NQCを使ったLEGOロボットのプログラミング

（英語版3.03、1999年10月2日）

著者　Mark　Overmars

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

アルベルト・パラシオス・パウロブスキ
訳　Alberto　Palacios　Pawlovsky

（日本語版2.0、1999年12月17日）

桐蔭横浜大学工学部電子情報工学科
〒225-8502神奈川県横浜市青葉区鉄町1614番地

Programming Lego Robots using NQC
by Prof. Mark Overmars.
Department of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.

Japanese translation done with the permission of Prof. Overmars
by Dr. Alberto Palacios Pawlovsky (Lecturer)
Department of Electronics and Information Engineering.
Toin University of Yokohama.
225-8502 Yokohama-shi, Aoba-ku, Kurogane-cho 1624. Japan.

December 17, 1999.

まえがき

LEGO社のMindStormsおよびCyberMasterのロボット・キットはすばらしい「おもちゃ」であり、多種多様の仕事を行ういろいろなロボットの作成やプログラムを可能にするものである。キットに付いているプログラミング環境は、ビシュアルで初心者向きであるが、その機能が制限されている。そのため、簡単な仕事にしか使用できない。ロボットのもつ機能をフルに使用するために、異なる環境が必要になる。NQCは、前記のロボット専用のものであり、Dave Baum氏に開発された言語である。プログラムの経験がなくても心配がいらない。このチュートリアルはNQCでのプログラミングを簡単に説明している。実は、LEGOのロボットのプログラミングは計算機のプログラミングに比べれば非常に簡単なものである。このため、容易にプログラミングになじむことができる。

プログラミングをより簡単にするためにRCX Command Center（以下RCXコマンド・センターと呼ぶ）というソフトウェアがある。このユーティリティではプログラムの作成、ロボットへの転送、ロボットのスタートおよびストップができる。RCXコマンド・センター（3.0以上の版）はワードプロセッサのように使用することができる。このチュートリアルはRCXコマンド・センターの使用を前提としている。RCXコマンド・センターを以下のWebページからダウンロードすることができる。

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

RCXコマンド・センターは、Windows95、98、およびNTが動作しているコンピュータで使用することができる。このユーティリティを使用する前に、RCXを少なくとも一回LEGOのプログラミング環境で使用して頂きたい。LEGOのプログラミング環境での使用によりRCXコマンド・センターに必要なものがRCXに転送されるためである。NQCはその他のプラットフォーム（OS）でも使用可能になっている。NQCを以下のWebページからダウンロードすることができる。

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

このチュートリアルの内容はその他のプラットフォームでも使用できる（ただし、NQC 2.0以上の版が必要）。しかし、RCXコマンド・センターで使用するカラフルな環境およびいくつかのツールを使用することができない。

このチュートリアルは、読者がLEGO社のRIS(Robotics Invention System）をもっていることが前提とされている。内容の大部分はLEGO社のCyberMastersロボットにも適用できるが、いくつかの機能は利用できないことがある。また、モータの名前も異なるから、例題のプログラムを使用可能とするために、多少の変更が必要である。

謝辞

NQCの開発者であるDave Baum氏に感謝し、また、このチュートリアルの最初の部分の初版をお書きいただいたKevin Saddi氏に感謝する。

訳のまえがき

この翻訳版は、NQCバージョン2.x以上の版用である。初版のため、今後NQCの版やRCXコマンド・センターの版に合わせながら改善をはかっていきたいと考えております。いくつかおかしい点や誤字や脱字も見つかっていますが、気の付いたところがありましたら、どうぞご意見、ご感想をお願いいたします。

この場を借りて、作成に際し、適切な御教示および御意見を賜りました（株）アスキーのLinuxマガジンの赤嶋映子様に厚く御礼申し上げます。

桐蔭横浜大学工学部電子情報工学科
アルベルト・パラシオス・パウロブスキ
palacios@cc.toin.ac.jp

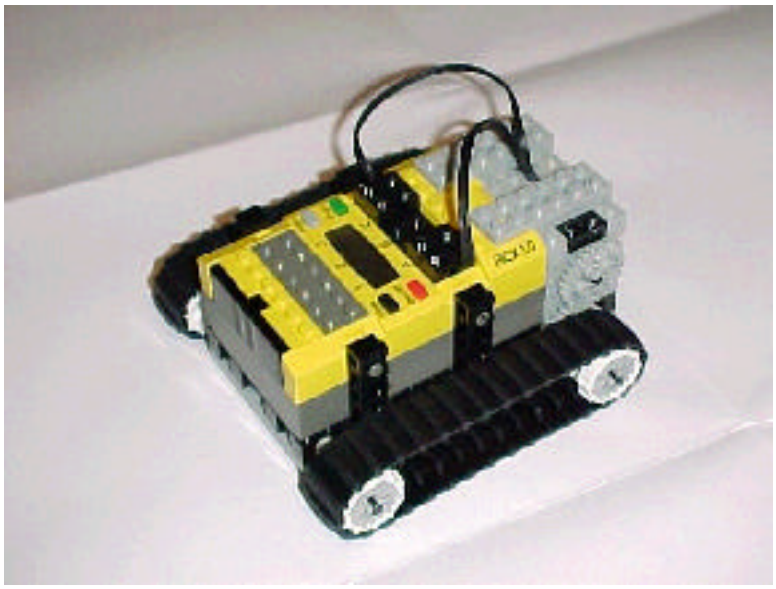
目次

Ｉ　最初のプログラムの作成

この章では非常に初歩的なプログラムの作成について触れていく。このプログラムでは、ロボットが４秒間前進してから、４秒間後退して止まる。複雑な動作ではないが、プログラミングの基本的なアイデアへの入門になる。また、プログラミングが簡単であることも示すことになる。始める前に対象になるロボットを作成しておこう。

ロボットの組立

このチュートリアルで使用するロボットは、RIS（Robotics Invention System）キットのA4サイズ・マニュアルのページ39～46に述べられている"Top Secret Plans"ロボットの簡略版である。そのロボットの車体に当たる部分と駆動機構のみを使用する。フロントの部分とアームは使用しない。また、モータへの接続線をRCX（RISの中心部である黄色のプログラマブル・ブリック）の横から出るように向きを変えよう。この変更はロボットを正確に動かすために重要である。組み立てられたロボットは以下のようになる。

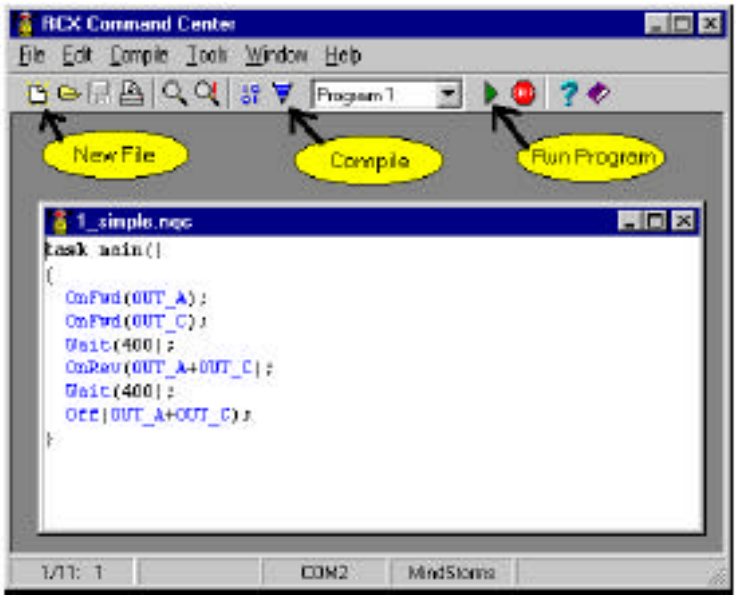


さらに、パソコンに繋ぐIRタワーの転送範囲は広範囲（IRタワーのフロントの下部にあるスイッチを大きい方の三角記号）に設定していることが必要である（できればRISのプログラミング環境で動作を確認してもらいたい）。

RCXコマンド・センターの起動

このチュートリアルでは、RCXコマンド・センターの環境でRCXをプログラミングする。このソフトウェアを起動するために、ソフトウェアのアイコン（RcxCC）をダブル・クリックする（既にこのソフトウェアが前書きに記されているサイトからダウンロードされ、インストールされているものと仮定している）。プログラムを起動してからロボットの位置を尋ねてくるので、ロボットをオンにしてO.K.のボタンをクリッ

クする。ほとんどの場合はプログラムがロボットを自動的に認識する。起動したプログラムの雰囲気は以下のようになる。ただし、真ん中に写っているテキスト・ウインドウは表示されない。



プログラムのインタフェースは通常のワードプロセッサのようなものであり、メニューやボタンはワープロにあるものと同様な役割をもち、ファイルを開いたり、セーブしたりすることなどができる。このほか、コンパイル（プログラムをRCXが実行できる形に変換すること）、RCXへコンパイルしたプログラムをダウンロード（転送）すること、もしくはRCXからいろいろな情報を得るための特殊なアイコンもある。この節ではそれらについて触れない。次に、一つのプログラムを作成する。このため、角の折ってある白紙のような形のアイコン（New File）をクリックして、新しいウインドウを開く。

プログラムの作成

次のプログラムをタイプしておこう。

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

最初のプログラムなので複雑に見えるが、解説してみよう。NQCで書かれたプログラムはタスク（task）で構成されている。上記のプログラムは一つのmain()というタスクで構成されている。RCXのプログラムには少なくとも一つのmain()という名のタスクがなければならない。タスクについては第IV章で触れる。タスクは、命令で構成される。命令のことを文（ステートメント）と呼ぶ。文の集まりがタスクに属し

ていることを明確にするために、それらをカッコ（｛　と　｝）で囲む。文の終わりには、セミコロン（；）を付ける。これによって、どこで文が終わり、どこから次の文が始まるのがわかる。すなわち、一般的にはタスクは次の構成になる。

```
task main()
{
 文1;
 文2;
 ...
}
```

前記の最初のプログラムは六つの文で構成されている。その一つずつについて見てみよう。

```
OnFwd(OUT_A);
```

この文はRCXに出力ポートA（OUT_A）のモータを起動させるための命令である。この命令でRCXのポートAのモータが前進（**On + Forward**）する。速度（スピード）を別途で指定しない限り、モータが最大の速度で回転する。スピードの設定について後で触れる。

```
OnFwd(OUT_C);
```

