Apostila NXC

NXT: CÓDIGOS E ESTRUTURA

por Rachel L. Tuma

Prefácio

A apostila NXC vem até você trazer de forma simples e clara as informações básicas das quais você precisa para entender, montar e programar o seu LEGO NXT, não antes de lhe contar um pouco sobre a história deste meio promissor e contagiante, hoje em sua sala de aula,a robótica

NXC, a linguagem que vamos aprender juntos é semelhante à linguagem C que você provavelmente já ouviu falar, do inglês *not exactly C foi concebida* por **John Hansen** especialmente para robôs LEGO, trata-se de uma ferramenta fácil para programar, por isso se você nunca escreveu um programa não se preocupe, esta apostila irá guiá-lo em seus primeiros passos. Para que seu trabalho torne-se ainda mais simples existe o ***Bricx Command Center*** (BricxCC), também criado por Hansen, utilitário que irá nos auxiliar na escrita e transferência do código para o robô.

Você pode adquirir gratuitamente o BricxCC versão3.3.7.16 a partir da pagina da web, http://bricxcc.sourceforge.net/nxc/ , o utilitário roda em PCs com Windows (95, 98, ME, NT, 2K, XP, Vista).

Sumário

01. A história 4
02. Programação de robôs.....5
2.1 As Linguagens de Programação.....6
03. A Linguagem NXC.....8
3.1 Identificadores em NXC.....9
3.2 Palavras- chave em NXC.....9
3.2.1. const.....10
3.2.2. enum (Enumerations).....10
3.2.3 typedef.....11
3.3 Instruções e Blocos em NXC.....12
3.3.1 Estrutura do Programa.....12
3.4 Tasks (tarefas).....12
3.4.1 Sub-rotinas.....14
3.5 Variáveis.....14
3.5.1 Tipos de dados em NXC.....15
3.5.2 Definindo Macros.....17
3.5.3 Definindo Constantes.....18
3.6 Arrays.....19
3.7 Expressões e Operadores.....20
3.7.1 Atribuição.....21
3.8 Estruturas de controle.....22
3.8.1 O comando if.....22
3.8.2 O comando switch.....23
3.8.3 O laço while.....24
3.8.3.1 O laço do- while.....24
3.8.4 O laço for.....25
3.8.5 O comando repeat.....25
3.8.6 O comando goto.....26
3.8.7 O comando until.....26
3.9 Outros Comandos.....26
3.9.1 Comando *start*.....26
3.9.2 Comando *stop*.....27
3.9.3 Comando continue (*continuar)*.....27
3.9.4 Comando *return*.....27
4.0 O Pré – processador.....27
4.0.1 #include.....28
4.0.2 #define.....28
4.0.3 # undef.....28
4.0.4 ## (Concatenação).....29
4.0.5 #import.....29
4.0.6 #download.....30
4.0.7 Compilação Condicional.....30
4.1 Funções de tempo.....31
4.2 Funções de controle de programa.....32

4.3 Funções numéricas.....34
4.4 Funções para String.....35
4.5 Fazendo uso dos recursos de entrada (INPUT) – Sensores.....36
4.5.1 Informações do sensor.....37
4.6 Fazendo uso dos recursos de saída (OUTPUT) – Motores.....37
4.7 Fazendo uso dos recursos de Som.....42
4.8 Fazendo uso dos recursos de Display.....43
HORA DE POR EM PRÁTICA!.....46

01- A História:

Desde os tempos mais remotos o homem buscou soluções para facilitar, cada vez mais, a sua vida, épocas onde mitos faziam referência a mecanismos que ganhavam vida. Já na civilização grega mesmo não existindo sistemas complexos de produtividade, vislumbrava-se o que hoje conhecemos como robôs, eram os primeiros modelos, impressos em figuras com aparência humana e/ou animal, que usavam sistemas de pesos e bombas pneumáticas.

Robôs de acordo com a ISO 9283[1] são máquinas reprogramáveis e multifuncionais formadas por um mecanismo, incluindo diversos graus de liberdade, na maioria das vezes tendo a aparência de um ou diversos braços terminando num pulso capaz de segurar ferramentas, peças ou dispositivos."

O termo *robot* tem origem na palavra Checa *robota*, que significa "trabalho forçado/ serviço compulsório/servo", O precursor do termo foi Karel Capek, novelista e escritor, que o usou pela primeira vez em 1920 em um de seus mais prestigiados romances "*Rossum`s Universal Robots*"(**R.U.R**), essa peça se referia aos robôs serviçais de um cientista (Rossum). Os robôs de Carpek eram máquinas incansáveis, semelhantes fisicamente ao homem, que realizavam tarefas para tornar a vida mais simples e confortável. Histórias fictícias feitas para impressionar o público, mas acabou por deixar marcado nos pensamentos que robots terão sempre aspecto humano.

Desde então surgiram obras em que eles, os "astros do futuro", estavam presentes, como os filmes Metrópolis (1926), O Dia Em Que A Terra Parou (1951), Guerra Nas Estrelas (1977), 2001: Uma Odisséia No Espaço (1968), O Império Contra-Ataca (1980), e muitos outros que ate hoje enchem os contos de ficção cientifica com sistemas eletrônicos super complexos e vantajosamente passiveis de estimular idéias, contribuindo para o desenvolvimento da ciência.

A utilização de dispositivos automáticos vem crescendo há muito tempo, eles executam tarefas repetitivas e cansativas, atuando principalmente em áreas de atividades demasiadamente perigosas para qualquer ser humano.

Nesse contexto, a robótica pode ser definida como o estudo e a utilização de robôs. Enquanto ciência, ela preocupa-se com a montagem e posterior programação dos mecanismos, envolvendo mecânica, elétrica, eletrônica e computação. O termo foi usado foi pela primeira vez pelo cientista e escritor **Isaac Asimov**, em 1942, numa pequena história intitulada "Runaround". Asimov anos depois, publicou a obra "Eu Robô" *(I Robot*) que continha vários contos de ficção científica com aspirações bem futuristas a respeito dos robôs, contos que provavelmente serviram de inspiração para diversos sucessos do cinema. Este autor propôs a existência de três leis aplicáveis à robótica, às quais acrescentou, mais tarde, a lei zero.

LEIS DA ROBÓTICA

0- Um robô não pode fazer mal à humanidade e nem, por inação, permitir que ela sofra algum mal.
1 – Um robô não pode ferir um ser humano ou, por omissão, permitir que um ser humano sofra algum mal.
2 – Um robô deve obedecer as ordens que lhe sejam dadas por seres humanos, exceto nos casos em que tais ordens contrariem a Primeira Lei.
3 – Um robô deve proteger sua própria existência, desde que tal proteção não entre em conflito com a Primeira e a Segunda Lei.

02 – Programação de robôs

Um robô móvel é definido assim: agente artificial com capacidade de locomoção e capaz de atuar em um ambiente extraindo informações do mesmo e utilizando esse conhecimento do espaço no qual está inserido para deslocar-se com segurança, atuando e executando tarefas. Em contraste com os robôs móveis, os industriais geralmente são presos a uma superfície fixa formados por um braço articulado e um dispositivo de atuação. Ambos necessitam, porém, de um sistema para programação de modo a garantir o controle adequado sobre seus movimentos, viabilizando- os a realizar sua missão.

O programa de um dispositivo automático é um conjunto de instruções escritas na linguagem de programação do seu controlador, as quais se destinam a realização de tarefas por parte do mesmo. É possível definir o algoritmo de trabalho das seguintes maneiras:

- ✓ Programação textual - comandos escritos na linguagem de programação do controlador do robô; Por meio dessas linguagens, podem-se controlar os movimentos do robô, longe do ambiente de trabalho real da máquina

- ✓ Programação por aprendizagem - usando uma caixa de controle ("TeachPendant") move-se o braço do robô ao longo do espaço de trabalho a fim de definir pontos, estes pontos uma vez "ensinados" orientam a trajetória a ser realizada pelo dispositivo em determinada ação .

Os métodos de programação dividem-se em dois tipos:

a) **On-line**: Programação por Aprendizagem – é necessário o uso do robô para gerar o algoritmo.

b) **Off-line**: Programação Textual – a presença do dispositivo não é necessária, exceto na fase final de teste e implementação.

Também chamado "fora-de-linha", o método *off-line*, envolve a escrita do programa em uma linguagem de programação, por meio da qual é possível controlar os movimentos do robô longe do seu ambiente real de trabalho.

A principal desvantagem do método *on-line* se comparado ao *off-line*, reside na necessidade de interromper o trabalho normal de produção da máquina para "ensinar-lhe" uma nova tarefa. A maior vantagem está no fato de ser bastante fácil de fazer e não requerer nenhum conhecimento especial de programação ou treino.

2.1 As Linguagens de Programação

- O que são Linguagens de Programação? –

  Entende-se por linguagem de programação o método padronizado para expressar instruções para um computador por meio de um conjunto de regras sintáticas e semânticas usadas para construir um programa de computador. As linguagens permitem que um programador especifique precisamente sobre quais dados um computador vai atuar, como estes dados serão armazenados ou transmitidos e quais ações devem ser tomadas em determinadas circunstâncias. Desta forma então, o código fonte é o conjunto de palavras escritas de forma ordenada, que contém **instruções** em uma determinada linguagem de programação (ROCHA, 2006).

Durante seu desenvolvimento industrial, alguns equipamentos programáveis se beneficiaram com a padronização de suas linguagens de programação. Essa padronização é altamente vantajosa, permitindo uma rápida migração de programas entre plataformas e fazendo com que o aprendizado da linguagem seja útil em uma grande quantidade de equipamentos, dentre estes estão os Controladores Lógicos Programáveis (CLP) que usam, por exemplo, a linguagem LADDER e os Comandos Numéricos Computadorizados (CNC) que empregam em sua maioria a linguagem conhecida popularmente como Código G.

