



# API INDICADOR.dll

---

MANUAL DE SOFTWARE

VERSÃO 2.0

## Sumário

INTRODUÇÃO .....	4
Arquitetura API .....	5
Códigos de Retorno .....	6
Status_Balanca .....	6
Unidade_Medida .....	6
Status_Bateria .....	6
Nivel_Bateria .....	6
Status_Protocolo .....	6
Is_Open .....	7
Response .....	7
Tara_Response .....	7
Zero_Response .....	7
Imprime_Response .....	7
Funções Básicas .....	8
openSerial .....	8
openTcp .....	9
isOpen .....	9
close .....	9
zerar .....	10
tarar .....	10
tararManual .....	10
imprimir_calcularMedia .....	11
perguntarPeso .....	11
getPesoLiquido .....	11
getPesoBruto .....	12
getTara .....	12
getPrecisaoDecimal .....	12
getStatusBalanca .....	13
getUnidadeMedida .....	13
isModoContadoraDePecas .....	13
getProtocolo .....	14
getStatusBateriaIndicador .....	14
getNivelBateriaIndicador .....	14
getNivelBateriaTransmissorTx1 .....	15

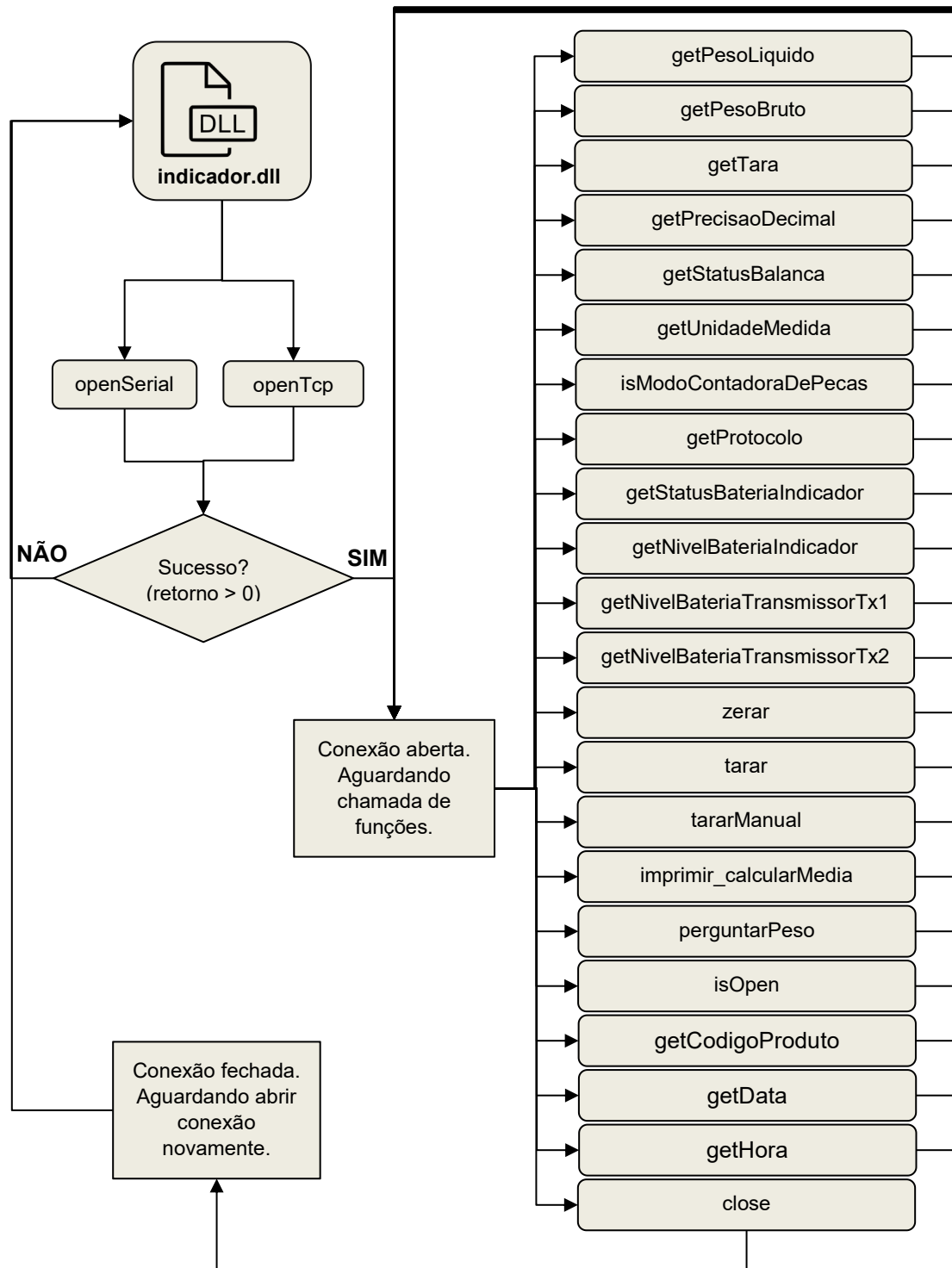
getNivelBateriaTransmissorTx2 .....	15
getCodigoProduto .....	15
getData .....	16
getHora .....	16
Funções Avançadas .....	17
ERF_tabelaProduto .....	17
cadastrarProduto .....	18
alterarProduto .....	18
deletarProduto .....	19
Funções de Tabela de Produto EXC .....	19
getTabelaProduto .....	20
deletarTabelaProduto .....	21
getUsuarioById .....	21
setUsuario .....	22
setConfiguracaoIndicador .....	23
getConfiguracaoIndicador .....	25
saveConfiguracaoIndicador .....	25
gerarRelatorio .....	26
freeStringMemory .....	28

