NXC

Version 1.2.1 r5

レゴ・マインドストームNXTを
高度に動かすために

NXC
プログラマーズガイド

2011年7月5日（改定版）

by John Hansen

2011年9月19日（改定版翻訳）

（訳：高本孝頼）

もくじ

はじめに

本プログラマーズガイドは、レゴ・マインドストームNXTの統合開発環境 BricxCC（Bricx Command Center）上で稼動するC言語ライクなNXC（Not eXactly C）のプログラミングをする上での簡易ガイドとなる。このリソースは、BricxCC上のヘルプ機能にある「Guide PFD」（全2000ページ以上）から抽出した内容となる。

これまでレゴ・マインドストームの情報が少ないと感じていた著者が、いろいろと調べた結果、多くの開発事例があるBicxCCとNXCと出会い、その関連情報も海外では豊富に揃っていることに驚かされた。

すでに、著者はNXCチュートリアルガイドを翻訳していて、初心者でも簡単にプログラミングができるガイドとして役立つものと考えている。もちろん、上級者としてのロボット操作上のセンサー操作やモータ制御、サウンド操作、画面操作、ファイル操作などの技術も含まれていて、幅広く活用できるガイドだとも思っている。

しかし、C言語の文法を知る上では、必ずしも情報が広く揃っている訳ではなく、初心者がいろいろとプログラムの拡張で試していると、ところどころでつまづくところが出てくると感じていた。そこで、ヘルプ画面を調べていると、ちょうどいい内容のガイドが見つかり、ここに翻訳することとした。

本ガイドは、標準C言語との違いを明確にしながら、NXCのコンパイラの処理する文法をまとめた資料となる。NXC初心者が、少し拡張してプログラミングしたいと思ったとき、一度はこの資料を読むことを勧める。

本ガイドは、まだ初校版ということで、間違いも多く含まれているものと思っていて、ぜひとも読者からのご指摘を受けながら、校正していく予定でいる。以下のメールアドレスにご意見・ご指摘をお願いするところである。

メールアドレス：takamoto0206@yahoo.co.jp

高本孝頼 2011.08.25

1 NXCプログラマーズ・ガイド

2011年3月13日
著者　John Hansen
•まえがき
• NXC 言語

2 まえがき

NXC は、 Not eXactly C の略称となる。
このNXCは、LEGOマインドストームNXT製品のプログラミングのためのやさしい言語である。NXTは、レゴ社が提供する**バイトコード・インタプリター**（２バイトによるプログラム実行できるもの）を持っていて、これによって実行形式のプログラムを使うことができる。このNXCコンパイラは、NXTのバイトコードに変換でき、ターゲットとする実行形式にすることができる。このNXCのプロセッサや制御構造はC言語によく似たものであるが、NXCそのものは、一般的なC言語ではなく、NXTバイトコード・インタプリターによって多くの制約がある。
NXCは、論理的に２つの部分に分かれている。１つは、**NXC言語**で、プログラム作成での文法規約がある。もう１つは、**NXC API**（アプリケーション・プログラミング・インタフェース）で、 プログラムで使われるシステム関数、定数、それにマクロなどがある。このAPIは、「**ヘッダーファイル**」で知られる特別ファイルで定義されるが、NXCでは省略でき、プログラムのコンパイル時に自動的に挿入される。
この仕様書では、このNXC言語とNXC APIの両方を記述している。要するに、NXCプログラムを作成するために必要な情報をまとめている。NXCでは、異なるインターフェイスのインプリメンテーション方法（コマンドラインのコンパイラとBricx Command Center）を持つが、この仕様書ではこのインプリメンテーションについては記述していない。これらについては、NXCユーザマニュアルに記載されているインプリメンテーション情報などを参照されたし。このNXCの最新の情報や仕様書については、つぎのNXC Webサイトを参照のこと。

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

3 NXC 言語

この章では、NXC言語について記述している。
この中では、コンパイラによる言語ルール、プログラム構造、ステートメント（文）や表現、それにプリプロセッサの使い方をまとめている。NXCは、C言語やC++言語と同じように、

大文字と小文字を区別する言語で、例えば「xYz」と「Xyz」とは識別される。同様に、「if」文は、キーワードと認識されない「iF」や「If」、それに「IF」らとは識別される。
本章では、以下の内容をまとめている。

- **言語ルール**
- **プログラム構造**
- **ステートメント（文）**
- **表現**
- **プリプロセッサ**

3.1 言語ルール

この言語ルールは、NXCのコンパイルがソースファイルを識別する規則となる。
ここでは、コメント記述法、空白文字の使い分け、それに識別のための文字評価についてまとめている。

- コメント
- 空白文字
- 数値定数
- 文字列定数
- 文字定数
- 識別子とキーワード

3.1.1 コメント

NXCでは、2つのコメント記述法がある。1つめは、典型的なコメントで、「/＊」と「*/」を使う方法となる。これらは、複数行に渡ってコメントとすることができるが、入れ子にすることはできない。

```
/* this is a comment（ここがコメント） */
/* this is a two （2行にまたがる）
line comment（コメント） */

/* another comment（他のコメント）...
  /* trying to nest（入れ子を試す）...
   ending the inner comment...（入れ子のコメントの終わり）*/
   this text is no longer a comment!（このテキストはコメントとならない） */
```

2つめのコメント記述は、「//」を記入し、行の終わりまでをコメントとする方法となる。この方法は、C++言語スタイルのコメントと同じ方法となる。

```
// a single line comment（ひとつの行のコメント文）
```

当然、これらコメントにした範囲は、コンパイラでは無視される。これらは、プログラムのソースコードを作成するところだけで使われる。

3.1.2 空白文字

ここでの空白文字には、スペースやタブ、それに改行といったものが含まれる。これらは、分割される**トークン**（識別する単位の語や変数名、演算子など）や、プログラムをより読みやすくするために使われる。この空白文字は、プログラム上でトークンが識別される範囲で、追加したり削除したりすることができる。例えば、つぎの行にある両方の意味は同じとなる。

```
x=2;
x = 2 ;
```

C++言語での複数文字は、いくつか意味を持つものがある。これらのトークンを保持するために、文字間に空白文字を含めない。例えば、以下のように、最初の行は、「<<」を使って対応しているが、つぎの行は「<」を空白文字で分けているためにコンパイルエラーとなる。

```
x = 1 << 4; // set x to 1 left shifted by 4 bits（xに1を4ビット左シフトして設定）
x = 1 < < 4; // error
```

3.1.3 数値定数

数値定数は、10進数か16進数のいずれかで記載される。10進数定数は、１つかそれ以上の10進数値で表記する。小数点を表すポイントを数値内に記載できる。16進数値では、先頭に「0x」か「0X」を含ませて表記する。

```
x = 10; // set x to 10 （10を設定）
x = 0x10; // set x to 16（10 hex）（16進数の10、10進数では16を設定）
f = 10.5; // set f to 10.5（実数値の10.5を設定）
```

3.1.4 文字列定数

NXCでは、C言語と同様に、ダブルクォート「"」で囲んで文字列を定義する。NXCは、C言語よりも簡単に、文字列を持たせることができる。NXCの文字列では自動的にバイト配列に変換され、配列の最後にゼロ（文字列の終わりの意味）を挿入する。この最後のゼロ（バイト）は、一般にnull文字として解釈される。

```
TextOut(0, LCD_LINE1, "testing");
```

3.1.5 文字定数

NXCでの文字定数は、シングルクォート「'」で囲まれた単一のアスキー文字となる。
この文字定数の値は、文字を数値で表したアスキーの値となる。

```
char ch = 'a' ; // ch == 97
```

3.1.6 識別子とキーワード

識別子は、変数名やタスク名、関数名、それにサブルーチン名として使われる。この識別子の定義において、先頭文字は大文字か小文字の英文字、もしくはアンダーライン「_」を使う。残りの文字群には、英文字か数値、もしくはアンダーラインを使う。
多くのトークンは、NXC言語自体で予約されている。これらを**キーワード**と呼び、識別子として使うことはできない。キーワードは、以下のようなリストになっている。

