Not eXactly C (NXC)
Programmer's Guide

Version 1.0.1 b25

by John Hansen

Contents

List of Tables

1 Introduction

NXC stands for Not eXactly C. It is a simple language for programming the LEGO MINDSTORMS NXT product. The NXT has a bytecode interpreter (provided by LEGO), which can be used to execute programs. The NXC compiler translates a source program into NXT bytecodes, which can then be executed on the target itself. Although the preprocessor and control structures of NXC are very similar to C, NXC is not a general-purpose programming language - there are many restrictions that stem from limitations of the NXT bytecode interpreter.

Logically, NXC is defined as two separate pieces. The NXC language describes the syntax to be used in writing programs. The NXC Application Programming Interface (API) describes the system functions, constants, and macros that can be used by programs. This API is defined in a special file known as a "header file" which should be included at the beginning of any NXC program. By default, this file is not automatically included when compiling a program.

This document describes both the NXC language and the NXC API. In short, it provides the information needed to write NXC programs. Since there are different interfaces for NXC, this document does not describe how to use any specific NXC implementation (such as the command-line compiler or Bricx Command Center). Refer to the documentation provided with the NXC tool, such as the *NXC User Manual*, for information specific to that implementation.

For up-to-date information and documentation for NXC, visit the NXC website at http://bricxcc.sourceforge.net/nxc/.

2 The NXC Language

This section describes the NXC language itself. This includes the lexical rules used by the compiler, the structure programs, statements, and expressions, and the operation of the preprocessor.

NXC is a case-sensitive language just like C and C++. That means that the identifier "xYz" is not the same identifier as "Xyz". Similarly, the "if" statement begins with the keyword "if" but "iF", "If", or "IF" are all just valid identifiers – not keywords.

2.1 Lexical Rules

The lexical rules describe how NXC breaks a source file into individual tokens. This includes the way comments are written, the handling of whitespace, and valid characters for identifiers.

2.1.1 Comments

Two forms of comments are supported in NXC. The first form (traditional C comments) begin with `/*` and end with `*/`. They may span multiple lines, but do not nest:

```
/* this is a comment */

/* this is a two
   line comment */

/* another comment...
   /* trying to nest...
      ending the inner comment...*/
   this text is no longer a comment! */
```

The second form of comments begins with `//` and ends with a newline (sometimes known as C++ style comments).

```
// a single line comment
```

The compiler ignores comments. Their only purpose is to allow the programmer to document the source code.

2.1.2 Whitespace

Whitespace (spaces, tabs, and newlines) is used to separate tokens and to make programs more readable. As long as the tokens are distinguishable, adding or subtracting whitespace has no effect on the meaning of a program. For example, the following lines of code both have the same meaning:

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

Some of the C++ operators consist of multiple characters. In order to preserve these tokens whitespace must not be inserted within them. In the example below, the first line

uses a right shift operator ('>>'), but in the second line the added space causes the '>' symbols to be interpreted as two separate tokens and thus generate an error.

```
x = 1 >> 4; // set x to 1 right shifted by 4 bits
x = 1 > > 4; // error
```

2.1.3 Numerical Constants

Numerical constants may be written in either decimal or hexadecimal form. Decimal constants consist of one or more decimal digits. Hexadecimal constants start with `0x` or `0X` followed by one or more hexadecimal digits.

```
x = 10;  // set x to 10
x = 0x10; // set x to 16 (10 hex)
```

2.1.4 Identifiers and Keywords

Identifiers are used for variable, task, function, and subroutine names. The first character of an identifier must be an upper or lower case letter or the underscore ('_'). Remaining characters may be letters, numbers, and an underscore.

A number of potential identifiers are reserved for use in the NXC language itself. These reserved words are call keywords and may not be used as identifiers. A complete list of keywords appears below:

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

Table 1. NXC Keywords

2.2 Program Structure

An NXC program is composed of code blocks and variables. There are two distinct types of code blocks: tasks and functions. Each type of code block has its own unique features, but they share a common structure.

2.2.1 Tasks

The NXT supports multi-threading, so a task in NXC directly corresponds to an NXT thread. Tasks are defined using the `task` keyword using the following syntax:

```
task name()
{
  // the task's code is placed here
}
```

The name of the task may be any legal identifier. A program must always have at least one task - named "main" - which is started whenever the program is run. The maximum number of tasks is 256.

The body of a task consists of a list of statements. Scheduling dependant tasks using the Precedes or Follows API function is the primary mechanism supported by the NXT for starting other tasks concurrently. Tasks may also be started using the start statement. Tasks cannot be stopped by another task, however. The only way to stop a task is by stopping all tasks using the Stop function or by a task stopping on its own via the ExitTo function or by task execution simply reaching the end of the task.

2.2.2 Functions

It is often helpful to group a set of statements together into a single function, which can then be called as needed. NXC supports functions with arguments and return values. Functions are defined using the following syntax:

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

The return type should be the type of data returned. In the C programming language, functions are specified with the type of data they return. Functions that do not return data are specified to return `void`.

The argument list may be empty, or may contain one or more argument definitions. An argument is defined by its *type* followed by its *name*. Commas separate multiple arguments. All values are represented as bool, char, byte, int, short, long, unsigned int, unsigned long, strings, struct types, or arrays of any type. NXC also supports passing argument types by value, by constant value, by reference, and by constant reference.

When arguments are passed by value from the calling function to the callee the compiler must allocate a temporary variable to hold the argument. There are no restrictions on the type of value that may be used. However, since the function is working with a copy of the actual argument, the caller will not see any changes it makes to the value. In the example below, the function `foo` attempts to set the value of its argument to 2. This is perfectly legal, but since `foo` is working on a copy of the original argument, the variable `y` from main task remains unchanged.

```
void foo(int x)
{
      x = 2;
}

task main()
{
      int y = 1;  // y is now equal to 1
      foo(y);     // y is still equal to 1!
}
```

The second type of argument, `const` *`arg_type`*, is also passed by value, but with the restriction that only constant values (e.g. numbers) may be used. This is rather important since there are a number of NXT functions that only work with constant arguments.

```
void foo(const int x)
{
      PlaySound(x);     // ok
      x = 1;      // error - cannot modify argument
}

task main()
{
      foo(2);     // ok
      foo(4*5);   // ok - expression is still constant
      foo(x);     // error - x is not a constant
}
```

The third type, *`arg_type`* `&`, passes arguments by reference rather than by value. This allows the callee to modify the value and have those changes visible in the caller. However, only variables may be used when calling a function using *`arg_type`* `&` arguments:

```
void foo(int &x)
{
      x = 2;
}

task main()
{
      int y = 1;  // y is equal to 1

      foo(y);     // y is now equal to 2
      foo(2);     // error - only variables allowed
}
```

The fourth type, `const` *`arg_type`* `&`, is rather unusual. It is also passed by reference, but with the restriction that the callee is not allowed to modify the value. Because of this restriction, the compiler is able to pass anything (not just variables) to functions using this type of argument. In general this is the most efficient way to pass arguments in NXC.

Functions must be invoked with the correct number (and type) of arguments. The example below shows several different legal and illegal calls to function `foo`:

```
void foo(int bar, const int baz)
{
      // do something here...
}

task main()
{
      int x;      // declare variable x

      foo(1, 2);  // ok
      foo(x, 2);  // ok
      foo(2, x);  // error - 2nd argument not constant!
      foo(2);     // error - wrong number of arguments!
}
```

NXC functions may optionally be marked as inline functions. This means that each call to a function will result in another copy of the function's code being included in the program. Unless used judiciously, inline functions can lead to excessive code size.

If a function is not marked as inline then an actual NXT subroutine is created and the call to the function in NXC code will result in a subroutine call to the NXT subroutine. The total number of non-inline functions (aka subroutines) and tasks must not exceed 256.

2.2.3 Variables

All variables in NXC are of the following types:

Type Name	Information
bool	8 bit unsigned
byte, unsigned char	8 bit unsigned
char	8 bit signed
unsigned int	16 bit unsigned
short, int	16 bit signed
unsigned long	32 bit unsigned
long	32 bit signed
mutex	Special type used for exclusive code access
string	Array of byte
struct	User-defined structure types
arrays	Arrays of any type

Table 2. Variable Types

Variables are declared using the keyword for the desired type followed by a comma-separated list of variable names and terminated by a semicolon (';'). Optionally, an initial value for each variable may be specified using an equals sign ('=') after the variable name. Several examples appear below:

```
int x;      // declare x
bool y,z;   // declare y and z
long a=1,b; // declare a and b, initialize a to 1
```

Global variables are declared at the program scope (outside of any code block). Once declared, they may be used within all tasks, functions, and subroutines. Their scope begins at declaration and ends at the end of the program.

Local variables may be declared within tasks and functions. Such variables are only accessible within the code block in which they are defined. Specifically, their scope begins with their declaration and ends at the end of their code block. In the case of local variables, a compound statement (a group of statements bracketed by '{' and '}') is considered a block:

```
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
}
```

2.2.4 Structs

NXC also support structs.

TBD.

2.2.5 Arrays

NXC also support arrays. Arrays are declared the same way as ordinary variables, but with an open and close bracket following the variable name.

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

To declare arrays with more than one dimension simply add more pairs of square brackets. The maximum number of dimensions supported in NXC is 4.

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

Global arrays with one dimension can be initialized at the point of declaration using the following syntax:

```
int X[] = {1, 2, 3, 4}, Y[]={10, 10}; // 2 arrays
```

The elements of an array are identified by their position within the array (called an index). The first element has an index of 0, the second has index 1, etc. For example:

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

Currently there are some limitations on how arrays can be used. These limitations will likely be removed in future versions of NXC.

To initialize local arrays or arrays with multiple dimensions it is necessary to use the ArrayInit function. The following example shows how to initialize a two-dimensional array using ArrayInit. It also demonstrates some of the supported array API functions and expressions.

```
#include "NXCDefs.h"
task main()
{
  int myArray[][];
  int myVector[];
  byte fooArray[][][];

  ArrayInit(myVector, 0, 10); // 10 zeros in myVector
  ArrayInit(myArray, myVector, 10); // 10 vectors myArray
  ArrayInit(fooArray, myArray, 2); // 2 myArrays in fooArray

  myVector = myArray[1]; // okay as of b25
  fooArray[1] = myArray; // okay as of b25
  myVector[4] = 34;
  myArray[1] = myVector; // okay as of b25

  int ax[], ay[];
  ArrayBuild2(ax, 5, 6);
  ArrayBuild4(ay, 2, 10, 6, 43);
  int axlen = ArrayLen(ax);
  ArraySubset(ax, ay, 1, 2); // ax = {10, 6}
  if (ax == ay) { // array comparisons supported as of b25
  }
}
```

2.3 Statements

The body of a code block (task or function) is composed of statements. Statements are terminated with a semi-colon (';').

