Programando Robôs Lego com NQC

(Versão 3.03, 2 de Outubro de 1999)

por Mark Overmars

Department of Computer Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands

Traduzido para a língua portuguesa por Anderson Grandi Pires em 23 de março de 2008
Professor do Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Campus III – Leopoldina – Minas Gerais – Brasil

Prefácio

Os kits de robôs Lego Mindstorms e CyberMaster são novos maravilhosos brinquedos em que uma ampla variedade de robôs podem ser construídos e programados para fazer todo tipo de tarefa complicada. Infelizmente, o software distribuído juntamente com tais kits é, apesar de visualmente atrativo, bastante limitado no que diz respeito à funcionalidade. Desse modo, tal software pode ser utilizado somente para tarefas simples. Para explorar todo o potencial dos robôs, faz-se necessário um ambiente de programação diferente. NQC é uma linguagem de programação, escrita por Dave Baum, projetada especialmente para os robôs Lego. Caso você nunca tenha escrito um programa antes, não se preocupe. NQC é realmente muito fácil de usar e este tutorial o ensinará tudo a este respeito. Atualmente, programar robôs com NQC é muito mais fácil do que programar um computador normal, sendo essa uma maneira fácil de se tornar um programador.

Para tornar a tarefa de programar realmente fácil, existe o RCX Command Center. Este utilitário o ajuda a escrever seus programas, enviá-los aos robôs, iniciar e parar os robôs. RCX Command Center funciona quase como um processador de textos, porém com algumas funcionalidades extras. Este tutorial usará o RCX Command Center (versão 3.0 ou superior) como ambiente de programação. Você pode efetuar o download desse utilitário, livremente, a partir do seguinte endereço eletrônico:

http://www.cs.uu.nl/people/markov/lego/

RCX Command Center roda sobre a plataforma Windows (versão 95, 98 e NT). (Você deve executar o software que vem com o kit Lego Mindstorms pelo menos uma vez, antes de executar o RCX Command Center. O software que acompanha o kit instala certos componentes que o RCX Command Center utiliza.) A linguagem NQC pode também ser usada sobre outras plataformas. Você pode baixá-lo na internet no endereço:

http://www.enteract.com/~dbaum/lego/nqc/

A maior parte deste tutorial também se aplica a outras plataformas (assumindo que se esteja usando o NQC versão 2.0 ou superior), exceto pelo fato de que algumas ferramentas são perdidas, além da perda da funcionalidade de realce de cor no código-fonte.

Neste tutorial eu assumo que você tenha o kit de robôs Lego Mindstorms. A maior parte dos conteúdos também se aplica ao kit de robôs CyberMaster, contudo algumas funcionalidades não se encontram disponíveis para este último. Além disso, para o kit de robôs CyberMaster, os nomes dos motores (por exemplo) são diferentes, o que torna necessário modificar os exemplos um pouco para fazê-los funcionar.

Agradecimentos

Eu gostaria de agradecer a Dave Braum por ter desenvolvido o NQC. Gostaria de agradecer também a Kevin Saddi por ter escrito a primeira versão da primeira parte deste tutorial.

Sumário

I. Escrevendo o seu primeiro programa

Neste capítulo você verá como escrever um programa extremamente simples. Iremos programar um robô para executar as seguintes tarefas: mover para frente durante 4 segundos; mover para trás outros 4 segundos; e então parar. Nada muito espetacular, mas isso o familiarizará com idéias básicas da programação. Além disso, mostrar-lhe-á o quando fácil isto é. Mas, antes de escrevermos um programa, precisaremos de um robô.

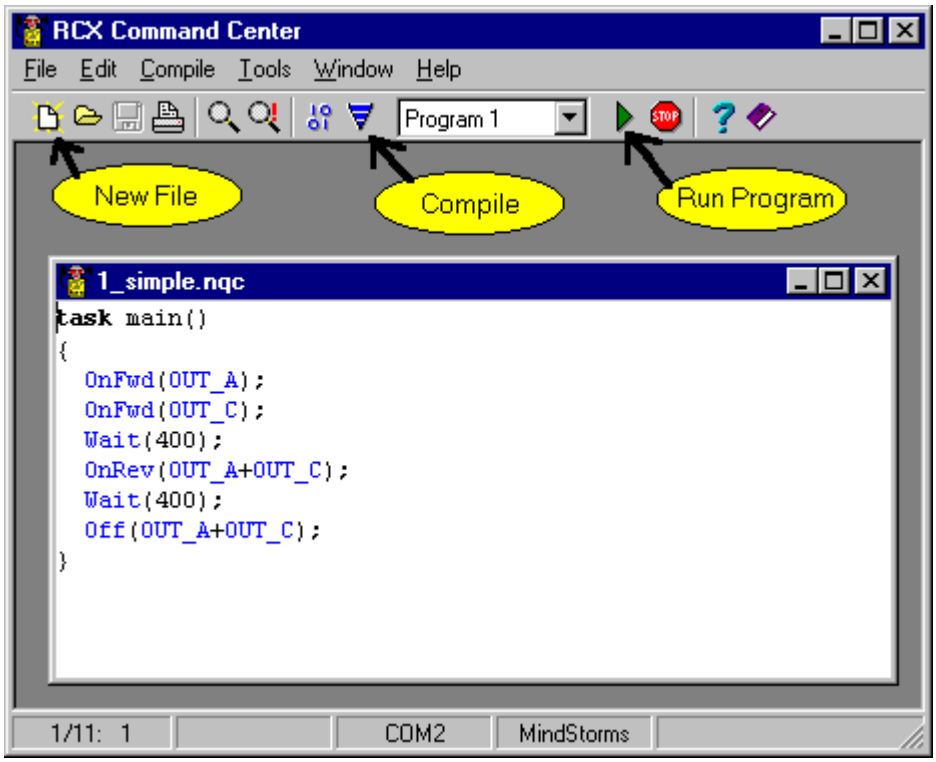
Construindo um robô

O robô que utilizaremos por todo este tutorial é uma versão simplificada do robô *top-secret* que é descrito nas páginas 39 a 46 do material contructopedia que acompanha o kit. Utilizaremos somente a base (chassi). Remova toda a frente, incluindo os dois braços e os sensores de toque. Além disso, conecte os motores ligeiramente diferentes, de modo que os cabos conectados no RCX fiquem dispostos lateralmente. Isto é importante para que o robô mova na direção correta. O robô parecerá tal como a figura abaixo:

Tenha certeza de que a torre de transmissão de dados (*Infrared Tower*) esteja conectada corretamente ao computador e que esteja configurada para longo alcance (*long range*). (Você pode querer checar com o software Robotics Invention System — RIS — se o robô está funcionando adequadamente)

Iniciando o RCX Command Center

Escreveremos nossos programas utilizando o RCX Command Center. Inicie-o por intermédio de um clique duplo no ícone RcxCC. (Eu presumo que você já tenha instalado o RCX Command Center. Caso não, faça o download da internet — veja o endereço eletrônico no prefácio — e o descompacte-o em qualquer diretório que você queira.) O programa tentará localizar o robô. Ligue o robô e pressione **OK**. O programa encontrará o robô automaticamente (é esperado que isso ocorra). A interface do usuário se apresenta conforme abaixo (sem a janela central).



A interface é parecida com aquelas existentes em editores de texto, com os menus usuais e botões para abrir, salvar, imprimir, editar arquivos, etc. Além disso, existem menus especiais para compilar e descarregar os programas nos robôs, além daqueles disponíveis para se obter informações a respeito do robô. Você pode ignorar os últimos menus por enquanto.

Iremos escrever um novo programa. Pressione o botão **New File** para criar uma nova janela vazia.

Escrevendo o programa

Digite as instruções abaixo no novo programa:

```
task main()
{
  OnFwd(OUT_A);
  OnFwd(OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Pode parecer um pouco complicado a princípio, então vamos analisá-lo. Programas em NQC consistem de tarefas (*task*). Nosso programa tem somente uma tarefa de nome `main` (tarefa principal). Cada programa necessita de uma tarefa nomeada `main`, que será automaticamente executada pelo robô. Você aprenderá mais a respeito de tarefas no Capítulo VI. Uma tarefa consiste de um conjunto de comandos, também referenciados como instruções (*statements*). Existem chaves em torno das instruções, o que deixa claro que aquelas instruções pertencem a uma determinada tarefa (Um par de chaves define um bloco de comandos). Cada instrução finaliza com um ponto-e-vírgula. Desse modo, torna-se claro onde uma instrução finaliza e onde a próxima começa. De maneira geral, uma tarefa se parece com a seguinte estrutura:

```
task main()            //tarefa principal
{
  statement1;          //primeira instrução
  statement2;          //segunda instrução
    …
}
```

Nosso programa (página anterior) tem seis comandos. Daremos uma olhada em cada um destes:

`OnFwd(OUT_A);`
Este comando diz ao robô para habilitar (energizar) a saída A, isto é, o motor conectado à saída A do RCX girará para frente. Ele moverá com a velocidade máxima, a menos que a velocidade seja configurada antes da execução deste comando. Posteriormente veremos como fazer isso.

`OnFwd(OUT_C);`
Semelhante à instrução anterior, porém agora girará o motor C. Após a execução desses dois comandos, ambos os motores estarão girando e o robô se move para frente.

`Wait(400);`
Agora é hora de esperar por um período de tempo. Esta instrução nos diz para aguardar pelo período de 4 segundos. O argumento, isto é, o número entre parênteses, define o número de "ticks". Cada tick corresponde a 1/100 segundos. Então você pode, de maneira bastante precisa, dizer ao programa quanto tempo aguardar. Assim sendo, pelo período de 4 segundos o programa não faz nada, embora o robô continue se movendo para frente.

`OnRev(OUT_A+OUT_C);`
Tendo o robô se movimentado o suficiente para frente, diremos a ele para se mover no sentido reverso, ou seja, para trás. Observe que nós podemos configurar ambos os motores de uma única vez, utilizando `OUT_A+OUT_C` como argumento. Nós poderíamos ter combinado as duas primeiras instruções desta maneira.

`Wait(400);`
Novamente, esperamos por 4 segundos.

`Off(OUT_A+OUT_C);`
E, finalmente, desligamos ambos os motores.

Este é todo o programa. Ele move os motores para frente por 4 segundos, move os motores para trás outros 4 segundos e finalmente os desliga.

Você provavelmente observou as cores quando estava digitando o programa. Elas aparecem automaticamente. Tudo que está em azul são comandos para o robô, ou uma indicação de um motor ou outra coisa que o robô conhece. A palavra **`task`** está em negrito por ser uma importante palavra-chave (reservada) em NQC. Outras importantes palavras aparecem em negrito, conforme veremos a seguir. As cores são úteis para mostrar que você não cometeu algum erro enquanto estava digitando o programa.

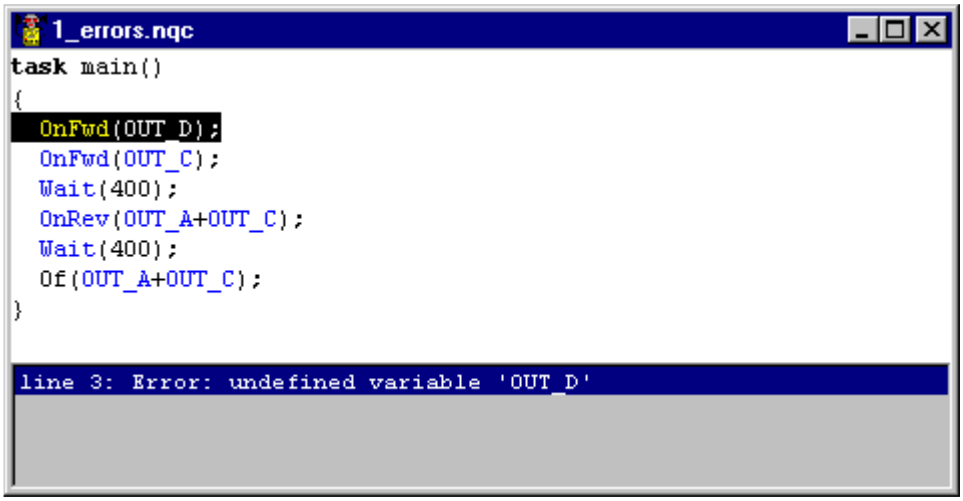
Executando o programa

Uma vez que um programa tenha sido escrito, ele precisa ser compilado (isto é, transformado em um código que o robô possa entender e executar) e enviado para o robô usando a torre de transmissão de dados (comumente denominado de processo de "descarregar" o programa). Existe um botão que efetua as duas tarefas (veja a figura acima). Pressione este botão e, assumindo que não foram inseridos erros na digitação, o programa irá compilar corretamente e ser descarregado no robô. (Caso existam erros no programa, tais erros serão notificados; veja abaixo.)

Agora você pode executar o programa. Para isso, pressione o botão verde (run) do RCX ou, de maneira mais fácil, pressione o botão **run** da janela do RCX Command Center (ver figura acima). O robô faz aquilo que é esperado? Caso não, provavelmente os cabos foram conectados de maneira errada.

Erros no programa

Quando estamos digitando um código de um programa existe a possibilidade de cometermos erros de digitação. O compilador verifica os erros e os relata na parte inferior da janela do RCX Command Center, conforme apresentado abaixo:



Ele automaticamente seleciona o erro (cometemos um equívoco ao digitar o nome do motor). Quando existem mais erros, você pode clicar nas mensagens de erros para localizá-los. Observe que erros existentes no início de um programa podem acarretar em erros em outros locais do código. Então, o melhor a fazer é corrigir alguns erros e compilar o programa novamente. Observe, também, que a cor associada às instruções de código ajuda bastante a evitar erros. Por exemplo, na última linha foi digitado `Of` ao invés de `Off`. Devido ao fato deste comando ser desconhecido, o mesmo não se apresenta na cor azul.

Existem alguns erros que não são identificados pelo compilador. Caso tenhamos digitado `OUT_B` isso teria passado despercebido devido ao fato de que tal saída existe (mesmo sabendo que nós não a usamos no robô). Então, caso o robô exiba um comportamento inesperado, existe ainda algum erro no seu programa.

Modificando a velocidade

Como observado, o robô se move de maneira bastante rápida. Por padrão, o robô se move tão rápido quanto ele pode. Para modificar esta velocidade você pode utilizar o comando `SetPower()`. A potência é um número entre 0 e 7. O número 7 representa a velocidade mais rápida, enquanto que 0 representa a mais devagar (mas o robô ainda se move). Aqui está uma nova versão do nosso programa nos quais o robô se move lentamente:

