Programando Robôs
Lego NXT usando NXC

Daniele Benedettelli

Título original: **Programming LEGO NXT Robots using NXC (beta 30 or higher)**

Autor: **Danielle Benedettelli**

Revisão: **John Hansen**

Versão 2.2, 7 de junho de 2007.

Tradução: **Rafael Bermudes**

Revisão: **Felipe Nascimento Martins**

NERA – Núcleo de Estudos em Robótica e Automação – http://nera.sr.ifes.edu.br

IFES – Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo

Edição: Janeiro de 2012.

Prefácio da edição brasileira

Este texto foi traduzido para o português com autorização do autor, que também permitiu sua distribuição para utilização educacional. A tradução deste trabalho foi uma iniciativa pessoal, com o objetivo de divulgar um material introdutório sobre robótica móvel e programação a alunos do NERA – Núcleo de Estudos em Robótica e Automação do IFES, e a todos que desejam aprender a programar o NXT.

A tradução deste material foi feita voluntariamente pelo aluno Rafael Bermudes, do curso de Engenharia de Controle e Automação do IFES. Gostaria de registrar meus agradecimentos ao Rafael por sua dedicação e pelo ótimo trabalho.

Agradeço, também, ao Danielle Benedettelli, por ter gentilmente aceitado que fizéssemos a tradução e divulgação de seu texto. A partir de agora, leitores de língua portuguesa interessados em iniciar seus estudos em robótica móvel e, em especial, na programação de robôs NXT da Lego, também podem aproveitar este excelente trabalho.

Felipe Nascimento Martins.

Serra, ES, Brasil. Janeiro de 2012.

Prefácio

Como aconteceu com os bons e velhos Mindstorms RIS, CyberMaster e Spybotics, para conseguir liberar o poder total dos blocos Mindstorms NXT você precisa de um ambiente de programação mais útil que o NXT-G, a linguagem gráfica similar ao National Instruments LabVIEW, que vem junto com o conjunto vendido.

NXC é uma linguagem de programação inventada por John Hansen que foi especialmente designada para os robôs Lego. Se você nunca escreveu um programa antes, não se preocupe. NXC é bem fácil de usar e este tutorial te guiará nos primeiros passos da programação.

Para que seja ainda mais fácil escrever os programas, existe o Bricx Command Center (BricxCC – Centro de Comando Bricx). Esse sistema o ajuda a escrever seus programas, fazer o *download* deles para o robô, começá-los e pará-los, navegar pela memória flash do NXT, converter arquivos de som para usar com os blocos e muito mais. BricxXCC funciona quase como um editor de texto, porém com algumas funções extras. Este tutorial usará o BricxCC (versão 3.3.7.16 ou superior) como ambiente de desenvolvimento integrado (IDE - *Integrated Development Environment*).

Você pode fazer o *download* de graça do programa pelo endereço:
http://bricxcc.sourceforge.net/

BricxCC roda em PCs com Windows (95, 98, ME, NT, 2000, XP, Vista). A linguagem NXC também pode ser utilizada em outras plataformas. Você pode fazer o download do compilador em:
http://bricxcc.sourceforge.net/nxc/

A maior parte desse tutorial deve também ser aplicável a outras plataformas, exceto que você perderá algumas das ferramentas incluídas no BricxCC, como a *color-coding* (que automaticamente troca as cores de palavras especiais da linguagem enquanto você escreve seu código).

Como comentário adicional, minha página da Web está cheia de conteúdo para Lego Mindstorms RCX e NXT, inclusive uma ferramenta de PC para comunicar com o NXT:
http://daniele.benedettelli.com

Agradecimentos

Muitos agradecimentos para John Hansen, cujo trabalho não tem preço!

Sumário

I. Escrevendo o seu primeiro programa

Neste capítulo irei lhe mostrar como escrever um programa extremamente simples. Nós vamos programar um robô para se mover para frente por 4 segundos e depois para trás por mais 4 segundos, e então parar. Nada muito espetacular, mas irá lhe apresentar uma ideia básica de programação e lhe mostrará como é fácil fazer isso. Mas, antes de que possamos escrever um programa, precisamos de um robô.

Construindo um robô

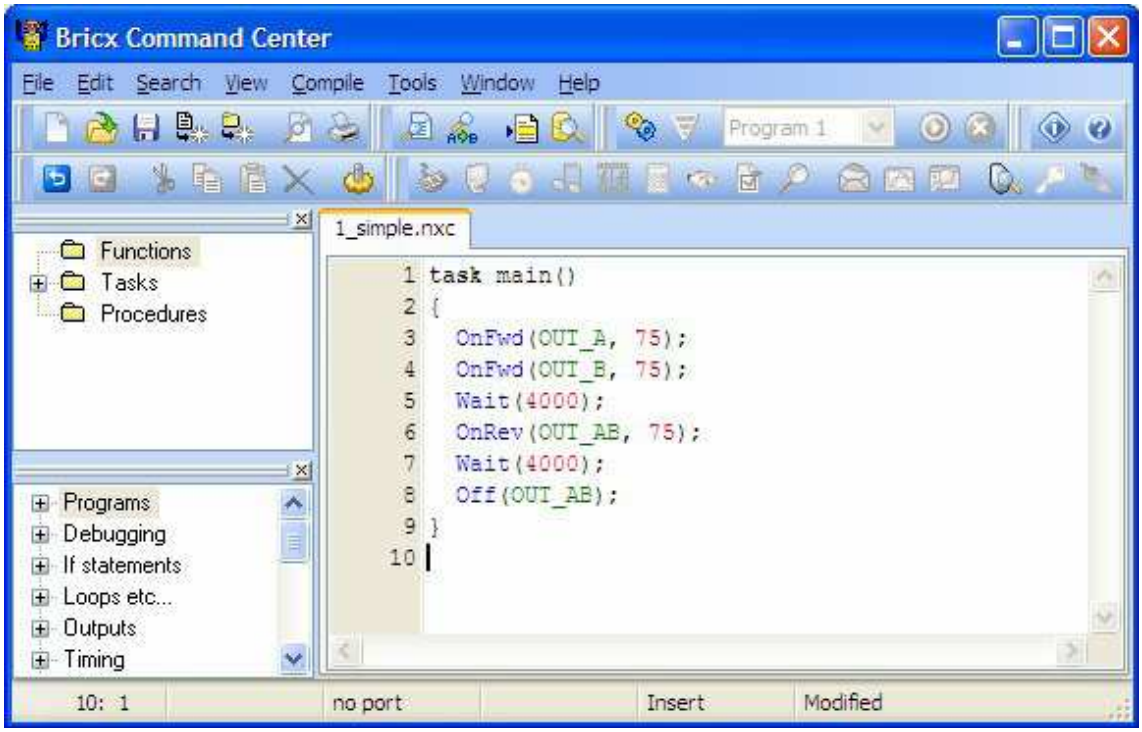
O robô que usaremos durante este tutorial é o Tribot, o primeiro "andarilho" que você é instruído a montar assim que você tira o conjunto NXT da caixa[1]. A única diferença é que você precisa conectar o motor direito na porta A, o motor esquerdo na porta C e o motor da garra na porta B.

Tenha certeza de ter instalado corretamente os *drivers* Fantom Mindstorms NXT que vêm com seu conjunto.

Inicializando o Bricx Command Center

Nós escrevemos nossos programas usando o Bricx Command Center. O inicie clicando duas vezes no ícone BricxCC. (Eu assumo que você já instalou o BricxCC. Caso não tenha feito, faça o *download* dele pelo Web Site (veja no prefácio) e instale o mesmo em qualquer diretório que desejar. O programa irá lhe perguntar onde localizar o robô. Ligue o robô e pressione **OK**. O programa irá (muito provavelmente) automaticamente achar o robô. Agora a interface de usuário aparecerá como mostrado a seguir (sem a aba de texto).

[1] N. do R.: O kit NXT 2.0 não traz instruções de montagem do robô Tribot. Você pode encontrar tais instruções em: http://www.superquest.net/webclass/sboost/ftc_training/gettingstarted/page1/index.html



A interface se parece com um editor de texto tradicional, com o menu usual, botões de abrir e salvar arquivos, editar arquivos etc. Porém, existem também menus especiais para compilação, fazer *download* de programas para o robô e para receber informação do robô. Esses você pode ignorar por agora.

Nós vamos escrever um novo programa, então aperte o botão **New File** para criar uma nova janela vazia.

Escrevendo o programa

Agora digite o seguinte programa:

```
task main()
{
	OnFwd(OUT_A, 75);
	OnFwd(OUT_C, 75);
	Wait(4000);
	OnRev(OUT_AC, 75);
	Wait(4000);
	Off(OUT_AC);
}
```

Poder parecer um pouco complicado de início, então vamos analisá-lo.

Programar em NXC consiste em tarefas. Nosso programa só possui uma tarefa, chamada de `main`. Cada programa precisa ter uma tarefa chamada `main` que é a que será executada pelo robô. Você aprenderá mais sobre tarefas no Capítulo VI. Uma tarefa consiste de um número de comandos, também chamados de *statements*. Existem chaves { } antes e depois dos *statements* para deixar claro que todos eles pertencem a uma respectiva tarefa. Cada *statement* termina com um ponto e vírgula. Dessa forma fica claro onde cada comando termina e onde o próximo comando começa. Então, uma tarefa parecerá geralmente como se segue:

```
task main()
{
	statement1;
	statement2;
	…
}
```

Nosso programa possui seis *statements*. Vamos olhar uma por vez:

```
OnFwd(OUT_A, 75);
```

Este comando manda o robô ligar a saída A, ou seja, o motor conectado à saída chamada A no NXT, para se mover pra frente. O número seguinte atribui a velocidade do motor para 75% da velocidade máxima.

```
OnFwd(OUT_C, 75);
```

Mesmo *statement*, mas agora nós ligamos o motor C. Depois desses dois comandos, ambos os motores estão rodando e o robô se move para frente.

```
Wait(4000);
```

Agora é hora de esperar um pouco. Este *statement* nos diz para esperar por 4 segundos. O argumento, isto é, o número entre parênteses, mostra o número em 1/1000 de segundo: então você pode precisamente dizer o quanto o programa deve esperar. Por 4 segundos (4000ms), o programa fica em espera (*sleep*) enquanto o robô continua a se mover para frente.

```
OnRev(OUT_AC, 75);
```

O robô já se moveu o suficiente, então agora dizemos para ele se mover em direção reversa, ou seja, para trás. Note que podemos acionar ambos os motores usando `OUT_AC` como argumento. Nós poderíamos ter combinado os dois primeiros comandos da mesma forma.

```
Wait(4000);
```

Novamente nós esperamos mais 4 segundos.

```
Off(OUT_AC);
```

E, finalmente, desligamos ambos os motores.

Esse é o programa inteiro. Ele move ambos os motores para frente por 4 segundos, então para trás por 4 segundos e finalmente os desliga.

Você provavelmente notou as cores enquanto digitava o programa. Elas aparecem automaticamente. As cores e estilos usados pelo editor quando ele destaca a sintaxe são customizáveis.

Rodando o programa

Uma vez que você tenha escrito o programa, ele precisa ser compilado (convertido em código binário para que o robô possa entender e executar) e enviado ao robô usando cabo USB ou adaptador Bluetooth (fazer o *download* para o robô).

Aqui você pode ver os botões que lhe permitem (da esquerda para direita) compilar, fazer o *download*, rodar e parar o programa.

Aperte o segundo botão e, assumindo que você não cometeu nenhum erro enquanto digitava o programa, ele será compilado e o *download* será feito. (Caso existam erros no seu programa você será notificado; veja a seguir.)

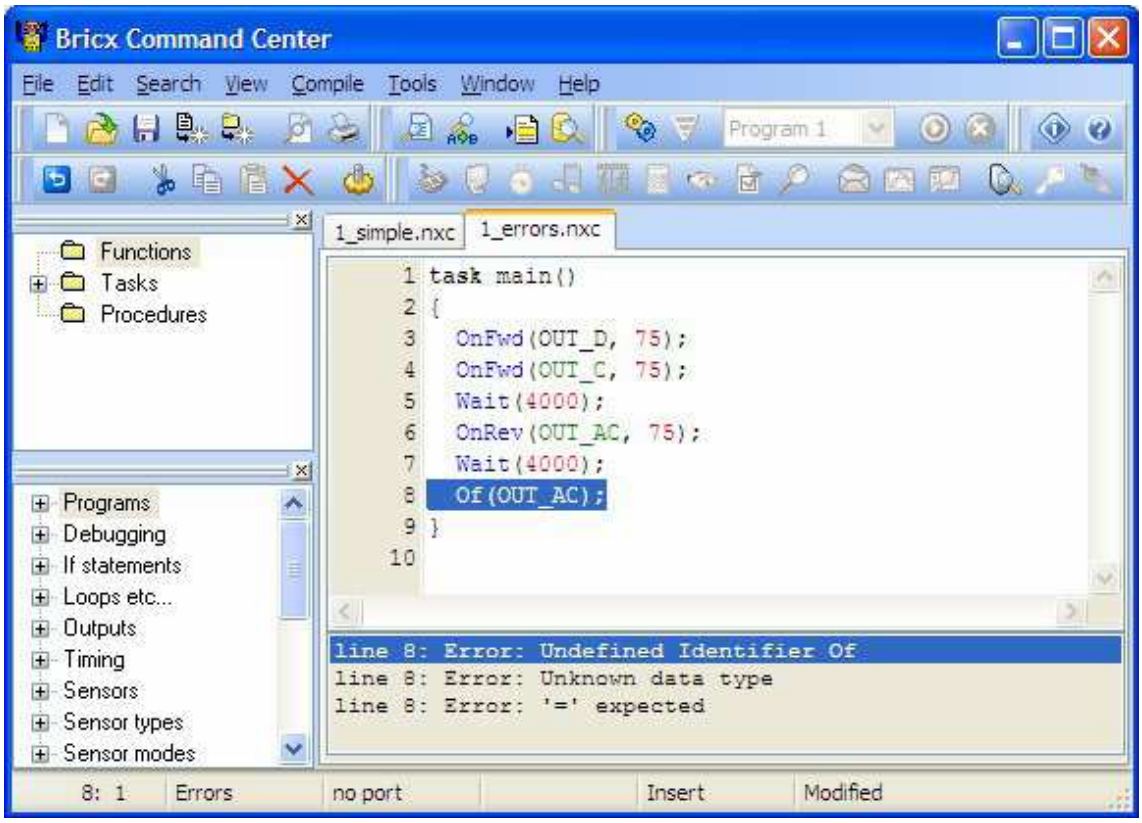
Agora você pode rodar o programa. Para fazer isso, vá para o menu *My Files OnBric*, *Software files*, e rode o programa `1_simple`. Lembre-se: arquivos de *software* do sistema de arquivos NXT possuem o mesmo nome que os arquivos de fonte NXT.

