Programming LEGO NXT Robots using NXC

(beta 30 or higher)

(Version 2.2, June 7, 2007)

by Daniele Benedettelli

with revisions by John Hansen

C言語ライクな

NXCを使った LEGO NXT ロボット プログラミング

（ベータ30 以上）

（バージョン2.2 ： 2007年6月7日）

<訳：2011年4月6日バージョン>

作者 Daniel Benedettelli
(John Hansen による バージョン)
翻訳 高本孝頼（補足説明付き）

はじめに

Mindstorms NXT をフルパワーで活用するには、以前のLEGO社のMindostorms RIS やCyberMaster、Spybotics と同様に、NationalInstruments 社のLab-view の様なNXT-G より強力な言語を使うほうが良いでしょう。
NXC はJohn Hansen 氏が開発したプログラム言語で、LEGOロボット専用となります。

これまでプログラムを作成してことがない人でも心配はいりません。 NXCは簡単に利用でき、このチュートリアルは最初の導入ステップを学ぶことができます。これまでプログラミングの経験がない場合でも、このチュートリアルは、NXC が簡単であることから、その初歩を学ぶ上で手助けとなります。

NXCのプログラミング作成をやさしく学ぶために、フリー（無料）の開発環境「Bricx Command Center (BricxCC)」が用意されています。この開発環境上でプログラムを書いたり、そのプログラムをロボットにダウンロードしたり、実行させたり、中止したり、またNXT のフラッシュメモリを閲覧したり、さらにNXT 上で音声ファイルの変換を行ったりすることができます。さらに「BricxCC」 はワープロと同じような使い方もできる多機能のソフトウェアです。このチュートリアルでは、この「BricxCC」（バーション3.3.7.16 以上）を、NXCプログラミングでの統括開発環境（IDE: Integrated Development Environment）として使います。是非とも、つぎのWebサイトからダウンロードして環境を整えてからスタートさせてください。

http://bricxcc.sourceforge.net/

「BricxCC」は、Windows（95、98、ME、NT、2K、XP、およびVista）上で利用できます。NXC言語自体は、その他のプラットフォーム（開発環境）でも使用できます。このNXC言語は、つぎのWebページからダウンロードできます。

http://bricxcc.sourceforge.net/nxc/

ここでのチュートリアルの内容の多くは、「BricxCC」以外やその他のツールでも利用できます。
また、このチュートリアルは、NXC ベータ30版以上で使えるようにアップデートしています。ここでのサンプルのプログラムのいくつかは、ベータ３０版より古いバージョンのコンパイラを使われている場合、動かないこともあります。

その他、以下のWebサイトでは、NXTと通信を行うPCツールを含む Mindstorms RCX と NXT 関連の情報を掲載しています。

http://daniele.benedettelli.com

謝辞

貴重な仕事を行っているJohn Hansen 氏に感謝します。

翻訳者：高本孝頼　takamoto0206@yahoo.co.jp

も く じ

I. はじめてのプログラム作成

この章では、とても簡単なプログラム作成について紹介します。このプログラムは、ロボットを4 秒間前進させ、その後4 秒間後退し停止するものです。このプログラムは特殊なものではなく、基本的なものとして紹介しています。また、プログラミングそのものが簡単であることを示しています。プログラム作成する前に、ロボットを組み立て準備してください。

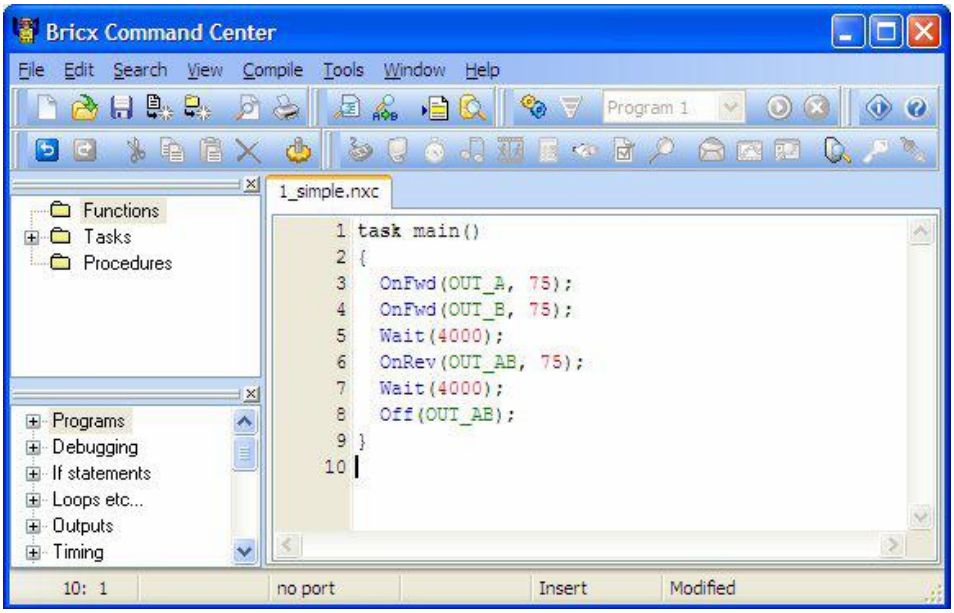
ロボットの組み立て
このチュートリアルでは、NXT キットで紹介している簡単な組み立てでできる「Tribot」ロボットを使っています。ただ異なる部分は、右モータを出力ポートAに、左モータを出力ポートCに接続し、補足追加したモータを出力ポートBに接続します。

あらかじめMinsdstormsNXTには、ファームウェアのFantomドライバーを正しくインストールしておく必要があります。

「Bricx Command Center」の起動
ここでは、「Bricx CommandCenter」（BricxCC）を使って、NXCプログラムを作成します。「BricxCC」アイコンをダブル・クリックして、「BricxCommandCenter」を起動します。あらかじめ、この「BricxCC」ソフトウェアをインストールしておく必要があります。まだインストールされていない場合には、序文に記載しているWebサイトにアクセスして、任意のディレクトリにインストールしてください。プログラムを起動した時点で、ロボットを探すための「Find Brick」ウィンドウが表示されます。
このウィンドウ上での「Port」（接続ポート）と「BrickType」（ブロック・タイプ）を設定し、OKボタンをクリックします。

＜補足説明:「Port」(接続ポート)は、USB接続の場合「USB0::0X0002::0016530C8BD5::RAW」(表示例)を選択するか、Bluetooth接続の場合「BTH::MYNXT::016:53:0C:8B:DB:34」(表示例)を選択します。また「BrickType」では、「NXT」を選択します＞

このプログラムは自動的に、NXTロボットを探しだして接続し、つぎの「BricxCommandCenter」ウィンドウを表示します。



このプログラムのインタフェースは、一般的なテキスト・エディタと同じように、通常のメニューを用意し、「ファイル（File）」の「開く（Open）」や「保存（Save）」、「プリント出力（Print）」や、ファイルの「編集（Edit）」などを持っています。
その他として特殊なプログラムの「コンパイル（Compile）」やプログラム・ファイルの「ダウンロード（Download）」、ロボットの各種情報取得の機能を持っていますが、まずは無視してつぎに進んでください。

それでは、新しいプログラムを作ってみましょう。メニューの中の「File」をヒットし、「New File」ボタンをクリックしてください。テキスト・エディタとして「Untitled1」のウィンドウがオープンします。

プログラムの作成

この「Untitled1」ウィンドウ上で、つぎのプログラムをタイプしてみてください。

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

最初は、何のことか分からないでしょうが、分かりやすく説明していきましょう。

まず、NXCプログラムは、「task」（タスク：仕事）で構成されています。上記のプログラムは、一つのタスクで、「main」（メイン：主プログラム）という名前が付けられています。

ここで重要な点として、それぞれのNXCプログラムは、一つ以上のタスクを持ち、その一つは「main」の名前を付ける必要があります。このタスクについては、後述のVI章で詳しく説明します。
一つの「task」は、コマンド（commands：命令）群で構成され、これらをステートメント（statements：文）と呼びます。

このタスクに関係するステートメントの集合（brock：ブロック）は、「｛」と「｝」の中カッコで囲んで明確にします。また、各ステートメントの終わりには、セミコロン「；」を付けます。この様にして、ステートメントの終わりと、つぎのステートメントの始まりを明確にします。以下には、一つのタスク内にある複数のステートメント記述を表示しています。

```
task main()
{
    statements1;
    statements2;
   …
}
```

上述した最初のプログラムには、6 つのステートメントで構成されています。それでは、一つずつ説明していきましょう。

OnFwd(OUT_A, 75);

このステートメントは、NXT の出力ポートA（OUT_A）に接続されたモータを前進「OnFwd」駆動するためのものです。この中の数値は、モータ速度を最大速度の７５%に設定した意味となります。

OnFwd(OUT_C, 75);

上述のステートメントと同様に、こちらは出力ポートC（OUT__C）のモータを駆動するためのものです。この２つのステートメント後に、両方のモータは動きだし、ロボットは前進します。

Wait(4000);

このステートメント「Wait」は、つぎのステートメントまでの待機（継続）時間を設定するもので、この例では4秒間ロボットは前進します。このカッコ内にある数値は、アーギュメント（引数）で、1/1000秒単位の整数で設定します。この「Wait」を使うことで待機時間を詳細に設定できます。

OnRev(OUT_AC, 75);

このステートメントで、ロボットは後退「OnRev」します。ここでの注意点は、引数「OUT_AC」を使って、２つのモータを同時に動かしています。つまり、この引数を使うことで、最初の２つのステートメントを、この１つのステートメントで置き換えることができます。

Wait(4000);

このステートメントで、後退は4秒間続きます。

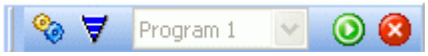
Off(OUT_AC);

最後のステートメント「Off」は、両方のモータ（OUT_AC）を停止するものです。
以上がプログラムのすべてです。これで、ロボットは4秒間前進し、その後4秒間後退して停止します。

このプログラムをタイプするとき、文字に色がつくのに気付かれたでしょう。これらは自動的に色が付きます。この色と文字スタイルは、専用のエディタを使って設定変更することができます。