```
task main()
{
  SetPower(OUT_A+OUT_C,2);
  OnFwd(OUT_A+OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Resumo

Neste capítulo você escreveu seu primeiro programa em NQC, usando o RCX Command Center. Até este ponto, você aprendeu a escrever um programa, descarregá-lo no robô e colocá-lo em execução. O software RCX Command Center pode fazer outras coisas mais. Para descobrir, leia a documentação que acompanha o software. Este tutorial é referente, principalmente, à linguagem NQC e somente menciona características do RCX Command Center quando realmente forem necessárias.

Você também aprendeu alguns importantes aspectos da linguagem NQC. Inicialmente, você aprendeu que cada programa tem uma tarefa denominada `main`, que é sempre executada pelo robô. Depois você aprendeu os quatro comandos referentes a motores mais importantes: `OnFwd()`, `OnRev()`, `SetPower()` e `Off()`. Por fim, você aprendeu a respeito do comando `Wait()`.

II. Um programa mais interessante

Nosso primeiro programa não era tão espetacular. Então, deixe-nos tentar fazer algo mais interessante. Faremos isso em vários passos, introduzindo algumas características importantes da linguagem de programação NQC.

Fazendo curvas

Você pode instruir o robô a fazer curvas por meio da paralisação ou da rotação reversa de um dos dois motores. Abaixo temos um exemplo. Digite e compile o código, transfira-o para o robô e então execute. Ele deve se mover para frente um pouco e então girar em um ângulo de 90 graus.

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  OnRev(OUT_C);
  Wait(85);
  Off(OUT_A+OUT_C);
}
```

Pode acontecer de você ter que tentar alguns números próximos de 85 na segunda chamada ao comando `Wait()` para fazer com que o robô gire precisamente em 90 graus. Isto depende do tipo de superfície nos quais o robô está sendo utilizado. Ao invés de modificar isto no corpo da tarefa `main`, é mais fácil atribuir um nome para este número. Em NQC você pode definir valores constantes como mostrado no seguinte programa:

```
#define TEMPO_DE_MOVIMENTACAO       100
#define TEMPO_DE_CURVATURA          85

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(TEMPO_DE_MOVIMENTACAO);
  OnRev(OUT_C);
  Wait(TEMPO_DE_CURVATURA);
  Off(OUT_A+OUT_C);
}
```

As duas primeiras linhas definem duas constantes. A partir da definição, as constantes poderão ser utilizadas por todo o programa. Definir constantes é bom por duas razões: torna o programa mais legível e facilita as mudanças de valores. Observe que o RCX Command Center estipula uma cor própria para as instruções `define`. Como será visto no Capítulo VI, você poderá definir outras coisas além de constantes.

Repetindo comandos

Vamos agora escrever um programa que faz o robô se mover em um caminho em forma de um quadrado. Mover-se em forma de um quadrado significa: mover para frente, girar 90 graus, mover para frente de novo, girar 90 graus, etc. Nós poderíamos repetir o pedaço de código acima quatro vezes, mas isto pode ser feito de maneira mais fácil com a estrutura de repetição **`repeat`**.

```
#define TEMPO_DE_MOVIMENTACAO       100
#define TEMPO_DE_CURVATURA          85

task main()
{
  repeat(4)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(TEMPO_DE_MOVIMENTACAO);
    OnRev(OUT_C);
    Wait(TEMPO_DE_CURVATURA);
  }
  Off(OUT_A+OUT_C);
}
```

O número entre parênteses na instrução **repeat** indica com que freqüência as coisas deverão ser repetidas. As instruções que devem ser repetidas são colocadas entre chaves (definindo o bloco de comandos associado à instrução `repeat`), semelhantemente ao que se faz em uma tarefa. Observe no programa acima que também endentamos as instruções (inserção de espaços na margem esquerda). Isso não é necessário, mas torna o programa mais legível.

Como um exemplo final, faremos o robô se movimentar 10 vezes em um caminho em forma de um quadrado. Aqui está o programa:

```
#define TEMPO_DE_MOVIMENTACAO       100
#define TEMPO_DE_CURVATURA          85

task main()
{
  repeat(10)
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(TEMPO_DE_MOVIMENTACAO);
      OnRev(OUT_C);
      Wait(TEMPO_DE_CURVATURA);
    }
  }
  Off(OUT_A+OUT_C);
}
```

Existem agora duas instruções `repeat`, uma dentro da outra. Nós chamamos isso de instruções `repeat` aninhadas. Você pode aninhar quantas instruções `repeat` você desejar. Dê uma olhada criteriosa nas chaves e na endentação utilizada no programa. A tarefa principal inicia na primeira abertura de chaves e finaliza na última chave. A primeira instrução `repeat` inicia na segunda abertura de chaves e finalize na quinta chave. A segunda instrução `repeat` inicia na terceira chave e finalize na quarta. Como se pode ver, as chaves sempre aparecem em pares e as instruções existentes entre as chaves são endentadas.

Incluindo comentários

Uma boa maneira para tornar seu programa mais legível é por meio da inclusão de comentários. Caso você coloque `//` (duas barras) em uma linha, todo código inserido após estes símbolos (na mesma linha) serão ignorados e poderão ser utilizados como comentários. Um comentário mais longo poderá ser incluído entre os símbolos `/*` e `*/` (comentário de múltiplas linhas). No RCX Command Center os comentários se apresentam na cor verde. O programa completo se parece conforme apresentado a seguir:

```
/*  10 QUADRAS

    por Mark Overmars

Este programa faz o robô se movimentar por 10 quadras
*/

#define TEMPO_DE_MOVIMENTACAO   100 //Tempo de movimentação para frente
#define TEMPO_DE_CURVATURA      85  //Tempo de curvatura em 90 graus

task main()
{
  repeat(10)                // Percorre as 10 quadras
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(TEMPO_DE_MOVIMENTACAO);
      OnRev(OUT_C);
      Wait(TEMPO_DE_CURVATURA);
    }
  }
  Off(OUT_A+OUT_C);         // Agora, desliga os motores
}
```

Resumo

Neste capítulo você aprendeu a usar a instrução **repeat** e inserir comentários. Além disso, você viu o propósito de se utilizar chaves aninhadas e o uso de endentação. Com tudo o que foi aprendido, você pode fazer o robô se mover por todo tipo de caminho. Um bom exercício seria tentar escrever algumas variações dos programas deste capítulo antes de avançar para o próximo.

III. Usando variáveis

Variáveis compreendem um importante aspecto de toda linguagem de programação. Variáveis consistem de localizações na memória onde podemos armazenar valores. Podemos usar o valor armazenado em diferentes lugares, além de podermos modificar tal valor. Deixe-me descrever o uso de variáveis por intermédio de um exemplo.

Movendo em forma de uma espiral

Vamos assumir que queremos adaptar o programa acima de tal maneira que o robô percorra um caminho em forma de espiral. Isso pode ser feito por meio de um aumento no tempo em que esperamos (Wait) em cada movimento posterior. Isto é, nós queremos aumentar o valor de do `TEMPO_DE_MOVIMENTACAO` cada vez que as instruções associadas à `repeat` forem executadas. Mas como fazer isso? `TEMPO_DE_MOVIMENTACAO` é uma constante e constantes não podem ser modificadas. Necessitamos, então, de uma variável. Variáveis podem ser facilmente definidas em NQC. Você pode ter 32 variáveis e dar a cada uma delas um nome diferente. A seguir é apresentado o programa espiral.

```
#define TEMPO_DE_CURVATURA 85

int tempo_de_movimentacao;        // declara uma variável

task main()
{
  tempo_de_movimentacao = 20;     // inicializa a variável
  repeat(50)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(tempo_de_movimentacao);  // utiliza a variável para esperar
    OnRev(OUT_C);
    Wait(TEMPO_DE_CURVATURA);
    tempo_de_movimentacao += 5;   // incrementa a variável
  }
  Off(OUT_A+OUT_C);
}
```

As linhas interessantes estão indicadas com comentários. Primeiro definimos uma variável por intermédio da utilização da palavra-chave **`int`**, seguida por um nome que nós mesmos escolhemos. (Normalmente usamos letras minúsculas para nomes de variáveis e letras maiúsculas para nomes de constantes, mas não é necessário.) O nome deve iniciar com uma letra, mas pode conter números e o caractere especial sublinhado (_). Nenhum outro símbolo é permitido. (O mesmo se aplica a constantes, nomes de tarefas, etc.) A palavra-chave em negrito **`int`** significa inteiro. Somente números inteiros podem ser armazenados nesse tipo de variável. Na segunda linha interessante nós inicializamos a variável com o valor 20 (atribuímos o valor 20 à variável). A partir deste ponto, caso você use a variável ela valerá 20. Agora, no corpo de `repeat`, observe que usamos a variável `tempo_de_movimentacao` para indicar o tempo de espera e no final do loop incrementamos o valor da variável em 5 unidades. Assim sendo, na primeira vez o robô espera por 20 ticks, na segunda vez 25, na terceira 30, etc.

Além de adicionar valores para uma variável, podemos multiplicar uma variável por um número utilizando *=, subtrair utilizando `-=` e dividir usando `/=`. (Observe que para a divisão o resultado é arredondado para o número inteiro mais próximo.) Você pode também adicionar uma variável a outra e desenvolver expressões mais complicadas. Temos aqui alguns exemplos:

```
int aaa;
int bbb, ccc;

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3);  // aaa agora é igual a 80
}
```

Observe nas duas primeiras linhas que podemos definir várias variáveis em uma única instrução. Poderíamos, também, ter combinado as três variáveis em uma única linha.

Números randômicos

Em todos os programas acima definimos exatamente o que seria suposto que o robô fizesse. Mas as coisas se tornam mais interessantes quando o robô está apto a fazer coisas que nós não sabemos, ou seja, de maneira imprevisível. Queremos alguma aleatoriedade nas movimentações do robô. Em NQC você pode criar números randômicos, ou seja, aleatórios. O seguinte programa usa isso para deixar o robô se movimentar de maneira aleatória. O robô constantemente se movimenta para frente uma quantidade de tempo aleatória e então efetua um giro também aleatório.