- 「asm」ステートメント
- 「bool」（ブーリアン）
- 「break」ステートメント
- 「byte」（バイト）
- 「case」ラベル
- 「char」（文字）
- 「const」（定数）
- 「continue」ステートメント
- 「default」ラベル
- 「do」ステートメント
- 「if-else」ステートメント
- 「enum」キーワード
- 「false」（ブーリアンの偽）
- 「float」（実数）
- 「for」ステートメント
- 「goto」ステートメント
- 「if」ステートメント
- 「inline」キーワード
- 「int」（16ビット整数）
- 「long」（32ビット整数）
- 「mutex」キーワード
- 「priority」ステートメント
- 「repeat」ステートメント
- 「return」ステートメント
- 「safecall」キーワード
- 「short」（16ビット整数）
- 「start」ステートメント
- 「static」
- 「stop」ステートメント
- 「string」（文字列）
- 「Structures」（構造体）
- 「sub」キーワード
- 「switch」ステートメント
- 「tasks」（タスク）
- 「true」（ブーリアン：真）
- 「typedef」
- 「unsigned」
- 「until」ステートメント
- 「void」キーワード
- 「while」ステートメント

3.1.6.1 「const」(定数)

この「const」キーワードは、 変数宣言されたものを数値に置き換えて使うもので、コンパイル時に初期化された段階で、値の変更はできなくなる。初期化において、変数宣言で行われた値が入る。

```
const int myConst = 23; // declare and initialize constant integer
task main() {
  int x = myConst; // this works fine
  myConst++; // compiler error - you cannot modify a constant's value
}
```

3.1.6.2 「enum」

この「enum」キーワードは、名前[name]を列挙タイプ {name-list} して作成するときに利用する。文法は以下の通りとなる。

```
enum [name] {name-list} var-list;
```

この列挙タイプは、名前リスト（name-list）の要素で構成される。ここでの「var-list」は、オプションで、宣言する変数名を表記する。
例として、つぎに列挙タイプとして色設定の宣言を行っている。

```
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
```

上記の例での「enum」（列挙）を使う効果としては、いくつかの定数定義をする際に切り替えことが簡単になる。省略値は、先頭の値にゼロで、そのあとは継続した整数に置き換わる。もし、必要に応じて、値を変更したい場合には、つぎのようする。

```
enum ColorT { red = 10, blue = 15, green };
```

上記の例では、緑は16（green=16）に設定される。一旦、この列挙タイプを使って宣言すること識別子などと同じように、変数などに設定することができるようになる。
ここで、いくつかの「enum」キーワードの事例を紹介しよう。

```
// values start from 0 and increment upward by 1 (それぞれ０，１，２が設定)
enum { ONE, TWO, THREE };
// optional equal sign with constant expression for the value (各独自設定)
enum { SMALL=10, MEDIUM=100, LARGE=1000 };
// names without equal sign increment by one from last name's value (１から)
enum { FRED=1, WILMA, BARNEY, BETTY };
// optional named type (like a typedef) (名前として宣言)
enum TheSeasons { SPRING, SUMMER, FALL, WINTER };
// optional variable at end (最後でオプション変数)
enum Days {
  saturday, // saturday = 0 by default
  sunday = 0x0, // sunday = 0 as well
  monday, // monday = 1
  tuesday, // tuesday = 2
  wednesday, // etc.
  thursday,
  friday
} today; // Variable today has type Days (変数名todayをDays宣言)
Days tomorrow;  // (変数名tomorrowをDays宣言)
task main()
{
  TheSeasons test = FALL;
  today = monday;
  tomorrow = today+1;
  NumOut(0, LCD_LINE1, THREE);
  NumOut(0, LCD_LINE2, MEDIUM);
  NumOut(0, LCD_LINE3, FRED);
  NumOut(0, LCD_LINE4, SPRING);
  NumOut(0, LCD_LINE5, friday);
  NumOut(0, LCD_LINE6, today);
```

```
  NumOut(0, LCD_LINE7, test);
  NumOut(0, LCD_LINE8, tomorrow);
  Wait(SEC_5);
}
```

3.1.6.3 「static」

「static」キーワードは、関数内の変数定義で静的に置き換える宣言で使われ、その変数値はプログラムで呼ばれる関数間で引き継がれる。つまり、「static」キーワードが無い場合には、関数内のみで宣言された変数となる。

注意すべき点として、初期値設定は、この「static」キーワードを付けると全く違ったものとなる。「static」キーワードが無い一般的な変数での初期値設定では、実行される毎に初期化が行われる。「static」キーワードが付いた場合の初期設定では、グローバル変数として扱われ、コンパイル時の設定となり、初期値は実行ファイルに組み込まれた簡単なものとなる。

```
void func() {
  static int x = 0; // x is initialized only once across three calls of func()
  NumOut(0, LCD_LINE1, x); // outputs the value of x
  x = x + 1;
}

task main() {
  func(); // prints 0
  func(); // prints 1
  func(); // prints 2
}
```

3.1.6.4 「typedef」

この「typedef」宣言は、その範囲（スコープ）の中で定義される「type-declaration」を、同意語としての名前「synonym」に割り当てる。

```
typedef type-declaration synonym;
```

この「typedef」宣言を使うことで、長い宣言などを短くして分かりやすい宣言文字にして使うことがでる。この「typeded」の名前は、要約した文字にも変更できる。ただ、既に宣言された名前を、改めて使うことはできない。ここでは、どう「typedef」キーワードを使うかの事例をいくつか紹介する。

```
typedef char FlagType;
const FlagType x;  //  x を const charで宣言している
typedef char CHAR; // Character type.
CHAR ch;           // ch  を charで宣言している。
typedef unsigned long ulong;  // unsigned long を ulong に割り当てている。
ulong ul; // Equivalent to "unsigned long ul;"
```

3.2 プログラム構造

NXCプログラムは、**コード群**と**変数**とで構成される。コード群は、タスクと関数の２つの区別がある。互いコード群の名前は同じものはなく、唯一で、シェアして利用される。タスクと関数のコード群の最大数は256個となる。

- **コード様式**
- **タスク**
- **関数**
- **変数**
- **構造**
- **配列**

3.2.1 コード様式（Code Order）

NXCでは、２つのコード様式を持っている。１つは、**ソースコード**ファイルのコード様式で、もう１つが実行ファイルのコード様式となる。前者は人間が読める言語的な**アスキー**（テキスト）様式で、後者は機械が実行するための**バイナリー**様式となる。

この言語的なアスキー様式は、NXCコンパイラで関係するが、NXTブロックでは関係なくなる。つまり、記述するタスクや関数の定義は、実行するバイナリー様式では影響なくなることを意味する。コンパイルするときの言語的な様式の**制御ルール**は、以下のようになる。

１．　タスクや関数の名前は、１つのコード群で使われる前に宣言しておく必要がある。
２．　タスクが実行する時間は、API「コマンドモジュール関数」の仕様で決定される。
３．　１つの関数は、コードの他のブロックから呼ばれて起動する。

この最後のルールは、ささいなことだが、マルチでタスクが実行する場合にはとても重要となる。もしあるタスクが、すでに他のタスクで先に呼び込まれて実行中である関数を呼び出した場合、どう挙動しどのような結果になるか予測できない。タスクが関数を共有して実行するとき、リソースを共有させるか、「mutex」を使って、他のタスクが使っている関数を呼び出す際の衝突を避けるようにする。また自分を呼び出す「safecall」キーワード（関数「Function」参照）を使って、コーディングを簡単にすることもできる。

言語的な**順序ルール**は、

１．　どんな識別できる名前をつけたタスクや関数は、コード群の中で使われる前に、コンパイラに知らせておく必要がある。
２．　タスクや関数の定義は、コンパイラにはそれぞれ識別さる名前を付けること。
３．　タスクや関数の宣言もまた、コンパイラには識別される名前であること。
４．　一度定義されたタスクや関数は、再定義や宣言はできない。
５．　一度宣言されたタスクや関数は、再宣言はできない。

ときどき、タスクや関数の宣言を不可能なところや都合の悪いところに使って実行しようとすると、コンパイラはコード群の中で使われる前に、タスクや関数の名前を知ることになり、エラーを出す。この場合、つぎのようなタスクや関数の宣言を、最初に使われるところのコード群の前に、挿入することができる。

```
task name();     // タスクの場合
return_type name(argument_list);  // 関数の場合 例： int func(int i);
```