Também, para que o programa rode automaticamente, você pode usar o atalho CTRL+F5 ou, após fazer o *download* do programa, apertar o botão verde *run*.

O robô fez o que você esperava? Caso não tenha feito, veja se os cabos estão devidamente conectados.

Erros no seu programa

Quando está digitando programas existe uma chance razoável que você cometa alguns erros. O compilador nota os erros e os relata na parte de baixo da janela, como na figura a seguir:



Ele automaticamente seleciona o primeiro erro (nós digitamos errado o nome do motor). Quando existem mais erros, você pode clicar nas mensagens de erro e ir até eles. Note que frequentemente erros no início do programa causam erros em outros lugares. Então é melhor corrigir apenas os primeiros erros e então tentar compilar o programa de novo. Note, também, que o destaque da sintaxe ajuda bastante a evitar os erros. Por exemplo, na última linha nós digitamos `Of` em vez de `Off`. Como este é um comando desconhecido, ele não foi destacado.

Existem também erros que não são encontrados pelo compilador. Se nós tivéssemos digitado `OUT_B` o motor errado seria ligado. Caso seu robô apresente comportamento inesperado, muito provavelmente existe algo de errado em seu programa.

Mudando a velocidade

Como você notou, o robô se moveu bem rápido. Para mudar a velocidade apenas mude o segundo parâmetro dentro dos parênteses. A potência é um número entre 0 e 100. 100 é o mais rápido, 0 significa parar (os servomotores do NXT irão manter posição). Aqui está uma nova versão do nosso programa que faz os motores se moverem mais devagar:

```
task main()
{
	OnFwd(OUT_AC, 30);
	Wait(4000);
	OnRev(OUT_AC, 30);
	Wait(4000);
	Off(OUT_AC);
}
```

Sumário

Neste capítulo você escreveu seu primeiro programa em NXC usando BricxCC. Agora você sabe como digitar um programa, como fazer o *download* dele para o robô e como fazer o robô executar o programa. BricxCC pode fazer muito mais coisas. Para descobri-las, leia a documentação que vem com ele. Este tutorial irá principalmente lidar com a linguagem NXC e só mencionar certas ferramentas do BricxCC quando você realmente precisar delas.

Você também aprendeu aspectos importantes da linguagem NXC. Em primeiro lugar, você aprendeu que cada programa possui uma tarefa principal chamada `main` que sempre será executada pelo robô. Aprendeu também os três comandos básicos para motor: `OnFwd()`, `OnRev()` e `Off()`. Por último, aprendeu sobre o comando `Wait()`.

II. Um programa mais interessante

Nosso primeiro programa não foi tão espantoso. Então vamos tentar torná-lo mais interessante. Nós vamos fazer isso em um número de etapas, introduzindo algumas características da nossa linguagem de programação NXC.

Fazendo curvas

Você pode fazer seu robô girar parando ou revertendo a direção de um dos dois motores. Aqui vai um exemplo. Digite o código, faça o *download* para o robô e deixe rodar. Ele deve andar um pouco e então fazer uma curva de 90º à direita.

```
task main()
{
	OnFwd(OUT_AC, 75);
	Wait(800);
	OnRev(OUT_C, 75);
	Wait(360);
	Off(OUT_AC);
}
```

Você poderá ter de testar alguns números diferentes de 360 no segundo `Wait()` para fazer a curva de 90º. Isso depende do tipo de superfície em que o robô está se movendo. Ao invés de mudar o valor no programa é mais fácil usar um nome para esse número. No NXC você pode definir valores constantes como mostrado no programa a seguir.

```
#define MOVE_TIME 1000
#define TURN_TIME 360
task main()
{
	OnFwd(OUT_AC, 75);
	Wait(MOVE_TIME);
	OnRev(OUT_C, 75);
	Wait(TURN_TIME);
	Off(OUT_AC);
}
```

As primeiras duas linhas definem duas constantes. Essas agora podem ser usadas por todo o programa. Definir constantes é bom por duas razões: torna o programa mais legível e é mais fácil de mudar valores. Note que BricxCC dá as instruções `define` sua própria cor. Como veremos no Capítulo VI, você pode definir outras coisas além de constantes.

Comandos de Repetição

Vamos tentar escrever um programa que faz o robô descrever uma trajetória quadrada. Isso significa: Andar para frente, virar 90º, andar para frente de novo, virar 90º etc. Nós poderíamos repetir o pedaço de código acima quatro vezes, mas isso pode ser feito de modo mais fácil usando o comando **`repeat`**.

```
#define MOVE_TIME 500
#define TURN_TIME 500
task main()
{
	repeat(4)
	{
		OnFwd(OUT_AC, 75);
		Wait(MOVE_TIME);
		OnRev(OUT_C, 75);
		Wait(TURN_TIME);
	}
	Off(OUT_AC);
}
```

O número dentro do parênteses do comando **`repeat`** indica quantas vezes o código dentro das chaves deve ser executado. Note que, no programa acima, nós também indentamos[2] os *statements*. Isso não é necessário, mas torna o programa mais legível.

Como exemplo final, vamos fazer o robô descrever a trajetória quadrada 10 vezes. Aqui está o programa:

```
#define MOVE_TIME 1000
#define TURN_TIME 500
task main()
{
	repeat(10)
	{
		repeat(4)
		{
			OnFwd(OUT_AC, 75);
			Wait(MOVE_TIME);
			OnRev(OUT_C, 75);
			Wait(TURN_TIME);
		}
	}
	Off(OUT_AC);
}
```

Agora um comando `repeat` está dentro de outro. Nós chamamos isso de comando `repeat` aninhado. Você pode aninhar quantos comandos `repeat` você quiser. Tome cuidado com as chaves e com a indentação usada no programa. A tarefa começa na primeira chave e termina na última. O primeiro comando `repeat` começa na segunda chave e termina na quinta. O `repeat` aninhado começa na terceira chave e termina na quarta. Como se observa, as chaves sempre vêm em pares e o pedaço de código entre as chaves é indentado.

Adicionando Comentários

Para fazer seu programa ainda mais legível, é bom adicionar alguns comentários. Sempre que você colocar um // em uma linha, o resto dessa linha é ignorado pelo compilador e pode ser usado para comentários. Um comentário muito longo pode ser colocado entre /* e */. Comentários são destacados no BricxCC. O programa inteiro se parecerá como se segue:

[2] N. do R.: Indentação é o ato de organizar os comandos do programa de forma que todos de um mesmo grupo sejam iniciados na mesma coluna. Comandos de um sub-grupo ficam mais à direita.

```
/* 10 SQUARES
Este programa faz o robô se deslocar 10 quadrados
*/

#define MOVE_TIME 500 // Tempo para movimento em linha reta
#define TURN_TIME 360 // Tempo para fazer curva de 90º

task main()
{
	repeat(10) // Faz 10 quadrados
	{
		repeat(4)
		{
			OnFwd(OUT_AC, 75);
			Wait(MOVE_TIME);
			OnRev(OUT_C, 75);
			Wait(TURN_TIME);
		}
	}
	Off(OUT_AC); // Desliga ambos os motores
}
```

Sumário

Neste capítulo você aprendeu a utilizar o comando **`repeat`** e o uso dos comentários. Você também viu a funcionalidade de chaves aninhadas e também o uso de indentação. Com tudo que você já sabe, você pode fazer o robô se locomover por qualquer tipo de caminho. Um bom exercício é praticar e escrever variações dos programas desse capítulo antes de continuar para o próximo.

III. Usando Variáveis

Variáveis constituem um aspecto muito importante de qualquer linguagem de programação. Variáveis são endereços de memória em que nós podemos guardar um valor. Nós podemos usar esse valor em lugares diferentes e podemos alterá-lo. Deixe-nos demonstrar o uso de variáveis com um exemplo.

Deslocamento em espiral

Assuma que nós queremos adaptar o programa acima de forma que o robô se desloque em espiral. Isso pode ser alcançado ao fazer o tempo de *sleep* maior para cada deslocamento em linha reta posterior. Ou seja, nós queremos aumentar o valor de `MOVE_TIME` a cada vez. Mas, como faremos isso? `MOVE_TIME` é uma constante, portanto não pode ser mudado. Nós precisamos de uma variável. Variáveis podem ser facilmente definidas em NXC. Aqui está o programa de espiral.

```
#define TURN_TIME 360
int move_time; // define uma variável
task main()
{
	move_time = 200; // define um valor inicial
	repeat(50)
	{
		OnFwd(OUT_AC, 75);
		Wait(move_time); // usa a variável para esperar
		OnRev(OUT_C, 75);
		Wait(TURN_TIME);
		move_time += 200; // aumenta o valor da variável
	}
	Off(OUT_AC);
}
```

As linhas de interesse estão indicadas com os comentários. Primeiro nós definimos uma variável digitando a palavra-chave **`int`** seguido por um nome de nossa escolha (normalmente usa-se palavras em caixa baixa para nomes de variáveis e em caixa alta para constantes, mas isso não é necessário). O nome precisa começar com uma letra, mas pode conter números e *underscore* (_). Nenhum outro símbolo é permitido (o mesmo se aplica a constantes, nomes de tarefa etc.). A palavra **`int`** é definida para inteiros. Somente números inteiros podem ser guardados nela. Na segunda linha nós designamos o valor de 200 para a variável. A partir de agora, em qualquer lugar que você use a variável, seu valor será 200. Em seguida está o *loop* de repetição no qual usamos a variável para indicar o tempo de *sleep* e, no final do *loop*, aumentamos o valor da variável em 200. Assim, o robô espera na primeira vez 200 ms, na segunda vez 400 ms, na terceira vez 600 ms e assim por diante.

Além de adicionar valor a uma variável nós podemos também multiplicar o valor de uma variável com um número usando `*=`, subtrair usando `-=` e dividir usando `/=` (Note que para divisão o resultado é arredondado para o inteiro mais próximo.). Você também pode adicionar uma variável a outra e escrever expressões mais complicadas. O próximo exemplo não tem nenhum efeito no *hardware* do seu robô, já que não sabemos usar o display do NXT ainda!

```
int aaa;
int bbb,ccc;
int values[];

task main()
{
	aaa = 10;
	bbb = 20 * 5;
	ccc = bbb;
	ccc /= aaa;
	ccc -= 5;
	aaa = 10 * (ccc + 3); // aaa agora é igual a 80
	ArrayInit(values, 0, 10); // Aloca 10 elementos = 0
	values[0] = aaa;
	values[1] = bbb;
	values[2] = aaa*bbb;
	values[3] = ccc;
}
```

Note nas primeiras duas linhas que podemos definir múltiplas variáveis em apenas uma linha. Nós também poderíamos ter combinado todas as três em apenas uma linha. A variável chamada `values` é um vetor (*array*), ou seja, uma variável que possui mais de um número: um *array* pode ser indexado com um número dentro de colchetes. Em NXC, vetores de valores inteiros são declarados como:

```
int nomedavariavel[];
```

Então, essa linha aloca 10 elementos e os inicializa como sendo 0.
`ArrayInit(values, 0, 10);`

Números randômicos

Em todos os programas anteriores nós definimos exatamente o que o robô deveria fazer. Só que as coisas ficam mais interessantes quando o robô faz coisas que não sabemos. Nós queremos alguma aleatoriedade nos movimentos. Em NXC você pode criar números aleatórios. O programa a seguir faz com que o robô se desloque de modo aleatório. Ele constantemente se desloca para frente por uma quantidade de tempo aleatória e então faz uma curva aleatória.

```
int move_time, turn_time;
task main()
{
	while(true)
	{
		move_time = Random(600);
		turn_time = Random(400);
		OnFwd(OUT_AC, 75);
		Wait(move_time);
		OnRev(OUT_A, 75);
		Wait(turn_time);
	}
}
```

Esse programa declara duas variáveis e então as inicializa com números randômicos. `Random(600)` significa um número aleatório entre 0 e 600 (o valor máximo não é incluído no escopo de números retornados). A cada vez que você chamar o comando `Random` os números serão diferentes.