```
int tempo_de_movimentacao, tempo_de_curvatura;

task main()
{
  while(true)
  {
    tempo_de_movimentacao = Random(60);
    tempo_de_curvatura = Random(40);
    OnFwd(OUT_A+OUT_C);
    Wait(tempo_de_movimentacao);
    OnRev(OUT_A);
    Wait(tempo_de_curvatura);
  }
}
```

O programa define duas variáveis e então atribui a elas valores aleatórios. `Random(60)` significa que um número entre 0 e 60 (inclusives) será gerado (0 ou 60 são valores possíveis). A cada chamada a `Random()` os números serão diferentes. (Observe que poderíamos evitar o uso de variáveis ao escrever, por exemplo, `Wait(Random(60))`.)

Você pode observar um novo tipo de loop aqui. Ao invés de utilizar a instrução `repeat`, escrevemos **`while(true)`**. A estrutura de repetição `while` repete as instruções associadas a ela (seu bloco de comandos) enquanto o resultado da condição entre parênteses for verdadeiro. A palavra especial **`true`** é sempre verdadeiro, assim as instruções entre chaves executarão indefinidamente, tanto quanto queiramos. Você aprenderá mais a respeito da instrução `while` no Capítulo IV.

Resumo

Neste capítulo você aprendeu a respeito de variáveis. Variáveis são muito úteis, mas, devido às restrições do robô, elas são um pouco limitadas. Você pode definir somente 32 variáveis e elas podem armazenar somente inteiros. Mas para muitas tarefas realizadas por robôs isso é bom o suficiente.

Você também aprendeu a criar números randômicos, tais que se pode dar ao robô um comportamento imprevisível. Finalmente, vimos o uso da estrutura de repetição `while` para criar um loop infinito que executa indefinidamente.

IV. Estruturas de controle

No capítulo anterior vimos as instruções `repeat` e `while`. Essas instruções controlam a maneira com que outras instruções são executadas em um programa. Elas são chamadas estruturas de "controle". Neste capítulo veremos algumas outras estruturas de controle.

A instrução if

Às vezes você deseja que uma parte específica do seu programa seja executada somente em certas situações. Em casos semelhantes a esse é que a instrução `if` é utilizada. Deixe-me mostrar um exemplo. Iremos modificar novamente o programa que temos utilizado, mas com uma abordagem diferente. Nós queremos que o robô se mova ao longo de uma linha reta e então efetue um giro para a esquerda ou para a direita. Para fazer isso, nós utilizaremos novamente os números randômicos. Nós pegaremos um número randômico entre 0 e 1 (inclusives), isto é, o número 0 ou o número 1. Se o número for 0 efetuaremos um giro para a direita; caso contrário, efetuaremos um giro para a esquerda. Veja o programa:

```
#define  TEMPO_DE_MOVIMENTACAO     100
#define  TEMPO_DE_CURVATURA        85

task main()
{
  while(true)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(TEMPO_DE_MOVIMENTACAO);
    if (Random(1) == 0)
    {
      OnRev(OUT_C);
    }
    else
    {
      OnRev(OUT_A);
    }
    Wait(TEMPO_DE_CURVATURA);
  }
}
```

A instrução `if` se parece um pouco com a instrução `while`. Caso a condição entre parênteses seja verdadeira, a parte entre chaves será executada. Caso falso, a parte entre as chaves após a palavra-chave **`else`** será executada. Vamos olhar um pouco melhor a condição que utilizamos: `Random(1) == 0`. Isso significa que `Random(1)` deve ser igual a 0 para fazer a condição verdadeira. Você poderia pensar por que utilizamos = = ao invés de =. A razão é para diferenciá-lo daquelas instruções que atribuem valores a uma variável. Você pode comparar valores de diferentes maneiras. A seguir são apresentadas as mais importantes:

= =	igual a
<	menor do que
<=	menor ou igual a
>	maior do que
>=	maior ou igual a
!=	diferente (não igual a)

Você pode combinar condições usando `&&`, que significa "e" ou `||`, que significa "ou". Aqui temos alguns exemplos de condições:

`true`	sempre verdadeiro
`false`	nunca verdadeiro (falso)
`ttt != 3`	verdadeiro quanto ttt for diferente de 3
`(ttt >= 5) && (ttt <= 10)`	verdadeiro quando ttt estiver entre 5 e 10 (inclusives)
`(aaa == 10) || (bbb == 10)`	verdadeiro tanto se aaa ou bbb (ou ambos) forem iguais a 10

Observe que a instrução `if` tem duas partes. A parte imediatamente após a condição, que é executada quando a condição é verdadeira, e a parte após `else`, que é executada quando a condição é falsa. A palavra-chave `else` e a parte após a mesma são opcionais. Então, pode-se fazer nada caso a condição seja falsa.

A instrução do

Existe outra estrutura de controle, a instrução `do`. Ela tem a seguinte forma geral:

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

As instruções entre as chaves após o `do` são executadas enquanto a condição for verdadeira. A condição tem a mesma forma daquela utilizada nas instruções `if`, conforme descrito acima. Aqui temos um exemplo de um programa. O robô se movimenta de maneira aleatória por 20 segundos e então pára.

```
int tempo_de_movimentacao, tempo_de_curvatura, tempo_total;

task main()
{
  tempo_total = 0;
  do
  {
    tempo_de_movimentacao = Random(100);
    tempo_de_curvatura = Random(100);
    OnFwd(OUT_A+OUT_C);
    Wait(tempo_de_movimentacao);
    OnRev(OUT_C);
    Wait(tempo_de_curvatura);
    tempo_total += tempo_de_movimentacao; tempo_total += tempo_de_curvatura;
  }
  while (tempo_total < 2000);
  Off(OUT_A+OUT_C);
}
```

Podemos notar que neste exemplo existem duas instruções em uma única linha. Isto é permitido. Você pode colocar tantas instruções em uma única linha quantas você quiser (desde que separe cada instrução por um ponto-e-vírgula). Mas por questões de legibilidade do programa, isso freqüentemente não é uma boa idéia.

Observe, também, que a instrução `do` se comporta de maneira semelhante à instrução `while`. A diferença é que na estrutura `while` a condição é testada antes de executar suas instruções, enquanto que na estrutura `do` a condição é testada no fim. Em uma instrução `while`, pode acontecer do bloco de comandos nunca ser executado, mas na instrução `do` o bloco de comandos é executado pelo menos uma vez.

Resumo

Neste capítulo vimos duas estruturas de controle novas: a instrução `if` e a instrução `do`. Juntamente com as instruções `repeat` e `while`, elas são as instruções que controlam a maneira com que o programa é executado. É muito importante que você entenda o que elas fazem. Sendo assim, seria bom que você tentasse mais alguns exemplos desenvolvidos por você mesmo antes de continuar.

Vimos, também, que podemos colocar várias instruções em uma mesma linha.

V. Sensores

Um dos aspectos interessantes dos robôs Lego é que você pode conectar a eles sensores e fazer o robô reagir a esses sensores. Antes de mostrar como fazer isto, devemos modificar o robô que temos utilizado um pouco, por intermédio da instalação de um sensor. Para isso, monte o sensor conforme apresentado na figura 4 da página 28 do construpedia. Você deve fazer isto de maneira ligeiramente diferente, de tal modo que o robô se pareça com a figura abaixo:

Conecte o sensor na entrada 1 do RCX.

Esperando por um sensor

Iniciaremos com um programa simples em que o robô se move para frente até que ele bata em algo. Aqui está:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);      //entrada 1 é um sensor de toque
  OnFwd(OUT_A+OUT_C);
  until (SENSOR_1 == 1);                 //espera SENSOR_1 se tornar 1
  Off(OUT_A+OUT_C);
}
```

Existem duas linhas importantes aqui. A primeira delas nos diz o tipo de sensor que nós estamos usando. `SENSOR_1` é o número da entrada em que o sensor está conectado. As outras duas entradas são denominadas `SENSOR_2` e `SENSOR_3`. `SENSOR_TOUCH` indica que se trata de um sensor de toque. Para o sensor de luz nós utilizaríamos `SENSOR_LIGHT`. Após termos especificado o tipo de sensor, o programa liga os dois motores e o robô se movimenta para frente. A próxima instrução (**`until`**) é uma construção muito útil. Ela aguarda até que a condição definida entre parênteses seja verdadeira. Essa condição diz que o valor do `SENSOR_1` deve ser 1, o que significa que o sensor está pressionado. Tão logo o sensor seja liberado (deixe de estar pressionado), o valor se torna 0. Então, esta instrução aguarda até que o sensor seja pressionado (a tarefa fica parada neste ponto aguardando a condição ser satisfeita). Quando isso ocorre os motores são desligados e a tarefa é finalizada.

Respondendo a um sensor de toque

Deixe-nos tentar fazer um robô desviar de obstáculos. Caso o robô bata em um objeto, nós o instruiremos a se mover para traz, efetuar um giro e então continuar. Um programa para essas tarefas se parece com:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C); Wait(30);
      OnFwd(OUT_A); Wait(30);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

Tal como apresentado no exemplo anterior, primeiro indicaremos o tipo de sensor. Em seguida o robô inicia seu movimento para frente. No loop infinito `while`, verificaremos continuamente se o sensor foi acionado e, em caso verdadeiro, o robô se moverá para trás por um tempo de 1/3 de segundo, moverá para a direita pelo mesmo período de tempo (1/3 de segundo) e então continuará se movimentando para frente.

Sensores de luz

Além dos sensores de toque, o kit Lego Mindstorms possui sensores de luz. O sensor de luz mede a quantidade de luz em uma determinada direção. Além disso, esse sensor emite luz. Sendo assim, é possível apontar o sensor para uma determinada direção e fazer a distinção entre a intensidade de luz refletida por um objeto existente naquela direção. Em particular, isso é útil quando se tenta fazer um robô seguir uma linha existente no chão. Isso é o que faremos no próximo exemplo. Primeiramente precisamos encaixar o sensor de luz no robô de tal modo que o sensor fique posicionado no meio do robô, apontando para baixo. Conecte-o na entrada 2. Por exemplo, faça uma construção conforme ilustrado abaixo:

Precisaremos também da pista que vem com o kit. (Aquele pedaço grande de papel com uma trilha preta nele.) A idéia agora é que o robô mantenha o sensor de luz sobre a trilha. Caso a intensidade do sensor de luz aumentar, significa que o sensor saiu da trilha e precisaremos ajustar a direção. A seguir é apresentado um exemplo simples de como fazer isso, porém somente funciona se o robô seguir a trilha no sentido horário.

```
#define VALOR_LIMITE 40

task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_2 > VALOR_LIMITE)
    {
      OnRev(OUT_C);
      until (SENSOR_2 <= VALOR_LIMITE);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

O programa indica, inicialmente, que o sensor 2 é um sensor de luz. Em seguida, ele configura o robô para se mover para frente e entra em um loop infinito. Caso o valor lido pelo sensor seja maior do que 40 (utilizamos uma constante aqui de modo que seja fácil modificar o programa no caso de exposição do robô a ambientes com grande variedade de luminosidade) revertemos o sentido de rotação de um dos motores (OUT_C) e aguardamos até que o robô retorne à trilha.

Como poderá ser observado quando o programa for executado, a movimentação não é muito regular. Tente adicionar um comando `Wait(10)` antes da instrução `until` de modo a fazer o robô se mover de maneira melhor. Observe que o programa não funciona para a movimentação no sentido anti-horário. Para permitir uma movimentação em caminhos arbitrários, faz-se necessário um programa mais complicado.

Resumo

Neste capítulo foi visto como trabalhar com sensores de toque e de luz. Também vimos o comando `until` que é útil quando se usa sensores.

Eu sugiro que você escreva uma variedade de programas com os conceitos vistos até este momento. Você tem agora todos os ingredientes necessários para dar ao robô comportamentos bastante complexos. Por exemplo, tente colocar dois sensores de toque no robô, um na parte frontal esquerda e o outro na parte frontal direita, e o faça se afastar dos obstáculos que porventura ele se choque. Além disso, tente manter o robô em uma área delimitada por uma linha preta no chão.

VI. Tarefas e sub-rotinas

Até agora todos os nossos programas consistiam de uma única tarefa. Entretanto, os programas em NQC podem ter múltiplas tarefas. É também possível colocar pedaços de código em sub-rotinas que poderão ser utilizadas em diferentes lugares em um programa. Usar tarefas e sub-rotinas torna os programas mais fáceis de entender e mais compactos. Neste capítulo veremos as várias possibilidades.

Tarefas

Um programa NQC pode possuir no máximo 10 tarefas. Cada tarefa tem um nome. Uma das tarefas deve ter o nome `main` (tarefa principal), e será automaticamente executada. As outras tarefas serão executes somente quando uma tarefa em execução pedir para elas executem, por intermédio do comando `start`. A partir deste ponto, as duas tarefas estarão sendo executadas simultaneamente (devido ao fato da primeira tarefa continuar em execução). Uma tarefa em execução pode finalizar a execução de outra tarefa por meio do comando `stop`. Posteriormente, essa tarefa poderá ser reiniciada novamente, mas iniciará a execução do início e não a partir do ponto em que foi finalizada.

Deixe-me demonstrar o uso de tarefas. Coloque novamente o sensor de toque no robô. Desejamos fazer um programa em que o robô se movimenta em um caminho com a forma de um quadrado, como anteriormente. No entanto, quando o robô chocar com um obstáculo ele deverá reagir ao obstáculo. É difícil fazer isto em uma única tarefa, devido ao fato do robô necessitar fazer duas coisas ao mesmo tempo: movimentar-se (isto é, ligar e desligar os motores na hora certa) e responder aos sensores. Então, o melhor a fazer é utilizar duas estratégias para isso, onde uma delas controla o movimento em forma de quadrado e a outra reage aos sensores. Segue um exemplo.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start verificarSensores;
  start movimentarEmQuadrados;
}

task movimentarEmQuadrados()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task verificarSensores()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop movimentarEmQuadrados;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start movimentarEmQuadrados;
    }
  }
}
```

A tarefa `main` simplesmente define o tipo de sensor e então inicializa as outras tarefas. Após isso, a tarefa `main` é finalizada. A tarefa `movimentarEmQuadrados` move o robô continuamente em um circuito com a forma de um quadrado. A tarefa `verificarSensores` verifica se o sensor de toque é pressionado. Caso tenha sido, ele responde com as seguintes ações: antes de tudo ele pára a tarefa `movimentarEmQuadrados`. Isso é muito importante. `verificarSensores` agora possui o controle sobre as movimentações do robô. Em seguida, ela move o robô um pouco para trás e o faz girar. Agora ela pode iniciar novamente a tarefa

`movimentarEmQuadrados` para possibilitar que o robô se movimente em um caminho com a forma de um quadrado.

É muito importante lembrar que todas as tarefas que você inicia estão sendo executadas ao mesmo tempo. Isso pode levar a resultados inesperados. O Capítulo X explica esses problemas em detalhes e oferece soluções para eles.

Sub-rotinas

Às vezes você precisa que um mesmo pedaço de código seja utilizado em vários lugares em um programa. Nesse caso, você pode colocar o pedaço de código em uma sub-rotina e lhe dar um nome. Agora você pode executar esse pedaço de código simplesmente efetuando uma chamada ao seu nome dentro de uma tarefa. NQC (ou melhor, o RCX) permite até 8 sub-rotinas. Vamos ver um exemplo.

```
sub darVoltas()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  darVoltas();
  Wait(200);
  darVoltas();
  Wait(100);
  darVoltas();
  Off(OUT_A+OUT_C);
}
```

Neste programa definimos uma sub-rotina que faz o robô dar voltar em torno de si mesmo. A tarefa `main` efetua uma chamada a essa rotina três vezes. Observe que efetuamos uma chamada à sub-rotina por meio da escrita de seu nome seguido de um par de parênteses. Isso se parece bastante com os demais comandos que temos visto. A diferença é uma sub-rotina não pode definir parâmetros, o que implica em nunca termos algo entre os parênteses de uma sub-rotina.

São necessários alguns esclarecimentos antes de continuarmos. Sub-rotinas são um pouco estranhas. Por exemplo, sub-rotinas não podem ser chamadas a partir de outras sub-rotinas. Por outro lado, sub-rotinas podem ser chamadas a partir de diferentes tarefas, entretanto isto não é indicado. É muito fácil ocorrerem problemas devido a duas tarefas distintas tentarem, ao mesmo tempo, executar uma mesma sub-rotina. Isso leva a efeitos indesejáveis. Além disso, quando se efetua chamadas a uma mesma sub-rotina a partir de tarefas distintas, devido a uma limitação do firmware do RCX, você fica impedido de utilizar expressões complicadas (nesse caso, não se pode ter variáveis locais ou efetuar cálculos que requerem variáveis temporárias). Então, a menos que você tenha certeza do que está fazendo, *não efetue chamadas a uma sub-rotina a partir de diferentes tarefas!*

Funções inline

Conforme citado acima, sub-rotinas causam certos problemas. A parte boa é que elas são armazenadas somente uma vez no RCX. Isso gasta menos memória e, pelo fato do RCX ter pouca memória livre, isso é útil. Mas quando as sub-rotinas são pequenas, o melhor a fazer é utilizar funções *inline*. Elas não são armazenadas separadamente, mas copiadas para cada lugar em que são utilizadas. Isso gasta mais memória, mas os problemas citados anteriormente relacionados a expressões complicadas deixam de existir. Além disso, não existe um limite para a quantidade de funções *inline*.

Definir e chamar funções *inline* ocorre exatamente da mesma maneira que sub-rotinas. A única diferença é a utilização da palavra-chave **void** ao invés de **sub**. (A palavra **void** é utilizada devido ao fato dessa mesma palavra ser usada em outras linguagens, tal como C, quando uma função não retorna um valor. As funções em NQC não retornam valor.) Desse modo, o programa acima, com o uso de funções *inline*, seria assim:

```
void darVoltas()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  darVoltas();
  Wait(200);
  darVoltas();
  Wait(100);
  darVoltas();
  Off(OUT_A+OUT_C);
}
```

Funções *inline* têm outra vantagem sobre sub-rotinas. Elas podem ter parâmetros. Argumentos podem ser usados para passar um valor para certas variáveis definidas na função *inline*. Por exemplo, assuma, no exemplo acima, que podemos tornar o tempo de giro um parâmetro para a função, conforme apresentado abaixo:

```
void darVoltas(int tempoDeGiro)
{
  OnRev(OUT_C); Wait(tempoDeGiro);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  darVoltas(200);
  Wait(200);
  darVoltas(50);
  Wait(100);
  darVoltas(300);
  Off(OUT_A+OUT_C);
}
```

Observe que foi especificado entre os parênteses associados à função *inline* o parâmetro da função. Neste caso nós indicamos que o argumento deverá ser um inteiro (existem outras opções) e o nome do parâmetro é `tempoDeGiro`. Quando existem mais parâmetros, os mesmos devem vir separados por vírgulas.

Definindo macros

Existe ainda outra maneira de associar um nome a pequenos pedaços de código. Você pode definir macros em NQC (não confunda com os macros do RCX Command Center). Foi visto que podemos definir constantes usando a instrução `#define`, dando-as um nome. Mas na verdade nós podemos definir qualquer pedaço de código. A seguir é apresentado o mesmo programa anterior com a definição de uma macro para executar as ações existentes em `darVoltas`.

```
#define darVoltas OnRev(OUT_C);Wait(340);OnFwd(OUT_A+OUT_C);

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  darVoltas;
  Wait(200);
  darVoltas;
  Wait(100);
  darVoltas;
  Off(OUT_A+OUT_C);
}
```

Após a instrução `#define` a palavra `darVoltas` representa o texto que a segue. Agora caso você digite `darVoltas`, essa palavra será substituída por este texto. Observe que o texto deve estar em uma única linha. (Na verdade existem maneiras de colocar uma instrução `#define` em múltiplas linhas, mas não é recomendado.)

Instruções `#define` são realmente muito mais poderosas. Elas também podem ter parâmetros. Por exemplo, podemos colocar o tempo de giro como um parâmetro na instrução. Aqui será apresentado um exemplo em que definimos quatro macros: uma para mover o robô para frente, uma para mover para trás, uma para virar à esquerda e outra para virar à direita.

```
#define virarDireita(s,t)  SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A);OnRev(OUT_C);Wait(t);
#define virarEsquerda(s,t)  SetPower(OUT_A+OUT_C,s);OnRev(OUT_A);OnFwd(OUT_C);Wait(t);
#define frente(s,t)  SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A+OUT_C);Wait(t);
#define tras(s,t)  SetPower(OUT_A+OUT_C,s);OnRev(OUT_A+OUT_C);Wait(t);

task main()
{
  frente(3,200);
  virarEsquerda(7,85);
  frente(7,100);
  tras(7,200);
  frente(7,100);
  virarDireita(7,85);
  frente(3,200);
  Off(OUT_A+OUT_C);
}
```

É muito útil definir macros. Isso faz o seu código mais compacto e legível. Além do mais, você pode modificar seu código facilmente quando, por exemplo, você modificar as conexões dos motores.

Resumo

Neste capítulo vimos o uso de tarefas, sub-rotinas, funções *inline* e macros. Elas têm utilidades distintas. As tarefas executam simultaneamente e tratam coisas distintas que têm que ser feitas ao mesmo tempo. Sub-rotinas são úteis quando grandes pedaços de código devem estar em diferentes lugares em uma mesma tarefa. Funções *inline* são úteis quando pedaços de código devem ser usados em vários locais em diferentes tarefas, porém com um consumo adicional de memória. Finalmente, macros são muito úteis para pequenos pedaços de código que devem ser usados em diferentes locais. Eles também podem conter parâmetros, o que os torna mais úteis.

Agora que você chegou até aqui, você possui todo tipo de conhecimento para fazer seu robô executar tarefas complexas. Os demais capítulos deste tutorial o ensinarão a respeito de outras coisas que são importantes somente em certas aplicações.

VII. Criando músicas

O RCX tem um alto-falante embutido que pode produzir sons e assim tocar pedaços de músicas simples. Isto é, em particular, útil quando você quiser fazer o RCX "dizer" o que está acontecendo. Mas pode ser divertido ter um robô que toca música enquanto se movimenta

Sons pré-programados

Existem seis sons pré-programados no RCX, numerados de 0 a 5, conforme apresentado a seguir:

0	Pressionamento de tecla
1	Beep beep
2	Mudança de freqüência decrescente
3	Mudança de freqüência crescente
4	Som de erro ('Buhhh')
5	Mudança crescente rápida

Você pode tocar estes sons por meio do comando `PlaySound()`. Aqui temos um pequeno programa que toca todos eles.

```
task main()
{
  PlaySound(0); Wait(100);
  PlaySound(1); Wait(100);
  PlaySound(2); Wait(100);
  PlaySound(3); Wait(100);
  PlaySound(4); Wait(100);
  PlaySound(5); Wait(100);
}
```

Você pode se perguntar por que existem estes comandos `Wait`. A razão é que o comando que toca o som não espera pelo seu término. Ele imediatamente executa o próximo comando. O RCX tem um pequeno buffer em que ele armazena sons, mas após um tempo este buffer enche e os sons são perdidos. Isso não faz muita diferença para sons, mas é muito importante para músicas, como veremos a seguir.

Observe que o argumento para `PlaySound()` deve ser uma constante. Você não pode utilizar uma variável aqui!

Tocando música

Para músicas mais interessantes, o NQC tem o comando `PlayTone()`. Ele tem dois parâmetros. O primeiro deles é a freqüência e o segundo a duração (em ticks de 1/100 segundos, tal como no comando `Wait`). Abaixo temos uma tabela de freqüências úteis:

Sound	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
G#	52	104	208	415	831	1661	3322	
G	49	98	196	392	784	1568	3136	
F#	46	92	185	370	740	1480	2960	
F	44	87	175	349	698	1397	2794	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1245	2489	
D	37	73	147	294	587	1175	2349	
C#	35	69	139	277	554	1109	2217	
C	33	65	131	262	523	1047	2093	4186
B	31	62	123	247	494	988	1976	3951
A#	29	58	117	233	466	932	1865	3729
A	28	55	110	220	440	880	1760	3520

Conforme citado anteriormente com relação aos sons, aqui também o RCX não espera uma nota terminar. Assim sendo, caso você use várias notas, o melhor a fazer é adicionar comandos `Wait` entre elas (fica um pouco maior). Eis um exemplo:

```
task main()
{
  PlayTone(262,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(330,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(262,160); Wait(200);
}
```

Você pode criar pedaços de músicas muito facilmente utilizando o RCX Piano que é parte do RCX Command Center.

