NQC Programmer's Guide

Version 2.2 r1, by Dave Baum
Contents

1 Introduction ..................................................................................................................1

2 The NQC Language .....................................................................................................2
  2.1 Lexical Rules ........................................................................................................2
    2.1.1 Comments ....................................................................................................2
    2.1.2 Whitespace ...................................................................................................3
    2.1.3 Numerical Constants ....................................................................................3
    2.1.4 Identifiers and Keywords .............................................................................3
  2.2 Program Structure ................................................................................................4
    2.2.1 Tasks ............................................................................................................4
    2.2.2 Functions ......................................................................................................5
    2.2.3 Subroutines ...................................................................................................8
    2.2.4 Variables ......................................................................................................9
    2.2.5 Arrays .........................................................................................................11
  2.3 Statements ..........................................................................................................11
    2.3.1 Variable Declaration ..................................................................................12
    2.3.2 Assignment .................................................................................................12
    2.3.3 Control Structures ......................................................................................13
    2.3.4 Access Control and Events ........................................................................15
    2.3.5 Other Statements ........................................................................................17
  2.4 Expressions and Conditions ...............................................................................18
    2.4.1 Expressions ................................................................................................18
    2.4.2 Conditions ..................................................................................................20
  2.5 The Preprocessor ................................................................................................21

Page i
3.9.1 RCX2 Events RCX2 ..........................................................................................42
3.9.2 Scout Events Scout......................................................................................47
3.10 Data Logging RCX ..........................................................................................49
3.11 General Features ............................................................................................50
3.12 RCX Specific Features ...................................................................................51
3.13 Scout Specific Features ..................................................................................52
3.14 CyberMaster Specific Features ......................................................................53
1 Introduction

NQC stands for Not Quite C, and is a simple language for programming several LEGO MINDSTORMS products. Some of the NQC features depend on which MINDSTORMS product you are using. This product is referred to as the target for NQC. Presently, NQC supports four different targets: RCX, CyberMaster, Scout, and RCX2 (an RCX running 2.0 firmware).

The preprocessor and control structures of NQC are very similar to C. NQC is not a general purpose language - there are many restrictions that stem from limitations of the targets.

Logically, NQC is defined as two separate pieces. The NQC language describes the syntax to be used in writing programs. The NQC API describes the system functions, constants, and macros that can be used by programs. This API is defined in a special file built in to the compiler. By default, this file is always processed before compiling a program.

This document describes both the NQC language and the NQC API. In short, it provides the information needed to write NQC programs. Since there are several different interfaces for NQC, this document does not describe how to use any specific NQC implementation. Refer to the documentation provided with the NQC tool, such as the NQC User Manual for information specific to that implementation.

For up-to-date information and documentation for NQC, visit the NQC Web Site at http://www.enteract.com/~dbbaum/nqc
2 The NQC Language

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

2.1 Lexical Rules

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

2.1.1 Comments

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

/* this is a comment */

/* this is a two
  line comment */

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

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

// a single line comment

Comments are ignored by the compiler. Their only purpose is to allow the programmer to document the source code.
2.1.2  Whitespace

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

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

Some of the C++ operators consist of multiple characters. In order to preserve these tokens whitespace must not be inserted within them. In the example below, the first line uses a right shift operator (’>>’), but in the second line the added space causes the ‘>’ symbols to be interpreted as two separate tokens and thus generate an error.

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

2.1.3  Numerical Constants

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

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

2.1.4  Identifiers and Keywords

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

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

```
__event_src    __type         acquire  break
__sensor       abs           asm      case
```
2.2  Program Structure

An NQC program is composed of code blocks and global variables. There are three distinct types of code blocks: tasks, inline functions, and subroutines. Each type of code block has its own unique features and restrictions, but they all share a common structure.

2.2.1  Tasks

The RCX implicitly supports multi-tasking, thus an NQC task directly corresponds to an RCX task. Tasks are defined using the `task` keyword using the following syntax:

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

The name of the task may be any legal identifier. A program must always have at least one task - named "main" - which is started whenever the program is run. The maximum number of tasks depends on the target - the RCX supports 10 tasks, CyberMaster supports 4, and Scout supports 6.

The body of a task consists of a list of statements. Tasks may be started and stopped using the `start` and `stop` statements (described in the section titled Statements). There is also an RCX API command, `StopAllTasks`, which stops all currently running tasks.
### 2.2.2 Functions

It is often helpful to group a set of statements together into a single function, which can then be called as needed. NQC supports functions with arguments, but not return values. Functions are defined using the following syntax:

```c
void name(argument_list)
{
    // body of the function
}
```

The keyword `void` is an artifact of NQC's heritage - in C functions are specified with the type of data they return. Functions that do not return data are specified to return `void`. Returning data is not supported in NQC, thus all functions are declared using the `void` keyword.

The argument list may be empty, or may contain one or more argument definitions. An argument is defined by its `type` followed by its `name`. Multiple arguments are separated by commas. All values in the RCX are represented as 16 bit signed integers. However NQC supports four different argument types which correspond to different argument passing semantics and restrictions:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>pass by value</td>
<td>none</td>
</tr>
<tr>
<td>const int</td>
<td>pass by value</td>
<td>only constants may be used</td>
</tr>
<tr>
<td>int&amp;</td>
<td>pass by reference</td>
<td>only variables may be used</td>
</tr>
<tr>
<td>const int &amp;</td>
<td>pass by reference</td>
<td>function cannot modify argument</td>
</tr>
</tbody>
</table>

Arguments of type `int` are passed by value from the calling function to the callee. This usually means that the compiler must allocate a temporary variable to hold the argument. There are no restrictions on the type of value that may be used. However, since the function is working with a copy of the actual argument, any changes it makes to the value will not be seen by the caller. In the example below, the function `foo` attempts to set the value of its argument to 2. This is perfectly legal, but since `foo` is working on a copy of the original argument, the variable `y` from main task remains unchanged.
void foo(int x)
{
    x = 2;
}

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

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

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

task main()
{

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

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

void foo(int &x)
{
    x = 2;
}
task main()
{
    int y = 1; // y is equal to 1
    foo(y);    // y is now equal to 2
    foo(2);    // error - only variables allowed
}

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

There is one important difference between int arguments and const int & arguments. An int argument is passed by value, so in the case of a dynamic expression (such as a sensor reading), the value is read once then saved. With const int & arguments, the expression will be re-read each time it is used in the function:

    void foo(int x)
    {
        if (x == x) // this will always be true
            PlaySound(SOUND_CLICK);
    }

    void bar(const int &x)
    {
        if (x == x) // may not be true..value could change
            PlaySound(SOUND_CLICK);
    }

    task main()
    {
        foo(SENSOR_1); // will play sound
        bar(2);        // will play sound
Functions must be invoked with the correct number (and type) of arguments. The example below shows several different legal and illegal calls to function foo:

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

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

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

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

### 2.2.3 Subroutines

Unlike inline functions, subroutines allow a single copy of some code to be shared between several different callers. This makes subroutines much more space efficient than inline functions, but due to some limitations in the RCX, subroutines have some significant restrictions. First of all, subroutines cannot use any arguments. Second, a subroutine cannot call another subroutine. Last, the maximum number of subroutines is limited to 8 for the RCX, 4 for CyberMaster, and 3 for Scout. In addition, if the subroutine is called from multiple tasks then it cannot have any local variables or perform calculations that require temporary variables (this restriction is lifted for the Scout and RCX2). These significant restrictions make subroutines less desirable than functions,
therefore their use should be minimized to those situations where the resultant savings in code size is absolutely necessary. The syntax for a subroutine appears below:

```
sub name()
{
    // body of subroutine
}
```

### 2.2.4 Variables

All variables in NQC are of the same type - specifically 16 bit signed integers. Variables are declared using the `int` keyword followed by a comma separated list of variable names and terminated by a semicolon (`;`). Optionally, an initial value for each variable may be specified using an equals sign (`=`) after the variable name. Several examples appear below:

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

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

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

```
int x;    // x is global

task main()
{
    int y;    // y is local to task main
    x = y;    // ok
    {
        // begin compound statement
        int z;    // local z declared
    }
}
```
\begin{verbatim}
y = z; // ok
}
y = z; // error - z no longer in scope
}

task foo()
{
    x = 1; // ok
    y = 2; // error - y is not global
}
\end{verbatim}

In many cases NQC must allocate one or more temporary variables for its own use. In some cases a temporary variable is used to hold an intermediate value during a calculation. In other cases it is used to hold a value as it is passed to a function. These temporary variables deplete the pool of variables available to the rest of the program. NQC attempts to be as efficient as possible with temporary variables (including reusing them when possible).