2.3.1 Variable Declaration

Variable declaration, as described in the previous section, is one type of statement. It declares a local variable (with optional initialization) for use within the code block. The syntax for a variable declaration is:

```
int variables;
```

where variables is a comma separated list of names with optional initial value:

```
name[=expression]
```

Arrays of variables may also be declared:

```
int array[][=initializer for global one-dimension arrays];
```

2.3.2 Assignment

Once declared, variables may be assigned the value of an expression:

```
variable assign_operator expression;
```

There are nine different assignment operators. The most basic operator, '=', simply assigns the value of the expression to the variable. The other operators modify the variable's value in some other way as shown in the table below

Operator	Action
=	Set variable to expression
+=	Add expression to variable
-=	Subtract expression from variable
*=	Multiple variable by expression
/=	Divide variable by expression
%=	Set variable to remainder after dividing by expression
&=	Bitwise AND expression into variable
|=	Bitwise OR expression into variable
^=	Bitwise exclusive OR into variable
||=	Set variable to absolute value of expression
+-=	Set variable to sign (-1,+1,0) of expression
>>=	Right shift variable by expression
<<=	Left shift variable by expression

Table 3. Operators

Some examples:

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

2.3.3 Control Structures

The simplest control structure is a compound statement. This is a list of statements enclosed within curly braces ('{' and '}'):

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

Although this may not seem very significant, it plays a crucial role in building more complicated control structures. Many control structures expect a single statement as their body. By using a compound statement, the same control structure can be used to control multiple statements.

The `if` statement evaluates a condition. If the condition is true it executes one statement (the consequence). An optional second statement (the alternative) is executed if the condition is false. The two syntaxes for an `if` statement is shown below.

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

Note that the condition is enclosed in parentheses. Examples are shown below. Note how a compound statement is used in the last example to allow two statements to be executed as the consequence of the condition.

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

The `while` statement is used to construct a conditional loop. The condition is evaluated, and if true the body of the loop is executed, then the condition is tested again. This process continues until the condition becomes false (or a `break` statement is executed). The syntax for a `while` loop appears below:

```
while (condition) body
```

It is very common to use a compound statement as the body of a loop:

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

A variant of the `while` loop is the `do-while` loop. Its syntax is:

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

The difference between a `while` loop and a `do-while` loop is that the `do-while` loop always executes the body at least once, whereas the `while` loop may not execute it at all.

Another kind of loop is the `for` loop:

```
for(stmt1 ; condition ; stmt2) body
```

A `for` loop always executes stmt1, then it repeatedly checks the condition and while it remains true executes the body followed by stmt2. The `for` loop is equivalent to:

```
stmt1;
while(condition)
{
      body
      stmt2;
}
```

The `repeat` statement executes a loop a specified number of times:

```
repeat (expression) body
```

The expression determines how many times the body will be executed. Note: It is only evaluated a single time and then the body is repeated that number of times. This is different from both the `while` and `do-while` loops which evaluate their condition each time through the loop.

A `switch` statement can be used to execute one of several different blocks of code depending on the value of an expression. One or more case labels precede each block of code. Each case must be a constant and unique within the switch statement. The switch statement evaluates the expression then looks for a matching case label. It will then execute any statements following the matching case until either a break statement or the end of the switch is reached. A single `default` label may also be used - it will match any value not already appearing in a case label. Technically, a switch statement has the following syntax:

```
switch (expression) body
```

The case and default labels are not statements in themselves - they are *labels* that precede statements. Multiple labels can precede the same statement. These labels have the following syntax

```
case constant_expression :
default :
```

A typical switch statement might look like this:

```
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 also supports using string types in the switch expression and constant strings in case labels.

The `goto` statement forces a program to jump to the specified location. Statements in a program can be labeled by preceding them with an identifier and a colon. A goto statement then specifies the label that the program should jump to. For example, this is how an infinite loop that increments a variable could be implemented using `goto`:

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

The `goto` statement should be used sparingly and cautiously. In almost every case, control structures such as `if`, `while`, and `switch` make a program much more readable and maintainable than using `goto`.

NXC also defines the `until` macro which provides a convenient alternative to the `while` loop. The actual definition of `until` is:

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

In other words, `until` will continue looping until the condition becomes true. It is most often used in conjunction with an empty body statement:

```
until(SENSOR_1 == 1);   // wait for sensor to be pressed
```

2.3.4 The asm Statement

The `asm` statement is used to define many of the NXC API calls. The syntax of the statement is:

```
asm {
 one or more lines of assembly language
}
```

The statement simply emits the body of the statement as NeXT Byte Codes (NBC) code and passes it directly to the NBC compiler backend. The asm statement can often be used to optimize code so that it executes as fast as possible on the NXT firmware. The following example shows an asm block containing variable declarations, labels, and basic NBC statements as well as comments.

```
asm {
//      jmp __lbl00D5
      dseg segment
        sl0000 slong
        sl0005 slong
        bGTTrue byte
      dseg ends
      mov   sl0000, 0x0
      mov   sl0005, sl0000
      mov   sl0000, 0x1
      cmp   GT, bGTTrue, sl0005, sl0000
      set bGTTrue, FALSE
      brtst EQ, __lbl00D5, bGTTrue
  __lbl00D5:
}
```

A few NXC keywords have meaning only within an asm statement. These keywords provide a means for returning string or scalar values from asm statements and for using temporary integer variables of byte, word, and long sizes.

ASM Keyword	Meaning
__RETURN__	Used to return a value other than __RETVAL__ or __STRRETVAL__
__RETVAL__	Writing to this 4-byte value returns it to the calling program
__STRRETVAL__	Writing to this string value returns it to the calling program
__TMPBYTE__	Use this temporary variable to write and return single byte values
__TMPWORD__	Use this temporary variable to write and return 2-byte values
__TMPLONG__	Use this temporary variable to write and return 4-byte values

Table 4. ASM Keywords

The asm block statement and these special ASM keywords are used throughout the NXC API. See the NXCDefs.h header file for several examples of how they can be put to use. To keep the main NXC code as "C-like" as possible and for the sake of better readability NXC asm block statements can be wrapped in preprocessor macros and placed in custom header files which are included using #include. The following example demonstrates using macro wrappers around asm block statements.

```
#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__ \
}
```

2.3.5 Other Statements

A function call is a statement of the form:

```
name(arguments);
```

The arguments list is a comma-separated list of expressions. The number and type of arguments supplied must match the definition of the function itself.

Tasks may be started with the start statement.

```
start task_name;
```

Within loops (such as a `while` loop) the `break` statement can be used to exit the loop and the `continue` statement can be used to skip to the top of the next iteration of the loop. The break statement can also be used to exit a switch statement.

```
break;
```

```
continue;
```

It is possible to cause a function to return before it reaches the end of its code using the `return` statement with an optional return value.

```
return [value];
```

Most expressions are not legal statements. One notable exception is expressions involving the increment (`++`) or decrement (`--`) operators.

```
x++;
```

The empty statement (just a bare semicolon) is also a legal statement.

2.4 Expressions

Values are the most primitive type of expressions. More complicated expressions are formed from values using various operators. The NXC language only has two built in kinds of values: numerical constants and variables.

Numerical constants in the NXT are represented as integers. The type depends on the value of the constant. NXC internally uses 32 bit signed math for constant expression evaluation. Numeric constants can be written as either decimal (e.g. `123`) or hexadecimal (e.g. `0xABC`). Presently, there is very little range checking on constants, so using a value larger than expected may have unusual effects.

Two special values are predefined: `true` and `false`. The value of `false` is zero (0), while the value of `true` is one (1). The same values hold for relational operators (e.g. `<`): when the relation is false the value is 0, otherwise the value is 1.

Values may be combined using operators. Several of the operators may only be used in evaluating constant expressions, which means that their operands must either be constants, or expressions involving nothing but constants. The operators are listed here in order of precedence (highest to lowest).

Operator	Description	Associativity	Restriction	Example
abs() sign()	Absolute value Sign of operand	n/a n/a		abs(x) sign(x)
++, --	Post increment, Post decrement	left	variables only	x++
- ~ !	Unary minus Bitwise negation (unary) Logical negation	right right right	 constant only 	-x ~123 !x
*, /, %	Multiplication, division, modulo	left		x * y
+, -	Addition, subtraction	left		x + y
<<, >>	Left and right shift	left		x << 4
<, >, <=, >=	relational operators	left		x < y
==, !=	equal to, not equal to	left		x == 1
&	Bitwise AND	left		x & y
^	Bitwise XOR	left		x ^ y
|	Bitwise OR	left		x | y
&&	Logical AND	left		x && y
||	Logical OR	left		x || y
? :	conditional value	n/a		x==1 ? y : z

Table 5. Expressions

Where needed, parentheses may be used to change the order of evaluation:

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

2.4.1 Conditions

Comparing two expressions forms a condition. There are also two constant conditions - `true` and `false` - that always evaluate to true or false respectively. A condition may be negated with the negation operator, or two conditions combined with the AND and OR operators. The table below summarizes the different types of conditions.

Condition	Meaning
`true`	always true
`false`	always false
`expr`	true if expr is not equal to 0
`expr1 == expr2`	true if expr1 equals expr2
`expr1 != expr2`	true if expr1 is not equal to expr2
`expr1 < expr2`	true if one expr1 is less than expr2
`expr1 <= expr2`	true if expr1 is less than or equal to expr2
`expr1 > expr2`	true if expr1 is greater than expr2
`expr1 >= expr2`	true if expr1 is greater than or equal to expr2
`! condition`	logical negation of a condition - true if condition is false
`cond1 && cond2`	logical AND of two conditions (true if and only if both conditions are true)
`cond1 || cond2`	logical OR of two conditions (true if and only if at least one of the conditions are true)

Table 6. Conditions

2.5 The Preprocessor

The preprocessor implements the following directives: `#include`, `#define`, `#ifdef`, `#ifndef`, `#endif`, `#undef`, `##`, `#line`, `#pragma`. Its implementation is fairly close to a standard C preprocessor, so most things that work in a generic C preprocessor should have the expected effect in NXC. Significant deviations are listed below.

2.5.1 #include

The `#include` command works as expected, with the caveat that the filename must be enclosed in double quotes. There is no notion of a system include path, so enclosing a filename in angle brackets is forbidden.

