# NeXT Byte Codes (NBC)

## Programmer's Guide

Version 1.0.1 b33

by John Hansen

# Contents

1. Introduction ........ 6
2. The NBC Language ........ 7
  2.1. Lexical Rules ........ 7
    2.1.1. Comments ........ 7
    2.1.2. Whitespace ........ 7
    2.1.3. Numerical Constants ........ 8
    2.1.4. Identifiers and Keywords ........ 8
  2.2. Program Structure ........ 9
    2.2.1. Threads ........ 9
    2.2.2. Subroutines ........ 10
    2.2.3. Macro Functions ........ 11
    2.2.4. Data segments ........ 12
    2.2.4.1. Type Definitions ........ 12
    2.2.4.2. Variable Declarations ........ 13
  2.3. The Preprocessor ........ 14
    2.3.1. #include ........ 14
    2.3.2. #define ........ 15
    2.3.3. ## (Concatenation) ........ 15
    2.3.4. Conditional Compilation ........ 15
  2.4. Compiler Tokens ........ 15
  2.5. Expression Evaluator ........ 16
  2.6. IO-Map Address (IOMA) Constants ........ 17
  2.7. Statements ........ 18
    2.7.1. Assignment Statements ........ 18
    2.7.2. Math Statements ........ 18
    2.7.3. Logic Statements ........ 20
    2.7.4. Bit Manipulation Statements ........ 20
    2.7.5. Comparison Statements ........ 22
    2.7.6. Control Flow Statements ........ 22
    2.7.7. System Call Statements ........ 23
    2.7.8. Timing Statements ........ 24
    2.7.9. Array Statements ........ 25
    2.7.10. String Statements ........ 25
    2.7.11. Scheduling Statements ........ 27
    2.7.12. Input Statements ........ 29
    2.7.13. Output Statements ........ 29
    2.7.14. Compile-time Statements ........ 30
3. NBC API ........ 31
  3.1. General Features ........ 32
    3.1.1. Timing Functions ........ 32
    3.1.2. Numeric Functions ........ 32
    3.1.3. Low-level System Functions ........ 33
  3.2. Input Module ........ 52
    3.2.1. Types and Modes ........ 52
    3.2.2. Sensor Information ........ 55

3.2.3. IOMap Offsets ........ 56
3.3. Output Module ........ 57
3.3.1. Convenience Calls ........ 61
3.3.2. IOMap Offsets ........ 65
3.4. Sound Module ........ 65
3.4.1. High-level functions ........ 66
3.4.2. Low-level functions ........ 67
3.4.3. IOMap Offsets ........ 69
3.5. IOCtrl Module ........ 69
3.5.1. IOMap Offsets ........ 69
3.6. Display module ........ 69
3.6.1. High-level functions ........ 70
3.6.2. Low-level functions ........ 71
3.6.3. IOMap Offsets ........ 73
3.7. Loader Module ........ 74
3.7.1. IOMap Offsets ........ 77
3.8. Command Module ........ 77
3.8.1. IOMap Offsets ........ 78
3.9. Button Module ........ 78
3.9.1. High-level functions ........ 78
3.9.2. Low-level functions ........ 78
3.9.3. IOMap Offsets ........ 79
3.10. UI Module ........ 79
3.10.1. IOMap Offsets ........ 83
3.11. LowSpeed Module ........ 83
3.11.1. High-level functions ........ 84
3.11.2. Low-level functions ........ 86
3.11.3. IOMap Offsets ........ 87
3.12. Comm Module ........ 87
3.12.1. High-level functions ........ 88
3.12.2. Low-level functions ........ 91
3.12.3. IOMap Offsets ........ 100
3.13. HiTechnic API Functions ........ 101
3.14. Mindsensors API Functions ........ 112

# Tables

Table 1. Scalar Types .......... 13
Table 2. Compiler Tokens .......... 16
Table 3. Constant Expression Operators .......... 16
Table 4. Constant Expression Functions .......... 17
Table 5. IOMA Constant Macros .......... 18
Table 6. Comparison Constants .......... 22
Table 7. System Call Function IDs .......... 24
Table 8. Input Field IDs .......... 29
Table 9. Output Field IDs .......... 30
Table 10. Input Module Constants .......... 52
Table 11. Sensor Type Constants .......... 52
Table 12. Sensor Mode Constants .......... 53
Table 13. Sensor Field Constants .......... 53
Table 14. Input Module IOMap Offsets .......... 57
Table 15. Output Module Constants .......... 57
Table 16. Output Field Constants .......... 60
Table 17. UpdateFlag Constants .......... 60
Table 18. OutputMode Constants .......... 60
Table 19. RunState Constants .......... 60
Table 20. RegMode Constants .......... 60
Table 21. Reset Constants .......... 61
Table 22. Output Port Constants .......... 61
Table 23. Output Module IOMap Offsets .......... 65
Table 24. Sound Module Constants .......... 66
Table 25. Sound Flags Constants .......... 67
Table 26. Sound State Constants .......... 67
Table 27. Sound Mode Constants .......... 67
Table 28. Miscellaneous Sound Constants .......... 67
Table 29. Sound Module IOMap Offsets .......... 69
Table 30. IOCtrl Module Constants .......... 69
Table 31. IOCtrl Module IOMap Offsets .......... 69
Table 32. Display Module Constants .......... 69
Table 33. Display Flags Constants .......... 71
Table 34. Display Module IOMap Offsets .......... 74
Table 35. Loader Module Constants .......... 74
Table 36. Loader Result Codes .......... 74
Table 37. Loader Module IOMap Offsets .......... 77
Table 38. Command Module Constants .......... 77
Table 39. Command Module IOMap Offsets .......... 78
Table 40. Button Module Constants .......... 78
Table 41. Button Constants .......... 78
Table 42. Button State Constants .......... 78
Table 43. Button Module IOMap Offsets .......... 79
Table 44. UI Module Constants .......... 80
Table 45. UI Command Flags Constants .......... 80

Table 46. UI State Constants ........................................................................................ 80
Table 47. UI Button Constants........................................................................................ 80
Table 48. UI Bluetooth State Constants.......................................................................... 81
Table 49. UI Module IOMap Offsets.............................................................................. 83
Table 50. LowSpeed Module Constants .......................................................................... 83
Table 51. Lowspeed (I2C) Return Value Constants ........................................................ 84
Table 52. Low Speed State Constants............................................................................. 86
Table 53. Low Speed Channel State Constants ............................................................... 86
Table 54. Low Speed Mode Constants ............................................................................ 86
Table 55. Low Speed Error Type Constants.................................................................... 86
Table 56. LowSpeed Module IOMap Offsets.................................................................. 87
Table 57. Comm Module Constants ................................................................................ 87
Table 58. Comm Miscellaneous Constants...................................................................... 92
Table 59. Comm BtState Constants................................................................................. 92
Table 60. Comm BtStateStatus Constants ....................................................................... 92
Table 61. Comm BtHwStatus Constants ......................................................................... 92
Table 62. Comm HsFlags Constants................................................................................ 92
Table 63. Comm HsState Constants ................................................................................ 93
Table 64. Comm DeviceStatus Constants........................................................................ 93
Table 65. Comm Module Interface Constants ................................................................. 93
Table 66. Comm Module IOMap Offsets....................................................................... 101

# 1. Introduction

NBC stands for NeXT Byte Codes. It is a simple language for programming the LEGO MINDSTORMS NXT product. The NXT has a byte-code interpreter (provided by LEGO), which can be used to execute programs. The NBC compiler translates a source program into LEGO NXT byte-codes, which can then be executed on the NXT itself. Although the preprocessor and format of NBC programs are similar to assembly, NBC is not a general-purpose assembly language – there are many restrictions that stem from limitations of the LEGO byte-code interpreter.

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

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

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

# 2. The NBC Language

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

Unlike some assembly languages, NBC is a case-sensitive language. That means that the identifier "xYz" is not the same identifier as "Xyz". Similarly, the subtract statement begins with the keyword "sub" but "suB", "Sub", or "SUB" are all just valid identifiers – not keywords.

## *2.1. Lexical Rules*

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

### 2.1.1. Comments

Three forms of comments are supported in NBC. 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 third form of comments begins with `;` and ends with a newline (sometimes known as assembly language style comments).

```
; another 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:

```
set x,2
set   x,   2
```

Generally, whitespace is ignored outside of string constants and constant numeric expressions. However, unlike in C, NBC statements may not span multiple lines. Aside from pre-processor macros invocations, each statement in an NBC program must begin and end on the same line.

```
add x, x, 2 ; okay
add x,      // error
    x, 2    // error

set x, (2*2)+43-12 ; okay
set x, 2 * 2 ; error (constant expression contains whitespace)
```

### 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.

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

### 2.1.4. Identifiers and Keywords

Identifiers are used for variable, thread, 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 NBC language itself. These reserved words are call keywords and may not be used as identifiers. A complete list of keywords appears below:

```
add         sub         neg         mul
div         mod         and         or
xor         not         cmp         tst
index       replace     arrsize     arrbuild
arrsubset   arrinit     mov         set
flatten     unflatten   numtostr    strtonum
strcat      strsubset   strtoarr    arrtostr
jmp         brcmp       brtst       syscall
stop        exit        exitto      acquire
release     subcall     subret      setin
setout      getin       getout      wait
gettick     thread      endt        subroutine
follows     precedes    segment     ends
typedef     struct      db          byte
sbyte       ubyte       dw          word
sword       uword       dd          dword
sdword      udword      long        slong
ulong       void        mutex       waitv
call        return      abs         sign
strindex    strreplace  strlen      shl
shr         sizeof      compchk     compif
compelse    compend     valueof     isconst
```

| | | | |
|---|---|---|---|
| `asl` | `asr` | `lsl` | `lsr` |
| `rotl` | `rotr` | `start` | `stopthread` |
| `priority` | `cmnt` | `fmtnum` | `compchktype` |

## 2.2. Program Structure

An NBC program is composed of code blocks and global variables in data segments. There are two primary types of code blocks: thread and subroutines. Each of these types of code blocks has its own unique features and restrictions, but they share a common structure.

A third type of code block is the preprocessor macro function. This code block type is used throughout the NBC API. Macro functions are the only type of code block, which use a parameter passing syntax similar to what you might see in a language like C or Pascal.

Data segment blocks are used to define types and to declare variables. An NBC program can have zero or more data segments, which can be placed either outside of a code block or within a code block. Regardless of the location of the data segment, all variables in an NBC program are global.

### 2.2.1. Threads

The NXT implicitly supports multi-threading, thus an NBC thread directly corresponds to an NXT thread. Threads are defined using the `thread` keyword with the following syntax:

```
thread name
  // the thread's code is placed here
endt
```

The name of the thread may be any legal identifier. A program must always have at least one thread. If there is a thread named "main" then that thread will be the thread that is started whenever the program is run. If none of the threads are named "main" then the very first thread that the compiler encounters in the source code will be the main thread. The maximum number of threads supported by the NXT is 256.

The body of a thread consists of a list of statements and optional data segments. Threads may be started by scheduling dependant threads using the `precedes` or `follows` statements. You may also start a thread using the `start` statement. With the standard NXT firmware threads cannot be stopped by another thread. The only way to stop a thread is by stopping all threads using the `stop` statement or by a thread stopping on its own via the `exit` and `exitto` statements. Using the NBC/NBC enhanced firmware you can also stop another thread using the `stopthread` statement.

```
thread main
  precedes waiter, worker
  /* thread body goes here */
  // finalize this thread and schedule the threads in the
  // specified range to execute
  exit // all dependants are automatically scheduled
endt
```

```
thread waiter
  /* thread body goes here */
//  exit
  ; exit is optional due to smart compiler finalization
endt

thread worker
  precedes waiter
  /* thread body goes here */
  exit // only one dependent – schedule it to execute
endt
```

### 2.2.2. Subroutines

Subroutines allow a single copy of some code to be shared between several different callers. This makes subroutines much more space efficient than macro functions. Subroutines are defined using the `subroutine` keyword with the following syntax:

```
subroutine name
  // body of subroutine
  return // subroutines must end with a return statement
ends
```

A subroutine is just a special type of thread that is designed to be called explicitly by other threads or subroutines. Its name can be any legal identifier. Subroutines are not scheduled to run via the same mechanism that is used with threads. Instead, subroutines and threads execute other subroutines by using the `call` statement (described in the section titled *Statements*).

```
thread main
  /* body of main thread goes here */
  call mySub // compiler handles subroutine return address
  exit // finalize execution (details handled by the compiler)
endt

subroutine mySub
  /* body of subroutine goes here */
  return // compiler handles the subroutine return address
ends
```

You can pass arguments into and out of subroutines using global variables. If a subroutine is designed to be used by concurrently executing threads then calls to the subroutine must be protected by acquiring a mutex prior to the subroutine call and releasing the mutex after the call.

You can also call a thread as a subroutine using a slightly different syntax. This technique is required if you want to call a subroutine which executes two threads simultaneously. The `subcall` and `subret` statements must be used instead of `call` and `return`. You also must provide a global variable to store the return address as shown in the sample code below.

```
thread main
  /* thread body goes here */
  acquire ssMutex
  call SharedSub ; automatic return address
  release ssMutex
  // calling a thread as a subroutine
  subcall AnotherSub, anothersub_returnaddress
  exit
endt

subroutine SharedSub
  /* subroutine body goes here */
  return ; return is required as the last operation
ends

thread AnotherSub
  /* threads can be subroutines too */
  subret anothersub_returnaddress ; manual return address
endt
```

After the subroutine completes executing, it returns back to the calling routine and program execution continues with the next statement following the subroutine call. The maximum number of threads and subroutines supported by the NXT firmware is 256.

### 2.2.3. Macro Functions

It is often helpful to group a set of statements together into a single function, which can then be called as needed. NBC supports macro functions with arguments. Values may be returned from a macro function by changing the value of one or more of the arguments within the body of the macro function.

Macro functions are defined using the following syntax:

```
#define name(argument_list) \
  // body of the macro function \
  // last line in macro function body has no '\' at the end
```

Please note that the newline escape character ('\') must be the very last character on the line. If it is followed by any whitespace or comments then the macro body is terminated at that point and the next line is not considered to be part of the macro definition.

The argument list may be empty, or it may contain one or more argument definitions. An argument to a macro function has no *type*. Each argument is simply defined by its *name*. Multiple arguments are separated by commas. Arguments to a macro function can either be inputs (constants or variables) for the code in the body of the function to process or they can be outputs (variables only) for the code to modify and return. The following sample shows how to define a macro function to simplify the process of drawing text on the NXT LCD screen:

```
#define MyMacro(x, y, berase, msg) \
  mov dtArgs.Location.X, x \
  mov dtArgs.Location.Y, y \
  mov dtArgs.Options, berase \
```

```
    mov dtArgs.Text, msg \
    syscall DrawText, dtArgs

  MyMacro(0, 0, TRUE, 'testing')
  MyMacro(10, 20, FALSE, 'Please Work')
```

NBC macro functions are always expanded inline by the NBC preprocessor. This means that each call to a macro function results in another copy of the function's code being included in the program. Unless used judiciously, inline macro functions can lead to excessive code size.

### 2.2.4. Data segments

Data segments contain all type definitions and variable declarations. Data segments are defined using the following syntax:

```
dseg segment
  // type definitions and variable declarations go here
dseg ends

thread main
  dseg segment
    // or here – still global, though
  dseg ends
endt
```

You can have multiple data segments in an NBC program. All variables are global regardless of where they are declared. Once declared, they may be used within all threads, subroutines, and macro functions. Their scope begins at the declaration and ends at the end of the program.

#### 2.2.4.1. Type Definitions

Type definitions must be contained within a data segment. They are used to define new type aliases or new aggregate types (i.e., structures). A type alias is defined using the `typedef` keyword with the following syntax:

```
type_alias typedef existing_type
```

The new alias name may be any valid identifier.  The existing type must be some type already known by the compiler.  It can be a native type or a user-defined type. Once a type alias has been defined it can be used in subsequent variable declarations and aggregate type definitions. The following is an example of a simple type alias definition:

```
big typedef dword ; big is now an alias for the dword type
```

Structure definitions must also be contained within a data segment. They are used to define a type which aggregates or contains other native or user-defined types. A structure definition is defined using the `struct` and `ends` keywords with the following syntax:

```
TypeName struct
  x byte
  y byte
TypeName ends
```

Structure definitions allow you to manage related data in a single combined type. They can be as simple or complex as the needs of your program dictate. The following is an example of a fairly complex structure:

```
MyPoint struct
  x byte
  y byte
MyPoint ends
ComplexStrut struct
  value1 big           // using a type alias
  value2 sdword
  buffer byte[]        /* array of byte */
  blen word
  extrastuff MyPoint[]  // array of structs
  pt_1 MyPoint         // struct contains struct instances
  pt_2 MyPoint
ComplexStruct ends
```

### 2.2.4.2. Variable Declarations

All variable declarations must be contained within a data segment. They are used to declare variables for use in a code block such as a thread, subroutine, or macro function. A variable is declared using the following syntax:

```
var_name type_name optional_initialization
```

The variable name may be any valid identifier. The type name must be a type or type alias already known by the compiler. The optional initialization format depends on the variable type, but for non-aggregate (scalar) types the format is simply a constant integer or constant expression (which may not contain whitespace). See the examples later in this section.

The NXT firmware supports several different types of variables which are grouped into two categories: scalar types and aggregate types. Scalar types are a single integer value which may be signed or unsigned and occupy one, two, or four bytes of memory. The keywords for declaring variables of a scalar type are listed in the following table:

| Type Name | Information |
|---|---|
| byte, ubyte, db | 8 bit unsigned |
| sbyte | 8 bit signed |
| word, uword, dw | 16 bit unsigned |
| sword | 16 bit signed |
| dword, udword, dd | 32 bit unsigned |
| sdword | 32 bit signed |
| long, ulong | 32 bit unsigned (alias for dword, udword) |
| slong | 32 bit signed (alias for sdword) |
| mutex | Special type used for exclusive subroutine access |

**Table 1. Scalar Types**

Examples of scalar variable declarations are as follow:

```
dseg segment
  x byte           // initialized to zero by default
  y byte 12        ; initialize to 12
  z sword -2048    /* a signed value */
  myVar big 0x12345 ; use a type alias
  var1 dword 0xFF   ; value is 255
  myMutex mutex     ; mutexes ignore initialization, if present
  bTrue byte 1      ; byte variables can be used as booleans
dseg ends
```

Aggregate variables are either structures or arrays of some other type (either scalar or aggregate). Once a user-defined struct type has been defined it may be used to declare a variable of that type. Similarly, user-defined struct types can be used in array declarations. Arrays and structs may be nested (i.e., contained in other arrays or structures) as deeply as the needs of your program dictate, but nesting deeper than 2 or 3 levels may lead to slower program execution due to NXT firmware memory constraints.

Examples of aggregate variable declarations are as follow:

```
dseg segment
  buffer byte[] // starts off empty
  msg byte[] 'Testing'
  // msg is an array of byte =
  // (0x54, 0x65, 0x73, 0x74, 0x69, 0x6e, 0x67, 0x00)
  data long[] {0xabcde, 0xfade0} ; two values in the array
  myStruct ComplexStruct ; declare an instance of a struct
  Points MyPoint[] ; declare an array of a structs
  msgs byte[][] ; an array of an array of byte
dseg ends
```

Byte arrays may be initialized either by using braces containing a list of numeric values ({val1, val2, ..., valN}) or by using a string constant delimited with single-quote characters ('Testing'). Embedded single quote characters are not supported. Arrays of any scalar type other than byte should be initialized using braces. Arrays of struct and nested arrays cannot be initialized.

## 2.3. The Preprocessor

The NBC preprocessor implements the following directives: `#include`, `#define`, `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif`, `#endif`, `#undef`, `##`. 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 NBC. Significant deviations are described below.

### 2.3.1. #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!
```

NBC programs can begin with `#include "NXTDefs.h"`. This standard header file includes many important constants and macros which form the core NBC API. Current versions of NBC no longer require that you manually include the NXTDefs.h header file. Unless you specifically tell the compiler to ignore the standard system files this header file will automatically be included for you.

### 2.3.2. #define

The `#define` command is used for simple macro substitution. Redefinition of a macro is an error (unlike in C where it is a warning). Use #define to define your own constants for use throughout the program.

```
#define TurnTime 3000 ; 3 seconds
```

Macros are normally terminated by the end of the line, but the newline may be escaped with the backslash ('\') to allow multi-line macros (as described in the *Macro Functions* section above):

```
#define square(x, result) \
  mul result, x, x
```

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

### 2.3.3. ## (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 parameters values.

### 2.3.4. Conditional Compilation

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

```
#ifdef symbol
#ifndef symbol
#if defined(expr)
#else
#elif
#endif
```

Conditions in #if directives use the same operators and precedence as in C. The defined() operator is supported.

## 2.4. Compiler Tokens

NBC supports special tokens which it replaces on compilation. The tokens are similar to preprocessor `#define` macros but they are actually handled directly by the compiler rather than the preprocessor. The supported tokens are as follows:

| Token | Usage |
|---|---|
| __FILE__ | This token is replaced with the currently active filename (no path) |

| | |
|---|---|
| __LINE__ | This token is replaced with the current line number |
| __VER__ | This token is replaced with the compiler version number |
| __THREADNAME__ | This token is replaced with the current thread name |
| __I__, __J__ | These tokens are replaced with the current value of I or J. They are both initialized to zero at the start of each thread or subroutine. |
| __ResetI__, __ResetJ__ | These tokens are replaced with nothing. As a side effect the value of I or J is reset to zero. |
| __IncI__, __IncJ__ | These tokens are replaced with nothing. As a side effect the value of I or J is incremented by one. |
| __DecI__, __DecJ__ | These tokens are replaced with nothing. As a side effect the value of I or J is decremented by one. |

**Table 2. Compiler Tokens**

The ## preprocessor directive can help make the use of compiler tokens more readable. __THREADNAME__##_##__I__: would become something like main_1:. Without the ## directive it would much harder to read the mixture of compiler tokens and underscores.

## *2.5. Expression Evaluator*

Constant expressions are supported by NBC for many statement arguments as well as variable initialization. Expressions are evaluated by the compiler when the program is compiled, not at run time. The compiler will return an error if it encounters an expression that contains whitespace. "4+4" is a valid constant expression but "4 + 4" is not.

The expression evaluator supports the following operators:

| Operator | Meaning |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| ^ | exponent |
| % | modulo (remainder) |
| & | bitwise and |
| \| | bitwise or |
| ~ | bitwise xor |
| << | shift left |
| >> | shift right |
| () | grouping subexpressions |
| PI | constant value |

**Table 3. Constant Expression Operators**

The expression evaluator also supports the following compile-time functions:

```
tan(x), sin(x), cos(x)
sinh(x), cosh(x)
arctan(x), cotan(x)
arg(x)
exp(x), ln(x), log10(x), log2(x), logn(x, n)
sqr(x), sqrt(x)
trunc(x), int(x), ceil(x), floor(x), heav(x)
abs(x), sign(x), zero(x), ph(x)
rnd(x), random(x)
max(x, y), min(x, y)
power(x, exp), intpower(x, exp)
```

**Table 4. Constant Expression Functions**

The following example demonstrates how to use a constant expression:

```
// expression value will be truncated to an integer
set val, 3+(PI*2)-sqrt(30)
```

## 2.6. IO-Map Address (IOMA) Constants

IOMA constants provide a simplified means for accessing input and output field values without having to use a variable or an input or output statement. The constants are defined in the NBCCommon.h header file which is included automatically in your NBC program.

There are IOMA constants for inputs and IOMA constants for outputs. They are defined as preprocessor macros. To specify the port for each IOMA you must use a constant such as IN_1 or OUT_A. You can often substitute an IOMA constant in statements which can accept a scalar variable argument. The IOMA macros are shown below.

| IO-Map Address Macros |
|---|
| InputIOType(port) |
| InputIOInputMode(port) |
| InputIORawValue(port) |
| InputIONormalizedValue(port) |
| InputIOScaledValue(port) |
| InputIOInvalidData(port) |
| OutputIOUpdateFlags(port) |
| OutputIOOutputMode(port) |
| OutputIOPower(port) |
| OutputIOActualSpeed(port) |
| OutputIOTachoCount(port) |
| OutputIOTachoLimit(port) |
| OutputIORunState(port) |
| OutputIOTurnRatio(port) |
| OutputIORegMode(port) |
| OutputIOOverload(port) |
| OutputIORegPValue(port) |
| OutputIORegIValue(port) |
| OutputIORegDValue(port) |

| |
|---|
| OutputIOBlockTachoCount(port) |
| OutputIORotationCount(port) |

**Table 5. IOMA Constant Macros**

## *2.7. Statements*

The body of a code block (thread, subroutine, or macro function) is composed of statements. All statements are terminated with the newline character.

### 2.7.1. Assignment Statements

Assignment statements enable you to copy values from one variable to another or to simply set the value of a variable. In NBC there are two ways to assign a new value to a variable.

The `mov` statement assigns the value of its second argument to its first argument. The first argument must be the name of a variable. It can be of any valid variable type except mutex. The second argument can be a variable or a numeric or string constant. If a constant is used, the compiler creates a variable behind the scenes and initializes it to the specified constant value.

Both arguments to the `mov` statement must be of compatible types. A scalar value can be assigned to another scalar variable, regardless of type, structs can be assigned to struct variables if the structure types are the same, and arrays can be assigned to an array variable provided that the type contained in the arrays are the same. The syntax of the `mov` statement is shown below.

```
mov x, y     // set x equal to y
```

The `set` statement also assigns its first argument to have the value of its second argument. The first argument must be the name of a variable. It must be a scalar type. The second argument must be a numeric constant or constant expression. The syntax of the `set` statement is shown below.

```
set x, 10     // set x equal to 10
```

Because all arguments must fit into a 2-byte value in the NXT executable, the second argument of the `set` statement is limited to a 16 bit signed or unsigned value (-32768..65535).

### 2.7.2. Math Statements

Math statements enable you to perform basic math operations on data in your NBC programs. Unlike high level programming languages where mathematical expressions use standard math operators (such as *, -, +, /), in NBC, as with other assembly languages, math operations are expressed as statements with the math operation name coming first, followed by the arguments to the operation. All statements in this family have one output argument and two input arguments except the negate statement, the absolute value statement, and the sign statement.

Math statements in NBC differ from traditional assembly math statements because many of the operations can handle arguments of scalar, array, and struct types rather than only scalar types. If, for example, you multiply an array by a scalar then each of the elements in the resulting array will be the corresponding element in the original array multiplied by the scalar value.

Only the absolute value and sign statements require that their arguments are scalar types. When using the standard NXT firmware these two statements are currently implemented by the compiler since it does not have built-in support for them. If you install the enhanced NBC/NXC firmware and tell the compiler to target it using the –EF command line switch then these statements will be handled directly by the firmware itself rather than by the compiler.

The `add` statement lets you add two input values together and store the result in the first argument. The first argument must be a variable but the second and third arguments can be variables, numeric constants, or constant expressions. The syntax of the `add` statement is shown below.

```
add x, x, y ; add x and y and store result in x
```

The `sub` statement lets you subtract two input values and store the result in the first argument. The first argument must be a variable but the second and third arguments can be variables, numeric constants, or constant expressions. The syntax of the `sub` statement is shown below.

```
sub x, x, y ; subtract y from x and store result in x
```

The `mul` statement lets you multiply two input values and store the result in the first argument. The first argument must be a variable but the second and third arguments can be variables, numeric constants, or constant expressions. The syntax of the `mul` statement is shown below.

```
mul x, x, x ; set x equal to x^2
```

The `div` statement lets you divide two input values and store the result in the first argument. The first argument must be a variable but the second and third arguments can be variables, numeric constants, or constant expressions. The syntax of the `div` statement is shown below.

```
div x, x, 2 ; set x equal to x / 2 (integer division)
```

The `mod` statement lets you calculate the modulus value (or remainder) of two input values and store the result in the first argument. The first argument must be a variable but the second and third arguments can be variables, numeric constants, or constant expressions. The syntax of the `mod` statement is shown below.

```
mod x, x, 4 ; set x equal to x % 4 (0..3)
```

The `neg` statement lets you negate an input value and store the result in the first argument. The first argument must be a variable but the second argument can be a variable, a numeric constant, or a constant expression. The syntax of the `neg` statement is shown below.

```
neg x, y ; set x equal to -y
```

The `abs` statement lets you take the absolute value of an input value and store the result in the first argument. The first argument must be a variable but the second argument can be a variable, a numeric constant, or a constant expression. The syntax of the `abs` statement is shown below.

```
abs x, y ; set x equal to the absolute value of y
```

The `sign` statement lets you take the sign value (-1, 0, or 1) of an input value and store the result in the first argument. The first argument must be a variable but the second argument can be a variable, a numeric constant, or a constant expression. The syntax of the `abs` statement is shown below.

```
sign x, y ; set x equal to -1, 0, or 1
```

### 2.7.3. Logic Statements

Logic statements let you perform basic logical operations on data in your NBC program. As with the math statements, the logical operation name begins the statement and it is followed by the arguments to the logical operation. All the statements in this family have one output argument and two input arguments except the logical not statement. Each statement supports arguments of any type, scalar, array, or struct.

The `and` statement lets you bitwise and together two input values and store the result in the first argument. The first argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `and` statement is shown below.

```
and x, x, y  // x = x & y
```

The `or` statement lets you bitwise or together two input values and store the result in the first argument. The first argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `or` statement is shown below.

```
or x, x, y  // x = x | y
```

The `xor` statement lets you bitwise exclusive or together two input values and store the result in the first argument. The first argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `xor` statement is shown below.

```
xor x, x, y  // x = x ^ y
```

The `not` statement lets you logically not its input value and store the result in the first argument. The first argument must be a variable but the second argument can be a variable, a numeric constant, or a constant expression. The syntax of the `not` statement is shown below.

```
not x, x  // x = !x (logical not – not bitwise)
```

### 2.7.4. Bit Manipulation Statements

Bit manipulation statements enable you to perform basic bitwise operations on data in your NBC programs. All statements in this family have one output argument and two input arguments except the complement statement.

Using the standard NXT firmware the basic shift right and shift left statements (shr and shl) are implemented by the compiler since the firmware does not support shift operations at this time. If you install the enhanced NBC/NBC firmware and tell the compiler to target it using the –EF command line switch, then these operations will be handled directly by the firmware itself rather than by the compiler. The other bit manipulation statements described in this section are only available when targeting the enhanced firmware.