The RCX (and other targets) provide a number of storage locations which can be used to hold variables in an NQC program. There are two kinds of storage locations - global and local. When compiling a program, NQC assigns each variable to a specific storage location. Programmers for the most part can ignore the details of this assignment by following two basic rules:

- If a variable needs to be in a global location, declare it as a global variable.
- If a variable does not need to be a global variable, make it as local as possible.

This gives the compiler the most flexibility in assigning an actual storage location.

The number of global and local locations varies by target

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
Target & Global & Local \\
\hline
RCX & 32 & 0 \\
\hline
CyberMaster & 32 & 0 \\
\hline
\end{tabular}
\end{center}
2.2.5 Arrays

The RCX2 target supports arrays (the other targets do not have suitable support in firmware for arrays). Arrays are declared the same way as ordinary variables, but with the size of the array enclosed in brackets. The size must be a constant.

```c
int my_array[3]; // declare an array with three elements
```

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

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

Currently there are a number of limitations on how arrays can be used. These limitations will likely be removed in future versions of NQC:

- An array cannot be an argument to a function. An individual array element, however, can be passed to a function.

- Neither arrays nor their elements can be used with the increment (++) or decrement (--) operators.

- Only ordinary assignment (=) is allowed for array elements. The math assignments (i.e. +=) are not allowed.

- The initial values for an array's elements cannot be specified - an explicit assignment is required within the program itself to set the value of an element.

2.3 Statements

The body of a code block (task, function, or subroutine) is composed of statements. Statements are terminated with a semi-colon (‘;’).
2.3.1 Variable Declaration

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

    int variables;

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

    name[=expression]

Arrays of variables may also be declared (for the RCX2 only):

    int array[size];

2.3.2 Assignment

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

    variable assign_operator expression;

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Set variable to expression</td>
</tr>
<tr>
<td>+=</td>
<td>Add expression to variable</td>
</tr>
<tr>
<td>-=</td>
<td>Subtract expression from variable</td>
</tr>
<tr>
<td>*=</td>
<td>Multiple variable by expression</td>
</tr>
<tr>
<td>/=</td>
<td>Divide variable by expression</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Bitwise AND expression into variable</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>Set variable to sign (-1,+1,0) of expression</td>
</tr>
</tbody>
</table>

Some examples:

    x = 2;       // set x to 2
    y = 7;       // set y to 7
    x += y;      // x is 9, y is still 7
2.3.3 Control Structures

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

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

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

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

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

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

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

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

The syntax for a `while` loop appears below:

```
while (condition) body
```

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

```
while (x < 10)
{
```
A variant of the \texttt{while} loop is the \texttt{do-while} loop. Its syntax is:

\begin{verbatim}
d o \ \textit{body} \ \textit{while} \ (\textit{condition})
\end{verbatim}

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

Another kind of loop is the \texttt{for} loop:

\begin{verbatim}
\texttt{for (stmt1 ; condition ; stmt2) body}
\end{verbatim}

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

\begin{verbatim}
stmt1;
while (condition) 
{
\textit{body}
stmt2;
}
\end{verbatim}

The \texttt{repeat} statement executes a loop a specified number of times:

\begin{verbatim}
\texttt{repeat (expression) body}
\end{verbatim}

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

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

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

```c
    case constant_expression :
    default :
```

A typical switch statement might look like this:

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

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

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

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

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

### 2.3.4 Access Control and Events

The Scout and RCX2 support access control and event monitoring. Access control allows a task to request ownership of one or more resources. In NQC, access control is provided by the `acquire` statement, which has two forms:
where \textit{resources} is a constant that specified the resources to be acquired and \textit{body} and \textit{handler} are statements. The NQC API defines constants for individual resources which may be added together to request multiple resources at the same time. The behavior of the acquire statement is as follows: Ownership of the specified resources will be requested. If another task of higher priority already owns the resources, then the request will fail and execution will jump to the handler (if present). Otherwise, the request will succeed, and the body will begin to be executed. While executing the body, if another task of equal or higher priority requests any of the owned resources, then the original task will lose ownership. When ownership is lost, execution will jump to the handler (if present). Once the body has completed, the resources will be returned back to the system (so that lower priority tasks may acquire them), and execution will continue with the statement following the acquire statement. If a handler is not specified, then in both the case of a failed request, or a subsequent loss of ownership, control will pass to the statement following the acquire statement. For example, the following code acquires a resource for 10 seconds, playing a sound if it cannot complete successfully:

```nqc
acquire (ACQUIRE_OUT_A)
{
  Wait(1000);
}
catch
{
  PlaySound(SOUND_UP);
}
```

Event monitoring is implemented with the \textit{monitor} statement, which has a syntax very similar to acquire:

```nqc
monitor (events) body
```

\textit{Events} is a constant that determines which events should be monitored. For the Scout, events are predefined, so there are constants such as \texttt{EVENT_1_PRESSED} which can be used to specify events. With RCX2, the meaning of each event is configured by the
programmer. There are 16 events (numbers 0 to 15). In order to specify an event in a monitor statement, the event number must be converted to an event mask using the EVENT_MASK() macro. The Scout event constants or event masks may be added together to specify multiple events. Multiple masks should be combined using bitwise OR.

The monitor statement will execute the body while monitoring the specified events. If any of the events occur, execution will jump to the handler if present. If a handler is not present, then control will continue with the statement following the monitor statement. The following example waits for 10 seconds while monitoring events 2 and 3 for RCX2:

```plaintext
monitor( EVENT_MASK(2) | EVENT_MASK(3) )
{
    Wait(1000);
}
catch
{
    PlaySound(SOUND_UP);
}
```

Note that the acquire and monitor statements are only supported for targets that implement access control and event monitoring - specifically the Scout and RCX2.

### 2.3.5 Other Statements

A function (or subroutine) call is a statement of the form:

```plaintext
name(arguments);
```

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

Tasks may be started or stopped with the following statements:

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

```c
break;
continue;
```

It is possible to cause a function to return before it reaches the end of its code using the return statement.

```c
return;
```

Any expression is also a legal statement when terminated by a semicolon. It is rare to use such a statement since the value of the expression would then be discarded. The one notable exception is expressions involving the increment (++) or decrement (--) operators.

```c
x++;
```

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

## 2.4 Expressions and Conditions

In C there is no distinction between expressions and conditions. However, within NQC they are two syntactically different entities which are used in different situations. Expressions can be assigned to variables, used as arguments in a function, or as the count in a repeat statement. Conditions are used in most of the conditional control structures (if, while, etc.). Although it is possible to use an expression where a condition is expected, a condition cannot be substituted for an expression.

### 2.4.1 Expressions

Values are the most primitive type of expressions. More complicated expressions are formed from values using various operators. The NQC language only has two built in
kinds of values: numerical constants and variables. The RCX API defines other values corresponding to various RCX features such as sensors and timers.

Numerical constants in the RCX are represented as 16 bit signed integers. NQC internally uses 32 bit signed math for constant expression evaluation, then reduces to 16 bits when generating RCX code. Numeric constants can be written as either decimal (e.g. 123) or hexadecimal (e.g. 0xABC). Presently, there is very little range checking on constants, so using a value larger than expected may have unusual effects.