プログラムの実行

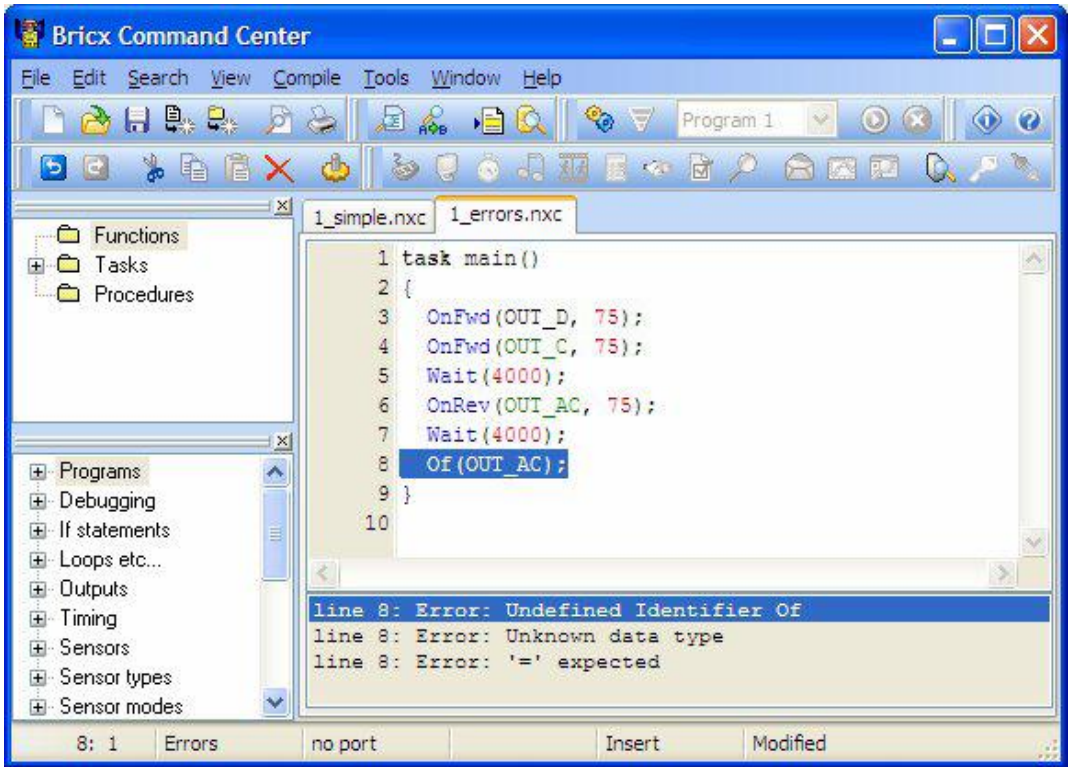
プログラムを作成した後、これらをコンパイル（ロボットが理解し実行できるバイナリ・コードに変換すること）し、USBケーブルかBluetoothドングルを使って、ロボットへ送信（プログラムのダウンロード）する必要があります。



このコマンドバーのアイコンは、左から「コンパイル（Compile Program）」、「ダウンロード（Download　Program）」、「実行（Run Program）」および「停止（Stop Program）」を意味します。2 番目の「ダウンロード（Download Program）」ボタンを押すと、タイプしたプログラムにエラー（誤り）がなければ、コンパイルされ、ロボットにダウンロードされます。（もしプログラムにエラーがあれば、以下を参照ください）
それでは、プログラムを実行してみてください。NXT上の「My Files」を選択し、ソフトウェアのファイル名を設定し、「1_shimple」プログラムを実行します。この名前は、あらかじめNXCファイル名として保存しておいてください。
このプログラムを自動的に実行するためのショートカットとして、「CTRL＋F5」でプログラムをコンパイルし、青のボタンの「実行（Run Program）」を押してプログラムを実行します。ロボットは期待どおりに実行しましたか。もし実行しない場合には、接続をチェックしてください。

プログラムのエラー（誤り）

プログラム・ソースをタイプするときは、入力ミスが起こりやすいものです。この場合のコンパイラは、以下のようなウィンドウの下部位置にエラーメッセージを表示します。



ここでは最初のエラーとして、モータの名前（OUT_D）をミスタイプしているのが発見できます。この場合には、文字カラーが緑色でなく、黒色に表示されています。他のエラーとして、表示されているメッセージを確認してください。その他にも最初はプログラムが原因によって、いろいろなところでエラーが発生している場合がありますので、注意してください。少ないエラーの場合には、すぐに修正し再コンパイルしてみてください。正しい文法でプログラムをタイプした場合には、自動的に文字カラーが変わりますので、エラーの発見が簡単です。
上図の場合には、モータを中止させるコマンドが「Off」ではなく「Of」と間違えているために、文字カラーが青色でなく、黒色で表示されています。つまりコマンドが間違っていると文字カラーは黒色のままとなります。

また、コンパイラによって探せないエラーもあります。例えば使っていもいない出力ポート「OUT_B」があった場合には、文字カラーは緑色に変わりますが、ロボットは異なる動きとなります。もしロボットが期待した動きにならない場合には、プログラムに何か悪い記述があるものと考えてください。

スピードの変更

上述のプログラムで、ロボットのスピードが早いと感じられたでしょうか。このスピードを変更する場合には、コマンド内（OnFwdやOnRevなど）の２番目のパラメータ（引数）を変更します。スピードのパワーは、「０」から「１００」までの数値で入力します。「100」 の場合が最大スピードで、「０」は停止（Stop）の意味となります。（つまりこの０の場合には、NXTのサーボモータは止まった状態となります）以下では、改めてスピードを「ゆっくり」に変更したプログラムを紹介します。

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

まとめ

この章では、「BricxCC（Bricxコマンドセンター）」を使って、NXCの最初のプログラム作成について紹介しました。これで、どうやってプログラムを作成するか、どうロボットにダウンロードするか、またどうプログラムを実行させてロボットを動かすかが分かったでしょう。この他にも「BricxCC」では、多くの機能を装備しています。いろいろと触ってみたり、付属のドキュメント（英文）を読んだりして勉強してみてください。このチュートリアルでは、初歩的なNXC言語の紹介と最低限の「BricxCC」の使い方を紹介しています。
ここでは、NXC言語のいくつかの重要な事項を学びました。まず互いのプログラムには、名前「main」を持つタスクが一つ必要であり、それによってロボットが実行できることです。それに、４つのモータに関する基本的なコマンド（「OnFwd()」、「OnRev()」および「Off()」）と、継続を意味する「Wait()」について学びました。

II. より興味のあるプログラム作成

最初に紹介したプログラムは、特に驚くものではありませんでした。もっと興味のあるプログラムに挑戦しましょう。ここでは、NXC言語プログラムについて、いくつかの興味ある機能を段階的に紹介していきます。

旋回プログラム

ロボットの片方か両方のモータを停止したり、逆回転したりすることで、ロボットの方向を変えてみることができます。例えば、つぎのプログラムをタイプして、ロボットで実行させてみてください。ロボットは、少し前進し、右回転します。

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

この中の２つめの継続コマンド「Wait()」を360から500に変えて、回転角度がどう変わるかを試してみてください。ロボットがどう回転するかは、この数値と関係していることが分かります。この数値をプログラム上で、簡単に変更できるように、数値に名前を付けてみましょう。NXCでは、つぎのように定数定義「#define」を記述します。

```
#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);
}
```

最初の2 行は定数が二つ定義されています。ここで定義した定数は、プログラム内で使うことができます。この定数の定義は、２つのメリットがあります。１つはプログラムを読みやすくすること、もう１つは簡単に値を変えることができることです。ここで注意点として、「BricxCC」では、この定数定義のステートメントの文字カラーを独自に設定することができます。VI章では、これらについてご紹介しています。

コマンドの繰り返し

つぎに、ロボットの動きとして正方形を描くようにしてみましょう。正方形の意味とは、ロボットが前進し、９０度回転し、再度前進し、９０度回転しの繰り返しで、もとの位置に戻るようにすることです。ここでは、４回の前進と90度回転の繰り返しのプログラムを、「repeat」ステートメントを使って簡単になるような記述を紹介します。

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

この「repeat」ステートメントの中の数値（４）は、繰り返しの回数を意味し、さらに中カッコ｛ と ｝とを使って、その繰り返しの範囲を明確にします。
それから、上記のプログラムでは段下げ（インデント）を使っています。これは必ずしも必要ではありませんが、プログラムを読みやすくする上では重要となります。

それでは最後の例として、ロボットを１０回も四角形を描くように実行させるプログラムを作ってみましょう。

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

このプログラムでは、「repeat」ステートメントの中にさらに「repeat」が使われています。この状態を「入れ子（nested：ネスト）」と呼びます。好きなように「repieat」ステートメントを使って入れ子を作ることができますが、開始「｛」と終了「｝」のカッコをペアで使い、さらにインデントを使うことに心がけてください。

コメントの追加

プログラムをより読みやすくするためには、コメントを入れることをお勧めします。各行の先頭に「//」を入れると、その行の実行を無視し、コメントとして扱います。長いコメントの場合には、「/*」と「*/」で囲んでコメントとします。「BricxCC」では、コメント文は、傾斜文字に変更されます。それでは、つぎのプログラムをご覧ください。

```
/* 10 SQUARES (10回の４角形)
  This program make the robot run 10 squares（本プログラムはロボットが１０回４角形を描く）
*/

#define MOVE_TIME 500          // Time for a straight move（直進する時間）
#define TURN_TIME 360          // Time for turning 90 degrees（回転する時間）
task main()
{
  repeat(10)                   // Make 10 squares（１０回繰り返し）
  {
    repeat(4)
    {
      OnFwd(OUT_AC, 75);
      Wait(MOVE_TIME);
      OnRev(OUT_C, 75);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_AC);                 // Now turn the motors off
}
```

まとめ

この章では、「repeat」ステートメントの使い方と、コメントについて学びました。それに、カッコを使った入れ子での関数の扱いや、インデントの使い方も学びました。ここまでで、ロボットをさまざまな経路で動かすことができるようになりました。つぎの章に進む前に、プログラムをいろいろと変更してみて、ロボットを動かしてみてください。

III. 変数の使い方

変数は、どのプログラミング言語でも非常に重要なものです。変数は、値をメモリに保存し、いろいろなステートメントで利用でき、値そのものを変更することもできます。それでは事例を使って変数の使い方をご紹介しましょう。

渦まき（螺旋）起動

上述したプログラムを変更してロボットを渦まき状に動かしてみましょう。このプログラムは、つぎつぎに直進する継続時間を増やすことで実現できます。この互いの時間設定に「MOVE_TIME」を使って、値を増やすことができるでしょうか。それではどうやって実現できるでしょうか。この「MOVE_TIME」は、定数（constant）ですから、変更はできません。代わりに変数を使う必要があります。この変数は、NXCでは簡単に定義できます。以下が、ロボットを渦まきに動かすプログラムです。

```
#define TURN_TIME 360
int move_time;                  // define a variable (変数定義)
task main()
{
   move_time = 200;             // set the initial value (変数の初期化)
   repeat(50)
   {
      OnFwd(OUT_AC, 75);
      Wait(move_time);          // use the variable for sleeping(継続時間の変数使用)
      OnRev(OUT_C, 75);
      Wait(TURN_TIME);
      move_time += 200;         // increase the variable (変数値の増加)
   }
  Off(OUT_AC);
}
```

ここで注意すべき行にコメントが入れてあります。まず、キーワード「int」を使ったステートメント で、任意の名前をつけた変数を定義します。（一般に、小文字を使った名前を変数名として、大文字を使った名前を定数名として使います）この変数名は、英文字で始まり、中に英数字やアンダーラインの文字を含むことができます。（同様に、定数名およびタスク名も同じことが言えます）この「int」は、整数の変数であることを意味し、整数のみがここで定義されます。つぎのコメント行では、この変数「move_taime」に200を設定しています。これ以降の行で、この変数を200として利用することができるようになります。つぎの「repeat」ステートメント内のループ（繰り返し）においては、継続時間にこの変数「move_time」を使い、ループの終わりには変数値を200増加（プラス）させるプログラムとなっています。ループ内の初回は、継続時間は200ms（ミリ秒）で、つぎの回は400msで、さらに３回目は600msで、繰り返します。