前記の命令と同様であるが、動かされるモータはCである。この二つの命令で両方のモータが動かされ、ロボットが前進する。

```
Wait(400);
```

待機（**Wait**）する（現状態保持の）ための命令。この命令で４秒間現在の状況（前進）が保たれる。カッコ内の数字はアーギュメント（引数）であり、400はティック数（時間の単位）を指定する。１ティックは1/100秒である。この単位でプログラムに待機時間をきめ細かく指定することができる。この命令では、ロボットが４秒間前進を続けることになる。

```
OnRev(OUT_A+OUT_C);
```

ロボットが方向を変えて動かすのに十分な距離を走ったところで、後退（**On + Rev**erse）させることにする。この命令のOUT_A+OUT_Cのアーギュメント（引数）で、両方のモータの回転方向を同時に変えている。同様なアーギュメントで最初の二つの命令を一つにすることができる。

```
Wait(400);
```

この命令で（RCXが）４秒間現在の状況（後退）を保ち続ける。

```
Off(OUT_A+OUT_C);
```

最後の命令で両方のモータを停止する（オフにする）。

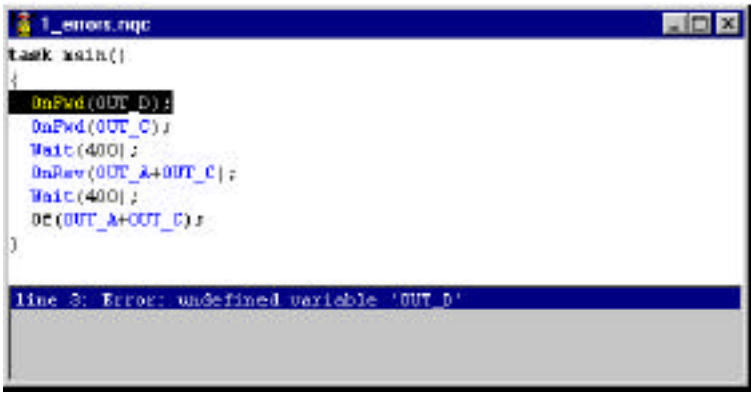
以上のプログラムの全命令で、ロボットの両方のモータが４秒間前進した後４秒間後退して止まる。プログラムをタイプするとき、色の変化に気づいただろう。色は自動的に変化する。青色の部分は命令やモータなどのRCXが意識できるものである。taskのような太字の語はNQCの予約語（ユーザが自由に使用できない語）を示す。その他にも予約語があり、後述のプログラムではそれらも太字で表される。この色づけによりタイプ誤りの発見が容易になる。

プログラムの実行

プログラムを記入した後、コンパイルおよび赤外線の送受信機（PCのシリアルポートにつなぐ付属品、以下IRタワーと呼ぶ）によるRCXへの転送（ダウンロード）が必要となる。このために逆三角（砂時計のようなアイコン）のボタン（前記の図でCompileの説明がついている）がある。このボタンを押すと、（誤りがなければ）プログラムがコンパイルされ、RCXに転送される。誤りがあるときは、その警告が表示される（以下の節を参照されたい）。ダウンロード後、プログラムを実行することができる。このためには、RCXの緑色のボタンを押すか、RCXコマンド・センターの右向き三角形のボタン（前記の図でRun Programの説明がついている）を押すかの二方法がある。ロボットが説明された通り動いただろうか。そうでない場合は、接続をチェックしよう。

プログラム内の誤り

プログラムをタイプするときには、ミスをしやすい。コンパイラはそのとき、RCXコマンド・センターの画面の下に、次の図に示すように誤りの位置などを報告する。



この例では３番目の行のFwdのdが抜けているため、エラーがその行にあると表示されている。しかも、このエラーが最初のものであるため、自動的に黒字で強調され、表示（ハイライト）される。多数のエラーの中から、一つについて調べたいとき、エラー・メッセージをクリックすると誤り位置の行が強調され、挿

入位置がそこに変わる。通常、先頭のエラーがプログラムのその他の部分に影響を与えるため、最初のエラーを訂正した後、再びコンパイルを試みよ。ほとんどの場合は、その他のエラーもこの操作で消える。前述の色つきのプログラミング環境によって、かなりの誤りを妨げる。上記の例の２番目のエラーは、最後の行のOfという命令（文）がないため、その部分が青色になっていない（Offが正しい）。

もちろん、コンパイラが発見できない誤りもある。例えば、前記の誤りがOUT_Bの場合は、このモータが指定可能なため発見されない（使用されていなくても）。このため、ロボットが予想した動きをしない場合は、プログラムに誤りがあると考えよう。

速度の変更

前記プログラム例の実行で、モータの回転はかなり速いことに気が付いただろう。速度を指定しない限りロボットが最大の速度で動く。その速度を変更するためにSetPower()という命令が使える。回転力（Power）の範囲は０から７までである。最大の回転力は７であり、０は最小値である（この値でもモータが動くが遅い）。次に示すプログラム例の最新版ではロボットがゆっくり動く。

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

まとめ

この章ではRCXコマンド・センターで最初のプログラムを書いた。これでプログラムがタイプ（作成）できるようになった。また、プログラムのコンパイル、ダウンロードおよび実行も学習した。RCXコマンド・センターでできることは多いので、これらについては、その説明書を参照されたい。このチュートリアルではNQCを中心にして説明が行われる。必要に応じてRCXコマンド・センターの機能について触れることにする。

ここでもNQCのいくつかの重要な特徴について学んだ。まず、プログラムには必ずmain()というRCXで実行されるタスクが必要であることが分かった。また、モータの制御用の最も重要な命令OnFwd()、OnRev()、SetPower()およびOff()の使用方法をみてきた。さらに、Wait()という命令の意味および使用方法について学習した。

Ⅱ　もう少し面白いプログラムを作ろう

最初のプログラムは簡単だった。そこでそのプログラムをもう少し面白いものにしよう。以降のいくつかのステップで、少しずつNQCのプログラミングの特徴を紹介していく。

曲がる

ロボットの片方のモータを止めたり、その回転方向を変えたりすることによって、ロボットに右折あるいは左折をさせることができる。ここでその例を示す。以下のプログラムをタイプしてロボットにダウンロードせよ。このプログラムでは、ロボットが少し前進した後９０度で右折するはずである。

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

場合によっては、９０度の右折をするために、上記プログラムの最後のWaitの値は85以外にしなければならないかもしれない。この値は床の素材に依存している。この定数を変えるのではなく、定数をプログラムの先頭に定義しておき、その定義を変えるのが普通である。NQCでは、以下のプログラムに示すように定数をプログラムの先頭に定義することができる。

```
#define MOVE_TIME 100
#define TURN_TIME 85

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

最初の２行には定数が二つ定義されている。定義された定数は、プログラムの中に使用することができる。このような定数の定義には次の二つの利点がある。まず、第一はプログラムが見やすくなる。第二は定数の変更が容易になる。RCXコマンド・センターではこの定義が色で表示される。後述の第IV章でその他の定義について述べる。

命令の繰り返し

次にロボットを正方形の経路で走らせよう。正方形を描くためにロボットが前進し、左折、前進、左折の繰り返しで動かなければいけない。このために上記のプログラムを４回コピーすればよいが、それより簡単な繰り返し専用の**repeat**文を以下のように使用する。

```
#define MOVE_TIME 100
#define TURN_TIME 85

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

repeatのカッコの中には繰り返しの数を指定する。繰り返される文をタスクの文と同様に｛　と　｝で囲む。上記のプログラムでは文を字下げして見やすくしてあるが、こうする必要はない。この章の最後の例として、ロボットに１０回正方形を描くように走らせよう。そのプログラムは次のようになる。

```
#define MOVE_TIME 100
#define TURN_TIME 85

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

このプログラムにはrepeatの中にrepeatの文がある。このような記述を入れ子（nested）のrepeat文と呼ぶ。このような記述をいくらでも使用できるが、各repeat文の始まりの｛と終わりの｝を明確にしよう。上記のプログラムでは２番目の｛と５番目の｝が一つのrepeatを、３番目の｛と４番目の｝がもう一つのrepeat文を指している。repeatの｛と｝が対で使用される。

コメントの追加

プログラムを分かりやすくするためには、コメントの追加がよい方法である。二つの // を記入すると、その右側をコメント欄として使用できる。いくつかのラインにわたるコメントを /* と */ で記入することができる。RCXコマンド・センターではコメントが緑色で表示される。

```
/* 10 SQUARES
   by Mark Overmars
This program makes the robot run 10 squares
*/

#define MOVE_TIME 100    // Time for a straight move
#define TURN_TIME 85     // Time for turning 90 degrees

task main()
{
 repeat(10)               // Make 10 squares
 {
  repeat(4)
  {
   OnFwd(OUT_A+OUT_C);
   Wait(MOVE_TIME);
   OnRev(OUT_C);
   Wait(TURN_TIME);
  }
 }
 Off(OUT_A+OUT_C);       // Now turn the motors OFF
}
```

まとめ

この章ではrepeat文の使用とコメントの記入について学習した。入れ子の概念にも触れて { と } を対で使用することもみてきた。ここまでの知識でロボットをいろいろな経路で動かすことができる。次の章に進む前に上記のプログラムを変更していろいろな動きを試みよう。

Ⅲ　変数を使う

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ある。変数はメモリの場所の指定に使い、値を格納するために用いられる。変数に格納された値を変えたり、プログラムのいろいろな場所に使用したりすることができる。一つの例を通して変数の使用について説明しよう。

旋回しながら動く

前述のプログラムを変更してロボットを螺旋状に進ませよう。これは、プログラムの前進開始後に実行されるスリープ命令の時間を少しずつ大きくすれば実現できる。つまり、MOVE_TIMEの値を一周ごとに大きくすればよい。しかし、これをどうやって実現するのか。MOVE_TIMEは定数であるためその値を変更することができない。この定数の代わりに変数が必要となる。NQCでは変数が容易に定義できる。32個の変数まで定義でき、各々の変数には個別の名前を与えることができる。螺旋状を描きながら動くロボット用のプログラムは次のようになる。