Caso você queira que o RCX toque uma música enquanto se movimenta, o melhor a fazer é separar uma tarefa para isso. A seguir é apresentado um exemplo de um programa bastante simples onde o robô se move constantemente para frente e para trás, tocando uma música.

```
task tocarMusica()
{
  while (true)
  {
    PlayTone(262,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
    PlayTone(330,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
  }
}

task main()
{
  start tocarMusica;
  while(true)
  {
    OnFwd(OUT_A+OUT_C); Wait(300);
    OnRev(OUT_A+OUT_C); Wait(300);
  }
}
```

Resumo

Neste capítulo você aprendeu como fazer o RCX produzir sons e músicas. Além disso, foi mostrado como usar uma tarefa específica para tocar música.

VIII. Mais a respeito de motores

Existem inúmeros comandos adicionais que você pode utilizar para controlar os motores de maneira mais precisa. Neste capítulo nós discutiremos isso.

Parando suavemente

Quando você utiliza o comando `Off()`, o motor para imediatamente, usando freios. Em NQC também é possível para os motores de maneira suave, sem utilizar freios. Para esse propósito utilizamos o comando `Float()`. Às vezes isso é melhor para a tarefa que o robô está executando. Utilizemos o seguinte exemplo. Primeiramente o robô pára usando freios; depois faz o mesmo sem usar os freios. Observe a diferença. (Provavelmente a diferença seja bastante sutil para o robô que estamos usando. Entretanto pode fazer uma grande diferença para outros tipos de robôs.)

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Off(OUT_A+OUT_C);
  Wait(100);
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Float(OUT_A+OUT_C);
}
```

Comandos avançados

Na verdade, o comando `OnFwd()` faz duas coisas: ele liga o motor e configura sua direção para frente. O comando `OnRev()` também faz duas coisas: ele liga o motor e configure a direção no sentido reverso. O NQC também tem comandos para fazer essas coisas separadamente. Se você quiser mudar somente uma de duas coisas, é mais eficiente utilizar esses comandos em separado; isso faz uso de menos memória no RCX, é mais rápido e pode resultar em movimentos mais suaves. Os dois comandos separados são `SetDirection()` que configura a direção (`OUT_FWD`, `OUT_REV` ou `OUT_TOGGLE` nos quais troca a direção atual) e `SetOutput()` que define o modo (`OUT_ON`, `OUT_OFF` ou `OUT_FLOAT`). A seguir é apresentado um programa simples que faz o robô se mover para frente, para trás e para frente de novo.

```
task main()
{
  SetPower(OUT_A+OUT_C,7);
  SetDirection(OUT_A+OUT_C,OUT_FWD);
  SetOutput(OUT_A+OUT_C,OUT_ON);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_REV);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_TOGGLE);
  Wait(200);
  SetOutput(OUT_A+OUT_C,OUT_FLOAT);
}
```

Cabe observar que no início de todos os programas, todos os motores são configurados automaticamente para se moverem para frente e a velocidade é definida em 7. Sendo assim, no exemplo acima os dois primeiros comandos são desnecessários.

Existem vários outros comandos para controlar os motores, nos quais consistem de uma abreviação da combinação dos comandos apresentados anteriormente. A seguir é apresentada uma lista completa:

`On('motores')`	Liga o motor
`Off('motores')`	Desliga o motor
`Float('motores')`	Desliga os motores sem frenagem
`Fwd('motores')`	Configura o motor na direção frente (não o coloca em movimento)
`Rev('motores')`	Configura o motor na direção trás (não o coloca em movimento)
`Toggle('motores')`	Muda a direção do motor (frente para trás e vice-versa)

`OnFwd('motores')`	Configura o motor na direção frente e o coloca em movimento
`OnRev('motores')`	Configura o motor na direção trás e o coloca em movimento
`OnFor('motores','ticks')`	Liga os motores por um período de tempo em ticks (100 ticks = 1s)
`SetOutput('motores','modo')`	Configura a saída (`OUT_ON`, `OUT_OFF` ou `OUT_FLOAT`)
`SetDirection('motores','dir')`	Configura a direção da saída (`OUT_FWD`, `OUT_REV` ou `OUT_TOGGLE`)
`SetPower('motores','potencia')`	Configura a potência da saída (0-7)

IX. Mais a respeito de sensores

No Capítulo V nós discutimos os aspectos básicos relacionados a sensores. Entretanto, existe muito mais que você pode fazer com sensores. Neste capítulo discutiremos as diferenças entre modo e tipo de sensores, veremos como usar o sensor de rotação (um tipo de sensor que não vem com o RIS, mas pode ser comprado separadamente e é muito útil), veremos alguns truques para usar mais do que três sensores e faremos um sensor de proximidade.

Modo e tipo de sensores

O comando `SetSensor()` que nós vimos anteriormente faz duas coisas: define o tipo e o modo de operação do sensor. Configurar o modo e o tipo de um sensor de maneira separada possibilita uma maior precisão no controle do comportamento do sensor, o que é útil para algumas aplicações.

O tipo do sensor é definido com o comando `SetSensorType()`. Existem quatro tipos diferentes: `SENSOR_TYPE_TOUCH`, para sensor de toque, `SENSOR_TYPE_LIGHT`, para sensor de luz, `SENSOR_TYPE_TEMPERATURE`, para sensor de temperatura (este tipo de sensor não faz parte do kit, mas pode ser comprado separadamente) e `SENSOR_TYPE_ROTATION`, para o sensor de rotação (também não faz parte do kit – disponível separadamente). Configurar o tipo de um sensor em particular é importante para indicar se o sensor necessita de energia (tal como, por exemplo, para a luz do sensor de luz). Eu desconheço alguma utilidade em se colocar um sensor em um tipo diferente daquele que ele é.

O modo do sensor é definido com o comando `SetSensorMode()`. Existem oito modos diferentes. O mais importante deles é o `SENSOR_MODE_RAW`. Nesse modo, o valor que se obtém quando se faz a leitura do sensor é um número compreendido entre 0 e 1023. É o valor bruto produzido pelo sensor. O que ele representa depende do sensor em uso. Por exemplo, para um sensor de toque, quando o sensor está em repouso (não pressionado) o valor é próximo de 1023. Quando ele estiver totalmente pressionado, o valor é próximo de 50. Quando estiver parcialmente pressionado o valor se encontra na faixa entre 50 e 1000. Então, caso você configure um sensor de toque no modo bruto, você pode realmente verificar se ele está parcialmente pressionado. Quando o sensor é um sensor de luz, a faixa de valores vai de 300 (muita luz) até 800 (muito escuro). Isso nos dá um valor muito mais preciso do que quando utilizamos o comando `SetSensor()`.

O próximo modo é `SENSOR_MODE_BOOL`. Nesse modo o valor é 0 ou 1. Quando o valor bruto lido for menor do que 550 o valor é 0; maior é 1. `SENSOR_MODE_BOOL` é o modo padrão do sensor de toque. Os modos `SENSOR_MODE_CELSIUS` e `SENSOR_MODE_FAHRENHEIT` são úteis somente para os sensores de temperatura e apresentam a temperatura na maneira indicada (Celsius ou Fahrenheit). `SENSOR_MODE_PERCENT` apresenta um valor bruto entre 0 e 100. Todo valor menor ou igual a 400 é mapeado para 100 por cento. Caso o valor seja maior do que 400, a porcentagem vagarosamente se torna 0. `SENSOR_MODE_PERCENT` é o modo padrão para o sensor de luz. `SENSOR_MODE_ROTATION` parece ser útil somente para o sensor de rotação (ver abaixo).

Existem dois outros modos interessantes: `SENSOR_MODE_EDGE` e `SENSOR_MODE_PULSE`. Eles contam transições, isto é, mudanças de valores altos para valores baixos ou vice-versa. Por exemplo, quando você pressiona um sensor de toque, isso causa uma transição de um valor bruto alto para um valor baixo. Quando você libera o sensor, você gera uma transição na outra direção (valor baixo para alto). Quando você define o modo do sensor para `SENSOR_MODE_PULSE`, somente transições de um valor baixo para um valor alto serão contadas. Então, cada toque e liberação em um sensor de toque contam como uma única transição. Quando você configura o modo do sensor para `SENSOR_MODE_EDGE`, ambas as transições são contadas. Então, cada toque e liberação em um sensor de toque contam como duas transições. Você pode utilizar isso para contar a freqüência com que um sensor de toque foi pressionado. Ou você pode usá-lo em combinação com um sensor de luz para contar a freqüência com que uma lâmpada é acessa e apagada. Normalmente, quando se está contando coisas é desejável zerar o contador. O comando `ClearSensor()` é utilizado para esse propósito. Ele zera o contador para os sensores indicados.

Vamos analisar um exemplo. O seguinte programa usa um sensor de toque para guiar o robô. Conecte o sensor de toque com um cabo longo na entrada 1. Caso o sensor de toque seja tocado rapidamente duas vezes o robô se movimente para frente. Caso seja tocado somente uma vez ele pára.

```
task main()
{
  SetSensorType(SENSOR_1,SENSOR_TYPE_TOUCH);
  SetSensorMode(SENSOR_1,SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(SENSOR_1);
    until (SENSOR_1 > 0);
    Wait(100);
    if (SENSOR_1 == 1) {Off(OUT_A+OUT_C);}
    if (SENSOR_1 == 2) {OnFwd(OUT_A+OUT_C);}
  }
}
```

Observe que primeiramente configuramos o tipo e o modo do sensor. Parece que isso é essencial, pois ao mudar o tipo também afetamos o modo de operação.

O sensor de rotação

O sensor de rotação é um tipo de sensor muito útil, porém infelizmente não faz parte do kit padrão. Ele pode ser comprado separadamente diretamente na Lego. O sensor de rotação tem um orifício onde se pode colocar um eixo. O sensor de rotação mede a quantidade de vezes que aquele eixo gira. Uma rotação complete do eixo contém 16 passos (ou -16 caso você rode no sentido contrário). Sensores de rotação são mais úteis para controlar os movimentos do robô de maneira mais precisa. Você pode fazer um eixo girar a quantia exata que você quiser. Se você quiser um controle com precisão maior do que 16 passos, você pode utilizar engrenagens, conectá-las a um eixo que se mova mais rápido e usá-la para contar os passos.

Uma aplicação básica utiliza dois sensores de rotação conectados a duas rodas de um robô que você controla por intermédio de dois motores. Para um movimento em linha reta você deseja que ambas as rodas girem na mesma velocidade. Infelizmente, os motores não giram exatamente na mesma velocidade. Usando sensores de rotação você pode constatar isso. Você pode, temporariamente, parar o motor que gira mais rápido (melhor se usar `Float()`) até que ambos os sensores possuam o mesmo valor. O programa abaixo faz isso. Ele simplesmente faz o robô se movimentar em linha reta. Para utilizá-lo, modifique seu robô, conectando dois sensores de rotação às duas rodas. Conecte os sensores nas entras 1 e 3.

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

O programa inicialmente indica que ambos os sensores são de rotação e estabelece 0 como seus valores de inicialização (por meio do comando `ClearSensor`). Posteriormente, inicia um loop infinito. No loop verificamos se as leituras dos sensores são iguais. Caso sejam, ele simplesmente move o robô em linha reta. Caso alguma leitura apresente um valor maior, o motor adequado é parado até que as leituras apresentem valores iguais.

Pode-se observar que o programa é bastante simples. Você pode estendê-lo para fazer o robô se movimentar em distâncias precisas, ou possibilitá-los fazer curvas com precisão.

Colocando vários sensores em uma única entrada

O RCX tem somente três entradas, o que implica que você só pode conectar três sensores. Quando você quer construir robôs mais complexos (e você compra sensores extras) isso pode não ser o suficiente. Felizmente, com alguns truques, você pode conectar dois sensores (ou até mais) em uma única entrada.

O mais fácil é conectar dois sensores de toque em uma única entrada. Se um deles (ou ambos) é pressionado, o valor será 1, caso contrário será 0. Você não tem como distinguir qual foi pressionado, mas às vezes isso não é necessário. Por exemplo, quando você coloca um sensor de toque na frente do robô e outro atrás, você sabe qual foi tocado com base na direção que o robô está se movendo. Mas você pode configurar o modo para entrada bruta (veja anteriormente). Agora, você pode obter muito mais informações. Se você estiver com sorte, o valor de leitura do sensor quando pressionado não será o mesmo para ambos os sensores. Sendo este o caso, você pode realmente distinguir entre os dois sensores. E, quando ambos são pressionados, você obtém um valor de leitura bem pequeno (em torno de 30), o que o possibilita detectar essa situação.

Você pode conectar um sensor de toque e um sensor de luz a uma mesma entrada. Defina o tipo para luz (de outro modo o sensor de luz não funcionará). Defina o modo de operação para bruto. Nesse caso, quando o sensor de toque for pressionado você obterá um valor bruto menor que 100. Caso não seja pressionado o valor de leitura do sensor nunca será menor do que 100. O programa abaixo ilustra essa idéia. O robô deve ser equipado com um sensor de luz apontado para baixo, e um anteparo na frente (para identificar quando o robô bateu em algo) conectado ao sensor de toque. Conecte ambos na entrada 1. O robô se moverá de maneira aleatória dentro de uma área de cor clara. Quando o sensor identificar uma linha preta (valor bruto > 750) ele recuará um pouco. Quando o sensor de toque bater em algo (valor bruto menor do que 100) ele fará o mesmo. Veja um programa exemplo:

```
int ttt,tt2;

task moverAleatoriamente()
{
  while (true)
  {
    ttt = Random(50) + 40;
    tt2 = Random(1);
    if (tt2 > 0)
      { OnRev(OUT_A); OnFwd(OUT_C); Wait(ttt); }
    else
      { OnRev(OUT_C); OnFwd(OUT_A);Wait(ttt); }
    ttt = Random(150) + 50;
    OnFwd(OUT_A+OUT_C);Wait(ttt);
  }
}

task main()
{
  start moverAleatoriamente;
  SetSensorType(SENSOR_1,SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_1,SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
    {
      stop moverAleatoriamente;
      OnRev(OUT_A+OUT_C);Wait(30);
      start moverAleatoriamente;
    }
  }
}
```

Eu espero que o programa esteja simples. Existem duas tarefas. A tarefa `moverAleatoriamente` faz o robô se mover de maneira aleatória. A tarefa principal inicialmente inicia `moverAleatoriamente`, define o sensor e aguarda algo acontecer. Caso a leitura do sensor se torne muito pequeno (sendo tocado) ou muito grande (fora da área clara), ele pára de se mover aleatoriamente, recua um pouco e inicia a movimentação aleatória novamente.

Também é possível conectar dois sensores de luz na mesma entrada. O valor bruto estará, de alguma maneira, relacionado com a combinação da quantidade de luz recebida pelos sensores. Mas isso é bastante confuso e pode parecer difícil de usar. Conectar outros sensores, tal como rotação e temperatura, parece não ser útil.

Fazendo um sensor de proximidade

Usando sensores de toque seu robô pode reagir quando ele bater em algo. Mas seria bastante mais interessante se o robô reagisse antes de bater em algo. Ele saberia que está próximo de um obstáculo. Infelizmente, não existem sensores para esse propósito. Existe um truque que podemos usar para fazer isso. O robô tem uma porta de luz infra-vermelha com os quais ele se comunica com o computador, ou com outros robôs. (Veremos mais a respeito de comunicação entre robôs no Capítulo XI.) Acontece que o sensor de luz que vem com o kit é muito sensível à luz infra-vermelha. Poderíamos, então, construir um sensor baseado nisso. A idéia é a seguinte: uma tarefa envia mensagem em infra-vermelho e uma outra tarefa mede as flutuações na intensidade de luz que são refletidas pelos objetos; quanto maior a flutuação mais perto o robô estará de um objeto.

Para usar essa idéia, posicione o sensor de luz sobre a porta de luz infra-vermelha do robô, apontando-o para frente. Desse modo ele somente medirá a reflexão da luz infra-vermelha. Conecte o sensor na entrada 2. Definiremos o modo do sensor de luz para bruto, de modo a identificarmos as flutuações da melhor maneira possível. A seguir é apresentado um programa simples que faz o robô se movimentar para frente até que ele se aproxima de um objeto e, devido a isso, o programa o faz girar 90 graus para a direita.

```
int nivelAnterior;          // Para armazenar o nível anterior

task enviarSinal()
{
  while(true)
    {SendMessage(0); Wait(10);}
}

task verificarSinal()
{
  while(true)
  {
    nivelAnterior = SENSOR_2;
    if(SENSOR_2 > nivelAnterior + 200)
      {OnRev(OUT_C); Wait(85); OnFwd(OUT_A+OUT_C);}
  }
}

task main()
{
  SetSensorType(SENSOR_2, SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_2, SENSOR_MODE_RAW);
  OnFwd(OUT_A+OUT_C);
  start enviarSinal;
  start verificarSinal;
}
```

A tarefa `enviarSinal` envia 10 sinais de infra-vermelho a cada segundo, usando o comando `SendMessage(0)`. A tarefa `verificarSinal` armazena o valor lido pelo sensor de luz, repetidamente. Então, ela verifica se um desses valores é maior do que o valor anterior em pelo menos 200 unidades, o que indicaria uma grande flutuação. Caso isso tenha ocorrido, ela faz o robô efetuar um giro de 90 graus para a direita. O valor 200 é arbitrário. Se você o fizer menor, o robô se desviará mais rapidamente do obstáculo, ou seja, notará o objeto a uma distância maior. Caso seja um valor maior, o robô chegará mais perto antes de notar o objeto. Mas isso também depende do tipo de material e da quantidade de luminosidade disponível no ambiente. Você deveria experimentar ou utilizar algum mecanismo mais fácil e adequado para aprender a determinar o valor correto.

Uma desvantagem da técnica é o fato da mesma funcionar somente quando o robô está se movimentando em uma direção. Você provavelmente precisará de sensores de toque localizados nas laterais para evitar colisões nesses locais. Mas a técnica é muito útil para robôs que necessitam se locomover em labirintos. Outra desvantagem é o fato de não ser possível estabelecer uma comunicação do computador com o robô porque a mesma irá interferir nas mensagens de infra-vermelho enviadas pelo robô. (O controle remoto da sua televisão provavelmente também não funcionará.)

Resumo

Neste capítulo foram vistas algumas questões adicionais relacionadas a sensores. Vimos como configurar o tipo e o modo de um sensor de maneira separada, além de como utilizar para se obter informações adicionais.

Aprendemos como utilizar um sensor de rotação. Vimos, também, como fazer para conectar vários sensores em uma mesma entrada do RCX. Finalmente, vimos um truque para utilizar a conexão de infra-vermelho e o sensor de luz do robô para criar um sensor de proximidade. Todos esses truques são extremamente úteis quando se está construindo robôs complexos. Sensores têm sempre um papel crucial nessas construções.

X. Tarefas paralelas

Como citado anteriormente, no NQC as tarefas são executadas simultaneamente, ou em paralelo como as pessoas normalmente dizem. Isso é extremamente útil, possibilitando-o efetuar a leitura de sensores, enquanto outra tarefa controla a movimentação do robô e, ainda, uma outra tarefa toca alguma música. Mas tarefas paralelas também podem causar problemas. Uma tarefa pode interferir em outra.

Um programa errado

Considere o seguinte programa. Nesse programa uma tarefa controla a movimentação do robô em caminhos com a forma de um quadrado (semelhante ao que foi feito em programas anteriores) e uma segunda tarefa verifica um sensor de toque. Quando o sensor for pressionado, ele moverá o robô um pouco para trás e fará um giro de 90 graus.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start verificarSensores;
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task verificarSensores()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C);
      Wait(50);
      OnFwd(OUT_A);
      Wait(85);
      OnFwd(OUT_C);
    }
  }
}
```

Provavelmente esse programa se pareça perfeito. Mas se você executá-lo, você provavelmente observará um comportamento inesperado. Tente o seguinte: faça o robô tocar em algo enquanto ele está girando. Ele começará a se mover para trás, mas imediatamente ele se moverá para frente novamente, batendo no obstáculo. A razão disso é que as tarefas estão interferindo umas nas outras. Olha o que acontece. O robô está girando à direita, isto é, a primeira tarefa está na sua segunda instrução `Wait`. Agora, o robô bate com o sensor em algo. Ele começa a se movimentar para trás, mas, ao mesmo tempo, a tarefa `main` finaliza a espera e move o robô para frente novamente; batendo no obstáculo. A segunda tarefa está em espera (instrução `Wait`) neste momento e, devido a isso, não percebe a colisão. Claramente, isto não é o comportamento que gostaríamos de ver. O problema é que enquanto a segunda tarefa está em espera, não percebemos que a primeira tarefa ainda estava em execução e que suas ações interferem com as ações da segunda tarefa.

Parando e reiniciando tarefas

Uma maneira de solucionar este problema é garantir que a qualquer momento somente uma tarefa estará controlando a movimentação do robô. Esta abordagem foi apresentada no Capítulo VI. Deixe repetir o programa aqui.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start verificarSensores;
  start movimentarEmQuadrados;
}

task movimentarEmQuadrados ()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task verificarSensores()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop movimentarEmQuadrados;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start movimentarEmQuadrados;
    }
  }
}
```

A questão aqui é que a tarefa `verificarSensores` somente move o robô após ter parado a tarefa `movimentarEmQuadrados`. Então, esta tarefa não pode interferir na movimentação de desvio de obstáculo. Uma vez que o procedimento de se afastar do objeto termina, ela inicia a tarefa `movimentarEmQuadrados` de novo.

Mesmo acreditando que esta seja uma boa solução para o problema descrito anteriormente, ainda existe um problema. Quando nós reiniciamos a tarefa `movimentarEmQuadrados`, ela inicia novamente executando a sua primeira instrução. Isso não traz problemas para o problema em questão, mas freqüentemente não é o comportamento desejado. Nós preferiríamos parar a tarefa em algum ponto e, posteriormente, continuar a sua execução daquele ponto. Infelizmente isso não pode ser feito de maneira muito fácil.

Usando semáforos

Uma técnica padrão utilizada para solucionar este problema é utilizar uma variável para indicar qual tarefa está controlando os motores em um determinado momento. As demais tarefas não conseguem obter o controle dos motores até que a primeira tarefa indique, por intermédio da variável, que está liberada. Tal variável é freqüentemente denominada semáforo. Abaixo, `sem` é um semáforo. Assumimos que o valor 0 em `sem` indica que nenhuma tarefa está controlando os motores. Agora, caso alguma tarefa queira fazer algo com os motores ela executa os seguintes comandos:

```
until (sem == 0);
sem = 1;
// Faça algo com os motores
sem = 0;
```

Então, inicialmente, esperamos até que ninguém precise dos motores. Quando isso ocorre, obtemos o controle por meio da definição da variável `sem` como 1. Agora, podemos controlar os motores. Quando tivermos terminado, definimos `sem` para 0. Neste ponto, chegamos ao programa acima, implementado com o uso de um semáforo. Quando o sensor de toque se choca em algo, o semáforo é definido em 1 e o procedimento de afastamento do obstáculo é executado. Durante esse procedimento a tarefa `movimentarEmQuadrados` deve esperar. No momento que o procedimento termina, o semáforo é definido em 0 e a tarefa `movimentarEmQuadrados` pode continuar.