The `shr` statement lets you shift right an input value by the number of bits specified by the second input argument and store the resulting value in the output argument. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `shr` statement is shown below.

```
shr x, x, y  // x = x >> y
```

The `shl` statement lets you shift left an input value by the number of bits specified by the second input argument and store the resulting value in the output argument. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `shl` statement is shown below.

```
shl x, x, y  // x = x << y
```

The `asr` statement lets you perform an arithmetic right shift operation. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `asr` statement is shown below.

```
asr x, x, y  // x = x >> y
```

The `asl` statement lets you perform an arithmetic left shift operation. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `asl` statement is shown below.

```
asl x, x, y  // x = x << y
```

The `lsr` statement lets you perform a logical right shift operation. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `lsr` statement is shown below.

```
lsr x, x, y
```

The `lsl` statement lets you perform a logical left shift operation. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `lsl` statement is shown below.

```
lsl x, x, y
```

The `rotr` statement lets you perform a rotate right operation. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `rotr` statement is shown below.

```
rotr x, x, y
```

The `rotl` statement lets you perform a rotate left operation. The output (first) argument must be a variable but the second and third arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `rotl` statement is shown below.

```
rotl x, x, y
```

The `cmnt` statement lets you perform a bitwise complement operation. The output (first) argument must be a variable but the second can be a variable, a numeric constant, or a constant expression. The syntax of the `cmnt` statement is shown below.

```
cmnt x, y // x = ~y
```

### 2.7.5. Comparison Statements

Comparison statements enable you to compare data in your NBC programs. These statements take a comparison code constant as their first argument. Valid comparison constants are listed in the table below. You can use scalar, array, and aggregate types for the compare or test argument(s).

| Comparison | Constant | Value | Alternative Token |
|---|---|---|---|
| Less than | LT | 0x00 | < |
| Greater than | GT | 0x01 | > |
| Less than or equal | LTEQ | 0x02 | <= |
| Greater than or equal | GTEQ | 0x03 | >= |
| Equal | EQ | 0x04 | == |
| Not equal | NEQ | 0x05 | != or <> |

**Table 6. Comparison Constants**

The `cmp` statement lets you compare two different input sources. The output (second) argument must be a variable but the remaining arguments can be a variable, a numeric constant, or a constant expression. The syntax of the `cmp` statement is shown below.

```
cmp EQ, bXEqualsY, x, y // bXEqualsY = (x == y);
```

The `tst` statement lets you compare an input source to zero. The output (second) argument must be a variable but the remaining argument can be a variable, a numeric constant, or a constant expression. The syntax of the `tst` statement is shown below.

```
tst GT, bXGTZero, x // bXGTZero = (x > 0);
```

### 2.7.6. Control Flow Statements

Control flow statements enable you to manipulate or control the execution flow of your NBC programs. Some of these statements take a comparison code constant as their first argument. Valid comparison constants are listed in Table 6 above. You can use scalar, array, and aggregate types for the compare or test argument(s).

The `jmp` statement lets you unconditionally jump from the current execution point to a new location. Its only argument is a label that specifies where program execution should resume. The syntax of the `jmp` statement is shown below.

```
jmp LoopStart // jump to the LoopStart label
```

The `brcmp` statement lets you conditionally jump from the current execution point to a new location. It is like the `cmp` statement except that instead of an output argument it has a label argument that specifies where program execution should resume. The syntax of the `brcmp` statement is shown below.

```
brcmp EQ, LoopStart, x, y // jump to LoopStart if x == y
```

The `brtst` statement lets you conditionally jump from the current execution point to a new location. It is like the `tst` statement except that instead of an output argument it has a label argument that specifies where program execution should resume. The syntax of the `brtst` statement is shown below.

```
brtst GT, lblXGTZero, x // jump to lblXGTZero if x > 0
```

The `stop` statement lets you stop program execution completely, depending on the value of its boolean input argument. The syntax of the `stop` statement is shown below.

```
stop bProgShouldStop // stop program if flag <> 0
```

### 2.7.7. System Call Statements

The `syscall` statement enables execution of various system functions via a constant function ID and an aggregate type variable for passing arguments to and from the system function. The syntax of the `syscall` statement is shown below.

```
// ptArgs is a struct with input and output args
syscall SoundPlayTone, ptArgs
```

| Function ID | Value |
|---|---|
| FileOpenRead | 0 |
| FileOpenWrite | 1 |
| FileOpenAppend | 2 |
| FileRead | 3 |
| FileWrite | 4 |
| FileClose | 5 |
| FileResolveHandle | 6 |
| FileRename | 7 |
| FileDelete | 8 |
| SoundPlayFile | 9 |
| SoundPlayTone | 10 |
| SoundGetState | 11 |
| SoundSetState | 12 |
| DrawText | 13 |
| DrawPoint | 14 |
| DrawLine | 15 |
| DrawCircle | 16 |
| DrawRect | 17 |
| DrawGraphic | 18 |
| SetScreenMode | 19 |
| ReadButton | 20 |
| CommLSWrite | 21 |

| | |
|---|---|
| CommLSRead | 22 |
| CommLSCheckStatus | 23 |
| RandomNumber | 24 |
| GetStartTick | 25 |
| MessageWrite | 26 |
| MessageRead | 27 |
| CommBTCheckStatus | 28 |
| CommBTWrite | 29 |
| KeepAlive | 31 |
| IOMapRead | 32 |
| IOMapWrite | 33 |
| IOMapReadByID | 34 |
| IOMapWriteByID | 35 |
| DisplayExecuteFunction | 36 |
| CommExecuteFunction | 37 |
| LoaderExecuteFunction | 38 |

**Table 7. System Call Function IDs**

### 2.7.8. Timing Statements

Timing statements enable you to pause the execution of a thread or obtain information about the system tick counter in your NBC programs. When using the standard NXT firmware NBC implements the wait and waitv statements as thread-specific subroutine calls due to them not being implemented. The enhanced NBC/NXC firmware implements these statements natively. If needed, you can implement simple wait loops using gettick.

```
add endTick, currTick, waitms
Loop:
  gettick currTick
  brcmp LT, Loop, currTick, endTick
```

The `wait` statement suspends the current thread for the number of milliseconds specified by its constant argument. The syntax of the `wait` statement is shown below.

```
wait 1000 // wait for 1 second
```

The `waitv` statement acts like `wait` but it takes a variable argument. If you use a constant argument with `waitv` the compiler will generate a temporary variable for you. The syntax of the `waitv` statement is shown below.

```
waitv iDelay // wait for the number of milliseconds in iDelay
```

The `gettick` statement suspends the current thread for the number of milliseconds specified by its constant argument. The syntax of the `gettick` statement is shown below.

```
gettick x // set x to the current system tick count
```

### 2.7.9. Array Statements

Array statements enable you to populate and manipulate arrays in your NBC programs.

The `index` statement extracts a single element from the source array and returns the value in the output (first) argument. The last argument is the index of the desired element. The syntax of the `index` statement is shown below.

```
// extract arrayValues[index] and store it in value
index value, arrayValues, index
```

The `replace` statement replaces one or more items in a source array and stores the modified array contents in an output array. The array source argument (second) can be the same variable as the array destination (first) argument to replace without copying the array. The index of the element(s) to be replaced is specified via the third argument. The new value (last) argument can be an array, in which case multiple items are replaced. The syntax of the `replace` statement is shown below.

```
// replace arValues[idx] with x in arNew (arValues is unchanged)
replace arNew, arValues, idx, x
```

The `arrsize` statement returns the number of elements in the input array (second) argument in the scalar output (first) argument. The syntax of the `arrsize` statement is shown below.

```
arrsize nSize, arValues  // nSize == length of array
```

The `arrinit` statement initializes the output array (first) argument using the value (second) and size (third) arguments provided. The syntax of the `arrinit` statement is shown below.

```
// initialize arValues with nSize zeros
arrinit arValues, 0, nSize
```

The `arrsubset` statement copies a subset of the input array (second) argument to the output array (first) argument. The subset begins at the specified index (third) argument. The number of elements in the subset is specified using the length (fourth) argument. The syntax of the `arrsubset` statement is shown below.

```
// copy the first x elements to arSub
arrsubset arSub, arValues, NA, x
```

The `arrbuild` statement constructs an output array from a variable number of input arrays, scalars, or aggregates. The types of all the input arguments must be compatible with the type of the output array (first) argument. You must provide one or more comma-separated input arguments. The syntax of the `arrbuild` statement is shown below.

```
// build data array from 3 sources
arrbuild arData, arStart, arBody, arEnd
```

### 2.7.10. String Statements

String statements enable you to populate and manipulate null-terminated byte arrays (aka strings) in your NBC programs.

The `flatten` statement converts its input (second) argument into its string output (first) argument. The syntax of the `flatten` statement is shown below.

```
flatten strData, args  // copy args structure to strData
```

The `unflatten` statement converts its input string (third) argument to the output (first) argument type. If the default value (fourth) argument type does not match the flattened data type exactly, including array sizes, then error output (second) argument will be set to TRUE and the output argument will contain a copy of the default argument. The syntax of the `unflatten` statement is shown below.

```
unflatten args, bErr, strSource, x  // convert string to cluster
```

The `numtostr` statement converts its scalar input (second) argument to a string output (first) argument. The syntax of the `numtostr` statement is shown below.

```
numtostr strValue, value  // convert value to a string
```

The `fmtnum` statement converts its scalar input (third) argument to a string output (first) argument. The format of the string output is specified via the format string (second) argument. The syntax of the `fmtnum` statement is shown below.

```
fmtnum strValue, fmtStr, value  // convert value to a string
```

The `strtonum` statement parses its input string (third) argument into a numeric output (first) argument, advancing an offset output (second) argument past the numeric string. The initial input offset (fourth) argument determines where the string parsing begins. The default (fifth) argument is the value that is returned by the statement if an error occurs while parsing the string. The syntax of the `strtonum` statement is shown below.

```
// parse string into num
strtonum value, idx, strValue, idx, nZero
```

The `strsubset` statement copies a subset of the input string (second) argument to the output string (first) argument. The subset begins at the specified index (third) argument. The number of characters in the subset is specified using the length (fourth) argument. The syntax of the `strsubset` statement is shown below.

```
// copy the first x characters in strSource to strSub
strsubset strSub, strSource, NA, x
```

The `strcat` statement constructs an output string from a variable number of input strings. The input arguments must all be null-terminated byte arrays. You must provide one or more comma-separated input arguments. The syntax of the `strcat` statement is shown below.

```
// build data string from 3 sources
strcat strData, strStart, strBody, strEnd
```

The `arrtostr` statement copies the input byte array (second) argument into its output string (first) argument and adds a null-terminator byte at the end. The syntax of the `arrtostr` statement is shown below.

```
arrtostr strData, arrData  // convert byte array to string
```

The `strtoarr` statement copies the input string (second) argument into its output byte array (first) argument excluding the last byte, which should be a null. The syntax of the `strtoarr` statement is shown below.

```
strtoarr arrData, strData  // convert string to byte array
```

The `strindex` statement extracts a single element from the source string and returns the value in the output (first) argument. The last argument is the index of the desired element. The syntax of the `strindex` statement is shown below.

```
// extract strVal[idx] and store it in val
strindex val, strVal, idx
```

The `strreplace` statement replaces one or more characters in a source string and stores the modified string in an output string. The string source argument (second) can be the same variable as the string destination (first) argument to replace without copying the string. The index of the character(s) to be replaced is specified via the third argument. The new value (fourth) argument can be a string, in which case multiple characters are replaced. The syntax of the `strreplace` statement is shown below.

```
// replace strValues[idx] with newStr in strNew
strreplace strNew, strValues, idx, newStr
```

The `strlen` statement returns the length of the input string (second) argument in the scalar output (first) argument. The syntax of the `strlen` statement is shown below.

```
strlen nSize, strMsg  // nSize == length of strMsg
```

### 2.7.11. Scheduling Statements

Scheduling statements enable you to control the execution of multiple threads and the calling of subroutines in your NBC programs.

The `exit` statement finalizes the current thread and schedules zero or more dependant threads by specifying start and end dependency list indices. The thread indices are zero-based and inclusive. The two arguments are optional, in which case the compiler automatically adds indices for all the dependencies. The syntax of the `exit` statement is shown below.

```
exit 0, 2  // schedule this thread's 3 dependants
exit // schedule all this thread's dependants
```

The `exitto` statement exits the current thread and schedules the specified thread to begin executing. The syntax of the `exitto` statement is shown below.

```
exitto worker  // exit now and schedule worker thread
```

The `start` statement causes the thread specified in the statement to start running immediately. Using the standard NXT firmware this statement is implemented by the compiler using a set of compiler-generated subroutines. The enhanced NBC/NXC firmware implements this statement natively. The syntax of the `start` statement is shown below.

```
start worker  // start the worker thread
```

The `stopthread` statement causes the thread specified in the statement to stop running immediately. This statement cannot be used with the standard NXT firmware. It is supported by the enhanced NBC/NXC firmware. The syntax of the `stopthread` statement is shown below.

```
stopthread worker  // stop the worker thread
```

The `priority` statement modifies the priority of the thread specified in the statement. This statement cannot be used with the standard NXT firmware. It is supported by the enhanced NBC/NXC firmware. The syntax of the `priority` statement is shown below.

```
priority worker, 50  // change the priority of the worker thread
```

The `precedes` statement causes the compiler to mark the threads listed in the statement as dependants of the current thread. A subset of these threads will begin executing once the current thread exits, depending on the form of the exit statement used at the end of the current thread. The syntax of the `precedes` statement is shown below.

```
precedes worker, music, walking  // configure dependant threads
```

The `follows` statement causes the compiler to mark the current thread as a dependant of the threads listed in the statement. The current thread will be scheduled to execute if all of the threads that precede it have exited and scheduled it for execution. The syntax of the `follows` statement is shown below.

```
follows main  // configure thread dependencies
```

The `acquire` statement acquires the named mutex.  If the mutex is already acquired the current thread waits until it becomes available. The syntax of the `acquire` statement is shown below.

```
acquire muFoo  // acquire mutex for subroutine
```

The `release` statement releases the named mutex allowing other threads to acquire it. The syntax of the `release` statement is shown below.

```
release muFoo  // release mutex for subroutine
```

The `subcall` statement calls into the named thread/subroutine and waits for a return (which might not come from the same thread). The second argument is a variable used to store the return address. The syntax of the `subcall` statement is shown below.

```
subcall drawText, retDrawText  // call drawText subroutine
```

The `subret` statement returns from a thread to the return address value contained in its input argument. The syntax of the `subret` statement is shown below.

```
subret retDrawText  // return to calling routine
```

The `call` statement executes the named subroutine and waits for a return. The argument should specify a thread that was declared using the subroutine keyword. The syntax of the `call` statement is shown below.

```
call MyFavoriteSubroutine  // call routine
```

The `return` statement returns from a subroutine. The compiler automatically handles the return address for call and return when they are used with subroutines rather than threads. The syntax of the `return` statement is shown below.

```
return  // return to calling routine
```

### 2.7.12. Input Statements

Input statements enable you to configure the four input ports and read analog sensor values in your NBC programs. Both statements in this category use input field identifiers to control which attribute of the input port you are manipulating. Valid input field identifiers are listed in the following table.

| Input Field ID | Value |
|---|---|
| Type | 0 |
| InputMode | 1 |
| RawValue | 2 |
| NormalizedValue | 3 |
| ScaledValue | 4 |
| InvalidData | 5 |