```
#define TURN_TIME 85

int move_time;               // define a variable

task main()
{
 move_time = 20;             // set the initial value
 repeat(50)
 {
  OnFwd(OUT_A+OUT_C);
  Wait(move_time);
  OnRev(OUT_C);
  Wait(TURN_TIME);
  move_time += 5;           // increase the variable
 }
 Off(OUT_A+OUT_C);
}
```

新しく導入された命令の行にはコメントが付加されている。タスクの前のint move_time;で、整数の値を保存できるmove_timeという変数が宣言されている。一般的に、intの後に変数名を書くことで変数を宣言することができる。定数の名前には大文字を使用したため、変数名には小文字を使用する。こうする必要はないが、プログラムの中の文字列が定数か変数かを区別するには便利である。タスク、変数および定数の名前は英字（文字）で始まらなければいけない。しかし、その後には数字および下線（_）の記号も使用できる。これ以外の記号は使用できない。変数の宣言のintは英語のintegerの先頭の文字をとったものであり、整数型の値が格納できることを宣言する。

タスクの最初の命令では変数が20という値に初期化されている。この命令以降、move_timeは20の値として使われる。繰り返しのループの中にはこの値が用いられ、一回目の繰り返し後にこの値に5が加えられ、変数の値が更新される。このように変数の値は繰り返しごとに25、30、...という値をとっていく。

増加だけでなく、変数には *= で値を掛けたり、-= で値を引いたり、/= である値で割ったりすることができる（割り算で、結果が丸められることに注意しよう）。さらに、変数に変数（の値）を加えたり、より複雑な式を書くことができる。このような例を次のプログラムに示す。

```
int aaa;
int bbb, ccc;

task main()
{
 aaa = 10;
 bbb = 20*5;
 ccc = bbb;
 ccc /= aaa;
 ccc -= 5;
 aaa = 10 * (ccc+3); // aaa is now equal to 80
}
```

変数の定義は二つの行で行われ、しかも、複数の変数を２行目に定義しているが、全変数を一行に記述することができる。

乱数

前述のほとんどのプログラムでは、ロボットの行動が明確になっている。しかし、予想外の動きをすれば、面白いことになるだろう。ランダムな動きが欲しくなる。NQCではランダムに数を発生させることができる。次のプログラムはこのことを利用して、ロボットをランダムに動かす。ロボットがランダムな時間前進し、その後、ランダムな方向に曲がる。

```
int move_time, turn_time;

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

このプログラムに二つの変数が定義されている。これらの変数の値がランダムに決定される。プログラムのRandom(60)は乱数を発生させるために使用され、0〜60という範囲内（0および60を含めて）の値がランダムに発生される。whileの繰り返しごとに変数に代入される値が異なる。ここでは、Wait(Random(60))で変数の使用を避けることができる。換言すると、Wait(Random(60))のように直接Random(60)

を引数として利用できる。

このプログラムはループの繰り返しに、今まで使用したrepeatではなく、**while(true)**を使用している。whileのカッコ内（繰り返しの条件）が成り立つ間、whileの文（｛と｝で囲まれている複合文）が繰り返し実行される。この例では、条件は**true**（真）という意味の文字列が使用されているため、そのwhile文が永遠に繰り返される。trueという文字列はNQCでは予約語であり、「真」と見なされている。while文については第IV章を参照されたい。

まとめ

この章では変数の使用について学習した。変数は非常に便利だが、ロボットの制限を受け継いでいろいろな制約がある。32個の変数しか定義できないし、整数の値にのみ使用できる。しかし、これでもいろいろな作業に十分に使用できる。この章で乱数の生成についても学んだ。乱数を使用して、ロボットに予想できない動きをさせることができた。章の最後に無限のwhileループについても触れた。

IV　制御文

前章までrepeat文およびwhile文についてみてきた。これらの文で他の文の実行順を制御することができる。このような文を制御文と呼ぶ。この章ではその他の制御文について学習する。

if文

プログラムの一部分をある条件が成り立つときのみ実行したいときがある。このような場合、ifという制御文が使用できる。ここまで使用したプログラムを多少異なった視点から変更してみる。ここでは、ロボットに直線上を走らせてから、右折か左折かさせることにする。このために再び乱数を用いるが、その範囲を0～1にする。この範囲を選ぶと、発生される数字は0か1のいずれかになるため、0で右折、1で左折するようにする。そのプログラムは次のようになる。

```
#define MOVE_TIME 100
#define TURN_TIME 85

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

if文はwhile文に似ているが繰り返しがなく、ifに続くカッコ（）内の条件が真であれば、条件の後の｛と｝で囲まれている文が実行され、ifの条件が成立しない場合は、**else**の後の｛と｝で囲まれている文が実行される。ifの条件は、Random(1)で生成された数が0に一致したとき真となり、ifに続く複合文（｛と｝の間の文）が実行される。比較には変数への代入に用いる=記号と区別するために、二つ、つまり　==　の記号が使用される。比較には、次の記号も使用することができる。

＝＝　等しい　　例：　A==B　　AがBに等しい？

＜　　小さい　　例：　A＜B　　AがBより小さい？

＜＝　小さいか等しい　　例：　A＜＝B　AがBより小さい、あるいは、等しい？

＞　　大きい　　例：　A＞B　　AがBより大きい？

＞＝　大きいか等しい　　例：　A＞＝B　AがBより大きい、あるいは、等しい？

！＝　異なる　　例：　A！=B　　AがBと異なる？

上記の基本条件を&&（かつ）もしくは｜｜（または）で組み合わせることができる。以下にはそのいくつかの例を示す。

true　　いつでも真

false　　いつでも偽

ttt　!=　3　　tttが3と異なる場合、（この条件が）真となる

(ttt >= 5) && (ttt <= 10)　　tttが5以上、かつ、10以下の場合、真となる

(aaa == 10) ｜｜ (bbb == 10)　　aaaが10、あるいは、bbbbが10のとき、真となる

上記のプログラムに使用したif文は、ifの条件の後の文とelseの後の文の二つの部分で構成され、ifの後の文は条件が成立する（真の）とき実行され、条件が成立しない（偽の）ときelseの後の部分が実行される。実は、elseとその後にある｛と｝で囲まれている文はなくてもよい。つまり、条件が成り立たないときに実行する文がない場合、elseを省略することができる。

do文

制御文の中には、do文の制御用の文がある。この文の書式は以下のようになる。

```
do
{
 文
}
while(条件)
```

doの後の｛と｝で囲まれている文は、少なくとも一回実行され、その繰り返しは条件の成立に依存する。条件が成り立っている限り｛と｝で囲まれる文が実行される。この文を使用する例を次に示す。このプログラムでは、ロボットが20秒間ランダムに動いてから止まる。

```
int move_time, turn_time, total_time;

task main()
{
 total_time = 0;
 do
 {
  move_time = Random(100);
  turn_time = Random(100);
  OnFwd(OUT_A+OUT_C);
  Wait(move_time);
  OnRev(OUT_C);
  Wait(turn_time);
  total_time += move_time; total_time += turn_time;
 }
 while (total_time < 2000);
 Off(OUT_A+OUT_C);
}
```

この例では一行に二つの文が記述されている。この書き方は許されている。一行にいくつかの文を書いてもかまわないが、文と文の間には　；　が必須である。プログラムを見やすくすることを考えると、一行に一つ以上の文を書くのはできるだけ避けたい。

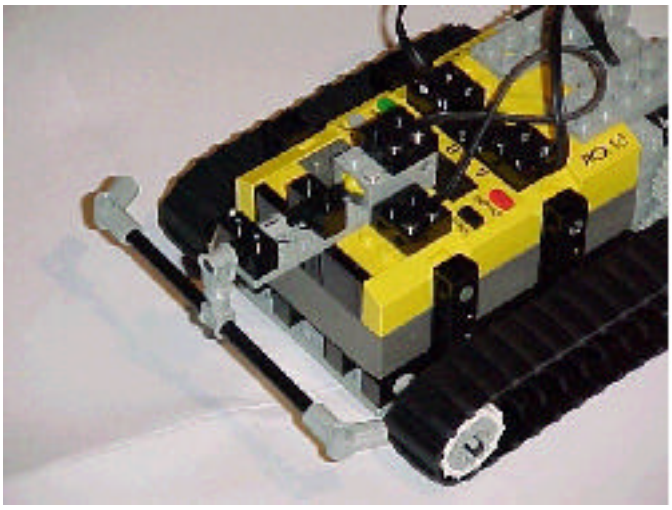
do文はwhile文に似ているが、whileでは先に条件が評価されるため、条件が成り立たなければ繰り返しの文が実行されないのに対して、doでは条件が後に評価されるため、繰り返しの文が少なくとも一回は実行される。

まとめ

この章では、if文およびdo文の新たな二つの制御用の文について学習した。この二つの文および前に述べたrepeat文とwhile文で、プログラムの命令（文）の実行を制御することができる。これらの文の使用や使い分けなどになれるために、ここまでに示したプログラムを土台にしていろいろなプログラムで練習してもらいたい。文の記述については、一行に二つ以上記述ができることを学んだ。

Ⅴ　センサー

LEGOのロボットの大きな特徴の一つはセンサーが使用できることである。しかも、センサーでロボットの制御ができる。これについて述べる前に、次の図に示すようにロボットにセンサーを付けておこう。LEGOのRISキットのマニュアル（Constructopedia）のページ28の図４のように、センサーを支える構造を作成しよう。センサーを付けたときの様子を次の図に示す。



センサーを入力１（ディスプレイ上部の１の番号のポート）につなぐ。

センサーの信号を待つ

まず、何かにぶつかるまで前進する簡単なロボットを作成しよう。そのプログラムは次のようになる。

