松本秀峰中等教育学校 夏期講座 2013年8月9～11日

自律型ロボットを作ろう！

～ マインドストームを使ったロボット製作の体験教室 ～

松本成司
matsu@johnen.shinshu-u.ac.jp

初版 2013-08-09
最終更新 2013-08-28

目次

0. この文書について

この文書は、 松本秀峰中等教育学校において2013年8月9～11日に開催される夏期講座の補助資料です。 基本的には2012年8月に同じく松本秀峰中等教育学校において開催された ロボティクス入門講座のための補助資料を 加筆・修正したものです。 ただし2012年度の講座と違い、 旧モデルのマインドストーム・ロボティクス・インベンション・システム（RIS)は使用せず、 マインドストームNXTだけを使用しますので、RIS用の説明やプログラム（NQC)は省略してあります。

この講座で使用するプログラミング環境のNXCを開発し、 フリーソフトウェアとして配布してくれたJohn Hansenさんに感謝します。 また、この資料を作るにあたって非常に多くの書籍やウェブ上の資料を 参考にさせていただきました。 あらためてこれらの製作者の方々に感謝します。 そして、このような講座を担当する貴重な機会を与えてくださった 松本秀峰中等教育学校の瀬川伸先生に感謝します。

Note

クリエイティブ・コモンズ・ライセンス（CCライセンス）を利用することで、 作者は著作権を保持したまま作品を自由に流通させることができ、 受け手は許諾条件の範囲内で再配布やリミックスなどをすることができます。 この文書で採用しているライセンス（CC BY-SA 3.0）は、簡単に表現すると 『原著作者のクレジット表示』と『二次的な著作物は同一のライセンスで頒布』という条件のもとで、商業利用も含めて自由に複製、頒布、改変することができます。

ただし、（おそらくたくさんあると思われる）間違いにお気づきの方は ご連絡いただけると幸いです。コメントも歓迎します。 また、３日間の集中講座で行う内容を優先して説明してありますので、 取り挙げられなかった内容の説明については、まだ作成途中の箇所が多いです。

1. マインドストームで簡単ロボット製作

自律型のロボットって何？

一般に、人間の代わりにいろいろな作業をしてくれる機械や さまざまな動物の形や動きを真似て動作する機械をロボットと呼びます。 後者は、昔から日本のアニメや特撮でよく登場しているので日本では 特に馴染み深いですね。 また実際に動くロボットとして、 ホンダのASIMOやソニーのAIBOを思い浮かべる人も多いでしょう。 今日では、産業用ロボット、医療ロボット、レスキューロボット、 水中探査ロボット、お掃除ロボット、軍事ロボットなどさまざまな種類のロボット

Note

自動的にネットを巡回するコンピュータ・プログラムなどのことも ロボットと呼びます。 略してボット(bot)とも言いますが、最近では自動的に迷惑メールを送信したり 個人情報を盗み出したりする悪意のあるプログラムのことをボットと 呼ぶことが多くなりました。

これらのロボットの中には人間が遠隔で操作するものもありますが、 この講座で製作するのは、人間が操作しなくても 自分自身で状況を判断して動作することのできるロボットです。 このような、自分自身を律することのできるロボット、 自分自身をコントロールできるロボットのことを **『自律ロボット』**と呼びます。 ちなみに**『自立**ロボット』と書く場合も、 単に自分で立てるという意味ではなく、 『自律ロボット』と同じ意味で使うことが多いようです。

さて、そうは言っても「どういう場合にどういう動きをさせるか」 というのは、あらかじめ人間がロボットに教えてあげる必要があります。 間違ったことを教えれば間違った動きをするので、 ロボットを動かす手順をあらかじめよく練っておくことが大切です。 この手順、つまりロボットを動かす命令をまとめて 記述したものを**『制御プログラム』**、 または単に**『プログラム』**と呼びます。

Note

このプログラムのことをソフトウェア（または簡単にソフト）と呼んでもかまいません。 が、この文書では、パソコン用に開発され一般に配布されている、または販売されている コンピュータ・プログラムのことをソフトウェアと呼んで、 ロボットを動かすためのプログラムと一応区別しています。 しかし、これは正しい使い分けではありません。 実際、NXTの液晶メニューにはユーザが作ったプログラムも 「Software files」として表現されています。 一般には、 コンピュータ本体や機械本体をハードウェアと呼び、 これらを動かすためのコンピュータ・プログラムや関連文書、 さらにはユーザ教育などの無形物のことをソフトウェアと呼びます。

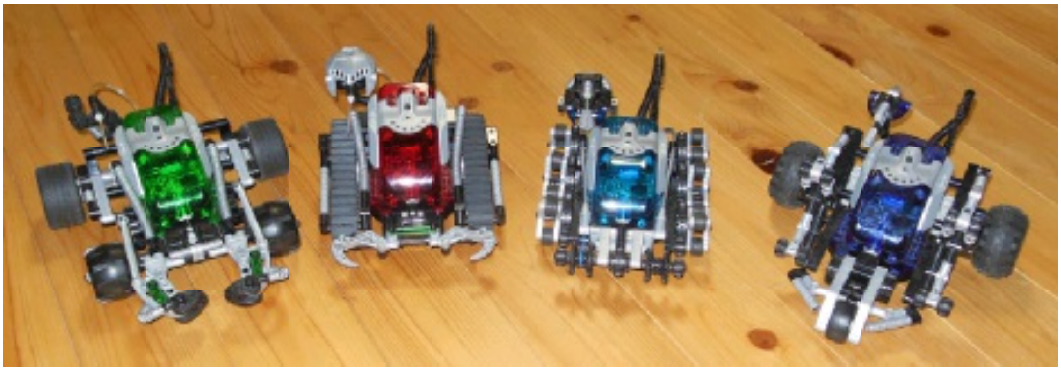
マインドストームって何？

昔は、個人で行うロボット製作と言えば、 必要な部品を自分で選び、探しまわって集め、 売られていない部品は自分で作る、というのが一般的でした。 そしてそのための知識もそれなりに必要でした。 また学校で使う教材についても、簡単に組み立てができて プログラムも自作できるというキットはほとんどありませんでした。

ところが、1998年に玩具のブロックで有名な**レゴ社（LEGO社）**が Mindstorms Robotics Invention System（マインドストーム・ロボティクス・インベンション・システム、略して**RIS**）とよばれる教育用の安価なロボットのキットを発売しました。 これはLEGO社がアメリカのマサチューセッツ工科大学(MIT)と共同開発したもので、 キットにはロボットの頭脳に当たるコンピュータをはじめ、 明るさを測る光センサ、触れたかどうか判定するタッチセンサ、モータ、 各種ギヤ類、タイヤ、従来のブロックなど700個以上の部品が含まれています。 もちろんLEGOの他のキットとも組み合わせて使うこともできます。 この安価なキットだけでオリジナルのロボットを作ることができ、 自作プログラムで自由に動かせるという点で この製品は非常に画期的なものでした。 ちなみにインベンションとは創作とか発明という意味です。

Note

RCXと呼ばれるのバッテリボックスを兼ねたRISのコントローラには、 日立（現ルネサステクノロジ）製のH8/300という16MHzで駆動 8ビットのマイクロプロセッサが搭載されています。 一方、NXTには 48MHzで駆動する32ビットのARM7マイクロプロセッサが搭載され、 モータもサーボモータになりました。



そして2006年には、RISの後継となる Mindstorms NXTが発売になりました。 コントローラの性能やモータの性能が大幅に向上し、 超音波センサや音センサなども追加されました。

これらのキットに含まれる部品の組み合わせ方は事実上無限通りで、 工夫しながらさまざまな目的の独自ロボットを簡単に作ることができます。 ちなみに箱に記載されているキットの対象年齢はRISが**12歳以上**、 NXTが**8歳以上**となっています。

ところで、ロボットを動かすためのプログラムはパソコンを使って作ります。 そのためのソフトウェアもキットに付属していますが、 このLEGO社純正のソフトウェア以外にも多くのソフトウェアが 開発されていて、自由に（もちろん無料で）使用できるものも少なくありません。

いずれにしても、まったく同じロボットでもプログラムを変更することで さまざまな動かし方をすることができます。 もちろんロボット自体を変更・改良した場合には、 プログラムも変更しなければいけないことが多いでしょう。 このようにロボットの機械的な工夫とプログラムの連携が ロボット製作の重要なポイントです。

Note

RIS以外にも スカウトと呼ばれるコントローラを含んだ Robotics Discovery Set (RDS)、 マイクロスカウトと呼ばれるコントローラを含んだ Droid Development Kit (DDK) や Dark Side Developer Kit (DSDK) なども発売され、 さらに2002年には モータやセンサが一体になったSPCと呼ばれるコントローラを含んだ 4種類の Spybotics (スパイボティクス) のキットも発売になりました。

Note

NXTでは、簡単なプログラムなら本体上でも作ることができます。

この講座の大まかな日程

一応の目標ということで次のような計画を立ててみました。

第１日

まずキットに慣れるために、付属のインストラクション（説明書）に従って左右のタイヤを 独立に動かすことのできるロボットを作ります。 そして、そのロボットを前や後ろに動かしたり、 左右に曲がらせるための簡単なプログラムをパソコンで作成します。 プログラムをロボットに転送して、実際にロボットを動かしてみます。

Note

LEGOに慣れている人なら初めからオリジナル・ロボットを作ってもかまいませんが、 最初はインストラクションに従って作った方が簡単ですし、勉強にもなります。

少し慣れてきたら、よりシンプルで改良しやすいプログラムを書くための手法を学習します。 このステップを省略しないことで、のちのちのプログラミングが楽に（楽しく）なります。

次のステップではセンサを使って周りの状況に合わせて動くように ロボットを改良します。

タッチセンサを使うと、障害物に当たったら向きを変えて進むロボットなどを簡単に作ることができます。 また明るさを測定できる光センサを使って、 黒い線に沿って動くロボットを作ってみましょう。

第２日

NXTではタッチセンサや光センサだけでなく超音波センサやサウンドセンサも使うことができます。 接触しなくても障害物を避けるロボットや音に反応して動くロボットを作ってみましょう。

ところで自分の作ったプログラムの動作を確認するのにいろいろな音を鳴らすと便利です。 そこで組み込まれた音を鳴らしたり、曲を演奏させるプログラムも作ってみます。

ひと通り、モータやセンサの使い方に慣れたら、次は二台のロボットで通信をしてみます。 相手に数や文字を送ったり、直接相手のロボットのモータを回すこともできます。 これだけでも相当な連携動作ができるようになります。

第３日

ここまで学習すれば、ロボットにかなり複雑な動作をさせることができるはずです。 いよいよオリジナル・ロボットへ挑戦です。 簡単なロボコンにチャレンジしてみましょう。

この講座で使うソフトウェアについて

基本的には、ロボットを動かすためのプログラムを作成するツールと プログラムをロボットが解釈できる形式に変換して ロボットに転送するツールがあれば十分です。 もちろん個人でもこれらツールを自分のパソコンに用意することは可能ですが、 なるべく簡単に作業が始められるように これらのツールを全部まとめて収録したUSBメモリ用のデータを独自に用意しました （以下ではUSBメモリにまるごとコピーできるデータのことをイメージと呼びます）。 各自のUSBメモリにこのイメージをまだコピーしていない場合は、 担当の先生に相談してください。

このUSBメモリ用のイメージは、 **Debian Live**というシステムをマインドストームの 開発用にカスタマイズしたものです。 もともとパソコンに入っているWindowsなどの OS（オペーレーティング・システム）を起動せずに、 このUSBメモリからパソコンを起動させることで、 必要なツールを即使うことができます。 このイメージにはマインドストームのプログラム開発用のソフトだけでなくホームページ閲覧ソフト（ブラウザ）をはじめ グラフィックスや表計算などの種々のソフトウェアも収録してありますので ぜひ活用してください。

Note

このカスタマイズしたDebian Liveシステムには、 オフィス・スィートの LibreOffice、 高機能グラフィックス・ソフトウェアの Gimp や Inkscape なども収録されています。 また C、C++、Ruby, Python などのプログラミング環境も入っているので、 これらの言語でもすぐにプログラミング学習が始められます。 もちろん GNU/Linux の勉強にも適しています。

準備するもの

この講座で使う物を以下にまとめてあります。

- 貸し出しまたは配布する物
  - ロボットの貸し出しキット（説明書を含むマインドストームNXTおよび拡張セット）
  - 模造紙（配布）
  - この資料（配布）

- 各自で用意する物
  - 単三のアルカリ乾電池6本（同一の型、同時期購入のもの）
  - 黒い太いペン（コースを描くため）
  - この講座用のDebian Liveシステムが入ったUSBメモリ

参考資料について

この文書で説明している内容は、初めてマインドストームに触る人を対象にした、ごく入門的なものです。 少しマインドストームに慣れてきたら、いろいろな文献を参考に さらに高度なロボット製作に挑戦してみてください。 マインドストームに関する書籍も豊富に出版されていますが、 インターネット上には優れた文献が非常に豊富にあります。 もしインターネットにアクセスすることができるなら、 ぜひ自分でも検索してみてください。

ここでは、NXCの入門的ガイドとして最も定番で日本語訳もある次の文書を 紹介しておきます。 いずれもインターネットからPDFファイルをダウンロードできます。 以下の入門ガイドを読めばこの文書を読む必要がないくらい とてもわかりやすく説明されてます。 その他の参考資料は、 この文書の最後にいくつか紹介してありますので参考にしてください。

- **『NXCを使ったLEGOのNXTロボットのプログラミング』**
  Daniele Benedettelliさん製作『Programming LEGO NXT Robots using NXC』 のAlberto Palacios Pawlovskyさんによる日本語訳
  http://www.cc.toin.ac.jp/sc/palacios/courses/undergraduate/freshman/micro_intro/NXCtutorial_j.pdf
- **『NXCを使ったLEGOのNXTロボットのプログラミング』**
  上と同じDaniele Benedettelliさんのガイドの高本孝頼さんによる日本語訳（補足説明付き）
  http://www2.ocn.ne.jp/~takamoto/NXCprogramingguide.pdf

2. まずは簡単なロボットを組み立てよう

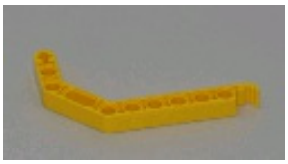
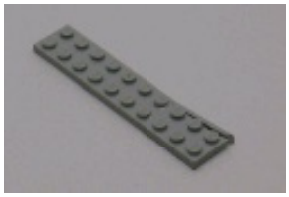
まずNXTに付属している説明書（インストラクション）に沿って ロボットを組み立てます。 が、**その前に次の注意事項をよく読んでください。**

キットの取り扱い上の注意

基本的な注意

コントローラ本体やモータ、センサ類は精密な電子部品ですので、 特に慎重に扱いましょう。 衝撃を与えたり無理な力を加えたり、というのは厳禁です。 プラスティックの部品は比較的丈夫ですが、 それでも無理な力をかけると当然壊れるので注意深く扱ってください。 また細かな部品もたくさんありますので無くさないように注意しましょう。

無理な力を加えると写真のように壊れます。


モータの扱いについて

モータに過度の負荷をかけてはいけません。 つまり力が足りなくてモータが回らないのに、無理に回そうとしてはいけません。 うまく回らない場合には、プログラムをすぐにストップさせましょう。 モータに負荷をかけ過ぎると、電池の消耗が激しくなったり、 モータそのものが壊れたりします。 力強く動かしたいときは、ギアを複数使って モータの回転数と最終出力（例えばタイヤ）の 回転数の比を調整してゆっくりと動かす必要があります。