Este cenário é realmente animador, no entanto para os robôs o cenário é diferente: esses equipamentos possuem grandes diferenças entre suas configurações além de poderem ser aplicados em tarefas extremamente distintas, dessa forma cada fabricante acabou desenvolvendo sua própria linguagem, fato que dificulta um pouco as coisas do ponto de vista da aplicação de conhecimentos. Então acabamos por ser remetidos à necessidade de nos manter atualizados, sempre.

Como já foi visto, quando utilizamos linguagens de programação, trata-se do método off-line de programar. Nesse sentido, a programação off-line de robôs é muito parecida com a implementação de programas de computadores usando as linguagens C, C++,Java e outras. Verificamos inúmeras vantagens ao utilizar esse método, como por exemplo: a seqüência de operações e movimentos do robô podem ser otimizados ou facilmente melhorados, uma vez que o programa básico já foi desenvolvido. Da mesma forma, os programas podem ser mais facilmente modificados e verificados.

Essas linguagens podem ser divididas em três categorias fundamentais, três tipos diferentes de níveis de programação, baseados nos diferentes tipos de ação básica especificados em cada um destes níveis: a)Programação a nível das juntas; b) programação a nível do robô; c) programação de alto nível {1. a nível de objetos. 2. a nível de tarefas;} .

Atualmente, existem centenas de linguagens de robôs disponíveis comercialmente. Muitas delas baseadas em linguagens clássicas tais como Pascal, C, Modula-2, BASIC, e Assembler.

Há uma máxima muito utilizada nesse contexto: “Existem tantas linguagens quanto existem variedades de robôs”. Veja alguns exemplos:

- Linguagens de programação a nível robô:

    - AL – Universidade de Stanford, USA (1974);
    - AML – IBM (1982);
    - LM – Universidade de Grenoble (1980);
    - VAL –II – Univations (1983);
    - V^+ - Adept (1994);

- Linguagens a nível objetos:

    - LAMA – MIT Laboratório de investigação (1976);
    - AUTOPASS – IBM (1977);
    - RAPT – Universidade de Edimburgo (1978 – 86);

- Outros exemplos :

  São elas: *ACL (Advanced Command Language), AR-BASIC (da American Robot Corporation), ARLA (ASEA Robot Language), CAP 1 (Conversational Auto Programming 1, da Fanuc), HELP (da General Eletric), ILMR (Intermediate Language for Mobile Robots), JARS (do JPL - Laboratório de Propulsão a Jato da NASA), KAREL (da Fanuc), PASRO (PAScal for RObots), POINT (da Milan Polytechnic – 1977), PROWLER.*

- LÉXICO -

03- A linguagem NXC

- Consideremos que NXC é formada por duas partes separadas: **a linguagem**, que define a sintaxe utilizada quando na escrita do algoritmo; a **API** (*Application Programming Interface*) **NXC** que por sua vez define o funcionamento do sistema, constantes e macros que serão utilizados em seus programas.

Assim como em C e C++ :

- NXC é *case-sensitive*, ou seja, diferencia letras maiúsculas de minúsculas, desta forma " AbC" não é o mesmo que "aBc", o mesmo vale para as palavras chaves e comandos do seu código, " While" é diferente da palavra chave válida, "while".
- Comentários em NXC: há duas formas para fazer comentários ao longo do seu programa:

a) Comentário de bloco ou multi-linhas. Inicie /* finalize */ . O comentário se estenderá por várias linhas até que o finalize com */ .Por exemplo:

```
/* line 1: O robô deve seguir em frente*/

/* line 2:
      O robô deve virar à esquerda*/

/* Outro  comentário.....
   Terá várias linhas.....
  Até que.....  */
```

b) Comentário de Linha. Inicie com / /, este comentário finaliza com nova linha. Exemplo:

```
// ajuste a velocidade
// outro comentário.....
```

OBS: O compilador não reconhece o texto após as **//** ou */ como instrução de código, ao invés disso ele o ignora e o trata como comentário. Este constitui ferramenta para que o programador organize e identifique seu código facilitando sua compreensão principalmente em programas extensos.

- O ponto-e-vírgula: a sintaxe da linguagem NXC determina o uso de ponto-e-vírgula, pois como sabemos um programa é uma lista de instruções a serem executadas pelo computador por isso é necessário distinguir um comando do outro, para tanto ao final de cada comando deve-se fazer uso deste recurso, se assim não for feito ocorrerá um erro de sintaxe e seu programa não passará na compilação.

3.1 Identificadores em NXC

Identificadores são usados para representações de nomes de variáveis, tarefas, funções e sub-rotinas. Os identificadores em NXC devem começar com uma letra (em maiúsculo ou minúsculo) ou um *underscore* "_".Os caracteres seguintes podem usar números de 0 a 9.

NXC é *case-sensitive* portanto o identificador **turn_left** não é o mesmo que **Turn_left**. Os identificadores não podem ter nomes iguais às palavras-chave, como: task, while, int, do, if, etc. Discutiremos mais sobre estas palavras mais tarde.

DICA:

a) Para identificadores com mais de uma palavra, a primeira letra da segunda palavra deve vir em maiúsculo. Por exemplo:

doSquares – repeatSong

b) Evite o uso de *undescores* no início de um identificador. Por exemplo:

_NomeDaTarefa.

3.2 Palavras- chave em NXC

Palavras-chave são identificadores que foram reservados para serem usados em propósitos específicos em NXC. Não se podem usar estes identificadores como nomes de variáveis, tarefa ou sub-rotinas. Abaixo, temos a lista com as palavras-chave em NXC.

__RETURN__	case	inline	struct
__RETVAL__	char	int	sub
__STRRETVAL__	const	long	switch
__TMPBYTE__	continue	mutex	task
__TMPWORD__	default	repeat	true
__TMPLONG__	do	return	typedef
abs	else	safecall	unsigned
asm	false	short	until
bool	for	sign	void
break	goto	start	while
byte	if	string	

Figura 1 - palavras-chave

Obs: true e false não são palavras-chave,no entanto, são palavras-reservadas, deste modo, não é permitido seu uso para atribuir nomes a variáveis, tarefas ou Sub- rotinas.

3.2.1. const

A palavra-chave *const* é usada para declarar uma variável de modo que esta não possa ter seu valor alterado depois que ele é inicializado.

A inicialização deve ocorrer no momento da declaração da variável. Veja o exemplo:

```
const int constante = 23;   // declaração e inicialização de uma constante inteira

task main()
 {
      int x = constante;

      constante ++; // erro – você não pode modificar o valor de uma const
}
```

3.2.2. enum (Enumerations)

As *enumerations* definem uma nova tipologia de variável e limita os seus valores, constituindo um método de definir constantes. Sua forma geral é a seguinte:

enum [identificador] {constante = [valor], constante [valor],... }

O tipo enumerado é composto dos elementos de uma lista de nomes. O *valor* é opcional. Veja o exemplo:

```
enum dias { dom, seg, ter, qua, qui, sex, sab }  // dom, seg… são variáveis do tipo dias
```

Uma maneira simples de interpretar uma *enumeration* é imaginá-la como uma matriz de uma linha onde temos o nome da linha e as várias células na linha. Cada enumerado (constante enumerada) tem um valor inteiro, se este **não** vier **especificado** ele **começa** em **zero** Exemplo:

enum cor {preto, azul, verde, roxo, branco }
0 1 2 3 5

Mas podemos definir valores para nossa linha de tipos:

enum carro {pneu=5, freio, amortecedor=27, filtro, farol }

Os nomes sem o sinal de igualdade são incrementados de 01(uma unidade) a partir do valor do nome anterior. Então a lista de tipos do último exemplo ficará assim:

enum carro {pneu=5,freio, amortecedor=27, filtro, farol }
6 28 29

3.2.3 typedef

NXC permite definamos novos nomes aos tipos de dados que estão sendo criados. Para definir esses nomes utiliza-se o comando *typedef*, o qual é utilizado apenas para definir um novo nome para um tipo já existente e não para introduzir um novo tipo. A sua forma geral é:

typedef *tipo* nome;

onde *tipo* é qualquer tipo de dados permitido pela linguagem e *nome* é o novo nome para esse tipo. O novo será uma opção ao nome antigo e não uma substituição.

Você pode usar *typedef* para definir nomes mais curtos e significativos para os tipos já definidos pela linguagem ou para tipos que você declarou. Veja um exemplo:

```
enum cor { azul = 1,verde,branco };

 typedef enum cor tipo_cores ;  // transformamos 2 palavras em uma -> tipo_cores

task main()
{

/*Agora usando o novo tipo, sem o typedef teríamos que colocar enum cores */

        tipo_cores cor = verde ;

        /* TextOut não será executado */
        if(cor == 1){

                TextOut (0, LCD_LINE1, “azul”);
        }

        /* TextOut será executado */
        if(cor == 2){

                TextOut (0, LCD_LINE2, “verde”);
        }

        /* TextOut não será executado */
        if(cor == 3 ){

                TextOut (0, LCD_LINE3, “branco”);
       }
}
```

3.3 Instruções e Blocos em NXC

Uma instrução é formada de uma ou mais linhas e devem ser terminadas por ponto-e-vírgula. Um exemplo simples de instrução pode ser:

```
OnFwd(OUT_AC, 75);   // liga motores conectados às portas A e C.
```

Um bloco é composto por uma ou mais instruções ordenadas entre chaves indicando que compõem o mesmo comando. Blocos podem ser organizados em agrupamentos distintos indefinidamente. Qualquer quantidade de espaços em branco é permitida. Um exemplo de bloco:

```
.
.
.
repeat(4)
{
   OnFwd(OUT_AC, 75);
   Wait(MOVE_TIME);
   OnRev(OUT_C, 75);
   Wait(TURN_TIME);

}
```

3.3.1 Estrutura do Programa

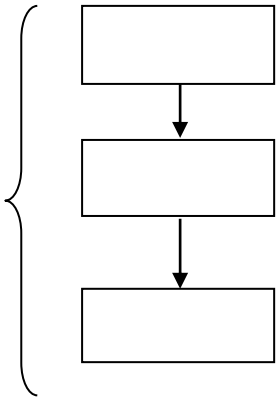
Um programa NXC é composto de blocos de código e variáveis. Existem dois tipos distintos de blocos: **tarefas** e **funções**. Cada tipo de bloco de código tem suas próprias características, no entanto eles compartilham uma estrutura comum.