Note que poderíamos evitar o uso de variáveis ao escrever diretamente, por exemplo, `Wait(Random(600))`.

Você pode ver um novo tipo de *loop* aqui. Em vez de usar o comando `repeat` nós usamos **`while(true)`**. O comando `while` repete todos os *statements* abaixo dele enquanto a condição entre parênteses for verdadeira. A palavra especial **`true`** é sempre verdadeira, então as condições entre as chaves se repetirão para sempre (ou até que você aperte o botão cinza escuro do NXT). Você aprenderá mais sobre o comando `while` no capítulo IV.

Sumário

Nesse capítulo você aprendeu o uso de variáveis e de vetores. Você pode declarar outros tipos de dados além de `int`: `short`, `long`, `byte`, `bool` e `string`.

Você também aprendeu como criar números aleatórios, com os quais você pode dar ao robô um comportamento imprevisível. No final vimos como usar o *statement* `while` para criar um *loop* infinito (que é executado para sempre).

IV. Estruturas de controle

Nos capítulos anteriores vimos os comandos `repeat` e `while`. Esses *statements* controlam a forma que outros comandos no programa são executados. Eles são chamados de "estruturas de controle". Nesse capítulo veremos outras estruturas de controle.

O comando if

Algumas vezes você pode querer que uma parte particular do seu programa só seja executada em certas ocasiões. Neste caso, o comando `if` é usado. Deixe-me dar um exemplo. Novamente mudaremos o programa em que viemos trabalhando até agora, mas de um jeito novo. Nós queremos que o robô se desloque em linha reta e, então, faça uma curva à direita ou à esquerda. Para fazer isso precisamos de números aleatórios de novo. Pegaremos um número aleatório que pode ser tanto positivo quanto negativo. Se for menor que 0, viramos à direita; caso contrário, viramos para a esquerda. Aqui vai o programa:

```
#define MOVE_TIME 500
#define TURN_TIME 360
task main()
{
	while(true)
	{
		OnFwd(OUT_AC, 75);
		Wait(MOVE_TIME);
		if (Random() >= 0)
		{
			OnRev(OUT_C, 75);
		}
		else
		{
			OnRev(OUT_A, 75);
		}
		Wait(TURN_TIME);
	}
}
```

O comando **`if`** é um pouco parecido com o comando **`while`** no sentido de que se a condição entre parênteses é verdadeira a parte entre as chaves é executada. Caso contrário, a parte entre as chaves depois da palavra **`else`** é executada. Vamos analisar melhor a condição que usamos: `Random() >= 0`. Isso significa que `Random()` precisa ser maior ou igual a 0 para que a condição seja verdadeira. Você pode comparar valores de maneiras diferentes. Aqui estão os mais importantes:

== igual a[3]
< menor que
<= menor ou igual a
> maior que
>= maior ou igual a
!= não é igual a (ou diferente de)

Você pode combinar condições usando &&, que significa "e", ou ||, que significa "ou". Alguns exemplos de condições:

`true`	sempre verdadeira
`false`	nunca verdadeira
`ttt != 3`	verdadeira quando `tt` não é igual a 3
`(ttt >= 5) && (ttt <= 10)`	verdadeira quando `ttt` estiver entre 5 e 10
`(aaa == 10) || (bbb == 10)`	verdadeira se `aaa` ou `bbb` (ou ambos) forem iguais a 10

[3] N. do R.: Deve-se ter atenção ao fazer a comparação de igualdade, que é representada por dois sinais de "igual" seguidos (==). O uso de apenas um sinal de "igual" (=) significa atribuição de valores, e não deve ser usado em comparação de igualdade!

Note que o comando **`if`** tem duas partes. A parte imediatamente após a condição, que é executada quando a condição é verdadeira, e a parte após o **`else`**, que é executada caso a condição seja falsa. A palavra-chave **`else`** e a parte após a mesma são opcionais. Significa que você pode omiti-las se não há nada a fazer quando a condição é falsa.

O comando do

Aqui vai outra estrutura de controle, o comando **`do`**. Ele tem a seguinte forma:

```
do
{
	statements;
}
while (condição);
```

Os *statements* entre as chaves depois do comando **`do`** serão executadas enquanto a condição for verdadeira. A condição possui a mesma forma do que a que descrevemos acima para o comando **`if`**. Aqui vai um exemplo de um programa. O robô se descola aleatoriamente por 20 segundos e então pára.

```
int move_time, turn_time, total_time;
task main()
{
	total_time = 0;
	do
	{
		move_time = Random(1000);
		turn_time = Random(1000);
		OnFwd(OUT_AC, 75);
		Wait(move_time);
		OnRev(OUT_C, 75);
		Wait(turn_time);
		total_time += move_time;
		total_time += turn_time;
	}
	while (total_time < 20000);
	Off(OUT_AC);
}
```

Note que o comando **`do`** se comporta quase da mesma forma que o comando **`while`**. Porém, no comando **`while`**, a condição é testada antes de se executar os *statements*, enquanto no comando **`do`** a condição só é testada no final. Para o comando **`while`**, talvez os *statements* nem sejam executados, mas para o comando **`do`** eles são executados ao menos uma vez.

Sumário

Neste capítulo nós vimos duas novas estruturas de controle: o comando **`if`** e o comando **`do`**. Juntos com os comandos **`repeat`** e **`while`** eles são *statements* que controlam a forma que o programa é executado. É muito importante que você entenda o que cada um faz, portanto, é melhor que treine por si só mais alguns exemplos antes de continuar.

V. Sensores

É claro que você pode ligar sensores ao NXT para que o robô reaja a estímulos externos. Antes que eu possa mostrar como fazer isso, precisamos mudar o robô um pouco adicionando um sensor de toque. Como antes, siga as instruções do Tribot para montar o pára-choque dianteiro.

Conecte o sensor de toque à porta de entrada 1 do NXT.

Esperando pelo sensor

Vamos começar com um programa muito simples no qual o robô se desloca para frente até bater em alguma coisa. Aqui vai:

```
task main()
{
	SetSensor(IN_1,SENSOR_TOUCH);
	OnFwd(OUT_AC, 75);
	until (SENSOR_1 == 1);
	Off(OUT_AC);
}
```

Aqui estão duas linhas importantes. A primeira linha do programa diz ao robô qual o tipo de sensor que usamos. `IN_1` é o numero da entrada que conectamos o sensor. As outras entradas para sensores são chamadas `IN_2`, `IN_3` e `IN_4`. `SENSOR_TOUCH` indica que este é um sensor de toque. Para o sensor de luz devemos usar `SENSOR_LIGHT`[4]. Depois que especificamos o tipo de sensor, o programa liga ambos os motores e o robô começa a se mover para frente. O próximo *statement* é uma construção muito útil. Ele espera até que a condição entre parênteses seja verdadeira. A condição diz que o valor do `SENSOR_1` precisa ser 1, o que significa que o sensor foi pressionado. Enquanto o sensor não for pressionado, o valor é 0. Então esse comando espera até o sensor ser pressionado. Quando a condição é cumprida, os motores são desligados e a tarefa se encerra.

[4] O kit NXT 2.0 não vem com o sensor de luz aqui mencionado. Em seu lugar, traz um sensor de cor, que também pode funcionar como um sensor de luz. A nota da próxima página explica a função para seu uso.

Agindo sobre um sensor de toque

Vamos tentar agora fazer com que o robô se desvie de obstáculos. Sempre que o robô atingir um objeto, nós faremos ele se mover para trás um pouco, fazer uma curva e então continuar. Eis o programa:

```
task main()
{
	SetSensorTouch(IN_1);
	OnFwd(OUT_AC, 75);
	while (true)
	{
		if (SENSOR_1 == 1)
		{
			OnRev(OUT_AC, 75); Wait(300);
			OnFwd(OUT_A, 75); Wait(300);
			OnFwd(OUT_AC, 75);
		}
	}
}
```

Como no exemplo anterior, nós primeiro indicamos o tipo de sensor. Em seguida o robô começa a se mover para frente. No *loop* infinito do comando **`while`** nós constantemente testamos se o sensor está sendo tocado, e, se acontecer, ele retrocede por 300ms, faz uma curva para direita por 300ms e continua a se mover para frente de novo.

Sensor de luz

Além do sensor de toque, você também recebe um sensor de luz, um sensor de som e um sensor digital ultrassônico com o sistema Mindstorm NXT[5]. O sensor de luz pode ser acionado para emitir luz ou não, então você pode mensurar a quantidade de luz refletida ou de luz ambiente em uma direção particular. Medir a luz refletida é particularmente útil quando se faz o robô seguir uma linha no chão. Isso é o que vamos fazer no próximo exemplo. Para seguir com as experimentações, termine de montar o Tribot. Conecte o sensor de luz à entrada 3, o sensor de som à entrada 2 e o sensor ultrassônico à entrada 4, como indicado pelas instruções.

[5] N. do R.: os sensores que acompanham o kit NXT 2.0 são de toque (duas unidades), de ultrassom e de cor/luz (que também pode determinar a cor de um objeto, além de funcionar como sensor de luz).

Nós também vamos precisar da zona de teste com a trilha preta que vem com o conjunto NXT. O princípio básico de seguir a linha é que o robô tenta se manter na borda da linha preta, afastando-se da linha se o nível de luz está muito baixo (e o sensor está no meio da linha) e tomando a direção da linha se o sensor está fora da trilha e detecta um nível de luz muito alto. Aqui está um programa muito simples de seguir a linha com um único valor limiar (*threshold*) de luz.[6]

```
#define THRESHOLD 40
task main()
{
	SetSensorLight(IN_3);  // ou SetSensor(S3,SENSOR_COLORRED);
	OnFwd(OUT_AC, 75);
	while (true)
	{
		if (Sensor(IN_3) > THRESHOLD)
		{
			OnRev(OUT_C, 75);
			Wait(100);
			until(Sensor(IN_3) <= THRESHOLD);
			OnFwd(OUT_AC, 75);
		}
	}
}
```

O programa primeiro configura a porta 3 como um sensor de luz. Depois ele comanda o robô para se mover para frente e entra num *loop* infinito. Sempre que o valor de luz é maior do que 40 (nós usamos uma constante aqui que pode ser adaptada facilmente, porque depende muito da luz no ambiente) nós revertemos um motor e esperamos até estarmos na trilha novamente.

Como você pode ver quando você executar o programa, o movimento não é muito suave. Experimente adicionar um `Wait(100)`[7] antes do comando **`until`** para que o robô se mova melhor. Note que o programa não funciona para uma trilha que se move no sentido anti-horário. Para habilitar movimento por um caminho arbitrário é necessário um programa muito mais complicado.

Para ler a intensidade da luz ambiente com o LED desligado, configure o sensor da seguinte forma:[8]

```
SetSensorType(IN_3,IN_TYPE_LIGHT_INACTIVE);
SetSensorMode(IN_3,IN_MODE_PCTFULLSCALE);
ResetSensor(IN_3);
```

Sensor de som

Usando o sensor de som você pode transformar seu conjunto NXT em um *clapper*[9]! Nós vamos escrever um programa que espera por um som alto e dirige o robô até que outro som seja detectado. Conecte o sensor de som na entrada 2, como descrito no guia de instruções do Tribot.

[6] N. do R.: Para usar o sensor de cor (presente no kit NXT 2.0) no lugar do sensor de luz, use a opção: `SetSensor(S3,SENSOR_COLORRED);`
Mais informações: http://bricxcc.sourceforge.net/nbc/nxcdoc/nxcapi/group___sensor_type_modes.html
[7] N. do R.: O valor de espera deve ser ajustado dependendo da espessura da trilha e da configuração do caminho a ser seguido pelo robô.
[8] N. do R.: Para o sensor de cor, substitua `IN_TYPE_LIGHT_INACTIVE` por `IN_TYPE_COLORNONE`.
[9] N. do R.: *Clapper* é um dispositivo ativado por som. O sensor de som não está presente no kit NXT 2.0.

```
#define THRESHOLD 40
#define MIC SENSOR_2

task main()
{
	SetSensorSound(IN_2);
	while(true){
		until(MIC > THRESHOLD);
		OnFwd(OUT_AC, 75);
		Wait(300);
		until(MIC > THRESHOLD);
		Off(OUT_AC);
		Wait(300);
		}
}
```

Primeiro definimos uma constante `THRESHOLD` e um outro nome para o `SENSOR_2:` na tarefa principal, nós configuramos a porta 2 para ler dados que vêm do sensor de som e começamos um *loop* infinito.

Usando o comando **`until`**, o programa espera o nível de som ser maior do que o limiar que escolhemos: note que o `SENSOR_2` não é apenas um nome, mas um macro que retorna o valor do som lido pelo sensor.

Se um som alto ocorrer, o robô começa a mover em linha reta até que outro som o faça parar.

O comando **`wait`** foi inserido porque de outra maneira começaria e pararia instantaneamente: de fato, o NXT é tão rápido que quase não gasta tempo para executar as duas linhas entre os dois comandos **`until`**. Se você transformar o primeiro e o segundo **`wait`** em comentários (para que as linhas sejam ignoradas) você entenderá isso melhor. Uma alternativa para o uso do **`until`** para esperar eventos é usar o **`while`,** colocando entre parênteses uma condição complementar, exemplo:

```
while(MIC <= THRESHOLD).
```

Não há muito mais coisas para saber sobre sensores analógicos do NXT; apenas lembre-se que ambos os sensores de luz e de som retornam uma leitura de 0 a 100.

