Version: 1.0

LEGO® MINDSTORMS® NXT Executable File Specification

TASK

This document describes the executable file format and bytecode instructions used by the virtual machine in the LEGO® MINDSTORMS® NXT Intelligent Brick.

Table of Contents

INTRODUCTION

This document provides information about the binary file format of executable programs compiled by LEGO® MINDSTORMS® NXT Software 1.0 for NXT intelligent bricks running firmware 1.03. This document also describes how the firmware virtual machine (VM) uses these program files at run-time.

You might find this document useful for any of the following reasons:

- You are developing a compiler for NXT firmware 1.03 or compatible.
- You are developing an alternative firmware which reuses all or part of NXT firmware 1.03.
- You are developing bytecode programs and want to examine the compiled code.

The document includes an **Introduction** to important concepts/terminology, a specification of the binary **File Format**, an **Example Program**, a full **Instruction Reference**, and a **Glossary**.

Assumptions and Conventions

This document assumes that you are familiar with basic programming concepts, including memory addressing, subroutines, data types, hexadecimal numbers, and little-endian vs. big-endian byte order.

This document assumes use of a compiler to produce NXT executable files. The binary file format is not designed to be hand-coded and there is currently no human-readable text format defined. Usage of a hex editor is recommended for examining file contents.

This document does not assume you are familiar with *NXT-G,* which is the graphical programming language used in LEGO MINDSTORMS NXT Software 1.0. However, an understanding of NXT-G might help with some of the concepts this document uses.. Many of the run-time semantics of this system were designed to express NXT-G block diagrams as directly as possible.

The NXT firmware 1.03 is implemented in ANSI C, so some behaviors are inherited from C. For example, the firmware stores text strings as null-terminated byte arrays, and integer conversion follows ANSI C rules.

Some knowledge of the NXT brick hardware architecture might help you as you use this document. In particular, the NXT brick uses the ARM7TDMI® core as the main CPU. The NXT brick also has flash memory and integrated RAM within a shared address space. Program files are stored in the flash memory, but while a program executes, the NXT brick keeps volatile run-time data in RAM that is much faster than the flash memory. Refer to the LEGO MINDSTORMS NXT hardware development kit for more information about the NXT hardware.

Unless otherwise noted, this document uses the following conventions:

- All offsets and/or array indexes are zero-based numbers.
- Multi-byte data fields are often referred to as *words* or *longs*. Words consist of 16 bits, whereas longs consist of 32 bits.
- All multi-byte fields are listed in little-endian byte order, because this is how NXT executable files store these fields.
- All individual bits are identified from left to right, starting with bit 0 on the left. That is, bit 0 is the most significant bit of the least significant byte.

This document uses the following typographical conventions:

- Instructions, arguments, IDs, opcodes, parameters, properties, methods, values, and so on are in `monospace` font.
- New terms and Boolean values *TRUE* and *FALSE* are in *italic* font.
- Cross-references to other sections of this document are in **bold** font.

MINDSTORMS NXT and LabVIEW

LEGO MINDSTORMS NXT Software 1.0 is based on National Instruments LabVIEW™ 7.1, so the program run-time components of the firmware mimic many of the semantics and behaviors of LabVIEW.

NXT Program Components and Execution

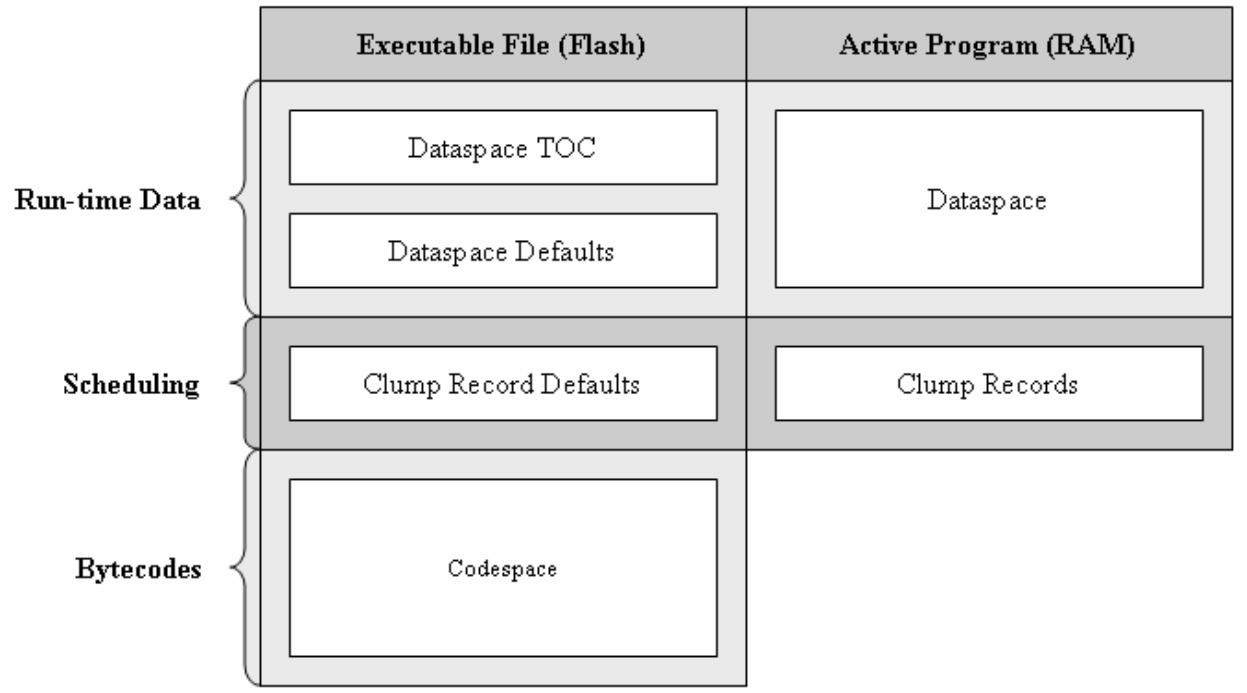
The MINDSTORMS NXT brick uses firmware that shares some architectural similarities with the earlier generation of MINDSTORMS, also known as the RCX brick. The NXT firmware consists of several modules, including those for sensors, motors, and a virtual machine (VM).

However, the MINDSTORMS NXT brick executes programs differently from the RCX. Whereas the RCX brick uses program slots, the MINDSTORMS NXT brick stores user-accessible programs as named executable files, similar to how a PC stores files. These files use the `.RXE` naming convention and contain all information necessary to run the program. In other words, one `.RXE` file represents one program.

You can break the program structure down into three high-level logical components: *bytecode instructions*, *bytecode scheduling information*, and *run-time data*. The `.RXE` files contain the information that represents these logical components.

When the VM runs a program, it reads the encoded `.RXE` file format from flash memory and initializes a 32KB pool of RAM reserved for use by user programs. The `.RXE` file specifies the layout and default content of this pool. After the RAM pool is initialized, the program is considered *active*, or ready to run. Most program functionality involves modifying this pool of RAM and performing I/O actions based on the values stored in RAM. Refer to the **Program Execution Overview** section of this document for more information about how the VM runs programs.

The following figure shows a logical view of how the three main program components are divided into sub-components and how these sub-components are arranged while a program is active.



In the previous figure, notice that the run-time data and scheduling components contain sub-components both in flash and RAM. This organization signifies that some of these sub-components are volatile at run-time, but others remain constant. Non-volatile sub-components remain in the file, which the VM can refer to at any time during program execution. The codespace is an example of a non-volatile sub-component. The bytecodes never change at run-time, so these bytecodes remain entirely in flash memory.

The following sections provide more information about these concepts.

Bytecode Instructions

The bytecode instructions constitute the main portion of a program. The VM interprets instructions to act on data in RAM and access the I/O systems of the NXT brick. For example, the `OP_ADD` instruction adds two values from RAM and stores the result back into RAM. The `OP_MOV` instruction copies data from one location in RAM to another.

The NXT supports the following six classes of bytecode instructions:

- *Math*—Includes basic math operations, such as `OP_ADD`.
- *Logic*—Includes Boolean logic operations, such as `OP_AND`.
- *Comparison*—Includes instructions for comparing values in RAM and producing Boolean outputs.
- *Data Manipulation*—Includes instructions for copying, converting, and manipulating data in RAM.
- *Control Flow*—Includes instructions for branching execution within clumps, scheduling clumps, calling subroutine clumps, and managing mutexes for parallel execution. Refer to the **Bytecode Scheduling** section of this document for information about clumps.
- *System I/O*—Includes instructions for interfacing with the built-in I/O devices and other system services of the NXT brick.

Refer to the **Instruction Reference** section of this document for information about each individual instruction.

Bytecode Scheduling and Parallelism

The program's bytecode instructions are organized into one or more batches of code. These code batches are called *clumps* and are used as subroutines for multi-tasking. Each clump consists of one or more bytecode instruction.

The VM decides how to schedule the various interdependent clumps at run-time. This decision is based on bytecode scheduling information. Programs contain one *clump record* for each clump of bytecode instructions. A clump record specifies which bytecodes belong to a given clump, the run-time execution state of the clump, and any dependent clumps that should execute after the current clump finishes.

A clump is said to be dependent on others when that clump requires data computed by one or more other clumps before executing. A dependent clump can execute only after all of its data dependencies are fulfilled. The VM uses clump records to track these data dependencies and schedules dependent clumps only when all upstream clumps have finished execution.

You also can call clumps directly from other clumps. The called clump is a *subroutine*. In this case, the caller clump temporarily suspends execution while the subroutine runs. You can share one subroutine between multiple caller clumps. However, each caller clump must acquire a mutex to call a shared subroutine safely. Subroutines might consist of as many dependent clumps as can fit in the program. Subroutines might call other subroutines, but a subroutine cannot call itself.

The VM is also capable of cooperative multi-tasking between clumps. The VM gives a portion of the CPU to any clumps deemed ready to run. Note that all clumps have equal access to data in RAM and other system resources, such as the display and motors. The language semantics of NXT-G help to prevent resource contention, but the VM does not impose any restrictions at run-time. You must ensure that parallel clumps do not interfere with each other.

Program Execution Overview

When the user runs a program, the VM executes the following four phases:

1. *Validation*—Reads the file and validates the version and other header contents.
2. *Activation*—Allocates and initializes run-time data structures in RAM. After a program has been activated, all run-time data , such as clump records, dataspace, and system bookkeeping data, is ready such that the bytecode instructions in clumps can run and operate on user data.
3. *Execution*—Interprets the bytecode instructions in the file, using the code scheduling information to decide order of clump execution. Execution continues until all clumps finish executing or the user aborts the program.
4. *Deactivation*—Re-initializes all I/O systems and data structures in RAM, and releases access to the program file.

Note that even at run-time, the VM never loads the bytecode instructions themselves into RAM. Instead, the VM executes these instructions directly out of the `.RXE` file, which resides in the flash memory address space of the NXT brick. This methodology allows a relatively small pool of RAM to be used only for volatile program data. The bytecode instructions never change during program execution.

Run-time Data

At run-time, the VM uses a reserved pool of RAM to store all data the program uses. This pool of RAM contains a segment reserved for user data; this segment is the *dataspace*. The dataspace is arranged as a collection of typed items. Each item has an entry in the *dataspace table of contents* (DSTOC), which keeps track of the data types and arrangement of all dataspace items in RAM. Instruction arguments refer to dataspace items via indexes in the DSTOC. These indexes are *dataspace item IDs,* which the instruction interpreter uses to find and operate on the data. Like the bytecode instructions, the DSTOC remains in flash memory while the program runs and cannot change at run-time.

Data Types

The NXT firmware 1.03 supports the following data types:

- *Integers*—Signed or unsigned scalar numbers. The NXT firmware 1.03 supports 8-, 16-, and 32-bit lengths. These bit lengths are scalar bytes, words, and longs, respectively.
- *Arrays*—A list of zero or more elements of any single sub-type. For example, you could use an array of unsigned bytes to express a list of port numbers for controlling motors. The program can resize arrays at run-time, which means that zero-length arrays are a valid concept. For example, you can use zero-length arrays to hold a spot in the DSTOC for data which the program might produce later.
- *Clusters*—A collection of typed fields, which are analogous to structures in C. Bytecode instructions can refer to whole clusters or any field contained in a cluster. Because clusters are ordered data structures, you must refer to sub-types, or fields, in the order in which the cluster defines these fields.
- *Mutex records*—32-bit data structures used for managing safe access to resources by parallel clumps.

Arrays and clusters are referred to as *aggregate types*. That is, these types are aggregates of one or more sub-types. Aggregates are a powerful way to organize data. Some bytecode instructions operate on aggregates as self-contained units, whereas others operate on individual elements. Furthermore, you can define nested aggregate data types, such as arrays of arrays, arrays of clusters, and clusters containing arrays.

Boolean true/false values are stored as unsigned bytes. *FALSE* is defined as `0`, and *TRUE* is defined as all other values. Note that all scalar numbers are integers. The NXT firmware 1.03 does not support floating point, or fractional, numbers.

ASCII text strings are a special kind of array. A text string is defined as an array of unsigned bytes with one extra byte added to the end. This extra byte implements C-style null-termination, which specifies that the last byte of a string must always equal zero. The NXT firmware supports this style to maintain compatibility with other code.

The NXT firmware stores mutex records as flat 32-bit data structures. However, mutex records are not treated as normal scalar values, and you cannot store mutex records in arrays or clusters. The only two instructions that can operate legally on mutex records are `OP_ACQUIRE` and `OP_RELEASE`. Refer to the **Instruction Reference** section of this document for information about these functions.

Static Data vs. Dynamic Data

Recall from the **Run-time Data** section of this document that the DSTOC specifies the dataspace layout in RAM. Also recall that the DSTOC does not change at run-time. This design means that all data types, including the initial sizes of any arrays, are fully specified at compile time. As the program activates, the VM initializes all user-defined data items to their *default* values, which are the values defined in the `.RXE` file. The program can then modify the data item values at run-time.

This user data is divided into two categories: *static* and *dynamic*. In the context of the NXT program dataspace, static dataspace items are items that the VM does not move or resize at run-time. All run-time data is static, except for data in arrays, which are dynamic. At run-time, the VM can resize or move arrays within RAM due to any side effects of bytecode operations or other internal factors. The VM handles this dynamic data management automatically, which means you do not have to specify array placement in memory. However, this management is subject to the RAM limitations of the NXT brick. If a program's array data grows beyond the maximum dataspace during program execution, the program aborts and the NXT brick displays a **`File Error`** message on the screen. Refer to the **Dynamic Data Management** section of this document for more information about how the VM handles dynamic data at run-time.

Static and dynamic data are stored in two separate sub-pools within the dataspace memory pool. The *static dataspace* sub-pool is always stored at a lower memory address than the *dynamic dataspace*. These two dataspaces never overlap.

This division separates the roles of the program compiler and the run-time execution system of the NXT brick. The compiler is responsible for specifying the data types and initial RAM locations, or *offsets*, of all dataspace items. The run-time execution system is responsible for managing the RAM locations of all dynamic dataspace items while the program runs.

Regardless of the category of a particular data item, the compiler and execution system combine to preserve proper address alignment. For example, the compiler is responsible for ensuring that 4-byte integers are always stored at addresses that are even multiples of 4, while the run-time system is responsible for ensuring that the start of array data is always aligned to 4-byte boundaries.

Recall from the **Run-time Data** section of this document that bytecode instruction arguments refer to dataspace items via indexes in the DSTOC, or *dataspace item IDs*. In the case of static dataspace items, resolving a DSTOC index to the actual RAM address of the data is involves reading an offset relative to the beginning of all user data. This resolution process can be stated as the following equation:

item address = dataspace start + DSTOC[dataspace item ID].offset

Dynamic Data Management

Dynamic data, or array, storage is specified and managed differently from static data. The compiler is responsible for specifying the DSTOC entries and default values for all dynamic data. The compiler accomplishes this task by encoding this data in the executable file. When the program activates, the NXT firmware uses a *memory manager* to handle the dynamic data.

The memory manager uses an allocation scheme to track and resize arrays inside the dynamic memory pool. After static data items have been placed in memory, the VM reserves the remaining space in the VM's 32KB memory pool for dynamic data. User data arrays are then given initial placements inside the dynamic data pool. After activation and during execution, the program can resize or move any array at any time. The memory manager automatically handles all run-time sizing and placement by using a set of bookkeeping data structures called *dope vectors*.

Dope Vectors

A dope vector (DV) is a data structure which describes an array in RAM. Each array in the dynamic dataspace has an associated dope vector. DVs are not arrays, but fixed-size records consisting of five fields. The following table describes these fields.

Field	Description
Offset	Offset to the start of this array's data in RAM relative to the beginning of all user data.
Element Size	The size, in bytes, of each array element.
Element Count	The number of elements currently contained in this array.
Back Pointer	An unused placeholder in NXT firmware 1.03.
Link Index	The index of the next DV in the memory manager's linked list.

Because the size and position of arrays can change as the program executes, the DV associated with each array must also be able to change. Therefore, the set of DVs is stored in a special dynamically-sized *dope vector array* (DVA). The DVA is stored in the same memory pool as user arrays. However, the DVA has the following special properties:

- The DVA is a singleton object, which means the memory can contain one and only one DVA at a given time. This DVA contains information about all DVs in memory.
- The first entry in the DVA, at index 0, is a DV that describes the DVA itself.
- Because the DVA is used solely for internal memory management, program bytecode instructions never refer to or modify the DVA.
- The memory manager treats the DVA as a linked-list by using the Link Index field of each DV. Maintaining this linked-list gives the memory manager a way to quickly traverse the DVA in ascending order according to the Offset fields in each DV. This design provides efficient allocation and compaction of dynamic data at run-time.