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
<th>Restriction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Absolute value</td>
<td>n/a</td>
<td></td>
<td>abs(x)</td>
</tr>
<tr>
<td>sign()</td>
<td>Sign of operand</td>
<td>n/a</td>
<td></td>
<td>sign(x)</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
<td>left</td>
<td>variables only</td>
<td>x++ or ++x</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td>left</td>
<td>variables only</td>
<td>x-- or --x</td>
</tr>
<tr>
<td>~</td>
<td>Unary minus</td>
<td>right</td>
<td></td>
<td>~x</td>
</tr>
<tr>
<td>~</td>
<td>Bitwise negation (unary)</td>
<td>right</td>
<td>constant only</td>
<td>~123</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>left</td>
<td></td>
<td>x * y</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>left</td>
<td></td>
<td>x / y</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
<td>left</td>
<td></td>
<td>123 % 4</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>left</td>
<td></td>
<td>x + y</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>left</td>
<td></td>
<td>x - y</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left shift</td>
<td>left</td>
<td>constant only</td>
<td>123 &lt;&lt; 4</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift</td>
<td>left</td>
<td>constant only</td>
<td>123 &gt;&gt; 4</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>left</td>
<td></td>
<td>x &amp; y</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>left</td>
<td>constant only</td>
<td>123 ^ 4</td>
</tr>
<tr>
<td></td>
<td>Bitwise OR</td>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>left</td>
<td>constant only</td>
<td>123 &amp;&amp; 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
<td>left</td>
</tr>
</tbody>
</table>

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

\[
x = 2 + 3 \times 4; \quad // \text{ set } x \text{ to } 14
y = (2 + 3) \times 4; \quad // \text{ set } y \text{ to } 20
\]

### 2.4.2 Conditions

Conditions are generally formed by comparing two expressions. There are also two constant conditions - `true` and `false` - which always evaluate to true or false respectively. A condition may be negated with the negation operator, or two conditions combined with the AND and OR operators. The table below summarizes the different types of conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>always true</td>
</tr>
<tr>
<td>false</td>
<td>always false</td>
</tr>
<tr>
<td>expr</td>
<td>true if expr is not equal to 0</td>
</tr>
<tr>
<td>expr1 == expr2</td>
<td>true if expr1 equals expr2</td>
</tr>
<tr>
<td>expr1 != expr2</td>
<td>true if expr1 is not equal to expr2</td>
</tr>
<tr>
<td>expr1 &lt; expr2</td>
<td>true if one expr1 is less than expr2</td>
</tr>
<tr>
<td>expr1 &lt;= expr2</td>
<td>true if expr1 is less than or equal to expr2</td>
</tr>
<tr>
<td>expr1 &gt; expr2</td>
<td>true if expr1 is greater than expr2</td>
</tr>
<tr>
<td>expr1 &gt;= expr2</td>
<td>true if expr1 is greater than or equal to expr2</td>
</tr>
<tr>
<td>! condition</td>
<td>logical negation of a condition - true if condition is false</td>
</tr>
<tr>
<td>cond1 &amp;&amp; cond2</td>
<td>logical AND of two conditions (true if and only if both conditions are true)</td>
</tr>
<tr>
<td>cond1</td>
<td></td>
</tr>
</tbody>
</table>
2.5 The Preprocessor

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

2.5.1 \#include

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

    \#include "foo.nqh" // ok
    \#include <foo.nqh> // error!

2.5.2 \#define

The \#define command is used for simple macro substitution. Redefinition of a macro is an error (unlike in C where it is a warning). Macros are normally terminated by the end of the line, but the newline may be escaped with the backslash (\") to allow multi-line macros:

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

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

2.5.3 Conditional Compilation

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


# if condition
# ifdef symbol
# ifndef symbol
#else
        #elif condition
        #endif

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

## 2.5.4 Program Initialization

The compiler will insert a call to a special initialization function, _init, at the start of a program. This default function is part of the RCX API and sets all three outputs to full power in the forward direction (but still turned off). The initialization function can be disabled using the #pragma noinit directive:

```
#pragma noinit // don't do any program initialization
```

The default initialization function can be replaced with a different function using the #pragma init directive:

```
#pragma init function // use custom initialization
```

## 2.5.5 Reserving Storage

The NQC compiler automatically assigned variables to storage locations. However, sometimes it is necessary to prevent the compiler from using certain storage locations. This can be done with the #pragma reserve directive:

```
#pragma reserve start
#pragma reserve start end
```

This directive forces the compiler to ignore one or more storage locations during variable assignment. Start and end must be numbers that refer to valid storage locations. If only a start is provided, then that single location is reserved. If start and end are both specified, then the range of locations from start to end (inclusive) are reserved. The most common use of this directive is to reserve locations 0 and/or 1 when using counters for RCX2.
This is because the RCX2 counters are overlapped with storage locations 0 and 1. For example, if both counters were going to be used:

    #pragma reserve 0 1
3  NQC API

The NQC API defines a set of constants, functions, values, and macros that provide access to various capabilities of the target such as sensors, outputs, timers, and communication. Some features are only available on certain targets. Where appropriate, a section's title will indicate which targets it applies to. The RCX2 is a superset of RCX features, so if RCX is listed, then the feature works with both the original firmware and 2.0 firmware. If RCX2 is listed, then the feature only applies to the 2.0 firmware.

The API consists of functions, values, and constants. A function is something that can be called as a statement. Typically it takes some action or configures some parameter. Values represent some parameter or quantity and can be used in expressions. Constants are symbolic names for values that have special meanings for the target. Often, a set of constants will be used in conjunction with a function. For example, the PlaySound function takes a single argument which determines which sound is to be played. Constants, such as SOUND_UP, are defined for each sound.

3.1  Sensors

There are three sensors, which internally are numbered 0, 1, and 2. This is potentially confusing since they are externally labeled as sensors 1, 2, and 3. To help mitigate this confusion, the sensor names SENSOR_1, SENSOR_2, and SENSOR_3 have been defined. These sensor names may be used in any function that requires a sensor as an argument. Furthermore, these names may also be used whenever a program wishes to read the current value of the sensor:

\[
x = \text{SENSOR}_1; \quad \text{// read sensor and store value in } x
\]

3.1.1  Types and Modes

RCX, CyberMaster

The sensor ports on the RCX are capable of interfacing to a variety of different sensors (other targets don’t support configurable sensor types). It is up to the program to tell the RCX what kind of sensor is attached to each port. A sensor's type may be configured by calling SetSensorType. There are four sensor types, each corresponding to a specific
LEGO sensor. A fifth type (SENSOR_TYPE_NONE) can be used for reading the raw values of generic passive sensors. In general, a program should configure the type to match the actual sensor. If a sensor port is configured as the wrong type, the RCX may not be able to read it accurately.

<table>
<thead>
<tr>
<th>Sensor Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR_TYPE_NONE</td>
<td>generic passive sensor</td>
</tr>
<tr>
<td>SENSOR_TYPE_TOUCH</td>
<td>a touch sensor</td>
</tr>
<tr>
<td>SENSOR_TYPE_TEMPERATURE</td>
<td>an temperature sensor</td>
</tr>
<tr>
<td>SENSOR_TYPE_LIGHT</td>
<td>a light sensor</td>
</tr>
<tr>
<td>SENSOR_TYPE_ROTATION</td>
<td>a rotation sensor</td>
</tr>
</tbody>
</table>

Both the RCX and CyberMaster allow a sensor to be configured in different modes. The sensor mode determines how a sensor's raw value is processed. Some modes only make sense for certain types of sensors, for example SENSOR_MODE_ROTATION is useful only with rotation sensors. The sensor mode can be set by calling SetSensorMode. The possible modes are shown below. Note that since CyberMaster does not support temperature or rotation sensors, the last three modes are restricted to the RCX only.

