
39 //condições iniciais
40 void setup()
41 {
42     Serial.begin(9600); //comunicação serial, velocidade 9600bps
43     SPI.begin(); //inicializa a comunicação com protocolo SPI
44     RC522.PCD_Init(); //inicia a instância do módulo RC522
45     servoMotor.attach(pinoServo); //informa o pino de controle do servo
46     servoMotor.write(0); //inicia colocando o servo motor na posição zero graus
47     Serial.println(F("Aguardando tag")); //Avisa o usuário
48 }
```

As condições iniciais ao funcionamento do Arduino estão estabelecidas na função `‘void setup()’`, entre as linhas 42 e 47. Na linha 42 declaramos que usaremos a interface serial com velocidade de 9600 *bits* por segundo, bps. A linha 42 inicializa a biblioteca de comunicações SPI e a linha 44 inicializa o módulo RFID. Na linha 45 informamos que conectamos um servo motor, e em qual pino (constante `pinoServo`, declarada na linha 26) e, na linha 46, passamos ao servo motor um comando para posicioná-lo em 0 graus (isto já foi visto). Finalmente, a linha 47 envia ao monitor serial a informação de que o circuito está pronto.

A seguir, temos mais duas funções. A de laço infinito (`‘void loop()’`), que controla o funcionamento do Arduino após o `setup()` ser executado, e uma função escrita para testar a identificação de uma *tag*, a qual retornará verdadeiro se ela for conhecida e falso se não for.

Vamos estudá-las.

```

50 //laço infinito
51 void loop()
52 {
53   if (RC522.PICC_IsNewCardPresent()) //Existe um novo cartão próximo ao leitor?
54   {
55     if (RC522.PICC_ReadCardSerial()) //Leia a ID do cartão
56     {
57       //código original adaptado
58       MFRC522::PICC_Type piccType = RC522.PICC_GetType(RC522.uid.sak); //le o 'sak', ver: https://www.newark.com/pdfs/techarticles/nxp/AN10833.pdf
59       Serial.println(RC522.PICC_GetTypeName(piccType)); //mostra informação lida
60       //Testa o tipo
61       if (piccType != MFRC522::PICC_TYPE_MIFARE_MINI && piccType != MFRC522::PICC_TYPE_MIFARE_1K && piccType != MFRC522::PICC_TYPE_MIFARE_4K) {
62         Serial.println(F("Sua tag nao e MIFARE Classic")); //informa se não for o tipo de tag esperada - pode ser modificado
63         return;
64       }
65       else{
66         //armazena o ID da tag na variável
67         for (byte i = 0; i < 4; i++) {
68           IDdaTagEmTeste[i] = RC522.uid.uidByte[i];
69         }
70         if(testaTag()){ //se for uma tag válida o valor retornado pela função será verdadeiro
71           servoMotor.write(90); //abre a cancela - aqui poderia ligar um led, emitir um som, etc ...
72           delay(5000); //aguarda 5 segundos
73           servoMotor.write(0); //fecha a cancela
74         }
75       }
76     }
77   }
78 }
79 delay(2000); //aguarda 2 segundos antes de testar novo cartão
80 }

```

O `loop()` ocorre entre as linhas 51 e 80. Na linha 53 temos um teste, correspondente à execução do método `'PICC_IsNewCardPresent()'` no objeto `RC522` (que foi declarado na linha 36); este método retorna um valor verdadeiro (`true`) se houver um novo cartão presente (nas proximidades do leitor), e falso (`false`) caso não exista/ não seja detectado. **SE (if)** o teste for verdadeiro (ou seja, se for detectada uma *tag*), será executado o bloco de instruções presente entre as linhas 54 e 78 (entre o `'{'` e `'}'` deste bloco `if`).

Se o teste for verdadeiro, na linha 55 será executado outro teste, agora quanto ao ao valor de retorno proporcionado pelo método `'PICC_ReadCardSerial()'`: foi possível ler? Caso tenha sido possível a leitura de informações da tag, será executado o bloco de instruções presente entre as linhas 56 e 77. Na linha 58 é declarada a variável `'piccType'`, a qual recebe o valor retornado pelo método `'PICC_GetType(RC522.uid.sak)'`, que informa o valor `'sak'`³ da *tag*. Este valor será testado na linha 61: caso não seja do tipo esperado o usuário será informado (linha 62) e o bloco termina (linha 63). Senão (`else`), caso o tipo seja conhecido, será executado o comando `for`, linhas 67 e 68, o qual irá copiar o valor da ID da *tag* que está sendo lida, do *byte* 0 até o *byte* 3, para a variável `IDdaTagEmTeste`, que foi definida na linha 33 (novamente, a declaração `'uint8_t IDdaTagEmTeste[4];'` é para um conjunto, neste caso de 4 *bytes* – ou seja, podemos armazenar o ID de uma *tag* de 4 *bytes*).

=> **Em resumo:** testamos se há um cartão presente e, se houver, obtemos seu tipo; se o tipo for válido, lemos a ID e armazenamos.

Na linha 70 o comando **SE (if)** testa o valor retornado pela função `testaTag()`, que será vista na sequência. Caso o valor de retorno seja verdadeiro (`true`), colocamos o servo em posição 90° (linha 71, `servoMotor.write(90)`), esperamos 5 segundos (linha 72, `delay(5000)`) e, finalmente, colocamos o servo em posição 0° (linha 73, `servoMotor.write(0)`).

Na linha 79 aguardamos 2 segundos (`delay(2000)`) antes de testar de novo, o que é realizado somente para não ficar testando continuamente (sendo o caso, pode ser excluída esta espera, ou modificada).

=> **Em resumo:** se a *tag* for conhecida, acionamos o servo motor para abrir a cancela, aguardamos 5 segundos e acionamos o servo motor para fechar a cancela.