コード類の扱いについて

コード類は絶対に強く引っ張ってはいけません。 またコードを抜くときは、断線を避けるために コード部分ではなく**必ず**端子部分をもってはずしましょう。

NXT付属のコードの端子は爪の部分を軽く押さえて抜きますが、 この爪は折れやすのでコードを抜くときや 絡まってしまったコードを解く場合には特に注意してください。 USBなどの端子を抜き差しするときは斜めにせずに**まっすぐ抜き差し**してください。

ゴム類について

ゴムの部品も何本かはいっていますが、とても切れやすいのでソフトに扱ってください。 無理に伸ばしたり、 ブロックの角に強く押し当ててしまうような使い方をしてはいけません。 プーリーで使う際にゴムをなるべく滑らないようにするには、ゴムを強く引っ張るのではなく、大きなプーリーを使ったり、 プーリーを２連にしたりしてください。

使用する電池について

使用する電池は必ず、同一の種類で同じ時期に購入したものを使います。 違う種類の電池や新旧違う電池を混ぜて使うと、発熱や液漏れの原因になり、 コンピュータ本体を壊してしまう可能性があります。

電池交換について

電源を切って電池交換をします。 RCX(RISのコントローラ)と違いNXTの場合は電池を抜いてもファームウェアは消えません。 電池の向きは交互にいれます。 それぞれの電池をまず電池のマイナス側をバネのようなっている端子に 当ててから、少し押さえてプラス側（突起になっている方）をはめ込みます。

プレート類の扱いについて

通常のブロックの1/3の厚さで板状の部品をプレート呼びます。 このプレートは同じ大きさのものをピタッと重ねてしまうと外すのに少し苦労します。 使わないときには重ねずしまうようにしましょう。 どうしても重ねる場合には、ポッチを一つずらして重ねておきましょう。 重なったプレートは大きなブロックに一度くっつけてからはずすと比較的用意にはずれます。

以上の注意を２回読みなおしてから次へ進みましょう。 特にコード類、ゴム部品の扱いには細心の注意を払ってください。

この講座で使用するNXT

この講座で使用するNXTの9797番については2006年発売の基本キット(v46)と 2011年発売の基本セット(v95)が混在していますが、それほど大きな違いはありません。 基本的にはバッテリがACからDCに変更になったのとL字の コネクタが6個増えたことです。 NXTのキットからは今回使用しないバッテリを抜いてあります。

また、コントローラにあらかじめ入れてあるファームウェアと呼ばれる ソフトウェアのバージョンは、enhanced NBC/NXC firmware 1.32です（配布時ファイル名は lms_arm_nbcnxc_132.rfw)。 これ以外のファームウェアも使用可能ですが、その場合には バージョンをコンパイル時のオプションとして指定する必要があります。

Note

NXTの教育用キット9797のv46とv95では実は本体の内部も変更になっています（いつ頃変更になったのかは不明)。 ウェブ上で故障が多数報告されている液晶の基盤 （おそらくハンダ付け不良） がCPU基盤に統合され、不具合が減ると期待されます。

では、さっそくロボットを組み立てましょう。

最初のロボットの組立て

まずはインストラクション（説明書）に従って、 モータを２個使って前後に進んだり左右に曲がるロボットを作ります。 インストラクションの製作例には、 LEGOのパーツを上手に使う技もけっこう含まれています。 特に「軽量」「丈夫」「機能ごとの部品に分解しやすい」を実現するための 組み立て方を常に意識しながら作業を進めてください。

LEGO Mindstorms NXTの組立例

前に触れたようにこの講座で使用する教育用キットにはv46（431パーツ）と v95（437パーツ）の２種類があります。

新しいv95を使用する場合には、 まず、インストラクションの2ページにある新旧の部品の対応表を確認します。 インストラクションは古い部品で説明されているの注意してください。 特に、インストラクションでは摩擦のある長さ3のピンの色が黒くなっていますが、 実際には青です。 これと似ている、黒いブッシュ付きの長さ3のピンと間違わないように。 この他、軸と軸をつなぐジョインター（黒）のデザインもイラストとは少し 変わっていますが、機能は同じです。

軸についてはそれぞれの組立図のそばに実寸大のものが描かれているので それを参考にしましょう。 RISと違って、偶数の長さのものは黒、奇数の長さのものは灰色、と色が 分かれているので間違いにくいはずです。

では、8ページから22ページまでを参考に組み立てましょう。

部品の収納方法

付属の図の通りに収納してください。

使っていない部品は、ケースのもとの位置にきちんと戻しておきましょう。 そうすることで、次にその部品を使うときにすぐに見つけることができます。 特にキットに慣れないうちは、組立て時間のうちのかなりの部分が 部品を探している時間であると気づくでしょう。

3. プログラムを作ってロボットを動かそう

プログラムって何？

ロボットを動かすための一連の命令を記述したものも **プログラム**と呼びます （一般にはコンピュータに計算させるための一連の 命令を記述したものをプログラムと呼びます）。 またプログラムを作成する作業のことを**プログラミング**と呼びます。

そしてプログラムを記述するための言葉が**プログラミング言語** と呼ばれているもので、世の中には非常に多くのプログラミング言語があります。 この講座では、**NXC**という マインドストーム専用に開発されたプログラミング言語を使ってプログラムを作成します。 これらの言語は、文法（プログラムを書く決まり）としてはC言語という プログラミング言語とよく似ていますが、 マインドストームのモータやセンサを扱う命令が豊富に用意されているので、 初心者でも簡単にプログラムを作成することができます。

Note

NXCは2006年にJohn Hansenさんによって開発が始められ、 アセンブリ・ライクな言語であるNBCとセットで開発・配布されています。 NXCはオープンソースで開発・配布されているため、誰でも自由に使用したり改良することができます。もちろん無料です。

プログラムを書いてみよう

プログラムを作成するためには**エディタ**と呼ばれるツール （ソフトウェア） を使います。 基本的には文字を入力することができて、いわゆる「テキスト形式のファイル」 として保存できるソフトであれば何でもかまいません。

この講座で使うライブUSBには「マウスパッド」というエディタが入っていますので これを起動します。 少しコンピュータに詳しい人向けに vi や emacs というエディタも使えるようになっています。

さっそくエディタを起動して、次のような簡単なプログラムを入力してみましょう。 これはロボットを3秒間前進させるためのプログラムです。

ただし、『 // モータBとCを前転 』のように // から行末までの部分は、ロボットへの命令ではなく プログラムの説明なので、 書いても書かなくてもロボットの動きに違いはありません。 しかし、他人がプログラムを見た時に理解しやすいように、 また、自分でどんなプログラムなのか忘れないためにも、 必要な箇所にはちゃんと説明を書いておきましょう。 これらの説明箇所は**コメント文**と呼ばれています。 コメント文だけは日本語で書いても大丈夫です。 また、コメント文を書くには、この方法以外に /* と */ で 囲む方法もあります。 この方法を使えば // を使う場合と違って 複数行にまたがってコメントを書くことができます。

NXCプログラム

```
task main ()
{
  OnFwd(OUT_BC,75); // モータBとCを75%のスピードで前転
  Wait(3000);       // 次の命令に進むまで3秒待つ（単位は 1/1000秒）
  Off(OUT_BC);      // モータBとCを停止
}
```

【例】3秒間前進するプログラム

NXCでは一連の動作（タスク）を task というキーワードで指定します。 タスクの名前は自分でつけることができますが、 必ず一つ main という名のタスクがなければいけません。 そして２つ以上の task がある場合でも プログラムは main タスクからスタートして、 他のタスクを呼び出すようにします。

タスクの中身は { と } で囲みます。 また、一つひとつの命令の後には ; を忘れないように。

少し命令を説明しておきましょう。 モータを前転（正回転）するには OnFwd という命令をつかいます。 スイッチオンを意味する On と前へという意味の Forward を短縮してたし合わせた名前です。

また、この例からわかるように、制御したいモータの名前は接続したポート（端子）の 名前に合わせて OUT_BC のように表します（以下の表を参考に）。 OUTというのはこの場合、**出力**（英語でアウトプット output）という意味です。 NXTのインストラクションでは右のモータがB端子に、左のモータがC端子に接続されているので、 これら例ではそれらにあわせて出力を指定しています。 もちろん違う端子を使う場合には適宜名前を変えてください。

NXCの OnFwd では、 モータの指定だけでなく回転スピードも , （コンマ）につづけて 指定しなければいけないことに注意してください。

Note

モータは力を出すので確かに**出力**ですが、 一般にコンピュータから出てくる情報のことを**出力**といいます。 出力は、モータを回す以外にも、画面への出力であったり、プリンタへの出力であったり、 スピーカーへの出力であったりさまざまです。 これに対しコンピュータに与える指示や情報のことを**入力**といいます。 キーボードやマウスは入力装置です。

出力名	モータ
OUT_A	端子Aにつないだモータ
OUT_B	端子B 〃
OUT_C	端子C 〃
OUT_AB	端子Aと端子B 〃
OUT_BC	端子Bと端子C 〃
OUT_AC	端子Aと端子C 〃
OUT_ABC	全部の端子 〃

さて次に出てきた Wait というのは、次の命令に進むまで待つ、 という命令です。英語のWaitそのままですね。 **1/1000秒の単位**で指定します。

ところで上の例のように、どこからどこまでが { と } の間の部分なのかを 見やすくするために空白を入れて字下げすることがよくあります。 これはC言語などのプログラミング言語でも同じです （勝手に空白や改行を入れることが禁止されている言語もあります）。 命令と命令の間に空白や改行はいくつあってもかまいませんが、 全角の空白はダメなので気をつけましょう。

NXCでは改行せずに次のように一行で書くこともできますが、 プログラムを読みやすくするために適切に改行や字下げを行いましょう。

NXCプログラム

```
task main (){ OnFwd(OUT_BC,75); Wait(3000); Off(OUT_BC);}
```

【例】一行で書いたプログラム

プログラムを入力できたら、ファイル名をつけて保存しましょう。 ファイル名は余計なトラブルを避けるために、 myrobo1.nxc のように半角のアルファベットと数字、 マイナス記号(-)、アンダースコア(_)だけを使ってつけましょう。 NXCのプログラムの後ろには、 .nxc をつけておくと何かと便利です。 このようにファイル名の後ろにつけてファイルの種類を表す文字の並びは **拡張子**と呼ばれています。

ところで上のプログラムは、 次のようにモータの命令を一つずつ書いても同じです。 というのは、コンピュータは非常に高速で命令を処理するので、 ２つの命令に分けて書いても実質的に同時の命令となります。 命令を遅らせるには、意図的に Wait で時間を指定してしなければいけません。

NXCプログラム

```
task main ()
{
  OnFwd(OUT_B,75); // モータBを75%のスピードで前転
  OnFwd(OUT_C,75); // モータCを75%のスピードで前転
  Wait(3000);
  Off(OUT_B);
  Off(OUT_C);
}
```

【例】3秒間前進する別のプログラム

プログラムをロボットに転送しよう

プログラムを保存することができたら、早速ロボットに転送してみましょう。

付属のUSBケーブルでロボットとパソコンを接続します。 NXTの電源が入っているか確認してください。

ライブUSBを使っている場合、基本的には２種類の転送方法があります。 一つ目は、ファイルマネージャ（ファイル一覧を表示するソフト）を起動して 作成したファイルのアイコンを右クリックして「NXTに転送」を選ぶ方法です。 おそらくこれがもっとも簡単でしょう。

２つ目は、シェルと呼ばれる、コマンドを入力するためのソフトを起動して、 次のように命令を直接入力する方法です。 シェルを起動すると、次のような表示になります。 パソコンの設定によっても違いますが、 表示されている文字は**プロンプト**とよばれ、 ユーザからの命令待ちの状態を示しています。

Note

コマンド入力では、特に指定しなくてもUSBポートを使うように設定してあります。 いろいろなオプションについては、 man nbc などで調べることができます。

```
~$
```

この状態で、次のように入力します。

```
~$ nbc -v=132 -EF -d myrobo1.nxc
```

myrobo1.nxc のところは各自のファイル名に変更してください。 ここで -v=132 はファームウェアのバージョンが1.32であること、 -EF はenhanced firmwareであることを指定しています。

ここでプログラムを「転送する」と書きましたが、 正確に言うと実際には我々が作成したプログラムを、（1）コンピュータ（コントローラ）の中に入っている**ファームウェア**が 解釈できる形に変換した上で、（2）転送するという、２段階のことを続けて行なってくれています。

このように人間が書いたプログラムをコンピュータがより理解しやすい形式に 翻訳（変換）することを**コンパイル**と呼びます。

うまく転送できない場合には

プログラムに文法的なエラーがあれば、エラーメッセージが表示されて プログラムは転送されません。 大文字、小文字を含めて命令のスペルがあっているか、 括弧を忘れていないか、セミコロンを忘れていないか、 などチェックしてみましょう。プログラムを訂正したら保存して再度転送してみましょう。

以下によくありがちな間違いを挙げておきます。

- 括弧の始まりと終わり、つまり｛と｝や（と）がきちんと対応していますか？
- 全角の文字は使っていませんか？ 特に全角の空白やセミコロン（；）が無いかチェックしてください。
- OnFwd は O と F だけが大文字になっていますか？
- 同様に Wait の W だけが、 Off の O だけが大文字になっていますか？
- OUT_AC や OUT_BC は全部大文字になっていますか？
- OnFwd のスピードをちゃんと指定してありますか？
- NXTの電源は入っていますか？ USBケーブルは正しくつながっていますか？

また、シェルで転送しようとしている場合は、 作成したプログラムがちゃんと存在しているか

```
~$ ls
```

というコマンドで確認しましょう。

さあ！うまく動くかな？

無事エラーもなく転送できたら、ロボットを動かしてみましょう。

横方向の矢印を動かして 「My Files」を選び（オレンジボタンで決定）、 さらに「Software files」を選びます。 そして自分が転送したファイル名を選んだ後、 Run を選べば動き始めるはずです。 一つ前のメニューに戻したり、プログラムを止める場合は下のボタンを押します。

うまく動かない場合には、モータがプログラムで指定した正しい端子に つながっているか、ケーブルがきちんとはまっているかなど、 確認してみましょう。

ところで、これからいろいろな練習問題が出題されますが、 ２問セットになっている問題の場合は、どちらか一方、 または両方挑戦してみてください。 ２人１組で作業している場合にはお互いに別の問題を選びましょう。

練習問題1	1.5秒間前進して1秒間停止、さらに2秒間前進するようにプログラムを改造しましょう。
練習問題2	1秒間前進して1.5秒間停止、というのを3回繰り返すようにプログラムを改造しましょう。

バックや旋回もさせてみよう

モータを後転させるには、 OnRev という命令を使います。

NXCプログラム

```
task main ()
{
  OnRev(OUT_BC,75); // モータAとCを後転、OnFwd(OUT_BC,-75)と同じ
  Wait(3000);
  Float(OUT_BC); // ゆっくり自然に止まる
}
```

【例】3秒間後進するプログラム

ここで、 Float という命令が出てきましたが、 これはモータに力をかけないようにする命令です。 つまり回す力も止める力もかかっていない、「浮いた」状態です。 これに対し、前に出てきた Off という命令は 実はモータを止めると同時に止まった状態に保つ、という命令でした。 もちろん無理やり回せば回ることは回りますが、 回すのにかなり抵抗があることがわかります。

さて、片方のモータだけを動かすことで、左右に曲がることもできます。 また、左右のモータを反対方向に動かすことで、その場で旋回させることもできます。

練習問題3	その場でロボットを時計回りに20秒間旋回させるプログラムを作って、 何周旋回するか数えましょう。
練習問題4	その場でロボットを半時計回りに20秒間旋回させるプログラムを作って、 何周旋回するか数えましょう。