このループの最後の行にある「+=」は、左の変数名「move_time」に入っている数値に、右の数値200を増加（プラス）することを意味します。同様に「*=」は、左の変数名に入っている数値に、右の数値を掛け合わせる意味があり、「-=」は右の数値で引き算し、「/=」は右の数値で割り算することを意味します。（注意：割り算の場合には、丸め処理した整数が算出されます）さらに変数を他の変数に加算したり、より複雑な式に使ったりするこもできます。つぎの例は、ロボットを動かすこととは無関係ですが、変数の使い方の例を紹介しています。

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

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3);          // aaa is now equal to 80 (aaa の値は 80)
  ArrayInit(values, 0 , 10);     // allocate 10 elements =0 (配列10 個の要素を0 に設定)
  values[0] = aaa;
  values[1] = bbb;
  values[2] = aaa*bbb;
  values[3] = ccc;
}
```

上述の最初の２行では、変数を複数定義できることを紹介しています。またこの２行で定義した３つの変数（aaa,bbb,ccc）を１つの行で定義することもできます。つぎの行の変数名「values」は配列定義で、複数の数値を扱うことができ、この配列は鍵カッコ「[]」内のインデックス（参照値）を使って利用します。NXCでの整数配列は、つぎの様に定義します。

```
int name[ ];
```

以下の関数は、変数名「values」に10個の配列要素を用意し、値をすべて０（ゼロ）に設定するものです。

```
ArrayInit(values, 0, 10);
```

乱数（ランダム数）

以上紹介したプログラムで、ロボットを思うように動かすことが、ただしく定義できました。しかし、ロボットを動かすとき、予想外の動きをするように、さらに興味あるものにしていきましょう。ある挙動においてランダムにしてみましょう。NXCでは、乱数（random number）を使うことができます。つぎのプログラムでは、この乱数を使って、ランダムにロボットを動かすようにしています。ここでのロボットは、ランダムな時間で前進し、ランダムな角度で回転します。

```
int move_time, turn_time;

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

このプログラムでは、２つの変数を定義し、それらにはランダム数値を設定しています。「Random(600)」は、乱数関数で、リターン値として０から600までの数値が返ってきます。（最大値600は、リターン値には含まれません）毎回数値が変わることをランダムと呼びます。
また変数を使わずに、直接「Wait(Random(600))」とするこもできます。
ここでは、新しいループとして、「**repeat**」の代わりに「**while(true)**」を使っています。この「**while**」のカッコ内

がtrue（真）の状態の場合に、以下のカッコ内のステートメントは、繰り返し実行されます。この中の「**true**」は特別な意味があり、常に真として、以下のカッコ内のステートメントは無限に繰り返し実行されます。（もしくはNXT上のグレーボタンを押すまでは繰り返します）「while」ステートメントについては、さらにIV章で学びます。

まとめ

この章では、変数と配列の使い方を学びました。NXCプログラム言語では、「int（16ビット整数型）」の他に、「short（16ビット整数：intと同様）」、「long（32ビット整数）」、「byte（8ビット：1バイト：0-255）」、「bool（ブーリアン：真/偽：true/false）」、および「string（文字列）」 の変数が使えます。
またここでは、乱数発生について学び、ロボットを予想外の動きをさせることができました。最後に、「while」ステートメントを使うことで無限に繰り返す方法を学びました。

IV. 制御構造

これまでの章で、「repeat」および「while」ステートメントを学びました。これらのステートメントは、プログラム内のほかの実行ステートメントをコントロールするものです。これらは、「制御構造（control structures）」と呼びます。この章では、ほかの制御構造を学びます。

if ステートメント

プログラムの一部分をある条件下で実行したいときがあります。この場合には、「**if**」 ステートメントを使います。それでは事例を使って紹介しましょう。これまで使ってきたプログラムを再び変更し、ロボットを新たに旋回して動かすものにします。ロボットを直進させ、左右の何れかに旋回させます。動作は再度乱数を使って実行させます。ゼロ（= 0）か正数（> 0）か、それ以外のゼロ未満の整数（< 0）かによって、右旋回させたり、左旋回させたりします。以下にプログラムを紹介します。

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

この「if」ステートメントは、「while」とも似ています。 カッコで囲まれた内容判定の状態が「true（真）」の状態のとき、つぎのカギ括弧内のステートメント群を実行します。もし「false（偽）」であれば、「else」以下の鍵カッコのステートメント群を実行します。ここで使っているカッコ内の判定条件を確認してください。ここでは「Random() >= 0」となっています。これは、乱数発生した数字が、ゼロ以上の整数の場合に、「if」のステートメントが「true（真）」となります。この他にも比較値としてどのようなものがあるかをご紹介しておきます。

==	等しい
<	より小さい
<=	以下
>	より大きい
>=	以上
!=	等しくない

また条件を組み合わせて使うこともでき、「and」の場合には「&&」を使い、「or」の場合には「||」を使います。その他の条件で使うものをまとめておきます。

True	常に真
False	常に偽
ttt != 3	ttt が3でない場合に真
(ttt >= 5) && (ttt <= 10)	ttt が5 以上、かつ10 以下の場合に真
(aaa == 10) || (bbb == 10)	aaa が10、あるいはbbbb が10 のとき真

上記の「if」ステートメントは、２つの部分に分かれていました。すぐつぎに続く中カッコ「｛と｝」内のステートメント群は、条件が真の場合に実行され、偽の場合には後に続く「else」ステートメント以下の中カッコ内のステートメント群が実行されます。また、「else」以下の中カッコのステートメント群はオプションで不要な場合もあります。

do ステートメント

ほかの制御構造として、「**do**」ステートメントがあり、以下のように使います。

```
do
{
    ステートメント群;
}
while (条件);
```

「do」ステートメントは、つぎに続く中カッコを実行し、その後の「while（条件）」ステートメントの条件 が真の場合には、さらにステートメント群を繰り返し実行します。条件が偽の場合には繰り返し実行を行いません。この状態は「if」ステートメントを使っても表現できます。それでは、つぎの事例プログラムを使って、ロボットを20秒間、ランダムに動かし、停止（ストップ）させてみましょう。

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

「do」ステートメントは、「while」ステートメントと同様に使用されていますが、「while」ステートメントでは、最初に条件を評価（テスト）して、以下のステートメント群を実行（繰り返す）するかどうかとなりますが、「do」ステートメントは最初にステートメント群を実行した後、条件を評価（テスト）して繰り返し実行するかどうかとなります。つまり「do」ステートメントでは、少なくとも１回は、カッコ内のステートメント群を実行します。

まとめ

この章では、新たらしく「if」と「do」の２つのステートメントの制御構造を学びました。ともに繰り返しステートメントの「repeat」や「while」を一緒に使うことで、繰り返し実行の制御を行います。これらは、ロボットを制御する上でも大変重要なものです。さらにいろいろとプログラムを編集し実行してみてください。

V. センサー

まわりの状況変化によってロボットの動きを変えるために、NXTではセンサー接続を使って実現します。タッチ・センサーを使って、ロボットを動かすために、まずは写真に示したように「Tribot」本体の前にバンパーを組み立てます。

このタッチ・センサーをNXT本体下部にある入力ポート１に接続します。

センサーの待機

それでは、ロボットが何かに接触するまで前進する簡単なプログラムを作成してみましょう。

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

このプログラムでは、２つの重要な行があります。最初の１つめの行は、どのセンサーを利用するかをプログラム上で設定しています。この行では、センサーを接続する入力番号の「IN_1」を設定しています。他のセンサー入力では、「IN_2」、「IN_3」、それに「IN_4」があります。また「SENSOR_TOUCH」は、タッチ・センサーを意味します。光センサーの場合には「SENSOR_LIGHT」を使います。この１行目のセンサー・タイプ仕様設定のちのつぎの行の「OnFwd (OUT_AC, 75) ; 」で、両方のモータを前進させています。さらにつぎの１行は、よく使われている構文で、「until(条件)」の条件が真（true）の状態になるまで、繰り返しを行います。つまり、ここでは設定されたセンサー「SNSOR_1」が「１」である状態、つまりタッチ・センサーが押されている状態になるまで繰り返します。タッチ・センサーが「０」の状態、つまり押されていない状態では継続します。すなわち、このステートメントは、タッチ・センサーが押されるまで継続（wait状態）します。またスイッチを切れば、タスクは終了します。

タッチ・センサーの反応

ロボットが障害物を避けて動くプログラムを作成してみましょう。ロボットが障害物に触れた（タッチ）ときに、ロボットを少し後退させ、方向を変え、再度前進するようにします。

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

前の例と同じように、まずセンサー定義を行います。先ほどと違って、ここでは「SetSensorTouch」関数を使って、入力ポート１「IN_1」に接続しています。つぎにロボットを前進させています。つぎの「while(true)」の無限ループを使い、タッチ・センサーが障害物に接触したかどうかを判断し、 接触した場合のみ、両方のモータで0.3秒間後退し、その後モータAを前進とモータCを後退して回転（右旋回）を同じく0.3秒間行い、改めて両方のモータで前進することを行っています。

ライト・センサー（光センサー）

NXTシステムの購入時には、タッチ・センサーの他に、ライト・センサー（光センサー）、サウンド・センサー（音センサー）、超音波センサーが標準セットとして付属しています。＜現在の「Mindstroms NXT 2.0」(玩具版：商品番号8547)では、タッチ・センサーが2個、超音波センサーとカラー・センサー(色センサー)が各1個づつ付属しています＞
ライト・センサーにある発光LEDをオン/オフすることで、特定の方向で反射する光量や周囲の光量を調べることができます。この反射光量を調べることで、床上のラインに沿ってロボットを動かすことができます。つぎの事例で、どう動くかを見てみましょう。そのためにはまず「Tribot」ロボットに各センサーを接続します。以下の写真のように、ライト・センサーを入力ポート３「IN_3」に、サウンド・センサーを入力ポート２「IN_2」に、それと超音波センサーを入力ポート４「IN_4」に接続します。

ここでは、NXT販売セットに付属しているテスト・パッド（黒いラインの入った大きな用紙）が必要となります。ロボットを、黒いライン上に沿って動かすには、ライトを黒いラインの境界線上に保つことが基本となり、ライトがライン上の場合には反射光が低くなり（暗い色の場合）、黒いライン上からずれるように動かし、ライトがラインの外にある場合には反射光が高くなり（明るい色の場合）、もと（ライン上）に戻るように動かします。

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

このプログラムの最初には、「SetSensorLight」関数 に、入力ポート3「IN_3」を設定し、ライト・センサーを接続定義しています。つぎに、両方のモータを前進「OnFwd（OUT_AC,75）」させています。さらに「while(true)」 の無限ループでは、センサーが感知した反射光「Sensor(IN_3)」の値が40（定数「THRESHOLD」）以上になると、片方のモータC の回転方向を変えてトラックに戻るまで動き続けるようにします。この間0.1秒「Wait(100)」ごとに再度センサーでの反射光をチェック（「until(・・・)」）し、黒いライン上に戻ったら、改めて両方のモータで前進させます。
このプログラムを実行してみるとお分かりのように、スムーズには動いてはくれません。上記のプログラムには、ロボットの動きで「until」 前に「Wait(100)」を入れ、モータAが前進とモータCが後退する回転を 0.1秒間 設定し、その後「until」のチェックで、黒いラインに戻るまで継続し、黒いラインの戻った時点で、つぎのステップで、再度前進「OnFwd(OUT_AC,75)」し続けます。このプログラムでは、反時計回り（左回り）ではうまく動きません。さらに経路（黒いライン）が恣意的な場合には、さらに複雑なプログラムにする必要があります。
LEDライトをオフの状態でまわりの光量を計測するには、センサーをつぎのように設定します。