```
task main()
{
 SetSensor(SENSOR_1,SENSOR_TOUCH);
 OnFwd(OUT_A+OUT_C);
 until (SENSOR_1 == 1);
 Off(OUT_A+OUT_C);
}
```

このプログラムには、重要なところが２カ所ある。第一は、タスクの一行目でセンサーのタイプを指定することである。第２は、３行目の待機用のuntil文である。SetSensorの文で上記のようにポート番号およびセンサーのタイプを指定することができる。ポート２はSENSOR_2で、ポート３はSENSOR_3で指定でき、タッチ・センサは前記のようにSENSOR_TOUCH、光センサーはSENSOR_LIGHTのように指定することができる。上記のプログラムは、センサーの指定の後の文で両方のモータがオンになって、ロボットが前進する。この（前進する）状態はその次の文でタッチ・センサが押されるまで保たれる。センサーが押されるとSENSOR_1（センサーの状態を表す変数）の値が１となり、untilの後の文の実行にプログラムが進み、両方のモータがオフになるため、ロボットが止まる。センサーが押されない限りSENSOR_1の値は0になる

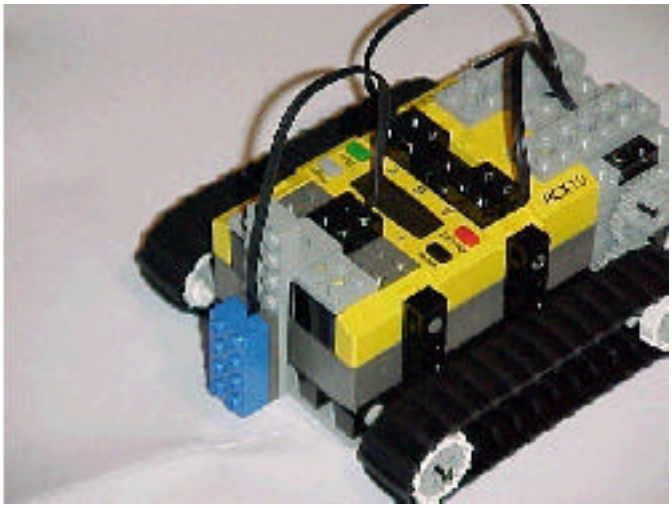
ため、その次の文は実行されないことになっている。

タッチ・センサで反応する

ロボットが障害物を避けるようにプログラミングしよう。このために上記のプログラムを改良して、ものにぶつかったときロボットが少し後退し、方向転換をしてから前進するようにしよう。そのプログラムを以下に示す。

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

前例と同様にまずセンサーを指定する。次にモータをオンにしてから前進させる。ifの無限ループ（永遠に繰り返される）内でセンサーの値をチェックして、ロボットが何かに当たったときは0.3秒間後退し、0.3秒間右折してから再び前進する。



光センサー

タッチ・センサの他にLEGOのキットには光センサーがある。この光センサーで光の量を測ることができる。キットの光センサーは発光ダイオードも含んでいるため、ものに反射される光によって判断などが可能

である。特に、ロボットに床に引いたラインを追跡させるには、このようなセンサーが便利である。この応用は次の例で実現する。そのために、まずロボットに光センサーを付けなければならない。光センサーをポート２に付けて、前記の図に示したように下に向けよう。

このロボットをテストするためにキットに付いているコース（真ん中に黒いトラックが引いてある大きな紙）が必要となる。基本的なアイデアは、ライトはいつも黒い線の上にあり、ラインから離れる（反射光が強くなる）と、方向を訂正する。次のプログラムは、ロボットが時計回り（右回り）に前進するときに使用できる。

```
#define THRESHOLD 40

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

このプログラムでは、入力ポート２に光センサーが接続されていることを指定している。その後モータをオンにし、ロボットに前進させる。whileの無限ループでは、反射光（センサーで検知された光）の値が40（照明に合わせてこの定数を変更する）以上になると、片方のモータの回転方向を変えてトラックに戻る（検知される値は４０以下になる）まで動き続ける。ロボットがトラックに戻ったら、両方のモータを同じ方向に回転させる。

上記のプログラムで得られるロボットの動きは少し荒い。より滑らかに動かすには、Wait(10)のような文を**until**文の前に実行すればよい。前述のように、このプログラムは左回りに使用できない。両方の方向に動くロボットを作成するためには、もっと複雑なプログラムが必要となる。

まとめ

この章ではタッチ・センサーおよび光センサーの使用、および、**until**文とセンサーの使用について学習した。ここまでみてきた概念の理解を深めるために、いろいろなプログラムの自作が必要となるだろう。ここまで述べてきた文でかなり複雑な動きのものを作成することができる。右側と左側にタッチ・センサーを取り付けたロボットや、周辺に線を引いた領域内でしか動かないロボットなどを試してみよう。

VI タスクとサブルーチン

ここまで作成してきたプログラムは一つのタスクしか含んでいない。しかし、NQCのプログラムは多数のタスクで構成することができる。また、プログラムのいろいろなところに使用されている文を一つのサブルーチンに纏めたりすることができる。タスクとサブルーチンの使用でプログラム全体が理解しやすくなり、コンパクトになることがある。このようないくつかの使い方についてここで述べる。

タスク

NQCのプログラムは最大10個までのタスクしか使用できない。各タスクには、重複のない名前が付けられる。その中の一つは必ずmain()という名前であり、このタスクが実行される。main()以外のタスクは、main()または他のタスクからstart命令で起動され、実行される。startを実行したタスクと起動されたタスクの両方がstartの実行時点から平行に実行される。実行中のタスクが他の実行中のタスクをstop命令で止めることができる。止められたタスクはstartで起動できるが、止まった時点から実行されるのではなく、最初から実行される。多数のタスクで構成されているプログラムを作成してみよう。

```
task main()
{
 SetSensor(SENSOR_1,SENSOR_TOUCH);
 start check_sensors;
 start move_square;
}

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

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

ロボットに、再びタッチ・センサを付けよう。ロボットに四角を描いて進ませるプログラムを再び作成するが、障害物にぶつかったとき反応するようにする。ロボットがモータを制御しながらセンサーの状態を

チェックしなければならないため、一つのタスクでは実現が困難である。このため、二つのタスクを用いることにする。その一つはロボットの動きを制御する。もう一つはセンサーの状態に反応する。

上記に示すプログラムでは、main()タスクがセンサーを指定してからその他の二つのタスクを機動させる。main()は他のタスクをスタートさせた後終了する。move_squareのタスクは、whileの無限ループでロボットに四角を描きながら進ませるものである。check_sensorsのタスクはタッチ・センサーをチェックするものであり、センサーが押された場合、まずmove_squareのタスクを止める。これは重要なことである。その後、停止しているロボットに後退させ、方向転換をさせた後move_squareをスタートさせる。この時点からmove_squareはロボットを制御する。

start命令を実行したタスクおよびstartで起動されたタスクが同時に、しかも、平行に実行されることは非常に重要なことである。しかし、平行に実行することが予想外の結果をもたらすこともある。第X章ではこれらの問題やそれに対する対策について述べる。

サブルーチン

プログラムを作成するとき、何カ所にも同様な文が必要になる場合がある。このとき、そのプログラムの一部分を一カ所に纏めてサブルーチンとして使用することができる。しかも、必要な場所でそのサブルーチンを呼び出すだけで目的の操作が行えるようになり、プログラムがコンパクトになる。

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

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

NQCでは八つのサブルーチンまで使用できる。前記のプログラムは一つのサブルーチンの使用例である。この例では、turn_aroundのサブルーチンでロボットの進行方向を変えることができる。このサブルーチンがmain()では３回呼び出されている。サブルーチンを呼び出すときは、その名を書いた後空の（）を書いてセミコロンで区切る。つまり、今まで使用したWaitなどのような命令と同じように使用するが、（）内にはなにも書かない（引数なし）。サブルーチンは便利であるが、扱いに注意が必要である。たとえば、サブルーチンは他のサブルーチンから実行できないが、タスクならどのタスクからでも実行できる。しかし、いくつかのタスクが一つのサブルーチンを同時に実行すると、予想外の動作が発生するため、これは危険でもある。また、２つ以上のタスクから一つのサブルーチンを呼ぶと、RCX（黄色い箱）本体のメモリ制限やソフトウェア機能の制約などのために、サブルーチンに複雑な式が使用できなくなる。目的の動作を正確に把握していない場合は、多数のタスクから一つのサブルーチンを呼び出すことは避けよう。

サブルーチンは便利であるが、前述のような問題も起こす。サブルーチンの長所の一つは呼び出しの回数とは関係なくRCXの中では一つだけ用意されることである。RCXの小さなメモリを考えれば、この特徴はとてもよい。小さなサブルーチンを書くとき、次に述べるインライン関数の方がよい。インライン関数はサブルーチンと違って呼び出される位置にその記述が複製される。そのため記述するプログラムは小さくなるが、実行のとき各呼び出しが展開され、必要なメモリが増える。しかし、その反面で、複雑な式などを書いたり制限なく多数のインライン関数が利用できる。また、その書式および呼び出しはサブルーチンと同様であり、subの代わりにvoidのキーワードを使用するだけである（C等のプログラミングにならって、ここでもvoidを用いる）。前記の例は、インライン関数を使うと次のようになる。

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

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

インライン関数はサブルーチンと比べると、もう一つ長所がある。これは、引数をもつことができることである。引数を用いればインライン関数内の変数値を指定することができる。たとえば、上記の例で回転する時間を一つの引数にすると、そのプログラムは次のようになる。

```
void turn_around(int turntime)
{
 OnRev(OUT_C);Wait(turntime);
 OnFwd(OUT_A+OUT_C);
}

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

インライン関数名の後の（）内で関数の引数を宣言する。この例では、引数は一つの整数型で、その名はturntimeであると宣言されている。二つ以上の引数を宣言するときは、カンマで区切って記述する。

マクロの定義

上記の記述の他に、コンパクトなプログラムコードを書くもう一つの方法がある。NQCでは、マクロを記述することができる。ただし、NQCのマクロはRCXコマンド・センターのマクロとは別のものである。これまでdefineを用いて定数を定義してきたが、定数だけではなくいろいろな命令の組み合わせも一つのマクロとして定義することができる。上記のプログラムをマクロを用いて書き直すと次のページに示すようになる。このプログラムでは、turn_aroundはその後に続く三つの文と等価になり、タスクの中にでてくるこの語には後に記述された文が代入される。マクロを書くときは、マクロと等価な文をすべて一行に書かなければならない。二つ以上の行に亘るマクロの書き方も可能であるが、あまりよい方法ではない。実は、define文は強力なものであり、引数を持たせることもできる。たとえば、マクロを実行するとき、スリープ時間などを指定することができる。