## INTRODUÇÃO

A biblioteca dinâmica (Dynamic Link Library - DLL) é um recurso fundamental para os desenvolvedores, agindo como uma API (Interface de Programação de Aplicações) que simplifica a implementação e o acesso aos seus recursos. A biblioteca indicador.dll, em particular, tem como principal propósito fornecer uma interface simplificada para interação com os terminais/indicadores da Líder Balanças, permitindo aos desenvolvedores de diferentes projetos e linguagens de programação utilizar esses recursos de forma eficiente.

Este documento oferece uma visão detalhada e prática sobre como utilizar as funções contidas na biblioteca dinâmica, acompanhadas de exemplos, facilitando a integração e utilização em seus projetos.

## Arquitetura API



## Códigos de Retorno

Os valores de retorno das funções que serão apresentadas no próximo tópico possuem uma padronização que possibilita a interpretação dos mesmos, segue abaixo a padronização dos retornos adotada:

### Status\_Balanca

STATUS_BALANCA_PESO_EXCESSO_POSITIVO	= -3
STATUS_BALANCA_PESO_EXCESSO_NEGATIVO	= -2
STATUS_BALANCA_DESCONHECIDO	= -1
STATUS_BALANCA_ESTAVEL	= 0
STATUS_BALANCA_INSTAVEL	= 1
STATUS_BALANCA_VALOR_PICO	= 2
STATUS_BALANCA_PESO_MEDIO	= 3
STATUS_BALANCA_AGUARDANDO	= 4
STATUS_BALANCA_PERDA_COMUNICACAO	= 5
STATUS_BALANCA_IMPRIME	= 6

### Unidade\_Medida

UNIDADE_MEDIDA_DESCONHECIDO	= -1
UNIDADE_MEDIDA_KN	= 0
UNIDADE_MEDIDA_LB	= 1
UNIDADE_MEDIDA_KG	= 2
UNIDADE_MEDIDA_N	= 3
UNIDADE_MEDIDA_OZ	= 4

### Status\_Bateria

STATUS_BATERIA_DESCONHECIDO	= -1
STATUS_BATERIA_CARREGANDO	= 0
STATUS_BATERIA_EM_BATERIA	= 1
STATUS_BATERIA_CARGA_COMPLETA	= 2

### Nivel\_Bateria

NIVEL_BATERIA_DESCONHECIDO	= -1
NIVEL_BATERIA_BAIXA	= 0
NIVEL_BATERIA_REGULAR	= 1
NIVEL_BATERIA_BOA	= 2
NIVEL_BATERIA_COMPLETA	= 3

### Status\_Protocolo

PROTOCOLO_DESCONHECIDO	= -1
PROTOCOLO_LIDER_1	= 0
PROTOCOLO_LIDER_2	= 1
PROTOCOLO_LIDER_3	= 2
PROTOCOLO_LIDER_4	= 3

PROTOCOLO_LIDER_10	= 10
--------------------	------

## Is\_Open

TRUE_TCP	= 2	// COMUNICAÇÃO TCP ABERTA
TRUE_SERIAL	= 1	// COMUNICAÇÃO SERIAL ABERTA
FALSE	= 0	// COMUNICAÇÃO FECHADA

## Response

// RESPOSTAS GERAIS		
OK	= 1	
ERRO	= 0	
ERRO_TIMEOUT	= -1	//INDICADOR NÃO RESPONDE
ERRO_INVALID_ARG	= -2	//ARGUMENTO INVÁLIDO
ERRO_MEMORY	= -3	//ERRO NA ALOCAÇÃO DE MEMÓRIA
ERRO_COMUN_OPEN	= -4	//ERRO, COMUNICAÇÃO ABERTA IMPEDINDO
ERRO_COMUN_CLOSED	= -5	//ERRO, COMUNICAÇÃO FECHADA IMPEDINDO

## Tara\_Response

// RESPOSTAS ESPECÍFICAS PARA TARA		
TARA_ACK_0	= 100	// OK, TARA SEMI AUTOMÁTICA
TARA_ACK_1	= 101	// OK, LIMPEZA MANUAL DE TARA
TARA_ACK_2	= 102	// OK, TARA SUCESSIVA
TARA_ACK_3	= 103	// OK, LIMPEZA DE TARA
TARA_ACK_4	= 104	// OK, TARA MANUAL
TARA_NOACK_0	= -100	// ERRO, TARA OU PESO > CAPACIDADE
TARA_NOACK_1	= -101	// ERRO, JÁ POSSUI TARA ATIVA
TARA_NOACK_2	= -102	// ERRO, TARA AUTOMÁTICA ATIVA
TARA_NOACK_3	= -103	// ERRO, PESO > 0 IMPEDE TARA MANUAL
TARA_NOACK_4	= -104	// ERRO, MODO PICO NÃO POSSUI TARA
TARA_NOACK_5	= -105	// ERRO, PESO NULO, SOMENTE TARA MANUAL
TARA_NOACK_6	= -106	// ERRO, TARA SUCESSIVA ATIVA, SOMENTE
		// SOLICITAÇÃO DE TARA COM PESO
TARA_NOACK_7	= -107	// ERRO, PESO NEGATIVO