**Table 8. Input Field IDs**

The `setin` statement sets an input field of a sensor on a port to the value specified in its first argument. The port is specified via the second argument. The input field identifier is the third argument. The syntax of the `setin` statement is shown below.

```
setin IN_TYPE_SWITCH, IN_1, Type ; set sensor to switch type
setin IN_MODE_BOOLEAN, IN_1, InputMode ; set to boolean mode
```

The `getin` statement reads a value from an input field of a sensor on a port and writes the value to its first argument. The port is specified via the second argument. The input field identifier is the third argument. The syntax of the `getin` statement is shown below.

```
getin rVal, thePort, RawValue  // read raw sensor value
getin sVal, thePort, ScaledValue  // read scaled sensor value
getin nVal, thePort, NormalizedValue  // read normalized value
```

### 2.7.13. Output Statements

Output statements enable you to configure and control the three NXT outputs in your NBC programs. Both statements in this category use output field identifiers to control which attribute of the output you are manipulating. Valid output field identifiers are listed in the following table.

| Output Field ID | Value |
|---|---|
| UpdateFlags | 0 |

| | |
|---|---|
| OutputMode | 1 |
| Power | 2 |
| ActualSpeed | 3 |
| TachoCount | 4 |
| TachoLimit | 5 |
| RunState | 6 |
| TurnRatio | 7 |
| RegMode | 8 |
| Overload | 9 |
| RegPValue | 10 |
| RegIValue | 11 |
| RegDValue | 12 |
| BlockTachoCount | 13 |
| RotationCount | 14 |

**Table 9. Output Field IDs**

The `setout` statement sets one or more output fields of a motor on one or more ports to the value specified by the coupled input arguments. The first argument is either a scalar value specifying a single port or a byte array specifying multiple ports. After the port argument you then provide one or more pairs of output field identifiers and values. You can set multiple fields via a single statement. The syntax of the `setout` statement is shown below.

```
set theMode, OUT_MODE_MOTORON  // set mode to motor on
set rsVal, OUT_RUNSTATE_RUNNING // motor running
set thePort, OUT_A  // set port to #1
set pwr, -75 // negative power means reverse motor direction
// set output values
setout thePort, OutputMode, theMode, RunState, rsVal, Power, pwr
```

The `getout` statement reads a value from an output field of a sensor on a port and writes the value to its first output argument. The port is specified via the second argument. The output field identifier is the third argument. The syntax of the `getout` statement is shown below.

```
getout rmVal, thePort, RegMode  // read motor regulation mode
getout tlVal, thePort, TachoLimit  // read tachometer limit value
getout rcVal, thePort, RotationCount // read the rotation count
```

### 2.7.14. Compile-time Statements

Compile-time statements and functions enable you to perform simple compiler operations at the time you compile your NBC programs.

The `sizeof(arg)` compiler function returns the size of the variable you pass into it. The syntax of the `sizeof` function is shown below.

```
dseg segment
  arg byte
  argsize byte
dseg ends
// ...
```

```
set argsize, sizeof(arg) ; argsize == 1
```

The `valueof(arg)` compiler function returns the value of the constant expression you pass into it. The syntax of the `valueof` function is shown below.

```
set argval, valueof(4+3*2) ; argval == 10
```

The `isconst(arg)` compiler function returns TRUE if the argument you pass into it is a constant and FALSE if it is not a constant. The syntax of the `isconst` function is shown below.

```
set argval, isconst(4+3*2) ; argval == TRUE
```

The `compchk` compiler statement takes a comparison constant as its first argument. The second and third arguments must be constants or constant expressions that can be evaluated by the compiler during program compilation. It reports a compiler error if the comparison expression does not evaluate to TRUE. Valid comparison constants are listed in Table 6. The syntax of the `compchk` statement is shown below.

```
compchk EQ, sizeof(arg3), 2
```

The `compif`, `compelse`, and `compend` compiler statements work together to create a compile-time if-else statement that enables you to control whether or not sections of code should be included in the compiler output. The `compif` statement takes a comparison constant as its first argument. The second and third arguments must be constants or constant expressions that can be evaluated by the compiler during program compilation. If the comparison expression is true then code immediate following the statement will be included in the executable. The compiler if statement ends when the compiler finds the next `compend` statement. To optionally provide an else clause use the `compelse` statement between the `compif` and `compend` statements. Valid comparison constants are listed in Table 6. The syntax of the `compif` , `compelse`, and `compend` statements is shown below.

```
compif EQ, sizeof(arg3), 2
  // compile this if sizeof(arg3) == 2
compelse
  // compile this if sizeof(arg3) != 2
compend
```

## 3. NBC API

The NBC API defines a set of constants and macros that provide access to various capabilities of the NXT such as sensors, outputs, and communication. The API consists of macro functions and constants. A function is something that can be called as a statement. Typically it takes some action or configures some parameter. 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
```

**GetFirstTick(out result)** **Function**

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.

```
GetFirstTick(x)
```

**GetSleepTime(out result)** **Function**

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

```
GetSleepTime(sleepy)
```

**GetSleepTimer(out result)** **Function**

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

```
GetSleepTimer(stime)
```

**ResetSleepTimer** **Function**

Reset the system sleep timer back to the SleepTime 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. Numeric Functions

**Random(out result, Max)** **Function**

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

```
Random(x, 10) // return a value of 0..9
```

### SignedRandom(out result) Function

Return a signed 16-bit random number.

```
SignedRandom(x)
```

### Sqrt(x, out result) Function

Return the square root of the specified value.

```
Sqrt(x, x) // x = sqrt(x)
```

### Sin(degrees, out result) Function

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

```
Sin(theta, x) // x = sin(theta)*100
```

### Cos(degrees, out result) Function

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

```
Cos(y, x) // x = cos(y)*100
```

### Asin(value, out result) Function

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

```
Asin(80, deg) // deg = asin(0.80)
```

### Acos(value, out result) Function

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

```
Acos(0, deg) // deg = acos(0.00)
```

### bcd2dec(bcdValue, out result) Function

Return the decimal equivalent of the binary coded decimal value provided.

```
bcd2dec(0x3a, dec)
```

## 3.1.3. Low-level System Functions

There are several standard structures that are defined by the NBC 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 NBC 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/NBC 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/NBC 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.

```
Tlocation struct
  X sword
  Y sword
TLocation ends
```

```
TSize struct
  Width sword
  Height sword
TSize struct
```

### syscall DrawText, args **Function**

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

```
TDrawText struct
  Result sbyte
  Location TLocation
  Text byte[]
  Options dword
TDrawText ends
```

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

```
dseg segment
  dtArgs TDrawText
dseg ends
set dtArgs.Location.X, 0
set dtArgs.Location.Y, LCD_LINE1
mov dtArgs.Text, 'Please Work'
set dtArgs.Options, 0x01 // clear before drawing
syscall DrawText, dtArgs
```

### syscall DrawPoint, args **Function**

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

```
TDrawPoint struct
  Result sbyte
  Location TLocation
  Options dword
TDrawPoint ends
```

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

```
dseg segment
  dpArgs TDrawPoint
dseg ends
set dpArgs.Location.X, 0
set dpArgs.Location.Y, 20
set dpArgs.Options, 0x04 // clear this pixel
syscall DrawPoint, dpArgs
```

### syscall DrawLine, args — Function

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

```
TDrawLine struct
  Result sbyte
  StartLoc TLocation
  EndLoc TLocation
  Options dword
TDrawLine ends
```

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

```
dseg segment
  dlArgs TDrawLine
dseg ends
set dlArgs.StartLoc.X, 20
set dlArgs.StartLoc.Y, 20
set dlArgs.EndLoc.X, 60
set dlArgs.EndLoc.Y, 60
set dlArgs.Options, 0x01 // clear before drawing
syscall DrawLine, dlArgs
```

### syscall DrawCircle, args — Function

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

```
TDrawCircle struct
  Result sbyte
  Center TLocation
  Size byte
  Options dword
TDrawCircle ends
```

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

```
dseg segment
  dcArgs TDrawCircle
dseg ends
set dcArgs.Center.X, 20
set dcArgs.Center.Y, 20
set dcArgs.Size, 10 // radius
```

```
set dcArgs.Options, 0x01 // clear before drawing
syscall DrawCircle, dcArgs
```

### syscall DrawRect, args — Function

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

```
TDrawRect struct
  Result sbyte
  Location TLocation
  Size TSize
  Options dword
TDrawRect ends
```

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

```
dseg segment
  drArgs TDrawRect
dseg ends
set drArgs.Location.X, 20
set drArgs.Location.Y, 20
set drArgs.Size.Width, 20
set drArgs.Size.Height, 10
set drArgs.Options, 0x00 // do not clear before drawing
syscall DrawRect, drArgs
```

### syscall DrawGraphic, args — Function

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

```
TDrawGraphic struct
  Result sbyte
  Location TLocation
  Filename byte[]
  Variables sword[]
  Options dword
TDrawGraphic ends
```

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

```
dseg segment
  dgArgs TDrawGraphic
dseg ends
set dgArgs.Location.X, 20
set dgArgs.Location.Y, 20
mov dgArgs.Filename, 'image.ric'
arrinit dgArgs.Variables, 0, 10 // 10 zeros
replace dgArgs.Variables, dgArgs.Variables, 0, 12
replace dgArgs.Variables, dgArgs.Variables, 1, 14
```

```
set dgArgs.Options, 0x00 // do not clear before drawing
syscall DrawGraphic, dgArgs
```

### syscall SetScreenMode, args — Function

This function lets you set the screen mode of the NXT LCD given the parameters you pass in via the TSetScreenMode 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/NBC 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.

```
TSetScreenMode struct
  Result sbyte;
  ScreenMode dword;
TSetScreenMode ends
```

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

```
dseg segment
  ssmArgs TSetScreenMode
dseg ends
set ssmArgs.ScreenMode, 0x00 // restore default NXT screen
syscall SetScreenMode, ssmArgs
```

### syscall SoundPlayFile, args — Function

This function lets you play a sound file given the parameters you pass in via the TSoundPlayFile 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.

```
TSoundPlayFile struct
  Result sbyte
  Filename byte[]
  Loop byte
  SoundLevel byte
TSoundPlayFile ends
```

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

```
dseg segment
  spfArgs TSoundPlayFile
dseg ends
mov spfArgs.Filename, 'hello.rso'
set spfArgs.Loop, FALSE
set spfArgs.SoundLevel, 3
syscall SoundPlayFile, spfArgs
```

### syscall SoundPlayTone, args — Function

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

```
TSoundPlayTone struct
  Result sbyte
  Frequency word
  Duration word
  Loop byte
  SoundLevel byte
TSoundPlayTone ends
```

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

```
dseg segment
  sptArgs TSoundPlayTone
dseg ends
set sptArgs.Frequency, 440
set sptArgs.Duration, 1000 // 1 second
set sptArgs.Loop, false
set sptArgs.SoundLevel, 3
syscall SoundPlayTone, sptArgs
```

### syscall SoundGetState, args — Function

This function lets you retrieve information about the sound module state via the TSoundGetState 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.

```
TSoundGetStateType struct
  State byte
  Flags byte
TSoundGetStateType ends
```

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

```
dseg segment
  sgsArgs TSoundGetState
dseg ends
syscall SoundGetState, sgsArgs
brcmp NEQ, lblEndIf, sgsArgs.State, SOUND_STATE_IDLE
  // do stuff
lblEndIf:
```

### syscall SoundSetState, args — Function

This function lets you set sound module state settings via the TSoundSetState 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.

```
TSoundSetState struct
  Result byte
```

```
  State byte
  Flags byte
TSoundSetState ends
```

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

```
dseg segment
  sssArgs TSoundSetState
dseg ends
set sssArgs.State, SOUND_STATE_STOP
syscall SoundSetState, sssArgs
```

### syscall ReadButton, args — Function

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

```
TReadButton struct
  Result sbyte
  Index byte
  Pressed byte
  Count byte
  Reset byte // reset count after reading?
TReadButton ends
```

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

```
dseg segment
  rbArgs TReadButton
dseg ends
set rbArgs.Index, BTNRIGHT
syscall ReadButton, rbArgs
brtst EQ, lblEndIf, rbArgs.Pressed
  // do stuff
lblEndIf:
```

### syscall RandomNumber, args — Function

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

```
TRandomNumber struct
  Result sword
TRandomNumber ends
```

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

```
dseg segment
  rnArgs TRandomNumber
  myRandomNumber sword
dseg ends
syscall RandomNumber, rnArgs
mov myRandomValue, rnArgs.Result
```

### syscall GetStartTick, args **Function**

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

```
TGetStartTick struct
  Result dword
TGetStartTick ends
```

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

```
dseg segment
  gstArgs TGetStartTick
  myStart dword
dseg ends
syscall GetStartTick, gstArgs
mov myStart, gstArgs.Result
```

### syscall KeepAlive, args **Function**

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

```
TKeepAlive struct
  Result dword
TKeepAlive ends
```

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

```
dseg segment
  kaArgs TKeepAlive
dseg ends
syscall KeepAlive, kaArgs // reset sleep timer
```

### syscall FileOpenWrite, args **Function**

This function lets you create a file that you can write to using the values specified via the TFileOpen 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.

```
TFileOpen struct
  Result dword
  FileHandle byte
  Filename byte[]
  Length dword
TFileOpen ends
```

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

```
dseg segment
  foArgs TFileOpen
dseg ends
```

```
mov foArgs.Filename, 'myfile.txt'
set foArgs.Length, 256 // create with capacity for 256 bytes
syscall FileOpenWrite, foArgs // create the file
brcmp NEQ, lblEndIf, foArgs.Result, NO_ERR
 // write to the file using FileHandle
lblEndIf:
```

## syscall FileOpenAppend, args — Function

This function lets you open an existing file that you can write to using the values specified via the TFileOpen 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.

```
TFileOpen struct
  Result dword
  FileHandle byte
  Filename byte[]
  Length dword
TFileOpen ends
```

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

```
dseg segment
  foArgs TFileOpen
dseg ends
mov foArgs.Filename, 'myfile.txt'
syscall FileOpenAppend, foArgs // open the file
brcmp NEQ, lblEndIf, foArgs.Result, NO_ERR
  // write to the file using FileHandle
  // up to the remaining available length in Length
lblEndIf:
```

## syscall FileOpenRead, args — Function

This function lets you open an existing file for reading using the values specified via the TFileOpen 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.