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

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

以下の例ではこのような機能を持つ四つのマクロが定義されている。

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

task main()
{
 forwards(3,200);
 turn_left(7,85);
 forwards(7,100);
 backwards(7,200);
 forwards(7,100);
 turn_right(7,85);
 forwards(3,200);
 Off(OUT_A+OUT_C);
}
```

最初のマクロは右折用、２番目は左折用、３番目は前進用、最後のは後退用のマクロである。各マクロには速度（s）と時間（t）の２つの引数がある。このようにマクロを定義するとプログラムが見やすくなり、コンパクトになる。たとえば、モータのつなぎ変えなどの変更のときも便利である。

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定義について学習した。これらの記述方式には色々な用途がある。タスクは平行動作のときは便利であり、サブルーチンはプログラムをコンパクトにするときに便利である。サブルーチンが短いときはインライン関数を用いることができる。より小さい文のときはマクロが便利で、しかも引数も使用できるため多く利用される。

ここまで述べたことは基本であり、ロボットのプログラミングには必須である。次章から述べる事柄は特殊な応用のときに活用する。

Ⅶ　音楽の作成

RCXには内蔵のスピーカがあり、音を鳴らしたり、簡単な音楽などが作成できる。音で状況などを知らせるのに使用できる。また、音楽を流しながら方向を変えたりするような面白いロボットが可能となる。

組み込みサウンドについて

RCXには五つのサウンドが用意されている。簡単に説明する。

0　キーを押す音　　1　ビープ　ビープ　　2　低くなる音

3　高くなる音　　4　ブー　（エラーなどを知らせるには便利）　　5　急速に高くなる音

これらの音をPlaySound()の命令で使用できる。次のプログラムはその例を示す。

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

上記プログラムのWait文は要らないと思うだろう。実は、音を鳴らす命令は音が終わるまで待たずに、次の命令の実行に移るため、スリープ文が必要である。RCXには音を格納するためのメモリがあるが、小さくてすぐに満杯になって音が失われてしまうことがある。音だけなら我慢できるが、音楽を鳴らすとき、これは問題になる。PlaySound()の引数は定数であることに注意されたい。変数が使用できない。

音楽の再生

音楽を作成するために、NQCにはPlayTone()の命令がある。このコマンドは、周波数を指定するものと音を鳴らす時間を指定するものの二つの引数をもつ。時間はWait命令と同様に1/100秒単位で指定する。使用できる周波数を次の表に示す。

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

サウンドと同様にPlayToneの命令は実行が終わるまで待たないため、スリープが必要となる。特に多数のPlayToneを用いると、スリープとの組み合わせが必須となる。次のプログラムはその使用例である。

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

RCXコマンド・センターにはピアノのような画面で弾ける鍵盤があり、これを用いれば簡単に音楽が作成できる。次のプログラムでは、ロボットが前進と後退を繰り返しながら音楽を鳴らす。

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

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

まとめ

この章では音および音楽の作成について学習した。

Ⅷ　モータについて（より詳しく）

モータを制御するために他にもいくつかの命令がある。これについて述べる。

滑らかに止まる

Off()の命令を用いれば、モータがRCXのモータ・ブレーキで直ちに止まる。NQCではモータをゆっくり止めることができる。これを行うためにFloat()の命令を用いる。用途によって、Off()よりこの命令の方が適していることがある。次のプログラムはその一つの例である。このプログラムではまずブレーキを用いてロボットを止める。プログラムの最後にもう一回モータを止めるが、Floatを使用する。効果の違いをよく見てもらいたい。実は、本ロボットでは差が小さいが、他の機械では大きな違いがみられる。

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

上級命令

使用してきたOnFwd()命令の実行で、実は二つの操作が順に行われる。すなわち、モータをオンにし、方向を前進に設定する。同様にOnRev()命令では、モータをオンにし、方向を後退に設定する。NQCではこれらの操作を別々に行うことができる。操作の一つのみ行いたいとき、これは便利である。これらの操作を別々に行うことでプログラムの効率が向上し、メモリの節約ができ、実行が高速になり、ロボットの動きも滑らかになる。

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

方向を設定するためにSetDirection()を使用する。このとき次のパラメータ（予め定義されている）

OUT_FWD（前進）、
OUT_REV（後退）および
OUT_FLIP（電流の方向を反転させる）
を用いることができる。

モータのモードを設定するためにSetOutput()の命令および次のパラメータ
OUT_ON（オンにする）、
OUT_OFF（オフにする）および
OUT_FLOAT（ブレーキを用いずにモータを止める）
が使用される。

上記のプログラムはロボットに前進させてから後退させ、その後再び前進させてから滑らかにロボットを止める。プログラムを実行するときは、特に指定をしなければ、すべてのモータは前進し、回転力は７となる。このため、上記のプログラムでは、最初の二つの命令は不要である。

モータを制御するためにその他多数の命令がある。すべての命令リストは次のようになる。

On(モータ)	モータをオンにする
Off(モータ)	モータをオフにする
Float(モータ)	モータを滑らかに止める
Fwd(モータ)	モータを前進に設定するが、オンにしない
Rev(モータ)	モータを後退に設定するが、オンにしない
Toggle(モータ)	モータの回転方向を変える
OnFwd(モータ)	モータを前進に設定し、オンにする
OnRev(モータ)	モータを後退に設定し、オンにする
OnFor(モータ,ティック)	ティックで指定された時間モータをオンにする
SetOutput(モータ,モード)	モータのモードを指定する（上記説明を参照）
SetDirection(モータ,方向)	モータの回転方向を指定する（上記説明を参照）
SetPower(モータ,回転力)	モータの回転力を指定する（次節の説明を参照）

モータの回転速度を変える

作成してきたプログラムでは速度を変えてもあまり変化がないと気付いただろう。実は、SetPower ()はモータの速度ではなく回転力を変えている。このため、モータが動かすものの重量が変わらない限り変化がみられない。本ロボットは、設定２と７の違いが小さい。速度効果を得るために、急速にモータをオンにしたりオフにしたりする「トリック」が必要である。次のプログラムにその例を示す。このプログラムは２つのタスクで構成されている。run_motorはモータを起動するタスクである。このタスクは速度変数の値をチェックして、値が正の場合は前進、負の場合は後退と解釈している。方向を設定した後待機してからモータをオフにする。main()のタスクは速度だけを設定する。

```
int speed, __speed;

task run_motor()
{
 while(true)
 {
  __speed = speed;
  if(__speed > 0) {OnFwd(OUT_A+OUT_C);}
  if(__speed < 0) {OnRev(OUT_A+OUT_C);__speed = -__speed;}
  Wait(__speed);
  Off(OUT_A+OUT_C);
 }
}

task main()
{
 speed = 0;
 start run_motor;
 speed = 1;   Wait(200);
 speed = -10; Wait(200);
 speed = 5;   Wait(200);
 speed = 2;   Wait(200);
 stop run_motor;
 Off(OUT_A+OUT_C);
}
```

実は、このプログラムを、方向転換やOff()の後にWait（）を挿入することなどで改善できる。

まとめ

この章では、モータを滑らかに止めるFloat()、モータの回転方向を設定するSetDirection()、モータのモードを設定するSetOutput()の命令およびこれらの命令で用いるパラメータ（OUT_FWD（前進）、OUT_REV（後退）、OUT_FLIP（電流の方向を反転させる）およびOUT_ON（オンにする）、OUT_OFF（オフにする）およびOUT_FLOAT（ブレーキを用いずにモータを止める））について学習した。また、モータを制御するためのすべての命令リストについてみてきた。さらに、モータの速度を制御するための「トリック」の一つを学んだ。

IX　センサーについて（より詳しく）

第V章ではセンサーについて述べたが、その基本に加えてここではセンサーについていろいろ学習していく。この章ではセンサー・モードとセンサー・タイプとの違い、角度センサー（RISに含まれていないが、別途で購入可能）の使用、三つ以上のセンサーの使用、および近接センサーの構築について学ぶ。

センサーのモードとタイプ

第V章で使用したSetSensor()の命令は二つの操作を行う。すなわち、センサーのタイプを設定し、センサーの（動作）モードを設定する。タイプとモードを別々に設定すると、センサーをより精密に制御することができ、いろいろな応用に役立つ。これらの設定を独立に行うために、SetSensorType()およびSetSensorMode()の命令がある。

SetSensorType()はセンサーのタイプを設定するために用いる。タイプには、SENSOR_TYPE_TOUCH（タッチ・センサー）、SENSOR_TYPE_LIGHT（光センサー）、SENSOR_TYPE_TEMPERATURE（温度センサー）、SENSOR_TYPE_ROTATION（角度センサー）がある。センサーのタイプを設定することによって、RCXはセンサーが電圧を必要としているかどうかを判断できるようになる。たとえば、光センサーのタイプを設定しないで光センサーを使用すると、センサーの発光ダイオードが光らなくなる。

SetSensorMode()はセンサーのモードを設定する命令である。八つのモードの中から選ぶことができる。その中で一番重要なモードは、SENSOR_MODE_RAW（未加工モード）である。このモードでは、センサーで得られる値の範囲は0～1023となる。その値が何を意味するのかはセンサーによって異なる。タッチ・センサーの場合、センサーが押されていないときは値が1023に近くなる。押されるときは50に近くなる。完全に押されていないときはセンサーから得られる値は50～1000の間になる。つまり、このセンサーを未加工(raw)モードで使用すると、タッチ・センサがどの程度押されているかをRCX側が値によって調べることができる。光センサーの場合も、このモードでは非常に暗い800から非常に明るい300までの値が得られる。このような方法でSetSensor()より厳密な値を測ることができる。

モードにはSENSOR_MODE_BOOLもある。このモードで得られる値は0か1のいずれかの一つになる。未加工(raw)モードでの値が550以上の場合、このモードでは値が１となり、それ以外は０となる。このSENSOR_MODE_BOOLモードはタッチ・センサーのデフォルト・モードとなっている。

モードには温度センサーで使用するSENSOR_MODE_CELCIUS（摂氏温度）およびSENSOR_MODE_FAHRENHEIT（華氏温度）のモードがある。SENSOR_MODE_PERCENTのモードは未加工の値を0～100の間の値に対応させるものである。しかし、この対応は均等なものではなく、未加工の値が400以下の場合は100に対応し、401以上の値が少しずつ0までの値に対応する。このモードは光センサーのデフォルト・モードである。後述の角度センサーの設定にSENSOR_MODE_ROTATIONを用いる。

前記のもの以外に、SENSOR_MODE_EDGEおよびSENSOR_MODE_PULSEのモードがある。これらのモードは変化を数えるために使用される。未加工モードでは、高い値から低い値への変化あるいはその逆が数えられる。タッチ・センサーの場合、ボタンが押されると高い値から低い値への変化が起こる。ボタンを離すと逆の変化がみられる。センサーをSENSOR_MODE_PULSEに設定すると、低い値から高い値への変化しか数えられない。つまり、タッチ・センサーの場合、ボタンの押し離しは一つの変化と見なされる。このセン

サーをSENSOR_MODE_EDGEに設定すると、ボタンの押し離しがそれぞれ一つの変化として見なされる。つまり、この場合はボタンの押し離しが二つの変化として数えられる。このモードでタッチ・センサーが何回押されたかを数えたり、光センサーがモニタしているライトが何回オンになったかを数えることができる。各センサーには一つのカウンタが用意されている。これらのカウンタをClearSensor()の命令でゼロに戻す（設定する）ことができる。

これらのいくつかのモードを使用するプログラムを作成してみよう。次のプログラムではロボットをタッチ・センサで起動する。このセンサーを一番長い接続線でポート１に接続しよう。センサーを二回押すとロボットが前進する。センサーのボタンを一回だけ押すと止まるようになっている。

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

センサー・タイプおよびセンサー・モードの設定を行うときは、必ず先にセンサーのタイプを設定しよう。タイプ設定でモードも設定されるから、この順序を守る必要がある。カウンタの値を保持する変数は、ポート指定に使用した変数（上記のSENSOR_1など）であることに注意されたい。

角度センサーについて

角度センサー（回転センサーとも呼ばれる）はRISに含まれていないが、別途でLEGOから購入できる。このセンサーには穴があり、モータの軸などをこの穴に挿入して、軸の回転量を測ることができる。

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

一回転は16ステップである。逆方向の一回転は-16ステップである。この角度センサーは、ロボットの動きを厳密に制御するには非常に便利である。また、ある軸をある角度まで動かしたりすることができる。歯車の組み合わせで16ステップより細かい制御の実現も可能である。

使用してきたロボットの各モータに角度センサーを付ければ、早い方のモータを一時的（Float()などで）に止めることによって他のモータの速度に合わせることができ、モータの速度制御に使用できる。このためには、センサーで得られた値を一致させればよい。この応用は上記のプログラムで実現されている。ロボットの各モータに角度センサーを付けて、モータＡのセンサーをポート１、モータＢのセンサーをポート３に接続しよう。

上記のプログラムでは、センサーが角度センサーであることが指定されている。また、それぞれのカウンタ（SENSOR_1およびSENSOR_3）をリセット（ゼロに設定）する。whileの無限ループ内では、センサーで得られる値が一致しているか否かチェックされる。一致する場合、前進を続ける。そうではない場合、一致するまで片方のモータが止められる。このプログラムを改良すれば、特定の距離までロボットを動かしたり、正確な右折や左折が可能である。

複数のセンサーを一つの入力に接続する

RCXは入力が三つしかないが、二つ以上のセンサーを一つのポートに接続して、より複雑なロボットを作成することができる。