この「argument_list」は、後で出てくる実際の関数で定義されたリストと一致しなければならない。

3.2.2 タスク（Tasks）

NXTがサポートしている**マルチスレッド**（マルチタスク環境下で、個々のタスクの中の独立した複数処理の流れ）により、NXCの中のタスクは、NXTスレッドに直接通信することになる。
タスク群は、以下のコード例で示される文法に沿って、「**task**」キーワードを使って宣言する。

```
task name()
{
    // the task's code is placed here  ここにタスクのコードが記載される
}
```

このタスク名は、文法上互いに識別できる必要がある。また１つのプログラムには、プログラムが実行スタートするときに使う「**main**」というタスク名を、ひとつ持つ必要がある。
タスク本体は、ステートメント（文）のリストで構成される。
プログラムのスタートとストップは、以下の事例のように、タスク内に「**Precedes**」や「**stop**」などのステートメントを使ってでもできる。しかし、タスクがスタートする基本的なメカニズムは、「Precedes」やつぎの API関数 で計画されたタスクに依存する。API関数の「**StopAllTasks**」は、現在実行中のすべてのタスクをストップさせる。もちろん「stop」を使ってもストップできる。ひとつのタスクは、「**ExitTo**」関数を使って自分自身をストップさせることもできる。当然、タスクのコード群の終わりになってもストップする。以下のコードのサンプルでの「main」タスクは、「music」タスクと「movement」タスク、さらに「controller」タスクを宣言し、これら３つのタスクが同時にスタートすることを設定している。ここでの「controller」タスクは、「music」タスクをストップする前に１０秒間待機し、その後さらにすべてのタスクを終了させるまで5秒間待機させている。

```
task music() {
  while (true) {
    PlayTone(TONE_A4, MS_500);
    Wait(MS_600);
  }
}
task movement() {
  while (true) {
    OnFwd(OUT_A, Random(100));
    Wait(Random(SEC_1));
  }
}
```

```
task controller() {
  Wait(SEC_10);
  stop music;
  Wait(SEC_5);
  StopAllTasks();
}
task main() {
  Precedes(music, movement, controller);
}
```

3.2.3 関数（Functions）

ときどきステートメント群をひとつのグループとして、必要に応じて呼び出される関数にまとめることで便利になることがある。NXCにおいて、関数は引数とリターン値（戻り値）によって定義できる。関数はつぎの文法に従って定義できる。

```
[safecall] [inline] return_type name(argument_list)
{
  // body of the function
}
```

ここでのリターン値の型（return_type）は、データの戻り値の型となる。Cプログラミング言語では、関数のリターン値の型は、明示しておく必要がある。関数にリターン値が無い場合の型は「void」を使う。

「safecall」、「inline」および「void」のキーワードの詳細については、後に追加として述べる。

- 「safecall」キーワード
- 「inline」キーワード
- 「void」キーワード

関数の引数リスト「argument_list」には、空か、１つかそれ以上のものを定義することができる。ひとつの引数は、型（タイプ）に続けて名前を入れて定義する。コンマ「,」は引数を分割するしるしとなる。ここでの型は、「bool」,「char」,「byte」,「int」,「short」,「long」,「unsigned int」,「unsigned long」,「float」,「string」,「struct」タイプ、または多くのタイプの配列からなる。

NXCでは、この引数の中で、「struct」（構造体）や配列以外では、省略値を持たせることができるようになっている。簡単に引数のあとに続けて「=」（イコール）を使い、さらにそのあとに値を記載して定義する。この引数の値は、関数が呼ばれたときに、設定されることになる。ここでの値を使う場合は、引数群の後ろから定義していく必要がある。

```
int foo(int x=10, int y = 20)
{
  return x*y;
}
task main()
{
  NumOut(0, LCD_LINE1, foo(10, 5)); // outputs 50
  NumOut(0, LCD_LINE2, foo(10));    // outputs 200
```

```
  NumOut(0, LCD_LINE3, foo());      // outputs 100
  Wait(SEC_10); // wait 10 seconds
}
```

NXCの関数を**通過**するパラメータには、値（value）を使ったり、定数(const)を使ったり、参照したり、さらに定数を参照したりすることができる。これら４つのモードによる関数の通過パラメータについて、以下に説明する。

呼び出す関数やタスクから、呼び出される関数までの間に、引数の値が通過する際、コンパイラは値を保持するために一時的に値を割り当てる必要がある。この場合、割り当てられる値の型（タイプ）の制限は無い。しかし、関数が実際の引数をコピーして実行することから、呼び出す側は、呼ばれる関数に値を持っていくのに、どう変更するかが見えなくなる。例えば、以下のように、関数 foo において、引数に「２」を設定している。これは文法上まったく問題は無いが、関数 foo 元の引数を複写して実行しようとした際、タスク mainで値ｙが設定されたものは、変更されないままとなる。

```
void foo(int x)
{
    x = 2;
}
task main()
{
    int y = 1; // y is now equal to 1 （ｙに値１を設定）
    foo(y); // y is still equal to 1! （関数を通過しても、ｙの値は1のまま）
}
```

つぎの2番目のタイプ「**const** arg_type」の場合も、先の値（value）と同じように通過する。もし関数が「inline」関数であれば、このような引数は、ときどきコンパイラによって真の定数とみなされ、コンパイル時に評価されることなる。もし関数が「inline」関数でなければ、コンパイラは、定数や値が通過しても引数として許可し、定数として参照される。コンパイル時に問題なく評価された関数の引数は、あるNXCのAPI関数では、引数が真の定数として働くことになることが重要である。

```
void foo(const int x)  //  引数 x を 整数(int)の定数として宣言
{
   PlayTone(x, MS_500);
   x = 1; // error - cannot modify argument (fooの関数内ではｘは定数でエラーとなる)
   Wait(SEC_1);
}
task main()
{
   int x = TONE_A4;
   foo(TONE_A5); // ok
   foo(4*TONE_A3); // expression is still constant
   foo(x); // x is not a constant but is okay
}
```

３番目の「arg_type **&**」は、値そのものではなく、参照された引数が通過することとなる。呼び出す側の関数は値に変更し、リターン後は、呼び出される側の関数でも引数の値が利用できるようする。

```
void foo(int &x)  // xは、引数として参照されるし、変更された値も返す。
{
  x = 2;
}
task main()
{
  int y = 1; // y is equal to 1
  foo(y); // y is now equal to 2  （yの値は、2に変更されている）
  foo(2); // error - only variables allowed
}
```

４番目の型「const arg_type &」は、とくに興味深いものとなる。これらの値も参照されて通過していくが、呼ばれる方の関数は、定数ということで値の変更を許可しない。この制限については、コンパイラが変数以外は何でも通過させ、関数はこの引数の型（タイプ）を利用することになる。NXTファームウェアの制限で、NXCで通過する参照された引数はC言語ほど最適化にはならない。
引数の複写が行われるが、値が呼ばれた関数内では、その値が変更されないように、コンパイラが制約を掛けることになる。
関数は、引数の現在の数値と型（タイプ）を使って実行されなければならない。つぎに示すコード例では、「foo」関数を呼び出す点で、正しいか正しくないかの違いが見えてくる。

```
void foo(int bar, const int baz)
{
  // do something here...  （このfooの関数内では、bazは定数のままで利用さる）
}
task main()
{
  int x; // declare variable x  ( x は、変数で宣言)
  foo(1, 2); // ok  （定数で宣言しているので問題なし）
  foo(x, 2); // ok
  foo(2); // error - wrong number of arguments!
}
```

3.2.3.1 「safecall」キーワード

関数のリターン値の型（タイプ）として先に明示されるべきオプションキーワードが、「**safecall**」キーワードとなる。
もし、関数が「safecall」を使っていた場合、コンパイラは「Acquire」と「Release」の中で呼ばれた関数と同じように、クロスした並列なスレッドで、この関数は同期をとって実行することになる。つまり、「safecall」が付いた関数は、自分自身が複数呼ばれた場合には、先に実行されているものがあれば、つぎの実行は待機状態となる。
以下のコード例では、「safecall」キーワードをどう使って、並列なスレッドで共有化して、同期を取ってどう実行しているかが分かる。

```
safecall void foo(unsigned int frequency)
{
  PlayTone(frequency, SEC_1);
  Wait(SEC_1);
}
```

```
task task1()
{
  while(true) {
    foo(TONE_A4);
    Yield();
  }
}
task task2()
{
  while(true) {
    foo(TONE_A5);
    Yield();
  }
}
task main()
{
  Precedes(task1, task2);
}
```