```
TFileOpen struct
  Result dword
  FileHandle byte
  Filename byte[]
  Length dword
TFileOpen ends
```

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

```
dseg segment
  foArgs TFileOpen
dseg ends
mov foArgs.Filename, 'myfile.txt'
```

```
syscall FileOpenRead, foArgs // open the file for reading
brcmp NEQ, lblEndIf, foArgs.Result, NO_ERR
  // read data from the file using FileHandle
lblEndIf:
```

### syscall FileRead, args — Function

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

```
TFileReadWrite struct
  Result dword
  FileHandle byte
  Buffer byte[]
  Length dword
TFileReadWrite
```

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

```
dseg segment
  frArgs TFileReadWrite
dseg ends
mov frArgs.FileHandle, foArgs.FileHandle
set frArgs.Length, 12 // number of bytes to read
syscall FileRead, frArgs
brcmp NEQ, lblEndIf, frArgs.Result, NO_ERR
  TextOut(0, LCD_LINE1, frArgs.Buffer)
  // show how many bytes were actually read
  NumOut(0, LCD_LINE2, frArgs.Length)
lblEndIf:
```

### syscall FileWrite,  args — Function

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

```
TFileReadWrite struct
  Result dword
  FileHandle byte
  Buffer byte[]
  Length dword
TFileReadWrite
```

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

```
dseg segment
  fwArgs TFileReadWrite
dseg ends
mov fwArgs.FileHandle,  foArgs.FileHandle
mov fwArgs.Buffer, 'data to write'
syscall FileWrite, fwArgs
brcmp NEQ, lblEndIf, fwArgs.Result, NO_ERR
  // display number of bytes written
```

```
    NumOut(0, LCD_LINE1, fwArgs.Length)
  lblEndIf:
```

### syscall FileClose, args — Function

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

```
TFileClose struct
  Result dword
  FileHandle byte
TFileClose ends
```

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

```
dseg segment
  fcArgs TFileClose
dseg ends
mov fcArgs.FileHandle, foArgs.FileHandle
syscall FileClose, fcArgs
```

### syscall FileResolveHandle, args — Function

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

```
TFileResolveHandle struct
  Result dword
  FileHandle byte
  WriteHandle byte
  Filename byte[]
TFileResolveHandle ends
```

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

```
dseg segment
  frhArgs TFileResolveHandle
dseg ends
mov frhArgs.Filename, 'myfile.txt'
syscall FileResolveHandle, frhArgs
brcmp NEQ, lblEndIfLdrSuccess, frhArgs.Result, LDR_SUCCESS
  // use the FileHandle as needed
  brtst EQ, lblElseIfWriteHandle, frhArgs.WriteHandle
    // file is open for writing
    jmp lblEndIfWriteHandle
  lblElseIfWriteHandle:
    // file is open for reading
  lblEndIfWriteHandle:
lblEndIfLdrSuccess:
```

### syscall FileRename, args — Function

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

```
TFileRename struct
  Result dword
  OldFilename byte[]
  NewFilename byte[]
TFileRename ends
```

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

```
dseg segment
  frArgs TFileRename
dseg ends
mov frArgs.OldFilename, 'myfile.txt'
mov frArgs.NewFilename, 'myfile2.txt'
syscall FileRename, frArgs
brcmp NEQ, lblEndIfLdrSuccess, frhArgs.Result, LDR_SUCCESS
  // do something
lblEndIfLdrSuccess:
```

## syscall FileDelete, args Function

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

```
TFileDelete struct
  Result dword
  Filename byte[]
TFileDelete ends
```

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

```
dseg segment
  fdArgs TFileDelete
dseg ends
mov fdArgs.Filename, 'myfile.txt'
syscall FileDelete, fdArgs // delete the file
```

## syscall CommLSWrite, args Function

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

```
TCommLSWrite struct
  Result sbyte
  Port byte
  Buffer byte[]
  ReturnLen byte
TCommLSWrite ends
```

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

```
dseg segment
  args TCommLSWrite
dseg ends
```

```
set args.Port, IN_1
mov args.Buffer, myBuf
set args.ReturnLen, 8
syscall CommLSWrite, args
// check Result for error status
```

### syscall CommLSCheckStatus, args — Function

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

```
TCommLSCheckStatus struct
  Result sbyte
  Port byte
  BytesReady byte
TCommLSCheckStatus ends
```

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

```
dseg segment
  args TCommLSCheckStatus
dseg ends
set args.Port, IN_1
syscall CommLSCheckStatus, args
// is the status (Result) IDLE?
brcmp NEQ, lblEndIf, args.Result, LOWSPEED_IDLE
  // proceed
lblEndIf:
```

### syscall CommLSRead, args — Function

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

```
TCommLSRead struct
  Result sbyte
  Port byte
  Buffer byte[]
  BufferLen byte
TCommLSRead ends
```

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

```
dseg segment
  args TCommLSRead
dseg ends
set args.Port, IN_1
mov args.Buffer, myBuf
set args.BufferLen, 8
syscall CommLSRead, args
// check Result for error status & use Buffer contents
```

## syscall MessageWrite, args — Function

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

```
TMessageWrite struct
  Result sbyte
  QueueID byte
  Message byte[]
TMessageWrite ends
```

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

```
dseg segment
  args TMessageWrite
dseg ends
set args.QueueID, MAILBOX1 // 0
mov args.Message, 'testing'
syscall MessageWrite, args
// check Result for error status
```

## syscall MessageRead, args — Function

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

```
TMessageRead struct
  Result sbyte
  QueueID byte
  Remove byte
  Message byte[]
TMessageRead ends
```

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

```
dseg segment
  args TMessageRead
dseg ends
set args.QueueID, MAILBOX1 // 0
set args.Remove, TRUE
syscall MessageRead, args
brcmp NEQ, lblEndIf, args.Result, NO_ERR
  TextOut(0, LCD_LINE1, args.Message)
lblEndIf:
```

## syscall CommBTWrite, args — Function

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

```
TCommBTWrite struct
  Result sbyte
  Connection byte
```

```
  Buffer byte[]
TCommBTWrite ends
```

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

```
dseg segment
  args TCommBTWrite
dseg ends
set args.Connection, 1
mov args.Buffer, myData
syscall CommBTWrite, args
```

### syscall CommBTCheckStatus, args **Function**

This function lets you check the status of a Bluetooth connection using the values specified via the TCommBTCheckStatus 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).

```
TCommBTCheckStatus struct
  Result sbyte
  Connection byte
  Buffer byte[]
TCommBTCheckStatus ends
```

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

```
dseg segment
  args TCommBTCommBTCheckStatus
dseg ends
set args.Connection, 1
syscall CommBTCheckStatus, args
brcmp NEQ, lblEndIf, args.Result, LDR_SUCCESS
  // do something
lblEndIf:
```

### syscall IOMapRead, args **Function**

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

```
TIOMapRead struct
  Result sbyte
  ModuleName byte[]
  Offset word
  Count word
  Buffer byte[]
TIOMapRead ends
```

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

```
dseg segment
  args TIOMapRead
```

```
dseg ends
mov args.ModuleName, CommandModuleName
set args.Offset, CommandOffsetTick
set args.Count, 4 // this value happens to be 4 bytes long
syscall IOMapRead, args
brcmp NEQ, lblEndIf, args.Result, NO_ERR
  // do something with the data
lblEndIf:
```

### syscall IOMapWrite, args — Function

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

```
TIOMapWrite struct
  Result sbyte
  ModuleName byte[]
  Offset word
  Buffer byte[]
TIOMapWrite ends
```

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

```
dseg segment
  args TIOMapWrite
dseg ends
mov args.ModuleName, SoundModuleName
set args.Offset, SoundOffsetSampleRate
mov args.Buffer, theData
syscall IOMapWrite, args
```

### syscall IOMapReadByID, args — Function (+)

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

```
TIOMapReadByID struct
  Result sbyte
  ModuleID dword
  Offset word
  Count word
  Buffer byte[]
TIOMapReadByID ends
```

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

```
dseg segment
  args TIOMapReadByID
dseg ends
mov args.ModuleID, CommandModuleID
set args.Offset, CommandOffsetTick
```

```
set args.Count, 4 // this value happens to be 4 bytes long
syscall IOMapReadByID, args
brcmp NEQ, lblEndIf, args.Result, NO_ERR
  // do something with the data
lblEndIf:
```

### syscall IOMapWriteByID, args — Function (+)

This function lets you write data to a firmware module's IOMap using the values specified via the TIOMapWriteByID 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.

```
TIOMapWriteByID struct
  Result sbyte
  ModuleID dword
  Offset word
  Buffer byte[]
TIOMapWriteByID ends
```

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

```
dseg segment
  args TIOMapWriteByID
dseg ends
mov args.ModuleID, SoundModuleID
set args.Offset, SoundOffsetSampleRate
mov args.Buffer, theData
syscall IOMapWriteByID, args
```

### syscall DisplayExecuteFunction, args — Function (+)

This function lets you directly execute the Display module's primary drawing function using the values specified via the TDisplayExecuteFunction 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.

```
TDisplayExecuteFunction struct
  Status byte
  Cmd byte
  On byte
  X1 byte
  Y1 byte
  X2 byte
  Y2 byte
TDisplayExecuteFunction ends
```

| 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.

```
dseg segment
  args TDisplayExecuteFunction
dseg ends
set args.Cmd, DISPLAY_ERASE_ALL
syscall DisplayExecuteFunction, args
```

### syscall CommExecuteFunction, args Function (+)

This function lets you directly execute the Comm module's primary function using the values specified via the TCommExecuteFunction 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.

```
TCommExecuteFunction struct
  Result word
  Cmd byte
  Param1 byte
  Param2 byte
  Param3 byte
  Name byte[]
  RetVal word
TCommExecuteFunction ends
```

| 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.

```
dseg segment
  args TCommExecuteFunction
dseg ends
set args.Cmd, INTF_BTOFF
syscall CommExecuteFunction, args
```

### syscall LoaderExecuteFunction, args Function (+)

This function lets you directly execute the Loader module's primary function using the values specified via the TLoaderExecuteFunction 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.

```
TLoaderExecuteFunction struct
  unsigned int Result;
  byte Cmd;
  string Filename;
  byte Buffer[];
  unsigned long Length;
TLoaderExecuteFunction ends
```

| 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.

```
dseg segment
  args TLoaderExecuteFunction
dseg ends
set args.Cmd, LDR_CMD_DELETEUSERFLASH // delete user flash
syscall LoaderExecuteFunction, args
```

## *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 10. 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 `IN_1`, `IN_2`, `IN_3`, and `IN_4` have been defined. These sensor names may be used in any function that requires a sensor port as an argument.

### 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 (IN_TYPE_NO_SENSOR) 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.

| NBC Sensor Type | Meaning |
|---|---|
| IN_TYPE_NO_SENSOR | no sensor configured |
| IN_TYPE_SWITCH | NXT or RCX touch sensor |
| IN_TYPE_TEMPERATURE | RCX temperature sensor |
| IN_TYPE_REFLECTION | RCX light sensor |
| IN_TYPE_ANGLE | RCX rotation sensor |
| IN_TYPE_LIGHT_ACTIVE | NXT light sensor with light |
| IN_TYPE_LIGHT_INACTIVE | NXT light sensor without light |
| IN_TYPE_SOUND_DB | NXT sound sensor with dB scaling |
| IN_TYPE_SOUND_DBA | NXT sound sensor with dBA scaling |
| IN_TYPE_CUSTOM | Custom sensor (unused) |
| IN_TYPE_LOWSPEED | I2C digital sensor |
| IN_TYPE_LOWSPEED_9V | I2C digital sensor (9V power) |
| IN_TYPE_HISPEED | Highspeed sensor (unused) |

**Table 11. 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 `IN_MODE_ANGLESTEP` is useful only with rotation sensors. Call SetSensorMode to set the sensor mode. The possible modes are shown below.

| NBC Sensor Mode | Meaning |
|---|---|
| IN_MODE_RAW | raw value from 0 to 1023 |
| IN_MODE_BOOLEAN | boolean value (0 or 1) |
| IN_MODE_TRANSITIONCNT | counts number of boolean transitions |
| IN_MODE_PERIODCOUNTER | counts number of boolean periods |
| IN_MODE_PCTFULLSCALE | value from 0 to 100 |
| IN_MODE_FAHRENHEIT | degrees F |
| IN_MODE_CELSIUS | degrees C |
| IN_MODE_ANGLESTEP | rotation (16 ticks per revolution) |

**Table 12. Sensor Mode 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. `IN_MODE_PCTFULLSCALE`), 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 11). |
| InputMode | The sensor mode (see Table 12). |
| RawValue | The raw sensor value |
| NormalizedValue | The normalized sensor value |
| ScaledValue | The scaled sensor value |
| InvalidData | Invalidates the current sensor value |

**Table 13. Sensor Field Constants**

## 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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
SetSensorType(IN_1, IN_TYPE_SWITCH)
```

## 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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
SetSensorMode(IN_1, IN_MODE_RAW) // raw mode

SetSensorMode(IN_1, IN_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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
SetSensorLight(IN_1)
```

### 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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
SetSensorSound(IN_1)
```

### SetSensorTouch(port) — Function

Configure the sensor on the specified port as a touch sensor. The port may be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
SetSensorTouch(IN_1)
```

### 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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
SetSensorLowspeed(IN_1)
```

### SetSensorUltrasonic(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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
SetSensorUltrasonic(IN_1)
```

### 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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`).

```
ClearSensor(IN_1)
```

### 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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable.

```
ResetSensor(x) // x = IN_1
```

### SetInCustomZeroOffset(const p, value) — Function

Sets the custom sensor zero offset value of a sensor. The port must be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`).

```
SetInCustomZeroOffset(IN_1, 12)
```

### SetInCustomPercentFullScale(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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`).

```
SetInCustomPercentFullScale(IN_1, 100)
```

### SetInCustomActiveStatus(const p, value) Function

Sets the custom sensor active status value of a sensor. The port must be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`).

```
SetInCustomActiveStatus(IN_1, true)
```

### SetInDigiPinsDirection(const p, value) Function

Sets the digital pins direction value of a sensor. The port must be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`). A value of 1 sets the direction to output. A value of 0 sets the direction to input.

```
SetInDigiPinsDirection(IN_1, 1)
```

### SetInDigiPinsStatus(const p, value) Function

Sets the digital pins status value of a sensor. The port must be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`).

```
SetInDigiPinsStatus(IN_1, false)
```

### SetInDigiPinsOutputLevel(const p, value) Function

Sets the digital pins output level value of a sensor. The port must be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`).

```
SetInDigiPinsOutputLevel(IN_1, 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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) unless otherwise specified.

### ReadSensor(n, out result) Function

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). A variable whose value is the desired sensor port may also be used.

```
ReadSensor(IN_1, x) // read sensor 1
```

### ReadSensorUS(n, out result) Function

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 ReadSensor(n, result) value. A variable whose value is the desired sensor port may also be used.

```
ReadSensorUS(IN_4, dist) // read sensor 4
```

### GetInSensorBoolean(const n, out value) Function

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`.

```
GetInSensorBoolean(IN_1, bvalue)
```

### GetInCustomZeroOffset(const p, out value) Function

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).

```
GetInCustomZeroOffset(IN_1, zoValue)
```

### GetInCustomPercentFullScale(const p, out value) Function

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).

```
GetInCustomPercentFullScale(IN_1, value)
```

### GetInCustomActiveStatus(const p, out value) Function

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).

```
GetInCustomActiveStatus(IN_1, value)
```

### GetInDigiPinsDirection(const p, out value) Function

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).

```
GetInDigiPinsDirection(IN_1, value)
```

### GetInDigiPinsStatus(const p, out value) Function

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).

```
GetInDigiPinsStatus(IN_1, value)
```

### GetInDigiPinsOutputLevel(const p, out value) Function

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).

```
GetInDigiPinsOutputLevel(IN_1, value)
```

## 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 |

**Table 14. 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 15. Output Module Constants**

Nearly all of the NBC 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. The NBC 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 17.<br><br>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 18.<br><br>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.<br><br>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.<br><br>Set the UF_UPDATE_RESET_COUNT flag in UpdateFlags to reset TachoCount and cancel any TachoLimit.<br><br>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.<br>Use UF_UPDATE_TACHO_LIMIT with the UpdateFlags field to commit changes to the TachoLimit.<br><br>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.<br><br>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.<br><br>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.<br><br>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 20.<br><br>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.<br><br>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.<br><br>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).<br>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/<br>Write | 0..255 | This field specifies the proportional term used in the internal proportional-integral-derivative (PID) control algorithm.<br><br>Set UF_UPDATE_PID_VALUES to commit changes to RegPValue, RegIValue, and RegDValue simultaneously. |
| RegIValue | ubyte | Read/<br>Write | 0..255 | This field specifies the integral term used in the internal proportional-integral-derivative (PID) control algorithm.<br><br>Set UF_UPDATE_PID_VALUES to commit changes to RegPValue, RegIValue, and RegDValue simultaneously. |
| RegDValue | ubyte | Read/<br>Write | 0..255 | This field specifies the derivative term used in the internal proportional-integral-derivative (PID) control algorithm.<br><br>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.<br><br>Refer to the UpdateFlags description for information about how to use block-relative position counts.<br><br>Set the UF_UPDATE_RESET_BLOCK_COUNT flag in UpdateFlags to request that the firmware reset the BlockTachoCount.<br><br>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.<br><br>Refer to the UpdateFlags description for information about how to use program-relative position counts.<br><br>Set the UF_UPDATE_RESET_ROTATION_COUNT flag in UpdateFlags to request that the firmware reset the RotationCount.<br><br>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 16. 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 17. 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 18. 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 19. 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 20. 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 21. 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 22. 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 22.

```
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 22. 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 21.