<table>
<thead>
<tr>
<th>Sensor Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR_MODE_RAW</td>
<td>raw value from 0 to 1023</td>
</tr>
<tr>
<td>SENSOR_MODE_BOOL</td>
<td>boolean value (0 or 1)</td>
</tr>
<tr>
<td>SENSOR_MODE_EDGE</td>
<td>counts number of boolean transitions</td>
</tr>
<tr>
<td>SENSOR_MODE_PULSE</td>
<td>counts number of boolean periods</td>
</tr>
<tr>
<td>SENSOR_MODE_PERCENT</td>
<td>value from 0 to 100</td>
</tr>
<tr>
<td>SENSOR_MODE_FAHRENHEIT</td>
<td>degrees F - RCX only</td>
</tr>
<tr>
<td>SENSOR_MODE_CELSIUS</td>
<td>degrees C - RCX only</td>
</tr>
<tr>
<td>SENSOR_MODE_ROTATION</td>
<td>rotation (16 ticks per revolution) - RCX only</td>
</tr>
</tbody>
</table>

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

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

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

**SetSensor(sensor, configuration)**  
Function - RCX

Set the type and mode of the given sensor to the specified configuration, which must be a special constant containing both type and mode information.

```
SetSensor(SENSOR_1, SENSOR_TOUCH);
```

**SetSensorType(sensor, type)**  
Function - RCX

Set a sensor's type, which must be one of the predefined sensor type constants.

```
SetSensorType(SENSOR_1, SENSOR_TYPE_TOUCH);
```

**SetSensorMode(sensor, mode)**  
Function - RCX, CyberMaster

Set a sensor's mode, which should be one of the predefined sensor mode constants. A slope parameter for boolean conversion, if desired, may be added to the mode (RCX only).
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW); // raw mode
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW + 10); // slope 10

ClearSensor(sensor)

Clear the value of a sensor - only affects sensors that are configured to measure a cumulative quantity such as rotation or a pulse count.

ClearSensor(SENSOR_1);

3.1.2 Sensor Information

There are a number of values that can be inspected for each sensor. For all of these values the sensor must be specified by its sensor number (0, 1, or 2), and not a sensor name (e.g. SENSOR_1).

SensorValue(n)

Returns the processed sensor reading for sensor n, where n is 0, 1, or 2. This is the same value that is returned by the sensor names (e.g. SENSOR_1).

\[ x = \text{SensorValue}(0); \] // read sensor 1

SensorType(n)

Returns the configured type of sensor n, which must be 0, 1, or 2. Only the RCX has configurable sensors types, other targets will always return the pre-configured type of the sensor.

\[ x = \text{SensorType}(0); \]

SensorMode(n)

Returns the current sensor mode for sensor n, which must be 0, 1, or 2.

\[ x = \text{SensorMode}(0); \]

SensorValueBool(n)

Returns the boolean value of sensor n, which must be 0, 1, or 2. Boolean conversion is either done based on preset cutoffs, or a slope parameter specified by calling SetSensorMode.
\[
x = \text{SensorValueBool}(0);
\]

**SensorValueRaw(n)**

Returns the raw value of sensor \(n\), which must be 0, 1, or 2. Raw values may range from 0 to 1023.

\[
x = \text{SensorValueRaw}(0);
\]

### 3.1.3 Scout Light Sensor

On the Scout, `SENSOR_3` refers to the built-in light sensor. Reading the light sensor's value (with `SENSOR_3`) will return one of three levels: 0 (dark), 1 (normal), or 2 (bright). The sensor's raw value can be read with `SensorValueRaw(SENSOR_3)`, but bear in mind that brighter light will result in a **lower** raw value. The conversion of the sensor's raw value (between 0 and 1023) to one of the three levels depends on three parameters: **lower limit**, **upper limit**, and **hysteresis**. The lower limit is the smallest (brightest) raw value that is still considered **normal**. Values below the lower limit will be considered **bright**. The upper limit is the largest (darkest) raw value that is considered **normal**. Values about this limit are considered **dark**.

Hysteresis can be used to prevent the level from changing when the raw value hovers near one of the limits. This is accomplished by making it a little harder to leave the dark and bright states than it is to enter them. Specifically, the limit for moving from normal to bright will be a little lower than the limit for moving from bright back to normal. The difference between these two limits is the amount of hysteresis. A symmetrical case holds for the transition between normal and dark.

**SetSensorLowerLimit(value)**

Set the light sensor's lower limit. Value may be any expression.

\[
\text{SetSensorLowerLimit}(100);
\]

**SetSensorUpperLimit(value)**

Set the light sensor's upper limit. Value may be any expression.

\[
\text{SetSensorUpperLimit}(900);
\]
SetSensorHysteresis (value)  

Set the light sensor's hysteresis. Value may be any expression.

SetSensorLowerLimit (20);  

CalibrateSensor()  

Reads the current value of the light sensor, then sets the upper and lower limits to 12.5% above and below the current reading, and sets the hysteresis to 3.12% of the reading.

CalibrateSensor();  

3.2 Outputs

3.2.1 Primitive Calls

All of the functions dealing with outputs take a set of outputs as their first argument. This set must be a constant. The names OUT_A, OUT_B, and OUT_C are used to identify the three outputs. Multiple outputs can be combined by adding individual outputs together. For example, use OUT_A+OUT_B to specify outputs A and B together. The set of outputs must always be a compile time constant (it cannot be a variable).

Each output has three different attributes: mode, direction, and power level. The mode can be set by calling SetOutput(output, mode). The mode parameter should be one of the following constants:

<table>
<thead>
<tr>
<th>Output Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT_OFF</td>
<td>output is off (motor is prevented from turning)</td>
</tr>
<tr>
<td>OUT_ON</td>
<td>output is on (motor will be powered)</td>
</tr>
<tr>
<td>OUT_FLOAT</td>
<td>motor can &quot;coast&quot;</td>
</tr>
</tbody>
</table>

The other two attributes, direction and power level, may be set at any time, but only have an effect when the output is on. The direction is set with the SetDirection(output, direction) command. The direction parameter should be one of the following constants:
# Direction Meaning

<table>
<thead>
<tr>
<th>Direction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT_FWD</td>
<td>Set to forward direction</td>
</tr>
<tr>
<td>OUT_REV</td>
<td>Set to reverse direction</td>
</tr>
<tr>
<td>OUT_TOGGLE</td>
<td>Switch direction to the opposite of what it is presently</td>
</tr>
</tbody>
</table>

The power level can range 0 (lowest) to 7 (highest). The names `OUT_LOW`, `OUT_HALF`, and `OUT_FULL` are defined for use in setting power level. The level is set using the `SetPower(outputs, power)` function.

Be default, all three motors are set to full power and the forward direction (but still turned off) when a program starts.

**SetOutput(outputs, mode)**

Set the outputs to the specified mode. Outputs is one or more of `OUT_A`, `OUT_B`, and `OUT_C`. Mode must be `OUT_ON`, `OUT_OFF`, or `OUT_FLOAT`.

```
SetOutput (OUT_A + OUT_B, OUT_ON);  // turn A and B on
```

**SetDirection(outputs, direction)**

Set the outputs to the specified direction. Outputs is one or more of `OUT_A`, `OUT_B`, and `OUT_C`. Direction must be `OUT_FWD`, `OUT_REV`, or `OUT_FLIP`.

```
SetDirection(OUT_A, OUT_REV);  // make A turn backwards
```

**SetPower(outputs, power)**

Sets the power level of the specified outputs. Power may be an expression, but should result in a value between 0 and 7. The constants `OUT_LOW`, `OUT_HALF`, and `OUT_FULL` may also be used.

```
SetPower(OUT_A, OUT_FULL);  // A full power
SetPower(OUT_B, x);
```

**OutputStatus(n)**

Returns the current output setting for motor n. Note that n must be 0, 1, or 2 - not `OUT_A`, `OUT_B`, or `OUT_C`.
3.2.2 Convenience Calls

Since control of outputs is such a common feature of programs, a number of convenience
functions are provided that make it easier to work with the outputs. It should be noted
that these commands do not provide any new functionality above the SetOutput and
SetDirection commands. They are merely convenient ways to make programs more
concise.

**On(outputs) Function - All**

Turn specified outputs on. Outputs is one or more of OUT_A, OUT_B, and OUT_C
added together.

```
On(OUT_A + OUT_C); // turn on outputs A and C
```

**Off(outputs) Function - All**

Turn specified outputs off. Outputs is one or more of OUT_A, OUT_B, and OUT_C
added together.