```
int sem;

task main()
{
  sem = 0;
  start movimentarEmQuadrados;
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      sem = 0;
    }
  }
}

task movimentarEmQuadrados ()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_A+OUT_C);
    sem = 0;
    Wait(100);
    until (sem == 0); sem = 1;
    OnRev(OUT_C);
    sem = 0;
    Wait(85);
  }
}
```

Você poderia dizer que não é necessário definir o semáforo para 1 e de volta para 0 na `movimentarEmQuadrados`. Todavia, isso é útil. A razão é que o comando `OnFwd()` é de fato dois comandos (veja o Capítulo VIII). Você não quer que esta seqüência de comandos seja interrompida por outra tarefa.

Semáforos são muito úteis e quando se está escrevendo programas complicados com tarefas paralelas eles são quase sempre necessários. (Existe ainda uma chance mínima de eles falharem. Tente descobrir por que.)

Resumo

Neste capítulo estudamos alguns dos problemas que podem ocorrer quando utilizamos várias tarefas. Esteja sempre atento aos efeitos colaterais. Muitos dos comportamentos inesperados são provenientes disso. Vimos duas maneiras diferentes de solucionar tais problemas. A primeira solução pára e reinicia tarefas para garantir que somente uma tarefa crítica esteja sendo executada a cada momento. A segunda abordagem utiliza semáforos para controlar a execução das tarefas. Isso garante que a cada momento somente a parte crítica de uma tarefa está em execução.

XI. Comunicação entre robôs

Se você possui mais de um RCX, este capítulo foi feito para você. Os robôs podem se comunicar por meio das portas de infra-vermelho. Usando-as, você pode ter vários robôs colaborando entre si (ou disputando entre si). Além disso, você pode construir um robô maior usando dois RCX, de modo a ter seis motores e seis sensores (ou até mais caso utilize as dicas do Capítulo IX).

Comunicação entre robôs funciona, de modo geral, da seguinte forma. Um robô pode usar o comando `SendMessage()` para enviar um valor (0-255) através de sua porta de infra-vermelho. Todos os demais robôs recebem aquela mensagem e a armazenam. O programa em um robô pode verificar a última mensagem recebida por intermédio do comando `Message()`. Com base no valor, o programa pode fazer o robô executar certas ações.

Dando ordens

Freqüentemente, quando você tem dois ou mais robôs, um deles é o líder. Nós o chamamos de mestre (*master*). Os demais robôs são os escravos (*slaves*). O robô mestre envia ordens para os robôs escravos e esses as executam. Às vezes, os robôs escravos podem enviar de volta informações para o mestre como, por exemplo, o valor lido em um sensor. Deste modo, você precisa escrever dois programas, um para o robô mestre e outro para o(s) escravo(s). Aqui, iremos assumir que temos somente um escravo. Deixe-nos iniciar com um exemplo bastante simples. Nesse exemplo o robô escravo pode executar três diferentes ordens: mover para a frente, mover para trás e parar. Seu programa consiste de um loop simples. Neste loop o valor da mensagem atual é defina em 0, por meio do comando `ClearMessage()`. Depois ele aguarda até que a mensagem se torne diferente de 0. Baseado no valor da mensagem, o robô escravo executa uma de três diferentes ordens. Eis o programa:

```
task main()          // ESCRAVO
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

O robô mestre tem também um programa bastante simples. Ele simplesmente envia mensagens referentes às ordens e aguarda por um tempo. No programa abaixo, ele ordena aos escravos se moverem para frente. Então, após dois segundos, ele os ordena a se moverem para trás. Após novos dois segundos, ele os ordena a parar.

```
task main()          // MESTRE
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}
```

Após ter escrito esses dois programas, você precisa efetuar o download dos mesmos para os robôs. Cada programa é de um dos robôs. Esteja certo de que você tenha desligado o outro robô no momento em que estiver efetuando o download (veja também as precauções apresentadas abaixo). Agora, ligue ambos os robôs e execute os programas: execute primeiro aquele que está no robô escravo e depois o mestre.

Caso você tenha vários escravos, você deverá efetuar o download para cada um deles em separado (não o faça simultaneamente; veja abaixo). Assim, todos os robôs escravos executarão as mesmas ações.

Para propiciar a comunicação dos robôs entre si, nós definimos o que é chamado de um protocolo: decidimos que um 1 significa mover para a frente, um 2 mover para trás e um 3 parar. É muito importante definir tais protocolos de maneira cuidadosa, em particular quando se está lidando com um volume grande de troca de informações. Por exemplo, quando existem muitos escravos, você poderia definir um protocolo em que dois

números são enviados (com um pequeno tempo de atraso entre eles): o primeiro é o número do robô escravo e o segundo é a ordem a ser executada. Os robôs escravos primeiramente verificam o número e somente executam a ação caso aquele seja o seu número. (Isso requer que cada robô escravo tenha seu próprio número, o que pode ser feito fazendo com que cada escravo tenha um programa ligeiramente diferente em que, por exemplo, uma constante seja diferente.)

Elegendo um líder

Conforme visto acima, quando lidamos com vários robôs, cada um deve ter seu próprio programa. Seria muito mais fácil se pudéssemos efetuar o download do mesmo programa para todos os robôs. Mas a questão que se apresenta é: quem é o líder? A resposta é fácil: deixe os robôs decidirem por si mesmos. Deixe que eles elejam um líder e os demais o seguirão. Mas como fazer isso? A idéia é bastante simples. Deixaremos cada robô aguardar por um período de tempo aleatório e então enviar uma mensagem. O primeiro a enviar a mensagem será o líder. Esse esquema pode falhar caso dois robôs esperem por exatamente a mesma quantidade de tempo, mas isso é bastante improvável. (Você pode construir esquemas mais complicados que detectam isso e tentam uma segunda eleição.) Segue um programa que faz o que foi descrito:

```
task main()
{
  ClearMessage();
  Wait(200);                // garante que todos os robôs estejam ligados
  Wait(Random(400));        // aguarda um tempo entre 0 e 4 segundos
  if (Message() > 0)        // algum outro robô foi o primeiro
  {
    start escravo;
  }
  else
  {
    SendMessage(1);         // eu sou o mestre agora
    Wait(400);              // garante que todos os demais saibam disso
    start mestre;
  }
}

task mestre()
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}

task escravo()
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

Efetue o download deste programa para todos os robôs (um por um, não o faça simultaneamente; veja abaixo). Inicie os robôs ao mesmo tempo e veja o que acontece. Um deles deverá tomar o comando e os outros seguirão as ordens. Em situações bastante raras, nenhum deles se tornará o líder. Como indicado anteriormente, isso requer um protocolo mais elaborado para solucionar tais situações.

Precauções

Você deve ser bastante cuidadoso quando estiver lidando com vários robôs. Existem dois problemas: se dois robôs (ou um robô e um computador) enviarem informação ao mesmo tempo, as mesmas poderão ser perdidas; o segundo problema é que, quando o computador envia um programa para vários robôs ao mesmo tempo, isso causa problemas.

Vamos iniciar com o segundo problema. Quando você efetua o download de um programa para um robô, o robô diz ao computador se ele está recebendo corretamente (as partes do) o programa. O computador reage a isso enviando novos pedaços ou reenviando partes do programa. Quando dois robôs estão ligados, ambos irão iniciar dizendo ao computador se ele receberam corretamente o programa. O computador não entende isso (o computador não sabe que existem dois robôs!). Como resultado, coisas podem dar errado e o programa é corrompido. Os robôs não farão as coisas certas. *Sempre garanta que quando você estiver fazendo o download de programas, somente um robô esteja ligado!*

O outro problema é que somente um robô pode enviar uma mensagem em um determinado momento. Caso duas mensagens sejam enviadas exatamente ao mesmo tempo, elas serão perdidas. Além disso, um robô não pode enviar e receber mensagens no mesmo instante. Isso não é um problema quando somente um robô envia mensagens (existe somente um mestre) mas, por outro lado, pode se tornar um problema sério. Por exemplo, você pode imaginar a escrita de um programa em que o robô escravo envia uma mensagem quando ele bate em algo, de modo que o robô mestre tome uma decisão. Mas, caso o mestre envie uma ordem ao mesmo tempo, a mensagem será perdida. Para resolver isso, é importante definir seu protocolo de comunicação de tal modo que, no caso de uma comunicação falhar, ela seja corrigida. Por exemplo, quando o mestre envia um comando, ele deve receber de volta uma resposta do escravo. Caso ele não receba uma resposta em um intervalo de tempo esperado, ele reenvia o comando. Isso poderia resultar em um pedaço de código que se parece com isso:

```
do
{
  SendMessage(1);
  ClearMessage();
  Wait(10);
}
while (Message() != 255);
```

Aqui, 255 é usado para informar que o comando foi recebido pelo escravo.

Às vezes, quando você está lidando com vários robôs, você deseja que somente aquele robô que está mais perto receba o sinal. Isso pode ser feito por meio da inserção do comando `SetTxPower(TX_POWER_LO)` ao programa do robô mestre. Neste caso, o sinal de infra-vermelho enviado é muito baixo e somente um robô próximo e de frente para o mestre poderá escutá-lo. Isso é em particular útil quando estiver construindo um robô maior que utiliza mais de dois RCX. Use `SetTxPower(TX_POWER_HI)` para configurar o robô novamente com uma taxa de transmissão longa.

Resumo

Neste capítulo estudamos alguns aspectos básicos da comunicação entre robôs. A comunicação utiliza comandos para enviar, limpar e verificar mensagens. Vimos que é importante definir um protocolo referente à maneira que a comunicação acontecerá. Tais protocolos desempenham um papel crucial em toda forma de comunicação entre computadores. Vimos, também, que existem restrições na comunicação entre robôs, nos quais evidencia a importância de se definir bons protocolos.

XII. Mais comandos

A linguagem NQC tem alguns comandos adicionais. Neste capítulo discutiremos três tipos: o uso de temporizadores (*timers*), comandos para controlar o display e o uso de características do datalog do RCX.

Temporizadores

O RCX possui quarto temporizadores internos. Esses temporizadores avançam em incrementos de 1/10 segundos. Os temporizadores são numerados de 0 a 3. Você pode tornar zero o valor de um temporizador (zerar o temporizador) com o comando `ClearTimer()`e obter o valor corrente com `Timer()`. Abaixo, tem-se um exemplo do uso de temporizadores. O programa deixa o robô se movimentar de forma meio aleatória por 20 segundos.

```
task main()
{
  ClearTimer(0);
  do
  {
    OnFwd(OUT_A+OUT_C);
    Wait(Random(100));
    OnRev(OUT_C);
    Wait(Random(100));
  }
  while (Timer(0)<200);
  Off(OUT_A+OUT_C);
}
```

Você pode querer comparar este programa com aquele apresentado no Capítulo IV que fazia exatamente a mesma coisa. Entretanto, este programa com o uso de temporizadores se torna definitivamente mais simples.

Temporizadores são muito úteis em substituição ao comando `Wait()`. Você pode aguardar (*sleep*) por uma determinada quantidade de tempo zerando o temporizador e esperando até que seu valor alcance um valor determinado. Mas você também pode reagir a outros eventos (como por exemplo, gerados por sensores) enquanto o temporizador estiver executando. O seguinte programa é um exemplo disso. Ele deixa o robô se movimentar até que uma de duas coisas aconteça: 10 segundos sejam passados; ou o sensor de toque perceba algo.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  ClearTimer(3);
  OnFwd(OUT_A+OUT_C);
  until ((SENSOR_1 == 1) || (Timer(3) >100));
  Off(OUT_A+OUT_C);
}
```

Não se esqueça que os temporizadores trabalham em intervalos de 1/10 segundos, enquanto que o comando `Wait` utiliza intervalos de 1/100 segundos.

O display

É possível controlar o display do RCX de duas maneiras diferentes. Na primeira delas você pode indicar o que será mostrado no display: o clock do sistema, um dos sensores ou um dos motores. Isso é equivalente a utilizar o botão view do RCX. Para definir o tipo de informação a ser apresentada, utilize o comando `SelectDisplay()`. A seguir são mostradas todas as sete possibilidades, uma após a outra.

```
task main()
{
  SelectDisplay(DISPLAY_SENSOR_1); Wait(100);  // Entrada 1
  SelectDisplay(DISPLAY_SENSOR_2); Wait(100);  // Entrada 2
  SelectDisplay(DISPLAY_SENSOR_3); Wait(100);  // Entrada 3
  SelectDisplay(DISPLAY_OUT_A);    Wait
```

XIII. NQC: guia de referência rápido

Abaixo você encontrará uma lista com todas as estruturas, comandos, constantes, etc. do NQC. A maioria deles foi tratada nos capítulos anteriores, de modo que será dada somente uma breve descrição.