```
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 22.

```
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 22. 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 21.

```
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 22. 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 22.

```
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 22. 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 21.

```
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 22.

```
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 22. 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 21.

```
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 22. Valid regulation modes are listed in Table 20.

```
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 22. Valid regulation modes are listed 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 21.

```
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 22. Valid regulation modes are listed in Table 20.

```
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 22. Valid regulation modes are listed 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 21.

```
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 22.

```
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 22. 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 21.

```
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 22.

```
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 22. 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 21.

```
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 22.

```
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 22. 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 22. 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 22. 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 22.

```
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 22.

```
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 22.

```
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 22.

```
ResetAllTachoCounts(OUT_AB)
```

### 3.3.2. 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 23. Output Module IOMap Offsets**

## *3.4. Sound Module*

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

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

**Table 24. 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 NBC 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.

### 3.4.1. 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)
```

### 3.4.2. 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 25. 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 26. 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 27. 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 28. Miscellaneous Sound Constants**

## GetSoundState(out state, out flags) Function

Return the current sound state. Valid sound state values are listed in Table 26. Valid sound flags values are listed in Table 25.

```
GetSoundState(state, flags)
```

## SetSoundState(state, flags, out result) Function

Set the current sound module state. Valid sound state values are listed in Table 26. Valid sound flags values are listed in Table 25.

```
SetSoundState(SOUND_STATE_STOP, SOUND_FLAGS_UPDATE, result)
```

## SetSoundFlags(n) Function

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

```
SetSoundFlags(SOUND_FLAGS_UPDATE)
```

### SetSoundModuleState(n) **Function**

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

```
SetSoundState(SOUND_STATE_STOP)
```

### GetSoundMode(out mode) **Function**

Return the current sound mode. Valid sound mode values are listed in Table 27.

```
GetSoundMode(mode)
```

### SetSoundMode(n) **Function**

Set the current sound mode. Valid sound mode values are listed in Table 27.

```
SetSoundMode(SOUND_MODE_ONCE)
```

### GetSoundFrequency(out freq) **Function**

Return the current sound frequency.

```
GetSoundFrequency(freq)
```

### SetSoundFrequency(n) **Function**

Set the current sound frequency.

```
SetSoundFrequency(440)
```

### GetSoundDuration(out duration) **Function**

Return the current sound duration.

```
GetSoundDuration(duration)
```

### SetSoundDuration(n) **Function**

Set the current sound duration.

```
SetSoundDuration(500)
```

### GetSoundSampleRate(out rate) **Function**

Return the current sound sample rate.

```
GetSoundSampleRate(rate)
```

### SetSoundSampleRate(n) **Function**

Set the current sound sample rate.

```
SetSoundSampleRate(4000)
```

### GetSoundVolume(out volume) **Function**

Return the current sound volume.

```
GetSoundVolume(volume)
```

### SetSoundVolume(n) **Function**

Set the current sound volume.

```
SetSoundVolume(3)
```

### 3.4.3. 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 29. Sound Module IOMap Offsets**

## *3.5. IOCtrl Module*

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

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

**Table 30. 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 NBC program.

```
RebootInFirmwareMode
```

### 3.5.1. IOMap Offsets

| IOCtrl Module Offsets | Value | Size |
|---|---|---|
| IOCtrlOffsetPowerOn | 0 | 2 |

**Table 31. IOCtrl Module IOMap Offsets**

## *3.6. Display module*

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

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

**Table 32. 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 NBC 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.

### 3.6.1. High-level functions

**NumOut(x, y, value)** **Function**

Draw a numeric value on the screen at the specified x and y location.

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

**NumOutEx(x, y, value, clear)** **Function**

Draw a numeric value on the screen at the specified x and y location. Clear the screen first if clear equals TRUE.

```
NumOutEx(0, LCD_LINE1, x, TRUE)
```

**TextOut(x, y, msg)** **Function**

Draw a text value on the screen at the specified x and y location.

```
TextOut(0, LCD_LINE3, 'Hello World!')
```

**TextOutEx(x, y, msg, clear)** **Function**

Draw a text value on the screen at the specified x and y location. Clear the screen first if clear equals TRUE.

```
TextOutEx(0, LCD_LINE3, 'Hello World!', FALSE)
```

**GraphicOut(x, y, filename)** **Function**

Draw the specified graphic icon file on the screen at the specified x and y location. 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)** **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. Clear the screen first if clear equals TRUE. If the file cannot be found then nothing will be drawn and no errors will be reported.

```
GraphicOutEx(40, 40, 'image.ric', variables, TRUE)
```

**CircleOut(x, y, radius)** **Function**

Draw a circle on the screen with its center at the specified x and y location, using the specified radius.

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

### CircleOutEx(x, y, radius, clear) Function

Draw a circle on the screen with its center at the specified x and y location, using the specified radius. Clear the screen first if clear equals TRUE.

```
CircleOutEx(40, 40, 10, TRUE)
```

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

Draw a line on the screen from x1, y1 to x2, y2.

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

### LineOutEx(x1, y1, x2, y2, clear) Function

Draw a line on the screen from x1, y1 to x2, y2. Clear the screen first if clear equals TRUE.

```
LineOutEx(40, 40, 10, 10, FALSE)
```

### PointOut(x, y) Function

Draw a point on the screen at x, y.

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

### PointOutEx(x, y, clear) Function

Draw a point on the screen at x, y. Clear the screen first if clear equals TRUE.

```
PointOutEx(40, 40, TRUE)
```

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

Draw a rectangle on the screen at x, y with the specified width and height.

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

### RectOutEx(x, y, width, height, clear) Function

Draw a rectangle on the screen at x, y with the specified width and height. Clear the screen first if clear equals TRUE.

```
RectOutEx(40, 40, 30, 10, TRUE)
```

### ClearScreen() Function

Clear the NXT LCD to a blank screen.

```
ClearScreen()
```

## 3.6.2. Low-level functions

Valid display flag values are listed in the following table.

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

**Table 33. Display Flags Constants**

### GetDisplayFlags(out flags) Function

Return the current display flags. Valid flag values are listed in Table 33.

```
GetDisplayFlags(flags)
```

### SetDisplayFlags(n) Function

Set the current display flags. Valid flag values are listed in Table 33.

```
SetDisplayFlags(x)
```

### GetDisplayEraseMask(out emask) Function

Return the current display erase mask.

```
GetDisplayEraseMask(emask)
```

### SetDisplayEraseMask(n) Function

Set the current display erase mask.

```
SetDisplayEraseMask(x)
```

### GetDisplayUpdateMask(out umask) Function

Return the current display update mask.

```
GetDisplayUpdateMask(umask)
```

### SetDisplayUpdateMask(n) Function

Set the current display update mask.

```
SetDisplayUpdateMask(x)
```

### GetDisplayDisplay(out addr) Function

Return the current display memory address.

```
GetDisplayDisplay(addr)
```

### SetDisplayDisplay(n) Function

Set the current display memory address.

```
SetDisplayDisplay(x)
```

### GetDisplayTextLinesCenterFlags(out flags) Function

Return the current display text lines center flags.

```
GetDisplayTextLinesCenterFlags(flags)
```

### SetDisplayTextLinesCenterFlags(n) Function

Set the current display text lines center flags.

```
SetDisplayTextLinesCenterFlags(x)
```

### GetDisplayNormal(x, line, count, out 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 `TEXTLINE_1` through `TEXTLINE_8` 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 `TEXTLINE_1` through `TEXTLINE_8` for the "line" parameter.

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

### GetDisplayPopup(x, line, count, out 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 `TEXTLINE_1` through `TEXTLINE_8` 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 `TEXTLINE_1` through `TEXTLINE_8` for the "line" parameter.

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

## 3.6.3. 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 34. Display Module IOMap Offsets**

## 3.7. Loader Module

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

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

**Table 35. 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 NBC 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 36. Loader Result Codes**

### GetFreeMemory(out mem) Function

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

```
GetFreeMemory(memory)
```

### CreateFile(filename, size, out handle, out result) Function

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 NBC file writing functions do not grow the file if its capacity is exceeded during a write attempt.

```
CreateFile('data.txt', 1024, handle, result)
```

### OpenFileAppend(filename, out size, out handle, out result) Function

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.

```
OpenFileAppend('data.txt', fsize, handle, result)
```

### OpenFileRead(filename, out size, out handle, out result) Function

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.

```
OpenFileRead('data.txt', fsize, handle, result)
```

### CloseFile(handle, out result) Function

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.

```
CloseFile(handle, result)
```

### ResolveHandle(filename, out handle, out bWriteable, out result)Function

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.

```
ResolveHandle('data.txt', handle, bCanWrite, result)
```

### RenameFile(oldfilename, newfilename, out result) Function

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.

```
RenameFile('data.txt', 'mydata.txt', result)
```

### DeleteFile(filename, out result) Function

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.

```
DeleteFile('data.txt', result)
```

### Read(handle, out value, out result) Function

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.

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

### ReadLn(handle, out value, out result) Function

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.

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

### ReadBytes(handle, in/out length, out buf, out result) Function

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.

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

### Write(handle, value, out result) Function

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.

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

### WriteLn(handle, value, out result) Function

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.

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

### WriteString(handle, str, out count, out result) Function

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.

```
WriteString(handle, 'testing', count, result)
```

### WriteLnString(handle, str, out count, out result) Function

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.

```
WriteLnString(handle, 'testing', count, result)
```

### WriteBytes(handle, data, out count, out result) Function

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.

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

### WriteBytesEx(handle, in/out length, buf, out result) Function

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.

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

### 3.7.1. IOMap Offsets

| Loader Module Offsets | Value | Size |
|---|---|---|
| LoaderOffsetPFunc | 0 | 4 |
| LoaderOffsetFreeUserFlash | 4 | 4 |

**Table 37. Loader Module IOMap Offsets**

## *3.8. 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 38. Command Module Constants**

### 3.8.1. 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 39. Command Module IOMap Offsets**

## *3.9. 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 40. Button Module Constants**

### 3.9.1. 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 41. Button Constants**

**ReadButtonEx(btn, reset, out pressed, out count, out result) 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 41.

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

### 3.9.2. 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 42. Button State Constants**

### GetButtonPressCount(btn, out cnt) Function

Return the press count of the specified button. Valid values for the btn argument are listed in Table 41.

```
GetButtonPressCount(BTN1, count)
```

### GetButtonLongPressCount(btn, out cnt) Function

Return the long press count of the specified button. Valid values for the btn argument are listed in Table 41.

```
GetButtonLongPressCount(BTN1, count)
```

### GetButtonShortReleaseCount(btn, out cnt) Function

Return the short release count of the specified button. Valid values for the btn argument are listed in Table 41.

```
GetButtonShortReleaseCount(BTN1, count)
```

### GetButtonLongReleaseCount(btn, out cnt) Function

Return the long release count of the specified button. Valid values for the btn argument are listed in Table 41.

```
GetButtonLongReleaseCount(BTN1, count)
```

### GetButtonReleaseCount(btn, out cnt) Function

Return the release count of the specified button. Valid values for the btn argument are listed in Table 41.

```
GetButtonReleaseCount(BTN1, count)
```

### GetButtonState(btn, out state) Function

Return the state of the specified button. Valid values for the btn argument are listed in Table 41. Button state values are listed in Table 42.

```
GetButtonState(BTN1, state)
```

### 3.9.3. 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 43. Button Module IOMap Offsets**

## *3.10. 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 44. UI Module Constants**

Valid flag values are listed in the following table.

| UI 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 45. 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 46. 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 47. 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 48. UI Bluetooth State Constants**

## GetVolume(out volume) Function

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

```
GetVolume(vol)
```

## SetVolume(value) Function

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

```
SetVolume(3)
```

## GetBatteryLevel(out blevel) Function

Return the battery level in millivolts.

```
GetBatteryLevel(blevel)
```

## GetBluetoothState(out bstate) Function

Return the Bluetooth state. Valid Bluetooth state values are listed in Table 48.

```
GetBluetoothState(bstate)
```

## SetBluetoothState(value) Function

Set the Bluetooth state. Valid Bluetooth state values are listed in Table 48.

```
SetBluetoothState(UI_BT_STATE_OFF)
```

## GetCommandFlags(out flags) Function

Return the command flags. Valid command flag values are listed in Table 45.

```
GetCommandFlags(flags)
```

## SetCommandFlags(value) Function

Set the command flags. Valid command flag values are listed in Table 45.

```
SetCommandFlags(UI_FLAGS_REDRAW_STATUS)
```

## GetUIState(out state) Function

Return the user interface state. Valid user interface state values are listed in Table 46.

```
GetUIState(state)
```

## SetUIState(value) Function

Set the user interface state. Valid user interface state values are listed in Table 46.

```
SetUIState(UI_STATE_LOW_BATTERY)
```

### GetUIButton(out btn) Function

Return user interface button information. Valid user interface button values are listed in Table 47.

```
GetUIButton(btn)
```

### SetUIButton(value) Function

Set user interface button information. Valid user interface button values are listed in Table 47.

```
SetUIButton(UI_BUTTON_ENTER)
```

### GetVMRunState(out state) Function

Return VM run state information.

```
GetVMRunState(state)
```

### SetVMRunState(value) Function

Set VM run state information.

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

### GetBatteryState(out state) Function

Return battery state information (0..4).

```
GetBatteryState(state)
```

### GetRechargeableBattery(out value) Function

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

```
GetRechargeableBattery(rbat)
```

### ForceOff(n) Function

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

```
ForceOff(TRUE)
```

### GetUsbState(out ustate) Function

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

```
GetUsbState(ustate)
```

### GetOnBrickProgramPointer(out ptr) Function

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

```
GetOnBrickProgramPointer(ptr)
```

### SetOnBrickProgramPointer(value) Function

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

```
SetOnBrickProgramPointer(2)
```

### GetLongAbort(out value) Function (+)

Return the enhanced NBC/NXC firmware's long abort setting (TRUE or FALSE). If set to true then a program has access the escape button. Aborting a program requires a long press of the escape button

```
GetLongAbort(value)
```

### SetLongAbort(value) Function (+)

Set the enhanced NBC/NXC firmware's long abort setting (TRUE or FALSE). If set to true then a program has access the escape button. Aborting a program requires a long press of the escape button.