上の練習問題の結果をもとにロボットが１周（360度）旋回するのに必要なおおよその時間を 計算することができます。これを使って次の練習問題に進みましょう。

練習問題5	その場でロボットを時計回りに２回転させるプログラムを作ってみましょう。
練習問題6	その場でロボットを反時計回りに２回転させるプログラムを作ってみましょう。

前進と旋回ができれば、次の練習問題のようなプログラムも簡単に作ることができます。

練習問題7	30cm前進した後Uターンしてもとの場所に戻ってくるようプログラムを改造しましょう。
練習問題8	30cm前進した後、１回転半してもとの場所に戻ってくるようプログラムを改造しましょう。

さてここまでの練習で、実際にプログラムを作ってロボットを 動かす、という一連の作業に慣れたでしょうか？

4. プログラムをかっこよくスリムに

時間調整はプログラムの始めに

前の練習問題では、左右のモータを動かす時間をうまく調整する必要がありました。 もちろん毎回 Wait の中の時間を編集して 調整することもできるのですが、 あらかじめプログラムの冒頭で名前をつけた『**定数**』 として指定しておくととても便利です。 特に次の例のように同じ時間が何度か登場するような場合には、 一箇所編集するだけで調整が可能になります。 また、たとえ何度も登場しなくても定数を使えば調整箇所をプログラムの 途中から探しだす必要がなく、 名前をつけることでどのような意味のある数なのか一目でわかります。

以下は2秒間前進したあと2秒間後進するプログラム例です。 このように #define に続けて『定数名』と『数値』を空白で 区切って書くだけで定数を定義することができます。 ちなみにdefineは英語で「定義する」といいう意味です。 NXTの例でもわかるように、時間以外でもプログラムが終了するまで 変更しない数値ならなんでも定義しておくことができます。

Note

#define で定義された定数は、コンパイル時に元の数に置き換えられます。

#define 行は基本的に一行で書く、ということに注意してください。 長くなって２行以上で書きたい場合には次の行への継続を示す＼（バックスラッシュ）が行の終わりに必要です。 この＼の後には何も書いては行けません。 この文書ではスペースの関係で＼を使って行を継続している箇所がありますが、 なるべく一行で書くようにしましょう。

NXCプログラム

```
#define MOVE_TIME 2000  // 定数を定義
#define SPEED 75        // 別の定数を定義

task main()
{
  OnFwd(OUT_BC,SPEED); // SPEEDは自動的に75に置き換えられる
  Wait(MOVE_TIME);     // MOVE_TIMEは自動的に2000に置き換えられる
  OnRev(OUT_BC,SPEED);
  Wait(MOVE_TIME);
  Off(OUT_BC);
}
```

【例】定数を使ったプログラム（2秒前進の後、2秒後進）

定数名（定数の名前）は各自で決めることができますが、アルファベット と数字、 _（アンダースコア）だけを使ってつけましょう。 ただし数字は定数名の最初に来てはいけません。 つまり LE5 のようは定数名はOKですが 5LE はダメです。 空白や他の記号は使えないので注意しましょう。 定数名は慣習的にすべて大文字にすることが多いようです。 また、NQCやNXCであらかじめ定義されているコマンド名や 定数名などの、いわゆる『**予約語**』は使用できません。 例えば task 、 OnFwd 、 OUT_A 、 start 、 stop などを定数名として使用することはできません。

練習問題9	今までに作ったプログラムを #define をつかって改良しましょう。

決まった動作にも名前をつけておこう

実は数だけでなく、一連の命令も冒頭で定義しておくことができます。 例えば「左折する」というコマンドは、 もとのNXCでは用意されていませんが、定数と同様に #define を使って自分で作ることができます。 この場合は、定数と呼ばずに**マクロ**と呼びます。

NXCプログラム

```
#define TURN_TIME 800  // 左折の時間を定義
#define SPEED 75
// turn_left という名のマクロを定義
#define turn_left Off(OUT_C); OnFwd(OUT_B,SPEED);\
  Wait(TURN_TIME); Off(OUT_B);

task main ()
{
  OnFwd(OUT_BC,SPEED); Wait(1000);
  turn_left;
  OnFwd(OUT_BC,SPEED); Wait(2000);
  turn_left;
  OnFwd(OUT_BC,SPEED); Wait(3000);
  Off(OUT_BC);
}
```

【例】マクロを使ったプログラム

マクロも原則一行で定義する必要がありますが、長くなる時には＼（バックスラッシュ）を使って行を継続させることができます。

このように turn_left という自分で定義した命令（マクロ）を 使うことよって task の中が簡潔になっているのがわかります。さらにこのプログラムは「直進」の命令をマクロとして定義することで より簡潔になります。

直進用のマクロはいろいろな直進時間に対応できるようにしてみましょう。 そのためには、マクロ名の後ろに( )をつけて、その中に 直進時間を与えるための変数を書きます。 次の例では t という変数が使われていますが、 これは定義内で Wait(t) の t として使われます。 もちろん同じであれば t 以外の文字を使ってもかまいません。 数学で登場する f(t) = 2t + 4 ような関数の記法を思い出しましょう。

NXCプログラム

```
#define TURN_TIME 800
#define SPEED 75
#define turn_left Off(OUT_C); OnFwd(OUT_B,SPEED); \
  Wait(TURN_TIME); Off(OUT_B);
// 時間tとスピードsの2つの引数を取るマクロ
#define go_forward(t,s) OnFwd(OUT_BC,s); Wait(t); Off(OUT_BC);

task main ()
{
  go_forward(1000,75); // 75%のスピードで1秒間前進
  turn_left;
  go_forward(2000,50); // 50%のスピードで2秒間前進
  turn_left;
  go_forward(3000,100);// 100%のスピードで3秒間前進
  Off(OUT_BC);
}
```

【例】引数を取るマクロの例

この t のように、マクロなどを呼び出すときに指定する変数 （マクロなどに渡してあげる変数）のことを **『引数（ひきすう）』**と呼びます。 ちなみに、「引（ひき）」は訓読み、「数（すう）」は音読みなので、 こういう読み方のことを湯桶（ゆとう）読みと呼びます。 重箱（じゅうばこ）読みの逆ですね。

上の例で定義されているように、 時間 t とスピード s のように２つ以上の 引数を使いたい場合には、 go_forward(t,s)のように 引数を ,(コンマ)で区切って定義します。 このマクロを呼び出す際には、時間 t とスピード s の順序を間違えてはいけません。

練習問題10	上の例を参考に、 2秒前進、右折、1秒前進、左折、1秒前進、左折、1秒前進、右折、2秒前進、停止、 という動作をさせるプログラムを作りましょう。

練習問題11	上の例を参考に、 1秒前進、左折、2秒前進、右折、2秒前進、右折、2秒前進、左折、1秒前進、停止、 という動作をさせるプログラムを作りましょう。

マクロは、このように基本的に一行で書くことのできる比較的短い命令を 定義するのに便利です。一方、もう少し複雑な一連の動作の定義には、 後ほど説明する、関数やサブルーチンあるいはタスクを使います。

繰り返しはとても簡単

繰り返しはコンピュータが得意とするところです。 英語で繰り返しを意味する repeat という命令を使えば、 例えば次のような「直進して左折」を４回繰り返すプログラムが 簡単にできます。

NXCプログラム

```
#define TURN_TIME 800
#define SPEED 75
#define turn_left Off(OUT_C); OnFwd(OUT_B,SPEED);\
  Wait(TURN_TIME); Off(OUT_B);
#define go_forward(t) OnFwd(OUT_BC,SPEED);Wait(t);Off(OUT_BC);

task main ()
{
  repeat(4) {  // {}内を4回繰り返す
    go_forward(1000);
    turn_left;
  }
}
```

【例】繰り返しは簡単に書ける

以上のように、マクロや repeat を使うことで、 main タスクの中のプログラムの流れが （コメント文で説明しなくてもよいくらい） 非常にシンプルでわかりやすいものになりました。 特にプログラムが長くなるときには、 同じ命令をダラダラと繰り返して書かないように常に注意しましょう。

練習問題12	一辺が30cmの五角形を描いて動くロボットのプログラムを作りましょう。
練習問題13	一辺が20cmの八角形を描いて動くロボットのプログラムを作りましょう。

次の例のように repeat のブロックを別の repeat に入れる ことができます。 このように、ある構造の中に同じような構造がある構造を **入れ子構造**と呼びます。

NXCプログラム

```
#define TURN_TIME 800
#define SPEED 75
#define turn_left Off(OUT_C); OnFwd(OUT_B,SPEED);\
  Wait(TURN_TIME); Off(OUT_B);
#define go_forward(t) OnFwd(OUT_BC,SPEED); Wait(t); Off(OUT_BC);

task main ()
{
  repeat(2) {
    repeat(4) {
      go_forward(1000);
      turn_left;
    }
  }
}
```

【例】 repeat を入れ子で使ったプログラム

練習問題14	一辺が30cmの三角形を4回描いて動くロボットのプログラムを作りましょう。
練習問題15	一辺が20cmの六角形を4回描いて動くロボットのプログラムを作りましょう。

少しずつ動かす時間を変化させるには

前の例では「決められた時間を動かすためには定数を使えば便利である」 ということを学習しました。 では、動かす時間を少しずつ変化させたい場合に何かいい方法はあるでしょうか？ もちろんあります。 そういう場合には**変数**と呼ばれるものを

使います。 数学で学習したｘやｙといった変数と同じように、 プログラムで使う変数にもいろいろな値を代入することができます。 「変数は値を格納する箱」という説明がしばしばなされます。

例として、一往復する度に往復の時間（距離）を伸ばしていくロボットを考えてみましょう。 一回目の往復は片道1秒、2回目は片道2秒、3回目は片道3秒、というふうに 進む時間を増やして行きます。 往復回数が少なければ次のプログラムのように命令を並べればよいだけなので簡単です。

NXCプログラム

```
#define SPEED 75
#define go_and_back(t) OnFwd(OUT_BC,SPEED); Wait(t);\
  OnRev(OUT_BC,SPEED); Wait(t); Off(OUT_BC);

task main ()
{
  go_and_back(1000);
  go_and_back(2000);
  go_and_back(3000);
}
```

【例】 一往復ごとに往復の時間を伸ばしていくプログラム

ところが繰り返し回数が多くなると命令を並べて書くのは大変です。 そこで変数の登場です。

まずは変数を使って上のプログラムを書きなおしてみましょう。 以下では、 move_time と名付けた変数を使います。 変数名は定数名と同じく、予約語以外で、 アルファベット、数字、 _（アンダースコア）を使って作ります。ただし、最初の文字に数字を使ってはいけません。

変数には、整数型、小数型、文字列型などいろいろありますが、 NQCで使用できるのは整数型のみです。 整数型の変数を宣言する際には、 int というキーワードを 使います。ちなみに int は整数を表す英語、integerに由来します。

また変数の宣言と同時に最初の値（初期値）を代入しておくこともできます。

Note

変数の宣言を task main の外で行った場合には、 その変数はグローバル変数（大域変数）と呼ばれプログラムの どこでも使用できます。 これに対し、例のように、あるタスクの中で宣言した場合は、 ローカル変数（局所変数）と呼ばれ、そのタスクの中だけ使用できます。

Note

NXCでは、通常の整数型に加え、 桁数の多い（32bitの）整数型 long 、 浮動小数点型 float 、 論理型 bool 、 文字型 char 、 文字列型 string 、 なども使えます。

NXCプログラム

```
#define SPEED 75
#define go_and_back(t) OnFwd(OUT_BC,SPEED); Wait(t); \
  OnRev(OUT_BC,SPEED); Wait(t); Off(OUT_BC);

task main ()
{ /* 整数型の変数 move_time を使うことを
     宣言して、その初期値を0にする */
  int move_time=0;

  move_time += 1000; //  move_timeに1000を加える
  go_and_back(move_time);

  move_time += 1000; //  move_timeの値は2000になる
  go_and_back(move_time);

  move_time += 1000; //  move_timeの値は3000になる
  go_and_back(move_time);
}
```

【例】 変数を使ったプログラム

move_time = 0; は **左辺の変数に右辺の値を代入する**（この場合 move_time に 0 を代入する）という命令で、 = という記号は代入演算子と呼ばれます。

次の命令に出てくる、 += は、左辺の変数を右辺の分だけ増やす、 という記号（演算子）です。 これと同じ意味で次のような表記も使うことができます。

```
move_time = move_time + 1000;
```

この式を見て驚く人もいるかもしれません。 左辺と右辺は等しくないのに =（等号）で結びついている！、と。 でも上で説明したことを思い出してください。 この場合、 = という記号は「等しい」という 意味ではなくて、**右辺の式の値を左辺の変数に代入する**という意味です。 つまりこの命令は、 move_time の値に 100 を足したものをあらためて move_time に代入する、という意味です。 もし仮に move_time に 2000 という値が 入っていたとすると、 この命令を実行した後には move_time には 3000 という値が入っていることになります。

次の表は、数の計算でよく使う記号（算術演算子）をまとめたものです。 + や - は数学の記号と同じですね。

整数の演算

表記	意味
x + y	x と y を加えた値
x - y	x から y を引いた値
x * y	x と y をかけた値
x / y	x を y で割った値
x % y	x を y で割った余り
x += y	x を y だけ増やす（x = x + y と同じ）
x -= y	x を y だけ減らす（x = x - y と同じ）
x *= y	x を y 倍する（x = x * y と同じ）
x /= y	x を 1/y 倍する（x = x / y と同じ）
x++	x を 1 増やす（x += 1 と同じ）
x--	x を 1 減らす（x -= 1 と同じ）

さて、プログラムの書き換えに戻りましょう。 ここまでくれば、あとは簡単です。 すでに学習した repeat を使えば短いプログラムになります。

NXCプログラム

```
#define SPEED 75
#define go_and_back(t) OnFwd(OUT_BC,SPEED); Wait(t); \
  OnRev(OUT_BC,SPEED); Wait(t); Off(OUT_BC);
task main ()
{
  int move_time = 0;

  repeat(3){
    move_time += 1000;
    go_and_back(move_time);
  }
}
```

【例】 repeat を使って簡単にしたプログラム（繰り返し回数を増やすのは簡単）

プログラムがこういう形になれば繰り返しの回数を増やすのはとても簡単ですね。

練習問題16	上の例を参考に、最初は4秒間前進、その後3.5秒間後進、3秒間前進、2.5秒間後進、 ・・・0.5秒間後進、最後は停止、と、0.5秒づつ動く時間を減らしながら最後は止まる ロボットのプログラムを作りましょう。
練習問題17	最初は4秒間前進の後、4秒間後進、次はその半分の時間の2秒間前進・2秒間後進、 と動く時間を半分ずつに減らして、最終的に、0.25秒間前進ののち後進したあと 停止するロボットのプログラムを作りましょう。

5. ロボットに触覚をつけよう

これまでの練習で、ロボットを動かすための基本的なプログラムの書き方が 理解できたでしょうか？ さて、次はロボットに触覚（バンバー）をつけて、何かに触ったり当たったりしたら 動きを変えるようにしてみましょう。

タッチセンサの取り付け

そのためには タッチセンサ と呼ばれるセンサを使います。 簡単な製作例がインストラクションにありますので、 まずはそのままバンパーを作ってみましょう。 インストラクションの40ページから44ページを参考に。

インストラクションでは、 ロボットの後ろにタッチセンサを装着するようになっていますが、 前にも装着できます。以下の説明では前に装着したことを前提にします。 後ろに装着した場合は、 OnFwd を OnRev に 置き換えるなどの変更をすれば、同じように動くはずです。

