Not eXactly C (NXC) Programmer's Guide

Version 1.0.1 b31

July 12, 2007

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

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:

```
[safecall] [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 few 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.

Another optional keyword that can be specified prior to the return type of a function is the safecall keyword. If a function is marked as safecall then the compiler will synchronize the execution of this function across multiple threads by wrapping each call to the function in Acquire and Release calls. If a second thread tries to call a safecall function while another thread is executing it the second thread will have to wait until the function returns to the first thread.

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 supports user-defined aggregate types known as structs. These are declared very much like you declare structs in a C program.

```
struct car
{
  string car_type;
  int manu_year;
};

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

```
myType fred = 23;
person myPerson;
```

After you have defined the structure type you can use the new type to declare a variable or nested within another structure type declaration. Members (or fields) within the struct are accessed using a dot notation.

```
  myPerson.age = 40;

  anotherPerson = myPerson;

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

You can assign structs of the same type but the compiler will complain if the types do not match.

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. Some of 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.

```
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[];
    ArrayBuild(ax, 5, 6);
    ArrayBuild(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
    }
  }
```

NXC also supports specifying an initial size for both global and local arrays. The compiler automatically generates the required code to correctly initialize the array to zeros. If a global array declaration includes both a size and a set of initial values the size is ignored in favor of the specified values.

```
task main()
{
  int myArray[10][10];
  int myVector[10];

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

/*
 The calls to ArrayInit are not required since
 we specified the equivalent initial sizes above.
 In fact, the myVector array is not needed unless
 we have a use for it other than to initialize myArray.
*/
}
```

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[n][=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 [expression];
```

Many 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. Unlike some compilers NXC does not support what is called "short-circuit" evaluation of conditions. If you combine conditions using logical operators all parts of the condition are evaluated before determining the condition value.

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`, `#if`, `#elif`, `#undef`, `##`, `#line`, and `#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 can begin with `#include "NXCDefs.h"` but they don't need to. This standard header file includes many important constants and macros which form the core NXC API. Current versions of NXC no longer require that you manually include the NXCDefs.h header file. Unless you specifically tell the compiler to ignore the standard system files this header file will automatically be included for you.

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
#if condition
#elif
```

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 General Features

3.1.1 Timing Functions

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
```

CurrentTick() **Value**

Return an unsigned 32-bit value which is the current system timing value (called a "tick") in milliseconds.

```
x = CurrentTick();
```

FirstTick() **Value**

Return an unsigned 32-bit value which is the system timing value (called a "tick") in milliseconds at the time that the program began running.

```
x = FirstTick();
```

SleepTimeout() **Value**

Return the number of minutes that the NXT will remain on before it automatically shuts down.

```
x = SleepTimeout();
```

SleepTimer() **Value**

Return the number of minutes left in the countdown to zero from the original SleepTimeout value. When the SleepTimer value reaches zero the NXT will shutdown.

```
x = SleepTimer();
```

ResetSleepTimer() **Function**

Reset the system sleep timer back to the SleepTimeout value. Executing this function periodically can keep the NXT from shutting down while a program is running.

```
ResetSleepTimer();
```

SetSleepTimeout(minutes) **Function**

Set the NXT sleep timeout value to the specified number of minutes.

```
SetSleepTimeout(8);
```

SetSleepTimer(minutes) **Function**

Set the system sleep timer to the specified number of minutes.

```
SetSleepTimer(3);
```

3.1.2 Program Control Functions

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
```

Acquire(mutex) **Function**

Acquire the specified mutex variable. If another task already has acquired the mutex then the current task will be suspended until the mutex is released by the other task. This function is used to ensure that the current task has exclusive access to a shared resource, such as the display or a motor. After the current task has finished using the shared resource the program should call Release to allow other tasks to acquire the mutex.

```
Acquire(motorMutex); // make sure we have exclusive access
// use the motors
Release(motorMutex);
```

Release(mutex) **Function**

Release the specified mutex variable. Use this to relinquish a mutex so that it can be acquired by another task. Release should always be called after a matching call to Acquire and as soon as possible after a shared resource is no longer needed.

```
Acquire(motorMutex); // make sure we have exclusive access
// use the motors
Release(motorMutex); // release mutex for other tasks
```

Precedes(task1, task2, ..., taskN) **Function**

Schedule the specified tasks for execution once the current task has completed executing. The tasks will all execute simultaneously unless other dependencies prevent them from doing so. Generally this function should be called once within a task – preferably at the start of the task definition.

```
Precedes(moving, drawing, playing);
```

Follows(task1, task2, ..., taskN) **Function**

Schedule this task to follow the specified tasks so that it will execute once any of the specified tasks has completed executing. Generally this function should be called once within a task – preferably at the start of the task definition. If multiple tasks declare that they follow the same task then they will all execute simultaneously unless other dependencies prevent them from doing so.

```
Follows(main);
```

ExitTo(task) **Function**

Immediately exit the current task and start executing the specified task.

```
ExitTo(nextTask);
```

3.1.3 String Functions

StrToNum(str) **Value**

Return the numeric value specified by the string passed to the function. If the content of the string is not a numeric value then this function returns zero.

```
x = StrToNum(strVal);
```

StrLen(str) **Value**

Return the length of the specified string. The length of a string does not include the null terminator at the end of the string.

```
x = StrLen(msg); // return the length of msg
```

StrIndex(str, idx) **Value**

Return the numeric value of the character in the specified string at the specified index.

```
x = StrIndex(msg, 2); // return the value of msg[2]
```

NumToStr(value) **Value**

Return the string representation of the specified numeric value.

```
msg = NumToStr(-2); // returns "-2" in a string
```

StrCat(str1, str2, ..., strN) **Value**

Return a string which is the result of concatenating all of the string arguments together.

```
msg = StrCat("test", "please"); // returns "testplease"
```

SubStr(string, idx, len) **Value**

Return a sub-string from the specified input string starting at idx and including the specified number of characters.

```
msg = SubStr("test", 1, 2); // returns "es"
```

StrReplace(string, idx, newStr) **Value**

Return a string with the part of the string replaced (starting at the specified index) with the contents of the new string value provided in the third argument.

```
msg = StrReplace("testing", 3, "xx"); // returns "tesxxng"
```

Flatten(value) **Value**

Return a string containing the byte representation of the specified value.

```
msg = Flatten(48); // returns "0" since 48 == ascii("0")
msg = Flatten(12337); // returns "10" (little-endian)
```

3.1.4 Array Functions

ByteArrayToStr(arr) **Value**

Convert the specified array to a string by appending a null terminator to the end of the array elements. The array must be a one-dimensional array of byte.

```
myStr = ByteArrayToStr(myArray);
```

ByteArrayToStrEx(arr, out str) **Function**

Convert the specified array to a string by appending a null terminator to the end of the array elements. The array must be a one-dimensional array of byte.

```
ByteArrayToStrEx(myArray, myStr);
```

StrToByteArray(str, out arr) **Function**

Convert the specified string to an array of byte by removing the null terminator at the end of the string. The output array variable must be a one-dimensional array of byte.

```
StrToByteArray(myStr, myArray);
```

ArrayLen(array) Value

Return the length of the specified array.

```
x = ArrayLen(myArray);
```

ArrayInit(array, value, count) Function

Initialize the array to contain count elements with each element equal to the value provided. To initialize a multi-dimensional array, the value should be an array of N-1 dimensions, where N is the number of dimensions in the array being initialized.

```
ArrayInit(myArray, 0, 10); // 10 elements == zero
```

ArraySubset(out aout, asrc, idx, len) Function

Copy a subset of the source array starting at the specified index and containing the specified number of elements into the destination array.

```
ArraySubset(myArray, srcArray, 2, 5); copy 5 elements
```

ArrayBuild(out aout, src1 [, src2, …, srcN]) Function

Build a new array from the specified source(s). The sources can be of any type. If a source is an array then all of its elements are added to the output array.

```
ArrayBuild(myArray, src1, src2);
```

3.1.5 Numeric Functions

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); // return a value of 0..9
```

Random() Value

Return a signed 16-bit random number.

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

Sqrt(x) Value

Return the square root of the specified value.

```
x = Sqrt(x);
```

Sin(degrees) Value

Return the sine of the specified degrees value. The result is 100 times the sine value (-100..100).

```
x = Sin(theta);
```

Cos(degrees) Value

Return the cosine of the specified degrees value. The result is 100 times the cosine value (-100..100).

```
x = Cos(y);
```

Asin(value) Value

Return the inverse sine of the specified value (-100..100). The result is degrees (-90..90).

```
deg = Asin(80);
```

Acos(value) Value

Return the inverse cosine of the specified value (-100..100). The result is degrees (0..180).

```
deg = Acos(0);
```

3.1.6 Low-level System Functions

There are several standard structures that are defined by the NXC API for use with calls to low-level system functions defined within the NXT firmware. These structures are the means for passing values into the system functions and for returning values from the system functions. In order to call a system function you will need to declare a variable of the required system function structure type, set the structure members as needed by the system function, call the function, and then read the results, if desired.

Many of these system functions are wrapped into higher level NXC API functions so that the details are hidden from view. Using these low-level API calls you can improve the speed of your programs a little.

If you install the NBC/NXC enhanced standard NXT firmware on your NXT all the screen drawing system function also supports clearing pixels in addition to setting them. To switch from setting pixels to clearing pixels just specify the `DRAW_OPT_CLEAR_PIXELS` value (0x0004) in the Options member of the structures. This value can be ORed together with the `DRAW_OPT_CLEAR_WHOLE_SCREEN` value (0x0001) if desired. Also, some of the system functions and their associated structures are only supported by the NBC/NXC enhanced standard NXT firmware. These functions are marked with **(+)** to indicate this additional requirement.

The first two structures define types are used within several other structures required by the screen drawing system functions.

```
struct LocationType {
  int X;
  int Y;
};

struct SizeType {
  int Width;
```

```
    int Height;
  };
```

SysDrawText(DrawTextType & args) Function

This function lets you draw text on the NXT LCD given the parameters you pass in via the DrawTextType structure. The structure type declaration is shown below.

```
   struct DrawTextType {
    char Result;
    LocationType Location;
    string Text;
    unsigned long Options;
  };
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
  DrawTextType dtArgs;
  dtArgs.Location.X = 0;
  dtArgs.Location.Y = LCD_LINE1;
  dtArgs.Text = "Please Work";
  dtArgs.Options = 0x01; // clear before drawing
  SysDrawText(dtArgs);
```

SysDrawPoint(DrawPointType & args) Function

This function lets you draw a pixel on the NXT LCD given the parameters you pass in via the DrawPointType structure. The structure type declaration is shown below.

```
  struct DrawPointType {
    char Result;
    LocationType Location;
    unsigned long Options;
  };
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
  DrawPointType dpArgs;
  dpArgs.Location.X = 20;
  dpArgs.Location.Y = 20;
  dpArgs.Options = 0x04; // clear this pixel
  SysDrawPoint(dpArgs);
```

SysDrawLine(DrawLineType & args) Function

This function lets you draw a line on the NXT LCD given the parameters you pass in via the DrawLineType structure. The structure type declaration is shown below.

```
  struct DrawLineType {
    char Result;
    LocationType StartLoc;
    LocationType EndLoc;
    unsigned long Options;
  };
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
DrawLineType dlArgs;
dlArgs.StartLoc.X = 20;
dlArgs.StartLoc.Y = 20;
dlArgs.EndLoc.X = 60;
dlArgs.EndLoc.Y = 60;
dlArgs.Options = 0x01; // clear before drawing
SysDrawLine(dlArgs);
```

SysDrawCircle(DrawCircleType & args) Function

This function lets you draw a circle on the NXT LCD given the parameters you pass in via the DrawCircleType structure. The structure type declaration is shown below.

```
struct DrawCircleType {
  char Result;
  LocationType Center;
  byte Size;
  unsigned long Options;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
DrawCircleType dcArgs;
dcArgs.Center.X = 20;
dcArgs.Center.Y = 20;
dcArgs.Size = 10; // radius
dcArgs.Options = 0x01; // clear before drawing
SysDrawCircle(dcArgs);
```