```
SetLongAbort(TRUE)
```

### 3.10.1. 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 49. UI Module IOMap Offsets**

## *3.11. LowSpeed Module*

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

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

**Table 50. 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 IN_TYPE_LOWSPEED or IN_TYPE_LOWSPEED_9V on a given port before using an I2C device on that port. Use IN_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 IN_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 51. Lowspeed (I2C) Return Value Constants**

### 3.11.1. High-level functions

## LowspeedWrite(port, returnlen, buffer, out result) Function

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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) 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 51.

```
LowspeedWrite(IN_1, 1, inbuffer, result)
```

### LowspeedStatus(port, out bytesready, out result) Function

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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) 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 51. 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`.

```
LowspeedStatus(IN_1, nRead, result)
```

### LowspeedCheckStatus(port, out result) Function

This method checks the status of the I2C communication on the specified port. The port may be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) 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 51. 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`.

```
LowspeedCheckStatus(IN_1, result)
```

### LowspeedBytesReady(port, out result) Function

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 the return value will be the number of bytes in the internal read buffer. The port may be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) 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.

```
LowspeedBytesReady(IN_1, result)
```

### LowspeedRead(port, buflen, out buffer, out result) Function

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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) 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 51. If the return value is negative then the output buffer will be empty.

```
LowspeedRead(IN_1, 1, outbuffer, result)
```

## ReadI2CBytes(port, inbuf, in/out count, out outbuf, out result) Function

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., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) 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.

```
ReadI2CBytes(IN_4, writebuf, cnt, readbuf, result)
```

### 3.11.2. 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 52. 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 53. 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 54. 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 55. Low Speed Error Type Constants**

## GetLSMode(port, out result) Function

This method returns the mode of the lowspeed communication over the specified port. The port must be a constant (IN_1..IN_4).

```
GetLSMode(IN_1, mode)
```

### GetLSChannelState(port, out result) Function

This method returns the channel state of the lowspeed communication over the specified port. The port must be a constant (IN_1..IN_4).

```
GetLSChannelState(IN_1, cstate)
```

### GetLSErrorType(port, out result) Function

This method returns the error type of the lowspeed communication over the specified port. The port must be a constant (IN_1..IN_4).

```
GetLSErrorType(IN_1, errtype)
```

### GetLSState(out result) Function

This method returns the state of the lowspeed module.

```
GetLSState(state)
```

### GetLSSpeed(out result) Function

This method returns the speed of the lowspeed module.

```
GetLSSpeed(speed)
```

### 3.11.3. 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 |

**Table 56. LowSpeed Module IOMap Offsets**

## *3.12. 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 57. 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.

### 3.12.1. High-level functions

**SendRemoteBool(connection, queue, bvalue, out result) Function**

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.

```
SendRemoteBool(1, queue, false, result)
```

**SendRemoteNumber(connection, queue, value, out result) Function**

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.

```
SendRemoteNumber(1, queue, 123, result)
```

**SendRemoteString(connection, queue, strval, out result) Function**

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.

```
SendRemoteString(1, queue, 'hello world', result)
```

**SendResponseBool(queue, bvalue, out result) Function**

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.

```
SendResponseBool(queue, false, result)
```

### SendResponseNumber(queue, value, out result) Function

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.

```
SendResponseNumber(queue, 123, result)
```

### SendResponseString(queue, strval, out result) Function

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.

```
SendResponseString(queue, 'hello world', result)
```

### ReceiveRemoteBool(queue, remove, out bvalue, out result) Function

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.

```
ReceiveRemoteBool(queue, true, bvalue, result)
```

### ReceiveRemoteNumber(queue, remove, out value, out result) Function

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.

```
ReceiveRemoteBool(queue, true, value, result)
```

### ReceiveRemoteString(queue, remove, out strval, out result) Function

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.

```
ReceiveRemoteString(queue, true, strval, result)
```

### ReceiveRemoteMessageEx(queue, remove, out strval, out val, out bval, out result) Function

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.

```
ReceiveRemoteMessageEx(queue, true, strval, val, bval, result)
```

### SendMessage(queue, msg, out result) Function

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

```
SendMessage(mbox, data, result)
```

### ReceiveMessage(queue, remove, out buffer, out result) Function

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.

```
RecieveMessage(mbox, true, buffer, result)
```

### BluetoothStatus(connection, out result) Function

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`.

```
BluetoothStatus(1, result)
```

### BluetoothWrite(connection, buffer, out result) Function

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.

```
BluetoothWrite(1, data, result)
```

### RemoteMessageRead(connection, queue, out result) Function

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

```
RemoteMessageRead(1, 5, result)
```

### RemoteMessageWrite(connection, queue, msg, out result) Function

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

```
RemoteMessageWrite(1, 5, 'test', result)
```

### RemoteStartProgram(connection, filename, out result) Function

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

```
RemoteStartProgram(1, 'myprog.rxe', result)
```

### RemoteStopProgram(connection, out result) Function

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

```
RemoteStopProgram(1, result)
```

### RemotePlaySoundFile(connection, filename, bLoop, out result) Function

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

```
RemotePlaySoundFile(1, 'click.rso', false, result)
```

### RemotePlayTone(connection, frequency, duration, out result) Function

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

```
RemotePlayTone(1, 440, 1000, result)
```

### RemoteStopSound(connection, out result) Function

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

```
RemoteStopSound(1, result)
```

### RemoteKeepAlive(connection, out result) Function

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

```
RemoteKeepAlive(1, result)
```

### RemoteResetScaledValue(connection, port, out result) Function

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

```
RemoteResetScaledValue(1, IN_1, result)
```

### RemoteResetMotorPosition(connection, port, bRelative, out result)Function

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

```
RemoteResetMotorPosition(1, OUT_A, true, result)
```

### RemoteSetInputMode(connection, port, type, mode, out result) Function

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

```
RemoteSetInputMode(1, IN_1,
  IN_TYPE_LOWSPEED, IN_MODE_RAW, result)
```

### RemoteSetOutputState(connection, port, speed, mode, regmode, turnpct, runstate, tacholimit, out result) Function

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

```
RemoteSetOutputState(1, OUT_A, 75, OUT_MODE_MOTORON,
  OUT_REGMODE_IDLE, 0, OUT_RUNSTATE_RUNNING, 0, result)
```

## 3.12.2. 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 58. 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 59. 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 | 0x80 |

**Table 60. Comm BtStateStatus Constants**

Valid BtHwStatus values are listed in the following table.

| Comm BtHwStatus Constants | Value |
|---|---|
| BT_ENABLE | 0x00 |
| BT_DISABLE | 0x01 |

**Table 61. Comm BtHwStatus Constants**

Valid HsFlags values are listed in the following table.

| Comm HsFlags Constants | Value |
|---|---|
| HS_UPDATE | 1 |

**Table 62. 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 63. 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 64. 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 65. Comm Module Interface Constants**

#### 3.12.2.1. *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)
```

### GetUSBInputBufferInPtr(out value) Function

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

```
GetUSBInputBufferInPtr(x)
```

### SetUSBInputBufferOutPtr(n) Function

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

```
SetUSBInputBufferOutPtr(0)
```

### GetUSBInputBufferOutPtr(out value) Function

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

```
GetUSBInputBufferOutPtr(x)
```

### 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)
```

### GetUSBOutputBufferInPtr(out value) Function

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

```
GetUSBOutputBufferInPtr(x)
```

### SetUSBOutputBufferOutPtr(n) Function

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

```
SetUSBOutputBufferOutPtr(0)
```

### GetUSBOutputBufferOutPtr(out value) Function

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

```
GetUSBOutputBufferOutPtr(x)
```

### 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)
```

### GetUSBPollBufferInPtr(out value) Function

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

```
GetUSBPollBufferInPtr(x)
```

### SetUSBPollBufferOutPtr(n) Function

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

```
SetUSBPollBufferOutPtr(0)
```

### GetUSBPollBufferOutPtr(out value) Function

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

```
GetUSBPollBufferOutPtr(x)
```

### SetUSBState(n) Function

This method sets the USB state to the specified value.

```
SetUSBState(0)
```

### GetUSBState(out value) Function

This method returns the USB state.

```
GetUSBPollBufferOutPtr(x)
```

#### *3.12.2.2. 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)
```

### GetHSInputBufferInPtr(out value) Function

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

```
GetHSInputBufferInPtr(x)
```

### SetHSInputBufferOutPtr(n) Function

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

```
SetHSInputBufferOutPtr(0)
```

### GetHSInputBufferOutPtr(out value) Function

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

```
GetHSInputBufferOutPtr(x)
```

### 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 input pointer of the High Speed output buffer to the specified value.

```
SetHSOutputBufferInPtr(0)
```

### GetHSOutputBufferInPtr(out value) Function

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

```
GetHSOutputBufferInPtr(x)
```

### SetHSOutputBufferOutPtr(n) Function

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

```
SetHSOutputBufferOutPtr(0)
```

### GetHSOutputBufferOutPtr(out value) Function

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

```
GetHSOutputBufferOutPtr(x)
```

### SetHSFlags(n) Function

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

```
SetHSFlags(0)
```

### GetHSFlags(out value) Function

This method returns the value of the High Speed flags.

```
GetHSFlags(x)
```

### SetHSSpeed(n) Function

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

```
SetHSSpeed(1)
```

### GetHSSpeed(out value) Function

This method returns the value of the High Speed speed.

```
GetHSSpeed(x)
```

### SetHSState(n) Function

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

```
SetHSState(1)
```

### GetHSState(out value) Function

This method returns the value of the High Speed state.

```
GetHSState(x)
```

## *3.12.2.3. 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)
```

### GetBTInputBufferInPtr(out value) Function

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

```
GetBTInputBufferInPtr(x)
```

### SetBTInputBufferOutPtr(n) **Function**

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

```
SetBTInputBufferOutPtr(0)
```

### GetBTInputBufferOutPtr(out value) **Function**

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

```
GetBTInputBufferOutPtr(x)
```

### 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)
```

### GetBTOutputBufferInPtr(out value) **Function**

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

```
GetBTOutputBufferInPtr(x)
```

### SetBTOutputBufferOutPtr(n) **Function**

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

```
SetBTOutputBufferOutPtr(0)
```

### GetBTOutputBufferOutPtr(out value) **Function**

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

```
GetBTOutputBufferOutPtr(x)
```

### GetBTDeviceCount(out value) **Function**

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

```
GetBTDeviceCount(x)
```

### GetBTDeviceNameCount(out value) **Function**

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

```
GetBTDeviceNameCount(x)
```

### GetBTDeviceName(idx, out value) Function

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

```
GetBTDeviceName(0, x)
```

### GetBTConnectionName(idx, out value) Function

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

```
GetBTConnectionName(0, x)
```

### GetBTConnectionPinCode(idx, out value) Function

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

```
GetBTConnectionPinCode(0, x)
```

### GetBrickDataName(out value) Function

This method returns the name of the NXT.

```
GetBrickDataName(x)
```

### 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)
```

### GetBTDeviceClass(idx, out value) Function

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

```
GetBTDeviceClass(idx, x)
```

### GetBTDeviceStatus(idx, out value) Function

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

```
GetBTDeviceStatus(idx, x)
```

### GetBTConnectionClass(idx, out value) Function

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

```
GetBTConnectionClass(idx, x)
```

### GetBTConnectionHandleNum(idx, out value) Function

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

```
GetBTConnectionHandleNum(idx, x)
```

### GetBTConnectionStreamStatus(idx, out value) Function

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

```
GetBTConnectionStreamStatus(idx, x)
```

### GetBTConnectionLinkQuality(idx, out value) Function

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

```
GetBTConnectionLinkQuality(idx, x)
```

### GetBrickDataBluecoreVersion(out value) Function

This method returns the bluecore version of the NXT.

```
GetBrickDataBluecoreVersion(x)
```

### GetBrickDataBtStateStatus(out value) Function

This method returns the Bluetooth state status of the NXT.

```
GetBrickDataBtStateStatus(x)
```

### GetBrickDataBtHardwareStatus(out value) Function

This method returns the Bluetooth hardware status of the NXT.

```
GetBrickDataBtHardwareStatus(x)
```

### GetBrickDataTimeoutValue(out value) Function

This method returns the timeout value of the NXT.

```
GetBrickDataTimeoutValue(x)
```

## 3.12.3. 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 |
| 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 | 1550 | 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 66. Comm Module IOMap Offsets**

## 3.13. HiTechnic API Functions

### SetSensorHTGyro(port) Function

Configure the sensor on the specified port as a HiTechnic Gyro sensor.

```
SetSensorHTGyro(IN_1)
```

## ReadSensorHTGyro(port, offset, out value) Function

Read the HiTechnic Gyro sensor on the specified port. The offset value should be calculated by averaging several readings with an offset of zero while the sensor is perfectly still.

```
ReadSensorHTGyro(IN_1, gyroOffset, value)
```

## ReadSensorHTCompass(port, out value) Function

Read the compass heading value of the HiTechnic Compass sensor on the specified port.

```
ReadSensorHTCompass(IN_1, value)
```

## ReadSensorHTColorNum(port, out value) Function

Read the color number from the HiTechnic Color sensor on the specified port.

```
ReadSensorHTColorNum(IN_1, value)
```

## ReadSensorHTIRSeekerDir(port, out value) Function

Read the direction value of the HiTechnic IR Seeker on the specified port.

```
ReadSensorHTIRSeekerDir(IN_1, value)
```

## ReadSensorHTAccel(port, x, y, z, result) Function

Read X, Y, and Z axis acceleration values from the HiTechnic Accelerometer sensor. Returns a boolean value indicating whether or not the operation completed successfully.

```
ReadSensorHTAccel(IN_1, x, y, z, bVal)
```

## ReadSensorHTColor(port, ColorNum, Red, Green, Blue, result)Function

Read color number, red, green, and blue values from the HiTechnic Color sensor. Returns a boolean value indicating whether or not the operation completed successfully.

```
ReadSensorHTColor(IN_1, c, r, g, b, bVal)
```

## ReadSensorHTRawColor(port, Red, Green, Blue, result) Function

Read the raw red, green, and blue values from the HiTechnic Color sensor. Returns a boolean value indicating whether or not the operation completed successfully.

```
ReadSensorHTRawColor(IN_1, r, g, b, bVal)
```

## ReadSensorHTNormalizedColor(port, ColorIdx, Red, Green, Blue, result) Function

Read color index and the normalized red, green, and blue values from the HiTechnic Color sensor. Returns a boolean value indicating whether or not the operation completed successfully.

```
ReadSensorHTNormalizedColor(IN_1, c, r, g, b, bVal)
```

## ReadSensorHTIRSeeker(port, dir, IN_1, IN_3, s5, s7, s9, result)Function

Read direction, and five signal strength values from the HiTechnic IRSeeker sensor. Returns a boolean value indicating whether or not the operation completed successfully.

```
ReadSensorHTIRSeeker(port, dir, IN_1, IN_3, s5, s7, s9, bVal)
```

### HTPowerFunctionCommand(port, ch, cmd1, cmd2, result) Function

Execute a pair of Power Function motor commands on the specified channel using the HiTechnic iRLink device. Commands are HTPF_CMD_STOP, HTPF_CMD_REV, HTPF_CMD_FWD, and HTPF_CMD_BRAKE. Valid channels are HTPF_CHANNEL_1 through HTPF_CHANNEL_4.

```
HTPowerFunctionCommand(IN_1, HTPF_CHANNEL_1, HTPF_CMD_STOP,
HTPF_CMD_FWD, result)
```