障害物にあたると止まるロボット

簡単な例として、障害物にあたると止まるプログラムを作ってみましょう。

NXCプログラム

```
task main()
{
  SetSensorTouch(S1);   // 端子1にはタッチセンサ

  OnFwd(OUT_BC,75);     // 前進
  until (SENSOR_1 == 1); // センサが押されるまで待つ
  Off(OUT_BC);          // 停止
}
```

【例】障害物にあたると止まるプログラム

センサを使うためには、まず、どのようなタイプのセンサがどの端子（ポート）に 接続されているのか指定する必要があります。 タッチセンサを使う場合、上の例のように、 SetSensorTouch という命令を使います。 1番目ではなく2番目のポートにタッチセンサをつないだときには、 S1 の代わりに S2 を指定してください。

Note

SetSensor(S1,SENSOR_TOUCH); という命令は使えますが、続けて ResetSensor(S1); という命令を実行する必要があります。

Note

SENSOR_1 は端子１のセンサの値を得るマクロで、Sensor(S1) と定義されています。

さて、上の例で登場した until というのが、センサの状況を 認識するための一つの命令です。 この命令は、( )の中の 条件が満たされるまでずっと待つ（次の命令に進まない）という意味です。 ちなみにuntilは英語で、時間的に「～まで」という意味です。

この ( ) の中にある SENSOR_1==1 が 条件を表しているわけですが、 このような条件のことを命題と言ったり 論理式 と言ったりします。 **論理式の特徴は true （真） と false （偽） の ２つの値のうちどちらか一方しか取らない**、ということです。

上の例ではタッチセンサ自身の値は、 SENSOR_1 で得られ、 押されていれば１，押されていなければ 0 です。 重要なのは = を２つ並べた == という記号で、 この記号の両辺（左右）の値を比較するためのものです。 そして == の両辺、この場合は SENSOR_1 と１、 が等しければ論理式 SENSOR_1==1 の値は true 、 等しくなければ論理式 SENSOR_1==1 の値は false になります。

この == のように値を比較するための記号を**比較演算子** と呼びます。以下の表によく使う比較演算子をまとめてあります。

比較演算子

記号	意味
==	等しい
<	小さい
<=	小さいか等しい
>	大きい
>=	大きいか等しい
!=	等しくない

練習問題18	条件式 1 + 2 == 3 の値は？（ true と false のどちら？）
練習問題19	条件式 2 / 3 == 0 の値は？（ true と false のどちら？）
練習問題20	タッチセンサ（ SENSOR_1 ）が押されている時、 条件式 SENSOR_1==0 の値は？
練習問題21	タッチセンサ（ SENSOR_1 ）が押されていない時、 条件式 SENSOR_1==1 の値は？

上のプログラムが理解できたら、次の練習問題に進みましょう。

練習問題22	障害物に当たるとUターンして進み、もう一度、障害物に当たると止まるようにプログラムを改良してみましょう。
練習問題23	障害物に当たると停止し、障害物を取り除くと2秒後にその場で180度旋回するようにプログラムを改良してみましょう。

障害物に何度当たっても動きつづけるように

上の例の、障害物に当たったら止まるプログラムを別の方法で書いてみましょう。 英語で「～している間」を意味する while を使っても 簡単にかけます。 while(論理式){命令} のように書くだけで、 論理式 が true の間は {命令} を繰り返します。

NXCプログラム

```
task main()
{
  SetSensorTouch(S1);
  OnFwd(OUT_BC,75);
  while(SENSOR_1==0){} // SENSOR_1が1の間繰り返す
  Off(OUT_BC);
}
```

【例】 while を使ったプログラム

この例の場合は、 OnFwd であらかじめ左右のモータを回しているので、 while の命令で何もする必要がなく、 {} の中は空っぽになっています。 もちろん最初の OnFwd を while の中に入れて while(SENSOR_1==0){OnFwd(OUT_BC,75);} のように書いても結果としては同じ動きをします。

Note

ある論理式を値を反対にするためには！という記号を論理式の前につけますが、この記号を使えば、 until(論理式); と while(!(論理式)){} が同じであるということに気づいたでしょうか？

この while には、とても便利な応用があります。 それは、論理式を常に真 true にして永遠に繰り返しをさせてしまう、 いわゆる 無限ループ です。

NXCプログラム

```
#define SPEED 75
#define go_forward OnFwd(OUT_BC,SPEED);
#define u_turn OnRev(OUT_BC,SPEED); Wait(500);\
  OnFwd(OUT_C,SPEED); Wait(1000); Off(OUT_BC);

task main()
{
  SetSensorTouch(S1);

  while (true) {   // 永遠に繰り返し
    go_forward;
    until (SENSOR_1==1);  // 接触すると次に進む
    u_turn;
  }
}
```

【例】障害物にあたるとUターンするプログラム

この例では、 while の中に 最初から true という論理式の値（定数）を指定していますが、 もちろん while (1+2==3) や while (4!=5) のように常に true になる 論理式であれば、どのように書いても同じことです。 が、最初から true を使うのが最も簡単ですね。

障害物に当たってUターンする回数を制限したい

上の例は、障害物に当たってもUターンして永遠に動き続けるプログラムでしたが、 一定の回数障害物に当たったら止まるように改良してみましょう。 そのためには、当たった回数を数えるための変数を使い、 当たる度に変数の値を増やしていきます。

今回は while (論理式) { 命令 … } の代わりに do { 命令 … } while (論理式); を使ってみましょう。 この方法は、先ほどの while の繰り返しと違って、 命令を実行した後で条件(論理式)を判定して、 true ならもとに戻って繰り返します。 つまり、最低 1 回は命令を実行するわけです。

NXCプログラム

```
#define SPEED 75
#define go_forward OnFwd(OUT_BC,SPEED);
#define u_turn OnRev(OUT_BC,SPEED); Wait(500);\
  OnFwd(OUT_C,SPEED); Wait(1000); Off(OUT_BC);

task main()
{
  SetSensorTouch(S1);
  int n=0;  // 衝突回数を数える変数 (初期値を0に)

  do {   // { } 内の命令を実行
    go_forward;
    until (SENSOR_1==1);
    u_turn;
    n++;          // Uターンする度にnを1増やす
  } while (n < 5);  // nが5未満なら繰り返す
}
```

【例】5回Uターンしたら停止するプログラム

このように、回数などを数えるための変数を **カウンタ** と呼びます。

練習問題24	上の例を、 while (論理式) { 命令 … } を使って書き直しましょう。

「もし～なら」という条件で命令を実行

これまでの例で、条件をする方法として until、 while 、 do～while を使うことを学習しました。 ここでもう一つ重要な条件指定の方法を学習しましょう。 それは、いろいろなプログラミング言語においてもよく使われる if を使う方法です。 英語の「if ～」は「もし～ならば」という意味です。

NXCプログラム

```
#define SPEED 75
#define go_forward OnFwd(OUT_BC,SPEED);
#define u_turn OnRev(OUT_BC,SPEED); Wait(500);\
  OnFwd(OUT_C,SPEED); Wait(1000); Off(OUT_C);

task main()
{
```

```
  SetSensorTouch(S1);

  while (true) {
    go_forward;

    if (SENSOR_1 == 1) {  // 接触すれば
      u_turn;             // Uターン
    }
  }
}
```

【例】 if を使ったプログラム

if の論理式が false の場合には、 何も実行せずに次に進みます。

左右のバンパーを使う

次に、左右に２個のタッチセンサをつけて両方のセンサを使ってみましょう。 インストラクションに載っていないので タッチセンサを一つ使った場合を参考に自分で考えてみましょう。

まずはどちらか一方のセンサが押されたら ロボットが3秒間止まるようなプログラムを考えてみましょう。 これは if ブロックを２つ並べることで簡単にできます。

NXCプログラム

```
#define wait3sec Off(OUT_BC);Wait(3000); //3秒停止

task main()
{
  SetSensorTouch(S1);
  SetSensorTouch(S3);

  while(true){
    OnFwd(OUT_BC,75);
    if (SENSOR_1==1) { wait3sec; }
    if (SENSOR_3==1) { wait3sec; }
  }
}
```

【例】 どちらか一方のタッチセンサが押されたらロボット3秒間止まるプログラム

このように２つ if ブロックを並べて実現することも可能ですが、 実行する同じ命令（この場合 wait3sec ）を繰り返し記述するのは面倒です。 そこでもう少し便利な方法を紹介しましょう。 それは、 ２つの論理式 SENSOR_1==1 と SENSOR_3==1 のどちらかが成り立っていれば真(true) となるように**論理式を組み合わせる**方法です。 これは２つの論理式を || という記号でつなぐだけで実現できます。 このような論理式の演算を行う記号を**論理演算子**といいます。

NXCプログラム

```
#define wait3sec Off(OUT_BC); Wait(3000);

task main()
{
  SetSensorTouch(S1);
  SetSensorTouch(S3);

  while(true){
    OnFwd(OUT_BC,75);

    if ((SENSOR_1==1)||(SENSOR_3==1)) {  // 論理和を使った条件式
      wait3sec;
    }
  }
}
```

【例】 ２つの論理式を || で組み合わせた論理式

このように２つの論理式のどちらか一方でも true であれば、 組み合わせた論理式も true である、というような組み合わせ方を 論理和 と呼びます。 逆に言えば、２つの論理式の論理和が false になるのは、 両方の論理式が false の時のみです。

下記の表は論理式 A と B が それぞれ true または false のとき論理和 （ A || B ）の値がどうなる表した表です。 このような true と false の値の表のことを **真理値表**といいます。

論理式 A と論理式 B の論理和
A||B

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

ついでに、 同時に両方のセンサが押された時に限ってロボットが止まるようなプログラム を作ってみましょう。 この場合は、２つの論理式を && でつなぎます。 そして、このように２つの論理式のどちらも true であるときに限り true である、というような組み合わせ方を 論理積 と呼びます。

NXCプログラム

```
#define wait3sec Off(OUT_BC); Wait(3000);

task main()
{
  SetSensorTouch(S1);
  SetSensorTouch(S3);

  while(true){
    OnFwd(OUT_BC,75);

    if ((SENSOR_1==1)&&(SENSOR_3==1)) { // 論理積を使った条件式
      wait3sec;
    }
  }
}
```

【例】 ２つの論理式を && で組み合わせた論理式

論理積についても真理値表にまとめておきましょう。

論理式 A と論理式 B の論理積
A&&B

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

ちなみに && を使った論理積を使わない場合、 次のように if ブロックのなかに別の if ブロックをいれる、つまり if を**入れ子**にすることで同じ動作をさせる、つまり論理積を組み立てることができます。

NXCプログラム

```
#define wait3sec Off(OUT_BC); Wait(3000);

task main()
{
  SetSensorTouch(S1);
  SetSensorTouch(S3);

  while(true){
    OnFwd(OUT_BC,75);
    /* if の入れ子構造 */
    if (SENSOR_1==1) {
      if (SENSOR_3==1)) {
        wait3sec;
      }
    }
  }
}
```

【例】 if ブロックを入れ子にして２つの条件を組み合わせる

練習問題25	論理式 (1+2==3)||(3-1==1) の値は？

練習問題26	論理式 (4/3>1)&&(4/2==2) の値は？
練習問題27	SENSOR_1 と SENSOR_3 がともに押されている時、 (SENSOR_1==0)||(SENSOR_3==1) の値は？
練習問題28	SENSOR_1 が押され SENSOR_3 が押されていない時、 (SENSOR_1!=0)&&(SENSOR_3!=1) の値は？

上の練習問題には、いじわる問題が混ざっているかもしれません。

練習問題29	左のバンパーが壁に接触したときは右折、右のバンパーが壁に接触したときは左折 するようなロボットを作りましょう。
練習問題30	机の上の落ちずに動きまわるロボットを作りましょう。

6. 時間を測ろう

以前の例では、 障害物に当たったらUターンを繰り返す、というプログラムを作りました。 このプログラムでは while(true){ } を利用して、 永久的に動作を繰り返すようにしました。 では一定時間だけそのような動きをさせたい場合はどうすればよいでしょう？

これまで時間の指定というと Wait を使って次の命令までの時間を コントロールしていたわけですが、今回のような目的には Wait は 使えません。なぜなら Wait で待っている間は、触れたかどうかの 判断すら行えないからです。

こういう場合、便利なのがストップウォッチのように時間を測るタイマー機能です。 タイマーを使えば「一定の時間内に限って」という条件のもとで繰り返し動作を 簡単に行うことができます。

基本的には、ある基準となる時刻を記憶しておき、 その時刻と現在との差を計算することで、どのくらい時間が経ったか分かります。 ただし、1/1000秒の単位で計算すると非常に大きな値になるので 通常の int 型でなく long 型を使います。

早速、例を見てみましょう。

NXCプログラム

```
#define MOVE_TIME 15000
#define SPEED 75
#define go_forward OnFwd(OUT_BC,SPEED);
#define u_turn OnRev(OUT_BC,SPEED); Wait(500);\
  OnFwd(OUT_C,SPEED); Wait(1000); Off(OUT_C);

task main()
{
  SetSensorTouch(S1);
   /* 測る直前の時刻をlong型変数 t0 に代入して
      今後はこの t0 を基準に時間測定 */
  long t0 = CurrentTick(); // CurrentTick()で現在の時刻が得られる
   /* CurrentTick()と t0 の差が MOVE_TIME 以下
       の時だけ繰り返し */
  while (CurrentTick()-t0 <= MOVE_TIME) {
    go_forward;
    until (SENSOR_1 == 1);
    u_turn;
  }
  Off(OUT_BC);
}
```

【例】タイマーを使ったプログラム

この例を見ればわかるように、基準となる時刻を変数で記憶させておくことが大切です。 いくつもの時間を同時に図りたいときには t0, t1, t2, .. のように複数の 基準時刻を記憶しておくと良いでしょう（もちろん変数名はご自由に）。

練習問題31	ある場所から前進し、障害物に触れたらそのまま後進してもとの位置にもどる プログラムを作成しましょう。 ヒントは「出発してから障害物に触れるまでの時間を測って記憶しておく」
練習問題32	障害物に触れたらUターンするプログラムをつくりましょう。 ただし、5秒以上障害物に触れないときには3秒間停止するようにしましょう。

7. 音を鳴らそう

マインドストームにはサウンド機能が備わっているので、 いろいろな音を出すことができます。 基本的には、もともと用意されている（組み込まれている）音を出す PlaySound という命令と、 音の高さ（周波数）と時間を指定して音を出す PlayTone という命令が あります。

もともとRCXやNXTに用意されている音を鳴らす

「ピッ」という音や「ブー」という音はもともと用意されていますので、 PlaySound という命令に音の種類を指定するだけで 簡単に音を出すことができます。 指定できる音の種類は次の表の通りです。

SOUND_CLICK	ピッ
SOUND_DOUBLE_BEEP	ピーッ、ピーッ
SOUND_DOWN	ピロピロ～と低くなる音
SOUND_UP	ピロピロ～と高く音
SOUND_LOW_BEEP	ブーという低い音
SOUND_FAST_UP	ピロピロ～と速く高くなる音

この表を参考にして PlaySound(SOUND_CLICK); のような命令を書くと 音が出ます。

NXCプログラム

```
task main()
{
  Wait(1000);
  PlaySound(SOUND_UP);  // ピロピロ～と高く音
  Wait(2000);           // 少し待つ
  PlaySound(SOUND_DOWN); // ピロピロ～と低くなる音
}
```