This memory management system provides a way to resolve the RAM address of any given array starting only with an index into the DSTOC. Remember that bytecode instructions use these indexes to refer to user data and that bytecode instructions do not change at run-time, even if these instructions refer to dynamic arrays.

To resolve the address of an array dataspace item, the VM indexes the DSTOC to find a secondary index, called the *DV index*. The VM them uses the DV index to find the true data offset in the DVA. The following two steps describe this process.

DV index address = dataspace start + DSTOC[dataspace item ID].offset
item address = dataspace start + DVA[DV index].offset

In these steps, the DV index value is a scalar value stored in the static dataspace, which means the address of the DV index is calculated the same way as any other static value. Refer to the **Static Data vs. Dynamic Data** section of this document for information about how static values are calculated. Because the DV index is static, and because programs cannot modify the DSTOC offset at run-time, the DV index cannot change at run-time.

Nested Arrays

Nested arrays are arrays in which each element is another array. You can think of nested arrays as having two components: top-level arrays and sub-arrays. The top-level array is the "container" array. A sub-array is an array contained by the top-level array. Nested arrays are subject to the following three rules:

1. Programs cannot create or delete top-level arrays.
2. Programs can resize top-level arrays.
3. Programs can create or delete sub-arrays according to how the top-level arrays are resized during program execution.

Because of these rules, the DV indexes of top-level arrays reside in the static dataspace, but the DV indexes of sub-arrays reside in the dynamic dataspace.

In practice, bytecode instruction arguments never refer directly to sub-arrays. Instead, the code refers to top-level arrays, and the bytecode interpreter automatically finds the correct sub-array data. You can access a specific sub-array only by using the `OP_INDEX` instruction to copy the sub-array to/from a top-level array. Refer to the **Data Manipulation Instructions** section of this document for information about this and other instructions you use to manipulate arrays.

Polymorphic Instructions and Data Type Compatibility

Most bytecode instructions accept more than one combination of input data types and produce a valid output based on the input data types. For example, you can use the `OP_ADD` instruction to produce a scalar number by adding two scalar numbers. However, you also can use this instruction to add two arrays or all scalar elements of two arrays. The `OP_ADD` instruction, like most instructions, is *polymorphic*, which means this function adapts to the data types of the inputs you provide. Most instructions are polymorphic with respect to data type.

Bytecode instructions have requirements with respect to the *compatibility* of the input data types. Two data types are compatible if and only if conversion between the two data types is trivial. If an instruction has only two inputs, those input data types must be compatible. For example, most of the Math, Logic, and Comparison functions have only two inputs, so these functions operate only on compatible data types. Other functions handle compatibility as noted in the **Instruction Reference** section of this document.

Data type compatibility relies on the following three rules:

1. Any simple scalar data type is compatible with any other simple scalar data type. Bytes, words, and longs are all compatible with each other.
2. Arrays are compatible if the sub-type, or data type of each element, is compatible.
3. Clusters are compatible if both of the following rules are true:
    a. Each cluster contains the same number of sub-elements.
    b. Each corresponding pair of sub-elements (examined in order) is compatible.

You can determine compatibility of nested data types recursively applying these rules to each level of nesting. Apply the relevant rule to each level of nesting until you reach scalar elements.

> **Note:** Mutex records are not compatible with any other data types and cannot be used with any polymorphic instructions. For example, you cannot add mutex records together.

Most Math and Logic instructions accept two compatible inputs and store the output in a dataspace item with a compatible data type. Output data types must obey the following three rules:

1. If both inputs are scalar, the output must also be scalar.
2. If either input is an array, the output must be a compatible array. The output array will be automatically sized to match the shortest input.
3. If either input is a cluster, the output must be a compatible cluster.

Similar to the general rules for data type compatibility, you can apply these rules recursively for nested data types. For example, if either input is an array of clusters, the output must be a compatible array of clusters.

Data Type Conversion

Remember that scalars of differing sizes (e.g., 16-bit vs. 32-bit) are always compatible, so it is legal to have size mismatches between inputs and outputs. The firmware automatically converts data types as necessary. The compiler typically chooses output data types that match the largest input. This choice minimizes the potential for lost data during conversion. For example, if you add an 16-bit integer to a 32-bit integer, the output is a 32-bit integer.

Polymorphic instructions internally convert scalar data types as needed. To do so, the VM performs the following three steps on each scalar element being processed.

1. Convert scalars up to 32-bit integers.
2. Perform instruction operation at 32-bit precision, that is, add two 32-bit integers to produce a 32-bit intermediate result.
3. Convert output as needed.

When aggregate types are involved in conversation, the VM applies these three steps to each scalar element individually. For example, consider the operation of adding two arrays together and storing the result in a destination array. Each element of the destination array contains the result of invoking the instruction on a corresponding pair of elements from the source data. The result is similar to writing a “for each” style loop in a programming language like C. In NXT bytecode, you can express such loops as a single instruction.

The easiest way to convert numbers (e.g., from unsigned bytes to signed bytes) is to use the `OP_MOV` instruction to copy from one dataspace item to another. At the scalar level, data type conversions behave identically to type casts in ANSI C.

Many instructions automatically resize output arrays. In cases where Math or Logic instructions operate on arrays of differing sizes, the instruction sets the output array to the same size as the shortest input array. Any remaining data in the longer input is ignored. Other classes of instructions employ instruction-specific semantics for output sizing.

Polymorphic Comparisons

Comparison instructions represent a special case of polymorphism: these instructions are polymorphic on output data types as well as input data types. Thinking of the behavior of instructions as dependent on their outputs may seem counter-intuitive, but remember that all data types must be fully determined at compile time, including output data types. This means that the VM can examine output data types at run-time and act accordingly.

For example, users might be interested in two distinct interpretations of the concept of array equality. Testing the equality of two numerical arrays using the comparison instruction `OP_CMP` may be stated as one of two questions:

1. "Are the arrays exactly equal in size, and is every corresponding pair of elements equal?" The answer to this question is expressed as a single *TRUE* or *FALSE* Boolean value. This scenario is called *aggregate comparison* because the instruction compares aggregate data types as complete units to produce a single *TRUE*/*FALSE* output.
2. "Which pairs of elements between the two arrays are equal?" The answer to this question is expressed as an array of Boolean values, where one *TRUE*/*FALSE* result applies to each pair of array elements. This scenario is called *element comparison* because the instruction compares elements of aggregate data types are compared individually to produce a *TRUE*/*FALSE* output for each pair.

The concepts of aggregate and element comparisons apply to clusters as well as arrays. Aggregate comparisons always produce a single Boolean output, whereas element comparisons produce an array or cluster of Boolean values, where one Boolean value corresponds to each pair of elements compared.

You also can compare scalars to aggregates. Similar to comparisons between aggregates, the data type of the output of this comparison can be either scalar or aggregate. If the output data type is scalar, all elements of the aggregate input are compared to the scalar input to produce a single *TRUE*/*FALSE* result, e.g., an aggregate comparison. If the output data type is aggregate, a separate *TRUE*/*FALSE* result is produced for each element of the aggregate input e.g., an element comparison.

To summarize, comparison instructions obey these three rules:

1. If both inputs are scalar, the output must be single Boolean value.
2. If either input is an array, the output may be a single Boolean value (aggregate comparison) or an array of Boolean values (element comparison).
3. If either input is a cluster, the output may be a single Boolean value (aggregate comparison) or a cluster of Boolean values (element comparison).

Similar to the general rules for data type compatibility, you can apply these rules recursively for nested data types. For example, if either input is an array of clusters, the output may be a single Boolean value or an array of clusters of Boolean values.

EXECUTABLE FILE FORMAT

This section specifies the binary file format of executable `.RXE` files for the NXT firmware 1.03. Unless otherwise noted, all tables list file contents in the order and format in which they appear in `.RXE` files.

Overview

`.RXE` files are divided into four main segments, as described in the following table.

Segment	Byte Count	Description
File Header	38	Specifies the file contents.
Dataspace	Variable	Describes data types and default values of each piece of data owned by the program.
Clump Records	Variable	Describes how clumps should be scheduled at run-time.
Codespace	Variable	Contains all bytecode instructions in a flattened stream.

All segments are 16-bit aligned, which means these segments must begin on even byte offsets relative to the beginning of the file. Since the three variably-sized segments might contain an odd number of bytes, padding bytes might come between the end of these segments and the beginning of the next segment. These padding bytes are ignored at run-time. This alignment is required because the VM needs to quickly access some 16-bit fields in the file at run-time, but the ARM7 microcontroller used in the NXT brick cannot directly access mis-aligned data. Because the NXT brick natively uses little-endian byte order, all multi-byte fields throughout the `.RXE` file are stored in little-endian byte order.

The following sections provide information about each of these segments.

File Header

All `.RXE` files contain a 38-byte file header that describes the layout of the rest of the file. The header segment contains four separate fields. The following table describes these fields.

Byte Count	Field	Description
16	Format String	Format string and version number. For NXT firmware 1.03, this string must contain the literal string `'MindstormsNXT'`, followed by a null padding byte, then `0x0005`, which is the supported file version number in big-endian byte order. In other words, all `.RXE` files supported by NXT firmware 1.03 start with these exact 16 bytes (hexadecimal): `4D 69 6E 64 73 74 6F 72 6D 73 4E 58 54 00 00 05`
18	Dataspace Header	Header sub-segment that describes the size and arrangement of static and dynamic dataspace information in the file. The section following this table provides more information about the Dataspace Header.
2	Clump Count	Unsigned 16-bit word that specifies the number of clumps in this file. NXT firmware 1.03 limits maximum number of clumps per program to 255. The most significant byte is padding in this header.
2	Code Word Count	Unsigned 16-bit word that specifies the number of all bytecode instruction words in this file.

Dataspace Header

The Dataspace Header is a sub-segment of the File Header that describes the count, size, and arrangement of the program's dataspace items. This header always begins at byte offset 16, relative to the file's start, and consists of nine unsigned 16-bit words. The following table describes these fields in the order in which they appear in `.RXE` files:

Field	Description
Count	Number of records in the DSTOC. The NXT firmware 1.03 limits the number of DSTOC records to 16.383. Refer to the **Dataspace Item IDs** section in this document for more information on this limitation.
Initial Size	Initial size, in bytes, of the dataspace in RAM, including both static and dynamic data items.
Static Size	Size, in bytes, of static segment of the dataspace in RAM. The value of this field must be less than or equal to the value of the Initial Size field. The value of this field must also be a multiple of 4, so that dynamic data is aligned on 4-byte boundaries. Round up from the actual size of the static data.
Default Data Size	Size, in bytes, of the flattened stream of dataspace default values stored in the file. This size includes both static and dynamic default data sub-segments.
Dynamic Default Offset	Offset, in bytes, to the start of dynamic default data, relative to the start of all default data in the file
Dynamic Default Size	Size, in bytes, of dynamic default data. This value always equals the Default Data Size minus the Dynamic Default Offset.
Memory Manager Head	DVA index to the initial head element of the memory manager's linked list of dope vectors.
Memory Manager Tail	DVA index to the tail element of the memory manager's linked list of dope vectors.
Dope Vector Offset	Offset, in bytes, to the initial location of DV data in RAM, relative to the start of the dataspace pool in RAM.

Note that `Initial Size` must be small enough to fit in the 32KB pool along with the program's run-time clump data. Refer to the **Clump Records** section in this document for more information about run-time clump data.

Dataspace

As covered in the **Introduction** section of this document, NXT programs support a rich set of options for storing and arranging run-time data. The dataspace file segment includes all information necessary to initialize the dataspace during program activation as well as the specifications of data types and layout needed during program execution. The following table describes these three sub-segments.

Sub-segment	Description
Dataspace Table of Contents	Compile-time specification of the data types and layout of all user data items.
Default Static Data	Initial values for static dataspace items.
Default Dynamic Data	Initial values for dynamic dataspace items.

All three of these sub-segments are variably-sized according to the needs of a given program. The following sections provide more information about these three sub-segments.

Dataspace Table of Contents

The DSTOC describes the data types and locations of all items in the program's dataspace. The DSTOC is the map the VM uses to locate data in RAM and use the proper data type semantics when operating on that data. Remember that the DSTOC for a particular program is constructed at compile-time and is does not change at run-time.

DSTOC Records

The DSTOC is organized as a statically-sized and constant value array of fixed-length records. Each record is four bytes long and has the following structure:

	DSTOC Record		
Field	Type	Flags	Data Descriptor
Bits	0..7	8..15	16..31

The Type field contains a simple integer type code. The following table describes legal type code values.

Code	Name	Description
0	TC_VOID	Code for unused or placeholder elements
1	TC_UBYTE	Unsigned 8-bit integer
2	TC_SBYTE	Signed 8-bit integer
3	TC_UWORD	Unsigned 16-bit integer
4	TC_SWORD	Signed 16-bit integer
5	TC_ULONG	Unsigned 32-bit integer
6	TC_SLONG	Signed 32-bit integer
7	TC_ARRAY	Array of any sub-type
8	TC_CLUSTER	Cluster of some list of sub-types
9	TC_MUTEX	Mutex data for clump scheduling

The Flags field is used at program initialization time. In NXT firmware 1.03, the DSTOC is parsed while run-time dataspace defaults are being constructed in RAM. A value of `0x01` in the Flags field instructs the firmware to fill a given dataspace item's memory with zero bytes instead of looking elsewhere in the file for default values.

The Data Descriptor field can take on different meanings depending on the data type involved.

Addressable vs. Non-Addressable Records

Remember that bytecode instruction arguments often take the form of dataspace item IDs that identify unique entries in the DSTOC. These dataspace item IDs are the "address" of the dataspace item itself. Even though bytecode instructions never specify the true address of items in RAM, the instruction interpreter needs only the dataspace item ID to locate the data in RAM and treat it properly according to data type.

Bytecode instructions cannot address all DSTOC records. That is, not all indexes are legal to use as arguments. There are two kinds of non-addressable DSTOC entries: TC_VOID entries and sub-type entries of addressable arrays. TC_VOID entries might occur in programs compiled by LEGO MINDSTORMS Software 1.0, but they are not strictly necessary. Arrays must be addressed by their top-level entry. The following section provides more detail about addressing sub-type entries of arrays.

DSTOC Grammar

DSTOC records follow a specific grammar that describes all the possible data types. This grammar is parsed top-down, which means that that complex data types are defined using an ordered list of multiple DSTOC records. The first entry, or lowest index, of an aggregate type definition is the "top-level" record. This entry contains a type code of TC_ARRAY or TC_CLUSTER.

The following three rules describe the DSTOC grammar.

1. Scalar data types use a single DSTOC record. The Data Descriptor of this record specifies an offset to the data in RAM.
2. Arrays use two or more DSTOC records. The Data Descriptor of the first record specifies the offset to the array's dope vector index. Subsequent records specify the complete sub-type of the array elements.
3. Clusters use two or more DSTOC records. The Data Descriptor of the first record specifies the number of elements in the cluster. Subsequent records specify a complete sub-type for each cluster element.

Remember that you can nest aggregate types. If sub-types are arrays or clusters, this grammar is recursively applied until the aggregate type is fully defined.

> **Note:** There is no limit on the number of nesting levels for aggregate types in NXT firmware 1.03, but using deeply-nested data types might lead to unpredictable run-time behavior.

Note that the Data Descriptors of both scalars and arrays specify an unsigned 16-bit offset to data in RAM, but these offsets are interpreted differently for top-level data type records versus the sub-type records owned by arrays. Offsets contained in top-level DSTOC records are relative to the beginning of the dataspace RAM pool, so these offsets are called *dataspace offsets*. Offsets contained in array sub-type records are relative to the beginning of an array element in RAM, so these offsets are called *array data offsets*.

Array data offsets are needed to describe the internal layout of array elements because the memory manager might move arrays at run-time. Furthermore, array elements might also be aggregates themselves. In this situation, the VM first resolves the address of the top-level array elements. The VM then uses the array data offsets to find any nested data.

DSTOC Example

This example illustrates the DSTOC structure and grammar by building an example TOC. In this example, the DSTOC is shown as a vertical table of records, with indexes increasing downward. Addressable records have a grey background. Thick black borders surround records that collectively describe a top-level aggregate type.

For example, imagine a program that uses only one signed 16-bit word. This program uses only the following DSTOC.

DSTOC			
Index (Dataspace Item ID)	**Type**	**Flags**	**Data Descriptor**
0	TC_SWORD	0x00	0x0000

This DSTOC has only one record, and that record is addressable by instructions as dataspace item ID 0. Because this record describes the only data in the dataspace, that data is to be stored at offset 0x0000.

Now consider the same example program with an array of unsigned bytes added to the dataspace. This program now uses the following DSTOC.

DSTOC			
Index (Dataspace Item ID)	**Type**	**Flags**	**Data Descriptor**
0	TC_SWORD	0x00	0x0000
1	TC_ARRAY	0x00	0x0200
2	TC_UBYTE	0x00	0x0000

Here, the DSTOC record at index 1 describes an addressable array with its dope vector index stored at dataspace offset 0x0002. Remember that multi-byte fields like Data Descriptor are listed in little-endian byte order.

The record immediately following a TC_ARRAY record specifies the sub-type. In this case, the record at index 2 specifies that all elements of array 1 are unsigned bytes. Like all sub-type records owned by an array, bytecode instructions cannot address entry 2. Furthermore, remember that the offsets stored in array sub-type data descriptors are relative to the base address of the array. For arrays containing only a simple scalar sub-type, this offset is always 0x0000.