### HTRCXSetIRLinkPort(port) Function

Set the global port in advance of using the HTRCX* and HTScout* API functions for sending RCX and Scout messages over the HiTechnic iRLink device.

```
HTRCXSetIRLinkPort(IN_1)
```

### HTRCXPoll(src, value, out result) Function

Send the Poll command to an RCX to read a signed 2-byte value at the specified source and value combination.

```
HTRCXPoll(RCX_VariableSrc, 0, x)
```

### HTRCXBatteryLevel(out result) Function

Send the BatteryLevel command to an RCX to read the current battery level.

```
HTRCXBatteryLevel(x)
```

### HTRCXPing() Function

Send the Ping command to an RCX.

```
HTRCXPing()
```

### HTRCXDeleteTasks() Function

Send the DeleteTasks command to an RCX.

```
HTRCXDeleteTasks()
```

### HTRCXStopAllTasks() Function

Send the StopAllTasks command to an RCX.

```
HTRCXStopAllTasks()
```

### HTRCXPBTurnOff() Function

Send the PBTurnOff command to an RCX.

```
HTRCXPBTurnOff()
```

### HTRCXDeleteSubs() Function

Send the DeleteSubs command to an RCX.

```
HTRCXDeleteSubs()
```

### HTRCXClearSound() Function

Send the ClearSound command to an RCX.

```
HTRCXClearSound()
```

### HTRCXClearMsg() Function

Send the ClearMsg command to an RCX.

```
HTRCXClearMsg()
```

### HTRCXMuteSound() Function

Send the MuteSound command to an RCX.

```
HTRCXMuteSound()
```

### HTRCXUnmuteSound() Function

Send the UnmuteSound command to an RCX.

```
HTRCXUnmuteSound()
```

### HTRCXClearAllEvents() Function

Send the ClearAllEvents command to an RCX.

```
HTRCXClearAllEvents()
```

### HTRCXSetOutput(outputs, mode) Function

Send the SetOutput command to an RCX to configure the mode of the specified outputs

```
HTRCXSetOutput(RCX_OUT_A, RCX_OUT_ON)
```

### HTRCXSetDirection(outputs, dir) Function

Send the SetDirection command to an RCX to configure the direction of the specified outputs.

```
HTRCXSetDirection(RCX_OUT_A, RCX_OUT_FWD)
```

### HTRCXSetPower(outputs, pwrsrc, pwrval) Function

Send the SetPower command to an RCX to configure the power level of the specified outputs.

```
HTRCXSetPower(RCX_OUT_A, RCX_ConstantSrc, RCX_OUT_FULL)
```

### HTRCXOn(outputs) Function

Send commands to an RCX to turn on the specified outputs.

```
HTRCXOn(RCX_OUT_A)
```

### HTRCXOff(outputs) Function

Send commands to an RCX to turn off the specified outputs.

```
HTRCXOff(RCX_OUT_A)
```

### HTRCXFloat(outputs) Function

Send commands to an RCX to float the specified outputs.

```
HTRCXFloat(RCX_OUT_A)
```

### HTRCXToggle(outputs) Function

Send commands to an RCX to toggle the direction of the specified outputs.

```
HTRCXToggle(RCX_OUT_A)
```

### HTRCXFwd(outputs) Function

Send commands to an RCX to set the specified outputs to the forward direction.

```
HTRCXFwd(RCX_OUT_A)
```

### HTRCXRev(outputs) Function

Send commands to an RCX to set the specified outputs to the reverse direction.

```
HTRCXRev(RCX_OUT_A)
```

### HTRCXOnFwd(outputs) Function

Send commands to an RCX to turn on the specified outputs in the forward direction.

```
HTRCXOnFwd(RCX_OUT_A)
```

### HTRCXOnRev(outputs) Function

Send commands to an RCX to turn on the specified outputs in the reverse direction.

```
HTRCXOnRev(RCX_OUT_A)
```

### HTRCXOnFor(outputs, duration) Function

Send commands to an RCX to turn on the specified outputs in the forward direction for the specified duration.

```
HTRCXOnFor(RCX_OUT_A, 100)
```

### HTRCXSetTxPower(pwr) Function

Send the SetTxPower command to an RCX.

```
HTRCXSetTxPower(0)
```

### HTRCXPlaySound(snd) Function

Send the PlaySound command to an RCX.

```
HTRCXPlaySound(RCX_SOUND_UP)
```

### HTRCXDeleteTask(n) Function

Send the DeleteTask command to an RCX.

```
HTRCXDeleteTask(3)
```

### HTRCXStartTask(n) — Function

Send the StartTask command to an RCX.

```
HTRCXStartTask(2)
```

### HTRCXStopTask(n) — Function

Send the StopTask command to an RCX.

```
HTRCXStopTask(1)
```

### HTRCXSelectProgram(prog) — Function

Send the SelectProgram command to an RCX.

```
HTRCXSelectProgram(3)
```

### HTRCXClearTimer(timer) — Function

Send the ClearTimer command to an RCX.

```
HTRCXClearTimer(0)
```

### HTRCXSetSleepTime(t) — Function

Send the SetSleepTime command to an RCX.

```
HTRCXSetSleepTime(4)
```

### HTRCXDeleteSub(s) — Function

Send the DeleteSub command to an RCX.

```
HTRCXDeleteSub(2)
```

### HTRCXClearSensor(port) — Function

Send the ClearSensor command to an RCX.

```
HTRCXClearSensor(IN_1)
```

### HTRCXPlayToneVar(varnum, duration) — Function

Send the PlayToneVar command to an RCX.

```
HTRCXPlayToneVar(0, 50)
```

### HTRCXSetWatch(hours, minutes) — Function

Send the SetWatch command to an RCX.

```
HTRCXSetWatch(3, 30)
```

### HTRCXSetSensorType(port, type) — Function

Send the SetSensorType command to an RCX.

```
HTRCXSetSensorType(IN_1, IN_TYPE_SWITCH)
```

### HTRCXSetSensorMode(port, mode) — Function

Send the SetSensorMode command to an RCX.

```
HTRCXSetSensorMode(IN_1, IN_MODE_BOOLEAN)
```

### HTRCXCreateDatalog(size) Function

Send the CreateDatalog command to an RCX.

```
HTRCXCreateDatalog(50)
```

### HTRCXAddToDatalog(src, value) Function

Send the AddToDatalog command to an RCX.

```
HTRCXAddToDatalog(RCX_InputValueSrc, IN_1)
```

### HTRCXSendSerial(first, count) Function

Send the SendSerial command to an RCX.

```
HTRCXSendSerial(0, 10)
```

### HTRCXRemote(cmd) Function

Send the Remote command to an RCX.

```
HTRCXRemote(RCX_RemotePlayASound)
```

### HTRCXEvent(src, value) Function

Send the Event command to an RCX.

```
HTRCXEvent(RCX_ConstantSrc, 2)
```

### HTRCXPlayTone(freq, duration) Function

Send the PlayTone command to an RCX.

```
HTRCXPlayTone(440, 100)
```

### HTRCXSelectDisplay(src, value) Function

Send the SelectDisplay command to an RCX.

```
HTRCXSelectDisplay(RCX_VariableSrc, 2)
```

### HTRCXPollMemory(address, count) Function

Send the PollMemory command to an RCX.

```
HTRCXPollMemory(0, 10)
```

### HTRCXSetEvent(evt, src, type) Function

Send the SetEvent command to an RCX.

```
HTRCXSetEvent(0, RCX_ConstantSrc, 5)
```

### HTRCXSetGlobalOutput(outputs, mode) Function

Send the SetGlobalOutput command to an RCX.

```
HTRCXSetGlobalOutput(RCX_OUT_A, RCX_OUT_ON)
```

**HTRCXSetGlobalDirection(outputs, dir)** **Function**

Send the SetGlobalDirection command to an RCX.

```
HTRCXSetGlobalDirection(RCX_OUT_A, RCX_OUT_FWD)
```

**HTRCXSetMaxPower(outputs, pwrsrc, pwrval)** **Function**

Send the SetMaxPower command to an RCX.

```
HTRCXSetMaxPower(RCX_OUT_A, RCX_ConstantSrc, 5)
```

**HTRCXEnableOutput(outputs)** **Function**

Send the EnableOutput command to an RCX.

```
HTRCXEnableOutput(RCX_OUT_A)
```

**HTRCXDisableOutput(outputs)** **Function**

Send the DisableOutput command to an RCX.

```
HTRCXDisableOutput(RCX_OUT_A)
```

**HTRCXInvertOutput(outputs)** **Function**

Send the InvertOutput command to an RCX.

```
HTRCXInvertOutput(RCX_OUT_A)
```

**HTRCXObvertOutput(outputs)** **Function**

Send the ObvertOutput command to an RCX.

```
HTRCXObvertOutput(RCX_OUT_A)
```

**HTRCXCalibrateEvent(evt, low, hi, hyst)** **Function**

Send the CalibrateEvent command to an RCX.

```
HTRCXCalibrateEvent(0, 200, 500, 50)
```

**HTRCXSetVar(varnum, src, value)** **Function**

Send the SetVar command to an RCX.

```
HTRCXSetVar(0, RCX_VariableSrc, 1)
```

**HTRCXSumVar(varnum, src, value)** **Function**

Send the SumVar command to an RCX.

```
HTRCXSumVar(0, RCX_InputValueSrc, IN_1)
```

**HTRCXSubVar(varnum, src, value)** **Function**

Send the SubVar command to an RCX.

```
HTRCXSubVar(0, RCX_RandomSrc, 10)
```

**HTRCXDivVar(varnum, src, value)** **Function**

Send the DivVar command to an RCX.

```
HTRCXDivVar(0, RCX_ConstantSrc, 2)
```

### HTRCXMulVar(varnum, src, value) — Function

Send the MulVar command to an RCX.

```
HTRCXMulVar(0, RCX_VariableSrc, 4)
```

### HTRCXSgnVar(varnum, src, value) — Function

Send the SgnVar command to an RCX.

```
HTRCXSgnVar(0, RCX_VariableSrc, 0)
```

### HTRCXAbsVar(varnum, src, value) — Function

Send the AbsVar command to an RCX.

```
HTRCXAbsVar(0, RCX_VariableSrc, 0)
```

### HTRCXAndVar(varnum, src, value) — Function

Send the AndVar command to an RCX.

```
HTRCXAndVar(0, RCX_ConstantSrc, 0x7f)
```

### HTRCXOrVar(varnum, src, value) — Function

Send the OrVar command to an RCX.

```
HTRCXOrVar(0, RCX_ConstantSrc, 0xCC)
```

### HTRCXSet(dstsrc, dstval, src, value) — Function

Send the Set command to an RCX.

```
HTRCXSet(RCX_VariableSrc, 0, RCX_RandomSrc, 10000)
```

### HTRCXUnlock() — Function

Send the Unlock command to an RCX.

```
HTRCXUnlock()
```

### HTRCXReset() — Function

Send the Reset command to an RCX.

```
HTRCXReset()
```

### HTRCXBoot() — Function

Send the Boot command to an RCX.

```
HTRCXBoot()
```

### HTRCXSetUserDisplay(src, value, precision) — Function

Send the SetUserDisplay command to an RCX.

```
HTRCXSetUserDisplay(RCX_VariableSrc, 0, 2)
```

### HTRCXIncCounter(counter) **Function**

Send the IncCounter command to an RCX.

```
HTRCXIncCounter(0)
```

### HTRCXDecCounter(counter) **Function**

Send the DecCounter command to an RCX.

```
HTRCXDecCounter(0)
```

### HTRCXClearCounter(counter) **Function**

Send the ClearCounter command to an RCX.

```
HTRCXClearCounter(0)
```

### HTRCXSetPriority(p) **Function**

Send the SetPriority command to an RCX.

```
HTRCXSetPriority(2)
```

### HTRCXSetMessage(msg) **Function**

Send the SetMessage command to an RCX.

```
HTRCXSetMessage(20)
```

### HTScoutCalibrateSensor() **Function**

Send the CalibrateSensor command to a Scout.

```
HTScoutCalibrateSensor()
```

### HTScoutMuteSound() **Function**

Send the MuteSound command to a Scout.

```
HTScoutMuteSound()
```

### HTScoutUnmuteSound() **Function**

Send the UnmuteSound command to a Scout.

```
HTScoutUnmuteSound()
```

### HTScoutSelectSounds(group) **Function**

Send the SelectSounds command to a Scout.

```
HTScoutSelectSounds(0)
```

### HTScoutSetLight(mode) **Function**

Send the SetLight command to a Scout.

```
HTScoutSetLight(SCOUT_LIGHT_ON)
```

### HTScoutSetCounterLimit(counter, src, value) **Function**

Send the SetCounterLimit command to a Scout.

```
HTScoutSetCounterLimit(0, RCX_ConstantSrc, 2000)
```

### HTScoutSetTimerLimit(timer, src, value) Function

Send the SetTimerLimit command to a Scout.

```
HTScoutSetTimerLimit(0, RCX_ConstantSrc, 10000)
```

### HTScoutSetSensorClickTime(src, value) Function

Send the SetSensorClickTime command to a Scout.

```
HTScoutSetSensorClickTime(RCX_ConstantSrc, 200)
```

### HTScoutSetSensorHysteresis(src, value) Function

Send the SetSensorHysteresis command to a Scout.

```
HTScoutSetSensorHysteresis(RCX_ConstantSrc, 50)
```

### HTScoutSetSensorLower Limit(src, value) Function

Send the SetSensorLowerLimit command to a Scout.

```
HTScoutSetSensorLower Limit(RCX_ConstantSrc, 100)
```

### HTScoutSetSensorUpper Limit(src, value) Function

Send the SetSensorUpperLimit command to a Scout.

```
HTScoutSetSensorUpper Limit(RCX_ConstantSrc, 400)
```

### HTScoutSetEventFeedback(src, value) Function

Send the SetEventFeedback command to a Scout.

```
HTScoutSetEventFeedback(RCX_ConstantSrc, 10)
```

### HTScoutSendVLL(src, value) Function

Send the SendVLL command to a Scout.

```
HTScoutSendVLL(RCX_ConstantSrc, 0x30)
```

### HTScoutSetScoutRules(motion, touch, light, time, effect) Function

Send the SetScoutRules command to a Scout.

```
HTScoutSetScoutRules(SCOUT_MR_FORWARD, SCOUT_TR_REVERSE,
SCOUT_LR_IGNORE, SCOUT_TGS_SHORT, SCOUT_FXR_BUG)
```

### HTScoutSetScoutMode(mode) Function

Send the SetScoutMode command to a Scout.

```
HTScoutSetScoutMode(SCOUT_MODE_POWER)
```

### *3.14. Mindsensors API Functions*

**ReadSensorMSRTClock(port, ss, mm, hh, dow, dd, MM, yy, result)Function**

Read real-time clock values from the Mindsensors RTClock sensor. Returns a boolean value indicating whether or not the operation completed successfully.

```
ReadSensorMSRTClock(IN_1, ss, mm, hh, dow, dd, mon, yy, result)
```

**ReadSensorMSCompass(port, out result) Function**

Return the Mindsensors Compass sensor value.

```
ReadSensorMSCompass(IN_1, result)
```