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

**Float(outputs) Function - All**

Make outputs float. Outputs is one or more of OUT_A, OUT_B, and OUT_C added
together.

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

**Fwd(outputs) Function - All**

Set outputs to forward direction. Outputs is one or more of OUT_A, OUT_B, and
OUT_C added together.

```
Fwd(OUT_A);
```

**Rev(outputs) Function - All**
Set outputs to reverse direction. Outputs is one or more of OUT_A, OUT_B, and OUT_C added together.

Rev (OUT_A);

**Toggle(outputs)**  
Function - All

Flip the direction of the outputs. Outputs is one or more of OUT_A, OUT_B, and OUT_C added together.

Toggle(OUT_A);

**OnFwd(outputs)**  
Function - All

Set outputs to forward direction and turn them on. Outputs is one or more of OUT_A, OUT_B, and OUT_C added together.

OnFwd (OUT_A);

**OnRev(outputs)**  
Function - All

Set outputs to reverse direction and turn them on. Outputs is one or more of OUT_A, OUT_B, and OUT_C added together.

OnRev (OUT_A);

**OnFor(outputs, time)**  
Function - All

Turn outputs on for a specified amount of time, then turn them off. Outputs is one or more of OUT_A, OUT_B, and OUT_C added together. Time is measures in 10ms increments (one second = 100) and may be any expression.

OnFor (OUT_A, x);

### 3.2.3 Global Control  
**RCX2, Scout**

SetGlobalOutput(outputs, mode)  
Function - RCX2, Scout

Disable or re-enable outputs depending on the mode parameter. If mode is OUT_OFF, then the outputs will be turned off and disabled. While disabled any subsequent calls to SetOutput() (including convenience functions such as On()) will be ignored.
Using a mode of `OUT_FLOAT` will put the outputs in float mode before disabling them. Outputs can be re-enabled by calling `SetGlobalOutput()` with a mode of `OUT_ON`. Note that enabling an output doesn’t immediately turn it on - it just allows future calls to `SetOutput()` to take effect.

```c
SetGlobalOutput(OUT_A, OUT_OFF); // disable output A
SetGlobalOutput(OUT_A, OUT_ON);  // enable output A
```

**SetGlobalDirection(outputs, direction)**  
Function - RCX2, Scout

Reverses or restores the directions of outputs. The direction parameter should be `OUT_FWD`, `OUT_REV`, or `OUT_FLIP`. Normal behavior is a global direction of `OUT_FWD`. When the global direction is `OUT_REV`, then the actual output direction will be the opposite of whatever the regular output calls request. Calling `SetGlobalDirection()` with `OUT_FLIP` will switch between normal and opposite behavior.

```c
SetGlobalDirection(OUT_A, OUT_REV); // opposite direction
SetGlobalDirection(OUT_A, OUT_FWD); // normal direction
```

**SetMaxPower(outputs, power)**  
Function - RCX2, Scout

Sets the maximum power level allowed for the outputs. The power level may be a variable, but should have a value between `OUT_LOW` and `OUT_FULL`.

```c
SetMaxPower(OUT_A, OUT_HALF);
```

**GlobalOutputStatus(n)**  
Value - RCX2, Scout

Returns the current global output setting for motor n. Note that n must be 0, 1, or 2 - not `OUT_A`, `OUT_B`, or `OUT_C`.

```c
x = GlobalOutputStatus(0);  // global status of OUT_A
```

### 3.3 Sound

**PlaySound(sound)**  
Function - All
Plays one of the 6 preset RCX sounds. The sound argument must be a constant. The following constants are pre-defined for use with \texttt{PlaySound}: \texttt{SOUND\_CLICK}, \texttt{SOUND\_DOUBLE\_BEEP}, \texttt{SOUND\_DOWN}, \texttt{SOUND\_UP}, \texttt{SOUND\_LOW\_BEEP}, \texttt{SOUND\_FAST\_UP}.

\texttt{PlaySound(SOUND\_CLICK)};

\textbf{PlayTone}(\texttt{frequency, duration}) \quad \textbf{Function - All}

Plays a single tone of the specified frequency and duration. The frequency is in Hz and can be a variable for RCX2 and Scout, but has to be constant for RCX and CyberMaster. The duration is in 100ths of a second and must be a constant.

\texttt{PlayTone(440, 50);} \quad // Play 'A' for one half second

\textbf{MuteSound}() \quad \textbf{Function - RCX2, Scout}

Stops all sounds and tones from being played.

\texttt{MuteSound();}

\textbf{UnmuteSound}() \quad \textbf{Function - RCX2, Scout}

Restores normal operation of sounds and tones.

\texttt{UnmuteSound();}

\textbf{ClearSound}() \quad \textbf{Function - RCX2}

Removes any pending sounds from the sound buffer.

\texttt{ClearSound();}

\textbf{SelectSounds}(\texttt{group}) \quad \textbf{Function - Scout}

Selects which group of system sounds should be used. The group must be a constant.

\texttt{SelectSounds(0);}

\section{3.4 LCD Display}

The RCX has seven different display modes as shown below. The RCX defaults to \texttt{DISPLAY\_WATCH}. 

\texttt{DISPLAY\_WATCH, DISPLAY\_TEXT, DISPLAY\_DATA, DISPLAY\_GRAPH, DISPLAY\_ALARM, DISPLAY\_DIAG, DISPLAY\_ERROR}.
The RCX2 adds an eighth display mode - DISPLAY_USER. User display mode continuously reads a source value and updates the display. It can optionally display a decimal point at any position within the number. This allows the display to give the illusion of working with fractions even though all values are stored internally as integers. For example, the following call will set the user display to show the value 1234 with two digits appearing after the decimal point, resulting in "12.34" appearing on the LCD.

```c
SetDisplayUser(1234, 2);
```

The following short program illustrates the update of the user display:

```c
int x;

task main()
{
    ClearTimer(0);
    SetUserDisplay(Timer(0), 0);
    until(false);
}
```

Because the user display mode continuously updates the LCD, there are certain restrictions on the source value. If a variable is used it must be assigned to a global storage location. The best way to ensure this is to make the variable a global one. There can also be some strange side effects. For example, if a variable is being displayed and later used as the target of a calculation, it is possible for the display to show some intermediate results of the calculation:

```c
int x;

// This is incorrect:
int y = 1234;

task main()
{
    // Now y will show 1234.4 because of intermediate calculation.
}
```
SetUserDisplay(x, 0);
while(true)
{
    // display may briefly show 1!
    x = 1 + Timer(0);
}

SelectDisplay(mode) Function - RCX

Select a display mode.

SelectDisplay(DISPLAY_SENSOR_1); // view sensor 1

SetUserDisplay(value, precision) Function - RCX2

Set the LCD display to continuously monitor the specified value. Precision specifies
the number of digits to the right of the decimal point. A precision of zero shows no
decimal point.

SetUserDisplay(Timer(0), 0); // view timer 0

3.5 Communication

3.5.1 Messages RCX, Scout

The RCX and Scout can send and receive simple messages using IR. A message can
have a value from 0 to 255, but the use of message 0 is discouraged. The most recently
received message is remembered and can be accessed as Message(). If no message has
been received, Message() will return 0.

ClearMessage() Function - RCX, Scout

Clear the message buffer. This facilitates detection of the next received message
since the program can then wait for Message() to become non-zero:

ClearMessage(); // clear out the received message
until(Message() > 0); // wait for next message
SendMessage(message) Function - RCX, Scout

Send an IR message. Message may be any expression, but the RCX can only send messages with a value between 0 and 255, so only the lowest 8 bits of the argument are used.

```
SendMessage(3); // send message 3
SendMessage(259); // another way to send message 3
```

SetTxPower(power) Function - RCX, Scout

Set the power level for IR transmission. Power should be one of the constants TX_POWER_LO or TX_POWER_HI.

3.5.2 Serial RCX2

The RCX2 can transmit serial data out the IR port. Prior to transmitting any data, the communication and packet settings must be specified. Then, for each transmission, data should be placed in the transmit buffer, then send using the SendSerial() function.