```
#include "foo.h"  // ok
#include <foo.h> // error!
```

NXC programs usually begin with `#include "NXCDefs.h"`. This standard header file includes many important constants and macros which form the core NXC API.

2.5.2 #define

The #define command is used for simple macro substitution. Redefinition of a macro is an error. The end of the line normally terminates macros, but the newline may be escaped with the backslash ('\') to allow multi-line macros:

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

The #undef directive may be used to remove a macro's definition.

2.5.3 ## (Concatenation)

The ## directive works similar to the C preprocessor. It is replaced by nothing, which causes tokens on either side to be concatenated together. Because it acts as a separator initially, it can be used within macro functions to produce identifiers via combination with parameter values.

2.5.4 Conditional Compilation

Conditional compilation works similar to the C preprocessor. The following preprocessor directives may be used:

```
#ifdef symbol
#ifndef symbol
#else
#endif
```

3 NXC API

The NXC API defines a set of constants, functions, values, and macros that provide access to various capabilities of the NXT such as sensors, outputs, and communication.

The API consists of functions, values, and constants. A function is something that can be called as a statement. Typically it takes some action or configures some parameter. Values represent some parameter or quantity and can be used in expressions. Constants are symbolic names for values that have special meanings for the target. Often, a set of constants will be used in conjunction with a function.

3.1 Input Module

The NXT input module encompasses all sensor inputs. There are four sensors, which internally are numbered 0, 1, 2, and 3. This is potentially confusing since they are externally labeled on the NXT as sensors 1, 2, 3, and 4. To help mitigate this confusion, the sensor port names `S1`, `S2`, `S3`, and `S4` have been defined. These sensor names may be used in any function that requires a sensor port as an argument. Alternatively, the NBC port name constants `IN_1`, `IN_2`, `IN_3`, and `IN_4` may also be used when a sensor port is required.

Sensor value names `SENSOR_1`, `SENSOR_2`, `SENSOR_3`, and `SENSOR_4` have also been defined. These names may also be used whenever a program wishes to read the current value of the sensor:

```
x = SENSOR_1; // read sensor and store value in x
```

3.1.1 Types and Modes

The sensor ports on the NXT are capable of interfacing to a variety of different sensors. It is up to the program to tell the NXT what kind of sensor is attached to each port. Calling SetSensorType configures a sensor's type. There are 12 sensor types, each corresponding to a specific LEGO RCX or NXT sensor. A thirteenth type (SENSOR_TYPE_NONE) is used to indicate that no sensor has been configured.

In general, a program should configure the type to match the actual sensor. If a sensor port is configured as the wrong type, the NXT may not be able to read it accurately. Use either the Sensor Type constants or the NBC Sensor Type constants.

Sensor Type	NBC Sensor Type	Meaning
SENSOR_TYPE_NONE	IN_TYPE_NO_SENSOR	no sensor configured
SENSOR_TYPE_TOUCH	IN_TYPE_SWITCH	NXT or RCX touch sensor
SENSOR_TYPE_TEMPERATURE	IN_TYPE_TEMPERATURE	RCX temperature sensor
SENSOR_TYPE_LIGHT	IN_TYPE_REFLECTION	RCX light sensor
SENSOR_TYPE_ROTATION	IN_TYPE_ANGLE	RCX rotation sensor
SENSOR_TYPE_LIGHT_ACTIVE	IN_TYPE_LIGHT_ACTIVE	NXT light sensor with light
SENSOR_TYPE_LIGHT_INACTIVE	IN_TYPE_LIGHT_INACTIVE	NXT light sensor without light
SENSOR_TYPE_SOUND_DB	IN_TYPE_SOUND_DB	NXT sound sensor with dB scaling
SENSOR_TYPE_SOUND_DBA	IN_TYPE_SOUND_DBA	NXT sound sensor with dBA scaling
SENSOR_TYPE_CUSTOM	IN_TYPE_CUSTOM	Custom sensor (unused)

SENSOR_TYPE_LOWSPEED	IN_TYPE_LOWSPEED	I2C digital sensor
SENSOR_TYPE_LOWSPEED_9V	IN_TYPE_LOWSPEED_9V	I2C digital sensor (9V power)
SENSOR_TYPE_HIGHSPEED	IN_TYPE_HISPEED	Highspeed sensor (unused)

Table 7. Sensor Type Constants

The NXT allows a sensor to be configured in different modes. The sensor mode determines how a sensor's raw value is processed. Some modes only make sense for certain types of sensors, for example `SENSOR_MODE_ROTATION` is useful only with rotation sensors. Call SetSensorMode to set the sensor mode. The possible modes are shown below. Use either the Sensor Mode constant or the NBC Sensor Mode constant.

Sensor Mode	NBC Sensor Mode	Meaning
SENSOR_MODE_RAW	IN_MODE_RAW	raw value from 0 to 1023
SENSOR_MODE_BOOL	IN_MODE_BOOLEAN	boolean value (0 or 1)
SENSOR_MODE_EDGE	IN_MODE_TRANSITIONCNT	counts number of boolean transitions
SENSOR_MODE_PULSE	IN_MODE_PERIODCOUNTER	counts number of boolean periods
SENSOR_MODE_PERCENT	IN_MODE_PCTFULLSCALE	value from 0 to 100
SENSOR_MODE_FAHRENHEIT	IN_MODE_FAHRENHEIT	degrees F
SENSOR_MODE_CELSIUS	IN_MODE_CELSIUS	degrees C
SENSOR_MODE_ROTATION	IN_MODE_ANGLESTEP	rotation (16 ticks per revolution)

Table 8. Sensor Mode Constants

When using the NXT, it is common to set both the type and mode at the same time. The SetSensor function makes this process a little easier by providing a single function to call and a set of standard type/mode combinations.

Sensor Configuration	Type	Mode
SENSOR_TOUCH	SENSOR_TYPE_TOUCH	SENSOR_MODE_BOOL
SENSOR_LIGHT	SENSOR_TYPE_LIGHT	SENSOR_MODE_PERCENT
SENSOR_ROTATION	SENSOR_TYPE_ROTATION	SENSOR_MODE_ROTATION
SENSOR_CELSIUS	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_CELSIUS
SENSOR_FAHRENHEIT	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_FAHRENHEIT
SENSOR_PULSE	SENSOR_TYPE_TOUCH	SENSOR_MODE_PULSE
SENSOR_EDGE	SENSOR_TYPE_TOUCH	SENSOR_MODE_EDGE

Table 9. Sensor Configuration Constants

The NXT provides a boolean conversion for all sensors - not just touch sensors. This boolean conversion is normally based on preset thresholds for the raw value. A "low" value (less than 460) is a boolean value of 1. A high value (greater than 562) is a boolean value of 0. This conversion can be modified: a *slope value* between 0 and 31 may be added to a sensor's mode when calling `SetSensorMode`. If the sensor's value changes more than the slope value during a certain time (3ms), then the sensor's boolean state will change. This allows the boolean state to reflect rapid changes in the raw value. A rapid increase will result in a boolean value of 0, a rapid decrease is a boolean value of 1.

Even when a sensor is configured for some other mode (i.e. `SENSOR_MODE_PERCENT`), the boolean conversion will still be carried out.

Each sensor has six fields that are used to define its state. The field constants are described in the following table.

Sensor Field Constant	Meaning
Type	The sensor type (see Table 7).

InputMode	The sensor mode (see Table 8).
RawValue	
NormalizedValue	
ScaledValue	
InvalidData	

Table 10. Sensor Field Constants

SetSensor(const port, const configuration) Function

Set the type and mode of the given sensor to the specified configuration, which must be a special constant containing both type and mode information. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensor(S1, SENSOR_TOUCH);
```

SetSensorType(const port, const type) Function

Set a sensor's type, which must be one of the predefined sensor type constants. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorType(S1, SENSOR_TYPE_TOUCH);
```

SetSensorMode(const port, const mode) Function

Set a sensor's mode, which should be one of the predefined sensor mode constants. A slope parameter for boolean conversion, if desired, may be added to the mode. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorMode(S1, SENSOR_MODE_RAW); // raw mode
SetSensorMode(S1, SENSOR_MODE_RAW + 10); // slope 10
```

SetSensorLight(const port) Function

Configure the sensor on the specified port as a light sensor (active). The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorLight(S1);
```

SetSensorSound(const port) Function

Configure the sensor on the specified port as a sound sensor (dB scaling). The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorSound(S1);
```

SetSensorTouch(const port) Function

Configure the sensor on the specified port as a touch sensor. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorSound(S1);
```

SetSensorLowspeed(const port) Function

Configure the sensor on the specified port as an I2C digital sensor (9V powered). The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorLowspeed(S1);
```

SetInput(port, const field, value) Function

Set the specified field of the sensor on the specified port to the value provided. The port may be specified using a constant (e.g., S1, S2, S3, or S4) or a variable. The field must be a sensor field constant. Valid field constants are listed in Table 10. The value may be any valid expression.

```
SetInput(S1, Type, IN_TYPE_SOUND_DB);
```

ClearSensor(const port) Function

Clear the value of a sensor - only affects sensors that are configured to measure a cumulative quantity such as rotation or a pulse count. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
ClearSensor(S1);
```

ResetSensor(port) Function

Reset the value of a sensor. If the sensor type or mode has been modified then the sensor should be reset in order to ensure that values read from the sensor are valid. Use a port constant or a variable whose value is the desired sensor port.

```
ResetSensor(x); // x = S1
```

SetCustomSensorZeroOffset(const p, value) Function

Sets the custom sensor zero offset value of a sensor. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetCustomSensorZeroOffset(S1, 12);
```

SetCustomSensorPercentFullScale(const p, value) Function

Sets the custom sensor percent full scale value of a sensor. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetCustomSensorPercentFullScale(S1, 100);
```

SetCustomSensorActiveStatus(const p, value) Function

Sets the custom sensor active status value of a sensor. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetCustomSensorActiveStatus(S1, true);
```

SetSensorDigiPinsDirection(const p, value) Function

Sets the digital pins direction value of a sensor. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorDigiPinsDirection(S1, 1);
```

SetSensorDigiPinsStatus(const p, value) Function

Sets the digital pins status value of a sensor. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorDigiPinsStatus(S1, false);
```

SetSensorDigiPinsOutputLevel(const p, value) Function

Sets the digital pins output level value of a sensor. The port must be specified using a constant (e.g., S1, S2, S3, or S4).

```
SetSensorDigiPinsOutputLevel(S1, 100);
```

3.1.2 Sensor Information

There are a number of values that can be inspected for each sensor. For all of these values the sensor must be specified by a constant port value (e.g., S1, S2, S3, or S4) unless otherwise specified.