3.2.3.2 「inline」キーワード

NXCでは、「inline」関数をオプションとして利用できる。

この「inline」で定義された関数は、コンパイル時に、それぞれ呼ばれているところにコードがコピーされることになる。この「inline」関数を、下手に使うと、ファイルサイズが膨れ上がることになる。また、「inline」を使わない関数は、実際のNXTサブルーチンが作成され、NXCコードでの呼び出しでは、NXTサブルーチンの呼び出し時間が掛かってしまう。

この「inline」を使わない関数（サブルーチン）やタスクのすべてのものは、256個までの制限がある。

以下のコード例は、サブルーチンを呼び出す代わりに、「inline」を使った関数を呼び出している。

```
inline void foo(unsigned int frequency)
{
  PlayTone(frequency, SEC_1);
  Wait(SEC_1);
}
task main()
{
  foo(TONE_A4);
  foo(TONE_B4);
  foo(TONE_C5);
  foo(TONE_D5);
}
```

この「main」タスクでは、「inline」関数を４回使って呼び出すことで、４回のPlayToneと４回のWaitを呼び出している。

3.2.3.3 「void」キーワード

この「void」キーワードは、関数定義で、リターン値が無いことを意味する。何の値も返さない関数は、ときどきプロシジャやサブルーチンと同じように参照される。「sub」キーワードは、この「void」を使ったものと同じとなる。この両方のキーワードは、関数の宣言や

定義において利用される。C言語と違い、変数タイプの宣言では、この「void」は使えない。NQC内では、引数は持つがリターン値を持たない「inline」関数宣言で、この「void」キーワードは使えていた。NXC内の「void」関数は、NQCの中にあったような、自動的に「inline」になることはない。もし「inline」を使いたい場合には、上述の「Function（引数）」節で述べたように、この「inline」キーワードをリターン値の型（タイプ）の前に使って行うことになる。

・「sub」キーワード

3.2.3.3.1 「sub」キーワード：リターン値のデータが無い関数定義に利用

どんなリターン値も無い関数は、プロシジャやサブルーチンと同等に扱われる。

この「sub」キーワードは、「void」キーワードの別称となる。両方のキーワードは、関数の宣言か定義で使われる。

NQCでは、このキーワードは、引数を持たず、リターン値も無い場合のサブルーチンの定義で使われていた。C言語との互換性のためには、リターン値が無い関数を定義する場合には、この「void」を使うことが望ましい。

3.2.4 変数値（Variables）

NXCでの全ての変数は、以下のタイプのひとつを使って定義される。

- bool（８ビットブーリアン）
- byte（８ビット：バイト）
- char（８ビット：文字）
- int（１６ビット：整数）
- short（１６ビット：整数）
- long（３２ビット：整数）
- unsigned（０以上の整数）
- float（実数）
- mutex（mutex変数）
- string（文字列）
- Structures（構造体）
- Arrays（配列）

ここでの変数は、これらのキーワードを使った必要な型と変数名のリストを、スペースとカンマ（「,」）で宣言し、セミコロン「;」で終了する。オプションとして、初期値のある変数には、変数の後にイコール「=」を使って設定する。いくつかの事例を以下に紹介する。

```
int x; // declare x
bool y,z; // declare y and z
long a=1,b; // declare a and b, initialize a to 1
float f=1.15, g; // declare f and g, initialize f
int data[10]; // an array of 10 zeros in data
bool flags[] = {true, true, false, false};
string msg = "hello world";
```

グローバル変数は、プログラム範囲（いくつものコード群の外）で宣言する。
一度宣言したこれらは、その中に含まれるすべてのタスク、関数、およびサブルーチンで利用できる。これらの範囲は、宣言文から始まり、プログラムの終了で終わる。
ローカル変数は、タスクや関数内で宣言する。
これらの変数は、宣言されているコード群の中だけでアクセスできる。特殊な場合として、これらの範囲は、宣言されたところで始まり、コード群の終了で終わる。ローカル変数の場合には、「 {」と「} 」で囲まれた範囲が有効範囲となる。

```
int x; // x is global
task main()
{
  int y; // y is local to task main
  x = y; // ok
  { // begin compound statement
    int z; // local z declared
    y = z; // ok
   }
  y = z; // error - z no longer in scope
}
task foo()
{
  x = 1; // ok
  y = 2; // error - y is not global
}
```

3.2.4.1 bool（ブーリアン）

NXCの中「bool」型は、シングルの８ビット値となる。
普段は、「０」（false）か「１」(true)の値を使うが、ゼロから「UCHAR_MAX」までの値を持つことができる。

```
bool flag=true;
```

3.2.4.2 byte（バイト）

NXCの中「byte」型は、シングルの８ビット値となる。
この型の範囲は、ゼロから「UCHAR_MAX」までの値を持つことができる。また、つぎのように「unsigned」キーワードを使った８ビット値の「char」型と同様に定義することもできる。

```
byte x=12;
unsigned char b = 0xE2;
```

3.2.4.3 char（文字）

NXCの中の「char」型は、シングルの８ビット値となる。
この型の範囲は、「SCHAR_MIN」から「SCHAR_MAX」までの値を持つことができる。
この「char」型は、ときどきシングル文字のアスキー「ASCII」値でストアされて使われる。
「Character定数」のページにて、さらにこの使い方を詳しく紹介している。

```
char ch=12;
```

```
char test = ' A' ;
```

3.2.4.4 int（16ビット整数）

NXCでの「int」型は、シングルの16ビット値となる。
この型の範囲は、「INT_MIN」から「INT_MAX」までの値を持つことができる。「unsigned」キーワードを付けた16ビット値の「int」型も宣言できる。この「unsigned int」の値の範囲は、ゼロから「UNIT_MAX」までの値を持つことができる。

```
int x = 0xfff;
int y = -23;
unsigned int z = 62043;
```

3.2.4.5 short（１６ビット整数）

NXCでの「shot」型は、シングルの16ビット値となる。
この型の範囲は、「SHRT_MIN」から「SHRT_MAX」までの値を持つことができる。この「short」は、「int」の別名となる。

```
short x = 0xfff;
short y = -23;
```

3.2.4.6 long（３２ビット整数）

NXCでの「long」型は、シングルの３２ビット値となる。
この型の範囲は、「LONG_MIN」から「LONG_MAX」までの値を持つことができる。
符号なし16ビット値を宣言するとき、「long」の前に「unsigned」キーワードを利用する。
この「unsigned long」値の範囲は、ゼロから「ULONG_MAX」までとなる。

```
long x = 2147000000;
long y = -88235;
unsigned long b = 0xdeadbeef;
```

3.2.4.7 unsigned（符号なし）

「unsigned」キーワードは、「char」、「int」、「long」の型で使うことができる。
この「unsigned」型は、８ビット、１６ビット、３２ビットのデータで、符号として使う１ビットまでフルに数値として利用することになる。つまり、マイナス側を含めて２倍のエリアで値を使うことができるようになる。

```
unsigned char uc = 0xff;
unsigned int ui = 0xffff;
unsigned long ul = 0xffffffff;
```

3.2.4.8 float（実数）

NXCでは、３２ビット（IEEE754）のシングル精度の実数表現の「float」型を持つことができる。このデータは、32ビット（４バイト）を占有するバイナリ―フォーマットで、仮数部が２４ビット（１０進数で約７桁）の精度となる。浮動小数点の算術は、整数のものより遅くなるので、もし必ずしも実数型が必要でなければ、整数型を使うことを薦める。

標準のNXTファームウェアでは、実数型を使う上での「sqrt」関数を提供している。拡張した「NBC/NXC」ファームウェアでは、C言語の数学ライブラリの標準的な多くの組込み命令コードを持つ。

```
float pi = 3.14159;
float e = 2.71828;
float s2 = 1.4142;
```