3.4 Tasks (tarefas)

NXC introduziu o conceito de tarefa (*task*), que corresponde diretamente a um fluxo seqüencial de comandos de controle do *Brick* NXT ao qual dar-se o nome *Thead*, do inglês, **linha de execução**. O NXT suporta a execução de mais de uma tarefa concorrentemente, por isso NCX é *multithread*, entenda o que significa:

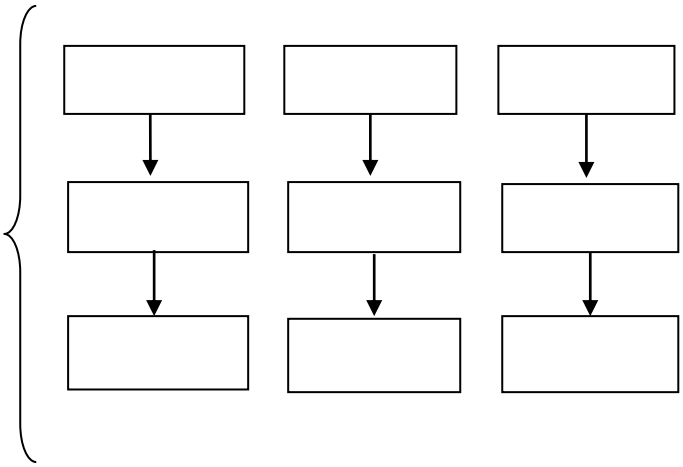
- Definição: um thread é um fluxo único de controle seqüencial dentro de um programa.

Programa - Sequência



- NXC suporta mais de um thread no mesmo programa.

Programa - Multithread



- O suporte à thread é fornecido pelo sistema operacional (SO). Aos sistemas que suportam uma única linha de execução dar-se o nome monothread e àqueles sistemas que suportam múltiplas linhas de execução, multithread.

O nome de sua tarefa poderá ser qualquer identificador que não conflite com quaisquer das "palavras reservadas", o corpo é composto por diversos comandos, ela não poderá ser interrompida por outra tarefa, mas somente finalizando todas as tarefas. Definimo-la usando a palavra-chave task, como na *syntax* a seguir:

```
task nome ()
{
    / / o código da sua tarefa é escrito aqui
}
```

Semelhantemente ao C, NXC é uma linguagem estruturada, isto é, composta por blocos de código que se interligam através de três mecanismos básicos, que são **sequência**, **seleção** e **Iteração**.

- **Estruturas de sequência:** uma tarefa é executada após a outra, linearmente.
- **Estruturas de decisão:** a partir de um teste lógico, determinado trecho de código é executado, ou não.
- **Estruturas de iteração:** a partir de um teste lógico, determinado trecho de código é repetido por um número finito de vezes.

Desta forma é necessário ter uma tarefa principal chamada "*main*", que será o ponto de partida do programa. O número Maximo de tarefas é 256.

3.4.1 Sub-rotinas

Os programas NXC podem ter múltiplas tarefas, como já sabemos. Além disso é possível criar partes de código em sub-rotinas que você pode usar em diferentes lugares dentro do seu programa.

> DICA:
>
> Usar tarefas e sub-rotinas torna os programas mais compreensíveis e mais compactos.

Você poderá fazer uso de sub-rotinas sempre que precisar de uma mesma instrução do seu código várias vezes durante seu programa. Para tanto coloque a(as) instrução(ões) desejada(s) dentro de sua sub-rotina e dê-lhe um nome, como no exemplo abaixo:

```
sub turn_left()
        {
            OnRev(OUT_C, 75); Wait(900);
            OnFwd(OUT_AC, 75);
         }
```

O nome atribuído poderá ser quaisquer identificadores que não conflitem com as “palavras reservadas”.

3.5 Variáveis

Variáveis são localizações de memória nas quais podemos guardar um valor, este valor poderá ser utilizado em diferentes lugares e poderá ser modificado.

As variáveis possuem tipos específicos que dizem ao computador que tipo de dado será armazenado e quanta memória ele requer ou quanta memória deve ser reservada. Em seu programa NXC as variáveis são referenciadas por nomes significativos que você atribuir. Para nomear suas variáveis fique atento às seguintes considerações:

- O nome de uma variável pode ser qualquer identificador desde que respeite as “palavras reservadas”.
- A nomenclatura das variáveis é *case sentitive*.
- O primeiro caractere do nome não ser um número, os caracteres seguintes poderão ser qualquer caractere alfanumérico.
- Jamais devem conter espaço de tabulação, para substituí-los utilize o caractere *underscore* “_” ex: *move_time*.

Para utilizá-las em seu programa o compilador deve reconhecê-las por isso as variáveis devem ser declaradas, isto é feito antes da tarefa principal (*task main*) usando a palavra-chave para o tipo desejado seguido de espaço e o nome atribuído a ela terminando por um ponto-e-vírgula (';'). Como no exemplo abaixo:

```
int nome_variavel;
task main()
{
    // instruções
}
```

DICAS:

1- Você poderá atribuir um valor inicial na declaração utilizando o operador de atribuição (' = ') após o nome da variável. Veja o exemplo abaixo:

```
int nome_variavel = 1;
task main()
{
  Repeat(5)
  {
    nome_variavel =  nome_variavel +1;
  }
}
```

2- Atribua nomes com significado para suas variáveis. Se usar uma variável para armazenar a velocidade do robô , declare-a com o nome 'velocidade' ao invés de uma letra qualquer, 'x' por exemplo.

3- É recomendável que seja declarada uma variável por linha, ao invés de várias na mesma linha. Por exemplo:

```
int vel;
int time;
```

e não: **int** vel, time;

3.5.1 Tipos de dados em NXC

O tipo define o conjunto de valores que a variável pode armazenar bem como o conjunto de operações que o programa poderá realizar com os dados.

As variáveis em NXC podem ser dos seguintes tipos:

Tipo da Variável	Informações
bool	8 bit unsigned
byte,unsigned char	8 bit unsigned
char	8 bit signed

unsigned int	16 bit unsigned
short int	8 bit signed
unsigned long	32 bit unsigned
long	32 bit signed
mutex	Tipo especial usado exclusivamente para code Access
string	Array de bytes
struct	Suporta todos os tipos de dados
Arrays (arranjo)	Arrays de qualquer tipo

Entendamos os tipos de dados em NXC:

- **bool:** ”bool” é a declaração de uma variável do tipo **Boolean**. Este é o tipo de dado mais simples em NXC podendo assumir apenas dois valores booleanos: **true** ou **false**.

- **char:** o tipo **char** armazena valores de caracteres. Um caractere é representado por um byte (08 bits) na memória, logo é possível representar 256 (2^8) caracteres distintos. Para representar um caractere usam-se apóstrofos. Para representar a letra "a" fazemos assim: 'a'.

- **byte:** Em NXC o tipo byte é um valor de 8 bits unsigned. Este tipo pode armazenar valores de zero a UCHAR_MAX (255). Você também pode definir uma variável unsigned 8-bit usando a palavra-chave unsigned seguido do tipo *cha*r.

- **int:** armazena números inteiros positivos e negativos.

- **mutex:** O tipo *mutex* é um valor de 32 bits usado para sincronizar o acesso aos recursos compartilhados entre as tarefas. Nunca declaramos uma variável *mutex* dentro de uma tarefa ou de uma função. Ela é projetada para variáveis globais de forma que todas as tarefas ou funções passam adquiri-la ou não, a fim de obter acesso exclusivo a um recurso que outras tarefas ou funções também estão tentando utilizar.

- **string:** Uma String é uma seqüência de caracteres (do tipo char) entre aspas (“ ”). Uma variável capaz de armazenar uma string deve ser declarada informando-se qual o número máximo de caracteres que ela poderá armazenar. Por exemplo:

char nome [10];
//a variável poderá armazenar uma string de até **9** caracteres.

Entenda os modificadores: os modificadores de tipo modificam o intervalo e valores que uma variável pode armazenar, para tanto colocamos os modificadores diante do tipo da variável no momento da declaração. Veja no exemplo:

```
unsigned int cont_voltas;
short int cor_branca;
```

- **modificador unsigned:** Na representação de uma variável do tipo int, o bit mais significativo do valor é o *bit* de sinal (positivo, negativo), o modificador unsigned faz com que o compilador utilize esse *bit* para representar valores positivos maiores. Então uma variável do tipo unsigned int pode armazenar valores no intervalo de 0 a 65.535.
- **modificador long:** os valores do tipo int são representados utilizando 16 *bits*, onde o *bit* mais significativo é o *bit* de sinal.O modificador long faz com que o compilador utilize 32 *bits*(4 *bytes)* para representar os inteiros.

3.5.2 Definindo Macros

- Podemos incluir inúmeras instruções no código-fonte que não são realmente parte da linguagem de programação NXC. Elas são chamadas de *diretivas do pré-processador.*
- *A diretiva # **define:** define um identificador (nome da macro) e um valor correspondente a este (uma string) que o substituirá sempre que for encontrado no programa.*
- *# defines* são declarações pré-processadas. Quando você os compilar, serão substituídos pelos valores pré-definidos.