```
int ttt, tt2;

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

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

一番簡単な接続方法は、一つのポートに二つのタッチ・センサーを付けることである。いずれかの一つまたは両方のセンサーが押されれば得られる値が１となる。一つも押されていないときはその値が０となる。この場合は押されたスイッチの区別ができないが、用途によっては必要ない。たとえば、ロボットの前と後ろにタッチ・センサーを付ける場合、進行方向で押されたセンサーの区別ができる。センサーを未加工(raw)モードに設定すれば、同種類のセンサーでも検知する値が異なるため、取得値でセンサーの区別が可能である。両方のセンサーが押されると、通常は30前後の値になる。

一つのポートにタッチ・センサーと光センサーを付けることもできる。この場合は光センサー用のタイプでないと光センサーが使用できなくなる。モードを未加工に設定しよう。この場合、タッチ・センサーが押されると、センサーから得られる値は100以下となる。押されていないときは光センサーの値が得られる。光センサーで得られる値は100以下にならないため、このような併用が可能である。上記のプログラムはこのことを利用している。ロボットが下向きの光センサーとフロントに付けられているタッチ・センサをもち、しかも、両方がポート１に接続されていることを仮定している。このようなロボットが前進し、地面上の黒い線を検知したとき、もしくは何かにぶつかると後退する。黒い線の上を通ると750以上の値が得られるため、ロボットが後退する。前進中に何かにぶつかると、得られる値が100以下となるため、ロボットがこの場合も後退する。プログラムには、二つのタスクがある。moverandomはランダムにロボットを動かし、main()はmoverandomを起動してセンサーを設定してからイベントの発生を待つ。センサーで得られる値が小さい（何かにぶつかった）とき、あるいは大きい（白い領域外）とき、main()がランダムな動きを止め、少しロボットを後退させた後、再び動きを開始させる。もちろん一つのポートに二つの光センサーを接続することができるが、得られる値が両方の光センサーで得た光の合計になるため、前記のようなセンサーの区別は困難である。他のセンサーの組み合わせも可能だが、センサーの区別などが困難と思われる。

接近センサーの作成

タッチ・センサーを用いれば、衝突したとき、ロボットに反応させることができる。しかし、ぶつかる前に反応させられればもっといい。このため、ロボットが物に接近していることを検知する必要がある。しかし、これを実現する専用のセンサーがない。コンピュータもしくは他のロボットとの通信のため、RCXには赤外線の送受信器のポートがある（後者の通信について第XI章に述べる）。光センサーは赤外線の光に反応するため、接近センサーに使用できる。
基本的には二つのタスクが必要となる。一つは、赤外線の送受信用のポートでメッセージを送信する。もう一つは光センサーでものの接近によって変動する反射の光を検知する。変動が激しければ激しいほどものが近いと判断される。このアイデアを使用するために光センサー（ポート２）を前向きに、赤外線のポートの上に設置する必要がある。こうすると、赤外線ポートから送信された光の反射光を測ることができる。反射される光の変動を容易に検知するために、未加工モードを使用する。

次のプログラムでは、前進中のロボットがものに接近すると90度で右折する。send_signalのタスクはSendMessage(0)を使用して、１秒間に10回赤外線（IR：Infra Red）のメッセージを送信する。
check_signalは光センサーで得られた値を格納し、少し時間をおいてからその値が200増加したか否かをチェックし、過去の値から200以上増加したとき、ロボットに90度で右折させる。200の値は適当で、これより小さい値を用いるともっと遠くでロボットが曲がり、200より大きな値を用いるともっと近くで曲が

る。実は、この値は周辺の材質や照明の強さなどに依存している。妥当な値は実験で得るようにしよう。

このテクニックの短所は一つの前進方向にしか使用できないことにある。両サイドの衝突を検知したいときは、タッチ・センサーを使用する必要がある。

```
int lastlevel;           //To store the previous level

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

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

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

ここで述べた方法は迷路でロボットを動かすには便利であるが、コンピュータとの通信ができなくなる。また、テレビのリモコンが使えなくなる場合もある。

まとめ

この章ではセンサーについてさらに詳しく触れて、センサーのタイプおよびモードの設定について学習した。また、モードの設定によってさらに詳しい情報を取得する方法についてみてきた。角度センサーの使用についても学んだ。さらに、一つのポートに複数のセンサーを使用する方法もみた。最後に、RCXの赤外線ポートを使用した接近センサーの作成について学習した。これらのトリックやヒントなどはより複雑な機械の作成に活用できる。

X　並列タスク

タスクは同時に（並列に）実行できることを学習してきた。これは、非常に有効な手段であり、センサーで得られる値を検知しながら、他のタスクが音楽を流したり、ロボットを動かしたりすることなどを可能にする。しかし、並列に実行されるタスクが問題を起こすこともある。たとえば、タスクが他のタスクの実行を妨害することがある。

間違ったプログラム

次のプログラムでは、一つのタスクはロボットが四角を描くように進行させる。もう一つのタスクはタッチ・センサーをチェックし、押された場合はロボットに後退させ、90度で右折させる。

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

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

このプログラムには問題がないように見えるが、実行中に予想外の動きが発生することがある。これを確かめるために、ロボットが右折している間にものにぶつけてみよう。後退し始めるが前進して再び衝突する。タスク間の実行干渉でこの現象が発生する。ロボットが右折するのは、main()の無限ループ内で二番目のスリープが実行中のときである。このとき、ロボットが何かにぶつかると２番目のタスクのifの条件が真となり、ロボットが後退し始めたときに前記のスリープが終了すると、ロボットが前進して再び衝突してしまう。このとき、後退用のスリープが実行中であるため、衝突が見逃される。明らかに、予想した動きではない。問題は、check_sensorsのスリープ中に行われる動作がmain()のタスクの起動で妨害されることにある。

タスクの起動の仕方と止め方

いずれのときもただ一つのタスクが実行中であるようにすれば、前記のような問題を避けることができる。第VI章ではこのアプローチを採用した。そのプログラムをもう一回みてみよう。

```
task main()
{
 SetSensor(SENSOR_1,SENSOR_TOUCH);
 start check_sensors;
 start move_square;
}

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

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

要点は、check_sensorsがmove_squareを止めてから、ロボットを動かすことにある。こうすることによって、後者が前者の動きを妨害することはなくなる。後退後、check_sensorsがmove_squareを起動する。この方法は効果的であるが、move_squareが最初から起動される。これは問題になることがある。タスクを実行途中に止め、その後そのポイントから再開するのが理想的だが、実現が困難である。

セマフォーの使用

この問題を解決する最も古典的な手法は、どちらのタスクがモータを使用しているかを示す変数を用いることである。一方のタスクがこの変数にモータを使用していることを示している間、他のタスクはモータを操作できない。この変数はセマフォーと呼ばれる。

セマフォー用の変数をsemと名付け、初期値を0（モータが停止）に設定したとする。この場合、変数の値に応じてモータを制御しようとするタスクは次のような命令を実行しなければならない。

```
until (sem == 0);
sem = 1;
// Do something with the motors
sem = 0;
```