Sensor(n) Value

Return the processed sensor reading for a sensor on port n, where n is 0, 1, 2, or 3 (or a sensor port name constant). This is the same value that is returned by the sensor value names (e.g. SENSOR_1). A variable whose value is the desired sensor port may also be used.

```
x = Sensor(S1); // read sensor 1
```

SensorUS(n) Value

Return the processed sensor reading for an ultrasonic sensor on port n, where n is 0, 1, 2, or 3 (or a sensor port name constant). Since an ultrasonic sensor is an I2C digital sensor its value cannot be read using the standard Sensor(n) value. A variable whose value is the desired sensor port may also be used.

```
x = SensorUS(S4); // read sensor 4
```

SensorType(n) Value

Return the configured type of a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). A variable whose value is the desired sensor port may also be used.

```
x = SensorType(S1);
```

SensorMode(n) Value

Return the current sensor mode for a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). A variable whose value is the desired sensor port may also be used.

```
x = SensorMode(S1);
```

SensorRaw(n) Value

Return the raw value of a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). A variable whose value is the desired sensor port may also be used.

```
x = SensorRaw(S1);
```

SensorNormalized(n) Value

Return the normalized value of a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). A variable whose value is the desired sensor port may also be used.

```
x = SensorNormalized(S1);
```

SensorScaled(n) Value

Return the scaled value of a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). A variable whose value is the desired sensor port may also be used. This is the same as the standard Sensor(n) value.

```
x = SensorScaled(S1);
```

SensorInvalid(n) Value

Return the value of the InvalidData flag of a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). A variable whose value is the desired sensor port may also be used.

```
x = SensorInvalid(S1);
```

SensorBoolean(const n) Value

Return the boolean value of a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). Boolean conversion is either done based on preset cutoffs, or a slope parameter specified by calling `SetSensorMode`.

```
x = SensorBoolean(S1);
```

GetInput(n, const field) Value

Return the value of the specified field of a sensor on port n, which must be 0, 1, 2, or 3 (or a sensor port name constant). A variable whose value is the desired sensor port may also be used. The field must be a sensor field constant. Valid field constants are listed in Table 10.

```
x = GetInput(S1, Type);
```

CustomSensorZeroOffset(const p) Value

Return the custom sensor zero offset value of a sensor on port p, which must be 0, 1, 2, or 3 (or a sensor port name constant).

```
x = CustomSensorZeroOffset(S1);
```

CustomSensorPercentFullScale(const p) Value

Return the custom sensor percent full scale value of a sensor on port p, which must be 0, 1, 2, or 3 (or a sensor port name constant).

```
x = CustomSensorPercentFullScale(S1);
```

CustomSensorActiveStatus(const p) Value

Return the custom sensor active status value of a sensor on port p, which must be 0, 1, 2, or 3 (or a sensor port name constant).

```
x = CustomSensorActiveStatus(S1);
```

SensorDigiPinsDirection(const p) Value

Return the digital pins direction value of a sensor on port p, which must be 0, 1, 2, or 3 (or a sensor port name constant).

```
x = SensorDigiPinsDirection(S1);
```

SensorDigiPinsStatus(const p) Value

Return the digital pins status value of a sensor on port p, which must be 0, 1, 2, or 3 (or a sensor port name constant).

```
x = SensorDigiPinsStatus(S1);
```

SensorDigiPinsOutputLevel(const p) Value

Return the digital pins output level value of a sensor on port p, which must be 0, 1, 2, or 3 (or a sensor port name constant).

```
x = SensorDigiPinsOutputLevel(S1);
```

3.2 Output Module

The NXT output module encompasses all the motor outputs. Nearly all of the NXC API functions dealing with outputs take either a single output or a set of outputs as their first

argument. The output or set of outputs may be a constant or a variable containing an appropriate output port value. The constants OUT_A, OUT_B, and OUT_C are used to identify the three outputs. Unlike NQC, adding individual outputs together does not combine multiple outputs. Instead, the NXC API provides predefined combinations of outputs: OUT_AB, OUT_AC, OUT_BC, and OUT_ABC. Manually combining outputs involves creating an array and adding two or more of the three individual output constants to the array.

Power levels can range 0 (lowest) to 100 (highest). Negative power levels reverse the direction of rotation (i.e., forward at a power level of -100 actually means reverse at a power level of 100).

The outputs each have several fields that define the current state of the output port. These fields are defined in the table below.

Field Constant	Type	Access	Range	Meaning
UpdateFlags	ubyte	Read/ Write	0, 255	This field can include any combination of the flag bits described in Table 12. Use UF_UPDATE_MODE, UF_UPDATE_SPEED, UF_UPDATE_TACHO_LIMIT, and UF_UPDATE_PID_VALUES along with other fields to commit changes to the state of outputs. Set the appropriate flags after setting one or more of the output fields in order for the changes to actually go into affect.
OutputMode	ubyte	Read/ Write	0, 255	This is a bitfield that can include any of the values listed in Table 13. The OUT_MODE_MOTORON bit must be set in order for power to be applied to the motors. Add OUT_MODE_BRAKE to enable electronic braking. Braking means that the output voltage is not allowed to float between active PWM pulses. It improves the accuracy of motor output but uses more battery power. To use motor regulation include OUT_MODE_REGULATED in the OutputMode value. Use UF_UPDATE_MODE with UpdateFlags to commit changes to this field.
Power	sbyte	Read/ Write	-100, 100	Specify the power level of the output. The absolute value of Power is a percentage of the full power of the motor. The sign of Power controls the rotation direction. Positive values tell the firmware to turn the motor forward, while negative values turn the motor backward. Use UF_UPDATE_POWER with UpdateFlags to commit changes to this field.
ActualSpeed	sbyte	Read	-100, 100	Return the percent of full power the firmware is applying to the output. This may vary from the Power value when auto-regulation code in the firmware responds to a load on the output.
TachoCount	slong	Read	full range of signed long	Return the internal position counter value for the specified output. The internal count is reset automatically when a new goal is set using the TachoLimit and the UF_UPDATE_TACHO_LIMIT flag. Set the UF_UPDATE_RESET_COUNT flag in UpdateFlags to reset TachoCount and cancel any TachoLimit. The sign of TachoCount indicates the motor rotation direction.

TachoLimit	ulong	Read/ Write	full range of unsigned long	Specify the number of degrees the motor should rotate. Use UF_UPDATE_TACHO_LIMIT with the UpdateFlags field to commit changes to the TachoLimit. The value of this field is a relative distance from the current motor position at the moment when the UF_UPDATE_TACHO_LIMIT flag is processed.
RunState	ubyte	Read/ Write	0..255	Use this field to specify the running state of an output. Set the RunState to OUT_RUNSTATE_RUNNING to enable power to any output. Use OUT_RUNSTATE_RAMPUP to enable automatic ramping to a new Power level greater than the current Power level. Use OUT_RUNSTATE_RAMPDOWN to enable automatic ramping to a new Power level less than the current Power level. Both the rampup and rampdown bits must be used in conjunction with appropriate TachoLimit and Power values. In this case the firmware smoothly increases or decreases the actual power to the new Power level over the total number of degrees of rotation specified in TachoLimit.
TurnRatio	sbyte	Read/ Write	-100, 100	Use this field to specify a proportional turning ratio. This field must be used in conjunction with other field values: OutputMode must include OUT_MODE_MOTORON and OUT_MODE_REGULATED, RegMode must be set to OUT_REGMODE_SYNC, RunState must not be OUT_RUNSTATE_IDLE, and Speed must be non-zero. There are only three valid combinations of left and right motors for use with TurnRatio: OUT_AB, OUT_BC, and OUT_AC. In each of these three options the first motor listed is considered to be the left motor and the second motor is the right motor, regardless of the physical configuration of the robot. Negative TurnRatio values shift power toward the left motor while positive values shift power toward the right motor. An absolute value of 50 usually results in one motor stopping. An absolute value of 100 usually results in two motors turning in opposite directions at equal power.
RegMode	ubyte	Read/ Write	0..255	This field specifies the regulation mode to use with the specified port(s). It is ignored if the OUT_MODE_REGULATED bit is not set in the OutputMode field. Unlike the OutputMode field, RegMode is not a bitfield. Only one RegMode value can be set at a time. Valid RegMode values are listed in Table 15. Speed regulation means that the firmware tries to maintain a certain speed based on the Power setting. The firmware adjusts the PWM duty cycle if the motor is affected by a physical load. This adjustment is reflected by the value of the ActualSpeed property. When using speed regulation, do not set Power to its maximum value since the firmware cannot adjust to higher power levels in that situation. Synchronization means the firmware tries to keep two motors in synch regardless of physical loads. Use this mode to maintain a straight path for a mobile robot automatically. Also use this mode with the TurnRatio property to provide

				proportional turning. Set OUT_REGMODE_SYNC on at least two motor ports in order for synchronization to function. Setting OUT_REGMODE_SYNC on all three motor ports will result in only the first two (OUT_A and OUT_B) being synchronized.
Overload	ubyte	Read	0..1	This field will have a value of 1 (true) if the firmware speed regulation cannot overcome a physical load on the motor. In other words, the motor is turning more slowly than expected. If the motor speed can be maintained in spite of loading then this field value is zero (false). In order to use this field the motor must have a non-idle RunState, an OutputMode which includes OUT_MODE_MOTORON and OUT_MODE_REGULATED, and its RegMode must be set to OUT_REGMODE_SPEED.
RegPValue	ubyte	Read/ Write	0..255	This field specifies the proportional term used in the internal proportional-integral-derivative (PID) control algorithm. Set UF_UPDATE_PID_VALUES to commit changes to RegPValue, RegIValue, and RegDValue simultaneously.
RegIValue	ubyte	Read/ Write	0..255	This field specifies the integral term used in the internal proportional-integral-derivative (PID) control algorithm. Set UF_UPDATE_PID_VALUES to commit changes to RegPValue, RegIValue, and RegDValue simultaneously.
RegDValue	ubyte	Read/ Write	0..255	This field specifies the derivative term used in the internal proportional-integral-derivative (PID) control algorithm. Set UF_UPDATE_PID_VALUES to commit changes to RegPValue, RegIValue, and RegDValue simultaneously.
BlockTachoCount	slong	Read	full range of signed long	Return the block-relative position counter value for the specified port. Refer to the UpdateFlags description for information about how to use block-relative position counts. Set the UF_UPDATE_RESET_BLOCK_COUNT flag in UpdateFlags to request that the firmware reset the BlockTachoCount. The sign of BlockTachoCount indicates the direction of rotation. Positive values indicate forward rotation and negative values indicate reverse rotation. Forward and reverse depend on the orientation of the motor.
RotationCount	slong	Read	full range of signed long	Return the program-relative position counter value for the specified port. Refer to the UpdateFlags description for information about how to use program-relative position counts. Set the UF_UPDATE_RESET_ROTATION_COUNT flag in UpdateFlags to request that the firmware reset the RotationCount. The sign of RotationCount indicates the direction of rotation. Positive values indicate forward rotation and negative values indicate reverse rotation. Forward and reverse depend on the orientation of the motor.