Clusters also require at least two DSTOC entries. Similarly to arrays, the first record specifies the start of a cluster definition with the TC_CLUSTER type code. However, the Data Descriptor field of a cluster contains a count of addressable sub-type entries, not an offset. Subsequent entries then describe the cluster's sub-types.

Consider this same example program with a cluster added. The cluster contains two signed bytes. This program now uses the following DSTOC.

DSTOC			
Index (Dataspace Item ID)	**Type**	**Flags**	**Data Descriptor**
0	TC_SWORD	0x00	0x0000
1	TC_ARRAY	0x00	0x0200
2	TC_UBYTE	0x00	0x0000
3	TC_CLUSTER	0x00	0x0200
4	TC_SBYTE	0x01	0x0400
5	TC_SBYTE	0x01	0x0500

The records at indexes 3–5 describe the two-element cluster. Index 3 specifies the cluster type code and that the next two addressable entries belong to this cluster. Indexes 4 and 5 obey the same rules as any other scalar record. Bytecode instructions can address these records independently. The Data Descriptors of these records specify dataspace offsets, exactly like index 0 does. In this case, the first byte is stored at offset 4 (immediately after the 16-bit DV index at offset 2), and the second byte is stored immediately after that.

The Flags fields for both indexes 4 and 5 are also set to `0x01`. This setting indicates that the example program file does not include explicit default values for these dataspace items. The VM will use the “default default” value of 0 for both items.

In the previous table, notice that the top-level cluster entry, Index 3 is addressable. In other words, instruction arguments can refer to top-level clusters as a whole or to any of their immediately-owned elements. When arguments refer to the whole cluster, the offset of the cluster’s first element is used to find the cluster in RAM.

This example now adds the following nested aggregate types:

1. An array of clusters with two fields: an unsigned byte and a signed word.
2. An array of arrays of unsigned bytes, which you can use to specify an array of strings
3. A cluster containing two unsigned 32-bit longs followed by an array of signed words.

This program now uses the following DSTOC, of which the last 12 records are shown.

DSTOC			
Index (Dataspace Item ID)	**Type**	**Flags**	**Data Descriptor**
. . .			
6	TC_ARRAY	0x00	0x0600
7	TC_CLUSTER	0x00	0x0200
8	TC_UBYTE	0x00	0x0000
9	TC_SWORD	0x00	0x0200
10	TC_ARRAY	0x00	0x0800
11	TC_ARRAY	0x00	0x0000
12	TC_UBYTE	0x00	0x0000
13	TC_CLUSTER	0x00	0x0300
14	TC_ULONG	0x00	0x0C00
15	TC_ULONG	0x00	0x1000
16	TC_ARRAY	0x00	0x1400
17	TC_SWORD	0x00	0x0000

Indexes 6–9 define an array of clusters. There are only 3 bytes of actual data in each (cluster) array element, but 4 bytes are used in RAM because the 16-bit words must be aligned on even addresses. In cases like this, there must be at least one byte “lost” for padding in each array element.

In this example, the padding byte occurs between the TC_UBYTE element and the TC_SWORD element of each cluster. This fact is encoded in the DSTOC via the Data Descriptors of indexes 8 and 9. Index 8 describes only one byte, but the TC_SWORD described by record 9 is stored at offset `0x0002` relative to the beginning of each array element. Remember that the data descriptor fields of array sub-type records specify array data offsets relative to the beginning of each array element. Also, like all array sub-type records, indexes 7–9 are not addressable by bytecode instructions.

Indexes 10–12 define an array of arrays. The DSTOC data type grammar is fairly straightforward, where index 10 defines a top-level addressable array and indexes 11 and 12 specify that each array element is an array of unsigned bytes. The run-time significance of the data descriptor fields is a little less obvious and involves nested usages of dope vectors. Refer to the **Dynamic Data Management** section of this document for information about dope vectors.

The Data Descriptor of index 10 is the offset to the top-level array's DV index, which is stored in the static dataspace. When the VM uses this DV to find the top-level array contents in RAM, it actually finds an array of DVs – one DV for each sub-array. Resolving the address to the actual byte arrays involves one more indirection through the DVA. This principle holds for all sub-arrays nested underneath a top-level addressable array.

Indexes 13–17 define a cluster that contains two 32-bit longs and an array of signed words. Notice that index 13's Data Descriptor specifies that there are three elements in the cluster, but defining the full cluster data type requires four additional DSTOC records. This arrangement signifies that the two-record array definition is "owned" by the cluster definition.

There is one more detail to point out for this DSTOC example. This example happens to have all records sorted by offsets into the static dataspace. Notice that index 16 is the last addressable record, and its Data Descriptor specifies that the array's DV is stored at offset `0x0014`. Because this DSTOC is sorted by offset, we can infer from this record that this program's static dataspace would be 22 bytes long by adding 2 bytes for the DV to the last offset of 0x0014. However, there is no requirement that the DSTOC be sorted by offset. The compiler can arrange the DSTOC in any order, as long as the grammar is preserved and bytecode instruction arguments always refer to addressable records.

Default Values for Static Data

Default values for the static dataspace items are listed immediately after the DSTOC, and the layout of this sub-segment depends entirely on the DSTOC contents.

This sub-segment is best thought of as a flattened stream of default values that obeys the following three rules:

1. Static default values must be packed into the stream in the same order in which their corresponding records occur in the DSTOC.
2. Static default values are tightly packed in the stream. That is, padding is not used to enforce alignment of multi-byte fields. This arrangement is possible because these values are processed as a flat stream of bytes at program activation time and are never accessed during program execution.
3. Not every static dataspace item actually has a default value in the stream. If the corresponding DSTOC record's Flags field contains the value of `0x01`, the RAM associated with that item is automatically initialized to 0.

A corollary to these rules is that the size of this default value stream must be less than or equal to the Static Size field listed in the dataspace header. This stream will be smaller than Static Size for a vast majority of programs, though, because rules 2 and 3 mean that the static default stream is often compressed by ignoring alignment and zero default values.

Furthermore, the actual size of the static default stream must be consistent with the dataspace header fields Default Data Size and Dynamic Default Offset. Because Default Data Size includes both static and dynamic defaults and the dynamic defaults are stored immediately after the static default stream, the Dynamic Default Offset field also specifies the static default stream's size.

Remember that each addressable array record in the DSTOC uses its Data Descriptor field to refer to a dope vector index stored in the static dataspace. These DV indexes must always have default values in the default data stream. In other words, the DSTOC ignores the Flags field of array records.

Note: Mutex record default data resides in the static default stream, but has one special restriction: the 32 bits reserved for each mutex record must always have a default value of `0xFFFFFFFF`, which signifies that the mutex is uninitialized. Mutex record values are initialized automatically at program run-time.

Default Values for Dynamic Data

Default values for the dynamic dataspace are handled differently from the static dataspace defaults. First, remember that only arrays are stored in the dynamic dataspace. As such, think of the dynamic default stream as a collection of default arrays, including default values for all of their initial elements. This collection of arrays also includes the default dope vector array. Refer to the **Default Dope Vectors** section of this document for more information about this array.

The second difference is that the dynamic default stream is a direct image of all initial values in the program's dynamic dataspace. In other words, the dynamic default stream must be formatted such that it can be copied directly into the dynamic dataspace segment of RAM without modification.

This formatting means that this default stream must include any necessary padding for any data types it includes. Because all arrays must start on 4-byte boundaries, it is common to have some padding bytes between individual array entries. The compiler must also provide padding as needed within array data. For example, consider an array of 3-byte clusters that each contain a 16-bit word and a byte. This array would require a padding byte after each element to keep the 16-bit words aligned on the appropriate boundaries.

It is important to distinguish between alignment requirements within the dynamic default stream from the requirements on the stream itself. There are no requirements that the default stream start on any particular alignment boundary within the file, since it is copied as an arbitrary stream of bytes. As discussed above, though, alignment rules must be followed within the stream such that all internal data fields are aligned relative to the start of the stream.

Default Dope Vectors

Remember that the dope vector array (DVA) is itself a dynamic array. Refer to the **Dynamic Data Management** section of this document for more information about dope vectors. Each dope vector in the DVA has a default value, and these default dope vectors must reside inside the dynamic default stream.

For simplicity, default dope vector data should be placed at the front of the dynamic default stream, but this placement is not required. As long as the Dope Vector Offset field of the Dataspace Header can be used to find the DVA in RAM, the DVA can exist anywhere within the dynamic data.

Each dope vector consists of five fields, and each field is a 2-byte unsigned word. The following table describes these fields.

Field	Description
Offset	Offset to the start of this array's data in RAM relative to the beginning of all user data.
Element Size	The size, in bytes, of each array element.
Element Count	The number of elements currently contained in this array.
Back Pointer	An unused placeholder in NXT firmware 1.03.
Link Index	The index of the next DV in the memory manager's linked list.

Remember that the compiler is responsible for calculating the initial layout of all data in RAM. This calculation includes the Dope Vector Offset field in the dataspace header and each individual dope vector's `Offset` field. All offsets used by the memory manager are relative to the start of the dataspace pool in RAM.

The `Element Size` field specifies the size of each array element in the array described by a dope vector. This size must include any padding required by cluster sub-types. If an array contains sub-arrays, `Element Size` is always two because the element actually stored in RAM is a 2-byte dope vector index.

Dope vectors describing empty arrays have an `Offset` equal to `0xFFFF` and an `Element Count` equal to `0`.

> **Note:** The default value of the `Back Pointer` field is ignored in NXT firmware 1.03. However, you must include these two bytes because the dynamic default stream is copied directly into RAM.

Remember that the DVA contains its own dope vector as its first entry. This special DV, or the *root dope vector*, is required in every program. The `Offset` field of the root DV must have a default value equal to the Dope Vector Offset in the dataspace header and an `Element Size` of 10. This value of 10 is equal to the fixed size of each DV entry in the DVA. If the program does not contain any actual user arrays, the DVA must contain only the root DV entry, with the `Element Count` equal to 1.

The Link Index field is used to initialize this linked-list, so this field must contain a valid index into the DVA for all DVs, with three exceptions:

1. The root DV is not included in the memory manager's linked-list, so its Link Index field is ignored. It is recommended that the compiler always set this field to `0xFFFF`.
2. Because the root DV has a DVA index of 0, no DV can contain a Link Index of 0.
3. The final dope vector in the linked-list must have a `Link Index` value of `0xFFFF`, which terminates the list.

The `Link Index` values of all non-root DVs must be arranged such that traversing the linked-list yields ascending values for the Offset field. Also remember that the dataspace header contains two fields which specify the head and tail indices of this linked-list: `Memory Manager Head` and `Memory Manager Tail`. These two fields must be consistent with the link indexes in the default DVA.

Clump Records

The Clump Record segment describes how the codespace is partitioned into chunks of instructions, or clumps, and how these clumps should be scheduled at run-time. The total number of clump records is recorded in Clump Count field in the file header. Refer to the **File Header** section of this document for more information about this field.

The following table describes the fields in the Clump Record segment.

Field	Byte Count	Description
Fire Count	1	Unsigned byte that specifies when this clump is ready to run.
Dependent Count	1	Unsigned byte that specifies the number of clumps in the file that are dependent on this clump.
Code Start Offset	2	Unsigned word that specifies the offset into the codespace at which this clump's instructions begin.
Dependent List	variable	Array of unsigned bytes that specifies the indexes of all clumps that are dependent on this one.

In practice, the dependent lists are packed in a separate sub-segment of the file immediately after all of the fixed-length clump records, or the first four bytes. This practice keeps the fixed-length records aligned in memory while reducing wasted space.

So, for a program consisting of *n* code clumps, the clump record segment will consist of *n* four-byte records (Fire Count, Dependent Count, and Code Start Offset), followed by the packed list of all dependents. The Dependent Count field might be 0 for any or all clumps, so the full dependent list might actually be empty, e.g., have a length of 0. Dependent lists for each clump are packed together in the same order as their corresponding fixed-length clump records are listed in the file.

When a program is initialized, the file's clump record data is inflated into an array of run-time clump records in RAM. This array contains bookkeeping data internal to the VM. These run-time clump records remain in the same order as they occur in the file, and each entry in any given clump's dependent list is an index into this array.

The NXT firmware 1.03 permits a maximum of 255 clumps per program. Furthermore, each clump requires 15 bytes of run-time clump data in the program's 32KB RAM pool. This clump data must fit in the pool alongside all static and dynamic data.

Codespace

This section provides information about the Codespace segment of executable files. The codespace consists of 16-bit code words which are interpreted as variable-length instructions. The codespace is a constant stream of instructions, that is, there must not be any padding bytes or other non-instruction data in between instructions. Remember that the VM logically divides this stream into clumps via the data in the Clump Records.

> **Note**: In order to be consistent with how bytes are stored in `.RXE` files, the structure of all code words is presented here in little-endian byte order. Also, individual bits are identified from left to right, that is, bit 0 is the most significant bit of the least significant byte.

Instructions always consist of one or more code words, but there are two types of encoding: long instructions and short instructions. You can use both types together in the codespace. Regardless of the encoding type, bits 12..15 of an instruction's first code word is always reserved for the Flags field.

Bit 12 determines the type of encoding. If this bit is `0`, the instruction uses long encoding. If this bit is `1`, the instruction uses short encoding.

Long Instruction Encoding

The long encoding is the simplest instruction format, and most classes of instructions support only long encoding.

The first word of a given instruction contains the opcode, size, and flags. The second and any subsequent words are instruction-specific arguments.

For example, the following table describes the structure of a two-argument instruction.

	Word 1			Word 2	Word 3
Field	Opcode	Size	Flags	Argument 1	Argument 2
Bits	0..7	8..11	12..15	0..15	0..15

In short, a long-encoded instruction consists of one opcode word and one or more argument words.

Opcode

The opcode is an unsigned byte that uniquely identifies what class of instruction this is. Refer to the **Instruction Reference** section for information about individual opcodes, or instructions.

Size

For most instructions, four bits of Word 1 are used for the total instruction size in bytes. This includes the opcode word and all arguments. For example, the size of the two-argument example in the above table is 6. This design allows efficient traversal of the codespace at run-time. Most instructions fit into a four bit size specification. Larger instructions use a reserved size field value of `0xE`, and the actual size is stored as the first 16-bit argument. `OP_ARRBUILD` is an example of an instruction that uses this encoding.

Flags

Instructions involving comparison of two values include a comparison code in the lower three bits of the Flags field. Refer to the **Comparison Operations** section of this document for information about supported comparison codes.

Short Instruction Encoding

Some instances of common instructions might use an alternate encoding to save memory space. For example, `OP_MOV` can be encoded in two words (instead of three in the long encoding) if its Source and Destination arguments are close to each other in the DSTOC.

Using the short encoding depends on the compiler's ability to arrange the dataspace such that either:

- The sole argument of a one-argument instruction will fit into a single byte.

OR

- The first argument of a two-argument instruction can be expressed as a signed-byte offset from this instruction's second argument.

For each instruction that meets one of these criteria, that instruction's first code word is reorganized. The Size field remains the same as the long encoding type, but the Flags and Opcode fields are used differently from that type.

For a short instruction, bit 12 is set to `1` and the remaining three bits of Flags are set to a special short opcode. This reuse of the Flags field means that instructions requiring comparison codes may not be optimized in this way. The Opcode field is replaced with a single byte argument that is interpreted differently depending on which short encoding variant is being used.

For instructions with only one argument, the short encoding fits into only one code word. The following table describes the structure of this word.

	Word 1			
Field	Argument	Size	1	Op
Bits	0..7	8..11	12	13..15

In the above table, the instruction's sole argument fits into the 8-bit field that the long encoding uses for the Opcode field. At run-time, the value in this 8-bit argument is treated as if it were a normal 16-bit unsigned code word.

For instructions with two arguments, the short encoding has the following structure:

	Word 1				**Word 2**
Field	Offset	Size	1	Op	Argument 2
Bits	0..7	8..11	12	13..15	0..15

For two argument short instructions, the true value of the first 16-bit unsigned argument, *Argument 1*, is calculated at run-time using this simple equation:

Argument 1 = Argument 2 + Offset

Remember that Offset is a signed 8-bit value, which means the value of Argument 1 must fall within a range +127 or -128 from Argument 2.

The table below lists supported values for the Op field, along with the corresponding instruction. Refer to the **Instruction Reference** section of this document for information about these instructions.

Op Value	Name	Instruction
0	SHORT_OP_MOV	OP_MOV
1	SHORT_OP_ACQUIRE	OP_ACQUIRE
2	SHORT_OP_RELEASE	OP_RELEASE
3	SHORT_OP_SUBCALL	OP_SUBCALL

Argument Formats

Regardless of long or short encoding, instruction arguments are all ultimately resolved as 16-bit values. A given argument word might be interpreted as either a reference to an item in the dataspace or as an immediate value, depending on the particular instruction.

Dataspace Item IDs

Most instructions operate exclusively on typed data stored in the program's dataspace. Arguments referring to these dataspace items take the form of an unsigned 14-bit index into the DSTOC stored in a 16-bit code word. These indexes uniquely identify items in the dataspace, so they are called *dataspace item IDs.*

> **Note:** The two most significant bits of dataspace item ID code words are reserved for internal use by the NXT firmware 1.03. This means that programs are limited to a maximum of 16,383 addressable dataspace items.

These indexes can refer to DSTOC entries with any of the legal data types: scalar, array, cluster, or mutex. The instruction interpreter indexes the DSTOC table, resolves the data's actual location in RAM, and then takes the appropriate actions according to the bytecode instruction and data types involved.

Remember that not all indexes in the DSTOC can be used as legal instruction arguments. Specifically, bytecode instructions cannot refer to any sub-type entries associated with a top-level array. Instead, top-level arrays must be indexed or otherwise modified to access their sub-elements. Refer to the **Addressable Records vs. Non-Addressable Records** section of this document for more information about these restrictions.