## Zero\_Response

// RESPOSTAS ESPECÍFICAS PARA ZERO		
ZERO_ACK_0	= 200	// SOLICITAÇÃO DE ZERO ACEITA
ZERO_NOACK_0	= -200	// SOLICITAÇÃO DE ZERO NEGADA

## Imprime\_Response

// RESPOSTAS ESPECÍFICAS PARA IMPRIME		
IMPRIME_ACK_1	= 301	// CÁLCULO DE MÉDIA ACEITO
IMPRIME_ACK_0	= 300	// IMPRESSÃO ACEITA
IMPRIME_NOACK_0	= -300	// IMPRESSÃO/MÉDIA NEGADA

## Funções Básicas

### openSerial

Abre a porta serial.

Obs.: abrir uma conexão serial com um código de retorno “OK” não significa necessariamente que a conexão foi estabelecida corretamente, sempre verifique se os parâmetros conferem com as configurações estabelecidas no indicador.

#### Sintaxe

```
int openSerial(  
    const char*    port,  
    int            baudRate,  
    int            dataBits,  
    int            stopBits,  
    int            parity  
);
```

#### Parâmetros

##### **const char\* port**

Nome da porta serial.

Exemplo : “COM1”, “COM2”, “COM4”

##### **int baudRate**

BaudRate da comunicação:

2400

4800

9600

14400

19200

38400

57600

115200

##### **int dataBits**

Quantidade de data bits:

7 ou 8

##### **int stopBits**

Quantidade de stop bits

1 ou 2

##### **int parity**

Paridade:

0 (None)

1 (Odd - ímpar)

2 (Even - par)



### Valor de Retorno

Retorna um código de retorno Response.

---

## openTcp

Abre a comunicação de rede TCP.

### Sintaxe

```
int openTcp(  
    const char*    ip,  
    int            port  
);
```

### Parâmetros

<b>const char* ip</b>
Endereço IP do indicador.
Exemplo : "10.20.0.200"

<b>int port</b>
Número da porta TCP do indicador.
Exemplo: 502

### Valor de Retorno

Retorna um código de retorno Response.

---

## isOpen

Verifica se a conexão com o indicador está aberta.

### Sintaxe

```
int isOpen( );
```

### Valor de Retorno

Retorna um código de retorno Is\_Open.

---

## close

Fecha uma comunicação que foi aberta, seja ela serial ou TCP.

### Sintaxe

```
int close( );
```

#### Valor de Retorno

Retorna um código de retorno Response.

### zerar

Executa a função de zerar o indicador.

#### Sintaxe

```
int zerar( );
```

#### Valor de Retorno

Retorna um código de retorno Response ou Zero\_Response.

### tarar

Executa a função de tarar o indicador.

#### Sintaxe

```
int tarar( );
```

#### Valor de Retorno

Retorna um código de retorno Response ou Tara\_Response.

### tararManual

Executa a função de tarar manualmente o indicador quando o valor do peso for igual a zero.

#### Sintaxe

```
int tararManual( int tara );
```

#### Parâmetros

**int tara**

Valor da tara sem o separador de casa decimal (“,” ou “.”).

Exemplo: 500

#### Valor de Retorno

Retorna um código de retorno Response ou Tara\_Response.

---

## **imprimir\_calcularMedia**

Executa a função de imprimir ou calcular a média do peso (caso F110 = 2) do indicador.

### **Sintaxe**

```
int imprimir_calcularMedia( );
```

### **Valor de Retorno**

Retorna um código de retorno Response ou Imprime\_Response.

---

## **perguntarPeso**

Solicita o peso do indicador, caso o modo de transmissão esteja configurado como sob demanda (F303 = 0).

### **Sintaxe**

```
int perguntarPeso( );
```

### **Valor de Retorno**

Retorna um código de retorno Response.

---

## **getPesoLiquido**

Obtém o valor do peso líquido.

Obs.: quando a função modo contadora de peças é ativada o valor retornado por esta função será a quantidade de peças(a função isModoContadoraDePecas pode ser utilizada em conjunto com esta).

### **Sintaxe**

```
double getPesoLiquido( );
```

### **Valor de Retorno**

Retorna um valor do tipo *double*.

## getPesoBruto

Obtém o valor do peso bruto.

Obs.: disponível somente quando F300 assume o valor 1, 3 ou 10 (protocolos Líder 2, 4 ou 10).

### Sintaxe

```
double getPesoBruto( );
```

### Valor de Retorno

Retorna um valor do tipo *double*.

## getTara

Obtém o valor da tara do indicador.

Obs.: disponível somente quando F300 assume o valor 1, 3 ou 10 (protocolos Líder 2, 4 ou 10).

### Sintaxe

```
double getTara( );
```

### Valor de Retorno

Retorna um valor do tipo *double*.