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

サウンド・センサー（音センサー）

サウンド・センサーを使って、まわりの音を感知して動くロボットをつくることもできます。大きい音を感知すれば停止し、つぎの音を感知してふたたび前進するようなものです。それではマニュアルに沿って「Tribot」の入力ポート2にサウンド・センサーを接続してください。

```
#define THRESHOLD 40
#define MIC SENSOR_2
task main()
{
  SetSensorSound(IN_2);
  while (true)  {
    until (MIC > THRESHOLD);
    OnFwd(OUT_AC, 75);
    Wait(300);
    until (MIC > THRESHOLD);
    Off(OUT_AC);
    Wait(300);
  }
}
```

このプログラムの最初には、音量閾値の定数「THRESHOLD」と「SENSOR_2」の略名「MIC」を定義しています。つぎの「main」タスクでは、まず「SetSensorSound」関数でサウンド・センサーを入力ポート２「IN_2」に設定し、「MIC」に「SENSOR_2」の値データが読めるように設定します。さらに無限ループの「while(true)」に移ります。
（注意：この「MIC」には、サウンド・センサーで計測された音量レベルの値が変化量として入ります）

この無限ループ内の「until」ステートメントを使って、音量レベル（MIC）が閾値（THRSEHOLD）より大きくなるまで待機します。ここでの「SENSOR_2」は、センサー名を表わすのではなく、サウンド・センサーの値を返すマクロの一つとなります。

大きい音が発生するとロボットは動きだし、また大きい音の発生で停止し、その繰り返しを行います。

この中の「Wait」ステートメントが無い場合には、一瞬のうちにロボットは動いて止まる動作となります。実際にこの２つの「until」ステートメント間での前進と停止の実行時間もない状況となります。また、この２つの「Wait」ステートメントをコメント行にして、どう動くかを見てください。
つぎに、「until」行の代わりに「while」を使ったときのステートメントは以下のようになります。

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

ライト・センサーとサウンド・センサーのようなNXTのアナログ・センサーは、単純な０から１００までの値のみを返します。

超音波センサー

超音波センサーはソナーと同様な働きがあります。概要としては、超音波を発信し、障害物に当たって返ってくる超音波の反射時間を計測するものです。このセンサーはデジタル・タイプで、内蔵にはデータ解析と送信のできる機能を備えています。このセンサーでは障害物を検知したり、その障害物を避けて通るようにしたりすることができます。

```
#define NEAR 15 //センチ単位
task main()
{
    SetSensorLowspeed(IN_4);
    while (true)  {
      OnFwd(OUT_AC, 75);
      while (SensorUS(IN_4) > NEAR);
      Off(OUT_AC);
      OnRev(OUT_C, 100);
      Wait(800);
    }
}
```

このプログラムでは、「SetSensorLowspeed」関数 で超音波センサーを入力ポート4 に設定しています。このプログラムでは、無限ループで障害物までの距離が15 センチ以下で発見されるまで前進し続けます。15センチ以下になると、両輪を止め、0.8秒間だけモータC「OUT_C」を後進して方向を変え、「while」ステートメントの最初の命令に戻って再び前進します。

まとめ

この章ではNXTの標準セットに入っている全てのセンサーがどう働くかについて見てきました。また、「until」と「while」ステートメントの命令とセンサーの利用のしかたについても見てきました。
もっと上達する上では、自分自身でここでのプログラムを変更・編集しロボットを動かしてみてください。より複雑な挙動をするプログラムを組込んでいくことができ、さらにNXTのロボット・センター内にあるソフトウェアをNXCプログラムに変更してみてください。

VI. タスクとサブルーチン

ここまで紹介してきたプログラムは１つのタスクしか含まれていませんでした。NXCプログラムでは、複数のタスクを持つことができます。また、プログラム内のいろいろなところで使われるプログラムを１つのサブルーチンと呼ばれるブログラムに作成することができます。これらのタスクやサブルーチンは、プログラム全体を分かり易くすることができ、さらにコンパクト（簡素化）することができます。この章では、いろいろな可能性についてご紹介していきます。

タスク

NXC のプログラムでは、最大255 までのタスクを持つことができ、各タスク名には重複しない（ユニーク：唯一な）名前を付けます。また、最初に実行するタスク名には、必ず「main」と呼ばれる名前を付けます。「main」以外のタスクは、他のタスクから呼ばれて実行されるか、「main」で記述されているとき実行されます。「main」タスクから実行される場合には、他のタスクがスタートする前に「main」を終了する必要があります。その時点から、両方のタスクは同時に並行して実行します。（並列処理）

それでは、タスクの使い方を見てみましょう。まずは、前述したようにロボットが四角形の軌道を描くプログラムを作成します。ただ、障害物があった場合には、それを避けるように動かします。これを一つのタスクで動かすには難しいものがあります。それは、同時に２つのことをする必要があるからです。１つは、モータを動かしたり止めたりして四角形軌道を動かし、もう１つはセンサーで監視することです。この場合には、２つのタスクを使って、１つは四角形の軌道を動くプログラムと、もう１つはセンサーで反応して動くことです。以下にプログラムを紹介します。

```
mutex moveMutex;

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

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

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

このプログラムの「main」タスクでは、センサー・タイプを設定し、他のタスクを２つ実行させているだけです。この２つのタスクは、スケジューラのキュー（queue:待ち行列）にタスクを挿入し、「main」タスクはその後終了しています。ここでの「move_square」タスクは、ロボットを四角形に動くようにプログラミングされています。また「check_sensors」タスクは、タッチ・センサーが障害物に触れたかどうかを判定し、その障害物を回避するプログラミングとなっています。

ここでの重要点は、これら２つのタスクが同時に動くことと（並列処理）、前進し続けながら予想しない障害物があってもプログラムが実行する（割り込み処理）ができることです。

このプログラム内での問題解決として、「mutex」（意味は「**mut**ual **ex**clusion」：相互排除）による変数「moveMutex」を定義し、「Acquire」と「Release」の２つの関数で、ある時点でのモータの動きは１つのプログラムでしか制御できないという排他処理を行っています。

ここでの排他処理のための変数を「セマフォー（semaphore:信号装置）」と呼び、この様なプログラムを並列処理（コンカレント）プログラムと呼びます。さらに詳しくは、X章にて述べています。

サブルーチン

プログラムの中のいくつもの場所に、同じコードが必要になることがあります。この場合、サブルーチンに名前を付けたコードを組込みことができます。このサブルーチン名を使ってタスク内で簡単に呼び出すことで、これらのコードを実行させることができます。以下に事例を使って紹介しましょう。

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

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

このプログラムでは、ロボットをセンターまわりに回転するサブルーチン「sub」を定義しています。メイン・タスクでは、このサブルーチンを3回呼び出しています。サブルーチンは、その名前の記述と、後に続く括弧内の数値引数をもって設定します。もしサブルーチンに引数が無い場合には、括弧内に何も無い状態で設定します。

これまで見てきたコマンドで確認してみましょう。

サブルーチンの主な利点は、一つのみがNXT上で保存され、メモリを削減できることです。しかしながら、サブルーチンが短い場合には、直接ライン上に「inline」関数を使った方が良い場合があります。これらは個別に保存されるのではなく、使われているところにそれぞれ配置されます。これはメモリを余分に使いますが、「inline」関数の制限はありません。これらは以下のように宣言します。

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

「inline」関数の宣言や呼び出しは、サブルーチンと同じようになります。

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

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

上記の例では、関数の引数として、回転する時間を設定することができ、以下にその例を示します。

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

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

インライン関数名の後の括弧内には、関数の引数群を宣言します。この例では、引数は整数型「int」で、その名はpwr と turntime が宣言されています。複数の引数を宣言するときは、カンマで区切って記述します。NXCでは、「sub」は「void」型と同じで、値を返さないものとなります。関数は値を返すもので、整数や文字などとなります。詳細はNXCガイドを参照ください。

マクロの定義

NXCでは、その他にコード名を使ってより小さくする方法として、マクロ宣言があります。（「BricxCC」のマクロとは異なります）すでに定数を宣言するものがマクロ宣言で、「#define」を使って名前を宣言します。もちろん他のコードを宣言することもできます。ここでは、前と同じプログラムとなりますが、回転におけるマクロを使っています。

```
#define turn_around      ¥
   OnRev(OUT_C, 75);Wait(3400);OnFwd(OUT_AC, 75);

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

「#define」ステートメントの後に設定されている「turn_around」には、その後のテキストが定義されています。これによってその後に「turn_around」をタイプしたところには、コンパイル時にこれらのテキストが入れ替わります。（ただ、この「#define」宣言されるとき複数行にまたがることがありますが、これは推奨できません）

「Define」ステートメントは、特に強力なものです。引数を持つこともできます。例えば、ステートメント中に引数として時間を設定することができます。以下の例では、４つのマクロを宣言しています。それぞれ右回転、左回転、前進、それに後退としています。これらは、互いに２つの引数を持っていて、スピード時間と持続時間となっています。

```
#define turn_right(s,t) OnFwd(OUT_A, s);OnRev(OUT_C, s);Wait(t);
#define turn_left(s,t) OnRev(OUT_A, s);OnFwd(OUT_C, s);Wait(t);
#define forwards(s,t) OnFwd(OUT_AC, s);Wait(t);
#define backwards(s,t) OnRev(OUT_AC, s);Wait(t);

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

これらは、特に便利なマクロを宣言しています。これらは、コードをよりコンパクトにし、読みやすくしています。もちろん接続されているモータの変更などでも、より簡単に修正できます。

まとめ

この章では、タスク、サブルーチン、インライン関数およびマクロの使用について学習しました。これらの使用方法はそれぞれで異なります。タスク「task」は、同時に実行するときや、異なる作業を同時に動かす時などに使います。サブルーチン「sub」は、同じタスクを異なる場所で使うコードが、大きい場合に便利となります。インライン「inline」関数は、異なるタスクで、多くの異なる場所で使う場合に便利となります。ただ、メモリは増大します。最後のマクロ「define」は、異なる場所で利用する小さいコードの場合には非常に便利です。これらは全て引数を持つことができ、より便利な使い方となります。

これからの章では、ロボットをより複雑に動かすことを学習していきます。その他、このチュートリアルでは、特殊な応用で重要なことも紹介しています。

VII. ミュージックの作成

NXT は内蔵のスピーカを持ち、音を出力したり、サウンド・ファイルを再生したりできます。また特殊な使い方としては、NXTに何かが起きた時に、音で知らせてくれるなどができます。さらに、ロボットがミュージックを作成したり、動いている間に喋ったりなど、面白く工夫することもできます。

サウンド・ファイルの再生

「BricxCC」は、メニューツールで、サウンド・ファイルの「.wav」ファイルを「.rso」ファイルに変換する機能を組み込んでいます（サウンド変換ツール）。

NXTのフラッシュメモリに「.rso」ファイルを読み込んで、NXTメモリに表示させ（Tools->NXTエクスプローラ）、「PlayFileEx」コマンドを使ってプレイすることができます。

PlayFileEx (filename,　volume,　loop?)

ここでの引数「filename」はサウンド・ファイル名、「volume」はボリューム（音量：０～４）、それと「loop？」は繰り返しの意味で、「１」（TRUE）の設定で繰り返し、「０」（FALSE）で１回のみのプレイ（再生）となります。

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

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

このプログラムを実行させると、既にお分かりのようなリズムの音を出します。よく使うサウンドのマクロを定義し、それをmainタスクでは簡単に何度も使用しています。ボリュームを変えたり、リズムを変えたりして、遊んでみてください。

ミュージックの再生

音の再生では「PlayToneEx(frequency, duration, volume, loop?)」コマンドを使います。

このコマンドには４つの引数があります。「frequency」は周波数（音の高さ）、「duration」は間隔（音の長さ：１/１０００秒間（ミリ秒）単位の設定で「Wait」の引数と同じ）、また「volume」と「loop?」は前述したものとなります。また「PlayTone (frequency, duration)」コマンドも同様に使いますが、「volume」はNXT上のメニューで設定し、「loop?」は不要です。

以下に、周波数（音の高さ）の一覧表を示します。

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

単に「PlayFileEx」コマンドを使った場合、サウンドは聞こえてきません。2 つの音の間に「Wait」を入れて、「PlayFlexEx」コマンドの持続間隔を追加する必要があります。つぎに使用例を紹介します。