```

82 boolean testaTag(){ //esta função retorna um valor booleano verdadeiro se a tag for conhecida e falso se não for
83     for (int tagEmTeste = 0; tagEmTeste < quantidadeTags; tagEmTeste++){ //para cada uma das tags conhecidas
84         if (IDdaTagEmTeste[0] == tagsConhecidas[tagEmTeste][0] &&
85             IDdaTagEmTeste[1] == tagsConhecidas[tagEmTeste][1] &&
86             IDdaTagEmTeste[2] == tagsConhecidas[tagEmTeste][2] &&
87             IDdaTagEmTeste[3] == tagsConhecidas[tagEmTeste][3] )
88             {
89                 return true; //se todos os bytes forem iguais a tag é conhecida
90             }
91     }
92     return false;
93 }

```

A função `testaTag()` está declarada entre as linhas 82 e 93 e deve retornar um valor booleano (`boolean`): verdadeiro (`true`) ou falso (`false`) em função de sua execução. Seu objetivo é verificar se o cartão que está sendo testado tem uma ID que faz parte do conjunto de cartões conhecidos (declarado na linha 32).

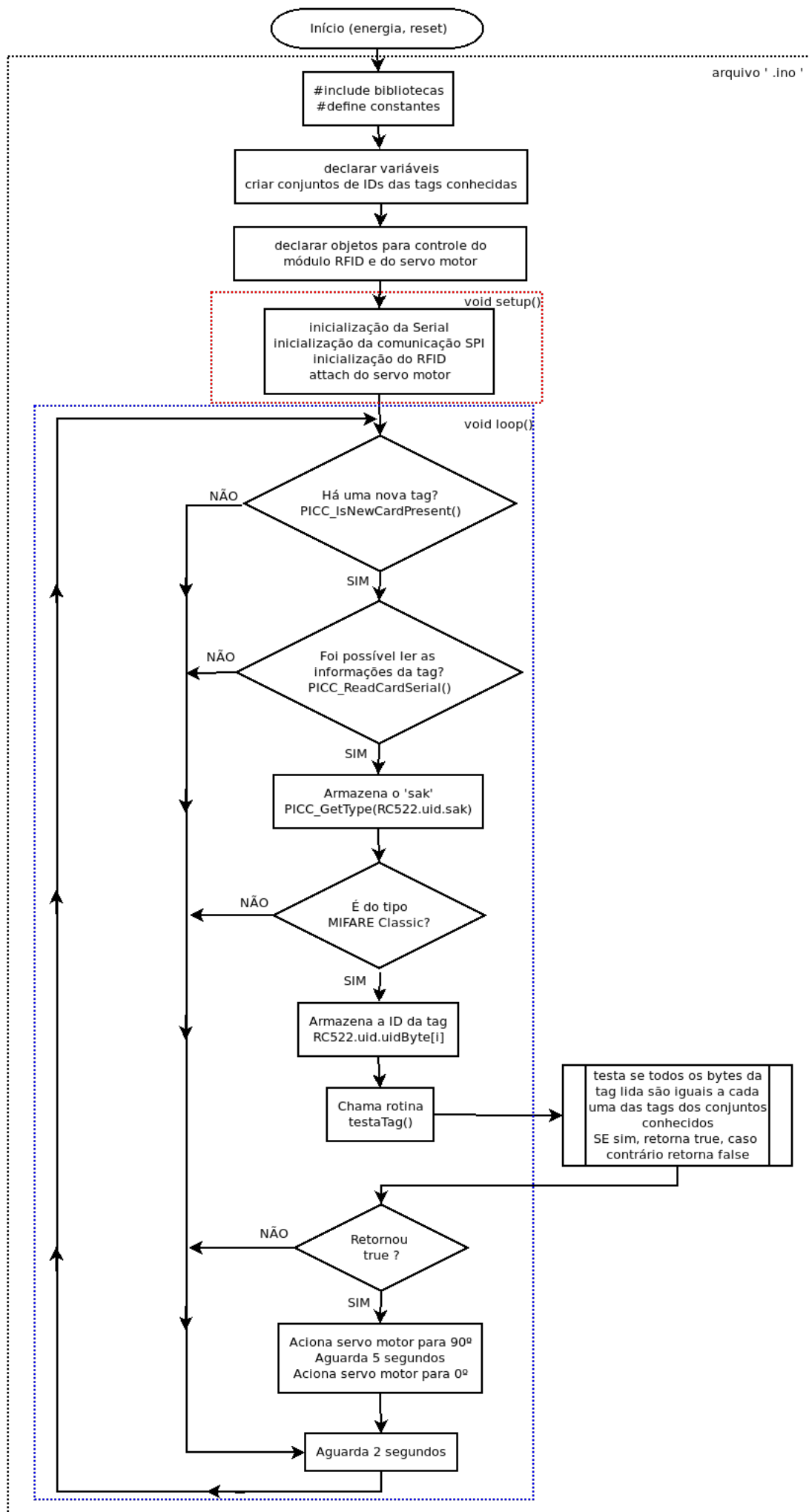
A declaração da linha 83 (`for (int tagEmTeste = 0; tagEmTeste < quantidadeTags; tagEmTeste++)`) permitirá que testemos da primeira (0) até a última (`quantidadeTags`, constante declarada na linha 27) *tag* conhecida, avançando de uma em uma (`++`). O teste ocorre entre as linhas 84 e 87, por meio do comando `if`.

SE o teste realizado pela comparação do ID da tag que está sendo testada (`IDdaTagEmTeste`), em todos os seus bytes (do `[0]` ao `[3]`), for igual (`==`) a um dos IDs (em todos os seus *bytes*) do conjunto `tagsConhecidas`, a função retornará verdadeiro (`true`). Ao retornar, o teste acaba (mesmo que existam outros IDs a testar).