There is one special reserved dataspace item argument, NOT_A_DS_ID, signified by the value 0xFFFF. NOT_A_DS_ID is never a valid index into the DSTOC. Some instructions allow their arguments to take this value and assign special meaning to it. This behavior may be thought of in similar terms to the use of default function arguments in other languages.

For example, you can use NOT_A_DS_ID for the Index argument of the array-indexing instruction OP_INDEX. This instruction then substitutes the default value of 0 instead of looking in the dataspace for an index. In this situation, using NOT_A_DS_ID avoids the overhead of storing common default values in the dataspace.

Refer to the *Instruction Reference* section of this document for details on instruction-specific behavior, including how certain instructions use NOT_A_DS_ID.

Refer to the **Run-time Data**, **Polymorphic Instructions**, and **Dataspace Table of Contents** sections of this document for information about general dataspace structure, polymorphic behavior, and the DSTOC.

Immediate Values

Some instructions take arguments in which the value used at run-time is stored directly in the code words rather than using the argument code words as references to dataspace items. These arguments are called *immediate values*.

Immediate values take on very instruction-specific meanings. The Instruction Reference section of this document lists these meanings. Examples of instructions that use immediate values include `OP_JMP`, `OP_SYSCALL`, and `OP_SETIN`. All variable-length instructions, such as `OP_ARRBUILD`, also use an immediate argument to specify the size of the instruction.

Clump Termination

All clumps must include at least one clump termination instruction. Use the `OP_FINCLUMP` or `OP_FINCLUMPIMMED` instructions to terminate clumps that have dependent clumps. Use `OP_SUBRET` to terminate subroutine clumps. Refer to the **Bytecode Scheduling** section of this document for information about how the VM schedules clumps.

Clump termination instructions are typically the last instruction in a given clump.

EXAMPLE PROGRAM: ADDING SCALAR NUMBERS

The following example analyzes the contents of actual executable `.RXE` file as it might appear when viewed via a PC hex editor. This example assumes you are familiar with the file segments described earlier in this document. Refer to the **File Header**, **Dataspace**, **Clump Records**, and **Codespace** sections for more information about the file segments.

All file data is presented in tables of hexadecimal 16-bit code words in little-endian byte order. The left-most column shows the hexadecimal offset of the first byte for a given line, or that line's base offset. Since each data column represents two bytes, the offset of a given word is computed by adding two to the line's base offset for each word to its left.

Unless otherwise noted, all offsets in this example are zero-based and given in bytes.

Note that `.RXE` files are packed binary files, meaning that organizing them into 16-byte lines is done only for the purposes of this document – at run-time, the firmware treats files as arbitrarily-addressable byte streams.

Example Code

This example illustrates a simple "one line" program which adds two integer source numbers and stores the result into a destination in the dataspace. A pseudocode representation of this program looks like this:

```
Destination is a signed 32-bit integer with an initial value of 0.
Source1 is a signed 32-bit integer with a value of 5000.
Source2 is an unsigned 8-bit integer with a value of 1.

Destination = Source1 + Source2
```

The entire file contents (84 bytes) are listed in this table:

Offset	Data							
0000000	4D69	6E64	7374	6F72	6D73	4E58	5400	0005
0000010	0300	1600	0C00	0F00	0500	0A00	0000	0000
0000020	0C00	0100	0700	0601	0000	0600	0400	0100
0000030	0800	8813	0000	010C	000A	0001	00FF	FFFF
0000040	FFFF	0000	0000	0080	0000	0100	0200	2A60
0000050	FFFF	FFFF						

Header Segment

The fixed-length header fields take up the first 38 bytes of the file. This segment is repeated in the table below:

Offset	Data							
0000000	4D69	6E64	7374	6F72	6D73	4E58	5400	0005
0000010	0300	1600	0C00	0F00	0500	0A00	0000	0000
0000020	0C00	0100	0700					

The first line, of course, is the format string. The first fourteen bytes are ASCII characters, spelling out 'MindstormsNXT' (followed by a null padding byte). The last two bytes contain the file format version, 0x0005 (big-endian). Remember that all RXE files supported by NXT firmware version 1.03 share these exact 16 bytes in common.

Starting from offset 0x10, the next nine words describe the count, size, and arrangement of the program's dataspace items. The last two words hold the 'Clump Count' and 'Code Word Count' fields.

Converting these header words to decimal fields, we can see that:

- This program has 3 addressable dataspace items.
- The initial dataspace size in RAM will be 22 bytes.
- Of those 22 bytes of RAM, 12 belong to statically-sized dataspace items.
- There are 15 bytes of default data in the file.
- Dynamic data defaults start at byte offset 5, relative to the first byte of all default data.
- There are 10 bytes of flattened dynamic default data in the file.
- The memory manager's head and tail indices both default to 0.
- Dope vector data will start at byte offset 12, relative to the first byte of the dataspace in RAM.
- This program has 1 clump.
- The codespace of this program contains 7 code words.

This particular example does not actually use any dynamically-sized data, so only the fields having to do with statically-sized data are actually used.

Dataspace Segment

The dataspace segment begins at offset 0x26 and takes up 27 bytes.

Offset	Data							
0000020				0601	0000	0600	0400	0100
0000030	0800	8813	0000	010C	000A	0001	00FF	FFFF
0000040	FF**FF**							

First, note the byte shown in bold at offset 0x41. Because all segments must start on a 16-bit boundary and this segment has an odd size, a padding byte is needed here. The value of the padding byte is unimportant – MINDSTORMS NXT Software 1.0 happens to use 0xFF.

The dataspace table-of-contents takes up the first 6 words of this segment, and is comprised of a 4-byte record for each of the 3 dataspace items. Decoding these records yields the following table:

DSTOC			
Index	**Type**	**Flags**	**Data Descriptor**
0	0x06 (TC_SLONG)	0x01	0x0000
1	0x06 (TC_SLONG)	0x00	0x0004
2	0x01 (TC_UBYTE)	0x00	0x0008

Note that all of the data items in this example are simple scalar numbers and all records in the DSTOC are addressable by bytecode arguments.

The 'Flags' field for item 0 contains 0x01. This means that its default value is zero, so there was no need to store default data in the file. When the run-time dataspace is constructed in RAM, the appropriate bytes will automatically be initialized to zero.

For items 1 and 2, 'Flags' does not have the zero-default bit set, so we must look in the next file sub-segment for default values. Item 1's default value is four bytes long (as item 1 is an SLONG) and can be found at file offset 0x32. Converting the little-endian SLONG value of 0x88130000 to decimal yields what we expect: 5000. Similarly, item 2's single byte default data is at offset 0x36 and contains the decimal value 1.

Remember that for scalar dataspace items, the data descriptor specifies a given item's address in RAM relative to the dataspace base address. Since the size of each data item is determined solely by the type code, the DSTOC entry is all that is needed for the VM to access static scalar data.

The 10 bytes of dynamic default data at offset 0x37 represent an empty default dope vector array. This program does not use any dynamic data, but remember that at least one default dope vector is always required. Refer to the **Default Dope Vectors** section of this document for more information.

Clump Record Segment

The clump record segment of this file starts at offset 0x42. This program contains only one clump, so the clump record segment is exceedingly simple.

Offset	Data			
0000040		0000	0000	

Decoding this clump record data, we see that this program's only clump has a 'Fire Count' field of zero, no dependents, and its code is (of course) right at the front of the codespace.

Fire Count	Dependent Count	Code Start Index
0x00	0x00	0x0000

Codespace Segment

The remaining words of this file, starting at offset 0x46, are comprised of two bytecode instructions.

Offset	Data						
0000040			0080	0000	0100	0200	2A60
0000050	FFFF	FFFF					

When examining instructions in the little-endian file format, it is easiest to examine code words from right to left.

Decoding the first code word, we see that:

- This instruction uses the long encoding – bit 12 is '0'.
- The instruction size is 8 bytes, including this code word.
- The opcode is `0x00`, or `OP_ADD`.

Putting these facts together and converting the three arguments to big-endian notation yields a more human-readable instruction:

```
OP_ADD 0x0000, 0x0001, 0x0002
```

This instruction performs the task this program set out to accomplish: add Source1 (0x0001) to Source2 (0x0002) and store the result in the Destination (0x0000).

Applying the same process to the remaining code words yields the next (and last) instruction:

```
OP_FINCLUMP 0xFFFF, 0xFFFF
```

This instruction terminates the program's only clump. The value 0xFFFF is used as a special flag value for both arguments because this clump has no dependent clumps.

INSTRUCTION REFERENCE

All supported instructions are listed in this section, sorted by instruction class. There are six instruction classes:

- Math
- Logic
- Comparison
- Data Manipulation
- Control Flow
- System I/O

Notes specific to particular classes of instructions are included in the sub-sections below.

In general, instruction descriptions include a summary of the instruction's actions followed by any applicable notes or restrictions. Unless otherwise noted, instruction arguments must be valid dataspace item IDs (see **Argument Formats**).

> **Note:** Compiler authors should be careful to ensure that instruction arguments are valid for each given instruction. Invalid dataspace item IDs and/or data types may result in a fatal run-time error condition, causing the program to halt and display of **`File Error`** on the NXT brick's screen.

Math Instructions

Use math instructions to perform mathematical operations on items in the dataspace.

All math instructions require valid dataspace item IDs for all arguments and are polymorphic on input data types. All inputs and outputs must follow the data type compatibility rules described in the **Polymorphic Instructions** section. Instructions with two inputs accept any compatible combination of input data types.

Instruction: `OP_ADD`
Opcode: `0x00`
Arguments: `Destination`, `Source1`, `Source2`
Description: Add `Source1` and `Source2`; and store result into `Destination`.

Instruction: `OP_SUB`
Opcode: `0x01`
Arguments: `Destination`, `Source1`, `Source2`
Description: Subtract `Source2` from `Source1`; store result into `Destination`.

Instruction: `OP_NEG`
Opcode: `0x02`
Arguments: `Destination`, `Source`
Description: Negate `Source` by using the two's complement method; store result into `Destination`.

Instruction: `OP_MUL`
Opcode: `0x03`
Arguments: `Destination`, `Source1`, `Source2`
Description: Multiply `Source1` by `Source2`; store result into `Destination`.

Instruction: OP_DIV
Opcode: 0x04
Arguments: Destination, Source1, Source2
Description: Divide Source1 by Source2; store result into Destination. If Source2 is 0, store 0 into Destination.

Instruction: OP_MOD
Opcode: 0x05
Arguments: Destination, Source1, Source2
Description: Divide Source1by Source2; store remainder into Destination. If Source2 is 0, store Source1 into Destination.

Logic Instructions

Use logic instructions to perform logical (binary) operations on items in the dataspace.

All logic instructions require valid dataspace item IDs for all arguments and are polymorphic on input data types. All inputs and outputs must follow the data type compatibility rules described in the **Polymorphic Instructions** section of this document. Instructions with two inputs accept any compatible combination of input data types.

Instruction: `OP_AND`
Opcode: `0x06`
Arguments: `Destination`, `Source1`, `Source2`
Description: Perform bitwise AND operation on `Source1` and `Source2`; store result into `Destination`.

Instruction: `OP_OR`
Opcode: `0x07`
Arguments: `Destination`, `Source1`, `Source2`
Description: Perform bitwise OR operation on `Source1` and `Source2`; store result into `Destination`.

Instruction: `OP_XOR`
Opcode: `0x08`
Arguments: `Destination`, `Source1`, `Source2`
Description: Perform bitwise XOR operation on `Source1` and `Source2`; store result into `Destination`.

Instruction: `OP_NOT`
Opcode: `0x09`
Arguments: `Destination`, `Source`
Description: Perform Boolean NOT operation on `Source`; store *TRUE* or *FALSE* (`1` or `0`) into `Destination`.

Comparison Instructions

Use comparison instructions to test dataspace values against each other, or against 0, and produce a Boolean result.

The exact kind of comparison performed depends on the instructions comparison code, which is stored in the Flags field, and the data types involved. The following table describes the legal comparison codes.

Comparison Type		Code
<	(less than)	0
>	(greater than)	1
<=	(less than or equal to)	2
>=	(greater than or equal to)	3
==	(equal to)	4
!=	(not equal to)	5

Comparison instructions can perform aggregate comparison or element comparison depending on the data types involved. In other words, the Boolean results described below may have a scalar or aggregate data type, and the instruction interpreter will apply the comparison code accordingly. See the **Polymorphic Comparisons** section for more details.

All comparison scenarios can be handled by one instruction, `OP_CMP`, but the NXT firmware 1.03 also includes an optimized variant, `OP_TST`. This instruction requires only one input because comparison against zero is implied. This saves the space needed to store otherwise useless dataspace items filled with zeroes.

Both comparison instructions require valid dataspace item IDs for all arguments and are polymorphic on input data types. All inputs and outputs must follow the data type compatibility rules described in the **Polymorphic Instructions** section. Output data types are further restricted by the rules described in the **Polymorphic Comparisons** section

Instruction: `OP_CMP`
Opcode: `0x11`
Arguments: `Destination`, `Source1`, `Source2`
Description: Compare `Source1` to `Source2` according to comparison code; store Boolean result into `Destination`.

The data types of all three arguments must be consistent with the rules described in the **Polymorphic Comparisons** section of this document.

Instruction: `OP_TST`
Opcode: `0x12`
Arguments: `Destination`, `Source`
Description: Compare `Source` to 0 according to comparison code; store Boolean result into `Destination`.

If `Source` has a scalar data type, this instruction compares `Source` against 0 to produce a single Boolean result.
If `Source` has an aggregate data type, this instruction compares each element of `Source` against 0 to produce an aggregate of Boolean results. Refer to the **Polymorphic Comparisons** section of this document for information about element comparison vs. aggregate comparison.

Data Manipulation Instructions

Use data manipulation instructions to move and otherwise manipulate items in the program's dataspace.

All array indices are zero-based; all instructions accepting an "Index" scalar input argument treat index value `0` as the first element of an array.

> **Note:** Compiler authors should take care to carefully distinguish between text strings and regular unsigned byte arrays. The string manipulation instructions below assume that text strings arguments include a null-termination byte, but the regular array manipulation instructions do not. The VM does not distinguish between these data types, so it is important that the compiler enforces the distinction.

Instruction: `OP_INDEX`
Opcode: `0x15`
Arguments: `Destination`, `Source`, `Index`
Description: Index array at `Source` by `Index`; store element into `Destination`.

The data type of `Destination` and sub-type of `Source` must be compatible.
`Source` must refer to a top-level array of any sub-type.
`Index` must refer to a scalar number or use `NOT_A_DS_ID`.
If `Index` is `NOT_A_DS_ID`, this instruction uses a default value of `0`.

Instruction: `OP_REPLACE`
Opcode: `0x16`
Arguments: `Destination`, `Source`, `Index`, `NewVal`
Description: Replace subset of array `Source`, starting at `Index`, with the contents of `NewVal`; store resulting array into `Destination`.

The data type of `Destination` and sub-type of `Source` must be compatible.
`Index` must refer to a scalar number or use `NOT_A_DS_ID`.
If `Index` is `NOT_A_DS_ID`, this instruction uses a default value of `0`.
`NewVal` can be any array compatible with `Source` or any data type compatible with the sub-type of `Source`.

If `Index` is out of range for `Source`, `NewVal` will be ignored and `Destination` will be replaced with the contents of `Source`.

Instruction: `OP_ARRSIZE`
Opcode: `0x17`
Arguments: `Destination`, `Source`
Description: Store count of elements in array `Source` into scalar `Destination`.

Instruction: `OP_ARRBUILD`
Opcode: `0x18`
Arguments: `InstrSize`, `Destination`, `Source1`, `Source2`, ... `SourceN`
Description: Build array `Destination` out of one or more `Source` items. Source items may be any data type, including arrays. Array sources will be concatenated to form the destination array.

`InstrSize` is an immediate value that specifies the total instruction size, in bytes.

Instruction: OP_ARRSUBSET
Opcode: 0x19
Arguments: Destination, Source, Index, Count
Description: Store subset of array at Source into array at Destination, starting at Index and including Count elements.

Index must refer to a scalar number or use NOT_A_DS_ID.
If Index is NOT_A_DS_ID, this instruction uses a default value of 0.

Count must be a scalar number or use NOT_A_DS_ID.
If Count is NOT_A_DS_ID, the source element at Index and all elements past it are copied into Destination. That is, the effective value of Count in the following way:

Count = (*length of* Source) – Index

Instruction: OP_ARRINIT
Opcode: 0x1A
Arguments: Destination, NewVal, Count
Description: Initialize array Destination with Count copies of NewVal.

If Count is 0, the Destination array will be empty.
If Count is NOT_A_DS_ID, this instruction uses a default value of 0.

Instruction: OP_MOV
Opcode: 0x1B
Arguments: Destination, Source
Description: Store copy of Source into Destination.

Instruction: OP_SET
Opcode: 0x1C
Arguments: Destination, Immediate
Description: Set scalar Destination to 16-bit value Immediate.

Destination must have a scalar data type.
The literal value of the 16-bit argument Immediate is read as an unsigned word and stored directly into Destination.

Instruction: OP_FLATTEN
Opcode: 0x1D
Arguments: Destination, Source
Description: Flatten data at Source into byte array (string) at Destination.

Instruction: OP_UNFLATTEN
Opcode: 0x1E
Arguments: Destination, Error, Source, Default
Description: Unflatten data from byte array (string) at Source and store into Destination.

Default must match the flattened data type exactly, including array sizes. If not, Error will be set to *TRUE* and Destination will contain a copy of Default.