```
#define VOL 3
task main()
{
  PlayToneEx(262, 400, VOL, FALSE); Wait(500);
  PlayToneEx(294, 400, VOL, FALSE); Wait(500);
  PlayToneEx(330, 400, VOL, FALSE); Wait(500);
  PlayToneEx(294, 400, VOL, FALSE); Wait(500);
  PlayToneEx(262, 1600, VOL, FALSE); Wait(2000);
}
```

「BricxCC」では簡単にピアン鍵盤を用意してサウンドを作成する「Brick Piano」を用意しています。NXTを動かしながら音楽を奏でるには、タスクを分けて使ってください。つぎに、NXTを前後に動かしながら、サウンドを演奏するプログラムを紹介します。

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

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

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

まとめ

この章では音やサウンドを作成することを学習しました。またサウンドを別タスクとして利用することも見てきました。

VIII. モータの詳細設定

モータをより精密に制御するためのコマンドは、他にもいろいろと持ち合わせています。この章では「ResetTachoCount」、「Coast(Float)」、「OnFwdReg」、「OnRevReg」、「OnFwdSync」、「OnRevSync」、「RotateMotor」、「RotateMotorEx」、および基本的な「PIDの 概念」について学習していきましょう。

減速停止

「Off()」コマンドを使った場合、サーボモータは即座に停止し、車軸も止まり、位置を固定します。NXTのサーボモータは、減速してゆっくりと停止することもできます。この場合、「Float()」や「Coast()」コマンドをそれぞれ使って、ゆっくりとモータの力を簡単に停止できます。ここでプログラム例を紹介します。最初は、「Off()」コマンドを使って即座に停止させ、つぎに「Float()」コマンドを使って、減速させて停止させています。この違いを理解してください。ここでのロボットでは、違いは微妙でしょう。他のロボットを用いてテストした場合には、違いが明確になることがあります。

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

上級コマンド

この「OnFwd()」と「OnRev()」コマンドは、モータを動かすための基本的なルーチンです。
NXTのサーボモータは、車軸の回転やスピードを正確に制御するための機能を持ち合わせていて、NXT内蔵のファームウェアには、モータ駆動でのフィードバックを行うPID制御（Proportional Integrative Derivative：後述）を実装しています。
もし、ロボットを正確に前進させるとき、両方のモータの同期をとるよう、つまり一方のモータが遅れたり、互いに遅くなったりしてもモータの動きを調整して同じになるようにこの制御を使います。さらに、右回転や左回転では、互いのモータのパワーにパーセントを使って同期をとって制御したりすることもできます。 これらのようにサーボモータをフルに活用するための多くのコマンドがあります。

「OnFwdReg(ports, speed, regmode)」コマンドでは、「potos」で出力ポートを、「speed」で速度を、それに「regmode」で調整モードを指定し、モータを制御します。ここでの「regmode」調整モードでは、OUT_REGMODE_IDLE（「IDLE」モード）、OUT_REGMODE_SPEED（「SPEED」モード）、およびOUT_REGMODE_SYNC（「SYNC」モード）を使用します。この「IDLE」モードは、PID 制御なしでの状態で、つぎの「SPEED」モードは、１つのモータがどんな場合でも一定速度で動くように調整する状態、それに「SYNC」モードは、前に述べたように同期をとって２つのモータを動かす状態のことを意味します。

「OnRevReg()」は、前述したコマンドの逆方向へ動かすものです。

```
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);
}
```

このプログラムを、ロボットの両方のモータが回転しないように手で止めた状態で動かしてみます。最初の「IDLE」モードでは、手の力の入れ方によって片方のモータが動かなくなる状態になります。つぎの「SPEED」モードでは、手で止めようとしても一定にモータを強く動かすような状態になります。最後の「SYNC」モードでは、１つのモータを手で止めようとすると、片方のモータも止まるような両モータが同期をとる状態になります。

「OnFwdSync(ports, speed, turnpct)」コマンドは、「OnFwdReg()」コマンドの「SYNC」モードと同様に動きます。しかし、新たな引数「turnpct」の設定によってスピードをパーセンテージ（－１００から１００まで）で調整できます。

「OnRevSync()」は、同様にモータを逆方法へ動かすもとなります。つぎのプログラムは、これらのコマンドを使っています。いろいろとパラメータを変更し実行させてみてください。

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

最後に、モータを回転角度で設定してみましょう（１回転は３６０度です）

つぎの両方のコマンドは、回転角度に符号を使ってモータを動かしています。マイナスを付けたモータは、逆方向へ回転することになります。

「RotateMotor(ports, speed, degrees)」の引数の「ports」は出力ポート、「speed」はスピード（０～１００）、それと「degrees」は回転角度となります。

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

「RotateMotorEx(ports, speed, degrees, turnpct, sync, stop)」は、上述の拡張コマンドで、両方のモータ（例えばOUT_AC）の同期をとって動かすもので、新たな引数「turnpct」は操縦パーセンテージ（－１００～１００）で、「sync」と「stop」は共に真偽値（真＝１、偽＝０：ブーリアン変数）で、「stop」は回転角度を正確に止めるかどうかの引数となる。

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

PID 制御

NXT のファームウェアでは、デジタルによるPID制御（Proportinal Integrative Derivative：比例・積分・微分）で、サーボモータの位置とスピードを正確に制御することができます。この制御方式は、より効果的な閉じたループのフィードバック制御の中の最も簡単な一つです。

簡単に言えば、ある値から、定値の定温度や定速度になるまで温度（速度）を上下させるとき、現値と定値の誤差による計算（P）、時間変化での値を、積分による計算（I）や微分の計算（D）によって、コントロール（制御）する方法となります。

プログラムでは、現在位置から達成値（定値） R(t) までに移動させるために、 モータ動作を指令 U(t) し、組み込んだエンコーダで新たな現在位置（現値） Y(t) を計測して、その達成値との差（誤差）を E(t)＝R(t)-Y(t) で求めます。ここで「閉ループ制御」は、計測された現在位置 Y(t) を新たな誤差計算に使っているからです。この制御では、ここで求まった誤差 E(t)を、モータを動かすための指令U(t)に変換します。そこで

U(t) = P(t) + I(t) + D(t), ここで
P(t) = Kp・E(t)
I(t) = Ki・(I(t+1) + E(t)) および
D(t) = Kd・(E(t) – E(t-1))

この式は、初心者にとってはとても難しく見えますが、できる限りこのメカニズムを分かりやすく説明しましょう。
この指令 U(t) は、３つの式の総和で、この比例部分が「P(t)」で、積分部分が「I(t)」、それに微分部分が「D(t)」となります。

「P(t)」は、装置(モータ)を瞬時に動かしますが、誤差をゼロにすることはできません。

「I(t)」は、制御において記憶装置を持ち、誤差変動を計測して、補正し、ゼロまでの安定状態を行います。

「D(t)」は、制御において予測装置を持ち、高速な応答を（微分計算で）実現しています。

これだけで理解できるとは思いません。もし時間があれば、ここで記述している内容を、専門書によってご確認ください。
ここではNXTを接続して、具体的にどう動くか見てみましょう。つぎの簡単なプログラムを使って学習してみましょう。

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

task main()  {
  RotateMotorPID(OUT_A, 100, 1800, P, I, D);
  Wait(3000);
}
```

この「RotateMotorPID(port, speed, angle, Pgain, Igain, Dgain)」を使って、いろいろな PID値でモータを動かしてみてください。例えばつぎのように値を設定してみてください。

(50, 0, 0) ：モータは、補正なしの誤差が残るため正確に180度回転しません。
(0, x, x) ：比例制御が無いため、誤差は大きくなります。
(40, 40, 0) ：目標位置を超え、戻ろうとします。
(40, 40, 90) ：精度よく起動します（時間は設定立ち上がり時間（指定位置に着くまでの時間）
(40, 40, 200)：微分のゲインは多すぎるため車輪は不安定に動きます。
他の値も変更してモータの反応にどう影響するかを試してみてください。

＜補足説明：本プログラムのコンパイル時にファームウェアエラーが表示された場合には、BricxCC画面上のメニューバーの「Edit」から、環境設定「Preferecnes…」を選択し、「Preferences」画面上の「Compiler」タグのなかのさらに「NBC/NXC」タグを選択します。この中の「Enhanced firmware」（拡張ファームウェア）をクリック（選択）し、「OK」ボタンで終了させ、BricxCCを再起動して再コンパイルしてみてください＞

まとめ

この章では、高度なモータ制御用のコマンド群を学びました。「Float()」と「Coast()」コマンドは、モータを減速停止する機能、「OnXxxReg()」と「OnXxxSync()」コマンドは、モータのスピードと同期をフィードバック制御する機能、さらに「RotateMotor()」と「RotateMotorEx()」コマンドは、正確に車軸を回転角度で制御する機能となります。さらにPID制御についても学びましたが、それほど詳しくは説明していませんので、Webサイトより調べて、もっと正確に学習してください。

IX. センサーについて

第V章では、センサー利用の基本的な特徴について述べました。ここではさらにセンサーの可能性について詳しく紹介します。この章では、センサーのモードとタイプについての違いについて学習し、古いコンパチブルのRCXセンサーをLEGO変換ケーブルでNXTに接続することを学びます。

センサーのモードとタイプ

前述の「SetSensor()」コマンドでは、２つの設定について学びました。１つは、センサーのタイプを設定し、もう１つがセンサーの働きのモードを設定しました。このセンサーをそれぞれモードとタイプに分けて設定することで、さらに特殊なアプリケーションのために使いやすくもっと正確にセンサーの挙動を制御することができます。