Table 11. Output Field Constants

Valid UpdateFlags values are described in the following table.

UpdateFlags Constants	Meaning
UF_UPDATE_MODE	Commits changes to the OutputMode output property
UF_UPDATE_SPEED	Commits changes to the Power output property
UF_UPDATE_TACHO_LIMIT	Commits changes to the TachoLimit output property
UF_UPDATE_RESET_COUNT	Resets all rotation counters, cancels the current goal, and resets the rotation error-correction system
UF_UPDATE_PID_VALUES	Commits changes to the PID motor regulation properties
UF_UPDATE_RESET_BLOCK_COUNT	Resets the block-relative rotation counter
UF_UPDATE_RESET_ROTATION_COUNT	Resets the program-relative rotation counter

Table 12. UpdateFlag Constants

Valid OutputMode values are described in the following table.

OutputMode Constants	Value	Meaning
OUT_MODE_COAST	0x00	No power and no braking so motors rotate freely
OUT_MODE_MOTORON	0x01	Enables PWM power to the outputs given the Power setting
OUT_MODE_BRAKE	0x02	Uses electronic braking to outputs
OUT_MODE_REGULATED	0x04	Enables active power regulation using the RegMode value
OUT_MODE_REGMETHOD	0xf0	

Table 13. OutputMode Constants

Valid RunState values are described in the following table.

RunState Constants	Value	Meaning
OUT_RUNSTATE_IDLE	0x00	Disable all power to motors.
OUT_RUNSTATE_RAMPUP	0x10	Enable ramping up from a current Power to a new (higher) Power over a specified TachoLimit goal.
OUT_RUNSTATE_RUNNING	0x20	Enable power to motors at the specified Power level.
OUT_RUNSTATE_RAMPDOWN	0x40	Enable ramping down from a current Power to a new (lower) Power over a specified TachoLimit goal.

Table 14. RunState Constants

Valid RegMode values are described in the following table.

RegMode Constants	Value	Meaning
OUT_REGMODE_IDLE	0x00	No regulation
OUT_REGMODE_SPEED	0x01	Regulate a motor's speed (Power)
OUT_REGMODE_SYNC	0x02	Synchronize the rotation of two motors

Table 15. RegMode Constants

3.2.1 Convenience Calls

Since control of outputs is such a common feature of programs, a number of convenience functions are provided that make it easy to work with the outputs. It should be noted that most of these commands do not provide any new functionality above lower level calls described in the following section. They are merely convenient ways to make programs more concise.

The Ex versions of the motor functions use special reset constants. They are defined in the following table.

Reset Constants	Value
RESET_NONE	0x00
RESET_COUNT	0x08
RESET_BLOCK_COUNT	0x20
RESET_ROTATION_COUNT	0x40
RESET_BLOCKANDTACHO	0x28
RESET_ALL	0x68

Table 16. Reset Constants

Off(outputs) Function

Turn the specified outputs off (with braking). Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
Off(OUT_A); // turn off output A
```

OffEx(outputs, const reset) Function

Turn the specified outputs off (with braking). Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
OffEx(OUT_A, RESET_NONE); // turn off output A
```

Coast(outputs) Function

Turn off the specified outputs, making them coast to a stop. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
Coast(OUT_A); // coast output A
```

CoastEx(outputs, const reset) Function

Turn off the specified outputs, making them coast to a stop. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
CoastEx(OUT_A, RESET_NONE); // coast output A
```

Float(outputs) Function

Make outputs float. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. Float is an alias for Coast.

```
Float(OUT_A); // float output A
```

OnFwd(outputs, pwr) Function

Set outputs to forward direction and turn them on. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
OnFwd(OUT_A, 75);
```

OnFwdEx(outputs, pwr, const reset) Function

Set outputs to forward direction and turn them on. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
OnFwdEx(OUT_A, 75, RESET_NONE);
```

OnRev(outputs, pwr) Function

Set outputs to reverse direction and turn them on. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
OnRev(OUT_A, 75);
```

OnRevEx(outputs, pwr, const reset) Function

Set outputs to reverse direction and turn them on. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
OnRevEx(OUT_A, 75, RESET_NONE);
```

OnFwdReg(outputs, pwr, regmode) Function

Run the specified outputs forward using the specified regulation mode. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. Valid regulation modes are listed in Table 15.

```
OnFwdReg(OUT_A, 75, OUT_REGMODE_SPEED); // regulate speed
```

OnFwdRegEx(outputs, pwr, regmode, const reset) Function

Run the specified outputs forward using the specified regulation mode. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. Valid regulation modes are listed in Table 15. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
OnFwdRegEx(OUT_A, 75, OUT_REGMODE_SPEED, RESET_NONE);
```

OnRevReg(outputs, pwr, regmode) Function

Run the specified outputs in reverse using the specified regulation mode. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. Valid regulation modes are listed in Table 15.

```
OnRevReg(OUT_A, 75, OUT_REGMODE_SPEED); // regulate speed
```

OnRevRegEx(outputs, pwr, regmode, const reset) Function

Run the specified outputs in reverse using the specified regulation mode. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. Valid regulation modes are listed in Table 15. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
OnRevRegEx(OUT_A, 75, OUT_REGMODE_SPEED, RESET_NONE);
```

OnFwdSync(outputs, pwr, turnpct) Function

Run the specified outputs forward with regulated synchronization using the specified turn ratio. Outputs can be OUT_AB, OUT_AC, OUT_BC, or a variable containing one of these values.

```
OnFwdSync(OUT_AB, 75, -100); // spin right
```

OnFwdSyncEx(outputs, pwr, turnpct, const reset) Function

Run the specified outputs forward with regulated synchronization using the specified turn ratio. Outputs can be OUT_AB, OUT_AC, OUT_BC, or a variable containing one of these values. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
OnFwdSyncEx(OUT_AB, 75, 0, RESET_NONE);
```

OnRevSync(outputs, pwr, turnpct) Function

Run the specified outputs in reverse with regulated synchronization using the specified turn ratio. Outputs can be OUT_AB, OUT_AC, OUT_BC, or a variable containing one of these values.

```
OnRevSync(OUT_AB, 75, -100); // spin left
```

OnRevSyncEx(outputs, pwr, turnpct, const reset) Function

Run the specified outputs in reverse with regulated synchronization using the specified turn ratio. Outputs can be OUT_AB, OUT_AC, OUT_BC, or a variable containing one of these values. The reset parameter controls whether any of the three position counters are reset. It must be a constant. Valid reset values are listed in Table 16.

```
OnRevSyncEx(OUT_AB, 75, -100, RESET_NONE); // spin left
```

RotateMotor(outputs, pwr, angle) Function

Run the specified outputs forward for the specified number of degrees. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
RotateMotor(OUT_A, 75, 45); // forward 45 degrees
RotateMotor(OUT_A, -75, 45); // reverse 45 degrees
```

RotateMotorPID(outputs, pwr, angle, p, i, d) Function

Run the specified outputs forward for the specified number of degrees. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. Also specify the proportional, integral, and derivative factors used by the firmware's PID motor control algorithm.

```
RotateMotorPID(OUT_A, 75, 45, 20, 40, 100);
```

RotateMotorEx(outputs, pwr, angle, turnpct, sync) Function

Run the specified outputs forward for the specified number of degrees. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. If a non-zero turn percent is specified then sync must be set to true or no turning will occur.

```
RotateMotorEx(OUT_AB, 75, 360, 50, true);
```

RotateMotorExPID(outputs, pwr, angle, turnpct, sync, p, i, d) Function

Run the specified outputs forward for the specified number of degrees. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. If a non-zero turn percent is specified then sync must be set to true or no turning will occur. Also specify the proportional, integral, and derivative factors used by the firmware's PID motor control algorithm.

```
RotateMotorExPID(OUT_AB, 75, 360, 50, true, 30, 50, 90);
```

ResetTachoCount(outputs) Function

Reset the tachometer count and tachometer limit goal for the specified outputs. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
ResetTachoCount(OUT_AB);
```

ResetBlockTachoCount(outputs) Function

Reset the block-relative position counter for the specified outputs. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
ResetBlockTachoCount(OUT_AB);
```

ResetRotationCount(outputs) Function

Reset the program-relative position counter for the specified outputs. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
ResetRotationCount(OUT_AB);
```

ResetAllTachoCounts(outputs) Function

Reset all three position counters and reset the current tachometer limit goal for the specified outputs. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values.

```
ResetAllTachoCounts(OUT_AB);
```

3.2.2 Primitive Calls

SetOutput(outputs, const field1, val1, …, const fieldN, valN) Function

Set the specified field of the outputs to the value provided. Outputs can be OUT_A, OUT_B, OUT_C, OUT_AB, OUT_AC, OUT_BC, OUT_ABC, or a variable containing one of these values. The field must be a valid output field constant. This function takes a variable number of field/value pairs.

```
SetOutput(OUT_AB, TachoLimit, 720); // set tacho limit
```

The output field constants are described in Table 11.

GetOutput(output, const field) Value

Get the value of the specified field for the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values. The field must be a valid output field constant.