## getPrecisaoDecimal

Obtém a precisão decimal dos valores dos pesos apresentados em [getPesoLiquido](#) (precisão de peso líquido somente se [isModoContadoraDePecas](#) é 0, *FALSE*), [getPesoBruto](#) e [getTara](#).

### Sintaxe

```
int getPrecisaoDecimal( );
```

### Valor de Retorno

Retorna um valor do tipo *int*.

### Exemplo de uso:

```
double peso = getPesoBruto();           // peso = 100.2
int precisao = getPrecisaoDecimal();    // precisao = 3
```

```
// Como o peso possui precisão de 3 casas decimais
// peso na verdade é 100.200
// Ou seja, apesar de terem sido ocultadas 2 casas decimais
// os valores de zero delas existem.
```

Como um valor de tipo *double* possui um número indefinido de casas decimais, *getPrecisaoDecimal* serve para definir quantas delas realmente são relevantes. No exemplo acima, o valor de **precisao** serve como um valor auxiliar para **peso**, sem o valor de **precisao** podemos interpretar o valor de **peso** como 100.2, sem a informação de precisão deste valor. Porém, como se sabe que o valor do peso possui 3 casas de precisão, interpreta-se o valor como 100.200.

---

## getStatusBalanca

Obtém o valor do status da pesagem da balança.

### Sintaxe

```
int getStatusBalanca( );
```

### Valor de Retorno

Retorna um código de retorno Status\_Balanca.

---

## getUnidadeMedida

Obtém a unidade de medida do indicador por meio do protocolo(disponível somente as unidades de medida kN, lb e kg).

Obs.: disponível somente quando F300 assume o valor 2 e 3 (protocolos Líder 3 e 4) e quando F312 (envio de unidade de medida) é igual a 1, ligado.

### Sintaxe

```
int getUnidadeMedida( );
```

### Valor de Retorno

Retorna um código de retorno Unidade\_Medida.

---

## isModoContadoraDePecas

Verifica se o indicador está com a contadora de peças ativada, se sim, o valor do peso líquido é a quantidade de peças contadas.

### Sintaxe

```
int isModoContadoraDePecas( );
```

### Valor de Retorno

Retorna um valor *inteiro*, 1 para *TRUE* e 0 para *FALSE* .

---

## getProtocolo

Obtém o protocolo no qual o indicador está operando.

Obs.: a biblioteca só é capaz de identificar os protocolos Líder 1, Líder 2, Líder 3, Líder 4 e Líder 10 nesta função.

### Sintaxe

```
int getProtocolo( );
```

### Valor de Retorno

Retorna um código de retorno Status\_Protocolo.

---

## getStatusBateriaIndicador

Obtém o status da bateria do indicador.

Obs.: disponível somente quando F300 assume o valor 2 e 3 (protocolos Líder 3 e 4) e quando F310 (protocolos com nível de bateria do indicador) é igual a 1, ligado.

### Sintaxe

```
int getStatusBateriaIndicador( );
```

### Valor de Retorno

Retorna um código de retorno Status\_Bateria.

---

## getNivelBateriaIndicador

Obtém o nível da bateria do indicador.

Obs.: disponível somente quando F300 assume o valor 2 e 3 (protocolos Líder 3 e 4) e quando F310 (protocolos com nível de bateria do indicador) é igual a 1, ligado.

### Sintaxe

```
int getNivelBateriaIndicador( );
```

#### Valor de Retorno

Retorna um código de retorno Nivel\_Bateria.

---

### getNivelBateriaTransmissorTx1

Obtém o nível da bateria do transmissor Tx1.

Obs.: disponível somente quando F300 assume o valor 2 e 3 (protocolos Líder 3 e 4) e quando F311 (protocolos com nível de bateria de transmissor) é igual a 1, ligado.

#### Sintaxe

```
int getNivelBateriaTransmissorTx1( );
```

#### Valor de Retorno

Retorna um código de retorno Nivel\_Bateria.

---

### getNivelBateriaTransmissorTx2

Obtém o nível da bateria do transmissor Tx2.

Obs.: disponível somente quando F300 assume o valor 2 e 3 (protocolos Líder 3 e 4) e quando F311 (protocolos com nível de bateria de transmissor) é igual a 1, ligado.

#### Sintaxe

```
int getNivelBateriaTransmissorTx2( );
```

#### Valor de Retorno

Retorna um código de retorno Nivel\_Bateria.

---

### getCodigoProduto

Obtém o código de produto selecionado no indicador, ou seja, o código do produto que está sendo pesado.

Obs.: disponível somente quando F300 assume o valor 10 (protocolos Líder 10).

#### Sintaxe

```
const char* getCodigoProduto( );
```

### Valor de Retorno

Retorna uma sequência de caracteres (*string*) de tamanho máximo 15, incluindo o caractere de fim de string '\0', contendo o código do produto.

---

### getData

Obtém a data do indicador.

Obs.: disponível somente quando F300 assume o valor 10 (protocolos Líder 10).

#### Sintaxe

```
const char* getData( );
```

### Valor de Retorno

Retorna uma sequência de caracteres (*string*) no formato “dd/MM/yyyy”, contendo a data do indicador.

---

### getHora

Obtém a hora do indicador.

Obs.: disponível somente quando F300 assume o valor 10 (protocolos Líder 10).