センサーのタイプは「SetSensorType()」コマンドで設定できます。このコマンドには、多くの異なるタイプがありますが、ここでは主なものを紹介します。タッチ・センサーは「SENSOR_TYPE_TOUCH」で、LEDの光センサーは「SENSOR_TYPE_LIGHT_ACTIVE」、サウンド・センサーは「SENSOR_TYPE_SOUND_DB」、それと超音波センサーは「SENSOR_TYPE_LOWSPEED_9V」となります。センサーのタイプ設定では、センサーのパワー（例えば、LEDの光センサーの光量など）の指示や、デジタル通信でI2Cセンサープロトコルを使うようにNXTへの指示などがあります。また、古いRCXのセンサーをNXTで使うように設定することもでき、温度センサーの「SENSOR_TYPE_TEMPERATURE」、古い光センサーの「SENSOR_TYPELIGHT」、RCXの回転センサーの「SENSOR_TYPE_ROTATION」（別途後述）などがあります。

センサーのモードは「SetSensorMode()」コマンドで設定できます。モードの設定には8種類あります。最も重要なひとつに「SENSOR_MODE_RAW」があります。このモードでは、センサーの値を 0 から 1023 の間の数値で確認できます。つまり、センサーの未処理の値（raw value）を出力します。現在のセンサーの値に依存したものとなります。例えば、タッチ・センサーでは、何も押されていない状態での値は1023に近づきます。また確実に押された状態での値は50に近づきます。部分的に押された状態での値は 50～1000 となります。この様に、タッチ・センサーのモード設定で未処理の値を使うことで、部分的に押されている状況かどうかを実際に知ることができます。光センサーの場合には、300（非常に明るい）から 800（非常に暗い）の値を示します。このコマンドは「SetSensor()」コマンドよりも、もっと正確な値を返します。詳細については、NXCプログラミングガイドを参照ください。

つぎに紹介するセンサー・モードは「SENSOR_MODE_BOOL」です。このモードでは、 0 か 1 かの値となります。上で説明した未処理の値（raw value）が 562 よりも上の場合には 0 になり、その他は 1 となります。この「SENSOR_MODE_BOOL」はタッチ・センサーの省略値ですが、他のセンサーでもアナログデータを簡略化したものとして利用できます。また「SENSOR_MODE_CELSIUS」と「SENSOR_MODE__FAHRENHEIT」モードは温度センサーのみで利用でき、示された方法での温度を返します。さらに「SENSOR_MODE_PERCENT」モードは未処理の値を 0 から 100 までの値で返します。このモードは光センサーの省略値のモードとなります。「SENSOR_MODE_ROTATION」モードは、回転センサーのみに利用されます。（後述）

他に２つの興味あるモード「SENSOR_MODE_EDGE」と「SENSOR_MODE_PULSE」があります。これらは変化量を測り、未処理の値が小さい値から大きい値への変化やその逆の値の変化となります。例えば、タッチ・センサーに触れたとき、大きい値から小さい値への未処理の値に変化します。また手を離した場合には、逆な方向に値が変わります。「SENSOR_MODE_PULSE」モードを設定した場合には、小さいほうから大きい値のみの変化量しか出力しません。それでタッチ・センサーに触れて離した場合は１回のカウントとなります。「SENSOR_MODE_EDGE」モードを設定した場合には、両方への変化でのカウントを行います。よってタッチ・センサーに触れて離した場合には２回カウントされます。これによって、何回タッチ・センサーが押されたかを知ることができます。もしくは光センサーとの組み合わせで、何回ライトのスイッチをオン・オフしたかのカウントに利用できます。もちろんこのモードを使う場合には、カウンタを 0 に戻すことも必要です。「ClearSensor()」コマンドを使って、計測したセンサーのカウンタを消去することができます。

それでは事例を使って学習していきましょう。つぎのプログラムはタッチ・センサーを使ってロボットを動かしています。タッチ・センサーを長いケーブルでNXTの入力ポートのひとつに接続してください。瞬時に２度タッチ・センサーに触れた場合には、ロボットは前進します。もし１度だけだと動きを止めます。

＜ I2C はInter-Integrated Circuit （I-squeqrd C）の略で、低速な周辺機器用の接続シリアルバス（フィリップス社開発）＞

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

ここで重要な点として、最初にセンサーのタイプを設定し、つぎにモードの設定を行うことです。このことは基本的なことで、センサー・タイプの変更は、モードに影響するからです。

回転センサー

回転センサーは非常に便利な機能で、NXTのサーボモータに内蔵された光学エンコードです。この回転センサーは、相対的な角度位置を計測でき、車軸穴を持っています。完全な１回転を１６段階（もしくは逆方向だと－１６）でカウントし、１段階を 22.5 度の回転となります。ただ、サーボモータが持つ1度ごとの回転に比べれば荒くなっています。この以前からある回転センサーは、モータを回転させることなく、車軸の回転を表示させ、モータを動かすトルク（回転モーメント）からモータを利用することができます。以前の回転センサーでは簡単にモータを回転できます。
もし、この１６段階でモータの回転を利用する場合には、１回転ごとこの数値を増やすことで、いつでも機械的にギアを利用することができます。
つぎの例は、古いRCXでのチュートリアルからの使っているものです。

標準的なアプリケーションでは、２つモータ制御をもつロボットで、２つの軸に接続した２つの回転センサーを持ちます。まっすぐにロボットを動かすには、両方の車輪を同等に回転させます。しかしながら、モータは必ずしも正確に同じスピードでは動きません。この回転センサーを使うことで、１つの車軸を高速に回転できます。両方の回転センサーが、同じ値になるまで、一時的にモータを停止（「Float()」を使って）させることができます。つぎのプログラムでこのことを行っています。これでロボットをまっすぐな線上で動かすことができます。このプログラムを使って、２つの回転センサーを２つの車軸に接続し、ロボットの動きを変更してみてください。接続するセンサーは、入力ポート１と３に接続してください。

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

このプログラムでは、まず両方のセンサーが回転センサーであることを設定し、値をゼロにリセットしています。つぎからは無限ループとなっています。このループ内では、２つのセンサーが同じであるかをチェックします。ロボットが前進し、一方の回転センサーの値が大きい場合には、両方の値が等しくなるまでモータを停止します。
明らかに、このプログラムは非常に簡単なものです。このプログラムを使って、ロボットを正確な距離まで動かすか、正確に回転するように拡張してみてください。

ひとつの入力ポートに複数センサー接続

ここで、注意事項として、以前（RCXで）は簡単に同じ入力ポートを増設・接続することができていましたが、NXTでは６個の増設ケーブル数を超えて拡張することはできなくなりました。ここでの提案として、唯一確実（かつ簡単）な拡張する方法としては、接続ケーブルを使って、タッチ・センサーをアナログによる重ね合せた装置で利用することです。もう１つをデジタルによる重ね合わせで、I2CによるNXTで接続して利用しますが、このことは初心者にとっては簡単な解決方法ではありません。

NXTには、４つの入力ポート数しかありません。より複雑（センサーを拡張追加した）なロボットをつくる場合には、この入力ポート数では満足いくものではありません。幸運にも、特殊な方法で、２つ（さらにもっと）のセンサーを１つの入力ポートに接続することができます。

最も簡単な方法としては、２つのタッチ・センサーを同じ入力ポートに接続することです。もし、これらのうちの１つか両方のタッチ・センサーの値は「１」か「０」になります。ただし、どちらのタッチ・センサーが押されたかは区別できませんが、この区別が必要ではない場合があります。例えば、ロボットの前方と後方にタッチ・センサーを設置し、ロボットがいずれかに動いたときに１つのタッチ・センサーが反応する場合です。しかも、上述したような未処理の値（raw value）を設定することもできます。さらに多くの情報を取得することができます。当然、多くの場合、両方のタッチ・センサーが同時に反応することはありません。２つのタッチ・センサーの違いを、正確に区別することもできます。２つのタッチ・センサーが押された場合には、３０よりもさらに低い値が返ってきてきます。

さらに、タッチ・センサーと光センサーを同じ入力ポートに接続（RCXでは１つ）できます。この場合、接続タイプを「光センサー」（ただし、光センサーは作動しません）にし、接続モードを未処理の値（raw value）に設定します。この場合、タッチ・センサーが押されたときに値は「１００」以下になります。タッチ・センサーが押されていない時は、光センサーの値が「１００」以下になることはありません。このロボットでは、光センサーを下方に向け、前方のバンパーにタッチ・センサーを配置します。両方のセンサーを入力ポート「１」に接続します。このロボットは、明るい状態ではランダムに動き回ります。光センサーが暗く反応した場合（未処理の値＞７５０）少し後退します。またタッチ・センサーが何かに触れた場合（未処理の値＜１００）同様に後退します。以下にプログラムを記載します。

```
mutex moveMutex;
int ttt, tt2;

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

task submain()
{
  SetSensorType(IN_1, SENSOR_TYPE_LIGHT);
  SetSensorMode(IN_1, SENSOR_MODE_RAW);
  while (true){
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750)){
      Acquire(moveMutex);
      OnRev(OUT_A, 75);Wait(30);
      Release(moveMutex);
    }
  }
}

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

このプログラムは２つのタスクから成り立ちます。「moverandom」タスクは、ロボットをランダムに動かすようにしています。「main」タスクでは、最初にこの「moverandom」を起動させ、センサーを設定し、その後何が起きるかを待ちます。もしセンサーの値が低くなる（タッチ・センサー押された）場合か、高くなる（光センサーが明るいところから外れた）場合には、ランダムに動くのを止め、少し後退し、再度ランダムに動くようになっています。

まとめ

この章ではセンサーについて多くの詳しい内容を学びました。センサーの接続タイプと接続モードをそれぞれどう設定するか、それにその他の情報をどう利用するかを見てきました。また回転センサーの使い方についても学びました。さらに、どう２つのセンサーを重複させてNXTの１つの入力ポートに接続するかも見てきました。これらの秘訣は、もっと複雑なロボットを組み立てるとき、とても便利なものとなります。センサーは、常時これらの重要な役割に活用できます。

X. 並列タスク

上で指摘したように、NXCのタスクは、同時もしくは並列で、実行することができます。このことは、非常に便利なものです。１つのタスクでセンサーを見ながら、他のタスクでロボットを動かし、さらに他のタスクで音楽をプレイすることができます。しかし並列処理は、問題発生の原因ともなります。たとえば、あるタスクが他のタスクを妨害することです。

間違ったプログラム

つぎのプログラムを見てみましょう。この１つめのタスクでは、ロボットが四角形を描くように動かし（以前記述したプログラム）、２つのタスクではタッチ・センサーをチェックしています。タッチ・センサーが何かに触れたとき、少し後退させ、９０度回転させています。

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

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

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

このプログラムでは、問題は無いように見えます。しかし、実行させてみると、必ずしも期待したような動きになっていないことに気付きます。つぎのことを実行してみてください。回転しているときに、何かをロボットにタッチしてみてください。後ろに動きだし、瞬時に再び前進し、障害物に衝突します。この理由は、タスクによる妨害があるためです。つぎに問題が発生しています。ロボットが右折している時、最初の「check_sensors」タスクでは休憩に入っています。今、ロボットがセンサーに衝突したとき、この最初の「check_sensors」タスクで、後退しはじめ、その一瞬で、「main」タスクでは休息待ち状態となり、再びロボットを障害物の方向へ前進させます。つぎの「submain」タスクは、この時休息に入っていて、衝突に気付きません。このことは、期待した動きをするかどうかを明確にすることできません。この問題は、２つめの「submain」タスクが休息の間、最初の「check_sensors」タスクが動いている状態にあるからで、しかも２つめの「submain」タスクの実行を妨害するようになっているからです。

危険範囲とmutex 変数

上記の問題を避ける１つの方法として、どんな状態でも１つのタスクでロボットを動かしていることを確認することです。このことは、すでに第Ⅵ章で紹介しています。ここでプログラムを見てみましょう。