The communication settings are set with SetSerialComm, and determine how bits are sent over IR. Possible values are shown below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIAL_COMM_DEFAULT</td>
<td>default settings</td>
</tr>
<tr>
<td>SERIAL_COMM_4800</td>
<td>4800 baud</td>
</tr>
<tr>
<td>SERIAL_COMM_DUTY25</td>
<td>25% duty cycle</td>
</tr>
<tr>
<td>SERIAL_COMM_76KHZ</td>
<td>76kHz carrier</td>
</tr>
</tbody>
</table>

The default is to send data at 2400 baud using a 50% duty cycle on a 38kHz carrier. To specify multiple options (such as 4800 baud with 25% duty cycle), combine the individual options using bitwise or (SERIAL_COMM_4800 | SERIAL_COMM_DUTY25).

The packet settings are set with SetSerialPacket and control how bytes are assembled into packets. Possible values are shown below.
<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIAL_PACKET_DEFAULT</td>
<td>no packet format - just data bytes</td>
</tr>
<tr>
<td>SERIAL_PACKET_PREAMBLE</td>
<td>send a packet preamble</td>
</tr>
<tr>
<td>SERIAL_PACKET_NEGATED</td>
<td>follow each byte with its complement</td>
</tr>
<tr>
<td>SERIAL_PACKET_CHECKSUM</td>
<td>include a checksum for each packet</td>
</tr>
<tr>
<td>SERIAL_PACKET_RCX</td>
<td>standard RCX format (preamble, negated data, and checksum)</td>
</tr>
</tbody>
</table>

Note that negated packets always include a checksum, so the `SERIAL_PACKET_CHECKSUM` option is only meaningful when `SERIAL_PACKET_NEGATED` is not specified. Likewise the preamble, negated, and checksum settings are implied by `SERIAL_PACKET_RCX`.

The transmit buffer can hold up to 16 data bytes. These bytes may be set using `SetSerialData`, then transmitted by calling `SendSerial`. For example, the following code sends two bytes (0x12 and 0x34) out the serial port:

```c
SetSerialComm(SERIAL_COMM_DEFAULT);
SetSerialPacket(SERIAL_PACKET_DEFAULT);
SetSerialData(0, 0x12);
SetSerialData(1, 0x34);
SendSerial(0, 2);
```

**SetSerialComm(settings) Function - RCX2**

Set the communication settings, which determine how the bits are sent over IR

```c
SetSerialComm(SERIAL_COMM_DEFAULT);
```

**SetSerialPacket(settings) Function - RCX2**

Set the packet settings, which control how bytes are assembled into packets.

```c
SetSerialPacket(SERIAL_PACKET_DEFAULT);
```

**SetSerialData(n, value) Function - RCX2**
Set one byte of data in the transmit buffer. N is the index of the byte to set (0-15),
and value can be any expression.

\[
\text{SetSerialData}(3, x); \quad // \text{set byte 3 to } x
\]

**SerialData(n)**

Returns the value of a byte in the transmit buffer (NOT received data). N must be a
constant between 0 and 15.

\[
x = \text{SerialData}(7); \quad // \text{read byte } #7
\]

**SendSerial(start, count)**

Use the contents of the transmit buffer to build a packet and send it out the IR port
(according to the current packet and communication settings). Start and count are
both constants that specify the first byte and the number of bytes within the buffer to
be sent.

\[
\text{SendSerial}(0, 2); \quad // \text{send first two bytes in buffer}
\]

### 3.5.3 VLL

**SendVLL(value)**

Sends a Visible Light Link (VLL) command, which can be used to communicate with
the MicroScout or Code Pilot. The specific VLL commands are described in the
Scout SDK.

\[
\text{SendVLL}(4); \quad // \text{send VLL command #4}
\]

### 3.6 Timers

All targets provide several independent timers with 100ms resolution (10 ticks per
second). The scout provides 3 such timers while the RCX and CyberMaster provide 4.
The timers wrap around to 0 after 32767 ticks (about 55 minutes). The value of a timer
can be read using \text{Timer}(n), where n is a constant that determines which timer to use (0-2
for Scout, 0-3 for the others). RCX2 provides the ability to read the same timers with
higher resolution by using `FastTimer(n)`, which returns the timer’s value with 10ms resolution (100 ticks per second).

**ClearTimer(n)**  
Function - All  
Reset the specified timer to 0.

```
ClearTimer(0);
```

**Timer(n)**  
Value - All  
Return the current value of specified timer (in 100ms resolution).

```
x = Timer(0);
```

**SetTimer(n, value)**  
Function - RCX2  
Set a timer to a specific value (which may be any expression).

```
SetTimer(0, x);
```

**FastTimer(n)**  
Value - RCX2  
Return the current value of specified timer in 10ms resolution.

```
x = FastTimer(0);
```

### 3.7 Counters  
**RCX2, Scout**

Counters are like very simple variables that can be incremented, decremented, and cleared. The Scout provides two counters (0 and 1), while the RCX2 provides three (0, 1, and 2). In the case of RCX2, these counters are overlapped with global storage locations 0-2, so if they are going to be used as counters, a `#pragma reserve` should be used to prevent NQC from using the storage location for a regular variable. For example, to use counter 1:

```
#pragma reserve 1
```

**ClearCounter(n)**  
Function - RCX2, Scout  
Reset counter n to 0. N must be 0 or 1 for Scout, 0-2 for RCX2.

```
ClearCounter(1);
```
**IncCounter(n)**  
Function - RCX2, Scout

Increment counter n by 1. N must be 0 or 1 for Scout, 0-2 for RCX2

```
IncCounter(1);
```

**DecCounter(n)**  
Function - RCX2, Scout

Decrement counter n by 1. N must be 0 or 1 for Scout, 0-2 for RCX2

```
DecCounter(1);
```

**Counter(n)**  
Value - RCX, Scout

Return the current value of counter n. N must be 0 or 1 for Scout, 0-3 for RCX2

```
x = Counter(1);
```

### 3.8 Access Control

Access control is implemented primarily by the `acquire` statement. The `SetPriority` function can be used to set a task's priority, and the following constants may be used to specify resources in an `acquire` statement. Note that the user defined resources are only available on the RCX2.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACQUIRE_OUT_A,</td>
<td>outputs</td>
</tr>
<tr>
<td>ACQUIRE_OUT_B,</td>
<td></td>
</tr>
<tr>
<td>ACQUIRE_OUT_C</td>
<td></td>
</tr>
<tr>
<td>ACQUIRE_SOUND</td>
<td>sound</td>
</tr>
<tr>
<td>ACQUIRE_USER_1,</td>
<td>user defined -</td>
</tr>
<tr>
<td>ACQUIRE_USER_2,</td>
<td>RCX2 only</td>
</tr>
<tr>
<td>ACQUIRE_USER_3,</td>
<td></td>
</tr>
<tr>
<td>ACQUIRE_USER_4</td>
<td></td>
</tr>
</tbody>
</table>

**SetPriority(p)** - Function - RCX2, Scout
Set a task’s priority to p, which must be a constant. RCX2 supports priorities 0-255, while Scout supports priorities 0-7. Note that lower numbers are higher priority.

```c
SetPriority(1);
```

## 3.9 Events

### RCX2, Scout

Although the RCX2 and Scout share a common event mechanism, the RCX2 provides 16 completely configurable events while the Scout has 15 predefined events. The only functions common to both targets are the commands to inspect or force events.

### ActiveEvents(task) Value - RCX2, Scout

Return the set of events that have been triggered for a given task.

```c
x = ActiveEvents(0);
```

### Event(events) Function - RCX2, Scout

Manually triggers the specified events. This can be useful in testing event handling of the program, or in other cases simulating an event based on other criteria. Note that the specification of the events themselves is slightly different between RCX2 and Scout. RCX2 uses the `EVENT_MASK` macro to compute an event mask, while Scout has predefined masks.

```c
Event(EVENT_MASK(3)); // triggering an RCX2 event
Event(EVENT_1_PRESSED); // triggering a Scout event
```

### 3.9.1 RCX2 Events

RCX2 has provides an extremely flexible event system. There are 16 events, each of which can be mapped to one of several event sources (the stimulus that can trigger the event), and an event type (the criteria for triggering). A number of other parameters may also be specified depending on the event type. For all of the configuration calls an event is identified by its event number - a constant from 0 to 15.