Sensor Ultrassônico

O sensor ultrassônico funciona como um sonar: falando grosseiramente, ele envia uma rajada de ondas ultrassônicas e mede o tempo necessário para que as ondas sejam refletidas de volta pelo objeto em vista. Esse é um sensor digital, significando que ele possui um dispositivo embutido integrado para analisar e enviar dados. Com esse novo sensor você pode fazer um robô enxergar e evitar um obstáculo antes de necessariamente atingi-lo (como no caso de um sensor de toque).

```
#define NEAR 15 //cm

task main(){
	SetSensorLowspeed(IN_4);
	while(true){
		OnFwd(OUT_AC,50);
		while(SensorUS(IN_4)>NEAR);
		Off(OUT_AC);
		OnRev(OUT_C,100);
		Wait(800);
	}
}
```

Este programa inicializa a porta 4 para ler dados do sensor digital de ultrassom; então roda um *loop* infinito em que o robô anda em linha reta até que algo mais perto do que NEAR cm (15 cm, no nosso exemplo) é avistado, então ele vira um pouco e começa a ir em linha reta novamente.

Sumário

Neste capítulo você viu como trabalhar com todos os sensores incluídos no conjunto NXT. Também vimos como os comandos **`until`** e **`while`** são úteis quando se usa sensores.

Eu recomendo que você escreva alguns programas por si só nesse ponto. Você tem todos os ingredientes para dar a seus robôs comportamentos bem complicados: tente traduzir para NXC os programas simples que são mostrados no guia de programação do Robo Center, o *software* padrão do NXT.

VI. Tarefas e Sub-rotinas

Até agora todos os nossos programas consistiram em uma única tarefa. Porém, programas NXC podem ter múltiplas tarefas. Também é possível usar trechos de código nas chamadas sub-rotinas. Usar tarefas e sub-rotinas torna seu programa mais fácil de entender e mais compacto. Neste capítulo nós iremos ver as várias possibilidades.

Tarefas

Um programa de NXC consiste em 255 tarefas no máximo; cada uma possui um nome único. A tarefa chamada `main` precisa sempre existir, já que é a primeira tarefa que será executada. As outras tarefas serão executadas apenas quando a tarefa em execução as chamar, ou se estão explicitamente agendadas no `main`; a tarefa `main` precisa ser finalizada antes que as outras possam começar. A partir daí, ambas as tarefas estão rodando simultaneamente.

Deixe-me mostrar o uso de tarefas. Nós queremos fazer um programa em que o robô se desloca em trajetória quadrada, como fizemos antes. Porém, quando atingir um obstáculo, ele deve reagir a isso. É difícil escrever o programa em apenas uma tarefa, porque o robô precisa fazer duas coisas ao mesmo tempo: se deslocar (ou seja, ligar e desligar os motores no tempo certo) e vigiar os sensores. Então é melhor usar duas tarefas para isso, uma tarefa o faz descrever a trajetória quadrada; a outra o faz reagir aos sensores. Eis o programa:

```
mutex moveMutex;

task move_square()
{
	while (true)
	{
		Acquire(moveMutex);
		OnFwd(OUT_AC, 75); Wait(1000);
		OnRev(OUT_C, 75); Wait(500);
		Release(moveMutex);
	}
}

task check_sensors()
{
	while (true)
	{
		if (SENSOR_1 == 1)
		{
			Acquire(moveMutex);
			OnRev(OUT_AC, 75); Wait(500);
			OnFwd(OUT_A, 75); Wait(500);
			Release(moveMutex);
		}
	}
}

task main()
{
	Precedes(move_square, check_sensors);
	SetSensorTouch(IN_1);
}
```

A tarefa `main` apenas seleciona o tipo de sensor e então inicializa as outras tarefas, adicionando-as na fila de espera; depois disso, a tarefa `main` termina. A tarefa `move_square` move o robô pra sempre em quadrados. A tarefa `check_sensors` checa se o sensor de toque foi apertado, e, caso tenha sido, dirige o robô para fora do obstáculo.

É muito importante lembrar que tarefas inicializadas estão rodando ao mesmo momento e isso pode levar a comportamentos inesperados, caso ambas as tarefas tentem mover os motores como é presumido que façam.

Para evitar esse tipo de problema, nós declaramos um tipo estranho de variável chamado **mutex** (***mut****ual* ***ex****clusion* ou exclusão mútua): só podemos agir sobre essa variável usando as funções **Acquire** e **Release**. Ao escrever pedaços de código entre essas funções temos a certeza que apenas uma tarefa por vez terá total controle sobre os motores.

Esses tipos de variáveis **mutex** são chamados de *semaphores* (sinalizadores) e esse tipo de técnica de programação se chama *concurrent programming* (programação coincidente); esse argumento será descrito em detalhes no capítulo X.

Sub-rotinas

Às vezes você precisará do mesmo pedaço de código em múltiplas partes do seu programa. Nesse caso você pode colocar esse pedaço de código em uma sub-rotina e nomeá-lo. Agora você pode executar esse pedaço de código simplesmente o chamando dentro de uma tarefa. Veja um exemplo:

```
sub turn_around(int pwr)
{
	OnRev(OUT_C, pwr); Wait(900);
	OnFwd(OUT_AC, pwr);
}

task main()
{
	OnFwd(OUT_AC, 75);
	Wait(1000);
	turn_around(75);
	Wait(2000);
	turn_around(75);
	Wait(1000);
	turn_around(75);
	Off(OUT_AC);
}
```

Nesse programa definimos uma sub-rotina que faz o robô girar em torno de seu centro. A tarefa principal chama a sub-rotina três vezes. Note que nós chamamos a sub-rotina escrevendo seu nome e passando um argumento numérico dentro dos parênteses após o nome. Caso a sub-rotina não aceite argumentos, apenas ponha parênteses vazios ().

Elas se parecem com vários comandos que já vimos.

O principal benefício das sub-rotinas é que elas são salvas uma única vez no NXT e isso economiza memória. Porém, quando as sub-rotinas são pequenas, pode ser melhor usar funções **inline** no lugar. Elas não são armazenadas separadamente, mas sim copiadas em cada lugar que são usadas. Isso pode usar mais memória, porém não há limite para o uso de funções **inline**. Elas podem ser declaradas da seguinte forma:

```
inline int Name( Args ) {
//corpo;
return x*y;
}
```

Definir e chamar funções **inline** se faz da mesma forma que se faz com as sub-rotinas. Então o exemplo acima, usando funções **inline**, se pareceria como se segue:

```
inline void turn_around()
{
	OnRev(OUT_C, 75); Wait(900);
	OnFwd(OUT_AC, 75);
}

task main()
{
	OnFwd(OUT_AC, 75);
	Wait(1000);
	turn_around();
	Wait(2000);
	turn_around();
	Wait(1000);
	turn_around();
	Off(OUT_AC);
}
```

No exemplo acima nós podemos fazer o tempo de virada um argumento para a função, como nos exemplos abaixo:

```
inline void turn_around(int pwr, int turntime)
{
	OnRev(OUT_C, pwr);
	Wait(turntime);
	OnFwd(OUT_AC, pwr);
}

task main()
{
	OnFwd(OUT_AC, 75);
	Wait(1000);
	turn_around(75, 2000);
	Wait(2000);
	turn_around(75, 500);
	Wait(1000);
	turn_around(75, 3000);
	Off(OUT_AC);
}
```

Note que nos parênteses após o nome da função `inline` nós especificamos o(s) argumento(s) da função. Indicamos neste caso que o argumento é um inteiro (existem outras escolhas) e que o nome é `turntime`. Quando existem mais argumentos, você deve separá-los com vírgulas. Note que no NXC, **sub** é o mesmo que **void**; Além disso, funções podem ter outros tipos de retorno além de **void**, podem retornar um valor de inteiro ou de string para quem chamou a função: para mais detalhes, veja o guia NXC[10].

Definindo macros

Existe ainda outro jeito de dar nome a pedaços pequenos de código. Você pode definir macros em NXC (não confunda com macros do BricxCC). Já vimos antes que podemos definir constantes usando `#define`, dando a elas um nome. Na verdade, podemos definir qualquer tipo de código. Aqui vai o mesmo programa, porém usando macros para virar.

[10] N. do R.: O guia NXC está disponível em: http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf

```
#define turn_around \
	OnRev(OUT_B, 75); Wait(3400); OnFwd(OUT_AB, 75);

task main()
{
	OnFwd(OUT_AB, 75);
	Wait(1000);
	turn_around;
	Wait(2000);
	turn_around;
	Wait(1000);
	turn_around;
	Off(OUT_AB);
}
```

Depois do *statement* `#define` a palavra `turn_around` irá representar todo o texto após ela. Agora aonde quer que você digite `turn_around`, ela será substituída por esse texto. Note que o texto deve ser em uma linha (existem formas de colocar um `#define` para múltiplas linhas, mas não é recomendado).

Comandos `#define` são na verdade muito mais poderosos. Eles também podem ter argumentos. Exemplo, nós podemos por o tempo de virada como argumento no *statement*. Aqui vai um exemplo no qual definimos quatro macros: uma para se mover para frente, uma para se mover pra trás, uma para virar à esquerda e uma para virar à direita. Cada uma tem dois argumentos: a velocidade e o tempo.

```
#define turn_right(s,t) \
	OnFwd(OUT_A, s);OnRev(OUT_B, s);Wait(t);
#define turn_left(s,t) \
	OnRev(OUT_A, s);OnFwd(OUT_B, s);Wait(t);
#define forwards(s,t) OnFwd(OUT_AB, s);Wait(t);
#define backwards(s,t) OnRev(OUT_AB, s);Wait(t);

task main()
{
	backwards(50,10000);
	forwards(50,10000);
	turn_left(75,750);
	forwards(75,1000);
	backwards(75,2000);
	forwards(75,1000);
	turn_right(75,750);
	forwards(30,2000);
	Off(OUT_AB);
}
```

É bem útil definir tais macros. Isso faz seu código mais compacto e legível. Você também pode mudar mais facilmente seu código quando, por exemplo, alterar as conexões dos motores.

Sumário

Neste capítulo você viu o uso de tarefas, sub-rotinas, funções **inline** e macros. Elas têm usos diferentes. Tarefas normalmente rodam no mesmo momento e tomam conta de coisas diferentes que precisam ser feitas simultaneamente. Sub-rotinas são úteis quando pedaços grandes de código precisam ser usados em lugares diferentes da mesma tarefa. Funções **inline** são úteis quando se precisam usar pedaços de código em lugares diferentes de tarefas diferentes, mas ocupam mais memória. Finalmente, macros são muito úteis para pequenos pedaços de código que precisam ser usados em lugares diferentes. Eles também podem ter parâmetros, o que os torna ainda mais úteis.
Agora que você passou por todos os capítulos até aqui, você possui todas as habilidades necessárias para fazer com que seu robô faça coisas complicadas. Os outros capítulos neste tutorial o ensinarão sobre outras coisas que são importantes apenas em certas aplicações.

VII. Fazendo Música

O NXT possui um alto-falante incluso que pode tocar tons e até arquivos de som. Isso é particularmente útil quando você quer fazer o NXT lhe dizer que algo está acontecendo. Também pode ser engraçado fazer com que o robô toque música ou fale enquanto se move.

Tocando arquivos de som

BricxCC possui uma ferramenta inclusa para converter arquivos .wav em arquivos .rso acesse Files -> Sound Conversion.

Então você pode guardar arquivos de som do formato .rso na memória flash do NXT usando uma outra ferramenta, o explorador de memória NXT (Tools -> NXT explorer) e tocá-los com o comando

```
PlayFileEx(nomedoarquivo, volume, loop?)
```

Os argumentos são o nome do arquivo do som, o volume (número de 0 a 4) e *loop*: este último argumento deve ser `TRUE` (ou 1) se você deseja que o arquivo entre em *loop*, ou `FALSE` (ou 0) se você deseja que seja tocado apenas uma vez.

```
#define TIME 200
#define MAXVOL 7
#define MINVOL 1
#define MIDVOL 3
#define pause_4th Wait(TIME)
#define pause_8th Wait(TIME/2)
#define note_4th \
       PlayFileEx("! Click.rso",MIDVOL,FALSE); pause_4th
#define note_8th \
       PlayFileEx("! Click.rso",MAXVOL,FALSE); pause_8th

task main()
{
       PlayFileEx("! Startup.rso",MINVOL,FALSE);
       Wait(2000);
       note_4th;
       note_8th;
       note_8th;
       note_4th;
       note_4th;
       pause_4th;
       note_4th;
       note_4th;
       Wait(100);
}
```

Este programa primeiro toca o som de inicialização que você provavelmente já conhece; então usa um som clássico de clique para tocar um jingle chamado “Shave and a haircut”! Os macros são especialmente úteis neste caso para simplificar a notação na tarefa `main`: tente modificar as opções de volume para acentuar o som.

Tocando música

Para tocar um som, você pode usar o comando `PlayToneEx(frequency, duration, volume, loop?)`. Ele possui quatro argumentos. O primeiro é a frequência em Hertz, o segundo é a duração (em 1/1000 de segundo, como no comando `wait`), e os últimos são volume e *loop* como antes. `PlayTone(frequency, duration)`também pode ser usado; nesse caso o volume é o escolhido pelo menu do NXT, e o loop está desativado.