Instruction: OP_NUMTOSTRING
Opcode: 0x1F

Arguments: Destination, Source
Description: Convert integer at Source into decimal text (string); store result into Destination.

Instruction: OP_STRINGTONUM
Opcode: 0x20
Arguments: Destination, IndexPast, Source, Index, Default
Description: Convert decimal number in text string at Source into an integer number; store result into Destination. Source string may contain multiple numbers and/or non-numeric characters; Index leading characters will be skipped and the integer value at Default will be used if no valid integer is found in Source. If a valid integer is found in Source, the index to the next character past the integer is stored in IndexPast.

Index must refer to a scalar number or use NOT_A_DS_ID.
If Index is NOT_A_DS_ID, this instruction uses a default value of 0.

Default must be a scalar number or use NOT_A_DS_ID.
If Default is NOT_A_DS_ID, a default value of 0 is used.

Instruction: OP_STRCAT
Opcode: 0x21
Arguments: InstrSize, Destination, Source1, Source2, ... SourceN
Description: Concatenate one or more source strings; store result into Destination.

InstrSize is an immediate value specifying the total instruction size in bytes. Destination and each Source argument must be a text string.

Instruction: OP_STRSUBSET
Opcode: 0x22
Arguments: Destination, Source, Index, Count
Description: Store subset of string at Source into string at Destination, starting at Index and including Count characters.

Index must refer to a scalar number or use NOT_A_DS_ID.
If Index is NOT_A_DS_ID, this instruction uses a default value of 0.

Count must be a scalar number or use NOT_A_DS_ID.
If Count is NOT_A_DS_ID, the source element at Index and all elements past it are copied into Destination. That is, the effective value of Count in the following way:

Count = (*length of* Source) – Index

Instruction: OP_STRTOBYTEARR
Opcode: 0x23
Arguments: Destination, Source
Description: Convert string at Source into an unsigned byte array; store result into Destination. This instruction removes the NULL terminator at the end of the array.

Instruction: OP_BYTEARRTOSTR
Opcode: 0x24
Arguments: Destination, Source
Description: Convert unsigned byte array at Source into string; store result into Destination. This instruction adds the NULL terminator to the end of the array.

Control Flow Instructions

Use control flow instructions to affect how the VM schedules clumps and to branch to new instructions within clumps.

Remember that all instructions in a program are “owned” by a clump. Control flow instructions are used to affect how the VM executes the clump owning those instructions. Thus, in the descriptions of control flow instructions, the phrase “this clump” refers to the clump that owns a particular instruction.

Likewise, “this clump’s program counter” refers to the current instruction index that the VM maintains for each clump. The instructions `OP_JMP`, `OP_BRCMP`, and `OP_BRTST` all modify their owning clump’s program counter to specify which of the clump’s instructions should execute next.

`OP_BRCMP` and `OP_BRTST` conditionally modify their owning clumps program counter only if their arguments and comparison code produce a *TRUE* result. Comparisons performed by these instructions follow the same rules as `OP_CMP` and `OP_TST`.

The instructions `OP_FINCLUMP`, `OP_FINCLUMPIMMED`, `OP_SUBCALL`, and `OP_SUBRET` affect clump execution in a different way. These instructions either reset or suspend the current clump and return control to the VM’s main bytecode scheduling algorithm.

Instruction: `OP_JMP`
Opcode: `0x25`
Arguments: `Offset`
Description: Adjust this clumps program counter by immediate value `Offset` (signed word).

Instruction: `OP_BRCMP`
Opcode: `0x26`
Arguments: `Offset`, `Source1`, `Source2`
Description: Compare `Source1` to `Source2` according to comparison code; if *TRUE*, adjust this clump’s program counter by the immediate value `Offset` (signed word).

Instruction: `OP_BRTST`
Opcode: `0x27`
Arguments: `Offset`, `Source`
Description: Compare `Source1` to zero according to comparison code; if *TRUE*, adjust clump program counter by the immediate value `Offset` (signed word).

Instruction: `OP_STOP`
Opcode: `0x29`
Arguments: `Confirm`
Description: Abort the currently running program if Boolean value in `Confirm` is *TRUE* (non-zero).

If `Confirm` is `NOT_A_DS_ID`, the default value is *TRUE*.

Note that this instruction aborts all clumps and causes the program to leave RAM immediately.

Instruction: `OP_FINCLUMP`
Opcode: `0x2A`
Arguments: `Start`, `End`

Description: Finish execution of this clump.

If immediate values `Start` and `End` are positive integers, they are used to index this clump's list of dependents, conditionally scheduling each one within the specified range.

If immediate values `Start` and `End` are negative, they are ignored. No other clumps are scheduled.

Instruction: `OP_FINCLUMPIMMED`
Opcode: `0x2B`
Arguments: `ClumpID`
Description: Finish execution of this clump using immediate value `ClumpID` to conditionally schedule one target clump.

Instruction: `OP_ACQUIRE`
Opcode: `0x2C`
Arguments: `MutexID`
Description: Acquire mutex record at dataspace item `MutexID`. If the mutex is already reserved, place this clump on wait queue for the specified mutex.

Instruction: `OP_RELEASE`
Opcode: `0x2D`
Arguments: `MutexID`
Description: Release mutex record at dataspace item `MutexID`. If the wait queue is not empty, the next clump in line will automatically acquire the specified mutex and resume execution.

Instruction: `OP_SUBCALL`
Opcode: `0x2E`
Arguments: `Subroutine`, `CallerID`
Description: Call clump specified by immediate clump ID `Subroutine` and suspend caller (this clump); save caller's clump ID at `CallerID`.

Instruction: `OP_SUBRET`
Opcode: `0x2F`
Arguments: `CallerID`
Description: Return from subroutine, resuming clump specified by `CallerID` (see `OP_SUBCALL`).

System I/O Instructions

Use system I/O instructions to interact with the NXT brick's built-in I/O devices and other system services.

Refer to the **Instruction Reference Appendix** section of this document for the valid PropID and SysCallID values, including legal data types.

Instruction: `OP_SYSCALL`
Opcode: `0x28`
Arguments: `SyscallID`, `ParmCluster`
Description: Invoke system call method matching immediate value `SyscallID`, with call-specific parameter data included in the cluster referenced by `ParmCluster`.

Refer to the **System Call Methods** section of this document for valid values of `SyscallID` and system call descriptions.

Instruction: `OP_SETIN`
Opcode: `0x30`
Arguments: `Source`, `Port`, `PropID`
Description: Set input configuration property specified by `Port` and `PropID` to the value at `Source`. `Port` and `Source` must be integers in the dataspace, while `PropID` must be an immediate value.

Refer to the **Input Port Configuration Properties** section of this document for valid values of for `PropID`.

Instruction: `OP_SETOUT`
Opcode: `0x31`
Arguments: `InstrSize`, `Port/PortList`, `PropID1, Source1, … PropIDN, SourceN`
Description: Set one or more configuration properties on one or more output ports. Configuration properties and source values are specified via a series of `PropID-Source` tuples.

The first argument `InstrSize` is an immediate value specifying the total instruction size in bytes.

The second argument can refer to either a single integer `Port` specifier or an integer array `PortList`. If you use a `PortList` array, each `PropID-Source` tuple is applied to each port in the array. The order of port specifiers in the list is unimportant. Duplicate port specifiers are permitted in the list, but doing so is redundant.

Source values are written to the corresponding configuration properties in the order that they occur in the list of `PropID-Source` tuples. Multiple tuples may specify the same `PropID`, but the last tuple in the list will override any earlier tuples with the same `PropID`.

Refer to the **Output Port Configuration Properties** section of this document for valid values of for `PropID`.

Instruction: `OP_GETIN`
Opcode: `0x32`
Arguments: `Destination`, `Port`, `PropID`

Description: Set `Destination` to the value of input configuration property specified by `Port` and `PropID`. `Port` and `Destination` must be integers in the dataspace, while `PropID` must be an immediate value.

Refer to the **Input Port Configuration Properties** section of this document for valid values of for `PropID`.

Instruction: `OP_GETOUT`
Opcode: `0x33`
Arguments: `Destination`, `Port`, `PropID`
Description: Set `Destination` to the value of output configuration property specified by `Port` and `PropID`. `Port` and `Destination` must be integers in the dataspace, while `PropID` must be an immediate value.

Refer to the **Output Port Configuration Properties** section of this document for valid values of for `PropID`.

Instruction: `OP_GETTICK`
Opcode: `0x35`
Arguments: `Destination`
Description: Store the system tick, or milliseconds since the NXT started up, into `Destination`. `Destination` must be an integer in the dataspace.

Instruction Reference Appendix

This appendix includes supplementary material that is relevant to the instruction descriptions listed in the main **Instruction Reference** section of this document.

Input Port Configuration Properties

The instructions `OP_GETIN` and `OP_SETIN` use the following set of `PropID` values to specify particular configuration properties.

Value	PropID	Data Type
0x0	IO_IN_TYPE	TC_UBYTE
0x1	IO_IN_MODE	TC_UBYTE
0x2	IO_IN_ADRAW	TC_UWORD
0x3	IO_IN_NORMRAW	TC_UWORD
0x4	IO_IN_SCALED_VAL	TC_SWORD
0x5	IO_IN_INVALID_DATA	TC_UBYTE

The following sections describe each input port configuration property in alphabetical order.

ADRAW

Data type: UWORD
Access: Read-only
Legal value range: [0, 1023]

This property specifies the raw 10-bit value last read from the analog to digital converter on this port. Raw values produced by sensors typically cover some subset of the full 10-bit range.

INVALID_DATA

Data type: UBYTE
Access: Read-Write
Legal values: *TRUE* (1), *FALSE* (0)

This property signifies that the values of `ADRAW`, `NORMRAW`, and `SCALED_VAL` might be invalid due to sensor configuration changes that the NXT firmware has not processed yet. For example, the NXT firmware might not have processed the sensor type and/or mode changes immediately due to hardware limitations.

In all cases where you change `TYPE` or `MODE`, use the `INVALID_DATA` property to ensure that the next value you read back has been properly processed. To do so, set `INVALID_DATA` to *TRUE* immediately after setting `TYPE` and/or `MODE`, and then write a while loop that does not exit until `INVALID_DATA` becomes *FALSE*. At that point, the Normalized and Scaled properties will return valid values for the new configuration.

MODE

Data type: UBYTE
Access: Read-Write
Legal values:

0x00	RAWMODE	Report scaled value equal to raw value.
0x20	BOOLEANMODE	Report scaled value as 1 (*TRUE*) or 0 (*FALSE*). The firmware uses inverse Boolean logic to match the physical characteristics of NXT sensors. Readings are *FALSE* if raw value exceeds 55% of total range; readings are *TRUE* if raw value is less than 45% of total range.
0x40	TRANSITIONCNTMODE	Report scaled value as number of transitions between *TRUE* and *FALSE*.
0x60	PERIODCOUNTERMODE	Report scaled value as number of transitions from *FALSE* to *TRUE*, then back to *FALSE*.
0x80	PCTFULLSCALEMODE	Report scaled value as percentage of full scale reading for configured sensor type.
0xA0	CELSIUSMODE	Scale TEMPERATURE reading to degrees Celsius.
0xC0	FAHRENHEITMODE	Scale TEMPERATURE reading to degrees Fahrenheit.
0xE0	ANGLESTEPMODE	Report scaled value as count of ticks on RCX-style rotation sensor.

This property specifies the sensor mode for this port. The sensor mode affects the scaled value, which the NXT firmware calculates depending on the sensor type and sensor mode.

If you write to this property, also write a value of *TRUE* to the INVALID_DATA property.

NORMRAW

Data type: UWORD
Access: Read-only
Legal value range: [0, 1023]

This property specifies a 10-bit sensor reading, scaled according to the current value of the TYPE property on this port. The NXT firmware automatically applies internal scaling factors such that the physical range of raw values produced by the sensor is mapped, or normalized, to the full 10-bit range.

MODE is ignored when calculating the Normalized value.

The INVALID_DATA property should be read in conjunction with this property. The Normalized value is only valid if INVALID_DATA is *FALSE*.

SCALED_VAL

Data type: UWORD
Access: Read-Write

The legal value range depends on MODE, as listed below:

RAWMODE	[0, 1023]
BOOLEANMODE	[0, 1]
TRANSITIONCNTMODE	[0, 65535]
PERIODCOUNTERMODE	[0, 65535]
PCTFULLSCALEMODE	[0, 100]
CELSIUSMODE	[-200, 700] (readings in 10^{th} of a degree Celsius)
FAHRENHEITMODE	[-400, 1580] (readings in 10^{th} of a degree Fahrenheit)
ANGLESTEPMODE	[0, 65535]

This property specifies a sensor reading, scaled according to the current sensor type and mode on this port.

The INVALID_DATA property should be read in conjunction with this property. The SCALED_VAL value is only valid if INVALID_DATA is *FALSE*.

Because some combinations of sensor types and modes might lead to accumulation of count values in the SCALED_VAL property, you can reset this count by writing 0 to the SCALED_VAL property at any time. Note that you can write any value to this property at any time, but doing so is not generally very useful because outside of counter modes, the value is usually overwritten very quickly.

TYPE

Data type: UBYTE
Access: Read-Write
Legal Values:

0x00	NO_SENSOR	No sensor configured.
0x01	SWITCH	NXT or RCX touch sensor
0x02	TEMPERATURE	RCX temperature sensor
0x03	REFLECTION	RCX light sensor
0x04	ANGLE	RCX rotation sensor
0x05	LIGHT_ACTIVE	NXT light sensor with floodlight enabled
0x06	LIGHT_INACTIVE	NXT light sensor with floodlight disabled
0x07	SOUND_DB	NXT sound sensor; dB scaling
0x08	SOUND_DBA	NXT sound sensor; dBA scaling
0x09	CUSTOM	Unused in NXT programs
0x0A	LOWSPEED	I2C digital sensor
0x0B	LOWSPEED_9V	I2C digital sensor; 9V power
0x0C	HIGHSPEED	Unused in NXT programs

This property specifies the sensor type for this port. The sensor type primarily affects scaling factors used to calculate the normalized sensor value NORMRAW, but some values have other side effects.

If you write to this property, also write a value of *TRUE (1)* to the INVALID_DATA property.

Unlike the RCX firmware, there are no default sensor modes associated with each sensor type.

Output Port Configuration Properties

The OP_GETOUT and OP_SETOUT instructions use the following set of PropID values to specify particular configuration properties:

Value	PropID	Data Type
0x0	IO_OUT_FLAGS	TC_UBYTE
0x1	IO_OUT_MODE	TC_UBYTE
0x2	IO_OUT_SPEED	TC_SBYTE
0x3	IO_OUT_ACTUAL_SPEED	TC_SBYTE
0x4	IO_OUT_TACH_COUNT	TC_SLONG
0x5	IO_OUT_TACH_LIMIT	TC_ULONG
0x6	IO_OUT_RUN_STATE	TC_UBYTE
0x7	IO_OUT_TURN_RATIO	TC_SBYTE
0x8	IO_OUT_REG_MODE	TC_UBYTE
0x9	IO_OUT_OVERLOAD	TC_UBYTE
0xA	IO_OUT_REG_P_VAL	TC_UBYTE
0xB	IO_OUT_REG_I_VAL	TC_UBYTE
0xC	IO_OUT_REG_D_VAL	TC_UBYTE
0xD	IO_OUT_BLOCK_TACH_COUNT	TC_SLONG
0xE	IO_OUT_ROTATION_COUNT	TC_SLONG

The following sections provide information about each property in alphabetical order.

ACTUAL_SPEED (Interactive Motors Only)

Data type: SBYTE
Access: Read-only
Legal value range: [-100, 100]

This property returns the actual percentage of full power that the NXT firmware is applying to the output currently. This value can vary from the SPEED set-point when the internal auto-regulation code of the NXT firmware responds to drag on the output axle.

BLOCK_TACH_COUNT (Interactive Motors Only)

Data type: SLONG
Access: Read-only
Legal value range: [-2147483648, 2147483647]

This property reports the block-relative position counter value for the specified port.

Refer to the output **FLAGS** section for more information about using block-relative position counts.

Set the UPDATE_RESET_BLOCK_COUNT flag in FLAGS to request that the firmware resets BLOCK_TACH_COUNT.

The sign of BLOCK_TACH_COUNT specifies the rotation direction. Positive values correspond to forward rotation while negative values correspond to backward rotation. “Forward” and “backward” are relative to a standard orientation for a particular type of motor.

FLAGS

Data type: UBYTE
Access: Read-Write

This property is a bitfield that can include any combination of the following flag bits:

0x01	UPDATE_MODE	Commits changes to the MODE property.
0x02	UPDATE_SPEED	Commits changes to the SPEED property.
0x04	UPDATE_TACHO_LIMIT	Commits changes to the TACH_LIMIT property (interactive motors only).
0x08	UPDATE_RESET_COUNT	Resets internal movement counters, cancels current goal, and resets internal error-correction system (interactive motors only).
0x10	UPDATE_PID_VALUES	Commits changes to PID regulation parameters REG_P_VALUE, REG_I_VALUE, and/or REG_D_VALUE (interactive motors only).
0x20	UPDATE_RESET_BLOCK_COUNT	Resets block-relative position counter (interactive motors only).
0x40	UPDATE_RESET_ROTATION_COUNT	Resets program-relative position counter (interactive motors only).

This property is an unsigned byte bitfield with zero or more of the bit values above set.