このようにすれば、モータが解放されるまで待ち（semがゼロ）、自分がモータを使用することを表明し（semを１に設定）、使用した後でモータを解放（semを0に設定）することになる。次のプログラムはこの概念を採用している。

```
int sem;

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

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

このプログラムのmove_squareにはsemの設定が不要と思われるが、Fwd()の実行（第VIII章で述べたように三つの設定操作を含む）が妨害されないようにするために必要である。セマフォーは並列タスクを含むプログラムの作成には便利であり、ほとんどの場合は必要である。

まとめ

この章では、並列に動くタスク、またその実行に関連する問題について学習した。並列実行の副次効果に注意すべきであることを学んだ。予想外の動作はほとんどの原因が並列実行にある。これらの問題に対応する対策二つについてみてきた。一つは実行中のタスク数を一つに制限するものである。もう一つは共用資源を管理するセマフォーの概念を採用するものである。後者は重要な部分のみ実行されることを保証する。

XI　ロボット間の通信

二つ以上のRCXを持っているときはこの章が必要となる。RCXは赤外線ポートでPCおよび他のロボットと交信ができる。この特徴を用いれば共同作業ロボットなどの作成が可能になる。また、二つ以上のRCXを組み合わせたロボットの作成も可能になる。二つのRCXを使ってロボットを作成すると、６つのモータおよび６つのセンサー（第IX章のトリックでそれ以上）が使用できる。

ロボット同士が通信するときには、まず一方のロボットが赤外線ポートを介してSendMessage()命令で0〜255の値を送信する。他のロボットはこの値を受信し、格納する。格納された値によってロボットの動作を変えることができる。

命令の転送

二つ以上のロボットを使用するとき、一つをリーダーにするのが通常である。ここではこのロボットをマスタと呼ぶ。他方のロボットをスレーブにする。マスタが命令を出し、スレーブがそれに従う。場合によっては、センサーの値のような情報をスレーブからマスタへ転送することもできる。この場合、マスタ用およびスレーブ用の二つのプログラムが必要となる。以降、一つのスレーブしかないものと仮定する。まず簡単なプログラムから始めよう。スレーブのロボットは、前進、後退および停止という三つの動作が行える。そのプログラムは一つのループで構成され、ClearMessage()で受信内容を0に初期化した後、その値が変わるまで待ち、受信した内容によって前記のいずれかの動作を行う。プログラムは次のようになる。

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

マスタのプログラムの方は簡単である。マスタが命令の送信後２秒待ち、次の命令を送信する。そのプログラムは次のようになる。

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

プログラムの作成後、各プログラムをそれぞれのロボットにダウンロードする。一つのロボットにダウン

ロードするときは、もう一つをオフにする。プログラムをダウンロードした後、マスタ、スレーブの順で実行する。三つ以上のスレーブを使用するときは、一つずつにプログラムをダウンロードする。こうすることによって、すべてのスレーブに同一のプログラムが転送される。ロボット間の通信ができるようにするためには、プロトコルと呼ばれる通信のルールが必要である。上記のプログラムでは、１は前進、２は後退、３は停止と定義されている。このような定義は特に送受信の多い応用では非常に重要なことである。多数のスレーブを用いるときは、マスタは二つのメッセージでスレーブの番号と動作を指定するような応用も考えられる。この場合、スレーブは最初のメッセージの番号を確認し、自分の番号のときだけ、２番目のメッセージに従うことになる。このためには、スレーブにそれぞれ異なる番号が付いていなければならないが、異なる定数を指定するなど、各スレーブにちょっとずつ違うプログラムを用意することによって実現できる。

リーダーを選ぶ

　多数のロボットを用いるときは、前述のように各ロボットが専用のプログラムを必要とする。それより、一つのプログラムのみを作成してすべてのロボットにダウンロードした方が負担が減る。しかし、この場合は、マスタをどのように選ぶかが問題になる。

```
task main()
{
 ClearMessage();
 Wait(200);              // make sure all robots are on
 Wait(Random(400));      // wait between 0 and 4 seconds
 if (Message() > 0)       // somebody else was first
   { start slave;}
 else
 {
  SendMessage(1);          // I am the master now
  Wait(400);              // make sure everybody else knows
  start master;
  }
}

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

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

ロボットにリーダー（マスター）を選ばせれば、この問題が解決できる。ロボットをランダムな時間待つようにして、その後メッセージを送信するようにする。最初に送信したロボットがマスタになる。この方法では、二つ以上のロボットが同時にメッセージを送信すると、失敗する。しかし、このケースの発生をチェックするプログラムを作成すれば問題はない。このようなケースは滅多に起こらないが、リーダーが二つ以上あるいは一つもない状態になる場合がある。この場合はより細かいプロトコル（複雑なプログラム）が必要となる。

注意事項

複数のロボットを扱うときは、二つの問題が生じる。まず、二つのロボットが同時に情報を送信したときに発生する情報の損失である。もう一つは、PCから複数のロボットへプログラムをダウンロードしようとして失敗することである。前者は、ロボットが送信と受信を同時に行えないことに原因がある。後者では、PCがプログラムを一回で転送するのではなく、一部分を送信後、受信確認のメッセージを待ち、それを受信した後、残りの転送を続けるためである。このため、二つ以上のロボットから確認メッセージを受信した場合はPCが混乱し、結局プログラムのダウンロードが失敗する。ダウンロードはPC－ロボット（1台）の一組で行うものなので、プログラムのダウンロードを行うときは、電源オンのロボットを一つにしよう。

複数のロボットが同時にメッセージを送信すると、その情報が失われることがある。一つのロボット（マスタが一つのみ）が送信するようにすれば、この問題を解決することができる。スレーブがマスタへ情報（衝突の発生など）を送信するとき同時にマスタが命令の送信を開始すれば、スレーブからのメッセージが失われる。スレーブに命令の受信確認を送信させれば、この問題が解決できる。マスタが受信確認メッセージをある時間内に受信しない場合、命令再送信を行うようなプログラム部分は次のようになる。

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

このプログラムでは、スレーブからの命令受信確認用の番号は255となっている。マスタに近いロボットにしか命令を送信したくない場合、SetTxPower(TX_POWER_LO)の命令が使用できる。この命令を用いれば、マスタに近いロボットしか命令が「聞こえない」ことになる。この命令は二つのRCXで構成されるロボットの作成に役立つ。再び遠くに送信を行うためには、SetTxPower(TX_POWER_HI)の命令が使える。

まとめ

この章ではロボット間の通信について学習した。通信には、送信用のSendMessage()、初期化用のClearMesage()およびメッセージ確認用のMessage()の命令が使用される。通信用のプロトコル（やり取りの決まり）が重要であることも学んだ。コンピュータ間の通信にはプロトコルが欠かせない。ロボット間の通信にいろいろな問題が起こりうることをみて、プロトコルの厳密な定義の必要性を確認した。

XII　その他の命令

NQCには、その他の多数の命令がある。この章では、タイマー、ディスプレイおよびデータログ用の３種類について述べる。

タイマーについて

RCXには内蔵の四つのタイマーがある。これらのタイマーはティック（時間の単位）が1/10秒である。タイマーは０から３まで番号が付けられている。タイマーの内容はClearTimer()で初期化することができ、格納してある時間をTimer()で取得する。次のプログラムはロボットを２０秒間ランダムに動かす。

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

このプログラムは、第IV章のプログラムと同一のことを行っている。しかし、上記のタイマー使用のプログラムの方が簡単である。タイマーはスリープ命令の代わりに使用できる。タイマーを初期化してタイマーの内容がある特定の値になるまで待つようにすれば、定まった時間まで待機することになる。また、待機中に発生するイベントに反応させることも実現できる。次のプログラムでは、ロボットが１０秒経過するか、タッチ・センサーが押されるかのいずれかのイベントが発生するまで前進する。

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

タイマーの単位は1/10秒だが、スリープの時間単位は1/100秒であることに注意されたい。

ディスプレイについて

RCXのディスプレイを二通りの方法で制御することができる。表示するものは、システムクロック、センサーで得られる値およびモータで得られる情報の中から選べる。これは、RCXの黒いViewボタンを使用することと同じである。第一の方法では、ディスプレイのタイプの設定にSelectDisplay()命令を使用する。次のプログラムはすべて選択する例を示す。SelectDisplay(SENSOR_1)のような命令を使ってはいけないことに注意されたい。

```
task main()
{
 SelectDisplay(DISPLAY_SENSOR_1); Wait(100);          // Input 1
 SelectDisplay(DISPLAY_SENSOR_2); Wait(100);          // Input 2
 SelectDisplay(DISPLAY_SENSOR_3); Wait(100);          // Input 3
 SelectDisplay(DISPLAY_OUT_A);    Wait(100);          // Output A
 SelectDisplay(DISPLAY_OUT_B);    Wait(100);          // Output B
 SelectDisplay(DISPLAY_OUT_C);    Wait(100);          // Output C
 SelectDisplay(DISPLAY_WATCH);    Wait(100);          // System clock
}
```

第二の方法では、システムクロックを制御することによってディスプレイを制御する。この方法は、診断用の情報などを表示するために用いることができる。この方法では、SetWatch()[1]命令を使用する。次のプログラムはこの使用例を示す。SetWatch()の引数には定数しか使用できないことに注意しよう。

```
task main()
{
 SetWatch(1,1); Wait(100);
 SetWatch(2,4); Wait(100);
 SetWatch(3,9); Wait(100);
 SetWatch(4,16); Wait(100);
 SetWatch(5,25); Wait(100);
}
```

データログの使用

RCXは変数の値、センサーで得られた数字およびタイマーの内容をデータログというメモリの一部に格納することができる。データログに格納されたデータはRCXでは使用できないが、コンピュータに転送することができる。これは、ロボットで得られたデータの加工や、ロボットの診断などに使用できる。RCXコマンド・センターでは専用のウインドウでデータログの現在の内容をみることができる。データログを使用するためには三つのステップが必要である。まず、NQCのプログラムはCreateDatalog()の命令でデータログのサイズを指定しなければならない。この命令は同時にデータログの内容を削除する。第２のステップでは、AddToDatalog()の命令でデータを書き込む。データが順番に格納される。書き込むときは、RCXの

1 NQCのバーション1.3以降およびRCXコマンド・センターバーション2.3以降でしか使用できない。

ディスプレイには、四分割の円の絵が現れる。データログが満杯のとき、この四つの部分が塗った状態で表示される。データログが満杯になると、その時点から新データが格納されなくなる。第３のステップでは、データログの格納データをPCへ送信する。このため、RCXコマンド・センターのツール（Tools）メニューのDatalog命令を選択し、Upload Datalogのボタンを押す。すると、データログの内容がウインドウに表示され、データを見たりセーブしたりすることができる。そのデータに基づいて他のPCのプログラムでデータを加工して、３次元（3D）スキャナーを作成した人がいる。

次のプログラムは、光センサーを使用するロボット用のものである。ロボットが10秒間前進し、１秒に５回光センサーで検知した値がデータログに書き込まれる。

```
task main()
{
 SetSensor(SENSOR_2,SENSOR_LIGHT);
 OnFwd(OUT_A+OUT_C);
 CreateDatalog(50);
 repeat(50)
 {
  AddToDatalog(SENSOR_2);
  Wait(20);
  }
 Off(OUT_A+OUT_C);
}
```