SysDrawRect(DrawRectType & args) Function

This function lets you draw a rectangle on the NXT LCD given the parameters you pass in via the DrawRectType structure. The structure type declaration is shown below.

```
struct DrawRectType {
  char Result;
  LocationType Location;
  SizeType Size;
  unsigned long Options;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
DrawRectType drArgs;
drArgs.Location.X = 20;
drArgs.Location.Y = 20;
drArgs.Size.Width = 20;
drArgs.Size.Height = 10;
drArgs.Options = 0x00; // do not clear before drawing
SysDrawRect(drArgs);
```

SysDrawGraphic(DrawGraphicType & args) Function

This function lets you draw a graphic image (RIC file) on the NXT LCD given the parameters you pass in via the DrawGraphicType structure. The structure type declaration is shown below.

```
struct DrawGraphicType {
  char Result;
  LocationType Location;
  string Filename;
  int Variables[];
  unsigned long Options;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
DrawGraphicType dgArgs;
dgArgs.Location.X = 20;
dgArgs.Location.Y = 20;
dgArgs.Filename = "image.ric";
ArrayInit(dgArgs.Variables, 0, 10); // 10 zeros
dgArgs.Variables[0] = 12;
dgArgs.Variables[1] = 14; // etc...
dgArgs.Options = 0x00; // do not clear before drawing
SysDrawGraphic(dgArgs);
```

SysSetScreenMode(SetScreenModeType & args) Function

This function lets you set the screen mode of the NXT LCD given the parameters you pass in via the SetScreenModeType structure. The standard NXT firmware only supports setting the ScreenMode to `SCREEN_MODE_RESTORE`, which has a value of 0x00. If you install the NBC/NXC enhanced standard NXT firmware this system function also supports setting the ScreenMode to `SCREEN_MODE_CLEAR`, which has a value of 0x01. The structure type declaration is shown below.

```
struct SetScreenModeType {
  char Result;
  unsigned long ScreenMode;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
SetScreenModeType ssmArgs;
ssmArgs.ScreenMode = 0x00; // restore default NXT screen
SysSetScreenMode(ssmArgs);
```

SysSoundPlayFile(SoundPlayFileType & args) Function

This function lets you play a sound file given the parameters you pass in via the SoundPlayFileType structure. The sound file can either be an RSO file containing PCM or compressed ADPCM samples or it can be an NXT melody (RMD) file containing frequency and duration values. The structure type declaration is shown below.

```
struct SoundPlayFileType {
  char Result;
  string Filename;
  bool Loop;
  byte SoundLevel;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
SoundPlayFileType spfArgs;
spfArgs.Filename = "hello.rso";
spfArgs.Loop = false;
spfArgs.SoundLevel = 3;
SysSoundPlayFile(spfArgs);
```

SysSoundPlayTone(SoundPlayToneType & args) Function

This function lets you play a tone given the parameters you pass in via the SoundPlayToneType structure. The structure type declaration is shown below.

```
struct SoundPlayToneType {
  char Result;
  unsigned int Frequency;
  unsigned int Duration;
  bool Loop;
  byte SoundLevel;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
SoundPlayToneType sptArgs;
sptArgs.Frequency = 440;
sptArgs.Duration = 1000; // 1 second
sptArgs.Loop = false;
sptArgs.SoundLevel = 3;
SysSoundPlayTone(sptArgs);
```

SysSoundGetState(SoundGetStateType & args) Function

This function lets you retrieve information about the sound module state via the SoundGetStateType structure. Constants for sound state are `SOUND_STATE_IDLE`, `SOUND_STATE_FILE`, `SOUND_STATE_TONE`, and `SOUND_STATE_STOP`. Constants for sound flags are `SOUND_FLAGS_IDLE`, `SOUND_FLAGS_UPDATE`, and `SOUND_FLAGS_RUNNING`. The structure type declaration is shown below.

```
struct SoundGetStateType {
  byte State;
  byte Flags;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
SoundGetStateType sgsArgs;
SysSoundGetState(sgsArgs);
if (sgsArgs.State == SOUND_STATE_IDLE) {/* do stuff */}
```

SysSoundSetState(SoundSetStateType & args) Function

This function lets you set sound module state settings via the SoundSetStateType structure. Constants for sound state are `SOUND_STATE_IDLE`, `SOUND_STATE_FILE`, `SOUND_STATE_TONE`, and `SOUND_STATE_STOP`. Constants for sound flags are `SOUND_FLAGS_IDLE`, `SOUND_FLAGS_UPDATE`, and `SOUND_FLAGS_RUNNING`. The structure type declaration is shown below.

```
struct SoundSetStateType {
  byte Result;
  byte State;
  byte Flags;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
SoundSetStateType sssArgs;
sssArgs.State = SOUND_STATE_STOP;
SysSoundSetState(sssArgs);
```

SysReadButton(ReadButtonType & args) Function

This function lets you read button state information via the ReadButtonType structure. The structure type declaration is shown below.

```
struct ReadButtonType {
  char Result;
  byte Index;
  bool Pressed;
  byte Count;
  bool Reset; // reset count after reading?
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
ReadButtonType rbArgs;
rbArgs.Index = BTNRIGHT;
SysReadButton(rbArgs);
if (rbArgs.Pressed) {/* do something */}
```

SysRandomNumber(RandomNumberType & args) Function

This function lets you obtain a random number via the RandomNumberType structure. The structure type declaration is shown below.

```
struct RandomNumberType {
  int Result;
};
```

Declare a variable of this type and then call the function, passing in your variable of this structure type.

```
RandomNumberType rnArgs;
SysRandomNumber(rnArgs);
int myRandomValue = rnArgs.Result;
```

SysGetStartTick(GetStartTickType & args) Function

This function lets you obtain the tick value at the time your program began executing via the GetStartTickType structure. The structure type declaration is shown below.

```
struct GetStartTickType {
  unsigned long Result;
};
```

Declare a variable of this type and then call the function, passing in your variable of this structure type.

```
GetStartTickType gstArgs;
SysGetStartTick(gstArgs);
unsigned long myStart = gstArgs.Result;
```

SysKeepAlive(KeepAliveType & args) Function

This function lets you reset the sleep timer via the KeepAliveType structure. The structure type declaration is shown below.

```
struct KeepAliveType {
  unsigned long Result;
};
```

Declare a variable of this type and then call the function, passing in your variable of this structure type.

```
KeepAliveType kaArgs;
SysKeepAlive(kaArgs); // reset sleep timer
```

SysFileOpenWrite(FileOpenType & args) Function

This function lets you create a file that you can write to using the values specified via the FileOpenType structure. The structure type declaration is shown below. Use the FileHandle return value for subsequent file write operations. The desired maximum file capacity in bytes is specified via the Length member.

```
struct FileOpenType {
  unsigned int Result;
```

```
    byte FileHandle;
    string Filename;
    unsigned long Length;
  };
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
  FileOpenType foArgs;
  foArgs.Filename = "myfile.txt";
  foArgs.Length = 256; // create with capacity for 256 bytes
  SysFileOpenWrite(foArgs); // create the file
  if (foArgs.Result == NO_ERR) {
    // write to the file using FileHandle
  }
```

SysFileOpenAppend(FileOpenType & args) Function

This function lets you open an existing file that you can write to using the values specified via the FileOpenType structure. The structure type declaration is shown below. Use the FileHandle return value for subsequent file write operations. The available length remaining in the file is returned via the Length member.

```
  struct FileOpenType {
    unsigned int Result;
    byte FileHandle;
    string Filename;
    unsigned long Length;
  };
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
  FileOpenType foArgs;
  foArgs.Filename = "myfile.txt";
  SysFileOpenAppend(foArgs); // open the file
  if (foArgs.Result == NO_ERR) {
    // write to the file using FileHandle
    // up to the remaining available length in Length
  }
```

SysFileOpenRead(FileOpenType & args) Function

This function lets you open an existing file for reading using the values specified via the FileOpenType structure. The structure type declaration is shown below. Use the FileHandle return value for subsequent file read operations. The number of bytes that can be read from the file is returned via the Length member.

```
  struct FileOpenType {
    unsigned int Result;
    byte FileHandle;
    string Filename;
    unsigned long Length;
  };
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
FileOpenType foArgs;
foArgs.Filename = "myfile.txt";
SysFileOpenRead(foArgs); // open the file for reading
if (foArgs.Result == NO_ERR) {
  // read data from the file using FileHandle
}
```

SysFileRead(FileReadWriteType & args) Function

This function lets you read from a file using the values specified via the FileReadWriteType structure. The structure type declaration is shown below.

```
struct FileReadWriteType {
  unsigned int Result;
  byte FileHandle;
  string Buffer;
  unsigned long Length;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
FileReadWriteType frArgs;
frArgs.FileHandle = foArgs.FileHandle;
frArgs.Length = 12; // number of bytes to read
SysFileRead(frArgs);
if (frArgs.Result == NO_ERR) {
  TextOut(0, LCD_LINE1, frArgs.Buffer);
  // show how many bytes were actually read
  NumOut(0, LCD_LINE2, frArgs.Length);
}
```

SysFileWrite(FileReadWriteType & args) Function

This function lets you write to a file using the values specified via the FileReadWriteType structure. The structure type declaration is shown below.

```
struct FileReadWriteType {
  unsigned int Result;
  byte FileHandle;
  string Buffer;
  unsigned long Length;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
FileReadWriteType fwArgs;
fwArgs.FileHandle = foArgs.FileHandle;
fwArgs.Buffer = "data to write";
SysFileWrite(fwArgs);
if (fwArgs.Result == NO_ERR) {
  // display number of bytes written
  NumOut(0, LCD_LINE1, fwArgs.Length);
}
```

SysFileClose(FileCloseType & args) Function

This function lets you close a file using the values specified via the FileCloseType structure. The structure type declaration is shown below.

```
struct FileCloseType {
  unsigned int Result;
  byte FileHandle;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
FileCloseType fcArgs;
fcArgs.FileHandle = foArgs.FileHandle;
SysFileClose(fcArgs);
```

SysFileResolveHandle(FileResolveHandleType & args) Function

This function lets you resolve the handle of a file using the values specified via the FileResolveHandleType structure. The structure type declaration is shown below.

```
struct FileResolveHandleType {
  unsigned int Result;
  byte FileHandle;
  bool WriteHandle;
  string Filename;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
FileResolveHandleType frhArgs;
frhArgs.Filename = "myfile.txt";
SysFileResolveHandle(frhArgs);
if (frhArgs.Result == LDR_SUCCESS) {
  // use the FileHandle as needed
  if (frhArgs.WriteHandle) {
    // file is open for writing
  }
  else {
    // file is open for reading
  }
}
```

SysFileRename(FileRenameType & args) Function

This function lets you rename a file using the values specified via the FileRenameType structure. The structure type declaration is shown below.

```
struct FileRenameType {
  unsigned int Result;
  string OldFilename;
  string NewFilename;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
FileRenameType frArgs;
frArgs.OldFilename = "myfile.txt";
frArgs.NewFilename = "myfile2.txt";
SysFileRename(frArgs);
if (frArgs.Result == LDR_SUCCESS) { /* do something */ }
```

SysFileDelete(FileDeleteType & args) Function

This function lets you delete a file using the values specified via the FileDeleteType structure. The structure type declaration is shown below.

```
struct FileDeleteType {
  unsigned int Result;
  string Filename;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
FileDeleteType fdArgs;
fdArgs.Filename = "myfile.txt";
SysFileDelete(fdArgs); // delete the file
```

SysCommLSWrite(CommLSWriteType & args) Function

This function lets you write to an I2C (Lowspeed) sensor using the values specified via the CommLSWriteType structure. The structure type declaration is shown below.

```
struct CommLSWriteType {
  char Result;
  byte Port;
  byte Buffer[];
  byte ReturnLen;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
CommLSWriteType args;
args.Port = S1;
args.Buffer = myBuf;
args.ReturnLen = 8;
SysCommLSWrite(args);
// check Result for error status
```

SysCommLSCheckStatus(CommLSCheckStatusType & args) Function

This function lets you check the status of an I2C (Lowspeed) sensor transaction using the values specified via the CommLSCheckStatusType structure. The structure type declaration is shown below.

```
struct CommLSCheckStatusType {
  char Result;
  byte Port;
  byte BytesReady;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
CommLSCheckStatusType args;
args.Port = S1;
SysCommLSCheckStatus(args);
// is the status (Result) IDLE?
if (args.Result == LOWSPEED_IDLE) { /* proceed */ }
```