You can use UPDATE_MODE, UPDATE_SPEED, UPDATE_TACHO_LIMIT, and UPDATE_PID_VALUES in conjunction with other properties to commit changes to the internal state of the NXT firmware. That is, you must set the appropriate flags after setting one or more of these properties before the changes actually take affect. For example, write a value of 0x03 (UPDATE_MODE | UPDATE_SPEED) to FLAGS immediately after you write new values to the MODE and SPEED properties.

The “reset” flags are independent of other properties and produce the side effects described above.

For UPDATE_RESET_BLOCK_COUNT, “block-relative” refers to the way this flag is used in the LEGO MINDSTORMS NXT Software 1.0. By convention, this flag is set every time an NXT-G motor control block starts execution. This convention means that the BLOCK_TACH_COUNT property always provides a position measurement relative to the last NXT-G motor control block to execute.

For UPDATE_RESET_ROTATION_COUNT, “program-relative” refers to the fact that this position counter is reset automatically at the beginning of every program. Set UPDATE_RESET_ROTATION_COUNT to reset this counter manually during program execution.

MODE

Data type: UBYTE
Access: Read-Write

This property is a bitfield that can include any combination of the following flag bits:

0x01	MOTORON	Enables pulse-width modulation (PWM) power to port(s) according to value of the SPEED property.
0x02	BRAKE	Applies electronic braking to port(s).
0x04	REGULATED	Enables active power regulation according to value of REG_MODE (interactive motors only).

This property is an unsigned byte bitfield with zero or more mode bits set. Clearing all bits (a MODE value of 0x00) is considered COAST mode; motors connected to the specified port(s) will rotate freely.

You must set the MOTORON bit for the NXT firmware to provide any power to the specified output port(s). Power is provided as a PWM waveform modulated by the SPEED property.

The BRAKE bit enables application of electronic braking to the circuit. "Braking" in this sense means that the output voltage is not allowed to float between active PWM pulses. Electronic braking improves the accuracy of motor output, but uses slightly more battery power.

You must use the REGULATED bit in conjunction with the REG_MODE property. Refer to the **REG_MODE** section of this document for more information about using the REGULATED bit with the REG_MODE property.

You must set the UPDATE_MODE bit in the FLAGS bitfield to commit changes to the MODE property.

OVERLOAD (Interactive Motors Only)

Data type: UBYTE
Access: Read-only
Legal values: *TRUE* (1), *FALSE* (0)

This property returns *TRUE* if the speed regulation functionality of the NXT firmware is unable to overcome physical load on the motor, e.g., the motor is turning more slowly than expected. Otherwise, this property returns *FALSE*.

You must set some other appropriately for the value of OVERLOAD to be meaningful. Use the following guidelines when setting this property:

- The MODE bitfield must include the MOTORON and REGULATED bits.
- REG_MODE must be set to REG_SPEED.
- RUN_STATE must be set to a non-IDLE value.

REG_D_VALUE (Interactive Motors Only)

Data type: UBYTE
Access: Read-Write
Legal value range: [0, 255]

This property specifies the derivative term used in the internal proportional-integral-derivative (PID) control algorithm.

Set UPDATE_PID_VALUES to commit changes to REG_P_VALUE, REG_I_VALUE, and REG_D_VALUE simultaneously.

REG_I_VALUE (Interactive Motors Only)

Data type: UBYTE
Access: Read-Write
Legal value range: [0, 255]

This property specifies the integral term used in the internal proportional-integral-derivative (PID) control algorithm.

Set UPDATE_PID_VALUES to commit changes to REG_P_VALUE, REG_I_VALUE, and REG_D_VALUE simultaneously.

REG_MODE (Interactive Motors Only)

Data type: UBYTE
Access: Read-Write
Legal Values:

0x00	REG_IDLE	Disables regulation.
0x01	REG_SPEED	Enables speed regulation.
0x02	REG_SYNC	Enables synchronization between any two motors.

This property specifies the regulation mode to use with the specified port(s).

This property is ignored if you do not set the REGULATED bit in the MODE property. Unlike the MODE property, REG_MODE is not a bitfield. You can set only one REG_MODE value at a time.

Speed regulation means that the NXT firmware attempts to maintain a certain speed according to the SPEED set-point. To accomplish this, the NXT firmware automatically adjusts the actual PWM duty cycle if the motor is affected by a physical load. This automatic adjustment is reflected by the value of the ACTUAL_SPEED property.

Synchronization means that the firmware attempts keep any two motors in synch regardless of varying physical loads. You typically use this mode is to maintain a straight path for a vehicle robot automatically. You also can use this mode with the TURN_RATIO property to provide proportional turning. You must set REG_SYNC on at least two motor ports to have the desired affect. If REG_SYNC is set on all three motor ports, only the first two (A & B) are synchronized.

REG_P_VALUE (Interactive Motors Only)

Data type: UBYTE
Access: Read-Write
Legal value range: [0, 255]

This property specifies the proportional term used in the internal proportional-integral-derivative (PID) control algorithm.

Set UPDATE_PID_VALUES to commit changes to REG_P_VALUE, REG_I_VALUE, and REG_D_VALUE simultaneously.

ROTATION_COUNT (Interactive Motors Only)

Data type: SLONG
Access: Read-only
Legal value range: [-2147483648, 2147483647]

This property returns the program-relative position counter value for the specified port.

Refer to the output **FLAGS** section for more information about using program-relative position counts.

Set the UPDATE_RESET_ROTATION_COUNT flag in FLAGS to request that the NXT firmware resets the ROTATION_COUNT property.

The sign of ROTATION_COUNT specifies rotation direction. Positive values correspond to forward rotation while negative values correspond to backward rotation. “Forward” and “backward” are relative to a standard orientation for a particular type of motor.

RUN_STATE

Data type: UBYTE
Access: Read-Write

Legal Values:

0x00	RUN_STATE_IDLE	Disables power to specified port(s).
0x10	RUN_STATE_RAMPUP	Enables automatic ramp-up to the SPEED set-point (interactive motors only).
0x20	RUN_STATE_RUNNING	Enables power to specified port(s) at the SPEED set-point.
0x40	RUN_STATE_RAMPDOWN	Enables automatic ramp-down to the SPEED set-point (interactive motors only).

This property specifies an auxiliary "state" to use with MODE, REG_MODE, and SPEED on the specified port(s). Set only one of the legal values at a given time.

RUN_STATE_RUNNING is the most common setting. Use RUN_STATE_RUNNING to enable power to any output device connected to the specified port(s).

RUN_STATE_RAMPUP enables automatic ramping to a new SPEED set-point that is greater than the current SPEED set-point. When you use RUN_STATE_RAMPUP in conjunction with appropriate TACH_LIMIT and SPEED values, the NXT firmware attempts to increase the actual power smoothly to the SPEED set-point over the number of degrees specified by TACH_LIMIT.

RUN_STATE_RAMPDOWN enables automatic ramping to a new SPEED set-point that is less than the current SPEED set-point. When you use RUN_STATE_RAMPDOWN in conjunction with appropriate TACH_LIMIT and SPEED values, the NXT firmware attempts to smoothly decrease the actual power to the SPEED set-point over the number of degrees specified by TACH_LIMIT.

SPEED

Data type: SBYTE
Access: Read-Write
Legal value range: [-100, 100]

This property specifies the power level set-point for the specified port(s).

The absolute value of SPEED is used as a percentage of the full power capabilities of the motor.

The sign of SPEED specifies rotation direction. Positive values for SPEED instruct the firmware to turn the motor forward, while negative values instruct the firmware to turn the motor backward. "Forward" and "backward" are relative to a standard orientation for a particular type of motor.

Note that direction is not a meaningful concept for outputs like lamps. Lamps are affected only by the absolute value of SPEED.

You must set some other properties appropriately for the SPEED setpoint to have the desired effect. Use the following guidelines when setting this property:

- The MODE bitfield must include MOTORON bit. The BRAKE bit is optional.
- RUN_STATE must be set to a non-IDLE value.

You must set the UPDATE_SPEED bit in the FLAGS bitfield to commit changes to the SPEED property.

TACH_COUNT (Interactive Motors Only)

Data type: SLONG
Access: Read-only
Legal value range: [-2147483648, 2147483647]

This property returns the internal position counter value for the specified port. This internal count is reset automatically when a new goal is set using the TACH_LIMIT and the UPDATE_TACHO_LIMIT flag.

Set the UPDATE_RESET_COUNT flag in FLAGS to specify that the NXT firmware resets TACH_COUNT and cancels any current programmed goals.

The sign of TACH_COUNT specifies rotation direction. Positive values correspond to forward rotation while negative values correspond to backward rotation. “Forward” and “backward” are relative to a standard orientation for a particular type of motor.

TACH_LIMIT (Interactive Motors Only)

Data type: ULONG
Access: Read-Write
Legal value range: [0, 4294967295]

This property specifies the rotational distance in degrees that you want to turn the motor.

Set the UPDATE_TACHO_LIMIT flag to commit changes to TACH_LIMIT. The NXT firmware treats the new TACH_LIMIT value as a relative distance from the motor position at the moment that the UPDATE_TACHO_LIMIT flag is processed. Remember that the sign of the SPEED property specifies the direction of rotation.

TURN_RATIO (Interactive Motors Only)

Data type: SBYTE
Access: Read-Write
Legal value range: [-100, 100]

This property specifies the proportional turning ratio for synchronized turning using two motors.

You must set some other properties appropriately on at least two motor ports for TURN_RATIO to have the desired effect. Use the following guidelines when setting this property.

- The MODE bitfield must include MOTORON and REGULATED bits. The BRAKE bit is optional.
- REG_MODE must be set to REG_SYNCH.
- RUN_STATE must be set to a non-IDLE value.
- SPEED must be set to a non-zero value.

After you set these property values, the NXT firmware uses the TURN_RATIO value to adjust relative power settings for the left and right motors automatically.

"Left" and "right" refer to the physical arrangement of the output plugs on an NXT brick (when facing the display screen). There are only three valid combinations of left and right motors for use with TURN_RATIO:

Left	***Right***
Output Port A	*Output Port B*
Output Port B	*Output Port C*
Output Port A	*Output Port C*

Note that this definition of "left" and "right" is independent of the LEGO model in use.

Negative TURN_RATIO values shift power towards the left motor, whereas positive TURN_RATIO values shift power towards the right motor. In both cases, the actual power applied is proportional to the SPEED set-point, such that an absolute value of 50% for TURN_RATIO normally results in one motor stopping, and an absolute value of 100% for TURN_RATIO normally results in the two motors turning in opposite directions at equal power.

System Call Methods

The NXT firmware 1.03 provides 33 special-purpose system call (syscall) methods. The firmware exposes these methods, or functions, with the `OP_SYSCALL` instruction. To call one of these functions, the compiler must construct an `OP_SYSCALL` instruction with a valid `SysCallID` argument and a cluster containing the appropriate parameters.

The following table lists valid `SysCallID` values:

Value	SysCallID
0x00	NXTFileOpenRead
0x01	NXTFileOpenWrite
0x02	NXTFileOpenAppend
0x03	NXTFileRead
0x04	NXTFileWrite
0x05	NXTFileClose
0x06	NXTFileResolveHandle
0x07	NXTFileRename
0x08	NXTFileDelete
0x09	NXTSoundPlayFile
0x0A	NXTSoundPlayTone
0x0B	NXTSoundGetState
0x0C	NXTSoundSetState
0x0D	NXTDrawText
0x0E	NXTDrawPoint
0x0F	NXTDrawLine
0x10	NXTDrawCircle
0x11	NXTDrawRect
0x12	NXTDrawPicture
0x13	NXTSetScreenMode
0x14	NXTReadButton
0x15	NXTCommLSWrite
0x16	NXTCommLSRead
0x17	NXTCommLSCheckStatus
0x18	NXTRandomNumber
0x19	NXTGetStartTick
0x1A	NXTMessageWrite
0x1B	NXTMessageRead
0x1C	NXTCommBTCheckStatus
0x1D	NXTCommBTWrite
0x1F	NXTKeepAlive
0x20	NXTIOMapRead
0x21	NXTIOMapWrite

Note: The NXT firmware 1.03 does not use `SyscallID` value `0x1E`.

These values are grouped into the following classes:

Class	Description
File Access Methods	Manipulate files in the NXT brick's flash memory file system.
NXT Display Methods	Draw to the display on the NXT brick.
NXT Button Method	Read the built-in buttons on the NXT brick.
Sound Playback Methods	Play back recorded or synthesized sound.
Digital I/O Communications Methods	Communicate with digital I/O port devices.
Bluetooth Communications Methods	Communicate with connected Bluetooth peers.
Low-Level System Methods	Access low-level system information.

All syscall methods require a cluster of parameters in the dataspace. These parameters can be inputs, outputs, or both. The format, or data type, of the parameter cluster is specific to a particular method.

All syscall method parameter clusters include at least one scalar, which is the *return value*. Return values are listed separately in this document, but must always be the first element of a syscall parameter cluster. Most syscall methods provide status codes as their return values. In general, status code values indicate one of the following three results:

- A status code of zero indicates a status of "OK"; no special action is required.
- A negative status code indicates an error.
- A positive status code indicates a warning.

The following sections provide information about each of these classes and the associated methods.

File Access Methods

Use the file access methods to create, modify, rename, or delete files stored on the NXT brick.

The NXT firmware organizes all files in a simple flat file system stored in the NXT bricks flash memory. Files are listed and referred to by name, and files can have any size up to the amount of available flash. The file system is flat because the file system does not support organizing files into hierarchical folders. NXT firmware 1.03 allows you to create up to 63 files (assuming flash space is available).

NXT filenames include up to 15 characters for the main name, a dot, and 3 characters for the extension (which specify the file type). This convention is referred to as "15.3 filenames". A file extension serves as a cue to the NXT firmware as to how and where files are listed and treated.

The NXT firmware 1.03 and the LEGO MINDSTORMS NXT Software 1.0 use the following main file extensions:

- `.RXE`: Executable files compiled from NXT-G, LabVIEW, or another compatible programming environment.
- `.RPG`: 5-step programs created using the "NXT Program" UI on the NXT brick.
- `.RTM`: Built-in "Try Me" programs.
- `.RIC`: Image files for use with the `NXTDrawPicture` syscall method in `.RXE` programs.
- `.RSO`: Sound files.
- `.SYS`: Internal NXT firmware files.
- `.CAL`: Sensor calibration file.
- `.TXT`: ASCII text file using carriage return/line feed (CR/LF, Windows) end-of-line convention.

Programs can create, modify, rename, or delete any file in the system. Typical NXT programs use simple text files with the `.TXT` extension. You then can upload these files easily to a PC, which then can read the files. You can use any extension or file encoding you choose, but make sure not to interfere with the various files that the NXT firmware and/or other programs use.

The default NXT firmware configuration includes several built-in files. You can delete any of these files to free up space, but you lose any associated functionality until you restore these files.

The following sections list these individual methods along with some concepts with which you need to be familiar.

File Handles

You must open a handle to a file before you can use read or write methods on the file. The `NXTFileOpenRead`, `NXTFileOpenWrite`, and `NXTFileOpenAppend` methods return a unique handle value. The NXT firmware also automatically registers open handles in an internal table such that the `NXTResolveHandle` method can look up any open handle by filename.

You must close each file handle with the `NXTFileClose` method before using the file to which it refers for any other purpose. When a program ends, the NXT firmware automatically closes any handles left open by the program.

The NXT firmware restricts the maximum number of concurrently open file handles to 16. Note that running programs uses up one file handle, and any current sound playback or other background process can take up additional file handles. For these reasons, a program can generally open 10–12 handles.

File Access Status Codes

File access methods return a different set of status codes from most other methods. File access status codes are unsigned 16-bit integers rather than signed 8-bit integers. In most cases, you only need to check if file access methods return status codes other than 0, which indicates a successful operation. However, the following table describes the valid file access status codes.

0x0000	SUCCESS	Operation succeeded.
0x8100	NO_HANDLES	The firmware is not able to allocate any more file handles for this operation.
0x8300	NO_FILES	The firmware is not able to create any more files.
0x8400	PARTIAL_WRITE	Write request exceeded space available in file; partial write performed.
0x8500	EOF	File operation reached end of file.
0x8700	FILE_NOT_FOUND	Specified file not found.
0x8800	FILE_CLOSED	Specified file or handle is already closed.
0x8900	NO_LINEAR_SPACE	The firmware is not able to allocate requested linear file system space for specified file.
0x8A00	GENERIC_ERROR	An unspecified error condition occurred.
0x8B00	FILE_BUSY	Cannot acquire file handle for specified file because some other operation has opened the file.
0x8C00	NO_WRITE_BUFFERS	Cannot open file for write operation because all write buffers are already in use. Only four files can be open for write simultaneously.
0x8D00	ILLEGAL_APPEND	Cannot open file for append.
0x8E00	FILE_FULL	Allocated space for specified file is full (no more write operations allowed).
0x8F00	FILE_EXISTS	Failure to create or rename file due to name collision.
0x9200	ILLEGAL_FILE_NAME	Illegal file name provided. Ensure file name consists of 15.3 (or fewer) printable characters.
0x9300	ILLEGAL_HANDLE	Allocated space for specified file is full.

File Access Performance Issues

Writing to flash memory is very slow compared to RAM access on the NXT brick. Writing to flash memory also monopolizes the CPU such that no other firmware operations can happen while flash writing is in progress. The firmware buffers file write operations in RAM whenever possible, but the firmware is subject to pauses of several milliseconds while these buffers are committed to flash memory. If you experience significant delays during program operations, consider minimizing the amount of flash writing you perform.

Most file access method calls are subject to delays averaging 6 ms or less, with `NXTFileOpenWrite` subject to the most delay per call. The following sections describe the methods and include specific notes about potential flash writing performance issues where applicable.