```
mutex moveMutex;

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

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

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

ここでのポイントは、「check_sensors」と「move_square」の２つのタスクは、他のタスクがこれらを利用していないときに、モータを制御することができ、この場合モータを利用する前にリリースされた相互排除の値「moveMutex」を待機する「Acquire」ステートメントを使います。この「Acquire」コマンドの対となる「Relese」コマンドは、「mutex」値をフリーにし、他のタスクでモータを制御することができるようになります。この「Acqire」と「Relese」との間を「重要（排他）領域（critical region）」と呼び、共有リソースが利用されている意味となります。この方法によって、互いのタスク間の衝突を避けることができます。

セマフォーの利用

ここで紹介している「Acquire」と「Release」コマンドに代わりのプログラミングとして「mutex」値の利用があります。

標準的な問題解決手法としては、タスク内でモータ制御状態を示す変数値を使うことです。他のタスクでモータ制御ができないように、この変数値を利用します。この変数値を「セマフォー（semaphore：シグナル）」と呼び、「sem」といったようなセマフォー（mutexと同じような）を使います。この場合の「sum」の値がゼロ（０）は、モータを利用しているタスクはないことを意味します。それでは、タスクでモータを動かしている状態を示す場合について、つぎに示してみましょう。

```
until (sem == 0);
sem = 1;    //Acquire(sem);
// Do something with the motors「モータ制御」
// critical region 「重要（排他）領域」

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

このプログラムでは、モータが使われなくなる「sem＝０」まで待機し、その後に排他制御として「sem」に「１」を設定します。この状態でモータを制御することができるようになります。処理が終わってから、「sem」を「０」に戻します。以上のプログラムで、このセマフォーを使った処理（procedure：プロシジャ）を行っています。例えば、タッチ・センサーが何かに接触したとき、セマフォーが設定され、バックアップされている処理が動き始めます。この処理を行っている間、「move_square」タスクは待機状態にあります。つまりバックアップされた状態で、セマフォーは「０」に設定されたまま「move_square」は継続されます。

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

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

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

この「move_square」内で、セマフォーを「１」にし、さらに「０」に戻すことに着目してみましょう。これは便利なもので、２つのコマンドで「OnFwd()」を使って処理をしています（第Ⅷ章参照）。他のタスクでこのコマンド処理を中断させる必要はなくなります。

セマフォーは、とても使い勝手がよく、並列処理での複雑なプログラムを作成する場合にもほとんど利用できます。（このプログラムを使って失敗させてみてください。何故そうなるかを理解してみてください）

まとめ

この章では、異なるタスクを使う場合に、問題が発生するのを学びました。その結果は、いつも注意する必要があります。期待しない挙動が多く出てきました。この問題を解決するための２つの手法を学習しました。最初の解決法が、一つのタスク内において、常に排他処理を制御するようにタスクの中断と再起動を行う方法です。つぎの解決法は、各タスク内の実施において、セマフォーを使って制御する方法です。このことで、並列処理において、いつの瞬間でも一つのタスクを実行させることが保証されます。

XI. ロボット間の通信

この章では、二つ以上のNXT を持っている場合に役立つものです。（またPCを使って１つのNXTとのデータ通信も含みます）ロボットはBluetooth（無線技術）を使って互いの通信ができ、複数のロボットで協業（もしくは戦闘）させることもでき、さらに２つのNXTを使って６つのモータと８つのセンサーを装備した複雑なロボットを構築することもできます。
古いRCXでは、「InfraRed」メッセージを送って、すべてのロボットでこれを受信することができます。
NXTでは、まったく違った通信方法となります。最初に、本体の「Bluetooth」メニューで、２つ以上のNXT（もしくはPC）と接続すれば、接続された装置にメッセージを送ることができるようになります。
最初に繋いだNXTをマスター（master：主人）と呼び、他をスレイブ（slave：奴隷）と呼び、３つのスレイブを３つの「ライン１」「ライン２」「ライン３」に接続します。スレイブ側は、「ライン０」に接続したマスターを常に見ています。メッセージとして最大１０個のメールボックスを送信することができます。

マスターとスレイブ間のメッセージ送信

２つのプログラムを見ていきましょう。一つはマスター側で、もう一つはスレイブ側となります。基本的なこれらのプログラムでは、無線ネットワーク上の２つのNXTが文字列のメッセージを高速に通信しあうことを教えています。
マスター側のプログラムでは、最初に「BluetoothStatus(conn)」機能を使い、正しくスレイブが「ライン１」に接続されているかどうかをチェックし、「M」文字を使って通信構築を行ってメッセージを送信し、増数の「SendRemoteString(conn,queue,string)」を使い、さらにメッセージがスレイブ側から「ReciveRemoteString(queue,crear,string)」を通じて受信でき、そのデータが画面に表示されるようになっています。

```
//MASTER
#define BT_CONN 1
#define INBOX 1
#define OUTBOX 5
sub BTCheck(int conn){
  if (!BluetoothStatus(conn) == NO_ERR){
    TextOut(5, LCD_LINE2, "Error");
    Wait(1000);
    Stop(true);
  }
}

task main(){
  string in, out, iStr;
  int i = 0;
  BTCheck(BT_CONN); //スレイブの接続をチェックする
  while(true){
    iStr = NumToStr(i);
    out = StrCat("M",iStr);
    TextOut(10, LCD_LINE1, "Master Test");
    TextOut(0, LCD_LINE2, "IN:");
    TextOut(0, LCD_LINE4, "OUT:");
    ReceiveRemoteString(INBOX, true, in);
    SendRemoteString(BT_CONN, OUTBOX, out);
    TextOut(10, LCD_LINE3, in);
    TextOut(10, LCD_LINE5, out);
    Wait(100);
    i++;
  }
}
```

スレイブ側も似た様なプログラムですが、「SendRemoveString」の代わりに「SendResponseString(queue,string)」を使っています。このことは、スレイブ側は「ライン０」を使ってマスター側にメッセージを送るだけとなるからです。

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

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

task main(){
  string in, out, iStr;
  int i = 0;
  BTCheck(0); //マスターの接続をチェックする
  while(true){
    iStr = NumToStr(i);
    out = StrCat("S",iStr);
    TextOut(10, LCD_LINE1, "Slave Test");
    TextOut(0, LCD_LINE2, "IN:");
    TextOut(0, LCD_LINE4, "OUT:");
    ReceiveRemoteString(INBOX, true, in);
    SendResponseString(OUTBOX, out);
    TextOut(10, LCD_LINE3, in);
    TextOut(10, LCD_LINE5, out);
    Wait(100);
    i++;
  }
}
```

このプログラムで、ひとつ問題があることに気づくでしょう。他のスレイブがメッセージを送り続けたとき、すべての送信されたメッセージが無くなっていることを知らずに、表示されている数値は増えつづけます。このことは、マスター側が他に受信できないことを意味します。この障害を避けるために、正しいプロトコルのための配信承認について学んでおく必要があります。

送信数の承認(ack)

ここで他に２つのプログラムを見てみましょう。マスターは、このとき「SendRemoteNumbaer(conn,queue,number)」を使って番号を送信し、スレイブ側の承認のためにの待機を中断し（「until」サイクルの中で「ReceiveRemoteString」を使用）、スレイブが受信中で承認を送信するだけとなり、マスターはつぎのメッセージを送信し続けます。スレイブは、単純に「ReceiveRemoteNumber(queue,clear,number)」を通じて数値を受信し、「SendResponseNumber」を通じて承認を送信します。このマスターとスレイブの両方のプログラムでは、承認のための共通コードを使って合意します。この中では、16進数の「0xFF」を使っています。

マスターは、乱数の数値を送信し、スレイブ側から正しいコードが返ってくることで承認を待ち続けますが、一方でマスターは、新しい承認なしで送信を続けます。理由は、ここでの値が変わっていくからです。

スレイブはメールボックスを常にチェックし、空でない場合にその値を読んでNXT画面に表示し、マスターに承認を返します。プログラムの最初に、メッセージを読み取ることなしにマスターに承認を送信しています。この秘策なしでは、マスターが最初に走り出し、スレイブが後でスタートした場合にハングアップ（エラー発生）します。
この方法では、最初のメッセージのいくつかは失われますが、マスターとスレイブのプログラムをハングアップのリスクなしに、異なる時間でスタートさせることができます。

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

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

task main(){
  int ack;
  int i;
  BTCheck(BT_CONN); //スレイブの接続をチェックする
  TextOut(10, LCD_LINE1, "Master Sending");
  while(true){
    i = Random(512);
    CLEARLINE(LCD_LINE3);
    NumOut(5, LCD_LINE3, i);
    ack = 0;
    SendRemoteNumber(BT_CONN, OUTBOX, i);
    until(ack == 0xFF){
      until(ReceiveRemoteNumber(INBOX, true, ack) == NO_ERR);
    }
    Wait(250);
  }
}
```

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

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

task main(){
  int in;
  BTCheck(0); //マスターの接続をチェックする
  TextOut(5, LCD_LINE1, "Slave Receiving");
  SendResponseNumber(OUT_MBOX, 0xFF); //マスターをアンブロックする
  while(true){
    if(ReceiveRemoteNumber(IN_MBOX, true, in) != STAT_MSG_EMPTY_MAILBOX){
      TextOut(10, LCD_LINE3, " ");
      NumOut(5, LCD_LINE3, in);
      SendResponseNumber(OUT_MBOX, 0xFF);
    }
    Wait(10);//オプションの待機時間
  }
}
```

直接的なコマンド

Bluetooth通信機能には他にすばらしい機能があり、マスターから直接スレイブを制御できるようになっています。

つぎの例では、マスターからスレイブへ直接コマンドを送り、音を出してモータを動かしています。この場合には、スレイブ側のプログラムは不要となり、スレイブ側のNXTで受信してメッセージを管理するファームウェアを通して実施します。