Legal event sources are sensors, timers, counters, or the message buffer. An event is configured by calling `SetEvent(event, source, type)`, where event is a constant...
event number (0-15), source is the event source itself, and type is one of the types shown below (some combinations of sources and types are illegal).

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Condition</th>
<th>Event Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_TYPE_PRESSED</td>
<td>value becomes on</td>
<td>sensors only</td>
</tr>
<tr>
<td>EVENT_TYPE_RELEASED</td>
<td>value becomes off</td>
<td>sensors only</td>
</tr>
<tr>
<td>EVENT_TYPE_PULSE</td>
<td>value goes from off to on to off</td>
<td>sensors only</td>
</tr>
<tr>
<td>EVENT_TYPE_EDGE</td>
<td>value goes from on to off or vice versa</td>
<td>sensors only</td>
</tr>
<tr>
<td>EVENT_TYPE_FASTCHANGE</td>
<td>value changes rapidly</td>
<td>sensors only</td>
</tr>
<tr>
<td>EVENT_TYPE_LOW</td>
<td>value becomes low</td>
<td>any</td>
</tr>
<tr>
<td>EVENT_TYPE_NORMAL</td>
<td>value becomes normal</td>
<td>any</td>
</tr>
<tr>
<td>EVENT_TYPE_HIGH</td>
<td>value becomes high</td>
<td>any</td>
</tr>
<tr>
<td>EVENT_TYPE_CLICK</td>
<td>value from low to high back to low</td>
<td>any</td>
</tr>
<tr>
<td>EVENT_TYPE_DOUBLECLICK</td>
<td>two clicks within a certain time</td>
<td>any</td>
</tr>
<tr>
<td>EVENT_TYPE_MESSAGE</td>
<td>new message received</td>
<td>Message() only</td>
</tr>
</tbody>
</table>

The first four event types make use of a sensor’s boolean value, thus are most useful with touch sensors. For example, to set event #2 to be triggered when a touch sensor on port 1 is pressed, the following call could be made:

```c
SetEvent(2, SENSOR_1, EVENT_TYPE_PRESSED);
```

EVENT_TYPE_FASTCHANGE should be used with sensors that have been configured with a slope parameter. When the raw value changes faster than the slope parameter an EVENT_TYPE_FASTCHANGE event will be triggered.

The next three types (EVENT_TYPE_LOW, EVENT_TYPE_NORMAL, and EVENT_TYPE_HIGH) convert an event sources value into one of three ranges (low,
normal, or high), and trigger an event when the value moves from one range into another. The ranges are defined by the lower limit and upper limit for the event. When the source value is lower than the lower limit, the source is considered low. When the source value is higher than the upper limit, the source is considered high. The source is normal whenever it is between the limits.

The following example configures event #3 to trigger when the sensor on port 2's value goes into the high range. The upper limit is set for 80, and the lower limit is set for 50. This configuration is typical of how an event can be triggered when a light sensor detected a bright light.

```
SetEvent(3, SENSOR_2, EVENT_TYPE_HIGH);
SetLowerLimit (3, 50);
SetUpperLimit (3, 80);
```

A hysteresis parameter can be used to provide more stable transitions in cases where the source value may jitter. Hysteresis works by making the transition from low to normal a little higher than the transition from normal to low. In a sense, it makes it easier to get into the low range than get out of it. A symmetrical case applies to the transition between normal and high.

A transition from low to high back to low will trigger a EVENT_TYPE_CLICK event, provided that the entire sequence is faster than the click time for the event. If two successive clicks occur and the time between clicks is also less than the click time, then an EVENT_TYPE_DOUBLECLICK event will be triggered. The system also keeps track of the total number of clicks for each event.

The last event type, EVENT_TYPE_MESSAGE, is only valid when Message() is used as the event source. The event will be triggered whenever a new message arrives (even if its value is the same as a previous message).

The monitor statement and some API functions (such as ActiveEvents() or Event()) need to handle multiple events. This is done by converting each event number to an event mask, and then combining the masks with a bitwise OR. The EVENT_MASK(event) macro converts an event number to a mask. For example, to monitor events 2 and 3, the following statement could be used:
monitor(EVENT_MASK(2) | EVENT_MASK(3))

**SetEvent(event, source, type) Function - RCX2**

Configure an event (a number from 0 to 15) to use the specified source and type. Both event and type must be constants, and source should be the actual source expression.

```
SetEvent(2, Timer(0), EVENT_TYPE_HIGH);
```

**ClearEvent(event) Value - RCX2**

Clear the configuration for the specified event. This prevents it from triggering until it is re-configured.

```
ClearEvent(2); // clear event #2
```

**ClearAllEvents() Value - RCX2**

Clear the configurations for all events.

```
ClearAllEvents();
```

**EventState(event) Value - RCX2**

Return the state of a given event. States are 0: Low, 1: Normal, 2: High, 3: Undefined, 4: Start calibrating, 5: Calibrating in process.

```
x = EventState(2);
```

**CalibrateEvent(event, lower, upper, hyst) Function - RCX2**

Calibrate the event by taking an actual sensor reading and then applying the specified lower, upper, and hyst ratios to determine actual limits and hysteresis value. The specific formulas for calibration depend on sensor type and are explained in the LEGO SDK. Calibration is not instantaneous - EventState() can be checked to determine when the calibration is complete (typically about 50ms).

```
CalibrateEvent(2, 50, 50, 20);
until(EventState(2) != 5); // wait for calibration
```

**SetUpperLimit(event, limit) Function - RCX2**
Set the upper limit for the event, where event is a constant event number and limit can be any expression.

```plaintext
SetUpperLimit(2, x); // set upper limit for #2 to x
```

**UpperLimit(event) Value - RCX2**

Return the current upper limit for the specified event number.

```plaintext
x = UpperLimit(2); // get upper limit for event 2
```

**SetLowerLimit(event, limit) Function - RCX2**

Set the lower limit for the event, where event is a constant event number and limit can be any expression.

```plaintext
SetLowerLimit(2, x); // set lower limit for #2 to x
```

**LowerLimit(event) Value - RCX2**

Return the current lower limit for the specified event number.

```plaintext
x = LowerLimit(2); // get lower limit for event 2
```

**SetHysteresis(event, value) Function - RCX2**

Set the hysteresis for the event, where event is a constant event number and value can be any expression.

```plaintext
SetHysteresis(2, x);
```

**Hysteresis(event) Value - RCX2**

Return the current hysteresis for the specified event number.

```plaintext
x = Hysteresis(2);
```

**SetClickTime(event, value) Function - RCX2**

Set the click time for the event, where event is a constant event number and value can be any expression. The time is specified in increments of 10ms, so one second would be a value of 100.

```plaintext
SetClickTime(2, x);
```
ClickTime(event) **Value - RCX2**

Return the current click time for the specified event number.

\[ x = \text{ClickTime}(2); \]

SetClickCounter(event, value) **Function - RCX2**

Set the click counter for the event, where event is a constant event number and value can be any expression.

\[ \text{SetClickCounter}(2, x); \]

ClickCounter(event) **Value - RCX2**

Return the current click counter for the specified event number.

\[ x = \text{ClickCounter}(2); \]

### 3.9.2 Scout Events

**Scout**

The Scout provides 15 events, each of which has a predefined meaning as shown in the table below.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_1_PRESSED</td>
<td>sensor 1 pressed</td>
</tr>
<tr>
<td>EVENT_1_RELEASED</td>
<td>sensor 1 released</td>
</tr>
<tr>
<td>EVENT_2_PRESSED</td>
<td>sensor 2 pressed</td>
</tr>
<tr>
<td>EVENT_2_RELEASED</td>
<td>sensor 2 released</td>
</tr>
<tr>
<td>EVENT_LIGHT_HIGH</td>
<td>light sensor &quot;high&quot;</td>
</tr>
<tr>
<td>EVENT_LIGHT_NORMAL</td>
<td>light sensor &quot;normal&quot;</td>
</tr>
<tr>
<td>EVENT_LIGHT_LOW</td>
<td>light sensor &quot;low&quot;</td>
</tr>
<tr>
<td>EVENT_LIGHT_CLICK</td>
<td>low to high to low</td>
</tr>
<tr>
<td>EVENT_LIGHT_DOUBLECLICK</td>
<td>two clicks</td>
</tr>
</tbody>
</table>
The first four events are triggered by touch sensors connected to the two sensor ports. EVENT_LIGHT_HIGH, EVENT_LIGHT_NORMAL, and EVENT_LIGHT_LOW are triggered by the light sensor's value changing from one range to another. The ranges are defined by SetSensorUpperLimit, SetSensorLowerLimit, and SetSensorHysteresis which were described previously.