NXTFileClose

Return Value:

`Status Code`	UWORD

Parameters:

`File Handle`	UBYTE	input	Unique handle to identify open file.

This syscall method closes the handle specified by `File Handle`.

If the return value is 0, the method successfully closed the file handle and removed the file from the list of open files that `NXTResolveHandle` uses.

If the return value is non-zero, the close operation failed. Either the specified file handle was invalid or no file was currently open using that file handle.

All files opened by an NXT program are automatically closed when the program finishes or is aborted.

Note that a successful `NXTFileClose` operation commits any pending file write buffers to flash memory, so this operation is subject to the performance issues described in the **File Access Performance Issues** section of this document.

NXTFileDelete

Return Value:

`Status Code`	UWORD

Parameters:

`Filename`	string	input	The name of the file, with a maximum of 19 characters (15.3 filename).

This syscall method deletes the file specified by `Filename`.

If the return value is 0, the delete operation succeeded. If the return value is non-zero, the delete operation failed. Either the specified file does not exist or an open handle is associated with the file.

Be sure to close any open handles associated with a file before deleting it.

Note that file deletion involves writing internal file system data to flash memory, so this syscall method is subject to the performance issues described in the **File Access Performance Issues** section of this document.

NXTFileOpenAppend

Return Value:

Status Code	UWORD

Parameters:

File Handle	UBYTE	output	Unique handle to identify open file.
Filename	string	input	The name of the file, with a maximum of 19 characters (15.3 filename).
Length	ULONG	output	Number of bytes remaining in file.

This syscall method attempts to open the file specified by Filename for append (write) operations. This operation is useful only if a file already exists and has been closed before all of its allocated space has been filled.

If the return value is 0, the file open operation succeeded. The File Handle output contains a unique handle for use with NXTFileWrite and NXTFileClose and the Length output contains the number of unused bytes remaining in the file. The internal file cursor is set automatically to the end of existing data such that future calls to NXTFileWrite do not overwrite any data. Furthermore, this file is registered for use with NXTFileResolveHandle.

If the return value is non-zero, an error occurred attempting to open the file. In this situation, you can ignore File Handle and Length because this file is not registered for use with NXTFileResolveHandle.

Only four files may be concurrently opened for write operations. This limit includes files opened with NXTFileOpenAppend.

NXTFileOpenRead

Return Value:

Status Code	UWORD

Parameters:

File Handle	UBYTE	output	Unique handle to identify open file.
Filename	string	input	The name of the file, with a maximum of 19 characters (15.3 filename).
Length	ULONG	output	The length of file, in bytes.

This syscall method attempts to open the file specified by Filename for read operations.

If the return value is 0, the file open operation succeeded. File Handle is assigned a unique handle for use with NXTFileRead and NXTFileClose, and Length is assigned the current length of the file. Furthermore, this file is registered for use with NXTFileResolveHandle.

If the return value is non-zero, an error occurred attempting to open the file. In this situation, you can ignore File Handle and Length because this file is not registered for use with NXTFileResolveHandle.

NXTFileOpenWrite

Return Value:

Status Code	UWORD

Parameters:

File Handle	UBYTE	output	Unique handle to identify open file.
Filename	string	input	The name of the file, with a maximum of 19 characters (15.3 filename).
Length	ULONG	input-output	Length of file, in bytes.

This syscall method attempts to create a file with the name specified by Filename and size in bytes specified by Length, then keep the file open for write operations. You must specify the total file size using the Length parameter when you create the file. If you do not use all of the memory allocated for a particular file before closing the file handle, you can open the file for further write operations using the NXTFileOpenAppend syscall method.

If the return value is 0, the file creation operation was successful. The File Handle output is assigned a unique handle for use with NXTFileWrite and NXTFileClose and the Length output is assigned the current length of the file. Furthermore, this file is registered for use with NXTFileResolveHandle.

If the return value is non-zero, an error occurred attempting to open the file. In this situation, you can ignore File Handle and Length because this file is not registered for use with NXTFileResolveHandle.

Only four files can be concurrently opened for write operations. This limit includes files opened with NXTFileOpenAppend.

Note that file creation involves writing data to flash, so this syscall method is subject to performance issues described. NXTFileOpenWrite is potentially subject to the most flash writing delay of any method. Creating a very large file can result in a delay of up to 30ms.

NXTFileRead

Return Value:

Status Code	UWORD

Parameters:

File Handle	UBYTE	input-output	Unique handle to identify open file.
Buffer	string	output	File data in string format.
Length	ULONG	input-output	Length of file in bytes.

This syscall method attempts to read Length bytes of data from the file opened with the handle specified by File Handle.

If the return value is 0, the file read operation succeeded. Buffer contains Length bytes of file data in string format. If the file contains text data, use the string like a normal text string, e.g., to display to the screen. If the file contains flattened binary data, use the OP_UNFLATTEN instruction to unflatten it.

If the return value is non-zero, an error occurred attempting to read bytes from the file. Note that Buffer might still contain valid data if the method encountered the end of the file before reading Length bytes. In this case, read the Length output to find out how many bytes were actually read.

Successive calls to NXTFileRead with the same file handle will read new data each time, that is, each read operation advances the internal file read cursor.

NXTFileRename

Return Value:

`Status Code`	UWORD

Parameters:

`Old Filename`	string	input	The current name of the file, with a maximum of 19 characters (15.3 filename).
`New Filename`	string	input	The new name of the file, with a maximum of 19 characters (15.3 filename).

This syscall method renames file specified by `Old Filename` to `New Filename`.

If the return value is 0, the rename operation succeeded. If the return value is non-zero, the rename operation failed. Either the specified file does not exist or an open handle is associated with the file.

Be careful to close any open handles associated with a file before renaming it.

Note that file renaming involves writing internal file system data to flash memory, so this syscall method is subject to the performance issues described in the **File Access Performance Issues** section of this document.

NXTFileResolveHandle

Return Value:

`Status Code`	UWORD

Parameters:

`File Handle`	UBYTE	output	Unique handle to identify open file
`Write Handle?`	UBYTE	output	Returns *TRUE (1)* if the handle is open for write operations. Otherwise, returns *FALSE (0)*.
`Filename`	string	input	The name of the file, with a maximum of 19 characters (15.3 filename).

The NXT firmware maintains a list of open file handles. This syscall method searches the list of open file handles by `Filename`. To succeed, `Filename` must contain the exact filename of an already opened file.

If the return value is 0, the file is already open. The `File Handle` output is assigned a unique handle for use with `NXTFileRead`, `NXTFileWrite`, and `NXTFileClose`. If the Boolean output `Write Handle?` is set to *TRUE*, the file is open for write operations. Otherwise the file is open for read operations.

If the return value is non-zero, the file is not yet open. You can ignore `File Handle` and `Write Handle?`. If you intend to use the file, call the appropriate open method for the file.

NXTFileWrite

Return Value:

`Status Code`	UWORD

Parameters:

`File Handle`	UBYTE	input-output	Unique handle to identify open file.
`Buffer`	string	input-output	File data in string format.
`Length`	ULONG	input-output	Number of bytes to write (in); number of bytes written (out).

This syscall method attempts to write `Length` bytes of data to the file opened with the handle specified by `File Handle`. If you use all of the memory allocated for the specified file, the `NXTFileWrite` method writes partial contents of the `Buffer` data to the file. The method sets `Length` to the number of bytes actually written, but does not modify the `Buffer`. If you intend to write more data, you need to close this file and use `NXTOpenWrite` to create a new file.

If the return value is 0, the file write operation succeeded. If the return value is non-zero, an error occurred attempting to write bytes from the file.

If you use all of the memory allocated for the specified file, the `NXTFileWrite` method writes partial contents of the `Buffer` data to the file. The method sets `Length` to the number of bytes actually written, but does not modify the `Buffer`. If you intend to write more data, you need to close this file and use `NXTOpenWrite` to create a new file.

Successive calls to `NXTFileWrite` with the same file handle write new data each time. Each write operation advances the internal file write cursor.

Note that `NXTFileWrite` involves writing to flash memory, so this syscall method is subject to the performance issues described above. Because `NXTFileWrite` often is called many times in quick succession (to stream data to a file), the NXT firmware provides some buffering to minimize the performance cost. This buffering means that a 4–6 ms delay might occur for every 256 bytes written.

NXT Display Methods

Use the NXT display methods to draw text, points, shapes, or graphic files to the built-in display on the NXT brick.

NXTDrawCircle

Return Value:

`Status Code`	SBYTE	Unused

Parameters:

`Center`	cluster SWORD (X) SWORD (Y)	input-output	XY coordinates, relative to the lower-left corner of the screen, that specify the center of the circle.
`Radius`	UBYTE	input	Radius, in pixels.
`Options`	ULONG	input	Bitfield of draw options.

This syscall method draws a circle outline specified by `Center` coordinates and `Radius` (in pixels).

All draw coordinates are relative to the lower left corner of the screen on the NXT brick.

Set the least significant bit of `Options` to `1` to clear the entire screen before drawing. If you do not set this bit, the method overlays the circle on top of any pixels already drawn by the program. The circle outline is transparent; the method never modifies pixels inside the circle.

The first drawing syscall method to execute in a program automatically clears the screen regardless of the value of `Options`.

The return value is always 0.

NXTDrawLine

Return Value:

`Status Code`	SBYTE	Unused

Parameters:

`StartLocation`	cluster SWORD (X) SWORD (Y)	input-output	XY coordinates, relative to the lower-left corner of the screen.
`EndLocation`	cluster SWORD (X) SWORD (Y)	input-output	XY coordinates, relative to the lower-left corner of the screen.
`Options`	ULONG	Input	Bitfield of draw options.

This syscall method draws a black line one pixel wide from `StartLocation` to `EndLocation`.

All draw coordinates are relative to the lower-left corner of the screen on the NXT brick.

Set the least significant bit of `Options` to `1` to clear the entire screen before drawing. If you do not set this bit, the method overlays the line on top of any pixels already drawn by the program.

The first drawing syscall method to execute in a program automatically clears the screen regardless of the value of `Options`.

The return value is always 0.

NXTDrawPicture

Return Value:

Status Code	UBYTE	

Parameters:

Location	cluster SWORD (X) SWORD (Y)	input-output	XY coordinates, relative to the lower-left corner of the screen.
Filename	string	input	Maximum of 19 characters (15.3 filename)
Variables	SLONG array	input	Optional parameters as defined by the .RIC file.
Options	ULONG	input	Bitfield of draw options.

This syscall method renders the .RIC-format graphic file specified by Filename. Location specifies coordinates of the lower left corner of the rendered image.

All draw coordinates are relative to the lower-left corner of the screen on the NXT brick.

The Variables argument specifies an array of arbitrary numeric parameters that certain .RIC files might use. Most files ignore these variables.

Set the least significant bit of Options to 1 to clear the entire screen before drawing. If you do not set this bit, the method overlays the graphic on top of any pixels already drawn by the program.

The first drawing syscall method to execute in a program automatically clears the screen regardless of the value of Options.

If return value is non-zero, an error occurred while attempting to draw the file. Either the specified file does not exist, or is not a valid .RIC format file.

NXTDrawPoint

Return Value:

Status Code	SBYTE	Unused

Parameters:

Location	cluster SWORD (X) SWORD (Y)	input-output	XY coordinates, relative to the lower-left corner of the screen.
Options	ULONG	input	Bitfield of draw options

This syscall method draws a single black pixel at the coordinates that Location specifies.

All draw coordinates are relative to the lower left corner of the screen on the NXT brick.

Set the least significant bit of Options to 1 to clear the entire screen before drawing. If you do not set this bit, the method overlays the point on top of any pixels already drawn by the program.

The first drawing syscall method to execute in a program automatically clears the screen regardless of the value of Options.

The return value is always 0.

NXTDrawRect

Return Value:

`Status Code`	SBYTE	Unused

Parameters:

`Location`	cluster SWORD (X) SWORD (Y)	input-output	XY coordinates, relative to the lower-left corner of the screen.
`Size`	cluster SWORD (Width) SWORD (Height)	input-output	Dimensions of rectangle, in pixels.
`Options`	ULONG	input	Bitfield of draw options.

This syscall method draws a rectangle outline with one corner at the coordinates specified by `Location`. The `Size` cluster specifies relative coordinates of the corner opposite from `Location`.

Set the least significant bit of `Options` to `1` to clear the entire screen before drawing. If you do not set this bit, the method overlays the rectangle on top of any pixels already drawn by the program. The rectangle outline is transparent; the method never modifies pixels inside the rectangle.

The first drawing syscall method to execute in a program automatically clears the screen regardless of the value of `Options`.

The return value is always 0.

NXTDrawText

Return Value:

`Status Code`	SBYTE	Unused

Parameters:

`Location`	cluster SWORD (X) SWORD (Y)	input-output	XY coordinates, relative to the lower-left corner of the screen.
`Text`	string	input	Text string to draw.
`Options`	ULONG	input	Bitfield of draw options.

This syscall method renders the string `Text` at the coordinates specified by `Location`. This method only renders printable characters; this method ignores non-ASCII or non-printable characters. This method does not wrap text at the screen edges.

All draw coordinates are relative to the lower left corner of the screen on the NXT brick. The `NXTDrawText` method coerces y-coordinates to multiples of 8 (rounding down) such that text always appears on one of eight distinct lines on the display.

Set the least significant bit of `Options` to `1` to clear the entire screen before drawing. If you do not set this bit, the method overlays the text on top of any pixels already drawn by the program.

The first drawing syscall method to execute in a program automatically clears the screen regardless of the value of `Options`.

The return value is always 0.

NXTSetScreenMode

Return Value:

Status Code	SBYTE	

Parameters:

ScreenMode	ULONG	input	New screen mode.

This syscall method sets a new mode for the NXT display screen. The only valid mode for NXT firmware 1.03 is `RESTORE_NXT_SCREEN`, or value `0`. Use this screen mode to restore the default status screen after using any of the `NXTDraw` syscall methods.

The return value is always 0

NXT Button Method

Use the NXT button method to read the status of the built-in buttons on the NXT brick. Note that you can only read the top three buttons, as the bottom button always aborts the program.

NXTReadButton

Return Value:

Status Code	SBYTE

Parameters:

Index	UBYTE	input	Button index: RIGHT, LEFT, or ENTER. Refer to the table below this one for the valid `Index` parameter values.
Pressed	UBYTE	output	*TRUE (1)* if specified button is currently depressed
Count	UBYTE	output	Number of times specified button has been pressed and released since last reset.
Reset?	UBYTE	input	Set to *TRUE (1)* to reset `Count` after reading state.

This syscall method reads the state of the built-in NXT button specified by `Index`. The following table displays the legal values of `Index`.

0x01	RIGHT	Right arrow button
0x02	LEFT	Left arrow button
0x03	ENTER	Center square button

Set `Reset?` to *TRUE* to reset `Count` after reading the state.

If the status code is 0, `Pressed` returns *TRUE* when the button is depressed, and `Count` returns the number of times button has been depressed and released since last reset. If the status code is non-zero, you specified an illegal value for `Index`.

Sound Playback Methods

Use the sound playback methods to control the sound module on the NXT brick.

NXTSoundGetState

Return Value:

State	UBYTE	

Parameters:

Flags	UBYTE	output	Bitfield of sound module flags.

This syscall method reads the internal state and flags of the sound module of the NXT brick. If Flags is non-zero, the sound module has playback operations pending or in progress. The following table describes the values that Flags can return.

0x00	SOUND_FLAGS_IDLE	No flags set. The sound module is idle.
0x01	SOUND_FLAGS_UPDATE	A request for playback is pending.
0x02	SOUND_FLAGS_RUNNING	Playback in progress.

The return value can be any of the following values:

0x00	SOUND_IDLE	The sound module is idle, but there might be a pending request to playback sound.
0x02	SOUND_FILE	The Sound module is playing a .RSO file.
0x03	SOUND_TONE	The sound module is playing a tone.
0x04	SOUND_STOP	A request to stop playback is in progress.

NXTSoundPlayFile

Return Value:

Status Code	SBYTE	Unused

Parameters:

Filename	string	input	The name of the file, with a maximum of 19 characters (15.3 filename).
Loop?	UBYTE	input	Set to *TRUE (1)* to enable automatic looping of sound file.
Volume	UBYTE	input	Volume of playback, between 0 and 4.

This syscall method starts playback of the sound file specified by Filename. The file must be a valid .RSO-format sound file.

Set Loop? to *TRUE* to loop playback automatically and indefinitely without further syscall methods.

The following table displays the legal values for Volume and the associated behavior.

0	Sound playback disabled.
1	25% of full volume.
2	50% of full volume.
3	75% of full volume.
4	100% of full volume.

NXTSoundPlayTone

Return Value:

Status Code	SBYTE	Unused

Parameters:

Frequency	UWORD	input	Frequency of tone, in Hertz.
Duration	UWORD	input	Duration of tone, in milliseconds.
Loop?	UBYTE	input	Set to *TRUE (1)* to enable automatic looping of tone.
Volume	UBYTE	input	Volume of playback, between 0 and 4.

This syscall method starts playback of a tone specified by Frequency in Hz and Duration in ms.

Set Loop? to *TRUE* to loop playback automatically and indefinitely without further syscall methods.

The following table displays the legal values for Volume and the associated behavior.

0	Sound playback disabled.
1	25% of full volume.
2	50% of full volume.
3	75% of full volume.
4	100% of full volume.

NXTSoundSetState

Return Value:

State	UBYTE	

Parameters:

State	UBYTE	input	New state for the sound module.
Flags	UBYTE	input	Bitfield of sound module flags.