【例】組み込まれた音を鳴らすプログラム

ここで注意したいのは、次の音を鳴らす時には必ず Wait で 待ち時間を入れる、ということです。 実は PlaySound は、音を出すコマンド、というよりは サウンドバッファと呼ばれるサウンドデータの一時的な格納場所に音を送る命令です。 サウンドバッファにあるデータは次から次への演奏されるので、 この待ち時間がないと音がつながったりぐちゃぐちゃに聞こえたりします。 ですので続けて音を鳴らす場合には、 前の音の長さより少し長目の待ち時間を指定するようにしましょう。

練習問題33	障害物に接触すると、 「ピッ」という音を鳴らしてUターンするプログラムを作りましょう。	
練習問題34	障害物に接触すると、 「ブー」という音を鳴らしてUターンするプログラムを作りましょう。	

音の高さを指定して音を鳴らす

あらかじめ用意されている音（組み込まれた音）以外に、 音の高さや時間を指定して音を鳴らすこともできます。

ところで音というは空気の振動です。 この振動が人間の鼓膜に伝わり、小さな骨を経由した後、 電気信号に変換され、 さらに神経を通って脳へ伝わりようやく音として認識されます。 音の高い低いというのは一秒間にどれだけ空気が振動するかで決まります。 この一秒間に空気が振動する回数を（音の） **振動数**または**周波数**と言い、 Hz（ヘルツ）という単位で表します。

音楽では、440Hzの音（1秒間に440回振動する音）を基準にして 他の音を調整していくことが一般的です （楽器などの音を調整することを**調律**と言います）。 この440Hzの音はピアノでいうと鍵盤の中央の『ラ』の音です。 この基準の『ラ』となる音が決まれば、 １オクターブ高い『ラ』の音（倍音と呼びます）は周波数を倍にすることで得られます。 逆に１オクターブ低い『ラ』の音は周波数を1/2にすることで得られます。

さて、楽譜の１オクターブの中には、ピアノの白鍵に相当するド、レ、ミ、ファ、ソ、ラ、シ、 と黒鍵に相当するド＃、レ＃、ファ＃、ソ＃、ラ＃の合計12個の音があります。 １オクターブ上がると周波数が２倍になる、ということから 12個の音を『均等』に振り分けて音の高さを決めるのが **平均律**と呼ばれる音の決め方です。

答えから言うと、一段階高い音（半音高い音）は元の音の周波数を 1.05946...倍することで得られます。 つまり『ラ＃』の音は 440Hz×1.05946 = 446Hz、 さらにそれより半音高い『シ』の音は 446Hz×1.05946 = 494Hz、 ・・・と計算して、最終的に1.05946 を12回かければ（1.05946の12乗倍すれば） 最終的に１オクターブ高い『ラ』の音、つまり周波数が倍の音が得られる、 という理屈です。

以下は、このように計算したそれぞれの音の周波数です。 1オクターブ高い音や低い音は、これらを2倍したり1/2倍することで得られます。 PlayTone では周波数を整数も時間も整数で指定する必要があります。

Note

周波数の比が簡単な整数比になる音を組み合わせるとよく響く和音になります。 1オクターブ違う音は 1:2なので最もよく響きます。 平均律の場合には、オクターブ以外は完全な整数比にはなりませんが、 例えば、「ド」と「ソ」は 2:3、 「ド」と「ファ」は 3:4、「ド」と「ミ」と「ソ」は 4:5:6、 「ド」と「ファ」と「ラ」は 3:4:5 に近い比になっているので、これらの和音はよく響きます。

音	ソ	ソ#	ラ	ラ#	シ	ド	ド#	レ	レ#	ミ	ファ	ファ#	ソ	ソ#	ラ	ラ#	シ	ド
	G	G#	**A**	A#	B	C	C#	D	D#	E	F	F#	G	G#	**A**	A#	B	C
周波数	392	415	**440**	466	494	523	554	587	622	659	698	740	784	831	**880**	932	988	1047

次のプログラムは、 ドレミの歌の冒頭だけを演奏しながら前進し、 障害物に当たると音楽もロボットも止まるプログラムの例です。

NXCプログラム

```
#define DO  523  // ドの周波数
#define RE  587  // レの周波数
#define MI  659  // ミの周波数

task play_music() // 音楽を演奏するタスク
{
  while (true){   // ひたすら繰り返す
    /* 時間の単位は1/1000秒
     Waitで音符の長さ決めて、それより少し短い
     時間をPlayToneで指定するときれいに聞こえる
    */
    PlayTone(DO,250); Wait(3000);
    PlayTone(RE, 50); Wait(1000);
    PlayTone(MI,250); Wait(3000);
    PlayTone(DO, 50); Wait(1000);
    PlayTone(MI,150); Wait(2000);
    PlayTone(DO,150); Wait(2000);
    PlayTone(MI,350); Wait(4000);
  }
}

task main()
{
  SetSensorTouch(S1);

  start play_music ; // 演奏を開始
  OnFwd(OUT_BC,75);  // 前進
  until (SENSOR_1 == 1);
  /* 次の命令は firmware v1.28 ではエラーになる */
  stop play_music ;  // 前進
  Off(OUT_BC);       // 停止
}
```

【例】~~ドレミの歌~~チューリップの歌（の冒頭だけ）を演奏しながら前進し、障害物に当たると止まるプログラム

この例では、音楽を演奏する部分を play_music という名をつけた 別のタスクとして定義しています。 別のタスクを開始するには、 main タスクの中で start play_music;のように 単に「 start タスク名;」とするだけです。 逆にタスクを終了するため、「 stop タスク名;」します。

Note

NXTのファームウェア1.28には不具合があり、あるタスクから他のタスクを止める stop のような命令は使えません。 John Hansenさんの"enhanced NBC/NXC firmware"を使いましょう。

注意してほしいのは、 別のタスクを開始すればそれは元のタスクと同時に実行される、ということです。 複数のタスクの命令が並行して処理されるので並列処理という言葉がよく使われます。 並列処理では、 モータをどのタスクがコントロールするか、ということがよく問題になります。 たまたまこの例では、 play_music は音の出力だけを扱い、 main はモータ出力とセンサ入力だけを扱っているので、 モータをコントロールする権利を奪いあう、ということはありませんでした。 この問題については後ほど再び触れます。

練習問題35	上の「~~ドレミの歌~~チューリップの歌」の冒頭を演奏するプログラムを、 次の２小節も演奏するように改良しましょう。
練習問題36	上の「~~ドレミの歌~~チューリップの歌」の冒頭を演奏するプログラムを、 障害物を取り去れば演奏を再開するように改良しましょう。

8. ロボットに目をつけよう

「目」と言っても、ここで使うのは明るさを感知できるだけの原始的な目ですが、 それでも周りの状況を感知してロボットの動きをコントロールすることができます。 NXTには光センサが付属しています。 この光センサを使って黒線に沿って動く、 いわゆるライントレース・ロボットの製作を目標にしましょう。

光センサの取り付け

インストラクションの32～34ページに光センサの使用例が載っています。 いずれもアタッチメントとして、 タッチセンサと交換して（あるいは前後に両方つけて）使うことができます。

付属の光センサは自分でも光を出すことができ、 近くの物体であれば、 この光の反射でその物体が白っぽいか黒っぽいか判断することができます。 ただし、黒い物体でも光沢のあるものであれば光をよく反射してしまい 「明るい」と判定されるので注意が必要です。 また、色については「黒」と「緑」のような違いであればある程度 見分けることができますが、 本格的に色を調べるにはNXT2.0付属のカラーセンサが必要です。

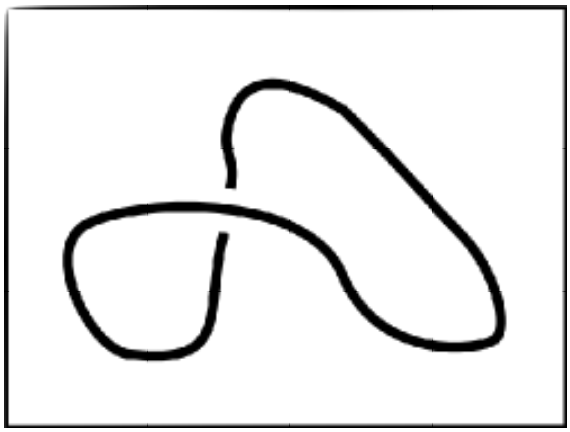
コースの作成

プログラムに取りかかる前に、 画用紙を使ってライントレースで使うコースを作りましょう。 黒い太いペンでコースを描きます。 単純過ぎず、複雑過ぎず、次のような箇所を採り入れた周回コースにしてみましょう。

- 直線
- 左右のカーブ
- ヘアピンのような少し急なカーブ
- 直角に交わる交差点

最初の段階では、十字の交差点を完全に描かず、 一方の線が他方の線と交わる3cmくらい手前で止めておき、 向こう側も3cm離れたところから描くようにします（将来的にはつなぎます）。 またコースを描く際、次のような点に注意しましょう。

- 黒線の太さは15～20mm程度で、太さも濃さもなるべく一定に。
- 画用紙の縁から10cm以内のところには線は描かない（タイヤやキャタピラが紙からはみ出ないように）。



黒い線にさしかかると止まるロボット

まずはタッチセンサの場合と同様に簡単な例として、 黒い線に差しかかったら止まる、という簡単なプログラムから作ってみましょう。 プログラムの構造は、タッチセンサの最初の例とほとんど同じです。

NXCプログラム

```
#define THRESHOLD 42 // しきい値
#define SPEED 50  // 少しゆっくり

task main()
{
  SetSensorLight(S3); // 端子3には光センサ

  OnFwd(OUT_BC,SPEED);
  until (SENSOR_3 <= THRESHOLD); // 黒線まで前進
  Off(OUT_BC); // 停止
}
```

【例】黒い線に差しかかると止まるプログラム

ある値よりも上がると（あるいは下がると） 条件の判定が変わるような境界の値のことを英語で、 **threshold**と言います。 発音はちょっと難しく「スレシュホウルド」とか「スレショウルド」のように読みます。 日本語では**閾値**（いきち）とか**しきい値**と と言いますが、閾値を「しきい値」と読むこともあります。 ただし「敷居値」と書くことはあまりありません。 もともと敷居は、 上部の鴨居（かもい）とセットで襖や戸などの下部のレールとなる家の部材のことです。 「敷居をまたがせない」という慣用句のように空間を区切る意味でよく使われます。

もちろん THRESHOLD という定数名でなくて IKICHI や SHIKIICHI という定数名をつけても 全く問題ありません。

NXTのセンサは、黒で25～30、白で55～60くらいなので、 その中間くらいの42を「しきい値」として設定しています。 光センサはモノによって多少のバラつきがありますし、 部屋の明るさによっても変化しますので、 実際に明るさを測定して、より適切な値を選びます。

ところで上のプログラムでは SetSensorLight で光センサを使うための宣言を行なっています。 この宣言によってセンサの値（明るさ） が0から100までの整数で得られるようになります。 それを until の条件式で THRESHOLD と比較しているわけです。 白と黒の中間くらいの値をしきい値として選べば、 比較演算子としてくを使うか <= を使うかの違いはそれほど大きくありません。

さて、プログラムを走らせると光センサが赤く光るはずです。 この赤い光の反射を見て黒い部分や白い部分の明るさを測定しているので、 あまりセンサが紙から離れてしまうと、 周りの光の影響で測定値が安定しません。 また、特にNXTの光センサの場合には構造上、紙に密着させると 光が受光部に届きにくくなり白い部分でも値が小さくなってしまいます。 ですのでセンサは紙から数ミリ～1cmくらい離して設置するのがよいでしょう。 部屋の明るさなども考慮して調整してみてください。

練習問題37	黒い線にさしかかると1秒間停止した後、再び前進する、というのを繰り返す プログラムを作りましょう。
練習問題38	黒い線にさしかかるとUターンする、というのを繰り返すプログラムを作りましょう。

実際に明るさを測ってみよう

実際の明るさを測定してみましょう。 トップメニューから「View」を選び、 左右の矢印ボタンで「Reflected light」を選びます。 そしてさらに左右の矢印ボタンで光センサを接続している 端子（port）番号を選べばディスプレイにセンサの値が表示されます。

周りの明るさの違いやセンサの個体差もあるので、 周りのグループのセンサの値と違っていても不思議ではありません。

練習問題39	作ったコースのいろいろな部分の明るさを値を測ってみましょう。 最も明るい値、最も暗い値はいくらでしょうか？
練習問題40	黒い線の中央に沿って移動し、どのくらい値にバラつきがあるか 調べてみましょう。

さらに次の練習問題もやってみましょう。

練習問題41	黒線と垂直方向に、白い部分から黒線の中央まで光センサをゆっくりと移動して 値の変化を観察しましょう。 値が最大値から最低値まで変化していく幅（センサの移動距離）はどのくらい？ RISの場合は２つの光センサの両方を調べてみましょう。
練習問題42	黒線の境界部分に沿って移動し、どのくらい値にバラつきがあるか調べてみましょう。

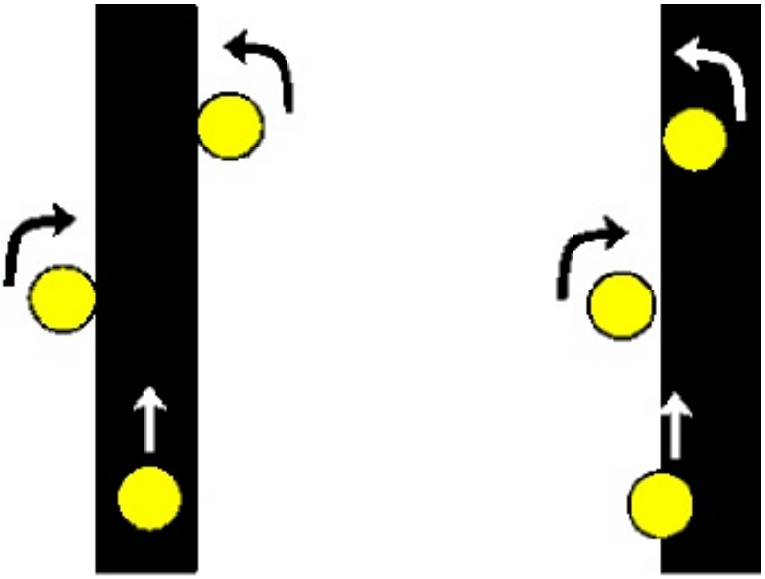
黒線に沿って走るロボット

さていよいよ黒線に沿って走るロボットを作ってみましょう。 このようなロボットのことを 英語では line follower（ライン・フォロアー）とか line tracer（ライン・トレーサー） と呼びます。 日本ではライントレース・ロボットやライントレーサーという言葉がよく使われます。

ライントレースの原理そのものは簡単です。 が、スピードや確実性をアップさせるにはいろいろな工夫が必要で とても奥深いテーマです。

光センサを１個使ってライントレースする主な方法としては 次の２つがあります。

1. 基本的に光センサが黒線の上を動くようにする。 光センサが白い部分にさしかかったら左または右にカーブさせる。
2. 基本的に光センサが黒線と白い部分の境界上を動くようにする。 光センサが白い部分にさしかかったら黒線側へ、 黒い部分にさしかかったら白い方へは向かうように、 カーブさせる。



【1】 【2】

１番目の方法では、白い部分にさしかかった時、 左に曲がべきか右に曲がるべきか、何らかの方法で決めなければなりません。 これに対し２番目の方法では、どちらに曲がるべきかは明らかです。 もちろん１番目の方法でも効率よく曲がる方向を決めていく良い方法がありますが、 ここでは簡単な２番目の方法で、 黒線の左側の境界をたどっていく、というプログラムを作ってみましょう。