#### Sintaxe

```
const char* getHora( );
```

### Valor de Retorno

Retorna uma sequência de caracteres (*string*) no formato “HH:mm:ss”, contendo a hora do indicador.



## Funções Avançadas

Estas funções são consideradas avançadas devido à complexidade de suas operações e à necessidade de gerenciamento cuidadoso de recursos, como memória e dados da tabela de produtos. Certifique-se de entender completamente o comportamento e os requisitos de cada função antes de utilizá-las em sua aplicação.

### ERF\_tabelaProduto

No gerenciamento da tabela de produtos do indicador, os novos produtos são armazenados de forma sequencial na memória, o que significa que o sistema procura por espaços consecutivos para alocá-los. Isso é vantajoso em termos de desempenho, pois evita a necessidade de procurar por espaços livres em toda a memória a cada novo cadastro. No entanto, quando um produto é removido da tabela, pode surgir um problema de fragmentação da memória. Isso ocorre porque os espaços deixados pelos produtos removidos podem se tornar inutilizáveis para futuros cadastros, já que o sistema continua procurando por espaços consecutivos a partir do último ponto acessado. Para resolver esse problema, é necessário reiniciar a busca por espaços livres na memória, utilizando a função **ERF\_tabelaProduto** que permite que o sistema identifique e utilize as lacunas deixadas pelas remoções anteriores. Isso ajuda a evitar desperdício de espaço e a manter a eficiência do sistema de gerenciamento de dados.

#### Sintaxe

```
int ERF_tabelaProduto( );
```

#### Valor de Retorno

Retorna um código de retorno *Response*.

#### Exemplo de uso

A função pode ser utilizada em duas situações:

Utilização antes de um ou mais cadastros consecutivos(menos eficiente e fácil utilizar):

```
if (ERF_tabelaProduto( ) > 0)
{
    cadastrarProduto("1234", "TESTE"); // 1 produto
    // ou mais em sequência
    // cadastrarProduto("5678", "TESTE2");
    // cadastrarProduto("9101", "TESTE3");
}
```

Utilização após a remoção de produtos da tabela(mais eficiente e difícil utilizar):

```
if (deletarProduto("1234") > 0) // Remoção de um produto
{
    ERF_tabelaProduto( );
}
if (deletarTabelaProduto( ) > 0) // Limpeza da tabela
{
    ERF_tabelaProduto( );
}
```

## cadastrarProduto

Esta função cadastra um novo produto na tabela do indicador.

### Sintaxe

```
int cadastrarProduto(  
    const char*    codigo,  
    const char*    descricao  
);
```

### Parâmetros

#### **const char\* codigo**

Código do produto de até 14 números.

Exemplo : “12345678901234”

#### **const char\* descricao**

Descrição do produto de até 21 caracteres.

Exemplo: “DESCRICAO EXEMPLO”

### Valor de Retorno

Retorna um código de retorno *Response*.

## alterarProduto

Esta função altera a descrição de um produto cadastrado na tabela do indicador.

### Sintaxe

```
int alterarProduto(  
    const char*    codigo,  
    const char*    descricao  
);
```

### Parâmetros

#### **const char\* codigo**

Código do produto cadastrado de até 14 números.

Exemplo : “12345678901234”

#### **const char\* descricao**

Nova descrição do produto de até 21 caracteres.

Exemplo: “NOVA DESCRICAO”

### Valor de Retorno

Retorna um código de retorno *Response*.

## deletarProduto

Esta função remove um produto cadastrado na tabela do indicador.

### Sintaxe

```
int deletarProduto(const char* codigo);
```

### Parâmetros

**const char\* codigo**

Código do produto já cadastrado, de até 14 números, que será excluído.

Exemplo : “12345678901234”

### Valor de Retorno

Retorna um código de retorno *Response*.

## Funções de Tabela de Produto EXC

As funções EXC servem para iniciar uma nova tabela de produtos no indicador do zero. Diferente das funções de manipulação da tabela de produtos já apresentadas, estas sobrescrevem todos os cadastros, se houver. A ideia deste conjunto de funções é oferecer uma opção de importar para o indicador uma tabela já existente em outro arquivo externo.

### Sintaxe

Função para iniciar a nova tabela:

```
int EXC_iniciarTabelaProduto( );
```

Função para cadastrar um novo produto na nova tabela:

```
int EXC_cadastrarProduto(  
    const char* codigo,  
    const char* descricao  
);
```

Função para finalizar a nova tabela:

```
int EXC_finalizarTabelaProduto(int quantidadeProdutosTabelaEXC);
```

### Parâmetros

**const char\* codigo**

Código do produto de até 14 números.

Exemplo : “123456”

**const char\* descricao**

Descrição do produto de até 21 caracteres.

Exemplo: “DESCRICAO EXC”

#### int quantidadeProdutosTabelaEXC

Quantidade de produtos que foram cadastrados na nova tabela EXC, sem ultrapassar a quantidade máxima permitida (128 produtos).

Exemplo: 100

### Valor de Retorno

Todas as três funções, *EXC\_iniciarTabelaProduto*, *EXC\_cadastrarProduto* e *EXC\_finalizarTabelaProduto*, retornam um código de retorno Response.