This syscall method writes new State and Flags values to the sound module of the NXT brick. Use this syscall method with caution because it directly influences the internals of the sound module on the NXT brick. Use this method only for stopping current playback by writing a new State value of SOUND_STOP.

The following value is the only legal value you can write to State:

0x04	SOUND_STOP	Request to stop playback.

The following value is the only legal value you can write to Flags:

0x00	SOUND_FLAGS_IDLE	No flags set.

Digital I/O Communication Methods

Use the digital I/O communication methods to access devices that use the I2C protocol on the NXT brick's four input ports.

You must set the `TYPE` property to `LOWSPEED` or `LOWSPEED_9V` on a given port before using an I2C device on that port. Use `LOWSPEED_9V` if your device requires 9V power from the NXT brick. Remember that you also need to set the `INVALID_DATA` property to *TRUE* after setting a new `TYPE`, then wait (e.g. with a while loop) for the NXT firmware to set `INVALID_DATA` 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.

When communicating with I2C devices, the NXT firmware uses a *master/slave* setup in which the NXT brick is always the master device. This setup means that the NXT firmware is responsible for controlling the write and read operations. Furthermore, the NXT firmware maintains write and read buffers for each port, and the three syscall methods provided enable you to access these buffers.

A call to `NXTCommLSWrite` constitutes the start of an asynchronous *transaction* between the NXT brick and a digital device, such that 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 `NXTCommLSWrite`, use `NXTCommLSCheckStatus` in a while loop to check the status of the port. If `NXTCommLSCheckStatus` returns a status code of 0 and a count of bytes available in the read buffer, the system is ready for you to use `NXTCommLSRead` to copy the data from the read buffer into another buffer.

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.

The following sections provide more information about each low speed communication method.

NXTCommLSCheckStatus

Return Value:

`Status Code`	SBYTE	

Parameters:

`Port`	UBYTE	input	Port. The four valid ports are `0`, `1`, `2`, and `3`.
`BytesReady`	UBYTE	input	Number of bytes ready for reading, if any.

This syscall method checks the status of the I2C communication on the specified port. If the last operation on this port was a successful `NXTCommLSWrite` operation that requested response data from a device, `BytesReady` indicates the number of bytes in the internal read buffer. You can access this information using `NXTCommLSRead`.

If the return value is 0, the port is idle and the last operation (if any) did not cause any errors.

The following table describes the status codes that indicate an error.

32	0x20	STAT_COMM_PENDING	The specified Port is busy performing a transaction.
-35	0xDD	ERR_COMM_BUS_ERR	The last transaction failed, possibly due to a device failure.
-33	0xDF	ERR_COMM_CHAN_INVALID	The specified Port is invalid. Port must be between 0 and 3.
-32	0xE0	ERR_COMM_CHAN_NOT_READY	The specified Port is not properly configured.

ERR_COMM_BUS_ERR typically means that either no digital device is connected to the specified port or the connected device is configured incorrectly. To clear the error condition, you can attempt to write new data to the device.

If this method returns ERR_COMM_CHAN_NOT_READY, ensure TYPE is set properly and that INVALID_DATA is *FALSE* for this port before attempting further transactions. Refer to the **Input Port Configuration Properties** section of this document for more information about TYPE and INVALID_DATA.

If this method returns STAT_COMM_PENDING, an operation is in progress. Do not attempt to interrupt operations in progress. Avoid calls to NXTCommLSRead or NXTCommLSWrite until NXTCommLSCheckStatus returns 0 or a negative error code.

NXTCommLSRead

Return Value:

Status Code	SBYTE	

Parameters:

Port	UBYTE	Input	Port. The four valid ports are 0, 1, 2, and 3.
Buffer	UBYTE array	Out	Bytes read from device, if any.
BufferLength	UBYTE	Input	Upper bound on number of bytes to read into Buffer. The actual number returned is limited to bytes available in the internal read buffer.

This syscall method attempts to copy BufferLength bytes from the internal read buffer to another buffer.

If the return value is 0, the read operation succeeded and Buffer contains all bytes available in the internal buffer. Successive NXTCommLSRead method calls read new data each time.

The following table describes the status codes that indicate an error.

32	0x20	STAT_COMM_PENDING	The specified Port is busy performing a transaction.
-35	0xDD	ERR_COMM_BUS_ERR	The last transaction failed, possibly due to a device failure.
-33	0xDF	ERR_COMM_CHAN_INVALID	The specified Port is invalid. Port must be between 0 and 3.
-32	0xE0	ERR_COMM_CHAN_NOT_READY	The specified Port is not properly configured.

If this method returns any negative status code, Buffer is an empty array.

NXTCommLSWrite

Return Value:

Status Code	SBYTE	

Parameters:

Port	UBYTE	input	Port. The four valid ports are 0, 1, 2, and 3.
Buffer	UBYTE array	input	Up to 16 bytes for writing to device.
ReturnLength	UBYTE	input	Number of bytes expected from device in response to writing data in Buffer; maximum 16.

This syscall method copies data from the buffer input to an internal write buffer and instructs the NXT firmware to perform a transaction by sending the write buffer to the device and reading ReturnLength bytes back into the internal read buffer.

If the return value is 0, the method successfully started a communication transaction. Use NXTCommLSCheckStatus to monitor the status of the transaction.

The following table describes the status codes that indicate an error.

-33	0xDF	ERR_COMM_CHAN_INVALID	The specified Port is invalid. Port must be between 0 and 3.
-32	0xE0	ERR_COMM_CHAN_NOT_READY	The specified Port is busy or not properly configured.
-19	0xED	ERR_INVALID_SIZE	Either Buffer or ReturnLength exceeded the 16-byte limit.

Bluetooth Communication Methods

Use the Bluetooth communication methods to send packets of information to other devices connected to the NXT brick via Bluetooth. You also use these methods to access the messaging queue system of the NXT firmware.

The NXT firmware uses a master/slave serial port system for Bluetooth communication. You must designate one Bluetooth device as the master device before you run a program using Bluetooth. If the master device is the NXT brick, you can configure up to three slave devices using serial ports 1, 2, and 3 on this brick. If the slave device is an NXT brick, port 0 on this brick is reserved for the master device.

Programs running on the master NXT brick can send packets of data to any connected slave devices using the NXTCommBTWrite method. However, slave devices cannot send packets to master devices. The firmware of slave NXT devices automatically handles responses sent by programs on master devices.

Refer to the protocol documentation for more information regarding Bluetooth packet structure.

This section also includes descriptions of the system call methods for accessing the NXT brick's *mailbox*, or message queues. By 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 NXTCommBTWrite on the master brick to send a MessageWrite protocol packet to the slave. Then, use NXTMessageRead on the slave brick to read the message. The slave NXT brick must be running a program when an incoming message packet is received. If no program is running, the slave NXT brick ignores the message, and the message is lost.

To exchange numeric data using the message system, use the OP_FLATTEN and OP_UNFLATTEN instructions to convert data to and from text strings.

NXTCommBTCheckStatus

Return Value:

Status Code	SBYTE	

Parameters:

Connection	UBYTE	input	Port. The four valid ports are 0, 1, 2, and 3.

This syscall method checks the status of the Bluetooth communication on the specified Port.

If the return value is 0, the port is idle and the last operation, if any did not cause any errors.

The following table describes the status codes that indicate an error.

32	0x20	STAT_COMM_PENDING	The specified Port is busy performing a transaction.
-35	0xDD	ERR_COMM_BUS_ERR	The last transaction failed, possibly due to a device failure.
-33	0xDF	ERR_COMM_CHAN_INVALID	The specified Port is invalid. Port must be between 0 and 3.
-32	0xE0	ERR_COMM_CHAN_NOT_READY	The specified Port is not properly configured.

If this method returns ERR_COMM_CHAN_NOT_READY, ensure a Bluetooth connection is configured on the specified port.

If this method returns STAT_COMM_PENDING, an operation is in progress. Do not attempt to interrupt operations in progress. Avoid calls to NXTCommBTRead or NXTCommBTWrite until NXTCommBTCheckStatus returns 0 or a negative error code.

NXTCommBTWrite

Return Value:

Status Code	SBYTE	

Parameters:

Connection	UBYTE	input	Port. The four valid ports are 0, 1, 2, and 3.
Buffer	UBYTE array	input	Up to 256 bytes for writing to specified port.

This syscall method copies data from the buffer input to an internal Bluetooth buffer and instructs the NXT firmware to send the data to the device configured on the specified port.

If the return value is 0, the method successfully started a communication transaction. Use NXTCommBTCheckStatus to monitor the status of the transaction.

The following table describes the status codes that indicate an error.

-33	0xDF	ERR_COMM_CHAN_INVALID	The specified Connection is invalid. Connection must be between 0 and 3.
-32	0xE0	ERR_COMM_CHAN_NOT_READY	The specified port is busy or not properly configured.
-19	0xED	ERR_INVALID_SIZE	The Buffer size exceeded the 256-byte limit.

If you are sending data to another NXT brick, the buffer should contain a complete packet conforming to the NXT communication protocol. Refer to the protocol documentation for LEGO MINDSTORMS NXT for more information regarding Bluetooth packet structure.

NXTMessageRead

Return Value:

Status Code	SBYTE	

Parameters:

QueueID	UBYTE	input	Mailbox queue. The valid queues are between 0 and 9.
Remove (T)	UBYTE	input	If *TRUE (1)*, remove message from specified queue after reading data.
Message	string	output	Message data.

This syscall method reads the oldest message available in the specified mailbox queue. You also can specify that this method removes that message from the queue.

If the return value is 0, the specified message queue was not empty and the Message output contains the oldest message from the queue.

The following table describes the status codes that indicate an error.

64	0x40	STAT_MSG_EMPTY_MAILBOX	The specified QueueID is empty
-18	0xEE	ERR_INVALID_QUEUE	The specified QueueID is invalid. The valid queues are between 0 and 9.

If you are calling NXTMessageRead on a master NXT brick with slave devices connected, this method will also periodically check for outgoing messages on the slave devices by automatically exchanging Bluetooth protocol packets with slaves.

NXTMessageWrite

Return Value:

Status Code	SWORD	

Parameters:

QueueID	UBYTE	input	Mailbox queue. The valid queues are between 0 and 9.
Message	string	input	Message data.

This syscall method writes a new message to the specified mailbox queue. If there are already five messages in the specified queue, this method deletes the oldest message.

If the return value is 0, the method succeeded in writing the message to the specified mailbox queue.

The following table describes the status codes that indicate an error.

-18	0xEE	ERR_INVALID_QUEUE	The specified QueueID is invalid. The valid queues are between 0 and 9.
-19	0xED	ERR_INVALID_SIZE	The Message is too large.

If you are calling NXTMessageWrite on a slave NXT brick, use mailbox queues 10 through 19 as outboxes. When the master NXT brick reads messages, it checks these upper 10 mailboxes for outgoing message on the slave.

Low-Level System Methods

Use the low-level system methods to access miscellaneous low-level features of the NXT firmware.

NXTGetStartTick

Return Value:

Program Start Tick	ULONG	

This syscall method returns the value of the system millisecond timer corresponding to the start of execution of the current program. This method is useful for measuring time difference relative to the start of program execution.

NXTIOMapRead

Return Value:

Status Code	SBYTE

Parameters:

ModuleName	string	input	The name of firmware module.
Offset	UWORD	input	The offset from beginning of the module's I/O map.
Count	UWORD	input	The count of bytes to read.
Buffer	UBYTE array	output	I/O map data.

This syscall method reads internal firmware module state information. This method is reserved for internal use only.

NXTIOMapWrite

Return Value:

Status Code	SBYTE

Parameters:

ModuleName	string	input	The name of the firmware module.
Offset	UWORD	input	The offset from beginning of the module's I/O map.
Buffer	UBYTE array	input	I/O map data.

This syscall method writes internal firmware module state information. This method is reserved for internal use only.

NXTKeepAlive

Return Value:

Sleep Time Limit	ULONG	

This syscall method resets the NXT brick's internal sleep timer and returns the current time limit, in milliseconds, until the next automatic sleep. Use this method to keep the NXT brick from automatically turning off. Use the NXT brick's UI menu to configure the sleep time limit.

NXTRandomNumber

Return Value:

Random Number	SWORD	

This syscall method returns a signed 16-bit random number. The firmware chooses new random seeds after every 20 calls to this method.

Reserved Opcodes

Opcode values not listed in the **Instruction Reference** section of this document are not supported in the NXT firmware version 1.03 and will result in fatal run-time errors if used. Also, the following opcode values are reserved for internal use.

- 0x0A
- 0x0B
- 0x0C
- 0x0D
- 0x0E
- 0x0F
- 0x10
- 0x13
- 0x14
- 0x34

GLOSSARY

Block Diagram

NXT-G is a graphical programming language. Source code in NXT-G takes the form of block diagrams consisting of code blocks and data wires.

Aggregate data type

An aggregate data type is an array or a cluster. See also **Scalar data type**.

Bytecode instruction

A bytecode instruction is a single operation that a program takes to modify data or access a feature of the NXT firmware. NXT programs are comprised of bytecode instructions, which are interpreted by the virtual machine. See also **Virtual machine**.

Refer to the **Codepsace** section of this document for information about the arrangement of bytecode instructions in an NXT program. Refer to the **Instruction Reference** section of this document for information about specific bytecode instructions.

Bytecode scheduling

Bytecode scheduling refers to the fact that `.RXE` files contain information that the virtual machine uses to decide when bytecode instructions and clumps run. See also **Bytecode instruction**, **Clump**, **Virtual machine**.

Clump

Clumps are batches of bytecode instructions which the virtual machine schedules to run. Clumps are loosely analogous to *tasks* in the RCX firmware, but the number of clumps is dynamic, up to a limit of 255.

By definition, a program is “running” when at least one of its clumps is active. Furthermore, the virtual machine can keep any number of clumps active in parallel. Clumps may be serialized (run in a pre-defined order) or parallelized depending on the output of the compiler.

In LEGO MINDSTORMS NXT 1.0 programming software, parallel loops are an example where at least two clumps may run in parallel. Subroutines, such as My Blocks in NXT-G, are also separate clumps of code.

Refer to the **Bytecode Scheduling** section of this document for information about how the virtual machine schedules clumps.

Clump Record

A clump record is a bookkeeping data structure that determines a given clump's state at run-time and its scheduling relationships with other clumps. Clump records stored in `.RXE` files contain only scheduling information. Run-time state information is maintained only while the program is loaded into RAM.

Refer to the **Clump Records** section of this document for more information about clump records.

Cluster

Clusters are dataspace structures similar to C-style structs. Clusters contain elements of various sub-types and can include nested clusters and/or arrays.

Codespace

All instructions in a program are stored in the codespace section of the file. In memory, the codespace is treated as an array of UWORD elements, or code words.

Refer to the **Codespace** section of this document for more information about the codespace.

Code word

A code word is a single 16-bit element of the codespace. Bytecode instructions are comprised of one or more code words.

Compatible Data Types

Many instructions accept arguments of various data types. If a given pair of arguments share the exact same data type or are easily convertible from one data type to the other, they are considered "compatible". For example, all scalar number data types are compatible with each other.

Refer to the **Polymorphic Instructions and Data Type Compatibility** for more information about compatible data types.

Dataspace

The dataspace is a pool of RAM in which all user data items reside. Most bytecode instructions operate on dataspace items via those items index (dataspace item ID) in the dataspace table of contents.

Refer to the **Dataspace** section of this document for more information about the dataspace.

Dataspace item

A dataspace item is an individually-addressable entity residing in the program's dataspace. These items may have simple scalar (numeric) data types, consist of aggregates (arrays or clusters) of one or more scalar sub-types, or have one of two special data types (VOID or MUTEX).

Dataspace item ID

Bytecode instruction arguments often take the form of dataspace item IDs. These arguments uniquely identify an addressable record in the dataspace table of contents. Refer to the **Argument Formats** section of this document for more information.

Dataspace table of contents

The dataspace table of contents (DSTOC) is a section of the executable file that describes the data types of all items in a programs dataspace. Most bytecode instruction arguments take the form of indexes into this table of contents. At run-time, this table also is used to calculate the actual location of data in RAM.

Refer to the **Dataspace Table of Contents** section of this document for more information about the DSTOC.

Dope vector

A dope vector (DV) is a data structure which describes an array in RAM. Each array in the dynamic dataspace has an associated dope vector.

Refer to the **Dope Vector** section of this document for more information about dope vectors.

Dynamic dataspace

The dynamic dataspace is the segment of RAM that contains array dataspace items. See also **Static dataspace**, **Aggregate data type**.

Instruction

See **Bytecode instruction**.

Mutex record

The virtual machine uses mutex records to control parallel access to subroutine clumps. Mutex records are stored in the static dataspace. See also **Virtual machine**, **Clump record**, **Static dataspace**.

NXT-G

The graphical programming language used by LEGO MINDSTORMS NXT Software is called NXT-G. It is based on National Instruments LabVIEW.

Opcode

Each instruction is identified via a unique one-byte operation code, or opcode. Common examples include `OP_ADD` (opcode `0x00`) and `OP_MOV` (opcode `0x1B`).

Scalar data type

A scalar data type is an integer. See also **Aggregate data type**.

Static dataspace

The static dataspace is the segment of RAM that contains dataspace items that are integers, mutexes, or dope vector indexes. See also **Dynamic dataspace**, **Mutex record**, **Dope vector**.

Type code

Type codes are used in the dataspace table of contents to specify the data type of a particular element. See also **Dataspace table of contents**.

Virtual machine

The virtual machine (VM) is the NXT firmware module responsible for running NXT programs. The VM reads `.RXE` files, manages the dataspace, and manages bytecode instructions. See also **Dataspace**, **Bytecode instruction**.