NXCプログラム

```
#define THRESHOLD 42
#define SPEED 50
#define turn_left OnFwd(OUT_B,SPEED);Off(OUT_C);
#define turn_right Off(OUT_B);OnFwd(OUT_C,SPEED);
#define STEP 1  // 一回の判定で進む時間

task main()
{
  SetSensorLight(S3);

  while (true) {
    if (SENSOR_3 < THRESHOLD){  // 線上なら
      turn_left;              // 左へ
    } else {                  // 線上でなければ
      turn_right;             // 右へ
    }
    Wait(STEP);  // STEPの値をいろいろ変えてみよう
  }
}
```

【例】黒い線（の境界）に沿って動くプログラム

この例のように if の条件式は else とセットで使用することで、条件が満たされた場合だけでなく、 条件が満たされない場合の動作も同時に記述できます。 ちなみに英語のelseは「～でなければ」とか「他の」という意味です。

練習問題43	上の例で定数 STEP の値をいろいろ変えた時にロボットの動きが どうなるか調べてみましょう。

練習問題44	上の例でどのくらい急なカーブまでなら、 光センサがちゃんと境界上を動いているか、 調べてみましょう。

上の例のように、 片方のタイヤを停止して左折・右折する場合には、 回転半径は左右のタイヤの間隔に等しくなるので、この半径の円のカーブよりも急なカーブはうまく曲がることができません。 しかし、内側のタイヤを逆転して旋回に近い動作をさせれば、 もっと急なカーブにも対応させることができます。

練習問題45	内側のタイヤを外側のタイヤの半分くらいの回転スピードで逆転させることで、 急なカーブでの動きが改善されるか試してみましょう。

直進もさせたい

前の例では、白か黒かの判定をして左折と右折だけで前進していくプログラムでした。 そこで少しプログラムを改良して、明るさが中間くらいの場合には直進するようにしてみましょう。

そのためにはしきい値を２つ用意する必要があります。 例えば、明るさが37未満の場合は左へ、37以上50未満の場合は直進、 50以上の場合は右へ、 と動かしたい場合は37と50という２つのしきい値を使います。

実際には次の例のように else のあとにさらに if ブロックを続けて書くことで、 条件を順番に調べていくことができます。

NXCプログラム

```
#define BLACK 30  // これ以下は黒（調整してください）
#define WHITE 50  // これ以上は白（調整してください）
#define SPEED 50
#define go_forward OnFwd(OUT_BC,SPEED);  // 直進を追加
#define turn_left OnFwd(OUT_B,SPEED);Off(OUT_C);
#define turn_right Off(OUT_B);OnFwd(OUT_C,SPEED);
#define STEP 1

task main()
{
  SetSensorLight(S3);

  while (true) {
    if (SENSOR_3 < BLACK) {       // 線上なら
      turn_left;                  //   左へ
    } else if (SENSOR_3 < WHITE) { // 境界付近なら
      go_forward;                  //   直進
    } else {                      // 線からはずれれば
      turn_right;                 //   右へ
    }
    Wait(STEP);
  }
}
```

【例】黒線（の境界）に沿って動くプログラム（直進あり）

この例で大切なことは、条件を判断する順番です。 最初の SENSOR_2 < BLACK が true ならば、 それ以降の条件は判断されません。 つまり、もし最初に if (SENSOR_2 < WHITE) というような 条件式を書いてしまうと、そのあと else if (SENSOR_2 < BLACK) のような条件式を書いても 判断されないので意味がありません。 このように

```
if ( … ){          // 条件その1
   …
} else if ( … ) {  // 条件その2
   …
} else if ( … ) {  // 条件その3
   …
} else {    // どの条件に合わない場合
   …
}
```

と条件式をつなげていく if ブロックでは、 条件式の順序に注意しましょう。

練習問題46	上の例で、しきい値を変えて境界領域の範囲をうまく調整しましょう。

練習問題47	上の例で、左右に曲がる際に内側のタイヤを逆転させるとどのような動きになるか調べましょう。

練習問題48	上の例で、直進の時と左折右折の時のスピードを違うものに すると動きがどうかわるか調べてみましょう。

これまでの練習で、どのような値を調整すればどのように動きが変化するか、 多少はつかめたでしょうか？ 上の方法で更に細かく状況を場合分けするのは簡単です。

練習問題49	上の例を改良して、 「完全に白」「白と境界付近の間」「境界付近」「黒と境界付近の間」「完全に黒」 の５段階で動きを変えてみましょう。

もっと滑らかに、もっと確実に、もっと速く

これまでのライントレースのプログラミングで、どういう場合うまく行って、 どういう場合に失敗するか、というのが徐々に理解できてきたと思います。

おそらくまず最初に改善したいのは、光センサが左右に振られすぎる、 という点ではないでしょうか。 左右に振られすぎることで、ぎこちない動きになったり、 光センサが黒い線を通り越して反対側に移動したりしてしまいます。 実はこの問題は、左右に曲がるスピードやパワーを減らすことで 以外と簡単に改善できます。 つまり、直線部分と曲線部分でスピードを変えるわけです。

また、光センサを左右の駆動輪の真ん中からあまり遠くない場所に持ってくる、 というロボット本体の改良も有効です。 そうすることで、 ロボットが左右に急激に振られた時の光センサの振られる距離を減らすことができ、 スピードがある程度あってもセンサが過敏に反応しなくなるので、 結果としてぎこちない動きが抑えられます。

この他、急なカーブで光センサが黒い線（の境界）に追従できていない場合には、 前の練習問題であったように、内側のタイヤを逆転させて その場で旋回させてみましょう。 ただ、旋回だと前に進まないので、よほど線からは外れた時や よほど黒線の中央部に寄ってしまったときだけ旋回させて、 その他のカーブは内側のタイヤを停止するだけにしておく、 あるいはゆっくりと回転させる、 という方法が考えられます。 もちろん境界付近は直線で動かすようにすれば、これも 前の練習でやったように５段階の場合分けになります。

以下は、これらの点の改善を試みたサンプル・プログラムです。

NXCプログラム

```
#define THRESHOLD 45
#define SPEED_H 50  // 直線用のスピード
#define SPEED_L 25  // カーブのスピード
#define OnRL(speedR,speedL) \
  OnFwd(OUT_B,speedR);OnFwd(OUT_C,speedL);
#define go_forward OnRL(SPEED_H,SPEED_H); // 直進
#define turn_left1 OnRL(SPEED_L,-SPEED_L);// 左旋回
#define turn_left0 OnRL(SPEED_L,0);       // 左折
#define turn_right0 OnRL(0,SPEED_L);      // 右折
#define turn_right1 OnRL(-SPEED_L,SPEED_L);// 右旋回
#define STEP 1 // 1回の判断で動く時間

task main()
{
  SetSensorLight(S3);

  while (true) {
    if (SENSOR_3 < THRESHOLD -15) {
      turn_left1;
    } else if (SENSOR_3 < THRESHOLD -7) {
      turn_left0;
    } else if (SENSOR_3 < THRESHOLD +7) {
      go_forward;
    } else if (SENSOR_3 < THRESHOLD +15) {
      turn_right0;
    } else {
      turn_right1;
    }
    Wait(STEP);
  }
}
```

【例】 滑らかに確実にラインをトレースするように少し改善したプログラム

練習問題50	上の例のような5段階で明るさを判断するプログラムを作り、 それぞれのしきい値を適切に調整しましょう。

練習問題51	上の例のような5段階で明るさを判断するプログラムを作り、 旋回時や右折左折時のスピードやパワーを適切に調整しましょう。

これまでの例では、光センサの値を何段階かに分けて動きを区別してきました。 しかし、現在の（あるいは少し前の）光センサの値から、 左右のモータの適切な回転数の差を割り出すことができれば、 より連続的でスムーズな動きが期待できます。 おそらく最初に思いつくのは、 光センサの**中間値からずれた分に比例**して回転数の差をつけ、 内側のタイヤの回転を遅くしたり、逆転したりする方法でしょう。

さらに少し進んだ方法として、 条件判断する時にその直前（あるいは少し前の）の判断を参考にする、 ということも考えられます。 例えば上のプログラムでは、 完全に黒くなった時に左旋回で方向を修正するわけですが、 その次の条件判断で少しだけ黒くなった場合には、 まだ少し黒いからといってさらに左に曲げるのではなく、 そのまま直進させてみる、ということも考えられます。 そうすることで方向が修正されすぎてしまうことを防ぎます。 このように、明るさの**変化の割合を考慮**することで、 よりスムーズな動作が期待できます。

また、直前の条件判断だけでなく、 少し前からの状態を考慮に入れることもできます。 例えば、ずっと黒が続いた場合にはもっと左に曲がる、など **変化の積算を考慮**する方法もあります。

結局のところ、 ライントレースというのは光センサの値やその時間変化に応じて 左右のモータをどのようなスピード（あるいはパワー）で回すとよいか、 という問題なのですが、実際に少し試してみるとその奥の深さを 実感できるでしょう。 この先は宿題です。

交差点はどうする？

交差点に差しかかった場合、これまでのプログラムではロボットは どのように動くでしょうか？ 黒線の左側の境界をトレースしている場合には、そのまま左折しようと するはずです。

そこで、交差点を非常に急なカーブと考えて、 左に何度も何度も繰り返して向きを修正しようとする場合には、 それを交差点とみなすことにしましょう。 つまり明るさ判断のループを繰り返す際、一定の回数連続して 黒になった場合には、進行方向を修正して、交差点を横断するように プログラムを書き直しましょう。 そのために、連続して黒になる回数を数える変数（カウンタ）を用意します。 ただし、黒にならなかった場合はすぐにリセットします。

NXCプログラム

```
#define THRESHOLD 45
#define SPEED_H 50
#define SPEED_L 25
#define OnRL(speedR,speedL) OnFwd(OUT_B,speedR);OnFwd(OUT_C,speedL);
#define go_forward  OnRL(SPEED_H, SPEED_H);
#define turn_left1  onRL(SPEED_L, -SPEED_L); // 左旋回
#define turn_left0  onRL(SPEED_L, 0);        // 左折
#define turn_right0 onRL(0, SPEED_L);        // 右折
#define turn_right1 OnRL(-SPEED_L, SPEED_L); // 右旋回
#define STEP 1  // 1回の判断で動作させる時間
#define nMAX 300  // 通常のカーブとして許容できる繰り返しの最大値（調整する）
#define short_break Off(OUT_BC); Wait(1000); // 小休止
#define CROSS_TIME 200  // 交差点通過にかかる時間
/* 交差点を渡る */
#define cross_line OnRL(SPEED_L,SPEED_L);Wait(CROSS_TIME);short_break;

task main()
{
  SetSensorLight(S3);
  int nOnline=0; // 続けて黒になった回数（カウンタ）

  while (true) {
    /* 黒を続けてnMAX回繰り返さない間、通常のライントレースをする */
    while (nOnline < nMAX) {
      if (SENSOR_3 < THRESHOLD-15) {
        turn_left1;
        nOnline++;  // カウンタを増やす
      } else {
        if (SENSOR_3 < THRESHOLD-7) {
          turn_left0;
        } else if (SENSOR_3 < THRESHOLD+7) {
          go_forward;
        } else if (SENSOR_3 < THRESHOLD+15) {
```

```
      turn_right0;
     } else {
      turn_right1;
     }
     nOnline=0; // カウンタをリセット
    }
    Wait(STEP);
   }
   short_break; // 小休憩
   turn_right1; Wait(nMAX*STEP); // 進行方向修正
   cross_line; // 交差点を渡る
   nOnline=0; //カウンタをリセット
  }
}
```

【例】 交差点を渡るプログラム

もちろんこのプログラムではヘアピンのような急なカーブと交差点を見分けるのは 少し難しいでしょう。 しかし、例えば、コースをいくつかのゾーンに分けた上で、 ゾーンごとに違う方法で交差点や急カーブを判別し、 タイマを使って（時間条件も加味して） おおよその交差点通過時刻を予想することで、 かなり精度よく交差点と認識できるはずです。

練習問題52	急なカーブを交差点と間違って認識した場合、 「交差点」を渡り終わった時点では黒線上にいないはずです。 このことを利用して、ご認識した場合には再度もとのコースに戻るよう 上のプログラムを改良しましょう。
練習問題53	カウンタで回数を数える代わりに、 タイマーを使って一定の時間黒い部分が続いたら交差点とみなす、 というプログラムにしてみましょう。

9. 音に反応するロボットを作ろう

NXTには音の大きさを測るサウンドセンサが付属しています。 使い方は光センサと同様です。 ある一定の大きさ以上の音がしたらUターンするプログラムを作ってみましょう。

NXCプログラム

```
#define SPEED 50
#define THRESHOLD 40
#define go_forward OnFwd(OUT_BC,SPEED);
#define u_turn OnRev(OUT_B,SPEED); OnFwd(OUT_C,SPEED);\
 Wait(1000); Float(OUT_C);

task main()
{
  SetSensorSound(S2); // 端子2にはサウンドセンサ

  while (true) {
    go_forward;

    /* 音がしたらUターン */
    if (Sensor(S2) <= THRESHOLD) { u_turn; }
  }
}
```

【例】音がしたらUターンするプログラム

練習問題54	音がしたら、ピクンと動くロボットのプログラムをつくりましょう。
練習問題55	音の大きさに比例して、より遠くにバックするロボットの プログラムをつくりましょう。

10. 超音波で距離を測ろう

NXTには超音波センサが標準装備されています。 まずはインストラクション従ってロボットに超音波センサを取り付けてみましょう。

人間の耳はだいたい 20～20,000Hz（ヘルツ）くらいの周波数の音を 認識できると言われています。 人間の耳では聞こえない高い周波数の音を超音波といいます。 超音波センサは、 発信した超音波が物体に反射して戻ってくるまでの時間を 測ることで物体までの距離を計算します。 NXTに付属の超音波センサでは40kHz(=40,000Hz)の超音波が使われています。

練習問題56	音速が約340mとして、40kHzの超音波センサが出す超音波が一回振動する間に 進む距離を求めてみましょう。一秒間に40,000回振動して340m進むということは？

さて、 超音波センサを使った例を見てみましょう。 センサの指定方法とセンサの値を得る方法に注意してください。 光センサやタッチセンサと違います。 それ以外は特に難しくないですね。

NXCプログラム

```
#define SPEED 50
#define go_forward OnFwd(OUT_BC,SPEED);
#define u_turn OnRev(OUT_B,SPEED); OnFwd(OUT_C,SPEED);\
 Wait(1000); Float(OUT_C);

task main()
{
  SetSensorLowspeed(S4); // 端子4には超音波センサ

  while (true) {
    go_forward;

    /* 25cm以内に障害物があるとUターン */
    if (SensorUS(S4) <= 25) { u_turn; }
  }
}
```

【例】 15cm以内に障害物があるとUターンするプログラム

実はこれまで紹介した他のセンサはアナログ・センサと呼ばれるもので 測定値を電圧(アナログ量)として出力するセンサでした。 一方、この超音波センサはI^2C（アイ・スクエアド・スィー）と呼ばれるデジタル通信方式を用いてデータを転送します。 上の例の SetSensorLowspeed はこのI^2Cのセンサを使用するためのものです。

付属の超音波センサの特性を少し理解するために、 プログラムを作る前に次の練習問題をやっておきましょう。

練習問題57	光センサの値を表示したのと同じ方法で、超音波センサで測った距離を表示させてみましょう。 ロボットの向きをいろいろ変えてみて、値がどう変化するか調べてみましょう。
練習問題58	どのくらい遠くまでまで測ることができるでしょうか？ どのくらい近くまで測ることができるでしょうか？