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

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

task main(){
  BTCheck(BT_CONN); //スレイブの接続をチェックする
  RemotePlayTone(BT_CONN, 4000, 100);
  until( BluetoothStatus(BT_CONN) == NO_ERR);
  Wait(110);
  RemotePlaySoundFile(BT_CONN, "! Click.rso", false);
  until( BluetoothStatus(BT_CONN) == NO_ERR);
  // Wait(500);
  RemoteResetMotor Position(BT_CONN, OUT_A, true);
  until( BluetoothStatus(BT_CONN) == NO_ERR);
  MOTOR(OUT_A, 100);
  Wait(1000);
  MOTOR(OUT_A, 0);
}
```

まとめ

この章では、２つ以上のロボット間でBluetooth通信を使って、文字列や数値の送信受信や、通信承認の待機などの接続について学びました。最後の事項は、安全な通信プロトコルが必要なときに重要なものとなります。
その他に直接コマンドを使ってスレイブを動かすことも学びました。

XII. その他の命令

NXC は、以上のほかにも多くのコマンドを用意しています。この章では３つのタイプのコマンド、タイマーの使い方、画面表示のコマンド、それとNXTファイルシステムの使用法について紹介していきます。

タイマーの使い方

NXTでは、実行を継続させるタイマーがあります。このタイマーは、1000分の1秒（ミリ秒）毎の刻みとなります。現時点の時刻は、「CurrentTick()」を使って値を取り出します。以下にタイマーを使った事例を紹介します。つぎのプログラムでは、１０秒間でのランダムな数値でロボットを動かしています。

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

ここでのプログラムは、既に第IV 章で紹介したプログラムと比較してみたくなるでしょう。このタイマーは、正確で簡単です。

「Wait()」コマンドをタイマーに置き換えることは非常に便利です。タイマーをリセットし、つぎの特別な値になるまで待機するように正確な総合時間で休止させることができます。しかも、この待機中に、別なイベント（例えば、センサーなど）で反応せることもできます。つぎの簡単なプログラムは、この事例となります。ここでは、ロボットを１０秒経過するまでか、タッチ・センサーが反応するまで動かします。

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

タイマーの単位は、「Wait」コマンドと同様に、1000分の１（1/1000） 秒（ミリ秒）であることに注意してください。

ドットマトリックス・ディスプレイ（NXT画面表示）

NXT はドット・マトリクス表示機能を持っていて、横100ピクセル、縦64ピクセルのモノクロのディスプレイとなります。ここではテキスト文字、数字、ドット、線分、長方形、円、それにビットマップイメージ(「.ric」ファイル)を表示させる多くのAPI機能があります。つぎの例では、これらすべてを含んだプログラムとなります。ここでのピクセル座標値（０，０）は、ディスプレイ上の右上を意味します。

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

task main(){
  int i = 1234;
  TextOut(15, LCD_LINE1, "Display", true);
  NumOut(60, LCD_LINE1, i);
  PointOut(1, Y_MAX-1);
  PointOut(X_MAX-1, Y_MAX-1);
  PointOut(1,1);
  PointOut(X_MAX-1, 1);
  Wait(200);
  RectOut(5, 5, 90, 50);
  Wait(200);
  LineOut(5, 5, 95, 55);
  Wait(200);
  LineOut(5, 55, 95, 5);
  Wait(200);
  CircleOut(X_MID, Y_MID-2,20);
  Wait(800);
  ClearScreen();
  GraphicOut(30, 10, "Smile 01.ric");
  Wait(500);
  ClearScreen();
  GraphicOut(30, 10, "Smile 02.ric");
  Wait(1000);
}
```

すべてのこれらの機能は、わかり易いものですが、パラメータの詳細を記載しておきます。

「ClearScreen()」は、ディスプレイ表示をクリアします。
「NumOut(x,y,number)」は、座標値（ｘ，ｙ）に、数字（number）を表示します。
「TextOut(x,y,string)」は、座標値（ｘ，ｙ）に、文字列（string）を表示します。
「GraphicOut(x,y,filename)」は、座標値（ｘ，ｙ）に 、ビットマップ(「.ric」)ファイルを表示します。
「CircleOut(x,y,radius)」は、座標値（ｘ，ｙ）を中心とした半径（radius）で円を描きます。
「LineOut(x1,y1,x2,y2)」は、座標点(ｘ１，ｙ１)から（ｘ２，ｙ２）までの線分を描きます。
「PointOut(x,y)」は、座標点（ｘ，ｙ）にドットを描画します。
「RectOut(x,y,width,height)」は、座標点（ｘ、ｙ）から右幅（width）と高さ（height）の長方形を描きます。
「ResetScreen()」は、スクリーンをリセットします。

ファイルシステム

NXT は、内臓されたフラッシュメモリを使って、ファイルの書き込みや読み込みができます。このことで、センサーデータによるログをとったり、ロボットを動かすプログラムを保存したりすることができます。ファイルの制限は、このフラッシュメモリの容量に依存します。NXTのAPI機能では、ファイル管理（作成、名前変更、削除、検索）、それに文字列や、数値およびシングルバイトでのファイルの読み込みと書き出しができます。

つぎの例では、ファイルをどう作成し、文字列を書き込み、名前を変更するところを紹介しています。

最初、プログラムはこれから使おうとするファイル名で、すでに存在するものを削除しています（本来であれば、既存ファイルの存在をチェックし、マニュアル操作で削除するか、自動的に他の名前に変更することがよいでしょう）。ただこの場合は単純にしていますが、特に問題は起きないでしょう。つぎに「Createfile("Danny.txt", 512, fileHndle)」を使って、ファイルを作成し、名前を付け、サイズ、およびファイルのハンドル（NXTファームウェア内で独自に管理する数字）が設定されます。

つぎに「WriteLnString(fileHandle,string,bytesWritten)」を使って、ファイルに文字列を入れ、さらにリターン値を付加して書き込みします。ここでのパラメータは、すべて変数名となります。最後に、ファイルを閉じ（close：クローズ）し、名前を変更（rename：リネーム）します。注意：つぎのファイル操作(ファイルへの書き込み、読み込み)をするまでファイルをクローズし、そして読み込みときに「OpenFileRead()」を使ってオープンします。また、ファイルを削除するか、名前変更する場合には、ファイルをクローズしておく必要があります。

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

結果はどうなっているでしょうか。それでは「BricxCC」から「tools」を選択し、「NXT Explorer」へ行き、ファイル名「Dannaysay.txt」を、PCへアップロード（upload）して、中を覗いて見ましょう。
つぎの例をみてください。ASCII(アスキー)文字を一覧表にして作成しています。

```
task main(){
  byte handle;
  if(CreateFile("ASCII.txt", 2048, handle) == NO_ERR) {
    for (int i=0; i < 256; i++) {
      string s = NumToStr(i);
      int slen = StrLen(s);
      WriteBytes(handle, s, slen);
      WriteLn(handle, i);
    }
    CloseFile(handle);
  }
}
```

これも単純なプログラムで、ファイルを作成し、エラーが無ければ、つぎに「NumToStr(i)」関数を使って、0から255までのコードによる文字変換を行い、「WriteBytes(handle,s,slen)」関数でこの文字に改行（キャリッジリターン）を追加してファイルに書き込んでいます。
この結果は、「ASCII.TXT」をオープンする際、テキスト・エディタ（例えばWindowsのNotepad）を使って見ます。この数値で書き込まれた内容は、16進数で書かれた数値も、アスキー（ASCII）コードに変換し、人にわかり易い文字列で見ることができます。

ここで２つの重要な関数で、ファイルから文字列を読み込む「ReadLnString」関数と、数字を読み込む「ReadLn」関数を紹介します。

ここでの事例での最初の行では、「main」タスクで、乱数から文字列にしてファイルを作成する「CreateRandomFile」サブルーチンを呼び出しています。またこの行をコメントにしたり、手動作成によるテキストファイルを使ったりすることができます。

つぎに「main」タスクでこのファイルを読み込むためにオープンし、ファイルのエンドになるまで「ReadLnString」関数を使って読み続け、テキストを表示します。

「CreateRandomFile」サブルーチンは、乱数で発生させ、あらかじめ文字に変換して、それをファイルに書き込みます。

「ReadLnString」関数は、ファイル・ハンドルと文字列の変数を引数として、呼び出されたときに文字列をテキストラインに書き込み、リターン値としてエラーコードを返します。ファイルの読み込みが最終行になった場合には、このエラーコードで分るようになっています

```
#define FILE_LINES 10
sub CreateRandomFile(string fname, int lines){
  byte handle;
  string s;
  int bytesWritten;
  DeleteFile(fname);
  int fsize = lines*5; //create file with random data
  if(CreateFile(fname, fsize, handle) == NO_ERR) {
    int n;
    repeat(FILE_LINES) {
      int n = Random(0xFF);
      s = NumToStr(n);
      WriteLnString(handle,s,bytesWritten);
    }
  CloseFile(handle);
  }
}

task main(){
  byte handle;
  int fsize;
  string buf;
  bool eof = false;
  CreateRandomFile("rand.txt", FILE_LINES);
  if(OpenFileRead("rand.txt", fsize, handle) == NO_ERR) {
    TextOut(10, LCD_LINE2, "Filesize:");
    NumOut(65, LCD_LINE2, fsize);
    Wait(600);
    until(eof == true){ // read the text file till the end
      if(ReadLnString(handle,buf) != NO_ERR) eof = true;
      ClearScreen();
      TextOut(20, LCD_LINE3, buf);
      Wait(500);
    }
  }
  CloseFile(handle);
}
```

最後のプログラムでは、ファイルから数値を読み込む方法を学習します。
少し状況が複雑な事例をご紹介しましょう。最初のコードで、INT（整数）が設定されていますが、マクロなどと重複しないようにしてください。

これらは、プリプロセッサ・ステートメントです。

```
#ifdef INT
  ....Code....
#endif
```

これによって、あらかじめ宣言した「#define INT」で、以下の２つのステートメント（「＃ifdef」と「#endif」）には挟まれたコードがコンパイルされます。もし、「#define LONG」で宣言した場合には、「INT」のコードではなく、下の「main」タスクのある「LONG」のコードがコンパイルされます。

この手法は、同じ関数「ReadLn（handle、val）」を使って、ファイル読み込み時に、「int」（16ビット）と「long」（32ビット）の両方のうち一つのプログラムをコンパイルするようにします。
ここでの２つの引数は、前述したファイル・ハンドルと数値変数で、リターン値はエラーコードとなります。

この関数では、「int」で宣言された場合には、2バイトごとにファイルから読み込みます。また「long」で宣言された場合には、4バイトごとにファイルから読み込みます。また「bool」（ブーリアン）値を使って同様に、読み込み、書き出しができます。

```
#define INT // INT 又はLONG
#ifdef INT

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

#endif
#ifdef LONG

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

まとめ

この章ではNXTが提供する拡張機能である、高性能のタイマー、描画表示の画面、およびファイルシステムについて紹介してきました。

XIII. あとがき

このチュートリアルを学んだことで、NXCの上級者になれたと思います。まだ途中であれば、独習して上達してください。さらに独創的な設計やプログラミングで、LEGOロボットを信じがたいものへと造りあげることができます。

このチュートリアルは、「BricxCC」のすべてを網羅している訳ではありません。これらなどの紹介を行っている「NXCガイド」を参考にされることをお勧めします。もちろん、NXCは、まだ開発途中段階であり、将来バージョンでは、追加機能が組み込まれていきます。多くのプログラミングの概念が、このチュートリアルで述べられている訳でもありません。特に、ロボットの動きや人工知能について学ぶものではありません。

もちろんPCから直接LEGOロボットを制御することができます。ここでの要求は、C++、Visual Basic,JAVA、さらにDelphiなどの言語を通じて、プログラミングすることができます。またNXT自身の中で、NXCプログラムを通じて、互いのプログラムを実行させることもできます。これらの組み合わせは強力なものです。ロボットのプログラミングにおいてこれらに興味を持たれる方は、レゴ マインドストーム webサイトの NXTream から、Fandom SDKおよびオープンソースのドキュメントをダウンロードされることをお勧めします。

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

このWebでは、追加情報としてより多くのものが提供されています。ほかの重要な情報として、非公式のLEGOユーザ・グループ・ネットワークの「LUGNET」をご紹介しておきます。

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

<つぎは、BricxCCについての解説本を手掛ける予定でいます。こちらは、NXCのプログラム開発環境において、より使いやすく、早いコーディング作成、NXTとの接続環境、ファイル管理など、NXTを使いこなす上での統合開発環境を提供しています。乞うご期待:高本 2011/1/5>