Aqui está uma tabela útil de frequências:

Sound	3	4	5	6	7	8	9
B	247	494	988	1976	3951	7902	
A#	233	466	932	1865	3729	7458	
A	220	440	880	1760	3520	7040	14080
G#		415	831	1661	3322	6644	13288
G		392	784	1568	3136	6272	12544
F#		370	740	1480	2960	5920	11840
F		349	698	1397	2794	5588	11176
E		330	659	1319	2637	5274	10548
D#		311	622	1245	2489	4978	9956
D		294	587	1175	2349	4699	9398
C#		277	554	1109	2217	4435	8870
C		262	523	1047	2093	4186	8372

Como no caso de `PlayFileEx`, o NXT não espera a nota acabar. Então se você quiser usar vários sons de uma vez então será melhor adicionar (intervalos um pouco maiores) comandos `wait` entre eles. Aqui está um exemplo:

```
#define VOL 3

task main()
{
	PlayToneEx(262,400,VOL,FALSE); Wait(500);
	PlayToneEx(294,400,VOL,FALSE); Wait(500);
	PlayToneEx(330,400,VOL,FALSE); Wait(500);
	PlayToneEx(294,400,VOL,FALSE); Wait(500);
	PlayToneEx(262,1600,VOL,FALSE); Wait(2000);
}
```

Você pode criar pedaços de música muito facilmente usando o *Brick Piano*, que é parte do BricxCC.

Se você quiser que o NXT toque música enquanto se move, é melhor usar uma tarefa à parte para isso. Aqui está um exemplo de um programa bem estúpido em que o NXT se move pra trás e pra frente, constantemente tocando música.

```
task music()
{
	while (true)
	{
		PlayTone(262,400); Wait(500);
		PlayTone(294,400); Wait(500);
		PlayTone(330,400); Wait(500);
		PlayTone(294,400); Wait(500);
	}
}

task movement()
{
	while(true)
	{
		OnFwd(OUT_AC, 75); Wait(3000);
		OnRev(OUT_AC, 75); Wait(3000);
	}
}

task main()
{
	Precedes(music, movement);
}
```

Sumário

Neste capítulo você aprendeu como fazer o NXT tocar sons e música. Também viu como usar uma tarefa em separado para música.

VIII. Mais sobre motores

Existe um número de comandos adicionais para motor que você pode usar para controlar os motores mais precisamente. Neste capítulo os discutiremos: `ResetTachoCount`, `Coast(Float)`, `OnFwdReg`, `OnRevReg`, `OnFwdSync`, `OnRevSync`, `RotateMotor`, `RotateMotorEx`, e conceitos básicos de **PID**[11].

Parando suavemente

Quando você usa o comando `Off()`, o servo-motor para imediatamente, freando o eixo e mantendo a posição. Também é possível parar os motores de um modo mais suave, não usando os freios. Para isso use o comando `Float()`ou `Coast()`(não há distinção), que simplesmente corta a força do motor. Aqui vai um exemplo. Primeiro o motor para usando os freios; depois, sem usar os freios. Note a diferença. Na verdade a diferença é muito pequena para esse robô em particular. Porém faz grande diferença em outros robôs.

```
task main()
{
	OnFwd(OUT_AC, 75);
	Wait(500);
	Off(OUT_AC);
	Wait(1000);
	OnFwd(OUT_AC, 75);
	Wait(500);
	Float(OUT_AC);
}
```

Comandos avançados

Os comandos `OnFwd()` e `OnRev()`são as mais simples rotinas para se mover motores.

Os servo-motores do NXT possuem um codificador interno (*encoder*) que permite que você controle precisamente a posição do eixo e sua velocidade;

O *Firmware* do NXT implementa um controlador **PID** em malha fechada para controlar a posição dos motores e suas velocidades usando a informação dos *encoders* como realimentação (*feedback*).

Se você quer que seu robô se mova perfeitamente em linha reta, você pode usar um **recurso de sincronização** que faz com que dois motores selecionados rodem juntos e esperem um pelo outro caso um deles fique lento ou bloqueado; de forma similar, você pode configurar os dois motores para se moverem em sincronia com uma porcentagem de direção para virar para esquerda, direita ou girar no próprio eixo, mas sempre mantendo a sincronia. Existem muitos comandos para liberar todo o potencial dos servo-motores!

`OnFwdReg('ports','speed','regmode')` aciona o motor especificado em 'ports' com força 'speed' aplicando um modo de regulação que pode ser `OUT_REGMODE_IDLE`, `OUT_REGMODE_SPEED`, ou `OUT_REGMODE_SYNC`. Se IDLE for selecionado, nenhuma regulação PID será aplicada; se o modo SPEED for selecionado, o NXT regula um motor para ter velocidade constante, mesmo se a carga do motor variar; finalmente, se SYNC é selecionado, ambos os motores especificados pelas 'ports' se movem em sincronia, como explicado anteriormente.

`OnRevReg()` age como o comando precedente, revertendo a direção.

[11] N. do R.: O termo PID (Proporcional-Integral-Derivativo) refere-se a um tipo de controlador cuja saída depende do valor atual do erro, de sua derivada temporal e de sua integral no tempo.

```
task main()
{
	OnFwdReg(OUT_AC,50,OUT_REGMODE_IDLE);
	Wait(2000);
	Off(OUT_AC);
	PlayTone(4000,50);
	Wait(1000);
	ResetTachoCount(OUT_AC);
	OnFwdReg(OUT_AC,50,OUT_REGMODE_SPEED);
	Wait(2000);
	Off(OUT_AC);
	PlayTone(4000,50);
	Wait(1000);
	OnFwdReg(OUT_AC,50,OUT_REGMODE_SYNC);
	Wait(2000);
	Off(OUT_AC);
}
```

Este programa mostra diferentes tipos de controle, que você pode notar se tentar parar as rodas segurando o robô nas mãos: primeiro (modo IDLE), ao parar uma roda você não vai notar nada; depois (modo SPEED), ao tentar diminuir a velocidade da roda, você verá que o NXT aumenta a força do motor para tentar sobrepujar sua resistência, tentando manter a velocidade constante; finalmente (modo SYNC), parar uma roda causará que a outra também pare, esperando a roda que está bloqueada.

`OnFwdSync('ports','speed','turnpct')` é o mesmo que o comando no modo SYNC, mas agora você pode especificar o percentual de direção "turnpct' (de -100 até 100).

`OnRevSync()` é o mesmo que antes, simplesmente revertendo a direção de um motor. O programa a seguir mostra esses comandos: tente mudar o número de direção para ver como ele se comporta.

```
task main()
{
	PlayTone(5000,30);
	OnFwdSync(OUT_AC,50,0);
	Wait(1000);
	PlayTone(5000,30);
	OnFwdSync(OUT_AC,50,20);
	Wait(1000);
	PlayTone(5000,30);
	OnFwdSync(OUT_AC,50,-40);
	Wait(1000);
	PlayTone(5000,30);
	OnRevSync(OUT_AC,50,90);
	Wait(1000);
	Off(OUT_AC);
}
```

Finalmente, os motores também podem ser configurados para girarem determinada quantidade de graus (lembre-se que um giro completo tem 360°).

Para ambos dos comandos a seguir, você pode atuar na direção do motor mudando ou o sinal da velocidade ou o sinal do ângulo: então, se a velocidade e ângulo possuem o mesmo sinal, o motor irá para frente; caso os sinais sejam opostos, o motor irá para trás.

`RotateMotor('ports','speed','degrees')` rotaciona o eixo do motor especificado por 'ports' em um ângulo 'degrees' na força 'speed' (de 0 até 100).

```
task main()
{
	RotateMotor(OUT_AC, 50,360);
	RotateMotor(OUT_C, 50,-360);
}
```

`RotateMotorEx('ports','speed','degrees','turnpct','sync', 'stop')` é uma extensão do comando precedente que deixa você sincronizar dois motores (e.g. `OUT_AC`) especificando uma porcentagem de direção (de -100 até 100) e um marcador booleano (que pode ser configurado para **true** ou **false**). Ele também deixa você especificar se os motores devem frear depois que o ângulo de rotação foi completo usando o marcador booleano **'stop'**.

```
task main()
{
	RotateMotorEx(OUT_AC, 50, 360, 0, true, true);
	RotateMotorEx(OUT_AC, 50, 360, 40, true, true);
	RotateMotorEx(OUT_AC, 50, 360, -40, true, true);
	RotateMotorEx(OUT_AC, 50, 360, 100, true, true);
}
```

Controle PID

O **firmware** do NXT implementa um controlador PID (Proporcional-Integrativo-Derivativo) para regular a posição e a velocidade dos servo-motores com precisão. Esse tipo de controlador é um dos mais simples, porém dos mais efetivos controladores de realimentação (em malha fechada) conhecidos na automação, e é frequentemente usado.

De modo grosseiro, ele funciona assim (Eu falarei sobre regulação de posição para um controlador de tempo discreto):

Seu programa manda para o controlador um determinado ponto de referência R(t) para alcançar; ele atua no motor com um comando U(t), medindo sua posição real Y(t) com o codificador interno (*encoder*) e calcula um erro E(t) = R(t) – Y(t): aqui está o porquê de ser chamado "controlador de malha fechado", porque a posição de saída Y(t) volta a entrada do controlador para calcular o erro. O controlador transforma o erro E(t) no comando U(t) da seguinte forma:

U(t) = P(t) + I(t) + D(t), onde
$P(t) = K_P \cdot E(t)$,
$I(t) = K_I \cdot (I(t-1) + E(t))$,
e $D(t) = K_D \cdot (E(t) - E(t-1))$.

Pode parecer difícil para um iniciante, mas tentarei explicar esse mecanismo o máximo que eu puder.

O comando é a soma de três termos: a parte proporcional P(t), a parte integral I(t) e a parte derivativa D(t).

P(t) faz o controlador ser rápido, mas não garante um erro nulo no equilíbrio;

I(t) dá "memória" ao controlador, no sentido que ele guarda o traço de erros acumulados e os compensa, com a garantia de erro zero no equilíbrio;

D(t) dá uma "predição futura" ao controlador (como a derivação na matemática), melhorando o tempo de resposta.

Sei que isso pode parecer confuso, considere que livros acadêmicos inteiros já foram escritos sobre este tema! Mesmo assim, podemos experimentá-lo online, com nossos blocos NXT! O programa mais simples para ajudar a fixar na memória é o seguinte.

```
#define P 50
#define I 50
#define D 50

task main(){
      RotateMotorPID(OUT_A, 100, 180, P, I, D);
      Wait(3000);
}
```

O comando `RotateMotorPID` deixa você mover o motor selecionando ganhos PID diferentes dos ganhos-padrões. Tente executar o programa com os seguintes valores:

(50,0,0): o motor não rotaciona 180° exatamente, já que resta um erro não compensado.

(0,x,x): sem a parte proporcional, o erro é muito grande.

(40,40,0): existe uma quebra de limite, significa que o eixo do motor se moverá além do ponto selecionado e então retornará.

(40,40,90): boa precisão e bom tempo para atingir o ponto determinado.

(40,40,200): o eixo oscila, já que o ganho de derivada é muito grande.

Tente outros valores para descobrir como esses ganhos influenciam no desempenho do motor.

Sumário

Neste capítulo você aprendeu sobre comandos avançados de motor disponíveis: `Float()`,`Coast()` que param o motor de modo suave; `OnXxxReg()`e `OnXxxSync()` que permitem controle de **feedback** na velocidade do motor e sincronismo; `RotateMotor()`e `RotateMotorEx()` são usados para girar o eixo do motor de um ângulo específico. Você também aprendeu alguma coisa sobre controle PID; não foi uma explanação exaustiva, mas talvez eu tenha causado alguma curiosidade em você: procure na internet sobre ele![12]

[12] N. do R.: O aluno Ivan Seidel fez um vídeo muito interessante com uma explicação básica sobre o PID: http://techlego.blogspot.com/2011/11/algoritmo-pid-basico.html.
Para uma explicação mais detalhada, veja http://en.wikipedia.org/wiki/PID_controller.

IX. Mais sobre sensores

No capítulo 5 nós discutimos os aspectos básicos do uso de sensores. Mas existe muito mais a se fazer com sensores. Neste capítulo, discutiremos a diferença entre o modo do sensor e tipo do sensor, veremos como usar sensores antigos compatíveis com RCX, ligando-os ao NXT usando cabos conversores da Lego.

Modo e tipo do sensor

O comando `SetSensor()` que nós vimos antes faz, na verdade, duas coisas: ele configura o **tipo** de sensor e o **modo** como o sensor opera. Selecionando o modo e o tipo do sensor separadamente, você pode controlar o comportamento do sensor mais precisamente, o que é útil para certas aplicações.

O tipo de sensor é selecionado com o comando `SetSensorType()`. Existem muitos tipos diferentes, mas eu mostrarei os principais[13]: `SENSOR_TYPE_TOUCH`, que é o sensor de toque, `SENSOR_TYPE_LIGHT_ACTIVE`, que é o sensor de luz (com LED ligado), `SENSOR_TYPE_SOUND_DB`, que é o sensor de som e `SENSOR_TYPE_LOWSPEED_9V`, que é o sensor ultrassônico. Selecionar o tipo de sensor é particularmente importante para indicar se o sensor precisa de energia (por exemplo para acender o LED no sensor de luz), ou indicar ao NXT que o sensor é digital e precisa ser lido via protocolo serial I²C. É possível usar sensores antigos do RCX com NXT: `SENSOR_TYPE_TEMPERATURE` para o sensor de temperatura, `SENSOR_TYPE_LIGHT` para o sensor antigo de luz e `SENSOR_TYPE_ROTATION` para o sensor de rotação RCX (este tipo será discutido mais tarde).