ある程度、超音波センサの特性を理解できたら、次の練習問題に進みましょう。

練習問題59	ロボットに手を近づけたとき距離25センチ以内に近づくと離れ、40センチ以上離れると 近づく、「つかず離れず」ロボットのプログラムを作りましょう。
練習問題60	手を近づけたときなるべく30センチの距離を保つようなプログラムを 作りましょう。ただし、少し近づいた時や少し離れた時はゆっくり 30センチの距離に近づき、 近づきすぎた時や離れすぎたとき

は全速力で30センチに戻るような プログラムにしてみましょう（30センチからのずれに比例した速さで30センチの距離に戻るような プログラム)。

11. 一連の動作を部品化しよう

これまでもマクロを使って、再利用可能な一連の動作は部品化してきました。 一行で書くマクロは比較的単純な一連の命令に適していますが、 複雑になってくると使い勝手がよくありません。 ここでは、サブルーチンや関数といったプログラムの部品を使う方法を 説明します。

インライン関数を除くサブルーチン（関数）、タスクは合計256個まで使用できます。

サブルーチン

ルーチンというのは決まりきった手順や一連の仕事を意味します。 プログラムでも、一連の命令をまとめたものを意味し、一番最初に実行されるものをメインルーチン、 そのメインルーチンから呼び出させるものをサブルーチンと呼びます。

マクロがコンパイル時にもとの命令の置き換えるのに対し、 サブルーチンは実際にプログラムが走っている時に 呼び出されます。ですので、サブルーチンを何回呼び出しても プログラムのサイズは増えません。

以前マクロを使って書いたプログラムを サブルーチンを使って書き直してみましょう。

NXCプログラム

```
#define TURN_TIME 800  // 左折の時間を定義
#define SPEED 75

/* turn_leftという名のサブルーチンを定義 */
sub turn_left()  // sub の代わりに void でもOK
{
  OnFwd(OUT_B,SPEED); Off(OUT_C);
  Wait(TURN_TIME);
  Off(OUT_B);
}

task main ()
{
  OnFwd(OUT_BC,SPEED); Wait(1000);
  turn_left();
  OnFwd(OUT_BC,SPEED); Wait(2000);
  turn_left();
  OnFwd(OUT_BC,SPEED); Wait(3000);
  Off(OUT_BC);
}
```

【例】サブルーチンを使ったプログラム

NXCのサブルーチンは引数を取ることができます。 この引数を取るサブルーチンを使えば、 上記の直進部分もサブルーチンとして定義できます。

NXCプログラム

```
#define TURN_TIME 800
#define SPEED 75
sub turn_left() // 引数なしのサブルーチン
{
  OnFwd(OUT_B,SPEED); Off(OUT_C);
  Wait(TURN_TIME);
  Off(OUT_B);
}
sub go_forward(int t) // 整数型の引数を取る
{
  OnFwd(OUT_BC,SPEED); Wait(t);
  Off(OUT_BC);
}
task main ()
{
  go_forward(1000);
  turn_left();
  go_forward(2000);
  turn_left();
  go_forward(3000);
}
```

【例】直進サブルーチンを使ったプログラム

マクロを定義する時と違い、引数の型（上の例の場合は整数型を表す int ）を指定するのを忘れないようにしましょう。

練習問題61	上のサブルーチンを改良してスピードも引数にできるようにしましょう。

関数

プログラミングで登場する「関数」には、 中学校で習うようなある独立変数(入力)に対して従属変数(出力)が決まる y=f(x) のような形の場合もあれば、ただ動作をするだけで何の値も返さない場合もあります。 値を返す関数の場合には、変数の型と同じく、整数型、小数型、文字型、ブール型などの型があります。 例えば整数を返す関数は整数型で、変数の定義と同様に関数の前に int という キーワードを指定します。

一方、値を返さない関数には void という特別な型を指定します。 ちなみに英語のvoidというのは「空っぽの」という意味です。 実は前に紹介した、サブルーチンは何の値も返さない「関数」で、 sub の代わりに void と書くこともできます。

以下は、進みたい距離から必要なタイヤの回転数を計算する関数を使ったプログラムの例です。

NXCプログラム

```
#define SPEED 50

float GetAngle(float d) // 距離 d からタイヤの回転数を計算する関数
{
  const float diameter = 5.45;       // タイヤの直径(cm)
  const float pi=3.1415;             // 円周率
  float ang = d/(diameter*pi)*360.0; // 角度を計算する
  return ang;                        // 角度を返す(float型)
}

task main ()
{
  Wait(2000); // スタートボタンを押してすぐに動き出さないように
  int angle = GetAngle(20.0);   // 20.0cm進むのに必要な回転角度
                    // 自動的にfloatからint型に変換して代入
  RotateMotor(OUT_BC,SPEED,angle); // 角度指定でモータを回転
}
```

【例】距離からタイヤの回転角を計算するfloat型の関数を使ったプログラム

この例では diameter や pi は途中で変更する必要がないので、 定数のfloat型として const というキーワードを追加してあります（定数のことを英語で constant と言います)。

計算される距離はかなり正確ですが、実際に動く距離は多少の誤差があります。 どの程度正確なのか何回か測って見ましょう。

この例では引数は一つですが、複数の引数を指定したい場合には、コンマで区切ります。

練習問題62	左右のモータを逆転して方向変換する時、方向変換したい角度から タイヤ（モータ）の回転角度を求める整数型の関数を作ってみましょう（上の例を参考に）。

インライン関数

「インライン関数」は上の「関数」と似ていますが、マクロと同じようにコンパイル時に実際の命令と置き換えられます。 つまり関数やサブルーチンと違って、プログラムの実行中にわざわざ呼び出して使う、 ということがないかわりに、多用すればプログラムのサイズが大きくなります。 定義のしかたは inline というキーワードが必要な以外、通常の関数の定義と同じです。

NXCプログラム

```
#define TURN_TIME 800
#define SPEED 75

inline void turn_left() // inline に注意
{
  OnFwd(OUT_B,SPEED); Off(OUT_C);
  Wait(TURN_TIME);
  Off(OUT_B);
}
inline void go_forward(int s, int t) // 2個の引数を取る
{
  OnFwd(OUT_BC,s);
  Wait(t);
  Off(OUT_BC);
}
task main ()
{
  go_forward(50,1000);
  turn_left();
  go_forward(75,2000);
  turn_left();
  go_forward(30,3000);
}
```

【例】インライン関数を使ったプログラム

少し長いマクロは、このインライン関数を使って書きなおすことで プログラムが見やすくなります。

練習問題63	これまでに作ったプログラム中の長いマクロをインライン関数に書き直してみましょう。

12. ディスプレイに文字を表示しよう

NXTには64x100ピクセルの液晶ディスプレイが備わっています。 このディスプレイにテキストや数値を表示させてみましょう。 現在のセンサの値やモータの回転数などを表示させることで、 プログラムの改良がしやすくなります。

基本的には、 TextOut という命令でテキストを、 NumOut という命令で数値を表示できます。 このときそれらを表示する位置は、左下を原点(0,0)とするxy座標で 指定することができます。ただしy軸(縦軸)は8の倍数である必要があるので、 あらかじめ定義された LCD_LINE1 ～ LCD_LINE8 を使ったほうが便利です。 ちなみに LCD_LINE1 は56、 LCD_LINE8 は0と定義されています。

NXCプログラム

```
task main()
{
  SetSensorLowspeed(S1);
  SetSensorSound(S2);

  while (true) {
    ClearScreen();
    TextOut(0,LCD_LINE1,"UltraSonic:"); // 1行目の左端に表示
    TextOut(0,LCD_LINE2,"Sound:");      // 2行目の左端に表示
    NumOut(80,LCD_LINE1,SensorUS(S1)); // 超音波センサの値を表示
    NumOut(80,LCD_LINE2,Sensor(S2));  // サウンド・センサの値を表示
    Wait(300);
  }
}
```

【例】 超音波センサとサウンド・センサの値を表示するプログラム

練習問題64	タッチセンサの値と光センサの値をディスプレイの表示するプログラムを作りましょう。
練習問題65	左右のモータの回転角度を表示するプログラムを作り、 ロボットを動かして表示がどのように変わるか調べてみましょう。 例えばB端子に接続したモータの回転角度は、 MotorTachoCount(OUT_B)で得られます。

ディスプレイにはテキストや数値だけでなく、点、直線、円、楕円、長方形、多角形なども描くことができます。

命令	説明
PointOut(40,50);	(40,50)に点を描く
LineOut(30,20,70,50);	(30,20)と(70,50)を結ぶ直線を描く
CircleOut(50,32,25);	(50,32)を中心に半径25の円を描く
EllipseOut(40,30,45,25);	(40,30)を中心に幅45高さ25の楕円を描く
RectOut(30,20,40,25);	(30,20)から幅40高さ25の長方形を描く
PolyOut(myPoints);	2次元配列myPointsを使って多角形を描く

多角形を描く際に用意する「二次元配列」は、 以下の例のように2つの整数の組(x,y)を複数並べたものものです。

NXCプログラム

```
task main()
{
  int myPoints[][] = {{10,10},{20,30},{60,40}};  // 配列を定義

  ClearScreen();
  PolyOut(myPoints); // 多角形を描画
  Wait(2000);        // 2秒間表示
}
```

【例】 (10,10) (20,30) (60,40) を頂点にもつ三角形を描くプログラム

13. ２台のロボットで通信しよう

NXTはBluetooth（ブルートゥース）と呼ばれる近距離の無線を 使って4台まで通信することができます。 4台のうち一台がマスター(主)と呼ばれる役割を果たし、 他の3台はスレーブ(従)という役割になります。 Bluetoothを使用するためにはまず設定をしなければいけません。

Bluetoothの設定

まず、メインメニューから「Bluetooth」を選び、 「On/Off」で「On」にします（マスター、スレーブ共）。 そうすると再度「Bluetooth」を選んだ時にサブメニューが現れるようになります。 この時、ディスプレイの左上にBluetoothのマークと「<」というマークが 表示されていることを確認してください。 この「<」というマークは、 「BluetoothがONになってはいるが相手とはまだ繋がっていない」 という意味です。

次に以前の接続の設定を消しておきます（毎回する必要はありません）。 「My contacts」というボタンを押して、 もし別のアイコン（例えばNXTのアイコン）が表示されるようでしたら、 それを選び、「Connect」ではなくゴミ箱のアイコンを選んで「Delete」を 押します。 マスター、スレーブ共に古い設定を消しておきましょう。

２台でつなぐ場合、どちらをマスターにするか決めます。 そしてマスター側で「Search」を押して通信相手を探します。 この時、スレーブ側は何もする必要はありません。

20秒程度待っていると、砂時計のアイコンが消えてNXTのアイコンが表示されます。 これはスレーブ側のNXTが見つかったことを示しています。 ここでそのアイコンを押すと1～3までの番号を選ぶことができます。 通常は1（第1のスレーブ）を選べばよいでしょう。 ちなみにマスターには「0番目」が自動的に割り振られます。

スレーブの番号を選ぶと「Connecting ...」と表示された後、 マスター側とスレーブ側の両方で同じ画面になり「Passkey」を入力するように求められます。 通信するためには同じPasskeyを設定する必要があります。 標準のPasskeyは「1234」になっていますが、 変更したい場合は数字やアルファベットを選びます。 マスター、スレーブ共にチェックマークを押せば設定が完了です。 この時、先程の「<」マークが「<>」に変わったことを確認してください。

「Line is busy」などのエラーが出てうまく繋がらない場合には、 一度Bluetooth機能をOffにしたり、上の設定を最初からやってみましょう。

接続できているかプログラムで確認しよう

NXCプログラム（マスター側）

```
#define CONN 1  // スレーブ(相手)の接続番号

task main(){
  if ( BluetoothStatus(CONN) == NO_ERR ) { // NO_ERRは0と定義されている
    TextOut(0,LCD_LINE1,"Connected");
  } else {
    TextOut(0,LCD_LINE1,"Not connected");
  }
  Wait(3000);
}
```

NXCプログラム（スレーブ側）

```
#define CONN 0  // マスター(相手)の接続番号

task main(){
  if ( BluetoothStatus(CONN) == NO_ERR ) {
    TextOut(0,LCD_LINE1,"Connected");
  } else {
    TextOut(0,LCD_LINE1,"Not connected");
  }
  Wait(3000);
}
```

【例】 接続できていれば「connected」、できていなければ「Not connected」を表示するプログラム

ここで BluetoothStatus(CONN) は CONN と接続できていれば 0 (= NO_ERR) を返す関数です。 もし２台がBluetoothで接続されて通信できる状態にあれば、 どちらのディスプレイにも「Connected」と表示されるはずです。

通信してみよう

実際に通信できる状態かどうか、直接相手に命令を送るプログラムを作って確認してみましょう。

NXCプログラム（マスター側）

```
#define CONN 1 // スレーブの接続番号

task main()
{
  until (BluetoothStatus(CONN) == NO_ERR); // 接続できるまで待つ
  RemotePlayTone(1, 440, 1000); // スレーブの音を出す
}
```

【例】 スレーブ側の音を出すプログラム

この他にも直接相手のモータやセンサ、プログラムなどをコントロールする命令がいろいろと用意されています。 例えば次の例のように相手のプログラムをスタートしたり、ストップしたりすることもできます。

適当なプログラムをスレーブ側で用意し、マスター側でプログラムを起動してみましょう。 ライントレースのプログラムでも何でもかまいません。 以下の例では test2.nxc という名前のNXCプログラムを使っています。 NXT上にあるプログラムの名前はコンパイル後の test2.rxe となります。

NXCプログラム（マスター側）

```
#define CONN 1 // スレーブの接続番号

task main()
{
  SetSensorTouch(S1);
  until (BluetoothStatus(CONN) == NO_ERR); // 接続できるまで待つ

  until (SENSOR_1 == 1);     // タッチセンサを押してスレーブのプログラムを起動
  RemoteStartProgram(CONN, "test2.rxe");  // ファイル名の拡張子に注意
  Wait(1000);                // 次のuntilまで少し時間をかせぐ

  until (SENSOR_1 == 1);     // タッチセンサを押せばスレーブのプログラムを停止
  RemoteStopProgram(CONN);   // 停止する時はファイル名は不要
}
```

【例】 スレーブ側のプログラムを起動・停止するプログラム

スレーブ側からマスター側のプログラムを起動するには、 上の例の CONN をマスターを表す 0 にしてください。

次に、メッセージを送ってみましょう。 メッセージの種類としては文字列、数値、ブール値を送ることができます。 メッセージのやり取りは、メールボックスを通じて行います。 メールボックスは0番から9番まで10個使うことができ、それぞれ MAILBOX1 から MAILBOX10 という名前を使うこともできます。

例としてロボットAのバンパーを押している間、 ロボットBが前進するプログラムを作ってみましょう。 送るメッセージとしては、数(Number)、文字列(String)などががあります。 ここでは、合言葉としてあらかじめ決めておいた「数」を メッセージとして送受信することでスレーブのモータを回転または停止させてみます。

NXCプログラム（マスター側）

```
#define CONN 1  // スレーブの接続番号
#define SIGNALON 11  // モータ回転のためのメッセージ
#define SIGNALOFF 12 // モータ停止のためのメッセージ

task main ()
{
 SetSensorTouch(S1); // 端子1にタッチセンサ(リモコンボタン)

 while (true) {
  if (SENSOR_1 == 1) {
   SendRemoteNumber(CONN,MAILBOX1,SIGNALON); // MAILBOX1に SIGNALON を送信
  } else {
   SendRemoteNumber(CONN,MAILBOX1,SIGNALOFF); // MAILBOX1に SIGNALOFF を送信
  }
  Wait(100);
 }
}
```

【例】 マスター側のタッチセンサを押しているときだけスレーブを前進させるプログラム（送信側）

NXCプログラム（スレーブ側）

```
#define SIGNALON 11
#define SIGNALOFF 12

task main ()
{
 int msg; // 受け取った値を格納する変数
 while (true) {
  ReceiveRemoteNumber(MAILBOX1,true,msg); // MAILBOX1の値を受け取りmsgに格納
                                          // trueを指定するとメールボックスが空に
  if (msg == SIGNALON) {
   OnFwd(OUT_AC);
  }
  if (msg == SIGNALOFF) {
   Off(OUT_AC);
  }
 }
}
```

【例】 スレーブ側のプログラム

ReceiveRemoteNumber(MAILBOX1,true,msg); の中の true はメッセージを受け取ったあと メールボックスをクリアするためのものですが、この例の場合はfalseでもかまいません。

マスターからスレーブにメッセージを送る場合と、 スレーブからマスターへメッセージを送る場合は若干命令が違うので注意してください。

以下に数を送る命令をまとめておきます。

命令	説明
SendRemoteNumber(1,MAILBOX1,123);	スレーブ1のMAILBOX1に123を送る（マスター側）
SendResponseNumber(MAILBOX1,123);	マスターのMAILBOX1に123を送る（スレーブ側）
ReceiveRemoteNumber(MAILBOX1,true,val);	MAILBOX1から変数valに値を読みこむ（メールボックスをクリア）
ReceiveRemoteNumber(MAILBOX1,false,val);	MAILBOX1から変数valに値を読みこむ（メールボックスをクリアしない）

.