```
x = GetOutput(OUT_A, TachoLimit);
```

The output field constants are described in Table 11.

MotorMode(output) Value

Get the mode of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorMode(OUT_A);
```

MotorPower(output) Value

Get the power level of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

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

MotorActualSpeed(output) Value

Get the actual speed value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorActualSpeed(OUT_A);
```

MotorTachoCount(output) Value

Get the tachometer count value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

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

MotorTachoLimit(output) Value

Get the tachometer limit value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorTachoLimit(OUT_A);
```

MotorRunState(output) Value

Get the RunState value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorRunState(OUT_A);
```

MotorTurnRatio(output) Value

Get the turn ratio value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorTurnRatio(OUT_A);
```

MotorRegulation(output) Value

Get the regulation value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorRegulation(OUT_A);
```

MotorOverload(output) Value

Get the overload value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorOverload(OUT_A);
```

MotorRegPValue(output) Value

Get the proportional PID value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorRegPValue(OUT_A);
```

MotorRegIValue(output) Value

Get the integral PID value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorRegIValue(OUT_A);
```

MotorRegDValue(output) Value

Get the derivative PID value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorRegDValue(OUT_A);
```

MotorBlockTachoCount(output) Value

Get the block-relative position counter value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorBlockTachoCount(OUT_A);
```

MotorRotationCount(output) Value

Get the program-relative position counter value of the specified output. Output can be OUT_A, OUT_B, OUT_C, or a variable containing one of these values.

```
x = MotorRotationCount(OUT_A);
```

MotorPwnFreq() Value

Get the current motor pulse width modulation frequency.

```
x = MotorPwnFreq();
```

SetMotorPwnFreq(val) Function

Set the current motor pulse width modulation frequency.

```
SetMotorPwnFreq(x);
```

3.3 IO Map Addresses

The NXT firmware provides a mechanism for reading and writing input (sensor) and output (motor) field values using low-level constants known as IO Map Addresses (IOMA). Valid IOMA constants are listed in the following table.

IOMA Constant	Parameter	Meaning
InputIOType(p)	S1..S4	Input Type value
InputIOInputMode(p)	S1..S4	Input InputMode value
InputIORawValue(p)	S1..S4	Input RawValue value
InputIONormalizedValue(p)	S1..S4	Input NormalizedValue value
InputIOScaledValue(p)	S1..S4	Input ScaledValue value
InputIOInvalidData(p)	S1..S4	Input InvalidData value
OutputIOUpdateFlags(p)	OUT_A..OUT_C	Output UpdateFlags value

OutputIOOutputMode(p)	OUT_A..OUT_C	Output OutputMode value
OutputIOPower(p)	OUT_A..OUT_C	Output Power value
OutputIOActualSpeed(p)	OUT_A..OUT_C	Output ActualSpeed value
OutputIOTachoCount(p)	OUT_A..OUT_C	Output TachoCount value
OutputIOTachoLimit(p)	OUT_A..OUT_C	Output TachoLimit value
OutputIORunState(p)	OUT_A..OUT_C	Output RunState value
OutputIOTurnRatio(p)	OUT_A..OUT_C	Output TurnRatio value
OutputIORegMode(p)	OUT_A..OUT_C	Output RegMode value
OutputIOOverload(p)	OUT_A..OUT_C	Output Overload value
OutputIORegPValue(p)	OUT_A..OUT_C	Output RegPValue value
OutputIORegIValue(p)	OUT_A..OUT_C	Output RegIValue value
OutputIORegDValue(p)	OUT_A..OUT_C	Output RegDValue value
OutputIOBlockTachoCount(p)	OUT_A..OUT_C	Output BlockTachoCount value
OutputIORotationCount(p)	OUT_A..OUT_C	Output RotationCount value

Table 17. IOMA Constants

IOMA(const n) Value

Get the specified IO Map Address value. Valid IO Map Address constants are listed in Table 17.

```
x = IOMA(InputIORawValue(S3));
```

SetIOMA(const n, val) Function

Set the specified IO Map Address to the value provided. Valid IO Map Address constants are listed in Table 17. The value must be a specified via a constant, a constant expression, or a variable.

```
SetIOMA(OutputIOPower(OUT_A), x);
```

3.4 Sound Module

The NXT sound module encompasses all sound output features. The NXT provides support for playing basic tones as well as two different types of files.

Sound files (.rso) are like .wav files. They contain thousands of sound samples that digitally represent an analog waveform. With sounds files the NXT can speak or play music or make just about any sound imaginable.

Melody files are like MIDI files. They contain multiple tones with each tone being defined by a frequency and duration pair. When played on the NXT a melody file sounds like a pure sine-wave tone generator playing back a series of notes. While not as fancy as sound files, melody files are usually much smaller than sound files.

When a sound or a file is played on the NXT, execution of the program does not wait for the previous playback to complete. To play multiple tones or files sequentially it is necessary to wait for the previous tone or file playback to complete first. This can be done via the `Wait` API function or by using the sound state value within a while loop.

The NXC API defines frequency and duration constants which may be used in calls to `PlayTone` or `PlayToneEx`. Frequency constants start with `TONE_A3` (the 'A' pitch in octave 3) and go to `TONE_B7` (the 'B' pitch in octave 7). Duration constants start with `MS_1` (1 millisecond) and go up to `MIN_1` (60000 milliseconds) with several constants in between. See NBCCommon.h for the complete list.

Valid sound flags constants are listed in the following table.

Sound Flags Constants	Read/Write	Meaning
SOUND_FLAGS_IDLE	Read	Sound is idle
SOUND_FLAGS_UPDATE	Write	Make changes take effect
SOUND_FLAGS_RUNNING	Read	Processing a tone or file

Table 18. Sound Flags Constants

Valid sound state constants are listed in the following table.

Sound State Constants	Read/Write	Meaning
SOUND_STATE_IDLE	Read	Idle, ready for start sound
SOUND_STATE_FILE	Read	Processing file of sound/melody data
SOUND_STATE_TONE	Read	Processing play tone request
SOUND_STATE_STOP	Write	Stop sound immediately and close hardware

Table 19. Sound State Constants

Valid sound mode constants are listed in the following table.

Sound Mode Constants	Read/Write	Meaning
SOUND_MODE_ONCE	Read	Only play file once
SOUND_MODE_LOOP	Read	Play file until writing SOUND_STATE_STOP into State.
SOUND_MODE_TONE	Read	Play tone specified in Frequency for Duration milliseconds.

Table 20. Sound Mode Constants

Miscellaneous sound constants from NBCCommon.h are listed in the following table.

Misc. Sound Constants	Value	Meaning
FREQUENCY_MIN	220	Minimum frequency in Hz.
FREQUENCY_MAX	14080	Maximum frequency in Hz.
SAMPLERATE_MIN	2000	Minimum sample rate supported by NXT
SAMPLERATE_DEFAULT	8000	Default sample rate
SAMPLERATE_MAX	16000	Maximum sample rate supported by NXT

Table 21. Miscellaneous Sound Constants

PlayTone(frequency, duration) Function

Play a single tone of the specified frequency and duration. The frequency is in Hz. The duration is in 1000ths of a second. All parameters may be any valid expression.

```
PlayTone(440, 500);    // Play 'A' for one half second
```

PlayToneEx(frequency, duration, volume, bLoop) Function

Play a single tone of the specified frequency, duration, and volume. The frequency is in Hz. The duration is in 1000ths of a second. Volume should be a number from 0 (silent) to 4 (loudest). All parameters may be any valid expression.

```
PlayToneEx(440, 500, 2, false);
```

PlayFile(filename) Function

Play the specified sound file (.rso) or a melody file (.rmd). The filename may be any valid string expression.

```
PlayFile("startup.rso");
```

PlayFileEx(filename, volume, bLoop) Function

Play the specified sound file (.rso) or a melody file (.rmd). The filename may be any valid string expression. Volume should be a number from 0 (silent) to 4 (loudest). bLoop is a boolean value indicating whether to repeatedly play the file.

```
PlayFileEx("startup.rso", 3, true);
```

SoundFlags() Value

Return the current sound flags. Valid sound flags values are listed in Table 18.

```
x = SoundFlags();
```

SetSoundFlags(n) Function

Set the current sound flags. Valid sound flags values are listed in Table 18.

```
SetSoundFlags(SOUND_FLAGS_UPDATE);
```

SoundState() Value

Return the current sound state. Valid sound state values are listed in Table 19.

```
x = SoundState();
```

SetSoundState(n) Function

Set the current sound state. Valid sound state values are listed in Table 19.

```
SetSoundState(SOUND_STATE_STOP);
```

SoundMode() Value

Return the current sound mode. Valid sound mode values are listed in Table 20.

```
x = SoundMode();
```

SetSoundMode(n) Function

Set the current sound mode. Valid sound mode values are listed in Table 20.

```
SetSoundMode(SOUND_MODE_ONCE);
```

SoundFrequency() Value

Return the current sound frequency.

```
x = SoundFrequency();
```

SetSoundFrequency(n) Function

Set the current sound frequency.

```
SetSoundFrequency(440);
```

SoundDuration() Value

Return the current sound duration.

```
x = SoundDuration();
```

SetSoundDuration(n) Function

Set the current sound duration.

```
SetSoundDuration(500);
```

SoundSampleRate() Value

Return the current sound sample rate.

```
x = SoundSampleRate();
```

SetSoundSampleRate(n) Function

Set the current sound sample rate.

```
SetSoundSampleRate(4000);
```

SoundVolume() Value

Return the current sound volume.

```
x = SoundVolume();
```

SetSoundVolume(n) Function

Set the current sound volume.

```
SetSoundVolume(3);
```

StopSound() Function

Stop playback of the current tone or file.

```
StopSound();
```

3.5 IOCtrl Module

The NXT ioctrl module encompasses low-level communication between the two processors that control the NXT. The NXC API exposes two functions that are part of this module.

PowerDown() Function

Turn off the NXT immediately.

```
PowerDown();
```

RebootInFirmwareMode() Function

Reboot the NXT in SAMBA or firmware download mode. This function is not likely to be used in a normal NXC program.

```
RebootInFirmwareMode();
```

3.6 Display module

The NXT display module encompasses support for drawing to the NXT LCD. The NXT supports drawing points, lines, rectangles, and circles on the LCD. It supports drawing graphic icon files on the screen as well as text and numbers.

The LCD screen has its origin (0, 0) at the bottom left-hand corner of the screen with the positive Y-axis extending upward and the positive X-axis extending toward the right. The NXC API provides constants for use in the NumOut and TextOut functions which makes it possible to specify LCD line numbers between 1 and 8 with line 1 being at the top of the screen and line 8 being at the bottom of the screen. These constants (`LCD_LINE1`, `LCD_LINE2`, `LCD_LINE3`, `LCD_LINE4`, `LCD_LINE5`, `LCD_LINE6`, `LCD_LINE7`, `LCD_LINE8`) should be used as the Y coordinate in NumOut and TextOut calls. Values of Y other than these constants will be adjusted so that text and numbers are on one of 8 fixed line positions.