Macros são de dois principais tipos:

a) Objeto: normalmente utilizado para dar nome a constantes.
b) Função: definir uma função.

Um uso para diretiva #define é o de definir uma macro. A Macro tipo função tem a seguinte forma geral:

#define nome_da_macro sequência_de_argumentos

Quando você usa esta diretiva, você está dizendo para o compilador que, ao encontrar o nome_da_macro no programa a ser compilado, ele deve substituí-lo pelo conjunto de argumentos fornecidos. Isto é muito útil para deixar o programa mais geral além de tornar a execução mais rápida. Veja o exemplo da **Macro tipo função**:

```
#define TURN_AROUND \  (caractere para continuar  a definição em nova linha)
OnRev(OUT_B, 75); Wait(3400);OnFwd(OUT_AB, 75);
task main()
{
OnFwd(OUT_AB, 75);
Wait(1000);
turn_around;
Wait(2000);
turn_around;
Wait(1000);
turn_around;
Off(OUT_AB);
}
```

DICA:

1 - Usar sempre maiúsculas ao definir macros, seja ela do tipo objeto ou uma função. Este padrão facilita a leitura e manutenção do código.

3.5.3 Definindo Constantes

Como vimos anteriormente, variáveis são como "caixas" que guardam informações. Enquanto **constantes** são caixas cujos valores não podem ser alterados, como o próprio nome já diz, eles são constantes ao longo do programa.

Para criar uma constante existe o comando *#define* que, em geral é colocado no início do programa e tem a seguinte forma geral de uma **Macro Objeto**:

#define nome_da_constante string (sem ponto-e-vírgula)

Definição de constantes é bom por dois motivos: torna o programa mais legível, e é mais fácil mudar os valores. Veja o exemplo:

```
#define MOVE_TIME 1000
#define TURN_TIME 360

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

Quando o pré-processador NXC encontra o nome da constante MOVE_TIME no seu programa ele o substitui pelo valor correspondente pré-definido.

3.6 Arrays

> A palavra *array* não tem uma boa tradução para o português, mas pode ser entendida como: "conjunto de dados posicionado em linha". É comum vermos palavras como vetor, matriz ou fila, associados à *array*, embora estes estejam ligados a outros conceitos.

Na declaração de variáveis, geralmente utilizamos um identificador e um tipo de dados. Para utilizarmos uma variável, deve-se chamá-la pelo nome a ela atribuído. Agora, considere a seguinte situação: você precisa utilizar varias variáveis para o mesmo fim, desta forma a declaração e a inicialização de tantas variáveis se tornaria uma tarefa um tanto tediosa.

Em NXC, assim como em outras linguagens de programação podemos utilizar uma variável para armazenar e manipular uma lista de dados com maior eficiência. Este tipo de variável é chamado de **array**. Podemos definir *Arrays* como uma coleção de um ou mais objetos, do mesmo tipo, armazenados em um bloco contínuo de memória, dividido em certa quantidade de **posições**.

Variável com cinco posições:

0	1	2	3	4	5
1	**2**	**3**	**4**	**5**	**6**

Os elementos de uma *array* são identificados por sua posição (chamado de índice). O primeiro elemento tem um índice 0, o segundo tem índice 1...

Imagine um *array* como uma variável esticada, a qual tem um nome que a identifica e pode conter mais de um valor para mesma variável.

Declarando uma *array*: a declaração e feita semelhante às variáveis comuns, onde define-se o tipo, nome que a identifica seguido por colchetes [ ] .Veja o exemplo:

```
int tons[ ];     // variável com 0 elementos
```

Para declarar uma *array* com mais de uma dimensão basta acrescentar mais um par de colchetes [ ]. NXC suporta até **quatro** dimensões. Veja o exemplo:

```
int X [ ] [ ];     // variável de duas dimensões
```

Uma *array* global de uma dimensão pode ser inicializada no momento da declaração. Veja o exemplo:

```
int X[ ] = {1, 2, 3, 4};  // array quatro posições e seus respectivos valores.
```

3.7 Expressões e Operadores

Valores são expressões em sua forma mais simples. Expressões mais complicadas são formadas a partir de valores usando vários operadores. A linguagem NXC só tem dois tipos de valores: constantes numéricas e variáveis.

Dois valores especiais são predefinidos: o verdadeiro e o falso. O valor de falso é zero (0), e o valor de verdadeiro é um (1). Os valores valem também para operadores relacionais (por exemplo, <): quando a relação é falsa o valor é 0, caso contrário o valor é 1.

Quando já declarada, pode ser atribuído às variáveis o valor de uma expressão. Em NXC existem os operadores têm funções que vão desde a simples atribuição de valores à modificação do valor de uma variável.

Podemos dividir os operadores em: Operadores Aritméticos,Operadores Relacionais, Operadores Lógicos e Operadores bit a bit.

Operadores Aritméticos

OPERADOR	AÇÃO	EXEMPLO
-	Subtração	X - Y
+	Adição	X + Y
*	Multiplicação	X * Y
/	Divisão	X / Y
%	Resto da divisão	X % Y
--	Decremento	X --
++	Incremento	X++

Operadores Relacionais: referem-se às relações que os valores podem ter uns com os outros.

OPERADOR	AÇÃO	EXEMPLO
>	Maior que	X > Y
>=	Maior que ou igual	X >= Y
<	Menor que	X< Y
< =	Menor que ou igual	X< = Y
==	Igualdade	X == Y
! =	Diferente	X!= Y

Operadores Lógicos: refere-se às maneiras como as relações podem ser executadas.

OPERADOR	AÇÃO	EXEMPLO
&&	AND	X&&Y
||	OR	X || Y
!	NOT	X **!** Y

Operadores bit a bit: refere-se a testar, atribuir ou deslocar os bits efetivos em um *byte* ou em uma palavra.

OPERADOR	AÇÃO	EXEMPLO
&	AND	X & Y
|	OR	X | Y
^	XOR (OR exclusivo)	X ^ Y
~	Complemento de um	~X
>>	Deslocamento à direita	X >> 5
<<	Deslocamento à esquerda	X << 5

3.7.1 Atribuição

A forma geral do operador de atribuição é:

nome_da_variável = *expressão*;

Em NXC existem nove operadores de atribuição diferentes:

OPERADOR	*AÇÃO*	Expressão exemplo
=	Atribui valor à variável	X = Y; // X recebe o valor Y
+ =	Atribui expressão de adição à variável	X + = 2; // X = X + 2
- +	Atribui expressão de subtração à variável	X - = 2; // X = X - 2
* =	Atribui expressão de multiplicação à variável	X * = 2; // X = X * 2
/=	Atribui expressão de divisão à variável	X / = 2; // X = X / 2
%=	Atribui à variável o resto da expressão	X%=2; // X = X%2
& =	Atribui à variável o resultado da expressão lógica bit a bit AND.	X &= Y; // X = X & Y
| =	Atribui à variável o resultado da expressão lógica bit a bit OR	X | = Y; // X = X | Y
^ =	Atribui à variável o resultado da expressão lógica bit a bit XOR.	X ^ = Y; // X = X ^ Y
|| =	Atribui à variável o resultado da expressão lógica OR	X || = Y; // X = X || Y
>>=	Atribui à variável o resultado da expressão de deslocamento à direita	
<<=	Atribui à variável o resultado da expressão de deslocamento à esquerda	

3.8 Estruturas de controle

Conforme seus programas se tornam mais complexos, eles precisarão algumas vezes executar um conjunto de instruções caso uma condição seja verdadeira e possivelmente, outro conjunto se esta for falsa. Outras vezes você poderá fazer uso de *loops,* que também obedecem a condições específicas e repetem o mesmo conjunto de instruções determinado número de vezes ou até que se satisfaça àquela condição.

Quando um programa realiza (ou não) e repete instruções baseado em condições específicas, ele estará realizando Processos denominados:

- *PROCESSO CONDICIONAL*
- *PROCESSO INTERATIVO*

Para realizar um **processamento condicional**, o programa avalia a condição determinada e retorna um resultado verdadeiro ou falso, que por sua vez determina o curso da ação.

No **processo interativo** cada repetição do(s) comando(s) é uma interação.

Para ajudar seus programas, NXC tem os comandos: *if, if-else switch, while, do-while, for, repeat, goto* e também define o *until*, uma alternativa conveniente para o *loop* while. Vejamos cada um destes, abaixo:

3.8.1 O comando if

A forma geral da sentença if é:

***if** (expressão) comando;*
***else** comando*;

Podendo ser também da seguinte forma:

***if** (expressão) comando*;

Onde *comando* pode ser uma única instrução ou um conjunto destas. O comando **else** é opcional.
Se a expressão for *verdadeira* a corpo do comando **if** será executado; caso contrário, o corpo do comando **else** será executado. Apenas um dos comandos associados às estruturas de controle será executado, nunca ambos. Veja o exemplo:

```
if ( a ==1 )
{
   PlayTone(600,400);
    Wait(500);            } else (OnFwd(OUT_AB, 75); Wait(3000));
```

// Quando seu programa precisar executar vários comandos ou repeti-los, você estará usando um comando composto, o qual deverá vir entre chaves.

3.8.2 O comando switch

Quando em seu programa for necessário comparar uma mesma variável com múltiplos valores, NXC lhe fornece o comando *switch*, que testa sucessivas vezes o valor de uma expressão com uma lista de constantes inteiras ou de caractere. Uma vez avaliada a expressão e gerado o resultado, NXC então compara este com cada um dos constantes definidos por você. Quando o valor coincide, os comandos correspondentes àquela constante serão executados. A forma geral do *switch* é:

```
 switch ( expressão )
{
 case constante1:
        conjunto de comandos
        break;
  case constante2:
        conjunto de comandos
        break;

  case constante3:
        conjunto de comandos
        break;
              .
              .
              .
 default:
         conjunto de comandos
 }
```

O comando *break* e *default:*

break: separa os comandos de um *case* para outro.Se você omiti-lo, NXC executará sucessivamente todas as instruções seguintes, independente do *case* ao qual pertence. *break* será utilizado sempre após o último comando correspondente a cada *case*.

Funciona como um comando de “desvio”, pois quando um *break* é encontrado a próxima linha a ser executada será aquela seguinte à estrutura *switch*;

default: utilizado quando você deseja executar determinada ação se nenhuma coincidência for detectada. *Default* é opcional, em sua omissão nenhuma ação será executada quando todos os testes falharem.

- **Switch** difere do comando **if** pois só pode testar igualdade. Já o **if** pode avaliar uma expressão lógica ou relacional.

3.8.3 O laço while

O laço *while* funciona como um *loop* condicional, no qual uma ou mais instruções irão se repetir até que determinada condição se torne falsa. A sua forma geral é:

while (*condição) comando;*

onde *comando* é um comando simples ou um bloco destes e, *condição* pode ser qualquer expressão.
No caso de comandos compostos, estes devem vir entre chaves. Como abaixo:

```
 while (condição)
{
      Comando1;
   Comando 2;
}
```

Ao encontrar um laço *while* no seu programa, NXC irá testar a condição especificada, se esta for verdadeira o corpo do *loop* será executado, então novo teste é realizado. O processo continua até que a condição se torne falsa ou um comando *break* seja encontrado.

3.8.3.1 O laço do- while

Esta instrução lhe permite executar um ou mais comandos pelo menos uma vez, isso porque diferentemente do laço *while*, *do-while* verifica a condição ao final do laço. A sua forma geral é:

```
  do
{
    comando;

 } while (condição);
```

Ao encontrar um comando *do*, NXC executará o que estiver entre as palavras *do* e *while*, então a condição especificada no *while* é testada para determinar se os comandos devem ou não ser repetidos. O laço *do-while* repete até que a *condição* se torne falsa.

DICA:

1 – Quando se trata de um comando simples, embora as chaves não sejam necessárias, elas devem ser usadas a fim de evitar confusão (para você) com o *while*.

3.8.4 O laço for

Para que seus programas possam repetir uma ou mais instruções um número específico de vezes, NXC também fornece o comando *for.* A sua forma geral é:

for (*inicialização; condição_final; incremento*) *comando*;

Você precisará de uma variável de controle para contar o número de vezes que as instruções serão executadas.

Para melhor compreensão iremos dividir este laço em quatro seções:

1 – Inicialização: atribui à variável de controle um valor inicial, geralmente 1 ou 0, este será o ponto de partida da contagem.
2 – Condição_final: é uma expressão relacional que determina quando o laço termina, ela testa o valor da variável de controle cada vez que o laço se repete para determinar se o programa executou as instruções o número desejado de vezes.
3- Incremento: define como a variável de controle varia cada vez que o laço é repetido. Geralmente o incremento é feito adicionando 1 à variável em cada repetição.
4- Comando: comando ou comandos que você deseja repetir.

Assim que a condição se torna falsa o programa executará a linha seguinte ao *for.*

OBS: As três primeiras seções devem vir separadas por ponto-e-vírgula.

3.8.5 O comando repeat

Algumas vezes em seu programa você precisará que determinado pedaço de código seja executado um número especifico de vezes sem que para tanto uma condição seja estipulada ou este trecho seja reescrito várias vezes.

O comando *repeat* lhe permite fazer isto. Sua forma geral é:

repeat (*expressão*) *comando;*

onde *expressão,* geralmente um número inteiro, determina quantas vezes o comando(simples ou composto) será executado.
Este *loop* só é realizado uma vez, ao encontrar o *repeat,* NXC repete a instrução X vezes e então passa para a próxima linha de código. Diferentemente dos comandos *while* e *do-while* que avaliam as condições cada vez que passam pelo *loop* para então seguir em frente.

3.8.6 O comando goto

Goto permite que a execução do seu programa desvie para um local específico denominado **rótulo**. Sua forma geral é:

```
goto rotulo;
      .
      .
      .

rotulo:
```

onde *rotulo* pode ser qualquer rotulo válido (uma letra, por exemplo) antes ou depois do *goto* e deve estar sempre na mesma tarefa do goto – voce não pode efetuar desvios entre tarefas.

> DICA: Restrinja o máximo que puder o uso do **goto,** pois este constitui uma tendência a tornar os programas ilegíveis (mas se usado prudentemente pode ser uma vantagem em certas situações), prefira usar construções como *if, if-else e while*.

3.8.7 O comando until

A definição de *until* é feita assim:

```
#define until (c)  while (! (c) )
```

Ou seja, until continuará realizando o *loop* até que a condição nele definida se torne verdadeira, deste modo notamos que este comando pode ser uma alternativa ao *while*. Veja o exemplo abaixo para compreender melhor:

```
until (Sensor_1 ==1);
```

este comando pode ter o corpo vazio ou trazer uma ou mais instruções, as quais deverão estar entre chaves.

3.9 Outros Comandos

NXC suporta ainda outros comandos, como os descritos abaixo:

3.9.1 Comando *start*

Você pode iniciar uma tarefa com o comando *start*. Sua forma geral é:

```
start task_name;
```

3.9.2 Comando *stop*

Você poderá interromper uma tarefa com o comando *stop.* A instrução só é suportada se você estiver executando em seu NXT o firmware NBC / NXC ou superior. Sua forma geral é:

```
stop task_name;
```

3.9.3 Comando continue (*continuar)*

Você pode usar a instrução *continue* em *loops. Quando o comando continue é encontrado, o loop* salta *para a próxima iteração dentro deste*, sem executar qualquer linha de código que segue o comando *continue.* Veja o exemplo:

```
task main()
{
        int i;
        for (i = 0; i < 5; i++)
        if (i == 1) continue;   // volta para  o início do loop
        else NumOut (0, LCD_LINE1, i);
}
```

3.9.4 Comando *return*

O comando *return* consiste de uma função que retorna o valor calculado para a função que chamou, no fim, ou antes, de chegar ao fim do seu código. Qualquer expressão pode aparecer no comando, que tem a seguinte forma geral:

return expressão;

A *expressão* é opcional, mas, quando presente, é o valor retornado pela função. O tipo da expressão deve ser compatível com o tipo de retorno da função. Quando o comando *return* não existe o valor de retorno é considerado indefinido. As funções que não retornam valores devem ser declaradas como do tipo *void*.

4.0 O Pré – processador

- O pré-processador é um programa que examina o código escrito e executa certas modificações no mesmo, baseado nas *Diretivas de Compilação.* Sendo ele a primeira fase na compilação do programa o qual é executado automaticamente sempre que seu código é compilado.
  As linhas que começam com um # são comandos para o pré-processador, chamados *diretivas,* onde tal linha é reservada inteira apenas para este comando onde não há a necessidade de usar o ponto-e-vírgula ao fim da linha.
- É o pré-processador que remove todos os comentários do código fonte antes do programa ser compilado. Por isso o compilador não “ vê ” realmente os comentários.

DICA:

1- As diretivas podem ser colocadas em qualquer parte do programa, mas é aconselhável colocá-las antes da tarefa principal. Isso tornará seu algoritmo mais organizado e compreensível.

O pré-processador NXC implementa as seguintes diretivas: # include,# define, # #, # undef e outros para compilação condicional. Sua implementação é bastante semelhante à de um pré-processador C padrão.

4.0.1 #include

A diretiva **#include** causa a inclusão de um programa-fonte em outro. Ela diz ao compilador para incluir, na hora da compilação, um arquivo especificado. Sua forma geral é:

```
#include "foo.h"   // nome do arquivo entre aspas
```

Programas NXC podem ou não começar com #include "NXCDefs.h". Este arquivo de cabeçalho padrão inclui muitas constantes importantes e macros que formam o núcleo NXC API. As versões atuais do NXC já não exigem que se inclua NXCDefs.h.

4.0.2 #define

A diretiva *#define* define constantes representadas por um identificador e uma string, toda vez que o compilador encontrar o *nome_da_macro* no programa a ser compilado, ele deve substituí-lo pela *string* fornecida. A esse processo dar-se o nome *substituição macro simples*. Isso é muito útil para deixar seu programa mais geral, facilitando a manutenção e compreensão do código. Sua forma geral é:

```
#define nome_macro string   // nome_macro refere-se ao identificador
```

Não há ponto-e-vírgula nesse comando. Pode haver quantos espaços for necessário entre o *nome_macro* e a *string*. O fim da linha, normalmente determina o fim da declaração de uma macro, mas a nova linha pode ser criada com a barra invertida ('\') o que permitirá a criação de uma macro multi-line.

4.0.3 # undef

A diretiva *#undef* remove a definição de macro feita anteriormente. Sua forma geral é:

```
#undef nome_da_macro;
```

Feito isto a macro não estará mais definida, o compilador não mais a reconhecerá.

4.0.4 ## (Concatenação)

A diretiva # # concatena duas palavras ou caracteres. Veja o exemplo:

```
#define IMPRIME (n) \
NumOut (0, LCD_LINE##n, b##n)

bool b1 = falso;
bool b2 = verdadeiro;

task main()
{
        IMPRIME (1);                    //  NumOut (0, LCD_LINE1, b1)
        IMPRIME (2);                    //  NumOut (0, LCD_LINE2, b2)

        Wait (SEC_2);
}
```

4.0.5 #import

A diretiva *#import* permite definir uma variável *array de bytes* global em seu programa NXC, a qual contém o conteúdo de um **arquivo** importado.

Assim como *#include*, esta diretiva é seguida por um nome de arquivo entre aspas. Após o qual **opcionalmente** pode-se incluir uma string para definição do nome da variável que contém o conteúdo do arquivo importado. Sua forma geral é:

```
#import "nome_arquivo.ext” string
```

Onde “*nome_arquivo”* é o nome do arquivo que será importado, “.*ext”* sua extensão, e *string* o nome da *array de bytes*.

Há três situações a serem analisadas quanto à sintaxe da diretiva:

1- O nome do arquivo sem qualquer extensão seria o próprio nome da variável *array de bytes.* Veja o exemplo:

```
#import "nome_arquivo”        // string = nome_arquivo
```

2- Se a *string* não for especificada, o nome da variável *array* de *bytes também* será "nome_arquivo"*.* Veja o exemplo:

```
#import "nome_arquivo.ext”        // string = nome_arquivo
```

3- Conseguintemente se nem a *extensão* ou a *string* forem especificadas o nome da variável *array de bytes* será o próprio nome do arquivo.

4.0.6 #download

A diretiva #*download* trabalha em conjunto com a capacidade de download embutida no compilador. Esta instrução permite dizer ao compilador para fazer o download de um arquivo auxiliar especificado que poderá ter outra extensão que não ***.nxc*** .

Se a extensão do arquivo corresponder àquelas que o compilador é capaz de compilar (como. rs . nbc ou .ric) , então ele irá primeiro compilar o código fonte antes de baixar o binário resultante. O nome do arquivo para download (e, opcionalmente, para compilação) deve vir entre aspas imediatamente após a diretiva. Se o compilador só é dito para compilar o código fonte original então a diretiva #*download* é ignorada.Veja o exemplo:

```
#download "meu_arquivo.rs"

#download "minha_imagem.ric"
```

4.0.7 Compilação Condicional

A Compilação Condicional NXC funciona semelhante à compilação condicional do pré-processador C. É possível que em algum momento em seus programas, você queira que o pré-processador teste se um símbolo foi definido anteriormente no código fonte e, em caso afirmativo ou negativo ele processará determinado pedaço de código. Para ajudar seus programas, o pré-processador NXC suporta os seguintes comandos: *#ifdef, #ifndef, #else, #endif.*

a) #ifdef: quando essa diretiva é encontrada, o pré-processador testa se o símbolo especificado foi definido anteriormente pelo programa. Em caso afirmativo, o conjunto de código que segue a diretiva é processado até que *#endif* seja encontrado. Veja o exemplo:

```
#ifdef símbolo
    //  comandos
#endif
```

b) #ifndef: Se você quiser que o programa processe determinados comandos caso o programa não tenha definido um símbolo, use essa diretiva. Veja o exemplo:

```
#ifndef símbolo
        #define símbolo
#endif
```

c) #else : O pré-processador poderá executar um conjunto de código quando a condição testada em #ifdef for verdadeira, e outro quando esta for falsa. Para isso usamos essa diretiva. Veja o exemplo:

```
#ifndef símbolo
        // comandos
#else
        // outros comandos
#endif
```

- NXC API -

- API, do inglês *Application Programming Interface* ou, em português, Interface de Programação de Aplicativos de uma forma geral, é o conjunto de padrões de programação que permite a construção de aplicativos e a sua utilização de maneira não tão evidente para os usuários.

  Como assim? - Um exemplo análogo é o automóvel: um motorista não tem de conhecer o funcionamento mecânico do motor de um veículo para poder conduzi-lo. Mas lhe é acessível, apenas uma interface composta por um volante, pedais (acelerador, embreagem, freio), e botões (quatro piscas, faróis, buzina, etc.): trata-se, de certa maneira, da interface proposta a quem utiliza.

  Afirmamos, portanto que API é a “matrix” dos aplicativos, ou seja, uma interface que roda por trás de tudo. Graças ao API, um programador não precisa se preocupar como uma aplicação distante funciona, ou como as funções foram aplicadas num programa para poder utilizá-las.

A API NXC define um conjunto de constantes, funções, valores e macros que fornecem acesso a vários recursos do NXT, tais como sensores, saídas e comunicação, consistindo de **funções**, **valores** e **constantes**.

4.1 Funções de tempo

I- wait (tempo)

Agora é hora de esperar por um tempo. Esta função fará o robô esperar por um tempo determinado, o qual será o argumento da função e deve ocupar o espaço entre parênteses. O *tempo* é um número dado em 1 / 1000 de segundo: para que você possa de forma muito precisa dizer ao programa quanto será o tempo de espera. Neste período determinado, o programa continua executando a instrução anterior. Sua forma geral é:

```
Wait(2000);   // espere dois segundos
```

Veja o exemplo da pg. 12

II- SetSleepTimeout (minutos)

Retorna os minutos pelo programador determinados em que o NXT permanecerá ligado antes de ser automaticamente desligado. Sua forma geral é:

```
SetSleepTimeout(minutos);
```

4.2 Funções de controle de programa

III- Stop (valor)

O programa em execução será interrompido se *valor* for verdade. De modo que qualquer código após o comando será ignorado. Sua forma geral é:

```
Stop(x == 10);     // interromper o programa se  x==10
```

IV- Acquire(Mutex)

A função fará a "aquisição" de uma variável *mutex* especificada. Se outra tarefa já adquiriu a variável, então a tarefa atual será suspensa até que a *mutex* seja liberada.
Esta função é usada para garantir que a tarefa atual tenha acesso exclusivo aos recursos compartilhados, como o display e o motor. Após a tarefa atual terminar de usar os recursos compartilhados, o programa deve chamar a **função *Release(Mutex)*** para permitir que outras tarefas possam adquirir a *mutex*. Veja o exemplo:

```
Acquire(moveMutex); // tenha certeza do acesso exclusivo
//  comandos
Release(moveMutex);
```

V- ExitTo(task)

Ao encontrar esta função a tarefa atual será imediatamente interrompida e uma tarefa especificada deverá começar ser executada.Sua forma geral é:

```
ExitTo(proxima_Task);
```

VI- Follows(task1, task2, ..., taskN)

Esta função deve ser chamada dentro de uma tarefa - de preferência no início da definição destas, é usada para fazer o controle de execução de tarefas, onde elas deverão ser executadas "a seguir" aquelas determinadas nos parâmetros. Se múltiplas tarefas forem declaradas "a seguir" uma mesma *task*, elas serão executadas simultaneamente. Veja o exemplo:

```
task main()
{
      // nenhum código
}

task music()
{
       Follows(main);

       while (true) {
        PlayTone(600,400);  Wait(500);
        }
}

 task movement()
{
       Follows(main);

       while(true) {
       OnFwd(OUT_AB, 75); Wait(3000);
       }
}
```

Se você preferir declarar suas tarefas antes da tarefa principal, para fazer a chamada você deverá usar a função **start task;** dentro da *main*, constituindo uma opção mais compreensível para seu código-fonte. Veja o exemplo:

```
task music()
{
    while (true){

    PlayTone(262,400);  Wait(500);
    PlayTone(294,400);  Wait(500);
    PlayTone(330,400);  Wait(500);
    PlayTone(294,400);  Wait(500);
    }
}
task main()
{
   start music;          // inicia a tarefa music
   while(true){

   OnFwd(OUT_AB, 75); Wait(3000);
   OnRev(OUT_AB, 75); Wait(3000);
```

VII- ExitTo(task)

Ao encontrar esta função a tarefa atual será imediatamente interrompida e uma tarefa especificada deverá começar ser executada.Sua forma geral é:

```
ExitTo(proxima_Task);
```

4.3 Funções numéricas

VIII- Random(n)

Retorna um número aleatório de 16 bits *unsigned* entre 0 e *n*, o qual pode ser uma constante ou variável.Veja o exemplo:

```
x = Random(100);    // retorna um valor ente  0 e 100.
```

IX- Random()

Retorna um número qualquer aleatório de 16 bits *signed*. Veja o exemplo:

```
x = Random ();
```

X- Sqrt(x)

Retorna a raiz quadrada de um valor especificado. Sua forma geral é:

```
x = Sqrt(n);
```

XI- Sin(degrees)

Retorna o seno de um valor especificado em graus. O resultado é 100 vezes o valor do seno de um ângulo no range (-100...100). Sua forma geral é:

```
x = Sin(θ);
```

XII- Cos(degrees)

Retorna o cosseno de um valor especificado em graus. O resultado é 100 vezes o valor do cosseno de um ângulo no range (-100...100). Sua forma geral é:

```
x = Cos(θ);
```

XIII- Asin(value)

Retorna inverso do seno de um valor especificado no range (-100 .. 100). O resultado é dado em graus (-90 .. 90). Veja o exemplo:

```
G = Asin(60);
```

XIV- Acos(value)

Retornar o inverso do cosseno de um valor especificado (-100 .. 100). O resultado é dado em graus (0 .. 180). Veja o exemplo:

```
G = Acos(0);
```

4.4 Funções para String

XV- StrLen(str)

O *strlen*, cujo nome é uma abreviatura de *string length*, recebe uma string e retorna o seu comprimento. O **comprimento** *(= length)* de uma string é o seu número de caracteres, sem contar o caractere nulo final(\0). Sua forma geral é:

```
x = StrLen(string);      // retorna o tamanho da string
```

XVI- StrIndex(str, indice)

Retorna o valor numérico correspondente da tabela ASCII de um caractere no índice especificado da referida *string.* Veja o exemplo:

```
x = StrIndex(string, 2);   // retorna o valor da  string[2]
```

XVII- StrCat(str1, str2, ..., strN)

Retorna uma string que é o resultado da concatenação de duas ou mais *strings*, adicionando o conteúdo da segunda ao final da primeira. Veja o exemplo:

```
Mensagem = StrCat("concat", "string");   // retorna "concatstring"
```

XVIII- SubStr(string, indice, nº caracteres)

Retorna uma string derivada da seqüência de entrada. A partir dos índices especificados os caracteres correspondentes serão incluídos a nova string. Os parâmetros são respectivamente: (string, índice, nº de caracteres derivados). Veja o exemplo:

```
Mensagem  = SubStr ("hoje", 2, 2);     // retorna "je"
```

XIX- StrReplace(string, indice, novaStr)

Retorna uma string com parte da seqüência substituída (no índice especificado) O novo conteúdo da string é fornecido no terceiro argumento. Veja o exemplo:

```
Mensagem = StrReplace("mover", 2 , "xx");      // retorna "moxxer"
```

4.5 Fazendo uso dos recursos de entrada (INPUT) – Sensores

Os quatro sensores serão conectados ao NXT através de portas seriais permitindo-o reagir a eventos externos. As portas são nomeadas externamente como: 1, 2, 3, 4, embora os sensores sejam internamente numerados como 0, 1, 2, 3. Durante a escrita do seu código estas portas podem denominadas como IN_1, IN_2, IN_3, IN_4 ou S1, S2, S3, S4, usadas para referenciar a porta na qual um determinado sensor está conectado.

Habilita_Sersor_Tipo	Porta
SetSensorTouch	(IN_1)
SetSensorSound	(IN_2)
SetSensorLight	(IN_3)
SetSensorLowspeed	(IN_4)

É necessário dizer ao programa que tipo de sensor está conectado a cada porta, para isso usamos a função de chamada **SetSensorType(porta, const type)** que irá habilitar os sensores.

Ex:
SetSensorTouch (IN_1);

ou

SetSensor(IN_1,SENSOR_TOUCH)

Ao longo do código você irá referenciar-se aos sensores da seguinte maneira:

SENSOR_1		**Sensor (IN_1)**
SENSOR_2	OU	**Sensor (IN_2)**
SENSOR_3		**Sensor (IN_3)**
SENSOR_4		**Sensor (IN_4)**

OBS: Há uma especificidade com o sensor ultra-sônico (Lowspeed), para referenciar-se a ele você deve fazer da seguinte maneira:

Ex: SensorUS(IN_4)

Se a porta for referenciada no código com o tipo errado (sensor diferente do conectado na referida porta), o NXT pode não ser capaz de realizar a leitura com precisão.

XX- ResetSensor(porta)

Reinicia o valor de um sensor. Em um *loop, por exemplo*, o valor do sensor deve receber o *reset* a cada leitura para garantir a validade desta. A porta na qual o sensor está conectado deve ser especificada (IN_1, IN_2, IN_3, IN_4).Sua forma geral é:

```
ResetSensor(x);        // x = PORTA
```

4.5.1 Informações do sensor

XXI- SensorType(porta)

Retorna o tipo do sensor configurado na porta especificada. Veja o exemplo:

```
x = SensorType (IN_1);
```

XXII- SensorRaw(porta)

Retorna o valor bruto lido de um sensor na porta especificada. Veja o exemplo:

```
x = SensorRaw (IN_1);
```

O NXT permite que um sensor seja configurado em diferentes modos. O modo mais importante é o **SENSOR_MODE_RAW**. Neste modo, o valor obtido é o valor bruto produzido pelo sensor, um número entre 0 e 1023, os quais são retornados pela função **SensorRaw(porta),** quando chamada.

Exemplificando: Veja o caso do sensor de toque, enquanto este não for acionado (pressionado) o valor está próximo de 1023. Quando é totalmente pressionado está perto de 50, se parcialmente o valor varia entre 50 e 1000. Então, se você habilitar um sensor de toque no modo **Raw**, poderá descobrir, por exemplo, se ele está parcialmente pressionado. Outro exemplo é o do Sensor de luz. Quando o valor varia em cerca de 300 corresponderá a um ambiente muito claro, se variar em 800 (muito escuro).

4.6 Fazendo uso dos recursos de saída (OUTPUT) – Motores

O módulo de saída NXT engloba todas as saídas dos motores. Os três servo-motores serão conectados ao NXT através de portas serias, nomeadas externamente como: A, B e C. A API NXC fornece combinações predefinidas de saídas: OUT_AB, OUT_AC, OUT_BC, e OUT_ABC as quais constituem o **primeiro argumento** das funções que se relacionam com os motores, identificando a porta na qual o(os) motor(es) a ser utilizado está conectado.
No **segundo argumento** são definidos níveis de potência (velocidade) que podem variar de 0% (mais baixo) a 100% (mais alto).

Constante de saída	Velocidade
OUT_A	,0...100
OUT_B	,0...100
OUT_C	,0...100
OUT_AB	,0...100
OUT_AC	,0...100
OUT_BC	,0...100
OUT_ABC	,0...100

Os servo-motores NXT tem um *encoder* embutido que permite controlar com precisão a posição do eixo e a velocidade. Para tanto, NXC define constantes **RegMode** (constantes de regulação), são elas:

Constantes RegMode	Se Modo selecionado
OUT_REGMODE_IDLE	Nenhuma regulação será aplicada
OUT_REGMODE_SPEED	NXT regula um motor para obter uma velocidade constante.
OUT_REGMODE_SYNC	Sincroniza a rotação de dois motores

XXIII- Off (outputs)

Desliga as saídas especificadas, nesse momento os motores param imediatamente mantendo a posição até que sejam habilitados novamente por uma função. As constantes de saída devem ser especificadas. Veja os exemplos:

```
Off(OUT_A);      //  desliga a saída  A .
Off(OUT_B);      //  desliga a saída  B .
```

ou

```
Off(OUT_AB);      //  desliga as saídas  A e B em uma única
```

linha de código.

XXIV- Coast (outputs)

Desliga as saídas especificadas. Constitui uma maneira mais " gentil" de parar os motores.Esta função é análoga a **Float (outputs)**, nas quais simplesmente corta-se a energia que está fluindo para o motor. Veja os exemplos:

```
Coast(OUT_A);

Float(OUT_A);
```

Testando a diferença entre as funções Off ( ), Coast ( ) e Float ( ):

```
task main()
{

        OnFwd(OUT_AC, 75);
        Wait(500);
        Off(OUT_AC);
        Wait(1000);
        OnFwd(OUT_AC, 75);
        Wait(500);
        Float(OUT_AC);
        Wait(1000);
        OnFwd(OUT_AC, 75);
        Coast(OUT_AC);
}
```

// primeiro o robô pára bruscamente depois mais gentilmente.

: Notaremos que a diferença quanto à forma de parada é mínima, mas tal diferença faria grande diferença em outros tipos de robôs.

XXV- OnFwd(outputs, pwr)

Ativa as saídas (motores) habilitando-as a mover-se para frente. As constantes de saída devem ser especificadas como primeiro argumento seguido da velocidade desejada, a qual deve estar dentro do range especificado. Veja o exemplo:

```
OnFwd(OUT_A, 35);  //motor conectado a saída A se moverá para frente com 35% da velocidade máxima.
```

XXVI- OnRev(outputs, pwr)

Ativa as saídas (motores) habilitando-as a mover-se para trás. As constantes de saída devem ser especificadas como primeiro argumento seguido da velocidade desejada, a qual deve estar dentro do range especificado. Veja o exemplo:

```
OnRev(OUT_AC, 50);  //motores conectados a saída A e C se moverão para trás com 50% da velocidade máxima.
```

XXVII- OnFwdReg(outputs, pwr, regmode)

Move as saídas especificadas para frente usando um **modo de regulação** determinado pelo programador.Os argumentos são respectivamente: (porta,velocidade,regmode). Veja o exemplo:

```
OnFwdReg(OUT_A, 50, OUT_REGMODE_SPEED);  // regula a velocidade
```

XXVIII- OnRevReg(outputs, pwr, regmode)

Move as saídas especificadas para trás usando um **modo de regulação** determinado pelo programador. Veja o exemplo:

```
OnRevReg(OUT_B, 50,OUT_REGMODE_SPEED); // regula a velocidade
```

Testando a diferença entre a função OnFwdRev ( ) nos diferentes modos de regulação:

Veja como se comporta o robô. Para realizar o teste segure o robô nas mãos, para melhor observação dos resultados:

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

Anote os resultados quando da parada dos servos em cada Regmode: _____

_____

_____

_____

_____

O robô NXT pode mover-se perfeitamente reto, para tanto você deve usar um recurso de sincronização fazendo com que o par de motores selecionados movam-se juntos e esperem um pelo outro no caso um deles estar retardado ou mesmo bloqueado.

XXIX- OnFwdSync(outputs, pwr, turnpct)

Move as saídas especificadas para frente com sincronização regulada. É o mesmo que OnFwdReg () no modo SYNC, mas agora você também pode especificar o *turnpct*, percentual de direção (de -100 a 100), para virar à esquerda, direita ou girar no lugar, mas sempre mantendo sincronia. Veja o exemplo:

```
OnFwdSync(OUT_AB, 35, -100);   // rotaciona para direita
```

XXX- OnRevSync(outputs, pwr, turnpct)

Mover as saídas especificadas para trás com sincronização regulada usando o *turnpct,* um valor percentual no range -100/100 que representa a porcentagem de rotação para esquerda ou direita . Veja o exemplo:

```
OnRevSync(OUT_AB, 35, -100);   // rotaciona para esquerda
```

XXXI- RotateMotor(outputs, pwr, angulo) Function

O motor irá rotacionar de acordo com um ângulo especificado em graus no terceiro argumento da função. Os argumentos *velocidade* e *ângulo* poderão mudar de sinal (positivo ou negativo): assim, se a velocidade e o ângulo tiverem o mesmo sinal, o motor irá mover-se para frente, se os sinais forem opostos, o motor irá mover-se para trás. Veja o exemplo:

```
RotateMotor(OUT_A, 75, 45);        //  em frente 45º

RotateMotor(OUT_AC, -75, 30);      // para trás  30º
```

XXXII- RotateMotorEx(outputs, pwr, angulo, turnpct, sync, stop)

Uma extensão da função anterior, esta permite que você sincronize dois motores especificando um *'turnpct*' (de -100 a 100), e o parâmetro booleano *sync* o qual pode ser definido como "**true**" ou "**false**" (*deverá ser true* se *turnpct* for diferente de zero). Permite também que você especifique se os motores devem frear após completar o ângulo de rotação usando o parâmetro booleano *stop*. Veja o exemplo:

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

XXXIII- MotorTachoCount(output)

Obter o valor de contagem do tacômetro, dispositivo que mede as rotações por minuto de um motor ou de um eixo, da saída especificada. Veja o exemplo:

```
x = MotorTachoCount(OUT_A);
```

XXXIV- ResetTachoCount(outputs)

Reinicia a contagem do tacômetro e valor limite de rotações para as saídas especificadas. Veja o exemplo:

```
ResetTachoCount(OUT_AB);
```

XXXV- MotorTachoLimit(output)

XXXVI- MotorPower(output)

Obtém o nível de potência da saída especificada. Veja o exemplo:

```
x = MotorPower(OUT_A);
```

XXXVII- MotorActualSpeed(output)

Obtém o valor real da velocidade da saída especificada.Veja o exemplo:

```
x = MotorActualSpeed(OUT_C);
```

4.7 Fazendo uso dos recursos de Som

O NXT fornece suporte para reprodução de tons básicos e até mesmo arquivos de som e melodia. Quando um som é reproduzido pelo NXT, a execução do programa não espera a reprodução anterior ser concluída. Para reproduzir tons múltiplos ou em seqüência é necessário aguardar o tom anterior finalizar, para tanto voce poderá utilizar a função **wait()**.
Os arquivos de som vem especificados pela extensão **.rso** (semelhantes aos arquivos .wav) e os arquivos melodia (Arquivos MIDI) pela extensão **.rmd**, esses últimos contêm múltiplos tons, cada um é definido por uma **frequência** em Hz e uma **duração** especificada em milissegundos. Os arquivos de melodia são geralmente muito menores do que arquivos de som.

Outros parametros a serem especificados são:

- **Volume:** um valor entre 0 (mudo) e 4(máximo).
- **Loop:** é um valor booleano que indica se o som deve ser reproduzido repetidamente

XXXVIII- PlayTone(frequencia, duração)

Reproduz um tom de freqüência e duração especificada. Veja o exemplo:

```
task music()
{
    while (true){

    PlayTone(600,400);  Wait(500);
    }
}
```

XXXIX- PlayToneEx(frequency, duration, volume, Loop?)

Reproduz um tom de freqüência, duração e volume especificados. Veja o exemplo:

```
PlayToneEx(5000, 500, 2, false);
```

XL- PlayFile(nome_do_arquivo)

Reproduzir um arquivo de som (.rso) ou um arquivo de melodia (.rmd) especificados. O parâmetro *nome_do_arquivo* pode ser qualquer string ou identificador válidos. Sua forma geral é:

```
PlayFile("nome_do_arquivo.extensão");
```

XLI- StopSound()

Interrompe a reprodução do arquivo ou tom atual. Sua forma geral é:

```
StopSound();
```

4.8 Fazendo uso dos recursos de Display

Esse é um display LCD do tipo Matriz com resolução **100 x 64 pixel** - gráfico branco & preto, no qual é possível desenhar linhas, pontos, retângulos e círculos, e até imagens bitmap (. ric), bem como também pode “imprimir” letras e números.

O LCD é uma matriz iniciada no canto inferior esquerdo da tela no eixo Y positivo (coordenada 0,0), expandindo-se para cima e para direita no eixo X positivo. As linhas dessa matriz são enumeradas entre 01 e 08, onde a linha 01 fica no topo da tela e a 08 no fundo, as quais são nomeadas da seguinte maneira: LCD_LINE1,LCD_LINE2, LCD_LINE3, LCD_LINE4, LCD_LINE5, LCD_LINE6, LCD_LINE 7,LCD_LINE8.

XLII- NumOut(x, y, valor, clear = falso)

Imprime a partir dos pontos especificados em x(coluna) e y (linha) da tela, um valor numérico. O parâmetro *valor* é uma variável que corresponde a um determinado número, podendo ser também o próprio valor. O parâmetro *clear* é um valor booleano opcional, ele limpa a tela, se este argumento não for especificado será falso. Veja o exemplo:

```
NumOut(0, LCD_LINE3, valor); // mostra valor na coluna 0, linha 01 do LCD
```

XLIII- TextOut(x, y, txt, clear = falso)

Imprime na tela um determinado texto a partir das coordenadas especificadas pelo programador. Os parâmetros são os seguintes: **x**=coluna; **y** = linhas, seguindo a constante de nomenclatura especificada na pagina anterior; **txt** = texto a ser mostrado no LCD. O último argumento é opcional, determinando se a tela deve ter seu conteúdo apagado ou não. Se este não for especificado fica entendido que tem valor booleano zero (falso). Veja o exemplo:

```
TextOut(60, LCD_LINE4, "Meu primeiro Olá!");
```

XLIV- GraphicOut(x, y, nome_arquivo, clear = falso)

Desenhar na tela o ícone gráfico do arquivo especificado a partir do ponto especificado por **x** e **y**, correspondentes a posição dos pixels de resolução. O argumento *clear* é análogo às “funções display” anteriores. O arquivo deve vir especificado pela extensão **.ric**, se este não puder ser encontrado, então nada será desenhado, nenhum erro será reportado. Veja o exemplo:

```
GraphicOut(30,10 , "imagem.ric");
```

XLV- CircleOut(x, y, raio, clear = falso)

Permite desenhar um círculo no LCD, tendo seu centro no ponto especificado por **x** e **y**, e raio determinado pelo programador no terceiro parâmetro da função. O argumento clear é análogo às funções anteriores. Veja o exemplo:

```
CircleOut(50, 32, 20);
```

XLVI- LineOut(x1, y1, x2, y2, clear = falso)

Permite desenhar uma linha no LCD que vai do ponto especificado por **x1**, **y1** até aquele especificado por **x2**, **y2**. O argumento *clear* é análogo às funções anteriores. Veja o exemplo:

```
LineOut(10, 10, 5, 5);
```

XLVII- PointOut(x, y, clear = falso)

Permite desenhar um ponto no LCD na coordenada determinada por **x** e **y**. O argumento *clear* é análogo às funções anteriores. Veja o exemplo:

```
PointOut(4, 4);
```

XLVIII- RectOut(x, y, largura, altura, clear = falso)

Permite desenhar um retângulo no LCD a partir do seu vértice inferior esquerdo especificado por **x** e **y**, com largura e altura determinadas pelo programador nos quarto e quinto parâmetros da função. O argumento *clear* é análogo às funções anteriores. Veja o exemplo:

```
RectOut(10, 10, 20, 5);
```

XLIX- ResetScreen()

Restaura a tela padrão do NXT no programa em execução. Sua forma geral é:

```
ResetScreen();
```

L- ClearScreen()

Limpar a tela do NXT para uma tela em branco. Sua forma geral é:

```
ClearScreen();
```

HORA DE POR EM PRÁTICA!

Com tantas informações, já podemos começar a por a “mão na massa”. Monte sua equipe e teste seus conhecimentos resolvendo os problemas propostos abaixo.Trabalhe em dupla se preferir

Vá até a **página ....** para informações sobre o compilador *BRICXCC,* o qual irá utilizar para escrever seus programas.

a. Desenvolver um programa para que o robô percorra um caminho qualquer até que um sensor seja acionado.

b. Desenvolver um programa para que o robô percorra um caminho qualquer até que um sensor seja acionado. Após esta ação o robô deverá fazer uma manobra de esquiva e continuar em frente por mais três segundos.

c. Desenvolver um programa para que o robô percorra um caminho qualquer quatro vezes.

d. Desenvolver um programa para que o robô tome decisões diante de valores lidos por três sensores diferentes.

e. Desenvolver um programa no qual conste duas tarefas distintas, sendo uma delas responsável por reproduzir determinado tom.

f. Desenvolver um programa para que o robô percorra um caminho qualquer com velocidade constante, determine o ângulo de rotação, faça uso de pelo menos um sensor.

g. Desenvolver um programa que utilize uma sub-rotina e tome decisões diante de valores lidos por pelo menos um sensor.

h. Desenvolver um programa para que o robô percorra um caminho na forma de um quadrado, repita isto 4 vezes, enquanto o NXT mostra em seu display a figura de um retângulo.

i. Desenvolver um programa para que o robô percorra um caminho para frente enquanto reproduz dois tons diferentes, um depois do outro, depois disto ele deve percorrer um caminho para trás e reproduzir outros dois tons diferentes.

j. Desenvolver um programa para que o robô detecte uma linha preta e siga por ela com os servos motores sincronizados por 5 segundos, quando o sensor de som detectar um ruído se 60Dcbs seguir para trás por três segundos, parar e recomeçar todo o processo novamente, utilize pelo menos uma outra função.

k. Desenvolver um programa para que o robô utilize os quatro sensores, tomando decisões diferentes diante de cada um.

l. Desenvolver um programa para que o robô percorra um caminho para frente, quanto um objeto for detectado em sua trajetória ele deve fazer uma manobra para esquerda, seguir em frente por mais três segundos até que a ”cor” preta seja detectada pelo sensor de luz e então parar. Depois disto mostre na tela do NXT a frase “Fim do percurso”, então desligue o NXT imediatamente.

m. Desenvolver um programa que utilize a palavra chave *enum* e a função *NumOut()*.

n. Desenvolver um programa que reproduza cinco tons diferente em dois volumes diferentes, utilize outras duas funções.