Instruções

Instruções	Descrição
`while (cond) body`	Executa o corpo zero ou mais vezes enquanto a condição for verdadeira
`do body while (cond)`	Executa o corpo uma ou mais vezes enquanto a condição for verdadeira
`until (cond) body`	Executa o corpo zero ou mais vezes até que a condição se torne verdadeira
`break`	Sai de uma estrutura do tipo while/do/until
`continue`	Pula a próxima iteração de uma estrutura do tipo while/do/until
`repeat (expression) body`	Repete o corpo um número de vezes específico
`if (cond) stmt1` `if (cond) stmt1 else stmt2`	Executa stmt1 se a condição for verdadeira. Executa stmt2 (caso exista) se a condição for falsa
`start task_name`	Inicia a tarefa especificada
`stop task_name`	Pára a tarefa especificada
`function(args)`	Chama uma função utilizando os argumentos passados
`var = expression`	Calcula a expressão e atribui o valor resultante para a variável
`var += expression`	Calcula a expressão e adiciona o valor resultante ao valor da variável
`var -= expression`	Calcula a expressão e subtrai o valor resultante do valor da variável
`var *= expression`	Calcula a expressão e multiplica o valor resultante pelo valor da variável
`var /= expression`	Calcula a expressão e divide o valor resultante pelo valor da variável
`var |= expression`	Calcula a expressão e executa a operação OR sobre a variável
`var &= expression`	Calcula a expressão e executa a operação AND sobre a variável
`return`	Retorna de uma função para o ponto que a chamou
`expression`	Calcula a expressão

Condições

Condições são usadas dentro de estruturas de controle para a tomada de decisões. Na maioria dos casos, a condição envolverá uma comparação entre as expressões.

Condição	Significado
`true`	sempre verdadeiro
`false`	sempre falso
`expr1 == expr2`	Testa se as expressões são iguais
`expr1 != expr2`	Testa se as expressões são diferentes
`expr1 < expr2`	Testa se uma expresão é menor do que a outra
`expr1 <= expr2`	Testa se uma expresão é menor ou igual do que a outra
`expr1 > expr2`	Testa se uma expresão é maior do que a outra
`expr1 >= expr2`	Testa se uma expresão é maior ou igual do que a outra
`! condition`	Negação lógica de uma expressão
`cond1 && cond2`	Lógica AND de duas condições (verdadeiro se, e somente se, ambas as condições forem verdadeiras)
`cond1 || cond2`	Lógica OR de duas condições (verdadeiro se, e somente se, pelo menos uma das condições for verdadeira)

Expressões

Existem diferentes valores que podem ser usados dentro de expresses, incluindo constantes, variáveis e valores de sensores. Observe que SENSOR_1, SENSOR_2 e SENSOR_3 são macros que se expandem para SensorValue(0), SensorValue(1) e SensorValue(2), respectivamente.

Valor	Descrição
número	Um valor constante (por exemplo "123")
variável	Uma variável nomeada (por exemplo "x")
`Timer(n)`	Valor do temporizador n, onde n pode ser 0, 1, 2 ou 3

`Random(n)`	Número randômico definido entre 0 e n (inclusives)
`SensorValue(n)`	Valor corrente do sensor n, onde n pode ser 0, 1 ou 2
`Watch()`	Valor do relógio do sistema
`Message()`	Valor da última mensagem de infra-vermelho recebida

Valores podem ser combinados utilizando operadores. Muitos dos operadores somente podem ser utilizados no cálculo de expressões constantes, o que significa que seus operandos devem ser constantes ou expressões envolvendo nada mais do que constantes. Os operadores são listados abaixo em ordem de precedência (da maior para a menor).

Operador	Descrição	Associatividade	Restrição	Exemplo
`abs()` `sign()`	Valor absoluto Sinal do operando	n/a n/a		`abs(x)` `sign(x)`
`++` `--`	Incremento Decremento	esquerda esquerda	Somente variáveis Somente variáveis	`x++ ou ++x` `x-- ou --x`
`-` `~`	Menos unário Negação Bitwise (unária)	direita direita	Somente constantes	`-x` `~123`

`*`
`/`
`%`
	Multiplicação
	Divisão
	Módulo
		esquerda
		esquerda
		esquerda
			Somente constantes	`x * y`
				`x / y`
				`123 % 4`

`+` `-`	Adição Subtração	esquerda esquerda		`x + y` `x - y`
`<<` `>>`	Deslocamento para a esquerda Deslocamento para a direita	esquerda esquerda	Somente constantes Somente constantes	`123 << 4` `123 >> 4`
`&`	Bitwise AND	esquerda		`x & y`
`^`	Bitwise XOR	esquerda	Somente constantes	`123 ^ 4`
`|`	Bitwise OR	esquerda		`x | y`
`&&`	Operador lógico AND	esquerda	Somente constantes	`123 && 4`
`||`	Operador lógico OR	esquerda	Somente constantes	`123 || 4`

Funções do RCX

A maioria das funções requer que todos os argumentos sejam expressões constantes (números ou operações envolvendo outras expressões constantes). As exceções são as funções que usam um sensor como argumento e aquelas que não usam qualquer expressão. No caso de sensores, o argumento deve ser um nome de um sensor: SENSOR_1, SENSOR_2 ou SENSOR_3. Em alguns casos existem nomes predefinidos (por exemplo, SENSOR_TOUCH) para constantes apropriadas.

Função	Descrição	Exemplo
`SetSensor(sensor, config)`	Configura um sensor.	`SetSensor(SENSOR_1, SENSOR_TOUCH)`
`SetSensorMode(sensor, mode)`	Define o modo do sensor.	`SetSensor(SENSOR_2, SENSOR_MODE_PERCENT)`
`SetSensorType(sensor, type)`	Define o tipo do sensor.	`SetSensor(SENSOR_2, SENSOR_TYPE_LIGHT)`
`ClearSensor(sensor)`	Limpa o valor do sensor.	`ClearSensor(SENSOR_3)`
`On(outputs)`	Liga uma ou mais saídas.	`On(OUT_A + OUT_B)`
`Off(outputs)`	Desliga uma ou mais saídas.	`Off(OUT_C)`
`Float(outputs)`	Deixa que a saída flutue.	`Float(OUT_B)`
`Fwd(outputs)`	Define a direção da saída para frente.	`Fwd(OUT_A)`
`Rev(outputs)`	Define a direção da saída para trás.	`Rev(OUT_B)`
`Toggle(outputs)`	Troca a direção da saída (Frente para trás e vice-versa)	`Toggle(OUT_C)`
`OnFwd(outputs)`	Liga em movimentação para frente.	`OnFwd(OUT_A)`
`OnRev(outputs)`	Liga no sentido reverso.	`OnRev(OUT_B)`
`OnFor(outputs, time)`	Liga por um período de tempo especificado por um número igual a 1/100 de	`OnFor(OUT_A, 200)`

	segundo. O tempo (*time*) pode ser uma expressão.	
`SetOutput(outputs, mode)`	Define o modo de saída.	`SetOutput(OUT_A, OUT_ON)`
`SetDirection(outputs, dir)`	Define a direção da saída.	`SetDirection(OUT_A, OUT_FWD)`
`SetPower(outputs, power)`	Define o nível de potência da saída (0-7). *power* pode ser uma expressão.	`SetPower(OUT_A, 6)`
`Wait(time)`	Aguarda por uma quantidade de tempo especificada em 1/100 segundos. *time* pode ser uma expressão.	`Wait(x)`
`PlaySound(sound)`	Toca um som específico (0-5).	`PlaySound(SOUND_CLICK)`
`PlayTone(freq, duration)`	Toca uma nota de uma freqüência especificada por um determinado período de tempo (em 1/10 segundos).	`PlayTone(440, 5)`
`ClearTimer(timer)`	Define o temporizador (0-3) em 0.	`ClearTimer(0)`
`StopAllTasks()`	Pára todas as tarefas atualmente em execução.	`StopAllTasks()`
`SelectDisplay(mode)`	Seleciona um de 7 modos do display: 0: relógio do display, 1-3: valores de sensor, 4-6: configuração de saída. *mode* pode ser uma expressão.	`SelectDisplay(1)`
`SendMessage(message)`	Envia uma mensagem em infra-vermelho (1-255). *message* pode ser uma expressão.	`SendMessage(x)`
`ClearMessage()`	Limpa o buffer de mensagens de infra-vermelho.	`ClearMessage()`
`CreateDatalog(size)`	Cria um novo datalog de um tamanho (*size*) especificado.	`CreateDatalog(100)`
`AddToDatalog(value)`	Adiciona um valor ao datalog. *value* pode ser uma exepressão.	`AddToDatalog(Timer(0))`
`SetWatch(hours, minutes)`	Define o valor do relógio do sistema.	`SetWatch(1,30)`
`SetTxPower(hi_lo)`	Define o nível de potência do transmissor de infra-vermelho para um nível baixo ou alto.	`SetTxPower(TX_POWER_LO)`

Constantes do RCX

Existem várias constantes nomeadas (constantes simbólicas) que podem ser passadas para as funções de modo a ajudar a tornar o código mais legível. Quando possível, utilize uma constante nomeada ao invés de um valor bruto.

Configurações do sensor para SetSensor()	`SENSOR_TOUCH, SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_CELSIUS, SENSOR_FAHRENHEIT, SENSOR_PULSE, SENSOR_EDGE`
Modos para SetSensorMode()	`SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELSIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION`
Tipos para SetSensorType()	`SENSOR_TYPE_TOUCH, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT, SENSOR_TYPE_ROTATION`
Saídas para On(), Off(), etc.	`OUT_A, OUT_B, OUT_C`
Modos para SetOutput()	`OUT_ON, OUT_OFF, OUT_FLOAT`
Direções para SetDirection()	`OUT_FWD, OUT_REV, OUT_TOGGLE`

Potência de saída para SetPower()	OUT_LOW, OUT_HALF, OUT_FULL
Sons para PlaySound()	SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP
Modos para SelectDisplay()	DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C
Nível da potência da taxa de transmissão para SetTxPower()	TX_POWER_LO, TX_POWER_HI

Palavras-chave

Palavras-chave são aquelas palavras reservadas pelo compilador do NQC. É um erro utilizar essas palavras para nomes de funções, tarefas ou variáveis. A seguir, tem-se uma lista das palavras-chave usadas: **__sensor, abs, asm, break, const, continue, do, else, false, if, inline, int, repeat, return, sign, start, stop, sub, task, true, void, while.**

XIV. Considerações finais

Caso você tenha trabalhado à sua maneira por todo este tutorial, você pode se considerar um *expert* em NQC. Caso você ainda não tenha feito isso, é hora de começar experimentando os exemplos. Com criatividade no projeto e programação, você pode instruir os robôs Lego a fazer coisas maravilhosas.

Este tutorial não cobre todos os aspectos do RCX Command Center. É recomendável que você leia a documentação em algum momento. Além do mais, o NQC encontra-se em desenvolvimento. Versões futuras devem incorporar funcionalidades adicionais. Muitos conceitos de programação não foram tratados neste tutorial. Em particular, não consideramos a aprendizagem dos robôs nem outros aspectos da inteligência artificial.

Também é possível controlar um robô Lego diretamente do PC. Isso requer que você escreva um programa em uma linguagem tal como, Visual Basic, Java ou Delphi. É possível, também, deixar um programa desses rodar juntamente com um outro programa em execução no RCX. Essa combinação é muito poderosa. Caso você esteja interessado nessa maneira de programar os seus robôs, o melhor a fazer é começar efetuando o download da referência técnica no site do Lego Mindstorms no site.

http://www.legomindstorms.com/

A internet é uma fonte repleta de informações adicionais. Alguns outros importantes “primeiros passos” estão disponíveis por meio de links na minha própria home page:

http://www.cs.uu.nl/people/markov/lego/

e em LUGNET, a rede de grupos de usuários LEGO® (não oficial):

http://www.lugnet.com/

Uma gama de informações podem também ser encontradas nos newsgroups lugnet.robotics e lugnet.robotics.rcx.nqc em lugnet.com.