3.6.1 High-level functions

NumOut(x, y, clear, value) Function

Draw a numeric value on the screen at the specified x and y location. Optionally clear the screen first depending on the boolean value of "clear".

```
NumOut(0, LCD_LINE1, true, x);
```

TextOut(x, y, clear, msg) Function

Draw a text value on the screen at the specified x and y location. Optionally clear the screen first depending on the boolean value of "clear".

```
TextOut(0, LCD_LINE3, false, "Hello World!");
```

GraphicOut(x, y, filename, clear) Function

Draw the specified graphic icon file on the screen at the specified x and y location. Optionally clear the screen first depending on the boolean value of "clear". If the file cannot be found then nothing will be drawn and no errors will be reported.

```
GraphicOut(40, 40, "image.ric", false);
```

CircleOut(x, y, radius, clear) Function

Draw a circle on the screen with its center at the specified x and y location, using the specified radius. Optionally clear the screen first depending on the boolean value of "clear".

```
CircleOut(40, 40, 10, false);
```

LineOut(x1, y1, x2, y2, clear) Function

Draw a line on the screen from x1, y1 to x2, y2. Optionally clear the screen first depending on the boolean value of "clear".

```
LineOut(40, 40, 10, 10, false);
```

PointOut(x, y, clear) Function

Draw a point on the screen at x, y. Optionally clear the screen first depending on the boolean value of "clear".

```
PointOut(40, 40, false);
```

RectOut(x, y, width, height, clear) Function

Draw a rectangle on the screen at x, y with the specified width and height. Optionally clear the screen first depending on the boolean value of "clear".

```
RectOut(40, 40, 30, 10, false);
```

ResetScreen() Function

Restore the standard NXT running program screen.

```
ResetScreen();
```

ClearScreen() Function

Clear the NXT LCD to a blank screen.

```
ClearScreen();
```

3.6.2 Low-level functions

Valid display flag values are listed in the following table.

Display Flags Constant	Read/Write	Meaning
DISPLAY_ON	Write	Display is on
DISPLAY_REFRESH	Write	Enable refresh
DISPLAY_POPUP	Write	Use popup display memory
DISPLAY_REFRESH_DISABLED	Read	Refresh is disabled
DISPLAY_BUSY	Read	Refresh is in progress

Table 22. Display Flags Constants

DisplayFlags() Value

Return the current display flags. Valid flag values are listed in Table 22.

```
x = DisplayFlags();
```

SetDisplayFlags(n) Function

Set the current display flags. Valid flag values are listed in Table 22.

```
SetDisplayFlags(x);
```

DisplayEraseMask() Value

Return the current display erase mask.

```
x = DisplayEraseMask();
```

SetDisplayEraseMask(n) Function

Set the current display erase mask.

```
SetDisplayEraseMask(x);
```

DisplayUpdateMask() Value

Return the current display update mask.

```
x = DisplayUpdateMask();
```

SetDisplayUpdateMask(n) Function

Set the current display update mask.

```
SetDisplayUpdateMask(x);
```

DisplayDisplay() Value

Return the current display memory address.

```
x = DisplayDisplay();
```

SetDisplayDisplay(n) Function

Set the current display memory address.

```
SetDisplayDisplay(x);
```

DisplayTextLinesCenterFlags() Value

Return the current display text lines center flags.

```
x = DisplayTextLinesCenterFlags();
```

SetDisplayTextLinesCenterFlags(n) Function

Set the current display text lines center flags.

```
SetDisplayTextLinesCenterFlags(x);
```

GetDisplayNormal(x, line, count, data) Function

Read "count" bytes from the normal display memory into the data array. Start reading from the specified x, line coordinate. Each byte of data read from screen memory is a vertical strip of 8 bits at the desired location. Each bit represents a single pixel on the LCD screen. Use TEXT_LINE1 through TEXT_LINE8 for the "line" parameter.

```
GetDisplayNormal(0, TEXTLINE_1, 8, ScreenMem);
```

SetDisplayNormal(x, line, count, data) Function

Write "count" bytes to the normal display memory from the data array. Start writing at the specified x, line coordinate. Each byte of data read from screen memory is a vertical strip of 8 bits at the desired location. Each bit represents a single pixel on the LCD screen. Use TEXT_LINE1 through TEXT_LINE8 for the "line" parameter.

```
SetDisplayNormal(0, TEXTLINE_1, 8, ScreenMem);
```

GetDisplayPopup(x, line, count, data) Function

Read "count" bytes from the popup display memory into the data array. Start reading from the specified x, line coordinate. Each byte of data read from screen memory is a vertical strip of 8 bits at the desired location. Each bit represents a single pixel on the LCD screen. Use TEXT_LINE1 through TEXT_LINE8 for the "line" parameter.

```
GetDisplayPopup(0, TEXTLINE_1, 8, PopupMem);
```

SetDisplayPopup(x, line, count, data) Function

Write "count" bytes to the popup display memory from the data array. Start writing at the specified x, line coordinate. Each byte of data read from screen memory is a vertical strip of 8 bits at the desired location. Each bit represents a single pixel on the LCD screen. Use TEXT_LINE1 through TEXT_LINE8 for the "line" parameter.

```
SetDisplayPopup(0, TEXTLINE_1, 8, PopupMem);
```

3.7 Loader Module

The NXT loader module encompasses support for the NXT file system. The NXT supports creating files, opening existing files, reading, writing, renaming, and deleting files.

Files in the NXT file system must adhere to the 15.3 naming convention for a maximum filename length of 19 characters. While multiple files can be opened simultaneously, a maximum of 4 files can be open for writing at any given time.

When accessing files on the NXT, errors can occur. The NXC API defines several constants that define possible result codes. They are listed in the following table.

Loader Result Codes	Value
LDR_SUCCESS	0x0000
LDR_INPROGRESS	0x0001
LDR_REQPIN	0x0002
LDR_NOMOREHANDLES	0x8100
LDR_NOSPACE	0x8200
LDR_NOMOREFILES	0x8300
LDR_EOFEXPECTED	0x8400
LDR_ENDOFFILE	0x8500
LDR_NOTLINEARFILE	0x8600
LDR_FILENOTFOUND	0x8700
LDR_HANDLEALREADYCLOSED	0x8800
LDR_NOLINEARSPACE	0x8900
LDR_UNDEFINEDERROR	0x8A00
LDR_FILEISBUSY	0x8B00
LDR_NOWRITEBUFFERS	0x8C00
LDR_APPENDNOTPOSSIBLE	0x8D00
LDR_FILEISFULL	0x8E00
LDR_FILEEXISTS	0x8F00
LDR_MODULENOTFOUND	0x9000
LDR_OUTOFBOUNDARY	0x9100
LDR_ILLEGALFILENAME	0x9200
LDR_ILLEGALHANDLE	0x9300
LDR_BTBUSY	0x9400
LDR_BTCONNECTFAIL	0x9500
LDR_BTTIMEOUT	0x9600
LDR_FILETX_TIMEOUT	0x9700
LDR_FILETX_DSTEXISTS	0x9800
LDR_FILETX_SRCMISSING	0x9900
LDR_FILETX_STREAMERROR	0x9A00
LDR_FILETX_CLOSEERROR	0x9B00

Table 23. Loader Result Codes

FreeMemory() Value

Get the number of bytes of flash memory that are available for use.

```
x = FreeMemory();
```

CreateFile(filename, size, out handle) Value

Create a new file with the specified filename and size and open it for writing. The file handle is returned in the last parameter, which must be a variable. The loader result code is returned as the value of the function call. The filename and size parameters must be constants, constant expressions, or variables.

```
result = CreateFile("data.txt", 1024, handle);
```

OpenFileAppend(filename, out size, out handle) Value

Open an existing file with the specified filename for writing. The file size is returned in the second parameter, which must be a variable. The file handle is returned in the last parameter, which must be a variable. The loader result code is returned as the value of the function call. The filename parameter must be a constant or a variable.

```
result = OpenFileAppend("data.txt", fsize, handle);
```

OpenFileRead(filename, out size, out handle) Value

Open an existing file with the specified filename for reading. The file size is returned in the second parameter, which must be a variable. The file handle is returned in the last parameter, which must be a variable. The loader result code is returned as the value of the function call. The filename parameter must be a constant or a variable.

```
result = OpenFileRead("data.txt", fsize, handle);
```

CloseFile(handle) Value

Close the file associated with the specified file handle. The loader result code is returned as the value of the function call. The handle parameter must be a constant or a variable.

```
result = CloseFile(handle);
```

ResolveHandle(filename, out handle, out bWriteable) Value

Resolve a file handle from the specified filename. The file handle is returned in the second parameter, which must be a variable. A boolean value indicating whether the handle can be used to write to the file or not is returned in the last parameter, which must be a variable. The loader result code is returned as the value of the function call. The filename parameter must be a constant or a variable.

```
result = ResolveHandle("data.txt", handle, bCanWrite);
```

RenameFile(oldfilename, newfilename) Value

Rename a file from the old filename to the new filename. The loader result code is returned as the value of the function call. The filename parameters must be constants or variables.

```
result = RenameFile("data.txt", "mydata.txt");
```

DeleteFile(filename) Value

Delete the specified file. The loader result code is returned as the value of the function call. The filename parameter must be a constant or a variable.

```
result = DeleteFile("data.txt");
```

Read(handle, out value) Value

Read a numeric value from the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The value parameter must be a variable. The type of the value parameter determines the number of bytes of data read.

```
result = Read(handle, value);
```

ReadLn(handle, out value) Value

Read a numeric value from the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The value parameter must be a variable. The type of the value parameter determines the number of bytes of data read. The ReadLn function reads two additional bytes from the file which it assumes are a carriage return and line feed pair.

```
result = ReadLn(handle, value);
```

ReadBytes(handle, in/out length, out buf) Value

Read the specified number of bytes from the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The length parameter must be a variable. The buf parameter must be an array or a string variable. The actual number of bytes read is returned in the length parameter.

```
result = ReadBytes(handle, len, buffer);
```

Write(handle, value) Value

Write a numeric value to the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The value parameter must be a constant, a constant expression, or a variable. The type of the value parameter determines the number of bytes of data written.

```
result = Write(handle, value);
```