O modo do sensor é configurado com o comando `SetSensorMode()`. Existem oito tipos diferentes de modos. O mais importante é o `SENSOR_MODE_RAW`. Nesse modo, o valor que você recebe quando chega o sensor é um número entre 0 e 1023. E o valor "cru" produzido pelo sensor. O que ele significa depende do sensor. Exemplo, para um sensor de toque, quando o sensor não é pressionado o valor é perto de 1023. Quando é totalmente pressionado, é perto de 50. Parcialmente pressionado, o valor está entre 50 e 1000. Então se você seta um sensor de toque para o modo **raw** você pode descobrir se ele está sendo tocado parcialmente. Quando o sensor é um sensor de luz, o valor varia entre 300 (bem claro) até 800 (bem escuro). Isso dá um valor muito mais preciso do que apenas usando o comando `SetSensor()`. Para mais detalhes, veja o guia de programação do NXC.

O segundo modo de sensor é `SENSOR_MODE_BOOL`. Nesse modo o valor é 0 ou 1. Quando o valor **raw** é acima de 562 o valor é 0, doutro modo é 1. `SENSOR_MODE_BOOL` é o modo padrão para um sensor de toque, mas pode ser usado para outros tipos, descartando informações analógicas. Os modos `SENSOR_MODE_CELSIUS` e `SENSOR_MODE_FAHRENHEIT` são uteis apenas com sensores de temperatura e dão a temperatura da maneira indicada. `SENSOR_MODE_PERCENT` converte o valor **raw** em um valor entre 0 e 100. `SENSOR_MODE_PERCENT` é o modo padrão para um sensor de luz. `SENSOR_MODE_ROTATION` é usado apenas para o sensor de rotação (veja abaixo).

Existem outros dois modos interessantes: `SENSOR_MODE_EDGE` e `SENSOR_MODE_PULSE` . Eles contam transições, isto é, mudanças de valores **raw** de baixo para alto ou o oposto. Por exemplo, quando você toca um sensor de toque isso causa uma transição de um valor alto para baixo. Então quando você solta terá uma transição na outra direção. Quando você seta o modo do sensor para `SENSOR_MODE_PULSE`, apenas transições de baixo para alto são contadas. Então cada vez que apertar e soltar o sensor contará como um. Quando você seta o modo do sensor para `SENSOR_MODE_EDGE`, ambas as transições são contadas. Então cada vez que apertar e soltar o sensor contará como dois. Você pode utilizar isso para calcular a frequência com que o sensor está sendo acionado. Ou você pode utilizar isso em uma combinação com um sensor de luz para contar quão frequente uma lâmpada (forte) é ligada ou desligada. É claro que, quando estiver contando **edges** ou **pulses**, você deve ser capaz de selecionar o contador de volta para 0. Para isso usa-se o comando `ClearSensor()`, que limpa os contadores para o sensor indicado.

[13] Veja outros tipos em: http://bricxcc.sourceforge.net/nbc/nxcdoc/nxcapi/group___sensor_types.html.

Vamos ver um exemplo. O seguinte programa usa um sensor do toque para direcionar o robô. Conecte o sensor de toque com um fio longo na entrada um. Se tocar o sensor duas vezes rapidamente, o robô se move para frente. Se tocar apenas uma vez, ele para de se mover.

```
task main()
{
	SetSensorType(IN_1, SENSOR_TYPE_TOUCH);
	SetSensorMode(IN_1, SENSOR_MODE_PULSE);
	while(true)
	{
		ClearSensor(IN_1);
		until (SENSOR_1 > 0);
		Wait(500);
		if (SENSOR_1 == 1) {Off(OUT_AC);}
		if (SENSOR_1 == 2) {OnFwd(OUT_AC, 75);}
	}
}
```

Note que primeiro nós configuramos o tipo do sensor e o modo. Isso é essencial, pois ao mudar o tipo também afetaremos o modo.

Sensor de rotação

O sensor de rotação é um tipo de sensor muito útil: ele é um codificador óptico (*encoder*), quase o mesmo que está dentro dos servo-motores do NXT. O sensor de rotação possui um orifício pelo qual você pode colocar um eixo, do qual a posição angular é medida. Uma rotação completa do eixo conta 16 passos (ou -16 se você girar inversamente), o que significa uma resolução de 22,5 graus, muito pequena em comparação com a resolução do servomotor, que é de 1 grau. Esse tipo antigo de sensor de rotação pode ser útil para monitorar um eixo sem a necessidade de desperdiçar um motor; considere também que usar um motor como codificador precisa de muito torque para move-lo, enquanto o sensor antigo de rotação é muito fácil de girar.

Se você precisar de uma resolução ainda melhor do que 16 passos por rotação, você pode usar engrenagens para aumentar mecanicamente o número de contagens por volta.

O próximo exemplo foi herdado do antigo tutorial para RCX.

Uma aplicação padrão é ter dois sensores de rotação conectados às duas rodas do robô que você controla com dois motores. Para um movimento em linha reta, ambas as rodas devem girar na mesma velocidade. Infelizmente, os motores normalmente não rodam na mesma velocidade de forma natural. Usando os sensores de rotação você pode ver que uma das rodas gira mais rápido. Você pode temporariamente parar o motor (é melhor se usar `Float()`) até que ambos os sensores registrem o mesmo valor novamente. O seguinte programa faz isso. Ele simplesmente deixa o robô andar em linha reta. Para usá-lo, mude seu robô conectando os dois sensores de rotação as duas rodas. Conecte os sensores nas entradas 1 e 3.

```
task main()
{
	SetSensor(IN_1, SENSOR_ROTATION); ClearSensor(IN_1);
	SetSensor(IN_3, SENSOR_ROTATION); ClearSensor(IN_3);
	while (true)
	{
		if (SENSOR_1 < SENSOR_3)
			{OnFwd(OUT_A, 75); Float(OUT_C);}
		else if (SENSOR_1 > SENSOR_3)
				{OnFwd(OUT_C, 75); Float(OUT_A);}
			else
				{OnFwd(OUT_AC, 75);}
	}
}
```

Este programa primeiro indica que ambos os sensores são sensores de rotação, e então limpa os valores registrados. Depois, ele começa um *loop* infinito. Dentro do *loop* checamos se a leitura dos dois sensores

está igual. Caso sim, então o robô simplesmente se move para sempre. Se um é maior, o motor correto é parado até que ambas as leituras fiquem iguais.

Claramente este programa é muito simples. Você pode estendê-lo para fazer o robô andar distâncias exatas ou fazer com que ele faça curvas mais precisas.

Colocando vários sensores em apenas uma entrada

É preciso uma pequena isenção de responsabilidade no topo desta seção. Devido à nova estrutura nos sensores melhorados do NXT e 6 cabos, não é tão fácil como antes (como era para RCX) de conectar mais de um sensor na mesma porta. Na minha honesta opinião, a única aplicação confiável (e fácil de fazer) seria construir um sensor de toque multiplexador analógico para usar em combinação com um cabo conversor. A alternativa é um multiplexador digital complexo que pode lidar com comunicação I²C com o NXT, mas isso não é uma solução acessível para iniciantes.

O NXT possui quatro entradas para conectar sensores. Quando você quer fazer robôs mais complicados (e você usou sensores extras) isso pode não ser o suficiente para você. Felizmente, com alguns truques, você pode conectar dois (ou talvez mais) sensores em uma entrada.

O mais fácil é conectar dois sensores de toque em uma entrada. Se um deles (ou ambos) é tocado o valor é 1, senão é 0. Você não pode distinguir os dois, porém algumas vezes isso não é necessário. Por exemplo, você pode colocar um sensor de toque na frente e outro atrás do robô, e saberá qual está sendo tocado baseado na direção em que o robô está se movendo. Mas, você também pode configurar o modo de entrada para **raw** (veja acima). Agora, você coletar muito mais informações. Se der sorte, o valor quando o sensor é pressionado não será o mesmo para ambos os sensores. Caso este seja o caso, você pode facilmente distinguir entre os dois sensores. Se, quando ambos são pressionados, você observar um valor muito baixo (perto de 30), você também consegue detectar isso.

Você também pode conectar um sensor de toque e um sensor de luz a uma entrada (sensores RCX apenas). Selecione o tipo para sensor de luz (senão o sensor de luz não funcionará). Configure o modo para **raw**. Nesse caso, quando o sensor de toque for pressionado você pegará um valor menor que 100. Se não for pressionado você pegará o valor do sensor de luz, que nunca é menor que 100. O programa a seguir usa essa ideia. O robô precisa ser equipado com um sensor de luz apontando para baixo e um pára-choque na frente conectado ao sensor de toque. Conecte ambos a porta 1. O robô se moverá aleatoriamente numa área com luz. Quando o sensor de luz enxergar uma linha escura (valor **raw** > 750) ele retrocede um pouquinho. Quando o sensor de toque tocar em algo (valor **raw** < 100) ele faz o mesmo. Aqui está o programa:

```
mutex moveMutex;
int ttt,tt2;

task moverandom()
{
	while (true)
	{
		ttt = Random(500) + 40;
		tt2 = Random();
		Acquire(moveMutex);
		if (tt2 > 0)
			{ OnRev(OUT_A, 75); OnFwd(OUT_C, 75);
			Wait(ttt); }
		else
			{ OnRev(OUT_C, 75); OnFwd(OUT_A, 75);
			Wait(ttt); }
		ttt = Random(1500) + 50;
		OnFwd(OUT_AC, 75); Wait(ttt);
		Release(moveMutex);
	}
}

task submain()
{
	SetSensorType(IN_1, SENSOR_TYPE_LIGHT);
	SetSensorMode(IN_1, SENSOR_MODE_RAW);
	while (true)
	{
		if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
		{
			Acquire(moveMutex);
			OnRev(OUT_AC, 75); Wait(300);
			Release(moveMutex);
		}
	}
}

task main()
{
	Precedes(moverandom, submain);
}
```

Espero que este programa seja claro. Aqui existem duas tarefas. A tarefa `moverandom` faz o robô se mover de forma aleatória. A tarefa principal primeiro roda `moverandom`, seleciona o sensor e então espera algo acontecer. Se a leitura do sensor for muito pequena (toque) ou muito alta (fora da área clara) ele para o movimento, retrocede um pouco e então começa a mover aleatoriamente de novo.

Sumário

Neste capítulo vimos um número adicional de questões sobre sensores. Vimos como configurar separadamente o tipo e o modo de um sensor e como isso pode ser usado para coletar informação adicional. Aprendemos como usar o sensor de rotação. E vimos como múltiplos sensores podem ser conectados a uma porta do NXT. Todos esses truques são extremamente úteis ao se construir robôs mais complicados. Sensores sempre possuem um papel crítico nisso.

X. Tarefas paralelas

Como foi indicado antes, tarefas no NXC são executadas simultaneamente ou em paralelo, como as pessoas normalmente dizem. Isso é extremamente útil. Permite que você cuide dos sensores em uma tarefa enquanto outra tarefa move o robô por aí e outra toca alguma música. Porém, tarefas paralelas podem causar problemas. Uma tarefa pode interferir com outra.

Um programa errado

Considere o seguinte programa. Aqui uma tarefa dirige o robô em quadrados (como fizemos tanto antes) e a segunda checa o sensor de toque. Quando o sensor é tocado, ele se move um pouco para trás e faz uma conversão de 90 graus.

```
task check_sensors()
{
	while (true)
	{
		if (SENSOR_1 == 1)
		{
			OnRev(OUT_AC, 75);
			Wait(500);
			OnFwd(OUT_A, 75);
			Wait(850);
			OnFwd(OUT_C, 75);
		}
	}
}

task submain()
{
	while (true)
	{
		OnFwd(OUT_AC, 75); Wait(1000);
		OnRev(OUT_C, 75); Wait(500);
	}
}

task main()
{
	SetSensor(IN_1,SENSOR_TOUCH);
	Precedes(check_sensors, submain);
}
```

Este exemplo provavelmente parece com um programa perfeitamente valido. Porém, se você executá-lo muito provavelmente encontrará um comportamento inesperado. Tente o seguinte: Faça o robô tocar em algo enquanto está virando. Ele começará a ir para trás, mas imediatamente seguirá em frente novamente, atingindo o obstáculo. A razão para isso é que as tarefas podem interferir. O seguinte está acontecendo: o robô está virando, ou seja, a primeira tarefa está em seu segundo *statement* de *sleep*. Agora o robô atinge o sensor e começa a ir para trás, mas, neste mesmo momento, a tarefa principal já saiu do *sleep* e está pronta para mover o robô para frente novamente; na direção do obstáculo. A segunda tarefa está "dormindo" nesse momento então não notará a colisão. Claramente esse não é o comportamento que gostaríamos de ver. O problema é que enquanto a segunda tarefa está em modo *sleep* nós não notamos que a primeira tarefa ainda está rodando e isso interfere com as ações da segunda tarefa.