### Exemplo de uso

Para iniciar uma nova tabela no indicador, basta utilizar uma combinação das 3 funções:

```
if ( EXC_iniciarTabelaProduto() > 0 ) // INICIO DA NOVA TABELA
{
    // CADASTRO DE 4 PRODUTOS
    EXC_cadastrarProduto("1", "Produto1");
    EXC_cadastrarProduto("2", "Produto2");
    EXC_cadastrarProduto("3", "Produto3");
    EXC_cadastrarProduto("4", "Produto4");
    // FINALIZAÇÃO DA TABELA
    if ( EXC_finalizarTabelaProduto(4) )
        // UM INICIO EXC SEMPRE DEVE TER UM FIM
} else
{
    // ERRO AO INICIAR NOVA TABELA
}
```

## getTabelaProduto

Esta função obtém a tabela de produtos do indicador.

### Sintaxe

```
const char* getTabelaProduto( );
```

### Valor de Retorno

Retorna uma cadeia de caracteres(*string*) contendo todos os produtos da tabela do indicador ou uma mensagem de erro iniciada por “#ERRO: ” e terminada por um código de retorno Response.

A *string* com os produtos da tabela é composta por uma linha no início, iniciada com “#ACKRWCP,”, com a quantidade de produtos da tabela e as próximas linhas que se seguem, iniciadas em “#RWCP,”, trazem todos os produtos da tabela do indicador:

Seja *N*, a quantidade de produtos da tabela

```
“#ACKRWCP,N\n”
```

```
#RWCP,codigo1,descricao1\n
#RWCP,codigoN-1,descricaoN-1\n
#RWCP,codigoN,descricaoN
```

### Exemplo de uso

No exemplo a seguir, suporemos que a tabela do indicador terá 4 produtos, os mesmos cadastrados no exemplo de *Funções de Tabela de Produto EXC*:

```
char* listaProdutos = getTabelaProduto();
// Se não houver erro, o valor esperado da string listaProdutos é:
// "#ACKRWCP,4\n
// #RWCP,1,Produto1\n
// #RWCP,2,Produto2\n
// #RWCP,3,Produto3\n
// #RWCP,4,Produto4"
// Mas caso haja algum erro, a string retornada será:
// "#ERRO: {Response}", por exemplo "#ERRO: -1", erro de timeout
```

## deletarTabelaProduto

Esta função deleta ou limpa toda a tabela de produtos do indicador, ou seja, todos os cadastros de produtos contidos no indicador serão removidos.

### Sintaxe

```
int deletarTabelaProduto( );
```

### Valor de Retorno

Retorna um código de retorno *Response*.

## getUsuarioById

Esta função obtém o nome de um usuário salvo no indicador com o número de *id* informado. Caso o *id* informado não possua um usuário salvo, a função irá retornar um nome vazio.

### Sintaxe

```
const char* getUsuarioById(int id);
```

### Parâmetros

<b>int id</b>
Um número inteiro identificador de um usuário salvo no indicador (verifique o intervalo de numeração disponível no indicador, normalmente, de 1 a 50).

**Exemplo : 12****Valor de Retorno**

Retorna uma cadeia de caracteres(*string*) contendo o nome do usuário correspondente ao número identificador informado ou uma mensagem de erro iniciada por “#ERRO: ” e terminada por um código de retorno *Response*.

A *string* com o nome do usuário é iniciada com “#F113: ”, concatenada com o nome do usuário:

```
“#F113: nomeUsuario”
```

**Exemplo de uso**

No exemplo a seguir, suporemos que o indicador possui 2 usuários, “FULANO” e “CICLANO”, salvos com o identificador 1 e 2, respectivamente:

```
char* usuario1 = getUsuarioById(1);
char* usuario2 = getUsuarioById(2);
char* usuario3 = getUsuarioById(3);

// Se não houver erro, espera-se que as 3 strings sejam:
// usuario1 = “#F113: FULANO”
// usuario2 = “#F113: CICLANO”
// usuario3 = “#F113: ”, não há nome cadastrado no usuario de id = 3

// Mas caso haja algum erro, as strings retornadas poderão ser:
// “#ERRO: {Response}”, por exemplo “#ERRO: -1”, erro de timeout
```

---

**setUsuario**

Esta função altera os dados (nome e senha) de um usuário determinado pelo *id*, mediante a informação da senha do menu do indicador. Caso a senha do menu informada estiver incorreta, o comando no indicador será ignorado e a mensagem de erro *TIMEOUT* será retornada.

**Sintaxe**

```
const char* setUsuario(
    const char* senhaMenu,
    int         idUsuario,
    const char* senhaUsuario,
    const char* nomeUsuario
);
```

**Parâmetros****const char\* senhaMenu**

String contendo a senha numérica do menu do Indicador.  
(no máximo 6 números)

Exemplo : “1234”

#### **int idUsuario**

Um número inteiro identificador de um usuário salvo no indicador (verifique o intervalo de numeração disponível no indicador, normalmente, de 1 a 50).

Exemplo : 12

#### **const char\* senhaUsuario**

String contendo a nova senha numérica do usuário.  
(no máximo 6 números)

Exemplo : “654321”

#### **const char\* nomeUsuario**

String contendo o novo nome do usuário.  
(no máximo 15 caracteres)

Exemplo : “FULANO”

### **Valor de Retorno**

Retorna uma cadeia de caracteres(*string*) contendo o nome do usuário correspondente ao número identificador informado ou uma mensagem de erro iniciada por “#ERRO: ” e terminada por um código de retorno *Response*.