WriteLn(handle, value) Value

Write a numeric value to the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The value parameter must be a constant, a constant expression, or a variable. The type of the value parameter determines the number of bytes of data

written. The WriteLn function also writes a carriage return and a line feed to the file following the numeric data.

```
result = WriteLn(handle, value);
```

WriteString(handle, str, out count) Value

Write the string to the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The count parameter must be a variable. The str parameter must be a string variable or string constant. The actual number of bytes written is returned in the count parameter.

```
result = WriteString(handle, "testing", count);
```

WriteLnString(handle, str, out count) Value

Write the string to the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The count parameter must be a variable. The str parameter must be a string variable or string constant. This function also writes a carriage return and a line feed to the file following the string data. The total number of bytes written is returned in the count parameter.

```
result = WriteLnString(handle, "testing", count);
```

WriteBytes(handle, data, out count) Value

Write the contents of the data array to the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The count parameter must be a variable. The data parameter must be an array. The actual number of bytes written is returned in the count parameter.

```
result = WriteBytes(handle, buffer, count);
```

WriteBytesEx(handle, in/out length, buf) Value

Write the specified number of bytes to the file associated with the specified handle. The loader result code is returned as the value of the function call. The handle parameter must be a variable. The length parameter must be a variable. The buf parameter must be an array or a string variable or string constant. The actual number of bytes written is returned in the length parameter.

```
result = WriteBytesEx(handle, len, buffer);
```

3.8 Comm Module

TBD

3.9 General Features

Wait(time) Function

Make a task sleep for specified amount of time (in 1000ths of a second). The time argument may be an expression or a constant:

```
Wait(1000); // wait 1 second
Wait(Random(1000)); // wait random time up to 1 second
```

Stop(bvalue) Function

Stop the running program if bvalue is true. This will halt the program completely, so any code following this command will be ignored.

```
Stop(x == 24); // stop the program if x==24
```

Random(n) Value

Return an unsigned 16-bit random number between 0 and n (exclusive). N can be a constant or a variable.

```
x = Random(10);
```

Random() Value

Return a signed 16-bit random number.

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

BatteryLevel() Value

Return the battery level in millivolts.

```
x = BatteryLevel();
```

Acquire(mutex);

Release(mutex);

Precedes(task1, task2, ..., taskn);

Follows(task1, task2, ..., taskn);

ExitTo(taskname);

val = ButtonCount(btn, reset);

val = ButtonPressed(btn, reset);

ReadButtonEx(btn, reset, pressed, count);

```
val = ButtonPressCount(b);
val = ButtonLongPressCount(b);
val = ButtonShortReleaseCount(b);
val = ButtonLongReleaseCount(b);
val = ButtonReleaseCount(b);
val = ButtonState(b);
SetButtonPressCount(b, n);
SetButtonLongPressCount(b, n);
SetButtonShortReleaseCount(b, n);
SetButtonLongReleaseCount(b, n);
SetButtonReleaseCount(b, n);
SetButtonState(b, n);


val = FirstTick();
val = CurrentTick();
ResetSleepTimer();

val = Volume();
SetVolume(n);

val = CommandFlags();
val = UIState();
val = UIButton();
val = VMRunState();
val = BatteryState();
val = SleepTimeout();
val = SleepTimer();
val = RechargeableBattery();
val = OnBrickProgramPointer();
SetCommandFlags(n);
SetUIState(n);
SetUIButton(n);
```

SetVMRunState(n);
SetBatteryState(n);
SetSleepTimeout(n);
SetSleepTimer(n);
SetOnBrickProgramPointer(n);
ForceOff(n);

val = StrToNum(str);
val = StrLen(str);
val = StrIndex(str, idx);
str = NumToStr(num);
str = StrCat(str1, str2, ..., strN);
str = SubStr(string, idx, len);
str = StrReplace(string, idx, strnew);
str = Flatten(num);

ByteArrayToStr(a, s);
StrToByteArray(s, a);
num = ArrayLen(a);
ArrayInit(a, val, cnt);
ArraySubset(aout, asrc, idx, len)
ArrayBuild1(aout, src1)
ArrayBuild2(aout, src1, src2)
ArrayBuild3(aout, src1, src2, src3)
ArrayBuild4(aout, src1, src2, src3, src4)

```
GetUSBInputBuffer(offset, cnt, data);
GetUSBOutputBuffer(offset, cnt, data);
GetUSBPollBuffer(offset, cnt, data);
val = UsbState();
SetUsbState(n);
SetUSBInputBuffer(offset, cnt, data);
SetUSBInputBufferInPtr(n);
SetUSBInputBufferOutPtr(n);
SetUSBOutputBuffer(offset, cnt, data);
SetUSBOutputBufferInPtr(n);
SetUSBOutputBufferOutPtr(n);
SetUSBPollBuffer(offset, cnt, data);
SetUSBPollBufferInPtr(n);
SetUSBPollBufferOutPtr(n);
SetUSBState(n);
val = USBInputBufferInPtr();
val = USBInputBufferOutPtr();
val = USBOutputBufferInPtr();
val = USBOutputBufferOutPtr();
val = USBPollBufferInPtr();
val = USBPollBufferOutPtr();
val = USBState();


GetHSInputBuffer(offset, cnt, data);
GetHSOutputBuffer(offset, cnt, data);
val = HSInputBufferInPtr();
val = HSInputBufferOutPtr();
val = HSOutputBufferInPtr();
val = HSOutputBufferOutPtr();
val = HSFlags();
val = HSSpeed();
val = HSState();
```

```
SetHSInputBuffer(offset, cnt, data);
SetHSInputBufferInPtr(n);
SetHSInputBufferOutPtr(n);
SetHSOutputBuffer(offset, cnt, data);
SetHSOutputBufferInPtr(n);
SetHSOutputBufferOutPtr(n);
SetHSFlags(n);
SetHSSpeed(n);
SetHSState(n);

val = SendMessage(queue, msg);
val = ReceiveMessage(queue, clear, msg);


val = LowspeedStatus(port, bready);
val = LowspeedWrite(port, retlen, buffer);
val = LowspeedRead(port, buflen, buffer);
GetLSInputBuffer(p, offset, cnt, data);
GetLSOutputBuffer(p, offset, cnt, data);
val = LSInputBufferInPtr(p);
val = LSInputBufferOutPtr(p);
val = LSInputBufferBytesToRx(p);
val = LSOutputBufferInPtr(p);
val = LSOutputBufferOutPtr(p);
val = LSOutputBufferBytesToRx(p);
val = LSMode(p);
val = LSChannelState(p);
val = LSErrorType(p);
val = LSState();
val = LSSpeed();
SetLSInputBuffer(p, offset, cnt, data);
SetLSInputBufferInPtr(p, n);
SetLSInputBufferOutPtr(p, n);
```

```
SetLSInputBufferBytesToRx(p, n);
SetLSOutputBuffer(p, offset, cnt, data);
SetLSOutputBufferInPtr(p, n);
SetLSOutputBufferOutPtr(p, n);
SetLSOutputBufferBytesToRx(p, n);
SetLSMode(p, n);
SetLSChannelState(p, n);
SetLSErrorType(p, n);
SetLSState(n);
SetLSSpeed(n);
```

```
val = BluetoothStatus(conn);
val = BluetoothWrite(conn, buffer);
val = BTDeviceCount();
SetBTDeviceCount(n);
val = BTDeviceNameCount();
SetBTDeviceNameCount(n);
GetBTInputBuffer(offset, cnt, data);
GetBTOutputBuffer(offset, cnt, data);
str = BTDeviceName(p);
str = BTConnectionName(p);
str = BTConnectionPinCode(p);
str = BrickDataName();
GetBTDeviceAddress(p, data);
GetBTConnectionAddress(p, data);
GetBrickDataAddress(data);
val = BTDeviceClass(p);
val = BTDeviceStatus(p);
val = BTConnectionClass(p);
val = BTConnectionHandleNum(p);
val = BTConnectionStreamStatus(p);
```

```
val = BTConnectionLinkQuality(p);
val = BrickDataBluecoreVersion();
val = BrickDataBtStateStatus();
val = BrickDataBtHardwareStatus();
val = BrickDataTimeoutValue();
val = BTInputBufferInPtr();
val = BTInputBufferOutPtr();
val = BTOutputBufferInPtr();
val = BTOutputBufferOutPtr();
SetBTDeviceName(p, str);
SetBTDeviceAddress(p, addr);
SetBTConnectionName(p, str);
SetBTConnectionPinCode(p, code);
SetBTConnectionAddress(p, addr);
SetBrickDataName(str);
SetBrickDataAddress(p, addr);
SetBTDeviceClass(p, n);
SetBTDeviceStatus(p, n);
SetBTConnectionClass(p, n);
SetBTConnectionHandleNum(p, n);
SetBTConnectionStreamStatus(p, n);
SetBTConnectionLinkQuality(p, n);
SetBrickDataBluecoreVersion(n);
SetBrickDataBtStateStatus(n);
SetBrickDataBtHardwareStatus(n);
SetBrickDataTimeoutValue(n);
SetBTInputBuffer(offset, cnt, data);
SetBTInputBufferInPtr(n);
SetBTInputBufferOutPtr(n);
SetBTOutputBuffer(offset, cnt, data);
SetBTOutputBufferInPtr(n);
SetBTOutputBufferOutPtr(n);
SetBluetoothState(n);
```

val = BluetoothState();

result = SendRemoteBool(conn, queue, bval)

result = SendRemoteNumber(conn, queue, val)

result = SendRemoteString(conn, queue, str)

result = SendResponseBool(queue, bval)

result = SendResponseNumber(queue, val)

result = SendResponseString(queue, str)

result = ReceiveRemoteBool(queue, clear, bval)

result = ReceiveRemoteNumber(queue, clear, val)

result = ReceiveRemoteString(queue, clear, str)

result = ReceiveRemoteMessageEx(queue, clear, str, val, bval)

result = RemoteMessageRead(conn, queue)

result = RemoteMessageWrite(conn, queue, msg)

result = RemoteStartProgram(conn, filename)

result = RemoteStopProgram(conn)

result = RemotePlaySoundFile(conn, filename, bloop)

result = RemotePlayTone(conn, frequency, duration)

result = RemoteStopSound(conn)

result = RemoteKeepAlive(conn)

result = RemoteResetScaledValue(conn, port)

result = RemoteResetMotorPosition(conn, port, brelative)

result = RemoteSetInputMode(conn, port, type, mode)

result = RemoteSetOutputState(conn, port, speed, mode, regmode, turnpct, runstate, tacholimit)