Seções críticas e variáveis mutex

Um jeito de resolver esse problema é ter certeza que a qualquer momento apenas uma tarefa está dirigindo o robô. Esse foi o caminho que tomamos no Capítulo IV. Deixe-me repetir o programa aqui.

```
mutex moveMutex;

task move_square()
{
	while (true)
	{
		Acquire(moveMutex);
		OnFwd(OUT_AC, 75); Wait(1000);
		OnRev(OUT_C, 75); Wait(850);
		Release(moveMutex);
	}
}

task check_sensors()
{
	while (true)
	{
		if (SENSOR_1 == 1)
		{
			Acquire(moveMutex);
			OnRev(OUT_AC, 75); Wait(500);
			OnFwd(OUT_A, 75); Wait(850);
			Release(moveMutex);
		}
	}
}

task main()
{
	SetSensor(IN_1,SENSOR_TOUCH);
	Precedes(check_sensors, move_square);
}
```

O ponto crucial é que ambas as tarefas `check_sensors` e `move_square` podem controlar os motores apenas se nenhuma outra tarefa os estiver usando: isto é feito usando o *statement* **Acquire** que espera que a variável de exclusão mútua seja liberada antes de usar os motores. O comando **Acquire** é o oposto do comando **Release**, que libera a variável **mutex** para que outras tarefas também possam usar o recurso crítico, motores em nosso caso. O código dentro do escopo *acquire-release* é chamado de região crítica: crítica significa que recursos compartilhados estão em uso. Deste jeito tarefas não podem interferir umas nas outras.

Usando sinalizadores

Existe uma alternativa feita à mão para as variáveis **mutex,** que é a implementação explícita dos comandos **Acquire** e **Release**.

Uma técnica padrão para resolver este problema é o uso de uma variável para indicar quais tarefas estão com o controle do motor. Outras tarefas não têm permissão para dirigir os motores até que a primeira tarefa indique, usando a variável, que pode. Tal variável é frequentemente chamada de sinalizador (*semaphore*). Seja `sem` um sinalizador (como **mutex**). Nós assumimos que um valor 0 indica que nenhuma tarefa está controlando os motores (o recurso está livre). Agora, sempre que uma tarefa desejar fazer algo com os motores executará os seguintes comandos:

```
until (sem == 0);
sem = 1; //Acquire(sem);

// Faz alguma coisa com os motores
// Região crítica

sem = 0; //Release(sem);
```

Primeiro, esperamos até que nenhuma tarefa precise dos motores. Então, tomamos o controle atribuindo o valor 1 para `sem`. Agora, podemos controlar os motores. Quando terminamos, atribuímos novamente o valor 0 a `sem`. A seguir você encontrará o programa anterior, implementado usando um sinalizador. Quando o sensor de toque detecta algo, o sinalizador é configurado e o procedimento de *backup* acontece. Durante esse procedimento a tarefa `move_square` precisa esperar. No momento que o *backup* está pronto, o sinalizador é configurado para 0 e `move_square` pode continuar.

```
int sem;

task move_square()
{
	while (true)
	{
		until (sem == 0); sem = 1;
		OnFwd(OUT_AC, 75);
		sem = 0;
		Wait(1000);
		until (sem == 0); sem = 1;
		OnRev(OUT_C, 75);
		sem = 0;
		Wait(850);
	}
}

task submain()
{
	SetSensor(IN_1, SENSOR_TOUCH);
	while (true)
	{
		if (SENSOR_1 == 1)
		{
			until (sem == 0); sem = 1;
			OnRev(OUT_AC, 75); Wait(500);
			OnFwd(OUT_A, 75); Wait(850);
			sem = 0;
		}
	}
}

task main()
{
	sem = 0;
	Precedes(move_square, submain);
}
```

Você pode argumentar que não é necessário configurar o semáforo para 1 e depois para 0. Mesmo assim, isso é útil. A razão é que o comando `OnFwd()` é, na verdade, a união de dois comandos (veja Capítulo VIII). Você não quer que essa sequência de comandos seja interrompida por outra tarefa.

Sinalizadores são muito úteis e, quando você está escrevendo programas complicados com tarefas paralelas, eles são quase sempre necessários. (Existe uma pequena chance que eles venham a falhar. Tente descobrir o porquê).

Sumário

Neste capítulo estudamos alguns dos problemas que podem ocorrer quando se usa diferentes tarefas. Sempre seja muito cuidadoso devido aos efeitos colaterais. Muitos comportamentos não esperados acontecem por causa disso. Vimos duas maneiras de resolver tais problemas. A primeira solução pausa e reinicia tarefas para ter certeza que apenas uma tarefa crítica está rodando a cada momento. A segunda aproximação usa sinalizadores para controlar a execução de tarefas. Isso garante que a cada momento apenas a parte crítica de uma tarefa seja executada.

XI. Comunicação entre robôs

Se você possui mais de um NXT este capítulo é para você (caso tenha apenas um NXT, você também pode fazê-lo se comunicar com o PC). Robôs podem se comunicar uns com os outros via *Bluetooth*: você pode ter múltiplos robôs colaborando (ou lutando uns com os outros), e você pode construir um robô grande e complexo usando dois NXTs, assim você pode usar seis motores e oito sensores.

Para os bons e velhos RCX, é simples: ele manda uma mensagem em infravermelho e todos os robôs por perto a recebem.

Para NXT é uma coisa totalmente diferente! Primeiro, você precisa conectar dois ou mais NXTs (ou NXT no PC) com o menu de Bluetooth do NXT; só assim você pode enviar mensagens a dispositivos conectados.

O NXT que começa a conexão se chama *Master* (mestre) e pode ter até três dispositivos *Slave* (escravos) conectados nas linhas 1,2,3; *Slaves* sempre enxergam o *Master* conectado na linha 0. Você pode enviar mensagens para 10 caixas de correio disponíveis.

Mensagens Master – Slave

Dois programas serão mostrados, um para o *master* e um para o *slave*. Esses programas básicos irão lhe ensinar o quão um fluxo rápido e contínuo de mensagens *string* podem ser gerenciados por uma rede *wireless* de dois NXT.

O programa *master* primeiro checa se o *slave* está corretamente conectado à linha 1 usando a função `BluetoothStatus(conn)`; então cria e envia mensagens com um prefixo `M` e um numero crescente com `SendRemoteString(conn,queue,string)`, enquanto recebe mensagens do *slave* com `ReceiveRemoteString(queue,clear,string)` e mostra os dados.

```
//MASTER
#define BT_CONN 1
#define INBOX 1
#define OUTBOX 5

sub BTCheck(int conn){
	if (!BluetoothStatus(conn)==NO_ERR){
		TextOut(5,LCD_LINE2,"Error");
		Wait(1000);
		Stop(true);
	}
}

task main(){
	string in, out, iStr;
	int i = 0;
	BTCheck(BT_CONN); //checa a conexão com o master
	while(true){
		iStr = NumToStr(i);
		out = StrCat("M",iStr);
		TextOut(10,LCD_LINE1,"Master Test");
		TextOut(0,LCD_LINE2,"IN:");
		TextOut(0,LCD_LINE4,"OUT:");
		ReceiveRemoteString(INBOX, true, in);
		SendRemoteString(BT_CONN,OUTBOX,out);
		TextOut(10,LCD_LINE3,in);
		TextOut(10,LCD_LINE5,out);
		Wait(100);
		i++;
	}
}
```

O programa *slave* é muito similar, mas usa `SendResponseString(queue,string)`ao invés de `SendRemoteString` porque o *slave* apenas pode enviar mensagens a seu *master*, enxergado na linha 0.

```
//SLAVE
#define BT_CONN 1
#define INBOX 5
#define OUTBOX 1

sub BTCheck(int conn){
	if (!BluetoothStatus(conn)==NO_ERR){
		TextOut(5,LCD_LINE2,"Error");
		Wait(1000);
		Stop(true);
	}
}

task main(){
	string in, out, iStr;
	int i = 0;
	BTCheck(0); // checa a conexão com o master
	while(true){
		iStr = NumToStr(i);
		out = StrCat("S",iStr);
		TextOut(10,LCD_LINE1,"Slave Test");
		TextOut(0,LCD_LINE2,"IN:");
		TextOut(0,LCD_LINE4,"OUT:");
		ReceiveRemoteString(INBOX, true, in);
		SendResponseString(OUTBOX,out);
		TextOut(10,LCD_LINE3,in);
		TextOut(10,LCD_LINE5,out);
		Wait(100);
		i++;
	}
}
```

Você notará que abortando um dos programas, o outro continuará enviando mensagens com números crescentes, sem saber que todas as mensagens enviadas serão perdidas, porque nenhum está ouvindo do outro lado. Para evitar este problema, podemos planejar um protocolo melhor, com notificação de entrega.

Enviando números com notificação

Aqui veremos outro par de programas; desta vez o *master* envia números com `SendRemoteNumber(conn,queue,number)` e pára, esperando a notificação do *slave* (ciclo `until`, dentro do qual encontramos `ReceiveRemoteString`); apenas se o *slave* estiver recebendo e enviando notificações é que o mestre prosseguirá enviando a próxima mensagem. O *slave* simplesmente recebe um número com `ReceiveRemoteNumber(queue,clear,number)` e envia a notificação com `SendResponseNumber`. Seus programas *master*-*slave* precisam aceitar um código em comum para as notificações, neste caso, escolhi o número hexadecimal 0xFF.

O mestre envia números aleatórios e espera a notificação do *slave*; toda vez que ele recebe uma notificação com o código certo, a variável de notificação precisa ser limpa, doutro modo o mestre continuará enviando sem nenhuma notificação nova, pois a variável está suja.

O *slave* checa continuamente a caixa de entrada e, se não está vazia, mostra o valor lido e envia uma notificação ao mestre. No início do programa, eu escolhi enviar uma notificação mesmo sem ter lido mensagens para desbloquear o mestre; de fato, sem esse truque, caso o programa *master* inicialize primeiro, ele travaria mesmo se inicializássemos o *slave* depois. Desse jeito as primeiras mensagens são pedidas, mas você pode iniciar os programas *master* e *slave* em momentos diferentes sem risco de travamento.

```
//MASTER
#define BT_CONN 1
#define OUTBOX 5
#define INBOX 1
#define CLEARLINE(L) \
TextOut(0,L,"                    ");

sub BTCheck(int conn){
	if (!BluetoothStatus(conn)==NO_ERR){
	TextOut(5,LCD_LINE2,"Error");
	Wait(1000);
	Stop(true);
	}
}

task main(){
	int ack;
	int i;
	BTCheck(BT_CONN);
	TextOut(10,LCD_LINE1,"Master enviando");
	while(true){
		i = Random(512);
		CLEARLINE(LCD_LINE3);
		NumOut(5,LCD_LINE3,i);
		ack = 0;
		SendRemoteNumber(BT_CONN,OUTBOX,i);
		until(ack==0xFF) {
			until(ReceiveRemoteNumber(INBOX,true,ack) == NO_ERR);
		}
		Wait(250);
	}
}
```

```
//SLAVE
#define BT_CONN 1
#define OUT_MBOX 1
#define IN_MBOX 5

sub BTCheck(int conn){
	if (!BluetoothStatus(conn)==NO_ERR){
		TextOut(5,LCD_LINE2,"Error");
		Wait(1000);
		Stop(true);
	}
}

task main(){
	int in;
	BTCheck(0);
	TextOut(5,LCD_LINE1,"Slave recebendo");
	SendResponseNumber(OUT_MBOX,0xFF); //desbloqueia o master
	while(true){
		if (ReceiveRemoteNumber(IN_MBOX,true,in) != STAT_MSG_EMPTY_MAILBOX) {
		TextOut(0,LCD_LINE3," ");
		NumOut(5,LCD_LINE3,in);
		SendResponseNumber(OUT_MBOX,0xFF);
		}
		Wait(10); //take breath (optional)
	}
}
```

Comandos diretos

Existe outra particularidade interessante na comunicação *Bluetooth*: o *master* pode controlar seus *slaves* diretamente.

No próximo exemplo, o *master* envia ao *slave* comandos diretos para tocar músicas e mover um motor; não há necessidade para um programa *slave*, já que o firmware do NXT *slave* foi feito para receber e gerenciar mensagens!

```
//MASTER
#define BT_CONN 1
#define MOTOR(p,s) RemoteSetOutputState(BT_CONN, p, s, \
       OUT_MODE_MOTORON+OUT_MODE_BRAKE+OUT_MODE_REGULATED, \
       OUT_REGMODE_SPEED, 0, OUT_RUNSTATE_RUNNING, 0)

sub BTCheck(int conn){
       if (!BluetoothStatus(conn)==NO_ERR){
              TextOut(5,LCD_LINE2,"Error");
              Wait(1000);
              Stop(true);
       }
}

task main(){
       BTCheck(BT_CONN);
       RemotePlayTone(BT_CONN, 4000, 100);
       until(BluetoothStatus(BT_CONN)==NO_ERR);
       Wait(110);
       RemotePlaySoundFile(BT_CONN, "! Click.rso", false);
       until(BluetoothStatus(BT_CONN)==NO_ERR);
       //Wait(500);
       RemoteResetMotorPosition(BT_CONN,OUT_A,true);
       until(BluetoothStatus(BT_CONN)==NO_ERR);
       MOTOR(OUT_A,100);
       Wait(1000);
       MOTOR(OUT_A,0);
}
```

Sumário

Neste capítulo estudamos alguns dos aspectos básicos de comunicação Bluetooth entre robôs: conectando dois *NXTs*, enviando e recebendo *strings*, números e esperando por notificações de entrega. O último aspecto é muito importante quando é preciso um protocolo de comunicação seguro.

Como particularidade extra, você também aprendeu como enviar comandos diretos ao NXT *slave*.

XII. Mais comandos

O NXC possui vários comandos adicionais. Neste capítulo discutiremos três tipos: o uso de um *timer* (temporizador), comandos para controlar o *display* e o uso do sistema de arquivos do NXT.