A *string* com o nome do usuário é iniciada com “#F113: ”, concatenada com o nome do usuário:

“#F113: **nomeUsuario**”

### **Exemplo de uso**

No exemplo a seguir, iremos supor que a senha do menu do indicador é “1234” e utilizaremos o mesmo contexto do exemplo da função *getUsuarioById*:

```
char* usuario1 = setUsuario("1234", 1, "0000", "FULANA");
char* usuario2 = setUsuario("0000", 2, "4321", "CICLANO");
char* usuario3 = setUsuario("1234", 3, "1234", "BELTRANO");

// Espera-se que as 2 strings sejam:
// usuario1 = "#F113: FULANA", o nome de usuário 1 foi alterado
// usuario2 = "#ERRO: -1", a senha do menu incorreta, comando ignorado
// usuario3 = "#F113: BELTRANO", agora usuário 3 possui um nome

// Mas caso haja algum outro erro, as strings retornadas poderão ser:
// "#ERRO: {Response}", por exemplo "#ERRO: -1", erro de timeout
```

## **setConfiguracaoIndicador**

Esta função altera o valor da configuração do indicador da função desejada (consultar manual do indicador para saber mais sobre as funções de configuração).

## Sintaxe

```
const char* setConfiguracaoIndicador(  
    int      funcao,  
    const char* input  
);
```

## Parâmetros

### int funcao

Um número inteiro identificando a função de configuração desejada.

Exemplo : 303 (função de frequência de transmissão)

### const char\* input

Uma string contendo o novo valor da configuração.

Exemplo :

“3” (valor que pode ser usado com funcao = 300, alterando o protocolo)

Ou

“NOVA EMPRESA” (valor de funcao = 212, alterando nome da empresa)

Ou

“01.01.15” (valor de funcao = 500 ou 501, alterando data ou hora)

...

(consulte o manual do indicador para saber mais sobre as funções e suas entradas de valores)

## Valor de Retorno

Retorna uma cadeia de caracteres(*string*) contendo o valor da configuração do indicador correspondente ao número da função ou uma mensagem de erro iniciada por “#ERRO: ” e terminada por um código de retorno *Response*.

A *string* com o valor da configuração é iniciada com “#F**funcao**: ”, concatenada com o valor da configuração:

```
“#Ffuncao: valorConfiguracao”
```

## Exemplo de uso

No exemplo a seguir, iremos alterar o valor da função 212(nome da empresa) e 303(frequência de transmissão):

```
char* valor212 = setConfiguracaoIndicador(212, “NOVA EMPRESA”);  
char* valor303 = setConfiguracaoIndicador(303, “0”);  
  
// Se não houver erro, espera-se que as 2 strings sejam:  
// valor212 = “#F212: NOVA EMPRESA”  
// valor303 = “#F303: 0” (0 - sob demanda)  
  
// Mas caso haja algum erro, as strings retornadas poderão ser:  
// “#ERRO: {Response}”, por exemplo “#ERRO: -1”, erro de timeout
```



## getConfiguracaoIndicador

Esta função obtém o valor da configuração do indicador, dado a função que se deseja verificar (consultar manual do indicador para saber mais sobre as funções de configuração).

### Sintaxe

```
const char* getConfiguracaoIndicador(int funcao);
```

### Parâmetros

**int funcao**

Um número inteiro identificando a função de configuração desejada.

Exemplo : 303 (função de frequência de transmissão)

### Valor de Retorno

Retorna uma cadeia de caracteres(*string*) contendo o valor da configuração do indicador correspondente ao número da função ou uma mensagem de erro iniciada por “#ERRO:” e terminada por um código de retorno *Response*.

A *string* com o valor da configuração é iniciada com “#*Ffuncao*:”, concatenada com o valor da configuração:

```
“#Ffuncao: valorConfiguracao”
```

### Exemplo de uso

No exemplo a seguir, iremos consultar o valor da função 212(nome da empresa) e 303(frequência de transmissão):

```
char* valor212 = getConfiguracaoIndicador(212);
char* valor303 = getConfiguracaoIndicador(303);

// Se não houver erro, espera-se que as 2 strings sejam:
// valor212 = “#F212: LIDER BALANCAS”
// valor303 = “#F303: 1” (1 - Contínua)

// Mas caso haja algum erro, as strings retornadas poderão ser:
// “#ERRO: {Response}”, por exemplo “#ERRO: -1”, erro de timeout
```

## saveConfiguracaoIndicador

Esta função salva e coloca em vigência todas as configurações feitas pela função *setConfiguracaoIndicador* até o momento.

### Sintaxe

```
int saveConfiguracaoIndicador( );
```

### Valor de Retorno

Retorna um código de retorno *Response*.

## gerarRelatorio

Esta função obtém um relatório do indicador, dado o tipo de relatório solicitado.