3.2.4.9 mutex

NXCでは、複数スレッドでのアクセス共有を承認して協調利用する３２ビット値のmutex型を持つ。

この「mutex」があるのは、タスクや関数内でmutex変数を宣言することが出来ないからである。グローバル変数として設定することで、すべてのタスクや関数において、「Acquire」や「Release」を使った処理内が排他的な処理となり、他のタスクや関数が動かないようにしている。

```
mutex motorMutex;
task t1()
{
  while (true) {
    Acquire(motorMutex);
    // use the motor(s) protected by this mutex.
    Release(motorMutex);
    Wait(MS_500);
  }
}
task t2()
{
  while (true) {
    Acquire(motorMutex);
    // use the motor(s) protected by this mutex.
    Release(motorMutex);
    Wait(MS_200);
  }
}
task main()
{
  Precedes(t1, t2);
}
```

3.2.4.10 文字列（string）

NXCの文字列型は、簡単に宣言でき、最後に０かnullが入った配列で成り立っている。

例えば、NXCメールボックスやファイル、それにLCD（画面）に文字列を書き込むことができる。文字列定数が入った文字列変数を初期化することができる。それでは、追加情報としての「string」構成を参考されたし。

```
string msg = "Testing";
string ff = "Fred Flintstone";
```

3.2.5 構造体（Structures）

NXCでは、ユーザ定義による「struct」として知れている
NXC supports user-defined aggregate types known as structs.
These are declared very much like you declare structs in a C program.

```
struct car
{
  string car_type;
  int manu_year;
};
struct person
{
  string name;
  int age;
  car vehicle;
};
person myPerson;
```

構造体型を定義したのち、新しい型の宣言において、変数や入れ子（ネストした）の構造体に使うことができる。構造体の中のメンバー（やフィールド）は、ドット「. 」記号を使って利用する。

```
myPerson.age = 40;
anotherPerson = myPerson;
fooBar.car_type = "honda";
fooBar.manu_year = anotherPerson.age;
```

同じタイプの構造体を割り当てることができるが、コンパイラでは型のタイプの一致を見ていないので、異なる構造体を利用した場合には、コンパイラではエラーになる場合がある。

3.2.6 配列（Arrays）

NXCでも配列をサポートしている。
配列も通常の変数と同じように宣言されるが、変数名のあとにオープンとクローズの角括弧を付ける。

```
int my_array[]; // declare an array with 0 elements
```

２つ以上の配列を宣言する場合には、ペアの角括弧を重ねて対応する。NXCでの最大の配列は、４次元までとしている。

```
bool my_array[][]; // declare a 2-dimensional array
```

２次元配列までの初期化の宣言は、つぎのように行う。

```
int X[] = {1, 2, 3, 4}, Y[]={10, 10}; // 2 arrays
int matrix[][] = {{1, 2, 3}, {4, 5, 6}};
string cars[] = {"honda", "ford", "chevy"};
```

配列の要素群は、配列での位置関係を明確にしたインデックスを使うようになる。この場合の要素の最初のインデックスはゼロ「０」からスタートし、つぎが「１」と続く。例えば、つぎのようになる。

```
my_array[0] = 123; // set first element to 123
my_array[1] = my_array[2]; // copy third into second
```

ローカルな配列や多次元配列の初期化は、「ArrayInit」関数を使って行うことができる。つぎの例では、この「ArrayInit」関数を使って２次元配列の初期化を行っている。また、その他の配列関連のAPI関数とその使い方についても紹介している。

```
task main()
{
  int myArray[][];
  int myVector[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  byte fooArray[][][];
  ArrayInit(myArray, myVector, 10); // 10 vectors
  ArrayInit(fooArray, myArray, 2); // 2 myArrays
  fooArray[1] = myArray;
  myArray[1][4] = 34;
  int ax[], ay[];
  ArrayBuild(ax, 5, 7);
  ArrayBuild(ay, 2, 10, 6, 43);
  int axlen = ArrayLen(ax);
  ArraySubset(ax, ay, 1, 2); // ax = {10, 6}
  if (ax == ay) {
    // compare two arrays
    NumOut(0, LCD_LINE1, myArray[1][4]);
  }
}
```

NXCでは、グローバルおよびローカルの配列の初期サイズを明示にする必要がある。コンパイラでの配列の初期化ではゼロ「０」が入る。もし、配列宣言にサイズ設定と初期値設定をしていた場合には、サイズは初期値で用意されたもので決まり、設定されているサイズは無視される。

```
task main()
{
  int myArray[10][10];
  int myVector[10];
  //ArrayInit(myVector, 0, 10); // 10 zeros in myVector
  //ArrayInit(myArray, myVector, 10); // 10 vectors myArray
}
```

上記の場合、さきに初期サイズがある配列を宣言した場合には、「ArrayInit」の呼び出しは不要となる。つまり、すでに配列宣言でゼロ設定の初期化がされているからとなる。実際

に、上記の「myVector」配列の初期化宣言は、「myArray」配列の初期化で「myVector」を使っているのは別として、必要とはならない。

3.3 ステートメント：文（Statements）

コード群（タスクや関数）の本体は、ステートメントで組み合わさる。
ステートメントは、セミコロン「；」を使って終了する。事例を以下に示す。

- 変数宣言
- 割り当て（Assignment）
- 制御構造（Control Structures）
- 「asm」ステートメント
- その他 NXC ステートメント

3.3.1 変数宣言（Variable Declaration）

変数宣言は、すでに記述したが、これもひとつのステートメントとなる。コード群の中で使う場合にも、（任意の初期化して）ローカル変数を宣言することとなる。変数宣言の文法は、以下の通りとなる。

```
arg_type variables;
```

ここでの「arg_type」は、NXCでサポートされている型（タイプ）の中のひとつでなければならない。つぎの型は、変数名とオプションの「＝」付きの初期値を使い、さらにコンマ「,」で区切ったリストでコードを作成する。

```
name[=expression]
```

変数の配列も同様に宣言する。

```
int array[n][=initializer];
```

また、ユーザ作成の集合した構造体型の変数宣言もできる。

```
struct TPerson {
  int age;
  string name;
};
TPerson bob; // cannot be initialized at declaration
```

3.3.2 割当て（Assignment）

一度宣言した変数には、以下のプログラム事例に示す文法表現を使って値を割り当てる。

```
variable assign_operator expression;
```

NXCでは、９個の異なる割り当て演算子がある。もっとも基本的な演算子は、「＝」で、変数に値を割り当てる。その他の演算子については、以下の一覧表にて紹介する。

演算子	実　施
=	表現したものを変数に設定
+=	表現した内容を変数に加えて設定
-=	変数から表現した内容を引いて設定
*=	変数に表現した内容を掛けて設定
/=	変数を表現した内容で割って設定
%=	変数を表現した内容で割った余りを設定
&=	変数とのビットのANDを取って設定
|=	変数とのビットのORを取って設定
^=	変数とのビット排他ORを取って設定
||=	絶対値の設定
+-=	変数の符号（-1，＋１，０）を設定
>>=	変数のビットの右移動をして設定
<<=	変数のビットの左移動をして設定

表 3. 演算子

つぎのコード例は、NXCで利用できるいくつかの演算子を紹介する。

```
x = 2; // set x to 2
y = 7; // set y to 7
x += y; // x is 9, y is still 7
```

3.3.3 制御構造（Control Structures）

NXCのタスクや関数は、ネスト（入れ子）の構造を持つことがある。つぎに制御構造のいくつかのタイプを紹介する。

- 複合ステートメント
- 「if」ステートメント
- 「if-else」ステートメント
- 「while」ステートメント
- 「do」ステートメント
- 「for」ステートメント
- 「repeat」ステートメント
- 「switch」ステートメント
- 「goto」ステートメント
- 「until」ステートメント

3.3.3.1 複合ステートメント（重文）

最も単純な制御構造は、複合ステートメント（重文）となる。
このステートメントのリストは、中括弧('{' と '}')で囲まれて表現する。

```
{
  x = 1;
  y = 2;
}
```

これだけでは、特に重要に見えないが、他の制御構造の中でよく使われることになる。多くの制御構造は、ボディとして単体のステートメントで表現されが、実際には複数のステートメント群で利用されることになる。

3.3.3.2 「if」ステートメント

この「if」ステートメントは、状態(condition)の判断として評価される。
もし状態が真（true）の場合には、つぎのひとつのステートメント（consequence）を実行する。状態の値が、ゼロ「０」の場合には、偽（false)とみなす。もしゼロ以外の場合には、真（true）とみなす。以下に「if」ステートメントの文法を表記する。

```
if (condition) consequence
```