SysCommLSRead(CommLSReadType & args) Function

This function lets you read from an I2C (Lowspeed) sensor using the values specified via the CommLSReadType structure. The structure type declaration is shown below.

```
struct CommLSReadType {
  char Result;
  byte Port;
  byte Buffer[];
  byte BufferLen;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
CommLSReadType args;
args.Port = S1;
args.Buffer = myBuf;
args.BufferLen = 8;
SysCommLSRead(args);
// check Result for error status & use Buffer contents
```

SysMessageWrite(MessageWriteType & args) Function

This function lets you write a message to a queue (aka mailbox) using the values specified via the MessageWriteType structure. The structure type declaration is shown below.

```
struct MessageWriteType {
  char Result;
  byte QueueID;
  string Message;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
MessageWriteType args;
args.QueueID = MAILBOX1; // 0
args.Message = "testing";
SysMessageWrite(args);
// check Result for error status
```

SysMessageRead(MessageReadType & args) Function

This function lets you read a message from a queue (aka mailbox) using the values specified via the MessageReadType structure. The structure type declaration is shown below.

```
struct MessageReadType {
  char Result;
  byte QueueID;
  bool Remove;
  string Message;
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
MessageReadType args;
args.QueueID = MAILBOX1; // 0
args.Remove = true;
SysMessageRead(args);
if (args.Result == NO_ERR) {
  TextOut(0, LCD_LINE1, args.Message);
}
```

SysCommBTWrite(CommBTWriteType & args) Function

This function lets you write to a Bluetooth connection using the values specified via the CommBTWriteType structure. The structure type declaration is shown below.

```
struct CommBTWriteType {
  char Result;
  byte Connection;
  byte Buffer[];
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
CommBTWriteType args;
args.Connection = 1;
args.Buffer = myData;
SysCommBTWrite(args);
```

SysCommBTCheckStatus(CommBTCheckStatusType & args) Function

This function lets you check the status of a Bluetooth connection using the values specified via the CommBTCheckStatusType structure. The structure type declaration is shown below. Possible values for Result include `ERR_INVALID_PORT`, `STAT_COMM_PENDING`, `ERR_COMM_CHAN_NOT_READY`, and `LDR_SUCCESS` (0).

```
struct CommBTCheckStatusType {
  char Result;
  byte Connection;
  byte Buffer[];
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
CommBTCheckStatusType args;
args.Connection = 1;
SysCommBTCheckStatus(args);
if (args.Result == LDR_SUCCESS) { /* do something */ }
```

SysIOMapRead(IOMapReadType & args) Function

This function lets you read data from a firmware module's IOMap using the values specified via the IOMapReadType structure. The structure type declaration is shown below.

```
struct IOMapReadType {
  char Result;
  string ModuleName;
  unsigned int Offset;
  unsigned int Count;
  byte Buffer[];
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
IOMapReadType args;
args.ModuleName = CommandModuleName;
args.Offset = CommandOffsetTick;
args.Count = 4; // this value happens to be 4 bytes long
SysIOMapRead(args);
if (args.Result == NO_ERR) { /* do something with data */ }
```

SysIOMapWrite(IOMapWriteType & args) Function

This function lets you write data to a firmware module's IOMap using the values specified via the IOMapWriteType structure. The structure type declaration is shown below.

```
struct IOMapWriteType {
  char Result;
  string ModuleName;
  unsigned int Offset;
  byte Buffer[];
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
IOMapWriteType args;
args.ModuleName = SoundModuleName;
args.Offset = SoundOffsetSampleRate;
args.Buffer = theData;
SysIOMapWrite(args);
```

SysIOMapReadByID(IOMapReadByIDType & args) Function (+)

This function lets you read data from a firmware module's IOMap using the values specified via the IOMapReadByIDType structure. The structure type declaration is shown below. This function can be as much as three times faster than using SysIOMapRead since it does not have to do a string lookup using the ModuleName.

```
struct IOMapReadByIDType {
  char Result;
  unsigned long ModuleID;
  unsigned int Offset;
  unsigned int Count;
  byte Buffer[];
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
IOMapReadByIDType args;
args.ModuleID = CommandModuleID;
args.Offset = CommandOffsetTick;
args.Count = 4; // this value happens to be 4 bytes long
SysIOMapReadByID(args);
if (args.Result == NO_ERR) { /* do something with data */ }
```

SysIOMapWriteByID(IOMapWriteByIDType & args) Function (+)

This function lets you write data to a firmware module's IOMap using the values specified via the IOMapWriteByIDType structure. The structure type declaration is shown below. This function can be as much as three times faster than using SysIOMapWrite since it does not have to do a string lookup using the ModuleName.

```
struct IOMapWriteByIDType {
  char Result;
  unsigned long ModuleID;
  unsigned int Offset;
  byte Buffer[];
};
```

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
IOMapWriteByIDType args;
args.ModuleID = SoundModuleID;
args.Offset = SoundOffsetSampleRate;
args.Buffer = theData;
SysIOMapWriteByID(args);
```

SysDisplayExecuteFunction(DisplayExecuteFunctionType & args)Function (+)

This function lets you directly execute the Display module's primary drawing function using the values specified via the DisplayExecuteFunctionType structure. The structure type declaration is shown below. The values for these fields are documented in the table below. If a field member is shown as 'x' it is ignored by the specified display command.

```
struct DisplayExecuteFunctionType {
  byte Status;
  byte Cmd;
  bool On;
  byte X1;
  byte Y1;
  byte X2;
  byte Y2;
};
```

Cmd	Meaning	Expected parameters
DISPLAY_ERASE_ALL	erase entire screen	()
DISPLAY_PIXEL	set pixel (on/off)	(true/false,X1,Y1,x,x)
DISPLAY_HORIZONTAL_LINE	draw horizontal line	(true/false,X1,Y1,X2,x)
DISPLAY_VERTICAL_LINE	draw vertical line	(true/false,X1,Y1,x,Y2)
DISPLAY_CHAR	draw char (actual font)	(true/false,X1,Y1,Char,x)
DISPLAY_ERASE_LINE	erase a single line	(x,LINE,x,x,x)
DISPLAY_FILL_REGION	fill screen region	(true/false,X1,Y1,X2,Y2)
DISPLAY_FILLED_FRAME	draw a frame (on / off)	(true/false,X1,Y1,X2,Y2)

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
DisplayExecuteFunctionType args;
args.Cmd = DISPLAY_ERASE_ALL;
SysDisplayExecuteFunction(args);
```

SysCommExecuteFunction(CommExecuteFunctionType & args)Function (+)

This function lets you directly execute the Comm module's primary function using the values specified via the CommExecuteFunctionType structure. The structure type declaration is shown below. The values for these fields are documented in the table below. If a field member is shown as 'x' it is ignored by the specified display command.

```
struct CommExecuteFunctionType {
  unsigned int Result;
  byte Cmd;
  byte Param1;
  byte Param2;
  byte Param3;
  string Name;
  unsigned int RetVal;
};
```

Cmd	Meaning	(Param1,Param2,Param3,Name)
INTF_SENDFILE	Send a file over a Bluetooth connection	(Connection,x,x,Filename)
INTF_SEARCH	Search for Bluetooth devices	(x,x,x,x)
INTF_STOPSEARCH	Stop searching for Bluetooth devices	(x,x,x,x)
INTF_CONNECT	Connect to a Bluetooth device	(DeviceIndex,Connection,x,x)
INTF_DISCONNECT	Disconnect a Bluetooth device	(Connection,x,x,x)
INTF_DISCONNECTALL	Disconnect all Bluetooth devices	(x,x,x,x)
INTF_REMOVEDEVICE	Remove device from My Contacts	(DeviceIndex,x,x,x)
INTF_VISIBILITY	Set Bluetooth visibility	(true/false,x,x,x)
INTF_SETCMDMODE	Set command mode	(x,x,x,x)
INTF_OPENSTREAM	Open a stream	(x,Connection,x,x)
INTF_SENDDATA	Send data	(Length, Connection, WaitForIt, Buffer)
INTF_FACTORYRESET	Bluetooth factory reset	(x,x,x,x)
INTF_BTON	Turn Bluetooth on	(x,x,x,x)
INTF_BTOFF	Turn Bluetooth off	(x,x,x,x)
INTF_SETBTNAME	Set Bluetooth name	(x,x,x,x)
INTF_EXTREAD	Handle external? read	(x,x,x,x)
INTF_PINREQ	Handle Blueooth PIN request	(x,x,x,x)
INTF_CONNECTREQ	Handle Bluetooth connect request	(x,x,x,x)

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
CommExecuteFunctionType args;
args.Cmd = INTF_BTOFF;
SysCommExecuteFunction(args);
```

SysLoaderExecuteFunction(LoaderExecuteFunctionType & args)Function (+)

This function lets you directly execute the Loader module's primary function using the values specified via the LoaderExecuteFunctionType structure. The structure type declaration is shown below. The values for these fields are documented in the table below. If a field member is shown as 'x' it is ignored by the specified display command.

```
struct LoaderExecuteFunctionType {
  unsigned int Result;
  byte Cmd;
  string Filename;
  byte Buffer[];
  unsigned long Length;
};
```

Cmd	Meaning	Expected Parameters
LDR_CMD_OPENREAD	Open a file for reading	(Filename, Length)
LDR_CMD_OPENWRITE	Creat a file	(Filename, Length)
LDR_CMD_READ	Read from a file	(Filename, Buffer, Length)
LDR_CMD_WRITE	Write to a file	(Filename, Buffer, Length)
LDR_CMD_CLOSE	Close a file	(Filename)
LDR_CMD_DELETE	Delete a file	(Filename)
LDR_CMD_FINDFIRST	Start iterating files	(Filename, Buffer, Length)
LDR_CMD_FINDNEXT	Continue iterating files	(Filename, Buffer, Length)
LDR_CMD_OPENWRITELINEAR	Create a linear file	(Filename, Length)
LDR_CMD_OPENREADLINEAR	Read a linear file	(Filename, Buffer, Length)
LDR_CMD_OPENAPPENDDATA	Open a file for writing	(Filename, Length)
LDR_CMD_FINDFIRSTMODULE	Start iterating modules	(Filename, Buffer)
LDR_CMD_FINDNEXTMODULE	Continue iterating modules	(Buffer)
LDR_CMD_CLOSEMODHANDLE	Close module handle	()
LDR_CMD_IOMAPREAD	Read IOMap data	(Filename, Buffer, Length)
LDR_CMD_IOMAPWRITE	Write IOMap data	(Filename, Buffer, Length)
LDR_CMD_DELETEUSERFLASH	Delete all files	()
LDR_CMD_RENAMEFILE	Rename file	(Filename, Buffer, Length)

Declare a variable of this type, set its members, and then call the function, passing in your variable of this structure type.

```
LoaderExecuteFunctionType args;
args.Cmd = 0xA0; // delete user flash
SysLoaderExecuteFunction(args);
```

SysCall(funcID, args) Function

This generic macro can be used to call any system function. No type checking is performed so you need to make sure you use the correct structure type given the selected system function ID. This is, however, the fastest possible way to call a system function in NXC. Function ID constants that can be used with this API call are: `FileOpenRead`, `FileOpenWrite`, `FileOpenAppend`, `FileRead`, `FileWrite`, `FileClose`, `FileResolveHandle`, `FileRename`, `FileDelete`, `SoundPlayFile`, `SoundPlayTone`, `SoundGetState`, `SoundSetState`, `DrawText`, `DrawPoint`, `DrawLine`, `DrawCircle`, `DrawRect`, `DrawGraphic`, `SetScreenMode`, `ReadButton`, `CommLSWrite`, `CommLSRead`, `CommLSCheckStatus`, `RandomNumber`, `GetStartTick`, `MessageWrite`, `MessageRead`, `CommBTCheckStatus`, `CommBTWrite`, `KeepAlive`, `IOMapRead`, `IOMapWrite`, `IOMapReadByID`, `IOMapWriteByID`, `DisplayExecuteFunction`, `CommExecuteFunction`, and `LoaderExecuteFunction`.

```
DrawTextType dtArgs;
dtArgs.Location.X = 0;
dtArgs.Location.Y = LCD_LINE1;
dtArgs.Text = "Please Work";
SysCall(DrawText, dtArgs);
```

3.2 Input Module

The NXT input module encompasses all sensor inputs except for digital I2C (LowSpeed) sensors.

Module Constants	Value
InputModuleName	"Input.mod"
InputModuleID	0x00030001

Table 7. Input Module Constants

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
```

3.2.1 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 8. 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 9. 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 10. 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 8).
InputMode	The sensor mode (see Table 9).
RawValue	The raw sensor value
NormalizedValue	The normalized sensor value
ScaledValue	The scaled sensor value
InvalidData	Invalidates the current sensor value

Table 11. Sensor Field Constants

SetSensor(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 may be specified using a constant (e.g., `S1`, `S2`, `S3`, or `S4`) or a variable.