菌　NQCの手引き

この章では、NQCで使用できる文、命令、定数などについて簡潔に纏められている。これらのものの大部分について既に触れてきたのでここでは概略のみ記されている。

文

命令	説明
while(条件)body	条件が成立する限りbodyを繰り返し実行する
do body while (条件)	bodyを実行し、条件が成立する限りbodyを繰り返し実行する
until(条件) body	条件が成立するまでbodyを繰り返し実行する
break	while/do/untilのbody（繰り返しループ）から脱出する
continue	この命令以降の命令を無視してwhile/do/untilのbody（繰り返しループ）の先頭に移動する。
repeat(式)body	式で指定された数bodyを繰り返し実行する
if(条件)　文 if(条件)　文１　else　文２	条件が成り立っていれば「文」を実行する 条件が成り立っていれば文１を、そうでない場合は文２を実行する
start　タスク名	指定されたタスク（の実行）をスタートする
stop　タスク名	指定されたタスク（の実行）を止める
function(引数)	指定された引数で関数を呼び出す（実行開始する）
変数　=　式	右の式を評価して、変数に代入する
変数　+=　式	右の式を評価して、変数値に加算し、変数の値を更新する
変数　-=　式	右の式を評価して、変数値からその値を引いて、変数の値を更新する
変数　*=　式	右の式を評価して、変数値にその値を掛けて、変数の値を更新する
変数　/=　式	右の式を評価して、変数値をその値で割って、変数の値を更新する
変数　|=　式	右の式を評価して、変数値との論理和をとり、変数の値を更新する
変数　&=　式	右の式を評価して、変数値との論理積をとり、変数の値を更新する
return	関数から、その関数を呼び出したもののところに戻る。
式	式を評価する

条件式

条件式は制御文と一緒用いられ、判断を下すために使用される。ほとんどの場合は、条件式は式と式を比較するものになる。

条件	意味
true	いつも真
false	いつも偽
式1 == 式2	式1（の値） は 式2（の値） に等しいかどうかをチェックする
式1 != 式2	式1 は 式2 と異なるかどうかをチェックする
式1 < 式2	式1 が 式2 より小さいかどうかをチェックする
式1 <= 式2	式1 が 式2 より小さいかそれに等しいかをチェックする
式1 > 式2	式1 が 式2 より大きいかどうかをチェックする
式1 >= 式2	式1 が 式2 より大きいかそれに等しいかをチェックする
! 条件	条件を否定する（条件が成り立たないこと）
条件1 && 条件2	条件1、かつ、条件2が成り立つことを表す
条件1 || 条件2	条件1、または、条件2が成り立つことを表す

式

式に用いるものの中には定数や変数やセンサーの値などがたくさんある。見てきたプログラムの中に用いたSENSOR_1,SENSOR_2およびSENSOR_3は、それぞれ以下のSensorValue(0),SensorValue(1)およびSensorValue(2)に展開されることに注意されたい。

値	説明
数字	123のような定数
変数	xのような変数
Timer(n)	n番目のタイマーの内容。ただし、nは0,1,2または3に限る
ramdom(n)	0～nの間の乱数
SensorValue(n)	n番目のセンサーの値。ただし、nは0,1,2または3に限る
watch()	システム（RCX）のwatchの値
message()	一番最近受信したメッセージの値

上記の値を演算子で組み合わせることができる。しかし、多数の演算子は定数もしくは定数のみを含む式にしか適用することができない。以下の表に示されている演算子は優先順位で示されている。

命令	説明	結合	制限	記述例
abs(被演算子) sign(被演算子)	被演算子の絶対値 被演算子の符号	なし		abs(x) sign(x)
++ --	インクリメント デクリメント	左 左	変数のみ 変数のみ	x++　　++x x--　　--x
- 単項演算子 ~ 単項演算子	条否定（符号） ビットの反転	右 右	定数のみ	-x ~123
* / %	乗算 割算 剰余（モジュール）演算	左 左 左	定数のみ	x * y x / y 123 % 4
+ -	加算 減算	左 左		x + y x - y
<< >>	左（ビット）シフト 右シフト	左 左	定数のみ 定数のみ	123 << 4 123 >> 4
&	（ビット）論理積	左		x & y
^	（ビット）排他的論理和	左	定数のみ	123 ^ 4
|	（ビット）論理和	左		x | y
&&	論理積（かつ）	左	定数のみ	123 && 4
||	論理和（または）	左	定数のみ	123 || 4

RCX関数

RCXの関数の引数は定数あるいは定数に評価される式のみに限られている。例外は、センサーの値を引数とする関数である。このとき、引数はSENSOR_1,SENSOR_2またはSENSOR_3のセンサー名でなければならない。また、場合によってはSENSOR_TOUCHのような名前が定数として使用される。

関数	説明	記述例
SetSensor(センサー,設定)	センサーの設定	SetSensor(SENSOR_1,SENSOR_TOUCH)
SetSensorMode(センサー,モード)	センサーモードの設定	SetSensorMode(SENSOR_2,SENSOR_MODE_PERCENT)
SetSensorType(センサー,タイプ)	センサータイプの設定	SetSensorType(SENSOR_2,SENSOR_TYPE_LIGHT)
ClearSensor(センサー)	センサー値のクリア	ClearSensor(SENSOR_3)
On(出力)	出力をオンにする	On(OUT_A + OUT_B)

（続き...）

関数	説明	記述例
Off(出力)	出力をオフにする	Off(OUT_C)
Float(出力)	出力をゆっくり止める	Float(OUT_B)
Fwd(出力)	出力を前進に設定する	Fwd(OUT_A)
Rev(出力)	出力を後退に設定する	Rev(OUT_B)
Toggle(出力)	出力を逆転する	Toggle(OUT_C)
OnFwd(出力)	オンにし、前進させる	OnFwd(OUT_A)
OnRev(出力)	オンにし、後退させる	OnRev(OUT_B)
OnFor(出力,時間)	1/100秒単位での時間出力をオンにする	OnFor(OUT_A, 200)
SetOutput(出力,モード)	出力モードを設定	SetOutput(OUT_A,OUT_ON)
SetDirection(出力,方向)	出力の方向を設定	SetDirection(OUT_A,OUT_FWD)
SetPower(出力,回転力)	出力の回転力を設定	SetPower(OUT_A,6)
Wait(時間)	1/100秒単位で指定された時間待機する	Wait(x)
PlaySound(音)	指定された音（0〜5）を鳴らす	PlaySound(SOUND_CLICK)
PlayTone(周波数,時間)	指定された周波数の音を指定された時間鳴らし続ける	PlayTone(440,5)
ClearTimer(タイマー)	指定されたタイマーの内容をゼロにクリアする	ClearTimer(0)
StopAllTasks()	実行中すべてのタスクを止める	StopAllTasks()
SelectDisplay(モード)	ディスプレイを７つモードから選んで設定する。0はシステムのwatch、1〜3センサーの値、4〜6出力の設定。モードの値は式でもよい	SelectDisplay(1)
SendMessage(メッセージ)	1〜255のIRメッセージを送る。その値は式でもよい	SendMessage(x)
ClearMessage()	IRメッセージをクリアする	ClearMessage()

（続き...）

関数	説明	記述例
CreateDatalog(サイズ)	指定されたサイズの新データログを作成する	CreateDatalog(100)
AddToDatalog(値)	指定された値をデータログに加える。その値は式でもよい	AddToDatalog(Timer(0))
SetWatch(時間,分)	システムのwatchを設定する	SetWatch(1,30)
SetTxPower(hi_lo)	IR送受信機の出力（届く範囲）を設定する	SetTxPower(TX_POWER_LO)

RCX定数

RCXの関数に使用される引数は名前付きの定数である。これらの定数を使うとプログラムがわかりやすくなる。できる限りこれらの「定数」を使ってもらいたい。

SetSensor()に使える設定	SENSOR_TOUCH,SENSOR_LIGHT,SENSOR_ROTATION, SENSOR_CELCIUS,SENSOR_FAHRENHEIT,SENSOR_PULSE, SENSOR_EDGE
SetSensorMode()のモード	SENSOR_MODE_RAW,SENSOR_MODE_BOOL,SENSOR_MODE_EDGE, SENSOR_MODE_PULSE,SENSOR_MODE_PERCENT, SENSOR_MODE_CELCIUS,SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION
SetSensorType()のタイプ	SENSOR_TYPE_TOUCH,SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT,SENSOR_TYPE_ROTATION
On(),Off()などの出力	OUT_A, OUT_B, OUT_C
SetOutput()のモード	OUT_ON, OUT_OFF, OUT_FLOAT
SetDirection()の方向	OUT_FWD, OUT_REV, OUT_TOGGLE
SetPower()の回転力	OUT_LOW, OUT_HALF, OUT_FULL
PlaySound()の音	SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP
SelectDisplay()のモード	DISPLAY_WATCH, DISPLAY_SENSOR_1,DISPLAY_SENSOR_2, DISPLAY_SENSOR_3,DISPLAY_OUT_A,DISPLAY_OUT_B, DISPLAY_OUT_C
SetTxPower()のパワー	TX_POWER_LO, TX_POWER_HI

予約語

予約語はNQCのコンパイラ専用の語であり、プログラム内の変数名や関数名に使用してはいけない。次の予約語がある。__sensor,abs,asm,break,const,continue,do,else,false,if,inline,int,repeat,return,sign,start,stop,sub,task,true,void,while。

拾　最後の論評

ここまでやってきた人はNQCのはエキスパートになっているだろう。まだ、そうでない場合、今すぐ始めよう。LEGOの試作およびプログラミングの工夫でいろいろすばらしいものを作成することができる。

このチュートリアルはNQC言語のすべてをカバーしていない。このため、NQCの資料を参照しよう。また、NQCはまだ開発中であり、これからのバーションではいろいろな機能が加えられるだろう。ここでは、人工知能や人工学習のプログラミングの面も触れていない。

RCXは直接PCから制御することができる。このためには、Visual BasicやJavaやDelphiのプログラミングが必要となる。このようなプログラミング言語で作成したプログラムは、RCXのプログラムとの共同作業が可能となる。この組み合わせは強力である。このような使用方法に興味のある方は、以下のLEGO MindStormsのWebサイトからspiritの技術参考書（technical reference）をダウンロードすることからスタートしてほしい。

http://www.legomindstorms.com/

インターネットは、理想的な情報源であり、その他関連のサイトについては、著者のサイト、

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

のリンク・ページ、もしくは、LEGOのユーザグループ

http://www.lugnet.com/

から情報を取得しよう。

lugnet.comのlugnet.robotics.rcx.nqcおよびlugnet.roboticsのニュース・グループにもたくさんの情報がある。