「if」ステートメントの状態（condition）は、以下に示すように、丸括弧で囲んで表現する。つぎのステートメント「consequence」が複数行になる場合には、複合ステートメントとして中括弧を付けて表現する。

```
if (x==1) y = 2;
if (x==1) { y = 1; z = 2; }
```

3.3.3.3 「if-else」ステートメント

この「if-else」ステートメントも、「if」と同様、状態の判断として評価される。
もし、「if」の状態が真（true）の場合、つぎの最初のステートメント（consequence）を実行する。もし、状態が偽（false）の場合には、後のステートメント（alternative）を実行する。この状態（condition）の値が、ゼロ「０」の場合は、偽（false）として評価され、それ以外の場合には、真（true）として評価される。「if-else」ステートメントは、つぎのような文法となる。

```
if (condition) consequence else alternative
```

「if」ステートメントの状態（condition）のところは、上記のように丸括弧で囲まれていなければならない。つぎの２つの複合文のステートメント（consequenceとalternative）は、以下の例のように、中括弧で囲んで表現する。

```
if (x==1)
  y = 3;
else
```

```
  y = 4;
if (x==1) {
  y = 1;
  z = 2;
}
else {
  y = 3;
  z = 5;
}
```

3.3.3.4 「while」ステートメント

この「while」ステートメントは、状態ループとして使う。

この中の状態（condition）が評価され、真(true)の場合には、ループのボディ（body）を（状態を再評価して）繰り返す。

この処理の継続は、状態が偽 (false) になるか、「break」ステートメントを実行するまで、繰り返される。つぎに、「while」ループの表現についての文法を示している。

```
while (condition) body
```

ここで「while」の中の状態ステートメントは、シングルのステートメントでなければならないが、一般にボディの中は複合ステートメントとして使われる。つぎの例は、よく使う事例として紹介しておく。

```
while(x < 10)
{
  x = x+1;
  y = y*2;
}
```

3.3.3.5 「do」ステートメント

「while」ループの変形として、「do-while」ループがある。この場合の制御構造の文法は以下のようになる。

```
do body while (condition)
```

「while」ループと「do-while」ループとの違いは、「while」ループは一度もボディ（body）を通らない場合もあるが、「do-while」ループは必ずボディ（body）を実行して最後に状態（condition）を評価して、ループを繰り返すかどうかとなっている。

```
do
{
  x = x+1;
  y = y*2;
} while(x < 10);
```

3.3.3.6 「for」ステートメント

その他のループの種類として「for」ループがある。

このループのタイプは、自動的に初期化し、カウンタ値の増加を行う処理となる。

この制御構造は以下の文法となる。

```
for(statement1 ; condition ; statement2) body
```

「for」ループは、まずはステートメント１（statement１）を実行し、つぎの状態（condition）でボディ（body）を繰り返すかどうかを評価する。この制御構造は、つぎに示す「while」ステートメントでも置き換えることができるように思える。

```
statementl;
while(condition)
{
  body
  statement2;
}
```

この場合には、頻繁にステートメント１（statement1）は、ループのカウンタの変数を文字列に設定する。ここでの状態（condition）では、どうループを終了するかのカウンタ変数をチェックするステートメントとなり、ステートメント２（statement2）ではこのカウンタ値を増加するか削減するかとなる。
ここでは、「for」ループを使うかの事例を紹介する。

```
for (int i=0; i<8; i++)
{
  NumOut(0, LCD_LINEl-i*8, i);
}
```

3.3.3.7 「repeat」ステートメント

この「repeat」ステートメントは、特別な数だけの繰り返しループとなる。
この制御構造は、標準のC言語のループ構造体の設定とは異なる。NXCは、NQCからの継承となり、文法は以下のとおりとなる。

```
repeat (expression) body
```

ここでの繰返し数（expression）は、ボディ（body）を何回繰り返すかの数値を返す評価ステートメントとなる。注意として、つぎの例での「repeat」キーワードは、一度だけ繰り返し数を評価し、その数だけボディ（body）を繰り返す。

この制御構造と「while」や「do-while」のループとの違いは、ループ数を固定して繰りかえすことにある。つぎに、「repeat」ループの使用例を紹介する。

```
int i=0;
repeat (8)
{
  NumOut(0, LCD_LINEl-i*8, i++);
}
```

3.3.3.8 「switch」ステートメント

この「switch」ステートメントは、**表記**（expression）の値に応じて、いくつかの異なるコード・セクションのうちのひとつを実行することになる。ケース「case」ラベルを使い、１つかそれ以上のコード・セクションを用意する。互いのケース（case）ラベルは、定数か、スイッチとしてユニーク（唯一）のものとなる。この「switch」ステートメントは表記（expression）を評価し、ケース「case」ラベルに一致するものを探しだす。一致したケースラベルの中を、「break」ステートメントが現れるまで中のステートメントを実行する。また、一つだけ現れる省略値「default」ラベルは、どのケースに当てはまらなかった場合に実行される。「switch」ステートメントの文法的な表現はつぎのようになる。

```
switch (expression) body
```

追加情報として、ケース（case）や省略値（default）のラベルや「break」ステートメントについて紹介しておく。

- 「case」ラベル
- 「default」ラベル
- 「break」ステートメント

基本的な「switch」ステートメントを以下のようになる。

```
switch(x)
{
  case 1:
    // do something when x is 1
    break;
  case 2:
  case 3:
    // do something else when x is 2 or 3
    break;
  default:
    // do this when x is not 1, 2, or 3
  break;
}
```

NXCでは、「switch」の表記（expression）やケースラベル値として、文字列も使えるようになっている。

3.3.3.8.1 「case」ラベル：

この「switch」ステートメント内の「case」ラベルは、それ自身はステートメントとはならない。またステートメントリストの前に表記される。複数のケース「case」ラベルになる場合にも、ステートメントの前に置かれる。この「case」ラベルの文法的な表記はつぎの通りとなる。

```
case constant_expression :
```

「switch」ステートメント項目では、どう「case」ラベルを使うかの事例を紹介している。

3.3.3.8.2 「default」ラベル

「switch」ステートメント内にあるこの「default」ラベル自身は、ステートメントとはならない。ステートメントリストの前にラベルが置かれる。「switch」ステートメント内では、一つだけの「default」ラベルを持つことになる。この「default」ラベルは、つぎのような文法で使われる。

```
default :
```

「switch」ステートメント項目で、「default」ラベルの事例を紹介している。

3.3.3.9 「goto」ステートメント

この「goto」ステートメントは、プログラム内で特別な位置にジャンプする機能を持つ。プログラム内のステートメント群には、識別子とコロンとで記載されたラベルが記入できる。「goto」ステートメントは、このラベルにジャンプすることになる。この分岐するためのラベルは、関数内やタスク内でひとつのみで、他の関数やタスクへのジャンプは行わない。ここでは、増加を続ける変数（ｘ）を使った無限ループの例を紹介している。

```
my_loop:
  x++;
goto my_loop;
```

この「goto」ステートメントは、めったに使わないか、注意して使う必要がある。いつもの場合には、この制御構造は、「if」や「while」、「switch」などを使ってプログラムを作成するようにし、「goto」ステートメントを使うよりも、読みやすく、また保守しやすいようにする。

3.3.3.10 「until」ステートメント

NXCもまたNQCと互換性を持たせた「until」マクロを定義している。
この構文は、「while」ループを使って定義されている。実際の「until」マクロは、つぎのようになっている。

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

言い換えれば、「until」ステートメントは、状態（condition）が真（true）になるまで繰り返すことになる。最もよく使われる例としては、ボディ（body）を持たないか、他のタスクに影響の少ないボディのときに利用される。

```
until(EVENT_OCCURS); // wait for some event to occur
```

3.3.4 「asm」文

この「asm」ステートメントは、多くがNXC APIを呼び出すときの宣言のために利用される。このステートメントの規約は、以下の通りとなる。

```
asm {
  one or more lines of NBC assembly language // NBC アセンブリ言語
}
```