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

SetSensorType(port, const type) Function

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

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

SetSensorMode(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 may be specified using a constant (e.g., `S1`, `S2`, `S3`, or `S4`) or a variable.

```
SetSensorMode(S1, SENSOR_MODE_RAW); // raw mode

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

SetSensorLight(port) — Function

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

```
SetSensorLight(S1);
```

SetSensorSound(port) — Function

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

```
SetSensorSound(S1);
```

SetSensorTouch(port) — Function

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

```
SetSensorSound(S1);
```

SetSensorLowspeed(port) — Function

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

```
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 11. 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. The port may be specified using a constant (e.g., `S1`, `S2`, `S3`, or `S4`) or a variable.

```
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);
```

3.2.2 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 11.

```
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);
```

3.2.3 IOMap Offsets

Input Module Offsets	Value	Size
InputOffsetCustomZeroOffset(p)	(((p)*20)+0)	2
InputOffsetADRaw(p)	(((p)*20)+2)	2
InputOffsetSensorRaw(p)	(((p)*20)+4)	2
InputOffsetSensorValue(p)	(((p)*20)+6)	2
InputOffsetSensorType(p)	(((p)*20)+8)	1
InputOffsetSensorMode(p)	(((p)*20)+9)	1
InputOffsetSensorBoolean(p)	(((p)*20)+10)	1
InputOffsetDigiPinsDir(p)	(((p)*20)+11)	1
InputOffsetDigiPinsIn(p)	(((p)*20)+12)	1
InputOffsetDigiPinsOut(p)	(((p)*20)+13)	1
InputOffsetCustomPctFullScale(p)	(((p)*20)+14)	1
InputOffsetCustomActiveStatus(p)	(((p)*20)+15)	1
InputOffsetInvalidData(p)	(((p)*20)+16)	1
InputOffsetSpareOne(p)	(((p)*20)+17)	1
InputOffsetSpareTwo(p)	(((p)*20)+18)	1
InputOffsetSpareThree(p)	(((p)*20)+19)	1

Table 12. Input Module IOMap Offsets

3.3 Output Module

The NXT output module encompasses all the motor outputs.

Module Constants	Value
OutputModuleName	"Output.mod"
OutputModuleID	0x00020001

Table 13. Output Module Constants

Nearly all of the NXC API functions dealing with outputs take either a single output or a set of outputs as their first argument. Depending on the function call, 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 15. 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 16. 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 18. 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 14. 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 15. 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 16. 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 17. 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 18. RegMode Constants

3.3.1 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. The Var versions of the motor functions require that the outputs argument be a variable while the non-Var versions require that the outputs argument be a constant.

Reset Constants	Value
RESET_NONE	0x00
RESET_COUNT	0x08
RESET_BLOCK_COUNT	0x20

RESET_ROTATION_COUNT	0x40
RESET_BLOCKANDTACHO	0x28
RESET_ALL	0x68

Table 19. Reset Constants

Output Port Constants	Value
OUT_A	0x00
OUT_B	0x01
OUT_C	0x02
OUT_AB	0x03
OUT_AC	0x04
OUT_BC	0x05
OUT_ABC	0x06

Table 20. Output Port Constants

Off(outputs) Function

Turn the specified outputs off (with braking). Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

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

OffEx(outputs, const reset) Function

Turn the specified outputs off (with braking). Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 19.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 19.

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

Float(outputs) Function

Make outputs float. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

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

OnFwdEx(outputs, pwr, const reset) Function

Set outputs to forward direction and turn them on. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 19.

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

OnRev(outputs, pwr) Function

Set outputs to reverse direction and turn them on. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

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

OnRevEx(outputs, pwr, const reset) Function

Set outputs to reverse direction and turn them on. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 19.

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

OnFwdReg(outputs, pwr, regmode) Function

Run the specified outputs forward using the specified regulation mode. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. Valid regulation modes are listed in Table 18.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. Valid regulation modes are listed in Table 18. 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 19.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. Valid regulation modes are listed in Table 18.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. Valid regulation modes are listed in Table 18. 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 19.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 19.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 19.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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, stop) Function

Run the specified outputs forward for the specified number of degrees. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. If a non-zero turn percent is specified then sync must be set to true or no turning will occur. Specify whether the motor(s) should brake at the end of the rotation using the stop parameter.

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

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

Run the specified outputs forward for the specified number of degrees. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. If a non-zero turn percent is specified then sync must be set to true or no turning will occur. Specify whether the motor(s) should brake at the end of the rotation using the stop parameter. Also specify the proportional, integral, and derivative factors used by the firmware's PID motor control algorithm.

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

ResetTachoCount(outputs) Function

Reset the tachometer count and tachometer limit goal for the specified outputs. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
ResetTachoCount(OUT_AB);
```

ResetBlockTachoCount(outputs) Function

Reset the block-relative position counter for the specified outputs. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
ResetBlockTachoCount(OUT_AB);
```

ResetRotationCount(outputs) Function

Reset the program-relative position counter for the specified outputs. Outputs can be a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20.

```
ResetAllTachoCounts(OUT_AB);
```

3.3.2 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 a constant or a variable containing the desired output ports. Predefined output port constants are defined in Table 20. 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 14.

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 14.

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);
```

3.3.3 IOMap Offsets

Output Module Offsets	Value	Size
OutputOffsetTachoCount(p)	(((p)*32)+0)	4
OutputOffsetBlockTachoCount(p)	(((p)*32)+4)	4
OutputOffsetRotationCount(p)	(((p)*32)+8)	4

OutputOffsetTachoLimit(p)	(((p)*32)+12)	4
OutputOffsetMotorRPM(p)	(((p)*32)+16)	2
OutputOffsetFlags(p)	(((p)*32)+18)	1
OutputOffsetMode(p)	(((p)*32)+19)	1
OutputOffsetSpeed(p)	(((p)*32)+20)	1
OutputOffsetActualSpeed(p)	(((p)*32)+21)	1
OutputOffsetRegPParameter(p)	(((p)*32)+22)	1
OutputOffsetRegIParameter(p)	(((p)*32)+23)	1
OutputOffsetRegDParameter(p)	(((p)*32)+24)	1
OutputOffsetRunState(p)	(((p)*32)+25)	1
OutputOffsetRegMode(p)	(((p)*32)+26)	1
OutputOffsetOverloaded(p)	(((p)*32)+27)	1
OutputOffsetSyncTurnParameter(p)	(((p)*32)+28)	1
OutputOffsetPwnFreq	96	1

Table 21. Output Module IOMap Offsets

3.4 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 22. IOMA Constants

IOMA(const n) Value

Get the specified IO Map Address value. Valid IO Map Address constants are listed in Table 22.

```
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 22. The value must be a specified via a constant, a constant expression, or a variable.

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

3.5 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.

Module Constants	Value
SoundModuleName	"Sound.mod"
SoundModuleID	0x00080001

Table 23. Sound Module Constants

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.

3.5.1 High-level functions

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);
```

3.5.2 Low-level functions

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 24. 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 25. 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 26. Sound Mode Constants

Miscellaneous sound constants 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 27. Miscellaneous Sound Constants

SoundFlags() Value

Return the current sound flags. Valid sound flags values are listed in Table 24.

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

SetSoundFlags(n) Function

Set the current sound flags. Valid sound flags values are listed in Table 24.

```
SetSoundFlags(SOUND_FLAGS_UPDATE);
```

SoundState() Value

Return the current sound state. Valid sound state values are listed in Table 25.

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

SetSoundState(n) Function

Set the current sound state. Valid sound state values are listed in Table 25.

```
SetSoundState(SOUND_STATE_STOP);
```

SoundMode() Value

Return the current sound mode. Valid sound mode values are listed in Table 26.

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

SetSoundMode(n) Function

Set the current sound mode. Valid sound mode values are listed in Table 26.

```
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();
```

3.5.3 IOMap Offsets

Sound Module Offsets	Value	Size
SoundOffsetFreq	0	2
SoundOffsetDuration	2	2
SoundOffsetSampleRate	4	2
SoundOffsetSoundFilename	6	20
SoundOffsetFlags	26	1
SoundOffsetState	27	1
SoundOffsetMode	28	1
SoundOffsetVolume	29	1

Table 28. Sound Module IOMap Offsets

3.6 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.

Module Constants	Value
IOCtrlModuleName	"IOCtrl.mod"
IOCtrlModuleID	0x00060001

Table 29. IOCtrl Module Constants

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();
```

3.6.1 IOMap Offsets

IOCtrl Module Offsets	Value	Size
IOCtrlOffsetPowerOn	0	2

Table 30. IOCtrl Module IOMap Offsets

3.7 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.

Module Constants	Value
DisplayModuleName	"Display.mod"
DisplayModuleID	0x000A0001

Table 31. Display Module Constants

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 make 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.

3.7.1 High-level functions

NumOut(x, y, value, clear = false) **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 the optional "clear" argument. If this argument is not specified it defaults to false.

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

TextOut(x, y, msg, clear = false) **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 the optional "clear" argument. If this argument is not specified it defaults to false.

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

GraphicOut(x, y, filename, clear = false) **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 the optional "clear" argument. If this argument is not specified it defaults to false. If the file cannot be found then nothing will be drawn and no errors will be reported.

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

GraphicOutEx(x, y, filename, vars, clear = false) **Function**

Draw the specified graphic icon file on the screen at the specified x and y location. Use the values contained in the vars array to transform the drawing commands contained within the specified icon file. Optionally clear the screen first depending on the boolean value of the optional "clear" argument. If this argument is not specified it defaults to false. If the file cannot be found then nothing will be drawn and no errors will be reported.

```
GraphicOutEx(40, 40, "image.ric", variables);
```

CircleOut(x, y, radius, clear = false) **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 the optional "clear" argument. If this argument is not specified it defaults to false.

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

LineOut(x1, y1, x2, y2, clear = false) **Function**

Draw a line on the screen from x1, y1 to x2, y2. Optionally clear the screen first depending on the boolean value of the optional "clear" argument. If this argument is not specified it defaults to false.

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

PointOut(x, y, clear = false) **Function**

Draw a point on the screen at x, y. Optionally clear the screen first depending on the boolean value of the optional "clear" argument. If this argument is not specified it defaults to false.

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

RectOut(x, y, width, height, clear = false) **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 the optional "clear" argument. If this argument is not specified it defaults to false.

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

ResetScreen() **Function**

Restore the standard NXT running program screen.

```
ResetScreen();
```

ClearScreen() **Function**

Clear the NXT LCD to a blank screen.

```
ClearScreen();
```

3.7.2 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 32. Display Flags Constants

DisplayFlags() **Value**

Return the current display flags. Valid flag values are listed in Table 32.

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

SetDisplayFlags(n) **Function**

Set the current display flags. Valid flag values are listed in Table 32.

```
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);
```

3.7.3 IOMap Offsets

Display Module Offsets	Value	Size
DisplayOffsetPFunc	0	4
DisplayOffsetEraseMask	4	4
DisplayOffsetUpdateMask	8	4
DisplayOffsetPFont	12	4
DisplayOffsetPTextLines(p)	(((p)*4)+16)	4*8
DisplayOffsetPStatusText	48	4
DisplayOffsetPStatusIcons	52	4
DisplayOffsetPScreens(p)	(((p)*4)+56)	4*3
DisplayOffsetPBitmaps(p)	(((p)*4)+68)	4*4
DisplayOffsetPMenuText	84	4
DisplayOffsetPMenuIcons(p)	(((p)*4)+88)	4*3
DisplayOffsetPStepIcons	100	4
DisplayOffsetDisplay	104	4
DisplayOffsetStatusIcons(p)	((p)+108)	1*4
DisplayOffsetStepIcons(p)	((p)+112)	1*5
DisplayOffsetFlags	117	1
DisplayOffsetTextLinesCenterFlags	118	1
DisplayOffsetNormal(l,w)	(((l)*100)+(w)+119)	800
DisplayOffsetPopup(l,w)	(((l)*100)+(w)+919)	800

Table 33. Display Module IOMap Offsets

3.8 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.