.

.

.

.

次の例は、スレーブがマスターからのメッセージを受信した後、返信をするプログラムの例です。 マスター側のタッチセンサを押す度に、スレーブ側に"Hello, NXT-1!"という文字列が 送られ、それを受信したスレーブが"Hello, NXT-0!"という文字列を送り返します。 そしてそれぞれのディスプレイに受信した文字列を表示させます。

NXCプログラム（マスター側）

```
#define CONN 1          // スレーブ(通信相手)の接続番号
#define OUTBOX MAILBOX4  // 受信用のメールボックス
#define INBOX MAILBOX5   // 送信用のメールボックス

task main ()
{
  SetSensorTouch(S1);
  string reply;  // 返信の文字列を格納する変数

  until(BluetoothStatus(CONN)==NO_ERR);
  RemoteStartProgram(CONN,"test-reply.rxe"); // スレーブのプログラムを起動

  while(true){
    ClearScreen();
    reply = "";    // 文字列replyを空に
    TextOut(0,LCD_LINE1,"Reply");
    TextOut(0,LCD_LINE2,"  from NXT-1:");


    until(Sensor(S1)==1);    // タッチセンサが押されたらメッセージを送信
    SendRemoteString(CONN,OUTBOX,"Hello, NXT-1!");
    TextOut(20,LCD_LINE4,"Waiting ...");

    while(reply == ""){   // 返事が来るまで待つ
      ReceiveRemoteString(INBOX, true, reply);
    }

    ClearLine(LCD_LINE4); PlaySound(SOUND_CLICK);
    TextOut(10,LCD_LINE4,reply);  // 返事のメッセージを表示
    Wait(2000);
  }
}
```

【例】 スレーブからの返事のメッセージを表示するプログラム

NXCプログラム（スレーブ側）

```
//
// test-reply.nxc
//
#define CONN 0          // マスター(通信相手)の接続番号
#define INBOX MAILBOX4    // 送信用のメールボックス
#define OUTBOX MAILBOX5   // 受信用のメールボックス

task main ()
{
  string hello;  // 受信の文字列を格納する変数
  until(BluetoothStatus(CONN)==NO_ERR);

  while(true){
    ClearScreen();
    TextOut(0,LCD_LINE1,"Message");
    TextOut(0,LCD_LINE2,"  from NXT-0:");

    hello = "";   // 文字列helloを空に
    TextOut(20,LCD_LINE4,"Waiting ...");

    while(hello == ""){   // メッセージが来るまで待つ
      ReceiveRemoteString(INBOX, true , hello);   // メールボックスのチェック
    }

    ClearLine(LCD_LINE4); PlaySound(SOUND_CLICK);
```

```
    TextOut(10,LCD_LINE4,hello);    // メッセージを表示
    Wait(1000);

    SendResponseString(OUTBOX,"Hello, NXT-0!");   // 返事を送信
    Wait(2000);
    ClearLine(LCD_LINE4);
  }
}
```

【例】 マスターからのメッセージを表示し、返信するプログラム

練習問題66	スレーブ側に接続した超音波センサの値をマスター側のディスプレイに表示させ、 0.5秒ごとに更新するようにしましょう。

14. サーボ・モータの機能をもっと使おう

NXTに付属のモータはいわゆるサーボ・モータと呼ばれるモータで、 回転数（回転角度）を指定して回転させたり、 左右のモータを正確に同期させたりすることが容易にできます。 また、角度センサとして角度を測ることもできます。 これらの機能を試してみましょう。

モータの回転角度を測る

例えばモータBの回転角度は MotorTachoCount(OUT_B)で得られます(long型)。 まず、モータの回転角度を測ってディスプレイに表示するプログラムを作ってみましょう。

NXCプログラム

```
task main ()
{
  ResetTachoCount(OUT_BC); // 回転角度をリセット
  while(true){
    ClearScreen();
    TextOut(0, LCD_LINE1, "motor B:");
    TextOut(0, LCD_LINE2, "motor C:");
    NumOut(60, LCD_LINE1, MotorTachoCount(OUT_B)); // モータBの回転角を表示
    NumOut(60, LCD_LINE2, MotorTachoCount(OUT_C)); // モータBの回転角を表示
    Wait(100);
  }
}
```

【例】 モータの回転角を表示するプログラム

角度を指定してモータを回転させる

角度を指定してモータを回すための命令もいろいろと用意されています。 例えば、角度を正確に測りながら左右のモータが常に同じ角度を保つようにすること、 つまり左右のモータを同期（シンクロナイズ）させる命令も用意されています。 この機能を使えば、 OnFwd だけでは難しかった「直進」も簡単です。

以下の表はNXCの用意されている関数の一部を抜き出したものです。

命令	説明
ResetTachoCount(OUT_BC);	モータBとCの回転角をリセットする
RotateMotor(OUT_A, 75, 45);	モータAを75%のスピードで45度前転
RotateMotor(OUT_A, -75, 45);	モータBを75%のスピードで45度後転
RotateMotorEx(OUT_BC, 75, 45, 0, true, true);	モータBとCを75%のスピードで同期させて45度前転(ブレーキ有り)
RotateMotorEx(OUT_BC, 75, 45, 0, false, true);	モータBとCを75%のスピードで同期させず45度前転(ブレーキ有り)
RotateMotorEx(OUT_BC, 75, 45, 100, true, false);	モータBとCを75%のスピードで反転同期させて45度前転(ブレーキなし)
OnFwdSync(OUT_BC, 75, 0);	モータBとCを75%のスピードで同期させる
OnFwdSync(OUT_BC, 75, -100);	モータBとCを75%のスピードで反転させる
OnFwdReg(OUT_BC, 75, OUTREG_MODE_IDLE);	負荷がかかると遅くなる
OnFwdReg(OUT_BC, 75, OUTREG_MODE_SPEED);	負荷がかかってもスピードを保とうとする
OnFwdReg(OUT_BC, 75, OUTREG_MODE_SYNC);	モータBCを同期させる（片方が遅くなるともう片方も遅くなる）

同期（シンクロ）させる場合の回転角の次の引数は、回転数の比率で-100から100までの値を取ります。-100または100の場合は完全に反転し、0の場合は同じ方向で同期します。

以前の練習問題で、壁に当たるまでの時間を測ってその同じ時間だけ後進するという プログラムを作りましたが、時間を使う代わりにモータの回転角度を使ってみましょう。

NXCプログラム

```
task main ()
{
  SetSensorTouch(S1);
  ResetTachoCount(OUT_BC); // 回転角度をリセット
  OnFwdSync(OUT_BC,75,0);  // モータBCを同期させて前進

  until(SENSOR_1 == 1);
  long angle=MotorTachoCount(OUT_B);  // 壁に当たったときの回転角度
  RotateMotorEx(OUT_BC, -75, angle, 0, true, true); // 0は左右の同じ比率
}
```

【例】 壁に当たったら元に戻ってくるプログラム

この例で使った RotateMotorEx という命令の引数で、 初めの true は2個のモータを同期させること、 後の true は回転後ブレーキをかけることを指示しています。 スピードを -75 ではなく 75 とした場合は、 angle を -angle にしておけば同じ動作になります。

練習問題67	上の例を RotateMotorEx の代わりに OnFwdSync あるいは OnRevSync を使って書きなおしてみましょう。
練習問題68	上の例でロボットが戻ってきた後、再度前進して、壁に当たる数センチ手前で止まるように プログラムを変更してみましょう。
練習問題69	左右のモータ逆回転してロボットをその場で360度旋回させるとき、 モータを何度回転すればよいか調べてみましょう。

15. オリジナルのロボットへ向けて

ひと通りNXTのプログラミングに慣れたら、次はオリジナル・ロボットを作ってみましょう。

16. 参考文献

ここでは主に、NQCやNXCを使ってマインドストームを動かす上で、参考になるウェブサイトや書籍を紹介します。 最近ではNXCについて取り上げる日本語のウェブサイトも増えていて、 ここで紹介する以外にも素晴らしいサイトがたくさんあるはずです。 ぜひ自分でも探してみましょう。

ウェブ

検索キーワードとしてNQCやNXCのコマンド名を含めると関連ページが見つかりやすいです。

- **LEGO.com Mindstorms Home**（英語） http://mindstorms.lego.com/
  レゴ社のマインドストーム公式サイト。 NXTのファームウェアもここからダウンロードできます。
- **レゴ マインドストーム公式サイト**（日本語） http://www.legoeducation.jp/mindstorms/
  レゴジャパン（株）による教育用レゴマインドストームの紹介。
- **NQC Web Site**（英語） http://bricxcc.sourceforge.net/nqc/
  Dave Baumさんの作成した Mindstorm用コンパイラ NQC のホームページ。 現在は John Hansen さんによってメンテナンスされています。 NQCのソースコードおよび、Mac用、 Windows用のバイナリーがダウンロード可。 ソースをコンパイルすれば GNU/Linux でも使えます。
- **Welcome to Next Byte Codes, Not eXactly C, and SuperPro C**（英語） http://bricxcc.sourceforge.net/nbc/
  NBC, NXC, SPCの配布サイト。ソースと各種バイナリ形式（Win32, Mac OSX, Linux ia32用）をダンロードできます。ドキュメントも豊富です。
- **NQC Programmer's Guide**（英語PDFファイル） http://bricxcc.sourceforge.net/nqc/doc/NQC_Guide.pdf
  NQCの開発者Dave BaumさんとJohn HansenさんによるNQCマニュアル。 まさにNQCの仕様書とも呼べる文書で、 NQCの正確な仕様を調べる時には非常に便利です。
- **Programming Lego Robots using NQC**（日本語PDFファイル） http://www.staff.science.uu.nl/~overm101/lego/tutorial_j.pdf
  Mark OvermarsさんによるNQCの入門書（Alberto Palacios Pawlovskyさんによる日本語訳）

・**NXCを使ったLEGOのNXTロボットのプログラミング**（PDFファイル）
http://www.cc.toin.ac.jp/sc/palacios/courses/undergraduate/freshman/micro_intro/NXCtutorial_j.pdf

Daniele Benedettelliさん製作『Programming LEGO NXT Robots using NXC』 のAlberto Palacios Pawlovskyさんによる日本語訳。

・**NXCを使ったLEGOのNXTロボットのプログラミング**（PDFファイル）
http://www2.ocn.ne.jp/~takamoto/NXCprogramingguide.pdf

上と同じDaniele Benedettelliさんのガイドの高本孝頼さんによる日本語訳。 補足説明付き。

・**NXC Programmer's Guide**（英語） http://bricxcc.sourceforge.net/nbc/nxcdoc/nxcapi/

NXCのすべての命令や定数などが載っているNXCの仕様書のオンライン版。 NQCに比べて桁違いに増えた膨大なコマンドや定数などが説明されています。

・**こうしろうのMindStorms日記**（日本語） http://itpro.nikkeibp.co.jp/article/MAG/20061214/256937/

2001年当時中学生のお子さんをお持ちの金宏和實さんによるブログ形式の特集記事。 前世紀から続いている長期連載記事で、 初期の頃はNQC、数年前はNXC、最近はAndronid、などの話題が中心ですが、 C言語やJava、PHP、SQLなど他の言語の話題もあります。

・**LEt'sGOstudio**（日本語,英語） http://www.isogawastudio.co.jp/legostudio/

五十川芳仁さんのサイト。LEGOのコミュニティではとても有名な方で、 こんなものまでレゴで作れるの？というようなすごい作品をたくさん製作されています。 RCXやNXTを使った作品も豊富です。

・**LUGNET - Robotics**（英語） http://news.lugnet.com/robotics/

LUGNET（LEGO User Group NETwork）は、LEGOに関する膨大な情報を提供している レゴファンによるコミュニティ・サイトです。 その中のロボティクス関連のページもとても充実していて、 さまざまなプログラミング環境についてのリンク集にもなっています。

・**MindStormsの洞窟 - はじめてのBricxCC**（日本語） http://line.to/mac/MindStorms/BricxCC/

MacさんによるBrixCCの解説。BrixCCはNQCやNXCなどマインドストーム用開発環境の 統合ソフトです。エディタやコンパイラの他サウンド入力用の鍵盤などの 支援ツールが一緒になっています。Windows上でしか動かないのが残念。

・**ロボティクス入門ゼミ@信州大学**（日本語） http://yakushi.shinshu-u.ac.jp/robotics/

マインドストームを使った信州大学の一年生向けの授業のページ。 最近はRISだけでなくNXTも使っています。 この講座で使用したDebian Liveシステムのダウンロードもできます。

書籍

（作成中）

・ Dave Baum et al.,『Definitive Guide to LEGO Mindstorms』, ISBN-13: 978-1590590638

・ Jin Sato,『Jin Sato の LEGO MindStorms 鉄人テクニック』, ISBN-13: 978-4274086823

・ Dave Baum et al.,『Extreme Mindstorms』, ISBN-13: 978-1893115842

・ John C. Hansen,『LEGO Mindstorms NXT Power Programming』, ISBN-13: 978-0973864922

・ Mario Ferrari et al.,『Building Robots With Lego Mindstorms』, ISBN-13: 978-1928994671

・Yoshihito Isogawa,『The LEGO Technic Idea Book: Wheeled Wonders』, ISBN-13: 978-1593272784

・Yoshihito Isogawa,『The LEGO Technic Idea Book: Simple Machines』, ISBN-13: 978-1593272777

・Yoshihito Isogawa,『The LEGO Technic Idea Book: Fantastic Contraptions』, ISBN-13: 978-1593272791

・ Mario Ferrari et al.,『Building Robots with LEGO Mindstorms NXT』, ISBN-13: 978-1597491525

・ Daniele Benedettelli,『Creating Cool MINDSTORMS NXT Robots』, ISBN-13: 978-1590599662

・ David J. Perdue,『The Unofficial Lego Mindstorms Nxt Inventor's Guide』, ISBN-13: 978-1593272159

・ 藤吉弘亘, 鈴木裕利, 石井成郎, 藤井隆司, 『実践ロボットプログラミング』, ISBN-13: 978-4764903784