### Sintaxe

```
const char* gerarRelatorio(  
    int      tipo,  
    const char* input  
);
```

### Parâmetros

#### int tipo

Um número inteiro, de 0 a 5, identificando o tipo de relatório solicitado:

- 0 (Relatório por número de impressões)
- 1 (Relatório por data)
- 2 (Relatório por código de produto)
- 3 (Relatório por usuário)
- 4 (Relatório por data e código)
- 5 (Relatório por data e usuário)

#### const char\* input

String contendo os parâmetros de entrada para gerar o relatório, dependendo do tipo de relatório escolhido:

**“numImpressoes”** (caso tipo = 0)

Exemplo: “100”

**“dataInicial,dataFinal”** (caso tipo = 1)

Exemplo: “01/01/2000,30/12/2000”

**“codigoProduto”** (caso tipo = 2)

Exemplo: “12345678”

**“usuarioId”** (caso tipo = 3)

Exemplo: “12”

**“dataInicial,dataFinal,codigoProduto”** (caso tipo = 4)

Exemplo: “01/01/2000,30/12/2000,12345678”

**“dataInicial,dataFinal,usuarioId”** (caso tipo = 5)

Exemplo: “01/01/2000,30/12/2000,12”

### Valor de Retorno

Retorna uma cadeia de caracteres(*string*) contendo o relatório correspondente ao tipo e ao parâmetro informados ou uma mensagem de erro iniciada por “#ERRO: ” e terminada por um código de retorno *Response*.

A *string* com relatório é iniciada com “#SNDRL,ACK,”(caso tipo = 0) ou “#SNDDT,ACK,”(caso tipo = 1) ou “#SNDRCP,ACK,”(caso tipo = 2) ou

“#SNDRL,ACK, nomeEmpresa\n” ou “#SNDDT,ACK, nomeEmpresa\n” ou “#SNDDTUSR,ACK, nomeEmpresa\n” (caso tipo = 5) , concatenada com o nome da empresa, as linhas seguintes são constituídas de linhas de pesagens, iniciadas por “#SPS,” , linhas de contagem de peças, iniciadas por “#SPC,” , linhas de total de pesagens, iniciadas por “#STLP,” e linhas de total de contagem de peças, iniciadas por “#STLC,”:

```

“#SNDRL,ACK, nomeEmpresa\n” ou
“#SNDDT,ACK, nomeEmpresa\n” ou
“#SNDRC,ACK, nomeEmpresa\n” ou
“#SNDRL,ACK, nomeEmpresa\n” ou
“#SNDDT,ACK, nomeEmpresa\n” ou
“#SNDDTUSR,ACK, nomeEmpresa\n” ou
+
“#SPS, data, hora, codProduto, descrProduto, pesoLiq, tara, sequencia, tipoUser
  r userId, nomeUser, unidMedida, cancelado\n”
+
“#SPC, data, hora, codProduto, descrProduto, pesoLiq, tara, sequencia, quantPe
  cas, pesoMedioPeca, tipoUser userId, nomeUser, unidMedida, cancelado\n”
+
“#STLP, totalPesado\n”
+
“#STLC, totalDePecasContadas\n”
...

```

### Exemplo de uso

No exemplo a seguir, iremos supor que na memória do indicador foram salvas 2 pesagens, 1 total de pesagem, 2 contagens de peças e 1 total de contagem de peças, nessa sequência:

```

char* relatorio = gerarRelatorio(0, "6");

// Espera-se que string retornada seja, algo do tipo:
// “#SNDRL,ACK,LIDER BALANCAS\n
// #SPS,01/01/2000,01:12:01,1,Produto1,1.200,0.000,1,USUARIO 0,,2,0\n
// #SPS,01/01/2000,01:12:02,2,Produto1,1.000,0.200,2,USUARIO 0,,2,0\n
// #STLP,2.200\n
// #SPC,01/01/2000,12:01:03,1,Produto1,1.200,0.000,1,6,0.2000,USUARIO
0,,2,0\n
// #SPC,01/01/2000,12:01:03,1,Produto1,1.200,0.200,2,5,0.2000,USUARIO
0,,2,0\n
// #STLC,11”

// Mas caso haja algum outro erro, as strings retornadas poderão ser:
// “#ERRO: {Response}”, por exemplo “#ERRO: -1”, erro de timeout

```

## freeStringMemory

Todas as funções desta seção de *Funções Avançadas* que possuem um retorno do tipo *const char\**, ou seja, um retorno do tipo *string*, retornam *strings* alocadas dinamicamente na linguagem C. Dessa forma, é necessário haver uma desalocação dessas *strings* em algum momento. A biblioteca por si só, já possui alguns pontos em que ela realiza essa desalocação: quando a função é chamada novamente, a *string* é desalocada para se alocar uma nova, e quando a comunicação é fechada, todos os recursos utilizados são liberados.

Caso a utilização dessas funções que alocam memória dinâmica não seja frequente ou a aplicação desenvolvida cria cópias ou obtém, de alguma forma, somente o valor das *strings* retornadas sem a necessidade da original, pode ser melhor e mais eficiente antecipar a desalocação da memória dinâmica. Para este fim, há a função *freeStringMemory* que desaloca a memória das *strings* dinâmicas utilizadas.

Obs.: utilize a função com cautela para não desalocar uma *string* em utilização e causar um estouro de memória. Caso não seja um problema para a sua aplicação haver memórias alocadas sem utilização por um certo período de tempo, pode-se abdicar o uso desta função.

### Sintaxe

```
void freeStringMemory( );
```