Module Constants	Value
LoaderModuleName	"Loader.mod"
LoaderModuleID	0x00090001

Table 34. Loader Module Constants

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 35. 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. A file created with a size of zero bytes cannot be written to since the NXC file writing functions do not grow the file if its capacity is exceeded during a write attempt.

```
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);
```

3.8.1 IOMap Offsets

Loader Module Offsets	Value	Size
LoaderOffsetPFunc	0	4
LoaderOffsetFreeUserFlash	4	4

Table 36. Loader Module IOMap Offsets

3.9 Command Module

The NXT command module encompasses support for the execution of user programs via the NXT virtual machine. It also implements the direct command protocol support that enables the NXT to respond to USB or Bluetooth requests from other devices such as a PC or another NXT brick.

Module Constants	Value
CommandModuleName	"Command.mod"
CommandModuleID	0x00010001

Table 37. Command Module Constants

3.9.1 IOMap Offsets

Command Module Offsets	Value	Size
CommandOffsetFormatString	0	16
CommandOffsetPRCHandler	16	4
CommandOffsetTick	20	4
CommandOffsetOffsetDS	24	2
CommandOffsetOffsetDVA	26	2
CommandOffsetProgStatus	28	1
CommandOffsetAwake	29	1
CommandOffsetActivateFlag	30	1
CommandOffsetDeactivateFlag	31	1
CommandOffsetFileName	32	20
CommandOffsetMemoryPool	52	32k

Table 38. Command Module IOMap Offsets

3.10 Button Module

The NXT button module encompasses support for the 4 buttons on the NXT brick.

Module Constants	Value
ButtonModuleName	"Button.mod"
ButtonModuleID	0x00040001

Table 39. Button Module Constants

3.10.1 High-level functions

Valid button constant values are listed in the following table.

Button Constants	Value
BTN1, BTNEXIT	0
BTN2, BTNRIGHT	1
BTN3, BTNLEFT	2
BTN4, BTNCENTER	3
NO_OF_BTNS	4

Table 40. Button Constants

ButtonCount(btn, reset) **Value**

Return the number of times the specified button has been pressed since the last time the button press count was reset. Optionally clear the count after reading it. Valid values for the btn argument are listed in Table 40.

```
value = ButtonCount(BTN1, true);
```

ButtonPressed(btn, reset) **Value**

Return whether the specified button is pressed. Optionally clear the press count. Valid values for the btn argument are listed in Table 40.

```
value = ButtonPressed(BTN1, true);
```

ReadButtonEx(btn, reset, out pressed, out count) **Function**

Read the specified button. Set the pressed and count parameters with the current state of the button. Optionally reset the press count after reading it. Valid values for the btn argument are listed in Table 40.

```
ReadButtonEx(BTN1, true, pressed, count);
```

3.10.2 Low-level functions

Valid button state values are listed in the following table.

Button State Constants	Value
BTNSTATE_PRESSED_EV	0x01
BTNSTATE_SHORT_RELEASED_EV	0x02
BTNSTATE_LONG_PRESSED_EV	0x04
BTNSTATE_LONG_RELEASED_EV	0x08
BTNSTATE_PRESSED_STATE	0x80

Table 41. Button State Constants

ButtonPressCount(btn) **Value**

Return the press count of the specified button. Valid values for the btn argument are listed in Table 40.

```
value = ButtonPressCount(BTN1);
```

SetButtonPressCount(btn, value) Function

Set the press count of the specified button. Valid values for the btn argument are listed in Table 40.

```
SetButtonPressCount(BTN1, value);
```

ButtonLongPressCount(btn) Value

Return the long press count of the specified button. Valid values for the btn argument are listed in Table 40.

```
value = ButtonLongPressCount(BTN1);
```

SetButtonLongPressCount(btn, value) Function

Set the long press count of the specified button. Valid values for the btn argument are listed in Table 40.

```
SetButtonLongPressCount(BTN1, value);
```

ButtonShortReleaseCount(btn) Value

Return the short release count of the specified button. Valid values for the btn argument are listed in Table 40.

```
value = ButtonShortReleaseCount(BTN1);
```

SetButtonShortReleaseCount(btn, value) Function

Set the short release count of the specified button. Valid values for the btn argument are listed in Table 40.

```
SetButtonShortReleaseCount(BTN1, value);
```

ButtonLongReleaseCount(btn) Value

Return the long release count of the specified button. Valid values for the btn argument are listed in Table 40.

```
value = ButtonLongReleaseCount(BTN1);
```

SetButtonLongReleaseCount(btn, value) Function

Set the long release count of the specified button. Valid values for the btn argument are listed in Table 40.

```
SetButtonLongReleaseCount(BTN1, value);
```

ButtonReleaseCount(btn) Value

Return the release count of the specified button. Valid values for the btn argument are listed in Table 40.

```
value = ButtonReleaseCount(BTN1);
```

SetButtonReleaseCount(btn, value) Function

Set the release count of the specified button. Valid values for the btn argument are listed in Table 40.

```
SetButtonReleaseCount(BTN1, value);
```

ButtonState(btn) Value

Return the state of the specified button. Valid values for the btn argument are listed in Table 40. Button state values are listed in Table 41.

```
value = ButtonState(BTN1);
```

SetButtonState(btn, value) Function

Set the state of the specified button. Valid values for the btn argument are listed in Table 40. Button state values are listed in Table 41.

```
SetButtonState(BTN1, BTNSTATE_PRESSED_EV);
```

3.10.3 IOMap Offsets

Button Module Offsets	Value	Size
ButtonOffsetPressedCnt(b)	(((b)*8)+0)	1
ButtonOffsetLongPressCnt(b)	(((b)*8)+1)	1
ButtonOffsetShortRelCnt(b)	(((b)*8)+2)	1
ButtonOffsetLongRelCnt(b)	(((b)*8)+3)	1
ButtonOffsetRelCnt(b)	(((b)*8)+4)	1
ButtonOffsetState(b)	((b)+32)	1*4

Table 42. Button Module IOMap Offsets

3.11 UI Module

The NXT UI module encompasses support for various aspects of the user interface for the NXT brick.

Module Constants	Value
UIModuleName	"Ui.mod"
UIModuleID	0x000C0001

Table 43. UI Module Constants

Valid command flag values are listed in the following table.

UI Command Flags Constants	Value
UI_FLAGS_UPDATE	0x01
UI_FLAGS_DISABLE_LEFT_RIGHT_ENTER	0x02
UI_FLAGS_DISABLE_EXIT	0x04
UI_FLAGS_REDRAW_STATUS	0x08
UI_FLAGS_RESET_SLEEP_TIMER	0x10
UI_FLAGS_EXECUTE_LMS_FILE	0x20
UI_FLAGS_BUSY	0x40
UI_FLAGS_ENABLE_STATUS_UPDATE	0x80

Table 44. UI Command Flags Constants

Valid UI state values are listed in the following table.

UI State Constants	Value
UI_STATE_INIT_DISPLAY	0
UI_STATE_INIT_LOW_BATTERY	1
UI_STATE_INIT_INTRO	2
UI_STATE_INIT_WAIT	3
UI_STATE_INIT_MENU	4
UI_STATE_NEXT_MENU	5
UI_STATE_DRAW_MENU	6
UI_STATE_TEST_BUTTONS	7
UI_STATE_LEFT_PRESSED	8
UI_STATE_RIGHT_PRESSED	9
UI_STATE_ENTER_PRESSED	10
UI_STATE_EXIT_PRESSED	11
UI_STATE_CONNECT_REQUEST	12
UI_STATE_EXECUTE_FILE	13
UI_STATE_EXECUTING_FILE	14
UI_STATE_LOW_BATTERY	15
UI_STATE_BT_ERROR	16

Table 45. UI State Constants

Valid UI button values are listed in the following table.

UI Button Constants	Value
UI_BUTTON_NONE	1
UI_BUTTON_LEFT	2
UI_BUTTON_ENTER	3
UI_BUTTON_RIGHT	4
UI_BUTTON_EXIT	5

Table 46. UI Button Constants

Valid UI Bluetooth state values are listed in the following table.

UI Bluetooth State Constants	Value
UI_BT_STATE_VISIBLE	0x01
UI_BT_STATE_CONNECTED	0x02
UI_BT_STATE_OFF	0x04
UI_BT_ERROR_ATTENTION	0x08
UI_BT_CONNECT_REQUEST	0x40
UI_BT_PIN_REQUEST	0x80

Table 47. UI Bluetooth State Constants

Volume() Value

Return the user interface volume level. Valid values are from 0 to 4.

```
x = Volume();
```

SetVolume(value) Function

Set the user interface volume level. Valid values are from 0 to 4.

```
SetVolume(3);
```

BatteryLevel() Value

Return the battery level in millivolts.

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

BluetoothState() Value

Return the Bluetooth state. Valid Bluetooth state values are listed in Table 47.

```
x = BluetoothState();
```

SetBluetoothState(value) Function

Set the Bluetooth state. Valid Bluetooth state values are listed in Table 47.

```
SetBluetoothState(UI_BT_STATE_OFF);
```

CommandFlags() Value

Return the command flags. Valid command flag values are listed in Table 44.

```
x = CommandFlags();
```

SetCommandFlags(value) Function

Set the command flags. Valid command flag values are listed in Table 44.

```
SetCommandFlags(UI_FLAGS_REDRAW_STATUS);
```

UIState() Value

Return the user interface state. Valid user interface state values are listed in Table 45.

```
x = UIState();
```

SetUIState(value) Function

Set the user interface state. Valid user interface state values are listed in Table 45.

```
SetUIState(UI_STATE_LOW_BATTERY);
```

UIButton() Value

Return user interface button information. Valid user interface button values are listed in Table 46.

```
x = UIButton();
```

SetUIButton(value) Function

Set user interface button information. Valid user interface button values are listed in Table 46.

```
SetUIButton(UI_BUTTON_ENTER);
```

VMRunState() Value

Return VM run state information.

```
x = VMRunState();
```

SetVMRunState(value) Function

Set VM run state information.

```
SetVMRunState(0); // stopped
```

BatteryState() Value

Return battery state information (0..4).

```
x = BatteryState();
```

SetBatteryState(value) Function

Set battery state information.

```
SetBatteryState(4);
```

RechargeableBattery() Value

Return whether the NXT has a rechargeable battery installed or not.

```
x = RechargeableBattery();
```

ForceOff(n) Function

Force the NXT to turn off if the specified value is greater than zero.

```
ForceOff(true);
```

UsbState() Value

Return USB state information (0=disconnected, 1=connected, 2=working).

```
x = UsbState();
```

SetUsbState(value) **Function**

Set USB state information (0=disconnected, 1=connected, 2=working).

```
SetUsbState(2);
```

OnBrickProgramPointer() **Value**

Return the current OBP (on-brick program) step;

```
x = OnBrickProgramPointer();
```

SetOnBrickProgramPointer(value) **Function**

Set the current OBP (on-brick program) step.

```
SetOnBrickProgramPointer(2);
```

3.11.1 IOMap Offsets

UI Module Offsets	Value	Size
UIOffsetPMenu	0	4
UIOffsetBatteryVoltage	4	2
UIOffsetLMSfilename	6	20
UIOffsetFlags	26	1
UIOffsetState	27	1
UIOffsetButton	28	1
UIOffsetRunState	29	1
UIOffsetBatteryState	30	1
UIOffsetBluetoothState	31	1
UIOffsetUsbState	32	1
UIOffsetSleepTimeout	33	1
UIOffsetSleepTimer	34	1
UIOffsetRechargeable	35	1
UIOffsetVolume	36	1
UIOffsetError	37	1
UIOffsetOBPPointer	38	1
UIOffsetForceOff	39	1

Table 48. UI Module IOMap Offsets

3.12 LowSpeed Module

The NXT low speed module encompasses support for digital I2C sensor communication.

Module Constants	Value
LowSpeedModuleName	"Low Speed.mod"
LowSpeedModuleID	0x000B0001

Table 49. LowSpeed Module Constants

Use the lowspeed (aka I2C) communication methods to access devices that use the I2C protocol on the NXT brick's four input ports.