Timers

Há um *timer* no NXT que roda continuamente. O *timer* conta em incrementos de 1/1000 de segundo. Você pode conseguir o valor atual do *timer* com o comando `CurrentTick()`. Aqui vai um exemplo de um uso do *timer*. O seguinte programa faz com que o robô se mova aleatoriamente por 10 segundos.

```
task main()
{
	long t0, time;
	t0 = CurrentTick();
	do
	{
		time = CurrentTick()-t0;
		OnFwd(OUT_AC, 75);
		Wait(Random(1000));
		OnRev(OUT_C, 75);
		Wait(Random(1000));
	}
	while (time<10000);
	Off(OUT_AC);
}
```

Você pode querer comparar esse programa com o que é dado no Capítulo IV, que faz exatamente a mesma tarefa. O que usa o *timer* é definitivamente mais simples.

Timers são muito úteis como substitutos de um comando `Wait()`. Você pode entrar em *sleep* por uma quantidade particular de tempo dando um *reset* no *timer* e então esperar até que ele atinja um valor particular. Mas, você também pode reagir a outros eventos (por exemplo, sensores) enquanto espera. O seguinte programa é um exemplo disso. Ele deixa o robô se mover até que 10 segundos se passem, ou até que o sensor toque em algo.

```
task main()
{
	long t3;
	SetSensor(IN_1,SENSOR_TOUCH);
	t3 = CurrentTick();
	OnFwd(OUT_AC, 75);
	until ((SENSOR_1 == 1) || ((CurrentTick()-t3) > 10000));
	Off(OUT_AC);
}
```

Não se esqueça de que *timers* funcionam com incrementos de 1/1000 de segundo, assim como o comando **wait**.

Display de Matriz de Pontos

O NXT possui um *display* de matriz de pontos preto e branco, com uma resolução de 100x64 *pixels*. Existem muitas funções API para desenhar *strings* de texto, números, pontos, linhas, retângulos, círculos e até imagens **bitmap** (arquivos **.ric**). O próximo exemplo tenta cobrir todos estes casos. O *pixel* de número (0,0) é o que está mais à esquerda no canto inferior da tela.

```
#define X_MAX 99
#define Y_MAX 63
#define X_MID (X_MAX+1)/2
#define Y_MID (Y_MAX+1)/2

task main(){
	int i = 1234;
	TextOut(15,LCD_LINE1,"Display", true);
	NumOut(60,LCD_LINE1, i);
	PointOut(1,Y_MAX-1);
	PointOut(X_MAX-1,Y_MAX-1);
	PointOut(1,1);
	PointOut(X_MAX-1,1);
	Wait(200);
	RectOut(5,5,90,50);
	Wait(200);
	LineOut(5,5,95,55);
	Wait(200);
	LineOut(5,55,95,5);
	Wait(200);
	CircleOut(X_MID,Y_MID-2,20);
	Wait(800);
	ClearScreen();
	GraphicOut(30,10,"faceclosed.ric"); Wait(500);
	ClearScreen();
	GraphicOut(30,10,"faceopen.ric");
	Wait(1000);
}
```

Todas essas funções são um pouco auto-explicativas, mas eu irei descrever todos os parâmetros com mais detalhes.

`ClearScreen()` limpa a tela;
`NumOut(x, y, número)` imprime um número nas coordenadas especificadas;
`TextOut(x, y, stringdetexto)` funciona como a de cima mas, a saída é uma *string* de texto;
`GraphicOut(x, y, nomedoarquivo)` mostra um arquivo de **bitmap.ric**;
`CircleOut(x, y, raio)` a saída é um circulo especificado pelas coordenadas do centro e o raio;
`LineOut(x1, y1, x2, y2)` desenha uma linha do ponto `(x1,x2)` até o ponto `(x2,y2)`;
`PointOut(x, y)` põe um ponto na tela.
`RectOut(x, y, largura, altura)` desenha um retângulo no vértice inferior esquerdo em (x,y) e com as dimensões especificadas;
`ResetScreen()` Reseta a tela.

Sistema de arquivos.

O NXT pode escrever e ler arquivos, gravados dentro da sua memória *flash*. Você pode salvar um *datalog* dos dados do sensor ou números lidos durante a execução do programa. O único limite do número de arquivos e sua dimensão é o tamanho da memória *flash*. As funções API do NXT deixam que você gerencie arquivos (crie, troque o nome, apague e encontre), e ainda leia ou escreva *strings* de texto, números e *bytes*.

No próximo exemplo, nós veremos como criar um arquivo, escrever *strings* nele e alterar seu nome.

Primeiro, o programa apaga arquivos com nomes que vamos usar: não é um bom hábito (devemos checar a existência do arquivo, apagá-lo manualmente ou escolher automaticamente outro nome para nosso arquivo de trabalho), mas não existe problema algum em nosso caso simples. Ele cria nosso arquivo com `CreateFile("Danny.txt", 512, fileHandle)`, especificando nome, tamanho e um *handle* para o arquivo, onde o *firmware* do NXT escreverá um número para seu próprio uso.

Então, ele cria *strings* e as escreve no arquivo com retorno de carro (o cursor volta para a primeira posição da linha onde se encontra) com o comando `WriteLnString(fileHandle,string, bytesWritten)`, onde todos os parâmetros precisam ser variáveis. Finalmente, o arquivo é fechado e renomeado. Lembre-se: um arquivo deve ser fechado antes de começar outra operação, então se você criou um arquivo, você pode escrever nele. Se quiser ler esse mesmo arquivo, precisa fechá-lo e então abri-lo com `OpenFileRead()`. Para apagar ou trocar seu nome, você precisa fechá-lo.

```
#define OK LDR_SUCCESS

task main(){
	byte fileHandle;
	short fileSize;
	short bytesWritten;
	string read;
	string write;
	DeleteFile("Danny.txt");
	DeleteFile("DannySays.txt");
	CreateFile("Danny.txt", 512, fileHandle);
	for(int i=2; i<=10; i++ ){
		write = "NXT is cool ";
		string tmp = NumToStr(i);
		write = StrCat(write,tmp," times!");
		WriteLnString(fileHandle,write, bytesWritten);
	}
	CloseFile(fileHandle);
	RenameFile("Danny.txt","DannySays.txt");
}
```

Para ver o resultado, vá até BricxCC->Tools->NXT Explorer, faça o *upload* de `DannySays.txt` para o PC e dê uma olhada. Pronto para o próximo exemplo! Nós vamos criar uma tabela de caracteres **ASCII**.

```
task main(){
	byte handle;
	if (CreateFile("ASCII.txt", 2048, handle) == NO_ERR) {

		for (int i=0; i < 256; i++) {
			string s = NumToStr(i);
			int slen = StrLen(s);
			WriteBytes(handle, s, slen);
			WriteLn(handle, i);
		}
		CloseFile(handle);
	}
}
```

Muito simples: o programa cria um arquivo e, se nenhum erro ocorrer, ele escreve um número variando de 0 até 255 (convertendo para *string* antes) com `WriteBytes(handle, s, slen)`, que é outra forma de escrever *strings* sem retorno de carro. Então, ele escreve um número como em `WriteLn(handle, value)`, que anexa um retorno de carro. O resultado, que você pode ver abrindo `ASCII.txt` com um editor de texto (como o **Notepad** do **Windows**), é explicável: o número escrito como *string* é mostrado de forma compreensível para humanos, enquanto números escritos como valores hexadecimais são interpretados e escritos em código ASCII.

Duas funções importantes faltam ser explicadas: `ReadLnString` para ler *strings* de arquivos e `ReadLn` para ler números.

Para exemplificar a primeira função: a tarefa **main** do programa a seguir chama a rotina `CreateRandomFile` que cria um arquivo com números randômicos dentro (escritos como **strings**); você pode comentar essa linha e usar outro arquivo de texto criado a mão, por exemplo.

Então a tarefa **main** abre o arquivo para leitura, lê uma linha por vez até o fim do arquivo, chamando a função `ReadLnString` e mostra o texto.

Na sub-rotina `CreateRandomFile` nós geramos uma quantidade pré-definida de números aleatórios, os convertemos para **string** e então os escrevemos no arquivo.

A função `ReadLnString` aceita um **handle** de arquivo e uma variável **string** como argumentos: depois de chamar, a **string** conterá uma linha de texto e a função irá retornar um código de erro que nós podemos usar para saber se o arquivo chegou ao fim.

```
#define FILE_LINES 10

sub CreateRandomFile(string fname, int lines){
	byte handle;
	string s;
	int bytesWritten;
	DeleteFile(fname);
	int fsize = lines*5;
	//cria um arquivo com dados aleatórios
	if(CreateFile(fname, fsize, handle) == NO_ERR) {
		int n;
		repeat(FILE_LINES) {
			int n = Random(0xFF);
			s = NumToStr(n);
			WriteLnString(handle,s,bytesWritten);
		}
		CloseFile(handle);
	}
}

task main(){
	byte handle;
	int fsize;
	string buf;
	bool eof = false;
	CreateRandomFile("rand.txt",FILE_LINES);
	if(OpenFileRead("rand.txt", fsize, handle) == NO_ERR) {
		TextOut(10,LCD_LINE2,"Filesize:");
		NumOut(65,LCD_LINE2,fsize);
		Wait(600);
		until (eof == true){ // lê o arquivo de texto até o fim
			if(ReadLnString(handle,buf) != NO_ERR) eof = true;
			ClearScreen();
			TextOut(20,LCD_LINE3,buf);
			Wait(500);
		}
	}
	CloseFile(handle);
}
```

No último programa, eu te mostrarei como ler números de um arquivo.
Tomarei a ocasião para te dar uma pequena amostra de compilação condicional. No início do código, há uma definição que não é usada para um macro nem para um pseudônimo: simplesmente definimos como **INT**.

Então, há um **statement** de pré-processamento

```
#ifdef INT
…Code…
#endif
```

que simplesmente diz ao compilador para compilar o código entre os dois **statements** se INT foi definido previamente. Então, se definimos **INT**, a tarefa **main** dentro da primeira parelha será compilada e se **LONG** for definido em vez de **INT**, a segunda versão do **main** será compilada.

Esse método me permite mostrar em apenas um programa como ambos os tipos `int` (16 bits) e `long` (32 bits) podem ser lidos de um arquivo ao chamar a mesma função `ReadLn(handle,val)`.

Como antes, ele aceita um **handle** de arquivo e uma variável numérica como argumentos, retornando um código de erro.

A função lerá 2 bytes do arquivo se a variável passada for declarada como `int`, e irá ler 4 bytes se a variável for `long`. Varíaveis `bool` também podem ser escritas e lidas do mesmo modo.

```
#define INT // INT ou LONG

#ifdef INT
task main () {
	byte handle, time = 0;
	int n, fsize,len, i;
	int in;
	DeleteFile("int.txt");
	CreateFile("int.txt",4096,handle);
	for (int i = 1000; i<=10000; i+=1000){
		WriteLn(handle,i);
	}
	CloseFile(handle);
	OpenFileRead("int.txt",fsize,handle);
	until (ReadLn(handle,in)!=NO_ERR){
		ClearScreen();
		NumOut(30,LCD_LINE5,in);
		Wait(500);
	}
	CloseFile(handle);
}
#endif

#ifdef LONG
task main () {
	byte handle, time = 0;
	int n, fsize,len, i;
	long in;
	DeleteFile("long.txt");
	CreateFile("long.txt",4096,handle);
	for (long i = 100000; i<=1000000; i+=50000){
		WriteLn(handle,i);
	}
	CloseFile(handle);
	OpenFileRead("long.txt",fsize,handle);
	until (ReadLn(handle,in)!=NO_ERR){
		ClearScreen();
		NumOut(30,LCD_LINE5,in);
		Wait(500);
	}
	CloseFile(handle);
}
#endif
```

Sumário

Neste último capítulo você aprendeu recursos avançados oferecidos pelo NXT: *timer* de alta resolução, *display* de matriz de pontos e o sistema de arquivos.

XIII. Lembretes finais

Se você praticou do início ao fim deste tutorial, você pode se considerar um *expert* em NXC. Caso não tenha feito isso até agora, é hora de começar a experimentar você mesmo. Com criatividade em projeto e programação, você pode fazer com que os robôs Lego façam coisas inacreditáveis.

Esse tutorial não cobriu todos os aspectos do BricxCC. Recomendo que leia o Guia NXC em todos os capítulos. O NXC ainda está em desenvolvimento, e uma versão futura pode incorporar alguma funcionalidade adicional. Muitos conceitos de programação não foram tratados nesse tutorial. Em particular, nós não consideramos comportamento de aprendizado dos robôs ou outros aspectos de inteligência artificial.

Também é possível controlar um robô lego diretamente do PC. Para isso você precisa escrever o programa em uma linguagem como C++, Visual Basic, Java ou Delphi. Também é possível fazer com que tal programa funcione conjuntamente com um programa do NXC rodando no próprio NXT. Tal combinação é muito poderosa. Se você estiver interessado nessa forma de programar o seu robô, é melhor começar fazendo o *download* do Fantom SDK e documentos *Open Source* da seção NXTreme do website da Lego MindStorms.

http://mindstorms.lego.com/Overview/NXTreme.aspx

A web é uma fonte perfeita de informações adicionais. Alguns outros pontos para se começar estão no LUGNET, a Rede de Grupos de Usuários da LEGO® (não-oficial):

http://www.lugnet.com/robotics/nxt