Caso o laço `for` seja executado para todas as `tagsConhecidas` e não corra o retorno (ou seja, nenhum teste foi válido), a função retorna falso (`return false`, linha 92), indicando que a *tag* em teste não faz parte do conjunto de *tags* conhecidas pelo programa.

=> **Em resumo:** testamos o ID da *tag* que foi aproximada no módulo e, se ele for conhecido retornamos um valor verdadeiro para o teste, e, caso contrário, retornamos um valor falso.

A seguir temos um **fluxograma** (um tipo de diagrama usado para representar algoritmos) que representa o que foi exposto para o funcionamento de nosso programa leitor de *tags* para acionamento de um servo motor que controla uma cancela.



Dois grupos de tags.

Agora temos um desafio: queremos ter dois grupos de *tags*, um dos quais aciona um servo motor e o outro que aciona um segundo servo motor. Podemos ter *tags* que fazem parte dos dois grupos. E, os grupos não precisam ter a mesma quantidade de *tags*. Vamos aproveitar o código existente e duplicar algumas declarações, ajustando seus nomes quando necessário.

```
//Constantes
#define pinoReset 9 //para acesso ao sinal RST do módulo
#define pinoSelect 10 //para acesso ao sinal de seleção do chip dp módulo
#define pinoServo1 5 //para acesso ao controle do servo do grupo 1
#define pinoServo2 6 //para acesso ao controle do servo do grupo 2
#define quantidadeTagsGrupo1 3 //quantas tags eu conheço que pertencem ao grupo 1?
#define quantidadeTagsGrupo2 4 //quantas tags eu conheço que pertencem ao grupo 2?