You must set the input port's Type property to `SENSOR_TYPE_LOWSPEED` or `SENSOR_TYPE_LOWSPEED_9V` on a given port before using an I2C device on that port. Use `SENSOR_TYPE_LOWSPEED_9V` if your device requires 9V power from the NXT brick. Remember that you also need to set the input port's `InvalidData` property to true after setting a new Type, and then wait in a loop for the NXT firmware to set `InvalidData` back to false. This process ensures that the firmware has time to properly initialize the port, including the 9V power lines, if applicable. Some digital devices might need additional time to initialize after power up.

The SetSensorLowspeed API function sets the specified port to `SENSOR_TYPE_LOWSPEED_9V` and calls ResetSensor to perform the InvalidData reset loop described above.

When communicating with I2C devices, the NXT firmware uses a master/slave setup in which the NXT brick is always the master device. This means that the firmware is responsible for controlling the write and read operations. The NXT firmware maintains write and read buffers for each port, and the three main Lowspeed (I2C) methods described below enable you to access these buffers.

A call to LowspeedWrite starts an asynchronous transaction between the NXT brick and a digital I2C device. The program continues to run while the firmware manages sending bytes from the write buffer and reading the response bytes from the device. Because the NXT is the master device, you must also specify the number of bytes to expect from the device in response to each write operation. You can exchange up to 16 bytes in each direction per transaction.

After you start a write transaction with LowspeedWrite, use LowspeedStatus in a loop to check the status of the port. If LowspeedStatus returns a status code of 0 and a count of bytes available in the read buffer, the system is ready for you to use LowspeedRead to copy the data from the read buffer into the buffer you provide.

Note that any of these calls might return various status codes at any time. A status code of 0 means the port is idle and the last transaction (if any) did not result in any errors. Negative status codes and the positive status code 32 indicate errors. There are a few possible errors per call.

Valid low speed return values are listed in the following table.

Low Speed Return Constants	Value	Meaning
NO_ERR	0	The operation succeeded.
STAT_COMM_PENDING	32	The specified port is busy performing a communication transaction.
ERR_INVALID_SIZE	-19	The specified buffer or byte count exceeded the 16 byte limit.
ERR_COMM_CHAN_NOT_READY	-32	The specified port is busy or improperly configured.
ERR_COMM_CHAN_INVALID	-33	The specified port is invalid. It must be between 0 and 3.
ERR_COMM_BUS_ERR	-35	The last transaction failed, possibly due to a device failure.

Table 50. Lowspeed (I2C) Return Value Constants

3.12.1 High-level functions

LowspeedWrite(port, returnlen, buffer) Value

This method starts a transaction to write the bytes contained in the array buffer to the I2C device on the specified port. It also tells the I2C device the number of bytes that should be included in the response. The maximum number of bytes that can be written or read is 16. The port may be specified using a constant (e.g., `S1`, `S2`, `S3`, or `S4`) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other threads. Lowspeed return values are listed in Table 50.

```
x = LowspeedWrite(S1, 1, inbuffer);
```

LowspeedStatus(port, out bytesready) Value

This method checks the status of the I2C communication on the specified port. If the last operation on this port was a successful LowspeedWrite call that requested response data from the device then bytesready will be set to the number of bytes in the internal read buffer. The port may be specified using a constant (e.g., `S1`, `S2`, `S3`, or `S4`) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other threads. Lowspeed return values are listed in Table 50. If the return value is 0 then the last operation did not cause any errors. Avoid calls to LowspeedRead or LowspeedWrite while LowspeedStatus returns `STAT_COMM_PENDING`.

```
x = LowspeedStatus(S1, nRead);
```

LowspeedRead(port, buflen, out buffer) Value

Read the specified number of bytes from the I2C device on the specified port and store the bytes read in the array buffer provided. The maximum number of bytes that can be written or read is 16. The port may be specified using a constant (e.g., `S1`, `S2`, `S3`, or `S4`) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other threads. Lowspeed return values are listed in Table 50. If the return value is negative then the output buffer will be empty.

```
x = LowspeedRead(S1, 1, outbuffer);
```

I2CWrite(port, returnlen, buffer) Value

This is an alias for LowspeedWrite.

```
x = I2CWrite(S1, 1, inbuffer);
```

I2CStatus(port, out bytesready) Value

This is an alias for LowspeedStatus.

```
x = I2CStatus(S1, nRead);
```

I2CRead(port, buflen, out buffer) Value

This is an alias for LowspeedRead.

```
x = I2CRead(S1, 1, outbuffer);
```

I2CBytes(port, inbuf, in/out count, out outbuf) Value

This method writes the bytes contained in the input buffer (inbuf) to the I2C device on the specified port, checks for the specified number of bytes to be ready for reading, and then tries to read the specified number (count) of bytes from the I2C device into the output buffer (outbuf). The port may be specified using a constant (e.g., `S1`, `S2`, `S3`, or `S4`) or a variable. Returns true or false indicating whether the I2C read process succeeded or failed.

This is a higher-level wrapper around the three main I2C functions. It also maintains a "last good read" buffer and returns values from that buffer if the I2C communication transaction fails.

```
x = I2CBytes(S4, writebuf, cnt, readbuf);
```

3.12.2 Low-level functions

Valid low speed state values are listed in the following table.

Low Speed State Constants	Value
COM_CHANNEL_NONE_ACTIVE	0x00
COM_CHANNEL_ONE_ACTIVE	0x01
COM_CHANNEL_TWO_ACTIVE	0x02
COM_CHANNEL_THREE_ACTIVE	0x04
COM_CHANNEL_NONE_ACTIVE	0x08

Table 51. Low Speed State Constants

Valid low speed channel state values are listed in the following table.

Low Speed Channel State Constants	Value
LOWSPEED_IDLE	0
LOWSPEED_INIT	1
LOWSPEED_LOAD_BUFFER	2
LOWSPEED_COMMUNICATING	3
LOWSPEED_ERROR	4
LOWSPEED_DONE	5

Table 52. Low Speed Channel State Constants

Valid low speed mode values are listed in the following table.

Low Speed Mode Constants	Value
LOWSPEED_TRANSMITTING	1
LOWSPEED_RECEIVING	2
LOWSPEED_DATA_RECEIVED	3

Table 53. Low Speed Mode Constants

Valid low speed error type values are listed in the following table.

Low Speed Error Type Constants	Value
LOWSPEED_NO_ERROR	0
LOWSPEED_CH_NOT_READY	1
LOWSPEED_TX_ERROR	2
LOWSPEED_RX_ERROR	3

Table 54. Low Speed Error Type Constants

GetLSInputBuffer(port, offset, count, out data) Function

This method reads data from the lowspeed input buffer associated with the specified port.

```
GetLSInputBuffer(S1, 0, 8, buffer);
```

SetLSInputBuffer(port, offset, count, data) Function

This method writes data to the lowspeed input buffer associated with the specified port.

```
SetLSInputBuffer(S1, 0, 8, data);
```

GetLSOutputBuffer(port, offset, count, out data) Function

This method reads data from the lowspeed output buffer associated with the specified port.

```
GetLSOutputBuffer(S1, 0, 8, outbuffer);
```

SetLSOutputBuffer(port, offset, count, data) Function

This method writes data to the lowspeed output buffer associated with the specified port.

```
SetLSOutputBuffer(S1, 0, 8, data);
```

LSInputBufferInPtr(port) Value

This method returns the value of the input pointer for the lowspeed input buffer associated with the specified port. The port must be a constant (S1..S4).

```
x = LSInputBufferInPtr(S1);
```

SetLSInputBufferInPtr(port) Function

This method sets the value of the input pointer for the lowspeed input buffer associated with the specified port. The port must be a constant (S1..S4).

```
SetLSInputBufferInPtr(S1, x);
```

LSInputBufferOutPtr(port) Value

This method returns the value of the output pointer for the lowspeed input buffer associated with the specified port. The port must be a constant (S1..S4).

```
x = LSInputBufferOutPtr(S1);
```

SetLSInputBufferOutPtr(port) Function

This method sets the value of the output pointer for the lowspeed input buffer associated with the specified port. The port must be a constant (S1..S4).

```
SetLSInputBufferOutPtr(S1, x);
```

LSInputBufferBytesToRx(port) Value

This method returns the bytes to receive for the lowspeed input buffer associated with the specified port. The port must be a constant (S1..S4).

```
x = LSInputBufferBytesToRx(S1);
```

SetLSInputBufferBytesToRx(port) Function

This method sets the bytes to receive for the lowspeed input buffer associated with the specified port. The port must be a constant (S1..S4).

```
SetLSInputBufferBytesToRx(S1, x);
```

LSOutputBufferInPtr(port) Value

This method returns the value of the input pointer for the lowspeed output buffer associated with the specified port. The port must be a constant (S1..S4).

```
x = LSOutputBufferInPtr(S1);
```

SetLSOutputBufferInPtr(port) Function

This method sets the value of the input pointer for the lowspeed output buffer associated with the specified port. The port must be a constant (S1..S4).

```
SetLSOutputBufferInPtr(S1, x);
```

LSOutputBufferOutPtr(port) Value

This method returns the value of the output pointer for the lowspeed output buffer associated with the specified port. The port must be a constant (S1..S4).

```
x = LSOutputBufferOutPtr(S1);
```

SetLSOutputBufferOutPtr(port) Function

This method sets the value of the output pointer for the lowspeed output buffer associated with the specified port. The port must be a constant (S1..S4).

```
SetLSOutputBufferOutPtr(S1, x);
```

LSOutputBufferBytesToRx(port) Value

This method returns the bytes to receive for the lowspeed output buffer associated with the specified port. The port must be a constant (S1..S4).

```
x = LSOutputBufferBytesToRx(S1);
```

SetLSOutputBufferBytesToRx(port) Function

This method sets the bytes to receive for the lowspeed output buffer associated with the specified port. The port must be a constant (S1..S4).

```
SetLSOutputBufferBytesToRx(S1, x);
```

LSMode(port) Value

This method returns the mode of the lowspeed communication over the specified port. The port must be a constant (S1..S4).

```
x = LSMode(S1);
```

SetLSMode(port) Function

This method sets the mode of the lowspeed communication over the specified port. The port must be a constant (S1..S4).

```
SetLSMode(S1, LOWSPEED_TRANSMITTING);
```

LSChannelState(port) Value

This method returns the channel state of the lowspeed communication over the specified port. The port must be a constant (S1..S4).

```
x = LSChannelState(S1);
```

SetLSChannelState(port) Function

This method sets the channel state of the lowspeed communication over the specified port. The port must be a constant (S1..S4).

```
SetLSChannelState(S1, LOWSPEED_IDLE);
```

LSErrorType(port) Value

This method returns the error type of the lowspeed communication over the specified port. The port must be a constant (S1..S4).

```
x = LSErrorType(S1);
```

SetLSErrorType(port) Function

This method sets the error type of the lowspeed communication over the specified port. The port must be a constant (S1..S4).

```
SetLSErrorType(S1, LOWSPEED_CH_NOT_READY);
```

LSState() Value

This method returns the state of the lowspeed module.

```
x = LSState();
```

SetLSState(n) Function

This method sets the state of the lowspeed module.

```
SetLSState(COM_CHANNEL_THREE_ACTIVE);
```

LSSpeed() Value

This method returns the speed of the lowspeed module.

```
x = LSSpeed();
```

SetLSSpeed(n) Function

This method sets the speed of the lowspeed module.

```
SetLSSpeed(100);
```

3.12.3 IOMap Offsets

LowSpeed Module Offsets	Value	Size
LowSpeedOffsetInBufBuf(p)	(((p)*19)+0)	16
LowSpeedOffsetInBufInPtr(p)	(((p)*19)+16)	1
LowSpeedOffsetInBufOutPtr(p)	(((p)*19)+17)	1
LowSpeedOffsetInBufBytesToRx(p)	(((p)*19)+18)	58
LowSpeedOffsetOutBufBuf(p)	(((p)*19)+76)	16
LowSpeedOffsetOutBufInPtr(p)	(((p)*19)+92)	1
LowSpeedOffsetOutBufOutPtr(p)	(((p)*19)+93)	1
LowSpeedOffsetOutBufBytesToRx(p)	(((p)*19)+94)	58
LowSpeedOffsetMode(p)	((p)+152)	4
LowSpeedOffsetChannelState(p)	((p)+156)	4
LowSpeedOffsetErrorType(p)	((p)+160)	4
LowSpeedOffsetState	164	1
LowSpeedOffsetSpeed	165	1
LowSpeedOffsetSpare	166	1

Table 55. LowSpeed Module IOMap Offsets

3.13 Comm Module

The NXT comm module encompasses support for all forms of Bluetooth, USB, and HiSpeed communication.