ここでのステートメントは、NBC(NeXT Byte Codes：アセンブリ開発言語)コードによるボディで構成され、直接NBCのコンパイラが裏で直接起動することになる。「asm」ステートメントは、しばしば効率化するために利用され、NXTのファームウェア上で、より高速に実行される。つぎの例は、「asm」ブロックで変数値の宣言とラベルを含んだもので、基本NBCステートメントにコメントも使っている。

```
asm {
// jmp __lbl00D5
  dseg segment
  sl0000 slong
  sl0005 slong
  bGTTrue byte
  dseg ends
  mov sl0000, 0x0
  mov sl0005, sl0000
  mov sl0000, 0xl
  cmp GT, bGTTrue, sl0005, sl0000
  set bGTTrue, FALSE
  brtst EQ, __lbl00D5, bGTTrue
  __lbl00D5:
}
```

NXCキーワードとして、「asm」ステートメント関連のものを幾つか用意している。これらキーワードは、バイト（byte）やワード（word）、整数（long）、および実数（float）タイプを使って、文字列や数値を「asm」ステートメントから返すことができるようになっている。

ASM キーワード	意　　味
__RETURN__, __RETURNS__	符号付き値を返すために利用（__RETVAL__ や __STRRETVAL__の代わりに）
__RETURNU__	符号無し値を返すために利用
__RETURNF__	浮動小数点値を返すために利用
_RETVAL__	プログラム呼ぶために4バイト符号付値を書込む
__GENRETVAL__	プログラムを呼ぶために一般の値を書込む
__URETVAL__	プログラムを呼ぶために4バイト符号無し値を書込む
__STRRETVAL__	プログラムを呼ぶために文字列を書込む
__FLTRETVAL__	プログラムを呼ぶために４バイト浮動小数点を書込む
__STRBUFFER__	中間的な文字列の値をストアして使うための最初の文字列バッファ
__STRTMPBUFFER__	２番目の文字列バッファ
__TMPBYTE__	符号付き１バイト値整数に書込んで返す一時的な変数として利用
__TMPWORD__	符号付き２バイト値整数に書込んで返す一時的な変数として利用
__TMPLONG__	符号付き４バイト値整数に書込んで返す一時的な変数として利用
__TMPULONG__	符号無し４バイト値整数に書込んで返す一時的な変数として利用
__TMPFLOAT__	４バイト値少数値に書込んで返す一時的な変数として利用
__I__	ローカルカウンタ変数
__J__	２番目のローカルカウンタ変数
__IncI__	名前Ｉとなる増加用ローカルカウンタ変数
__IncJ__	名前Ｊとなる増加用ローカルカウンタ変数
__DecI__	名前Ｉとなる減少用ローカルカウンタ変数
__DecJ__	名前Ｊとなる減少用ローカルカウンタ変数
__ResetI__	ローカルカウンタ変数Ｉのゼロリセット
__ResetJ__	ローカルカウンタ変数Ｊのゼロリセット
__THREADNAME__	カウンタスレッド名
__LINE__	カウンタライン番号
__FILE__	カウンタファイル名
__VER__	プロダクトバージョン番号

表4.　ASMキーワード

この「asm」ブロックステートメントとこれら一覧表で紹介した「ASM」キーワードは、NXCのAPIを通して利用できる。ヘッダーファイルの “NXCDef.h” で、どう利用するかの例を見つけ出すことができる。(この“NXCDef.h”は、BricxCC画面上のメニューバー「Edit」の「preferences」(環境設定)から、タブメニュー「APO」で表示されるところで表示可能) NXCのコードを、可能な限りC言語ライクに保ち、またNXC asmブロックステートメントを読みやすくするために、「#include」を使って、プロプロセッサのマクロで囲み、カスタムのヘッダーファイルとして配置する。つぎの例では、asmブロックとして周りを囲んだマクロを紹介している。

```
#define SetMotorSpeed(port, cc, thresh, fast, slow) ¥
asm { ¥
  set theSpeed, fast ¥
  brcmp cc, EndIfOut__I__, SV, thresh ¥
  set theSpeed, slow ¥
  EndIfOut__I__: ¥
  OnFwd(port, theSpeed) ¥
  __IncI__ ¥
}
```

3.3.5 他の NXC 文（Other NXC Statement)

NXCでは、いくつかの他のステートメントタイプもサポートしている。この他のNXCステートメントは、以下に記載している。

- 関数呼び出しステートメント
- 「start」ステートメント
- 「stop」ステートメント
- 「priority」ステートメント
- 「break」ステートメント
- 「continue」ステートメント
- 「return」ステートメント

多くの表現は、文法外のステートメントとなる。気になる表現としては、１加算の「++」と１減算「--」の命令（演算子）がある。

```
x++;
```

つぎの空のステートメント（空のセミコロン）も規約ステートメントとなる。

3.3.5.1 関数呼び出しステートメント

関数を呼び出すときは、つぎのような書式になる。

```
name(arguments);
```

ここでの「arguments」リストは、コンマ「,」で区切られたリストとなる。引数の数と型は、関数が定義されたものと一致する必要がある。リターン値は、任意に割り当てることが

できる。

3.3.5.2 「start」ステートメント

「start」ステートメントを使ってタスクを実行させることができる。
このステートメントは、標準または拡張したNBC/NXCの両方のファームウェアで利用できる。このオペレーションの結果は、拡張ファームウェアでは固有のオペコードとなるが、標準ファームウェアの実行に伴い、特殊コンパイラで作成さるサブルーチンとなる。

```
start task_name;
```

3.3.5.3 「stop」ステートメント

「stop」ステートメントを使ってタスクを停止することができる。
この「stop」ステートメントは、拡張NBC/NXCファームウェアが実行されているときだけ、サポートされる。

```
stop task_name;
```

3.3.5.4 「priority」ステートメント

「priority」ステートメントを使って、タスクの優先を調整することができる。
この「priority」ステートメントも拡張NBC/NXCファームウェアでのみサポートされている。タスクのプライオリティ（優先度）は、単にすでに実行している他のタスクの演算子の数値となる。普段は２０個のオペレーションとなる。

```
priority task_name, new_priority;
```

3.3.5.5 「break」ステートメント

ループ（例えば「while」ループ）の中で、「break」ステートメントを使って、ループを即座に中止することができる。ループ内で中止し、ループの外へ飛び出ることができる。

```
break;
```

ただ、この「break」ステートメントは、「switch」ステートメントでは、危機的な状態に陥ることもある。継続実行中のコードの「switch」ステートメントの中で突然意図しない終わり方をすることがある。「switch」で「break」ステートメントを忘れた場合には、しばしばバグ探しが難しくなることがある。
ここでは、「break」ステートメントをどう使うかの事例を紹介している。

```
while (x<100) {
  x = get_new_x();
  if (button_pressed())
    break;
  process(x);
}
```

3.3.5.6 「continue」ステートメント

ループの中で「continue」ステートメントを使うことで、続く以下のステートメントを処理せずに、ループのトップにスキップして処理を繰り返すことができる。

```
continue;
```

つぎの例は、「continue」ステートメントを使った事例となる。

```
while (x<100) {
  ch = get_char();
  if (ch != 's')
    continue;
  process(ch);
}
```

3.3.5.7 「return」ステートメント

もし、関数のコードの終わりに達する前に、リターン値を返すか、単にリターンするとき、「return」ステートメントを使う。

「return」キーワードの後に続けて「expression」表現をオプションとして続けるか、関数の値を返す。この表現の型は、関数の宣言型と同等のものでなくてはならない。

```
return [expression];
```

3.4 表記法（Expressions）

値は、多くの表記の基本的な型（タイプ）となる。

より複雑な表記は、さまざまな演算子を使った値で成り立つ場合である。NXTでの数値定数は、整数や浮動小数点の値で表記される。型（タイプ）は、定数の値に依存する。浮動小数点の数値定数は、NXC内部では３２ビットの表記を使って処理する。数値定数は、１０進数（例えば123や3.14）や16進数（例えば0xABC）で表記される。

定数の範囲チェックは、まずは最小値から行い、期待した以上のものであれば符号なしの値を使ったりする。また、２つの特別な値として、真（true）と偽（false）がある。偽の場合は、ゼロ「０」が設定され、真の場合には「１」が設定される。同じような値を比較演算子（例えば＜）で使う場合がある。この時の関係は、偽のとき0が、そうでなければ真の1が設定される。

値は、演算子を使って処理される。NXCの演算子は、以下の表で示され、上位から下位への処理順で優先される。

演算子	解　説	関連	制限	事例
abs()	絶対値	n/a		abs(x)
sign()	符号（負：－１、０、正：１）	n/a		sign(x)
++,--	後置き１加算/1 減算	左	変数のみ	x++