//IDs das 'quantidadeTags' conhecidas por grupo; os grupos não precisam ter o mesmo tamanho, e uma tag pode estar em mais de um grupo:
uint8_t tagsConhecidasGrupo1[quantidadeTagsGrupo1][4] = {{0x4B, 0xB1, 0x3B, 0x09},{0xD7, 0xBC, 0xD1, 0x4E},{0x2A, 0xA1, 0xF0, 0x1E}};
uint8_t tagsConhecidasGrupo2[quantidadeTagsGrupo2][4] = {{0x10, 0xB1, 0x82, 0x25},{0xD7, 0xBC, 0xD1, 0x4E},{0xF7, 0x97, 0x2F, 0x33},{0xC7, 0xB2, 0xF2, 0x4D}};

Servo servoMotor1; //para controle do servo motor do grupo 1 de tags
Servo servoMotor2; //para controle do servo motor do grupo 2 de tags

servoMotor1.attach(pinoServo1); //informa o pino de controle do servo 1
servoMotor1.write(0); //inicia colocando o servo motor 2 na posição zero graus
servoMotor2.attach(pinoServo2); //informa o pino de controle do servo 2
servoMotor2.write(0); //inicia colocando o servo motor 2 na posição zero graus

if(testaTagGrupo1()){ //se for uma tag válida o valor retornado pela função será verdadeiro
    servoMotor1.write(90); //abre a cancela - aqui poderia ligar um led, emitir um som, etc ...
    delay(5000); //aguarda 5 segundos
    servoMotor1.write(0); //fecha a cancela
}
if(testaTagGrupo2()){ //se for uma tag válida o valor retornado pela função será verdadeiro
    servoMotor2.write(90); //abre a cancela - aqui poderia ligar um led, emitir um som, etc ...
    delay(5000); //aguarda 5 segundos
    servoMotor2.write(0); //fecha a cancela
}

//para facilitar os códigos foram duplicados e os nomes de variáveis ajustados; poderia ter sido passado um parâmetro
boolean testaTagGrupo1(){ //esta função retorna um valor booleano verdadeiro se a tag for conhecida e falso se não for
    for (int tagEmTeste = 0; tagEmTeste < quantidadeTagsGrupo1; tagEmTeste++){ //para cada uma das tags conhecidas
        if (IDdaTagEmTeste[0] == tagsConhecidasGrupo1[tagEmTeste][0] &&
            IDdaTagEmTeste[1] == tagsConhecidasGrupo1[tagEmTeste][1] &&
            IDdaTagEmTeste[2] == tagsConhecidasGrupo1[tagEmTeste][2] &&
            IDdaTagEmTeste[3] == tagsConhecidasGrupo1[tagEmTeste][3] )
        {
            return true; //se todos os bytes forem iguais a tag é conhecida
        }
    }
    return false;
}

boolean testaTagGrupo2(){ //esta função retorna um valor booleano verdadeiro se a tag for conhecida e falso se não for
    for (int tagEmTeste = 0; tagEmTeste < quantidadeTagsGrupo1; tagEmTeste++){ //para cada uma das tags conhecidas
        if (IDdaTagEmTeste[0] == tagsConhecidasGrupo2[tagEmTeste][0] &&
            IDdaTagEmTeste[1] == tagsConhecidasGrupo2[tagEmTeste][1] &&
            IDdaTagEmTeste[2] == tagsConhecidasGrupo2[tagEmTeste][2] &&
            IDdaTagEmTeste[3] == tagsConhecidasGrupo2[tagEmTeste][3] )
        {
            return true; //se todos os bytes forem iguais a tag é conhecida
        }
    }
    return false;
}
```

Note que os códigos foram simplesmente duplicados, para agilidade da programação. Poderiam ter sido escritas funções que recebessem parâmetros (como, por exemplo, qual conjunto testar).

Teste o código e verifique a necessidade de, eventualmente, um terceiro ou um quarto grupo de *tags* (por exemplo, um conjunto para os zeladores, outro para técnicos, outro para professores, outro para alunos, outro para visitantes...).

Teste também outras possibilidades de atuação a partir da detecção das *tags*: ligar um led verde se for conhecida e vermelho se não for; emitir um som para cada caso; enviar uma mensagem ao monitor serial; ligar um *display* e mostrar mensagens (já vimos como, não esqueçam das conexões elétricas necessárias e da biblioteca), dentre outras possibilidades (incluindo juntar várias ações...).

=> **Em resumo:** para termos dois conjuntos de *tags* conhecidas, criamos duas variáveis, uma para cada conjunto; se precisarmos mais, criamos mais variáveis, sempre na forma

```
uint8_t nome_do_conjunto[quantidade necessária][4] = {{0x.., 0x.., 0x.., 0x..},{...}};.
```

Boas práticas.