Module Constants	Value
CommModuleName	"Comm.mod"
CommModuleID	0x00050001

Table 56. Comm Module Constants

You can use the Bluetooth communication methods to send information to other devices connected to the NXT brick. The NXT firmware also implements a message queuing or mailbox system which you can access using these methods.

Communication via Bluetooth uses a master/slave connection system. One device must be designated as the master device before you run a program using Bluetooth. If the NXT is the master device then you can configure up to three slave devices using connection 1, 2, and 3 on the NXT brick. If your NXT is a slave device then connection 0 on the brick must be reserved for the master device.

Programs running on the master NXT brick can send packets of data to any connected slave devices using the BluetoothWrite method. Slave devices write response packets to the message queuing system where they wait for the master device to poll for the response.

Using the direct command protocol, a master device can send messages to slave NXT bricks in the form of text strings addressed to a particular mailbox. Each mailbox on the slave NXT brick is a circular message queue holding up to five messages. Each message can be up to 58 bytes long.

To send messages from a master NXT brick to a slave brick, use BluetoothWrite on the master brick to send a MessageWrite direct command packet to the slave. Then, you can use ReceiveMessage on the slave brick to read the message. The slave NXT brick must be running a program when an incoming message packet is received. Otherwise, the slave NXT brick ignores the message and the message is dropped.

3.13.1 High-level functions

SendRemoteBool(connection, queue, bvalue) Value

This method sends a boolean value to the device on the specified connection. The message containing the boolean value will be written to the specified queue on the remote brick.

```
x = SendRemoteBool(1, queue, false);
```

SendRemoteNumber(connection, queue, value) Value

This method sends a numeric value to the device on the specified connection. The message containing the numeric value will be written to the specified queue on the remote brick.

```
x = SendRemoteNumber(1, queue, 123);
```

SendRemoteString(connection, queue, strval) Value

This method sends a string value to the device on the specified connection. The message containing the string value will be written to the specified queue on the remote brick.

```
x = SendRemoteString(1, queue, "hello world");
```

SendResponseBool(queue, bvalue) Value

This method sends a boolean value as a response to a received message. The message containing the boolean value will be written to the specified queue (+10) on the slave brick so that it can be retrieved by the master brick via automatic polling.

```
x = SendResponseBool(queue, false);
```

SendResponseNumber(queue, value) Value

This method sends a numeric value as a response to a received message. The message containing the numeric value will be written to the specified queue (+10) on the slave brick so that it can be retrieved by the master brick via automatic polling.

```
x = SendResponseNumber(queue, 123);
```

SendResponseString(queue, strval) Value

This method sends a string value as a response to a received message. The message containing the string value will be written to the specified queue (+10) on the slave brick so that it can be retrieved by the master brick via automatic polling.

```
x = SendResponseString(queue, "hello world");
```

ReceiveRemoteBool(queue, remove, out bvalue) Value

This method is used on a master brick to receive a boolean value from a slave device communicating via a specific mailbox or message queue. Optionally remove the last read message from the message queue depending on the value of the boolean remove parameter.

```
x = ReceiveRemoteBool(queue, true, bvalue);
```

ReceiveRemoteNumber(queue, remove, out value) Value

This method is used on a master brick to receive a numeric value from a slave device communicating via a specific mailbox or message queue. Optionally remove the last read message from the message queue depending on the value of the boolean remove parameter.

```
x = ReceiveRemoteBool(queue, true, value);
```

ReceiveRemoteString(queue, remove, out strval) Value

This method is used on a master brick to receive a string value from a slave device communicating via a specific mailbox or message queue. Optionally remove the last read message from the message queue depending on the value of the boolean remove parameter.

```
x = ReceiveRemoteString(queue, true, strval);
```

ReceiveRemoteMessageEx(queue, remove, out strval, out val, out bval)Value

This method is used on a master brick to receive a string, boolean, or numeric value from a slave device communicating via a specific mailbox or message queue. Optionally remove the last read message from the message queue depending on the value of the boolean remove parameter.

```
x = ReceiveRemoteMessageEx(queue, true, strval, val, bval);
```

SendMessage(queue, msg) Value

This method writes the message buffer contents to the specified mailbox or message queue. The maximum message length is 58 bytes.

```
x = SendMessage(mbox, data);
```

ReceiveMessage(queue, remove, out buffer) Value

This method retrieves a message from the specified queue and writes it to the buffer provided. Optionally removes the last read message from the message queue depending on the value of the boolean remove parameter.

```
x = RecieveMessage(mbox, true, buffer);
```

BluetoothStatus(connection) Value

This method returns the status of the specified Bluetooth connection. Avoid calling BluetoothWrite or any other API function that writes data over a Bluetooth connection while BluetoothStatus returns `STAT_COMM_PENDING`.

```
x = BluetoothStatus(1);
```

BluetoothWrite(connection, buffer) Value

This method tells the NXT firmware to write the data in the buffer to the device on the specified Bluetooth connection. Use BluetoothStatus to determine when this write request is completed.

```
x = BluetoothWrite(1, data);
```

RemoteMessageRead(connection, queue) Value

This method sends a MessageRead direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteMessageRead(1, 5);
```

RemoteMessageWrite(connection, queue, msg) Value

This method sends a MessageWrite direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteMessageWrite(1, 5, "test");
```

RemoteStartProgram(connection, filename) Value

This method sends a StartProgram direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteStartProgram(1, "myprog.rxe");
```

RemoteStopProgram(connection) Value

This method sends a StopProgram direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteStopProgram(1);
```

RemotePlaySoundFile(connection, filename, bLoop) Value

This method sends a PlaySoundFile direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemotePlaySoundFile(1, "click.rso", false);
```

RemotePlayTone(connection, frequency, duration) Value

This method sends a PlayTone direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemotePlayTone(1, 440, 1000);
```

RemoteStopSound(connection) Value

This method sends a StopSound direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteStopSound(1);
```

RemoteKeepAlive(connection) Value

This method sends a KeepAlive direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteKeepAlive(1);
```

RemoteResetScaledValue(connection, port) Value

This method sends a ResetScaledValue direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteResetScaledValue(1, S1);
```

RemoteResetMotorPosition(connection, port, bRelative) Value

This method sends a ResetMotorPosition direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteResetMotorPosition(1, OUT_A, true);
```

RemoteSetInputMode(connection, port, type, mode) Value

This method sends a SetInputMode direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteSetInputMode(1, S1,
  IN_TYPE_LOWSPEED, IN_MODE_RAW);
```

RemoteSetOutputState(connection, port, speed, mode, regmode, turnpct, runstate, tacholimit) Value

This method sends a SetOutputState direct command to the device on the specified connection. Use BluetoothStatus to determine when this write request is completed.

```
x = RemoteSetOutputState(1, OUT_A, 75, OUT_MODE_MOTORON,
  OUT_REGMODE_IDLE, 0, OUT_RUNSTATE_RUNNING, 0);
```

3.13.2 Low-level functions

Valid miscellaneous constant values are listed in the following table.

Comm Miscellaneous Constants	Value
SIZE_OF_USBBUF	64
USB_PROTOCOL_OVERHEAD	2
SIZE_OF_USBDATA	62
SIZE_OF_HSBUF	128
SIZE_OF_BTBUF	128
BT_CMD_BYTE	1
SIZE_OF_BT_DEVICE_TABLE	30
SIZE_OF_BT_CONNECT_TABLE	4
SIZE_OF_BT_NAME	16
SIZE_OF_BRICK_NAME	8
SIZE_OF_CLASS_OF_DEVICE	4
SIZE_OF_BDADDR	7
MAX_BT_MSG_SIZE	60000
BT_DEFAULT_INQUIRY_MAX	0
BT_DEFAULT_INQUIRY_TIMEOUT_LO	15
LR_SUCCESS	0x50
LR_COULD_NOT_SAVE	0x51
LR_STORE_IS_FULL	0x52
LR_ENTRY_REMOVED	0x53
LR_UNKNOWN_ADDR	0x54
USB_CMD_READY	0x01
BT_CMD_READY	0x02
HS_CMD_READY	0x04

Table 57. Comm Miscellaneous Constants

Valid BtState values are listed in the following table.

Comm BtState Constants	Value
BT_ARM_OFF	0
BT_ARM_CMD_MODE	1
BT_ARM_DATA_MODE	2

Table 58. Comm BtState Constants

Valid BtStateStatus values are listed in the following table.

Comm BtStateStatus Constants	Value
BT_BRICK_VISIBILITY	0x01
BT_BRICK_PORT_OPEN	0x02
BT_CONNECTION_0_ENABLE	0x10
BT_CONNECTION_1_ENABLE	0x20
BT_CONNECTION_2_ENABLE	0x40
BT_CONNECTION_3_ENABLE	

Table 59. Comm BtStateStatus Constants

Valid BtHwStatus values are listed in the following table.

Comm BtHwStatus Constants	Value
BT_ENABLE	0x00
BT_DISABLE	0x01

Table 60. Comm BtHwStatus Constants

Valid HsFlags values are listed in the following table.

Comm HsFlags Constants	Value
HS_UPDATE	1

Table 61. Comm HsFlags Constants

Valid HsState values are listed in the following table.

Comm HsState Constants	Value
HS_INITIALISE	1
HS_INIT_RECEIVER	2
HS_SEND_DATA	3
HS_DISABLE	4

Table 62. Comm HsState Constants

Valid DeviceStatus values are listed in the following table.

Comm DeviceStatus Constants	Value
BT_DEVICE_EMPTY	0x00
BT_DEVICE_UNKNOWN	0x01
BT_DEVICE_KNOWN	0x02
BT_DEVICE_NAME	0x40
BT_DEVICE_AWAY	0x80

Table 63. Comm DeviceStatus Constants

Valid module interface values are listed in the following table.

Comm Module Interface Constants	Value
INTF_SENDFILE	0
INTF_SEARCH	1
INTF_STOPSEARCH	2
INTF_CONNECT	3
INTF_DISCONNECT	4
INTF_DISCONNECTALL	5
INTF_REMOVEDEVICE	6
INTF_VISIBILITY	7
INTF_SETCMDMODE	8
INTF_OPENSTREAM	9
INTF_SENDDATA	10
INTF_FACTORYRESET	11
INTF_BTON	12
INTF_BTOFF	13
INTF_SETBTNAME	14
INTF_EXTREAD	15
INTF_PINREQ	16
INTF_CONNECTREQ	17

Table 64. Comm Module Interface Constants

3.13.2.1 USB functions

GetUSBInputBuffer(offset, count, out data) Function

This method reads count bytes of data from the USB input buffer at the specified offset and writes it to the buffer provided.

```
GetUSBInputBuffer(0, 10, buffer);
```

SetUSBInputBuffer(offset, count, data) Function

This method writes count bytes of data to the USB input buffer at the specified offset.

```
SetUSBInputBuffer(0, 10, buffer);
```

SetUSBInputBufferInPtr(n) Function

This method sets the input pointer of the USB input buffer to the specified value.

```
SetUSBInputBufferInPtr(0);
```

USBInputBufferInPtr() Value

This method returns the value of the input pointer of the USB input buffer.

```
byte x = USBInputBufferInPtr();
```

SetUSBInputBufferOutPtr(n) Function

This method sets the output pointer of the USB input buffer to the specified value.

```
SetUSBInputBufferOutPtr(0);
```

USBInputBufferOutPtr() Value

This method returns the value of the output pointer of the USB input buffer.

```
byte x = USBInputBufferOutPtr();
```

GetUSBOutputBuffer(offset, count, out data) Function

This method reads count bytes of data from the USB output buffer at the specified offset and writes it to the buffer provided.

```
GetUSBOutputBuffer(0, 10, buffer);
```

SetUSBOutputBuffer(offset, count, data) Function

This method writes count bytes of data to the USB output buffer at the specified offset.

```
SetUSBOutputBuffer(0, 10, buffer);
```

SetUSBOutputBufferInPtr(n) Function

This method sets the input pointer of the USB output buffer to the specified value.

```
SetUSBOutputBufferInPtr(0);
```

USBOutputBufferInPtr() Value

This method returns the value of the input pointer of the USB output buffer.

```
byte x = USBOutputBufferInPtr();
```

SetUSBOutputBufferOutPtr(n) Function

This method sets the output pointer of the USB output buffer to the specified value.