++,--	前置き 1 加算/1 減算	右	変数のみ	++x
-	単項マイナス	右		-x
~	ビットごとの NOT（単項）	右		~123
!	否定	右		!x
*,/,%	積,商,余	左		x*y
+,-	加算,減算	左		x+y
<<,>>	ビット演算子	左		x<<4
<,>,<=,>=	比較演算子	左		x<y
==,!=	関係演算子	左		x==1
&	ビット毎 AND	左		x&y
^	排他的論理和	左		x^y
|	ビット演算子 OR	左		x|y
&&	論理積 AND	左		x&&y
||	論理和 OR	左		x||y
?:	条件演算子（式 1？式 2 :式 3） 式 1 が真なら式 2/偽なら式 3	右		x==1?y:z

表5. 表記演算子

必要なところでは、丸括弧を使って評価の順序を変更する。

```
x = 2 + 3 * 4; // set x to 14
y = (2 + 3) * 4; // set y to 20
```

3.4.1 条件（Conditions）

２つの表現を比較するのに条件（condition）フォームを使う。

ひとつの条件(condition)で否定演算子「！」を使って否定するか、２つの条件で論理的なANDやORの演算子を組み合わせる。最新のコンピュータ言語のようにNXC言語でも条件の「短絡（short-circuit）」と呼ばれる評価を備えている。この意味は、条件の全体値が、条件の左側の評価から論理的に決定され、右側の評価がされないことがある。

以下のこの表では、条件の異なるタイプをまとめている。

条件	意　味
expr	exprが 0 でなければ真
expr1 == expr2	expr1とexpr2が等しいなら真
expr1 != expr2	expr1とexpr2が等しくないなら真
expr1 < expr2	expr1がexpr2より小さいなら真
expr1 <= expr2	expr1がexpr2と同じか小さいなら真
expr1 > expr2	expr1がexpr2より大きいなら真
expr1 >= expr2	expr1がexpr2と同じか大きいなら真
! condition	否定演算子（条件が真の場合に偽）
cond1 && cond2	２つの条件の論理積AND（共に真の場合に真）
cond1 || cond2	２つの条件の論理和OR (いずれかが真の場合に真)

表6. 条件

以下の２つの状態定数は、上記の条件においてどこでも使う特殊な状態となる。

- 真（true）状態
- 偽（false）状態

NXC制御構造では、ここでの条件（conditions）を「if」や「while」、「until」ステートメントのようなプログラムでの制御に使うことができる。

3.4.1.1 「true」（真）状態

この「true」キーワードは、「１」の値を持つ。
常に「true」で表現する。

3.4.1.2 「false」（偽）状態

この「false」キーワードは、ゼロ「０」の値を持つ。
常に「false」で表現する。

3.5 プリプロセッサ（The Preprocessor）

NXCは、標準C言語の**プリプロセッサ**をモデルとして実現している。
C言語のプロプロセッサは、コンパイルが実行される前に、ソースコードファイルを加工する。ソースコードに出てくるこれら条件付き挿入文や実行ブロックは、他のファイルから、含まれるコードをタスク内に展開する。
このNXCプリプロセッサは、以下の標準のプリプロセッサ（#include, #define, #ifdef, #ifndef, #endif, #if, #elif, #undef, ##, #line, #error, and #pragma）のコマンドを実行する。また、標準外の２つのプリプロッサの「**#download**」と「**#import**」も備えている。標準C言語と似たように実行するので、多くのプリプロセッサの命令は、NXCでもC言語のように期待した動きをする。いくつかの重要な命令を以下に説明する。

- #include
- #define
- ## (連結)
- 条件付きコンパイル
- #import
- #download

3.5.1 #include

この「#include」コマンドは、標準C言語と同じように働き、ダブルクォートで囲まれたファイル名を組込むことになる。ここでは、パス（path）を含むことはなく、カギ括弧を使ったファイル名の指定はエラーとなる。

```
#include "foo.h" // 正常に組込む
#include <foo.h> // エラーとなる
```

NXCプログラムは、「#include "**NXCDefs.h**"」によって開始するが、ソースコードには不要となる。このファイルは、NXC APIのコアファイルで、すでに多くの重要な定数やマクロを含む標準ヘッダーファイルとして組み込まれている。NXCでは、もはやNXCDefs.hファイルを手動で読み込むようなことはしない。コンパイラでは、標準システムファイルとして無視し、自動的にこのヘッダーファイルは読み込まれる。

3.5.2 #define

この「#define」コマンドは、**マクロ**サブルーチンとして利用される。
マクロを再定義すると、コンパイラ警告が出てくる。マクロは、一般に１行で定義するように制限されていて、行の終わりがマクロの終わりとして認識される。
しかし、複数行に渡る場合の定義では、つぎの新しい行が無視されないように、継続文字を入れる必要がある。新しい行につなげるためには、先の行の終わりに、文字‘**¥**’を使って対応する。つぎの例で、どう複数行に渡ってプロプロセッサのマクロを書くかを示している。

```
#define foo(x) do { bar(x); ¥
                    baz(x); } while(false)
```

また、「#undef」指示文は、マクロの終わりの定義として利用する。

3.5.3 ## (連結)

この「##」命令文は、C言語のプリプロセッサと同じように働く。何も置き換えることはなく、互いに連結していく。はじめはこのトークン「##」を含んだ組み合わせで記述されたマクロ関数で、パラメータ値を使って連結（展開）する。

```
#define ELEMENT_OUT(n) ¥
NumOut(0, LCD_LINE##n, b##n)  // 「##n」を引数「n」に置き換えて連結（展開）
bool b1 = false;
bool b2 = true;
task main()
{
  ELEMENT_OUT(1);
  ELEMENT_OUT(2);
  Wait(SEC_2);
}
```

これは同様に以下のようにも書ける。

```
bool b1 = false;
bool b2 = true;
task main()
{
  NumOut(0, LCD_LINE1, b1);
  NumOut(0, LCD_LINE2, b2);
  Wait(SEC_2);
}
```

3.5.4 条件付きコンパイル

条件付きコンパイルは、C言語プリプロセッサの条件付きコンパイラと同じように働く。つぎの一覧表のようにプリプロセッサ命令は実行される。

条件命令	説　明
#ifdef symbol	symbolが定義されていれば、以下のコードをコンパイル
#ifndef symbol	symbolが定義されてなければ、以下のコードをコンパイル
#else	コンパイルかコンパイルしないか、またその逆も含めたスイッチ
#endif	以上のコンパイルを終了
#if condition	状態（condition）が真の場合以下のコードをコンパイル
#elif	#ifを使った場合、 #elseと同じ意味

表 7. 条件付きコンパイル命令

[参照] ヘッダーファイルの「**NXTDefs.h**」と「**NXCDefs.h**」には、条件付きコンパイルの多くの事例が入っていている。

3.5.5 #import

この「#import」命令は、NXCプログラムの中に、グローバルなバイト配列を定義し、そこにファイルからデータを読み込み設定して利用する。「#include」のように、この命令の後に、ダブルクォートで囲まれたファイル名を記述する。このファイル名に続けて、データを読込む配列名を記述するが、このデータの配列名はオプション（省略可）となる。

```
#import   "myfile.txt" data
```

この配列名を省略することで、ファイル名の拡張子以下を取った名前が配列名として作成される。例えば、上記の事例のなかのでフォーマット名の”data“を記載しなかった場合、配列名は”myfile”が自動的に作成されて、バイト型配列データが組み込まれる。この例では、”data”のフォーマット名にバイト配列として値が読み込まれる。この「＃import」命令はときどきAPI関数の「**GraphicArrayOut**」や「**GraphicArrayOutEx**」と同じように利用される。

3.5.6 #download

この「#download」命令は、コンパイラによってソースコードから補助ファイル（拡張子「.rxe」を含む）をダウンロードして組み込む処理をする。
指定したファイル拡張子が、コンパイル（例えば.rsか.nbcか）するソースコードのタイプと一致したとき、バイナリーへの変換前にこのソースコードをコンパイルする。
ダウンロードするファイル名前は、ここでの命令の後に続けて、ダブルクォートで囲んでおく。もしコンパイラがオリジナルのソースのみしかコンパイルしない場合、#download命令は無視される。

```
#download "myfile.rs"
#download "mypicture.ric"
```