EVENT_LIGHT_CLICK and EVENT_LIGHT_DOUBLECLICK are also triggered by the light sensor. A click is a transition from low to high and back to low within a certain amount of time, called the *click time*.

Each counter has a counter limit. When the counter exceeds this limit, EVENT_COUNTER_0 or EVENT_COUNTER_1 is triggered. Timers also have a limit, and they generate EVENT_TIMER_0, EVENT_TIMER_1, and EVENT_TIMER_2.

EVENT_MESSAGE is triggered whenever a new IR message is received.

**SetSensorClickTime(value)**  
Function - Scout

Set the click time used to generate events from the light sensor. Value should be specified in increments of 10ms, and may be any expression.

```
SetSensorClickTime(x);
```

**SetCounterLimit(n, value)**  
Function - Scout

Set the limit for counter n. N must be 0 or 1, and value may be any expression.

```
SetCounterLimit(0, 100); // set counter 0 limit to 100
```

**SetTimerLimit(n, value)**  
Function - Scout
Set the limit for timer n. N must be 0, 1, or 2, and value may be any expression.

```
SetTimerLimit(1, 100); // set timer 1 limit to 100
```

### 3.10 Data Logging

The RCX contains a datalog which can be used to store readings from sensors, timers, variables, and the system watch. Before adding data, the datalog first needs to be created using the `CreateDatalog(size)` command. The 'size' parameter must be a constant and determines how many data points the datalog can hold.

```
CreateDatalog(100); // datalog for 100 points
```

Values can then be added to the datalog using `AddToDatalog(value)`. When the datalog is uploaded to a computer it will show both the value itself and the source of the value (timer, variable, etc). The datalog directly supports the following data sources: timers, sensor values, variables, and the system watch. Other data types (such as a constant or random number) may also be logged, but in this case NQC will first move the value into a variable and then log the variable. The values will still be captured faithfully in the datalog, but the sources of the data may be a bit misleading.

```
AddToDatalog(Timer(0)); // add timer 0 to datalog
AddToDatalog(x); // add variable 'x'
AddToDatalog(7); // add 7 - will look like a variable
```

The RCX itself cannot read values back out of the datalog. The datalog must be uploaded to a host computer. The specifics of uploading the datalog depend on the NQC environment being used. For example, in the command line version of NQC, the following commands will upload and print the datalog:

```
nqc -datalog
nqc -datalog_full
```

**CreateDatalog(size)**

Create a datalog of the specified `size` (which must be a constant). A size of 0 clears the existing datalog without creating a new one.

```
CreateDatalog(100); // datalog for 100 points
```
AddToDatalog(value) Function - RCX

Add the value, which may be an expression, to the datalog. If the datalog is full the call has no effect.

AddToDatalog(x);

UploadDatalog(start, count) Function - RCX

Initiate and upload of count data points beginning at start. This is of relatively little use since the host computer usually initiates the upload.

UploadDatalog(0, 100); // upload entire 100 point log

3.11 General Features

Wait(time) Function - All

Make a task sleep for specified amount of time (in 100ths of a second). The time argument may be an expression or a constant:

Wait(100); // wait 1 second
Wait(Random(100)); // wait random time up to 1 second

StopAllTasks() Function - All

Stop all currently running tasks. This will halt the program completely, so any code following this command will be ignored.

StopAllTasks(); // stop the program

Random(n) Value - All

Return a random number between 0 and n. N must be a constant.

x = Random(10);

SetRandomSeed(n) Function - RCX2

Seed the random number generator with n. N may be an expression.

SetRandomSeed(x); // seed with value of x
SetSleepTime(minutes) Function - RCX, CyberMaster

Set the sleep timeout the requested number of minutes (which much be a constant). Specifying 0 minutes disables the sleep feature.

```
SetSleepTime(5);  // sleep after 5 minutes
SetSleepTime(0);  // disable sleep time
```

SleepNow() Function - RCX, CyberMaster

Force the device to go to sleep. Only works if the sleep time is non-zero.

```
SleepNow();  // go to sleep
```

3.12 RCX Specific Features

Program() Value - RCX

Number of the currently selected program.

```
x = Program();
```

SelectProgram(n) Function - RCX2

Select the specified program and start running it. Note that programs are numbered 0-4 (not 1-5 as displayed on the LCD).

```
SelectProgram(3);
```

BatteryLevel() Value - RCX2

Return the battery level in millivolts.

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

FirmwareVersion() Value - RCX2

Return the firmware version as an integer. For example, version 3.2.6 is 326.

```
x = FirmwareVersion();
```

Watch() Value - RCX
Return the value of the system clock in minutes.

\[ x = \text{Watch}(); \]

**SetWatch(hours, minutes)**  
Function - RCX

Set the system watch to the specified number of hours and minutes. Hours must be a constant between 0 and 23 inclusive. Minutes must be a constant between 0 and 59 inclusive.

\[ \text{SetWatch}(3, 15); \quad // \text{set watch to 3:15} \]

### 3.13 Scout Specific Features

**SetScoutRules(motion, touch, light, time, fx)**  
Function - Scout

Set the various rules used by the scout in stand-alone mode.

**ScoutRules(n)**  
Value - Scout

Return current setting for one of the rules. N should be a constant between 0 and 4.

\[ x = \text{ScoutRules}(1); \quad // \text{get setting for rule #1} \]

**SetScoutMode(mode)**  
Function - Scout

Put the scout into stand-alone (0) or power (1) mode. As a programming call it really only makes sense to put into stand-alone mode since it would already be in power mode to run an NQC program.

**SetEventFeedback(events)**  
Function - Scout

Set which events should be accompanied by audio feedback.

\[ \text{SetEventFeedback}(\text{EVENT}_1\_\text{PRESSED}); \]

**EventFeedback()**  
Value - Scout

Return the set of events that have audio feedback.

\[ x = \text{EventFeedback}(); \]
SetLight(mode) Function - Scout

Control the Scout's LED. Mode must be LIGHT_ON or LIGHT_OFF.

    SetLight(LIGHT_ON); // turn on LED

### 3.14 CyberMaster Specific Features

*This section is woefully incomplete. The basic problems is that I don't have a CyberMaster, so I've never tried to understand these calls. If anyone has a good description of the functions, please let me know.*

CyberMaster provides alternate names for the sensors: SENSOR_L, SENSOR_M, and SENSOR_R. It also provides alternate names for the outputs: OUT_L, OUT_R, OUT_X.

**Drive(motor0, motor1)** Function - CyberMaster

**OnWait(motors, n time)** Function - CyberMaster

**OnWaitDifferent(motors, n0, n1, n2, time)** Function - CyberMaster

**ClearTachoCounter(motors)** Function - CyberMaster

**TachoCount(n)** Value - CyberMaster

**TachoSpeed(n)** Value - CyberMaster

**ExternalMotorRunning()** Value - CyberMaster

**AGC()** Value - CyberMaster

Return the current value of the automatic gain control on the RF receiver.

    x = AGC();