```
SetUSBOutputBufferOutPtr(0);
```

USBOutputBufferOutPtr() Value

This method returns the value of the output pointer of the USB output buffer.

```
byte x = USBOutputBufferOutPtr();
```

GetUSBPollBuffer(offset, count, out data) Function

This method reads count bytes of data from the USB poll buffer and writes it to the buffer provided.

```
GetUSBPollBuffer(0, 10, buffer);
```

SetUSBPollBuffer(offset, count, data) Function

This method writes count bytes of data to the USB poll buffer at the specified offset.

```
SetUSBPollBuffer(0, 10, buffer);
```

SetUSBPollBufferInPtr(n) Function

This method sets the input pointer of the USB poll buffer to the specified value.

```
SetUSBPollBufferInPtr(0);
```

USBPollBufferInPtr() Value

This method returns the value of the input pointer of the USB poll buffer.

```
byte x = USBPollBufferInPtr();
```

SetUSBPollBufferOutPtr(n) Function

This method sets the output pointer of the USB poll buffer to the specified value.

```
SetUSBPollBufferOutPtr(0);
```

USBPollBufferOutPtr() Value

This method returns the value of the output pointer of the USB poll buffer.

```
byte x = USBPollBufferOutPtr();
```

SetUSBState(n) Function

This method sets the USB state to the specified value.

```
SetUSBState(0);
```

USBState() Value

This method returns the USB state.

```
byte x = USBPollBufferOutPtr();
```

3.13.2.2 High Speed port functions

GetHSInputBuffer(offset, count, out data) Function

This method reads count bytes of data from the High Speed input buffer and writes it to the buffer provided.

```
GetHSInputBuffer(0, 10, buffer);
```

SetHSInputBuffer(offset, count, data) Function

This method writes count bytes of data to the High Speed input buffer at the specified offset.

```
SetHSInputBuffer(0, 10, buffer);
```

SetHSInputBufferInPtr(n) Function

This method sets the input pointer of the High Speed input buffer to the specified value.

```
SetHSInputBufferInPtr(0);
```

HSInputBufferInPtr() Value

This method returns the value of the input pointer of the High Speed input buffer.

```
byte x = HSInputBufferInPtr();
```

SetHSInputBufferOutPtr(n) Function

This method sets the output pointer of the High Speed input buffer to the specified value.

```
SetHSInputBufferOutPtr(0);
```

HSInputBufferOutPtr() Value

This method returns the value of the output pointer of the High Speed input buffer.

```
byte x = HSInputBufferOutPtr();
```

GetHSOutputBuffer(offset, count, out data) Function

This method reads count bytes of data from the High Speed output buffer and writes it to the buffer provided.

```
GetHSOutputBuffer(0, 10, buffer);
```

SetHSOutputBuffer(offset, count, data) Function

This method writes count bytes of data to the High Speed output buffer at the specified offset.

```
SetHSOutputBuffer(0, 10, buffer);
```

SetHSOutputBufferInPtr(n) Function

This method sets the Output pointer of the High Speed output buffer to the specified value.

```
SetHSOutputBufferInPtr(0);
```

HSOutputBufferInPtr() Value

This method returns the value of the Output pointer of the High Speed output buffer.

```
byte x = HSOutputBufferInPtr();
```

SetHSOutputBufferOutPtr(n) — Function

This method sets the output pointer of the High Speed output buffer to the specified value.

```
SetHSOutputBufferOutPtr(0);
```

HSOutputBufferOutPtr() — Value

This method returns the value of the output pointer of the High Speed output buffer.

```
byte x = HSOutputBufferOutPtr();
```

SetHSFlags(n) — Function

This method sets the High Speed flags to the specified value.

```
SetHSFlags(0);
```

HSFlags() — Value

This method returns the value of the High Speed flags.

```
byte x = HSFlags();
```

SetHSSpeed(n) — Function

This method sets the High Speed speed to the specified value.

```
SetHSSpeed(1);
```

HSSpeed() — Value

This method returns the value of the High Speed speed.

```
byte x = HSSpeed();
```

SetHSState(n) — Function

This method sets the High Speed state to the specified value.

```
SetHSState(1);
```

HSState() — Value

This method returns the value of the High Speed state.

```
byte x = HSState();
```

3.13.2.3 Bluetooth functions

GetBTInputBuffer(offset, count, out data) — Function

This method reads count bytes of data from the Bluetooth input buffer and writes it to the buffer provided.

```
GetBTInputBuffer(0, 10, buffer);
```

SetBTInputBuffer(offset, count, data) Function

This method writes count bytes of data to the Bluetooth input buffer at the specified offset.

```
SetBTInputBuffer(0, 10, buffer);
```

SetBTInputBufferInPtr(n) Function

This method sets the input pointer of the Bluetooth input buffer to the specified value.

```
SetBTInputBufferInPtr(0);
```

BTInputBufferInPtr() Value

This method returns the value of the input pointer of the Bluetooth input buffer.

```
byte x = BTInputBufferInPtr();
```

SetBTInputBufferOutPtr(n) Function

This method sets the output pointer of the Bluetooth input buffer to the specified value.

```
SetBTInputBufferOutPtr(0);
```

BTInputBufferOutPtr() Value

This method returns the value of the output pointer of the Bluetooth input buffer.

```
byte x = BTInputBufferOutPtr();
```

GetBTOutputBuffer(offset, count, out data) Function

This method reads count bytes of data from the Bluetooth output buffer and writes it to the buffer provided.

```
GetBTOutputBuffer(0, 10, buffer);
```

SetBTOutputBuffer(offset, count, data) Function

This method writes count bytes of data to the Bluetooth output buffer at the specified offset.

```
SetBTOutputBuffer(0, 10, buffer);
```

SetBTOutputBufferInPtr(n) Function

This method sets the input pointer of the Bluetooth output buffer to the specified value.

```
SetBTOutputBufferInPtr(0);
```

BTOutputBufferInPtr() Value

This method returns the value of the input pointer of the Bluetooth output buffer.

```
byte x = BTOutputBufferInPtr();
```

SetBTOutputBufferOutPtr(n) Function

This method sets the output pointer of the Bluetooth output buffer to the specified value.

```
SetBTOutputBufferOutPtr(0);
```

BTOutputBufferOutPtr() Value

This method returns the value of the output pointer of the Bluetooth output buffer.

```
byte x = BTOutputBufferOutPtr();
```

BTDeviceCount() Value

This method returns the number of devices defined within the Bluetooth device table.

```
byte x = BTDeviceCount();
```

BTDeviceNameCount() Value

This method returns the number of device names defined within the Bluetooth device table. This usually has the same value as BTDeviceCount but it can differ in some instances.

```
byte x = BTDeviceNameCount();
```

BTDeviceName(idx) Value

This method returns the name of the device at the specified index in the Bluetooth device table.

```
string name = BTDeviceName(0);
```

BTConnectionName(idx) Value

This method returns the name of the device at the specified index in the Bluetooth connection table.

```
string name = BTConnectionName(0);
```

BTConnectionPinCode(idx) Value

This method returns the pin code of the device at the specified index in the Bluetooth connection table.

```
string pincode = BTConnectionPinCode(0);
```

BrickDataName() Value

This method returns the name of the NXT.

```
string name = BrickDataName();
```

GetBTDeviceAddress(idx, out data) Function

This method reads the address of the device at the specified index within the Bluetooth device table and stores it in the data buffer provided.

```
GetBTDeviceAddress(0, buffer);
```

GetBTConnectionAddress(idx, out data) Function

This method reads the address of the device at the specified index within the Bluetooth connection table and stores it in the data buffer provided.

```
GetBTConnectionAddress(0, buffer);
```

GetBrickDataAddress(out data) Function

This method reads the address of the NXT and stores it in the data buffer provided.

```
GetBrickDataAddress(buffer);
```

BTDeviceClass(idx) Value

This method returns the class of the device at the specified index within the Bluetooth device table.

```
long class = BTDeviceClass(idx);
```

BTDeviceStatus(idx) Value

This method returns the status of the device at the specified index within the Bluetooth device table.

```
byte status = BTDeviceStatus(idx);
```

BTConnectionClass(idx) Value

This method returns the class of the device at the specified index within the Bluetooth connection table.

```
long class = BTConnectionClass(idx);
```

BTConnectionHandleNum(idx) Value

This method returns the handle number of the device at the specified index within the Bluetooth connection table.

```
byte handlenum = BTConnectionHandleNum(idx);
```

BTConnectionStreamStatus(idx) Value

This method returns the stream status of the device at the specified index within the Bluetooth connection table.

```
byte streamstatus = BTConnectionStreamStatus(idx);
```

BTConnectionLinkQuality(idx) Value

This method returns the link quality of the device at the specified index within the Bluetooth connection table.

```
byte linkquality = BTConnectionLinkQuality(idx);
```

BrickDataBluecoreVersion() Value

This method returns the bluecore version of the NXT.

```
int bv = BrickDataBluecoreVersion();
```

BrickDataBtStateStatus() Value

This method returns the Bluetooth state status of the NXT.

```
int x = BrickDataBtStateStatus();
```

BrickDataBtHardwareStatus() Value

This method returns the Bluetooth hardware status of the NXT.

```
int x = BrickDataBtHardwareStatus();
```

BrickDataTimeoutValue() Value

This method returns the timeout value of the NXT.

```
int x = BrickDataTimeoutValue();
```

3.13.3 IOMap Offsets

Comm Module Offsets	Value	Size
CommOffsetPFunc	0	4
CommOffsetPFuncTwo	4	4
CommOffsetBtDeviceTableName(p)	(((p)*31)+8)	16
CommOffsetBtDeviceTableClassOfDevice(p)	(((p)*31)+24)	4
CommOffsetBtDeviceTableBdAddr(p)	(((p)*31)+28)	7
CommOffsetBtDeviceTableDeviceStatus(p)	(((p)*31)+35)	1
CommOffsetBtConnectTableName(p)	(((p)*47)+938)	16
CommOffsetBtConnectTableClassOfDevice (p)	(((p)*47)+954)	4
CommOffsetBtConnectTablePinCode(p)	(((p)*47)+958)	16
CommOffsetBtConnectTableBdAddr(p)	(((p)*47)+974)	7
CommOffsetBtConnectTableHandleNr(p)	(((p)*47)+981)	1
CommOffsetBtConnectTableStreamStatus(p)	(((p)*47)+982)	1
CommOffsetBtConnectTableLinkQuality(p)	(((p)*47)+983)	1
CommOffsetBtConnectTableSpare(p)	(((p)*47)+984)	1
CommOffsetBrickDataName	1126	16

CommOffsetBrickDataBluecoreVersion	1142	2
CommOffsetBrickDataBdAddr	1144	7
CommOffsetBrickDataBtStateStatus	1151	1
CommOffsetBrickDataBtHwStatus	1152	1
CommOffsetBrickDataTimeOutValue	1153	1
CommOffsetBtInBufBuf	1157	128
CommOffsetBtInBufInPtr	1285	1
CommOffsetBtInBufOutPtr	1286	1
CommOffsetBtOutBufBuf	1289	128
CommOffsetBtOutBufInPtr	1417	1
CommOffsetBtOutBufOutPtr	1418	1
CommOffsetHsInBufBuf	1421	128
CommOffsetHsInBufInPtr	1549	1
CommOffsetHsInBufOutPtr	1549	1
CommOffsetHsOutBufBuf	1553	128
CommOffsetHsOutBufInPtr	1681	1
CommOffsetHsOutBufOutPtr	1682	1
CommOffsetUsbInBufBuf	1685	64
CommOffsetUsbInBufInPtr	1749	1
CommOffsetUsbInBufOutPtr	1750	1
CommOffsetUsbOutBufBuf	1753	64
CommOffsetUsbOutBufInPtr	1817	1
CommOffsetUsbOutBufOutPtr	1818	1
CommOffsetUsbPollBufBuf	1821	64
CommOffsetUsbPollBufInPtr	1885	1
CommOffsetUsbPollBufOutPtr	1886	1
CommOffsetBtDeviceCnt	1889	1
CommOffsetBtDeviceNameCnt	1890	1
CommOffsetHsFlags	1891	1
CommOffsetHsSpeed	1892	1
CommOffsetHsState	1893	1
CommOffsetUsbState	1894	1

Table 65. Comm Module IOMap Offsets