Not Quite C Compiler

version 1.0 b1

Introduction

NQC is a simple language for programming the LEGO RCX. The preprocessor and control structures of NQC are very similar to C. NQC is not a general purpose language - there are many restrictions that originate from actual limitations and lack of detailed information about the RCX itself.

A NQC program consists of tasks and subroutines. The first task is started automatically when the program is run, the remaining tasks can be started by the program. Subroutines can be shared across all of the tasks.

The compiler itself understands very little of the RCX. Most of the RCX functionality (such as playing a sound, reading an input sensor, etc) is added by the rcx.nqh include. This file is included automatically by the compiler (unless -s is specified on the command line). The file itself defines many high level macros (such as PlaySound()) that are implemented in terms of low level primitives within the compiler itself. This approach means that new "system" features can be easily added to rcx.nqh without modifying the compiler or introducing new keywords.

Usage

The nqcc command line useage looks like this:

```
nqcc [-s] [-d] [-l] [-e] [-o output_file] [-Dsym=[def]]
     [-Usym] source_file
```

The default behavior is to compile the specified source_file to an output file that is named by replacing the ".nqc" extension of the source filename with ".rcx". If the source filename does not end with ".nqc", the default output filename is the source filename with ".rcx" appended to it. The "-o" option may be used to specify an alternate name for the output file.

The source file name "-" causes nqcc to read stdin as the source file. In this case, the -o option must be used to explicitly name an output file if one is desired.

The "-s" option prevents the compiler for automatically including "rcx.nqh" before compiling the source file.

The "-d" option causes nqcc to atttempt downloading the program to an RCX after compiling it. If an output file was not explicitly named ("-o" option), then no output file is generated.

The "-l" option writes a bytecode listing to stdout.

The "-e" options sends compiler error messages to stdout instead of stderr. This option is only needed when running nqcc under Win95, which does not support redirection of stderr from the standard shell (command.com). The following examples show how to redirect stderr (to a file called errors) under other platforms:

```
MPW:    nqcc test.nqc •errors
WinNT:  nqcc test.nqc 2>errors
```

The -D*sym*[=*def*] option defines preprocessor symbol *sym* to the value *def*.

The -U option undefines preprocessor symbol *def*.

Two environment variables are used by nqcc:

NQCC_INCLUDE - this environment variable can be used to specifiy a directory to search for include files in addition to the current working directory. When building filenames, nqcc simply concatentates the value of NQCC_INCLUDE with the target filename, so NQCC_INCLUDE must end in the appropriate directory separator (’\’ for PC, ’:’ for Mac, ’/’ for Unix).

RCX_PORT - this environment variable can be used to specify a serial port other than the default (COM1 for PC, Modem for Mac). The value of RCX_PORT should be the name of the port (for example "COM2" for the PC, or "b" for the Mac’s printer port).

Preprocessor

The preprocessor implements the following directives: #include, #define, #ifdef, #ifndef, #if, #elif, #else, #endif, #undef.

The #include command works as expected, with the caveat that the filename must be enclosed in double quotes:

```
#include "foo.nqh"
```

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 to allow multi-line macros:

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

Conditional compilation works similar to the C preprocessor. Expressions in #if directives use the same operators and precedence as in C.

C and C++ style comments are supported

```
/* this is
   a comment */
```

```
// another comment
```

Tasks and Subroutines

Each program is made up of tasks and subroutines, each of which is composed of statements. Each program must have at least one task (named "main"), which serves as the entry point of the program.

A task looks like this:

```
task task_name
{
   statements
}
```

Tasks can be started or stopped with the following statements

```
start task_name;
stop task_name;
```

Subroutines look very similar to tasks, but use the sub keyword. Note that there are no parameters or return values for subroutines. The only variables available are global variables.

```
sub sub_name
{
    statements
}
```

Subroutines are called just like in C, with the caveat that there are not parameters or return value:

```
sub_name();
```

Due to a limitation in the RCX firmware, subroutines are not allowed to make subroutine calls (either to themselves or another subroutine).

Statements are contained within tasks and subroutines. Like C, the semicolon (';') is used as a statement terminator. Groups of statements can be combined into a a block using braces ('{' and '}').

Values

Many statements (such as variable assignment, or conditional testing) make use of "values". A value can be many things: a constant, the contents of a global variable, a random number, the reading from an input sensor, etc. Each of these is covered in more

detail below. In addition, some of the high level commands may take constants and/or values as arguments. In the description of a command its arguments will be noted as either "const" or "value". "const" arguments must be a numeric constant. "value" arguments may be either a numeric constant or another sort of value (such as a variable). Note that not all value types are legal for all commands that use values.

Numeric constants can be written as either decimal (e.g. 123) or hexidecimal (e.g. 0xABC). Presently, there is very little range checking on constants, so using a value larger than expected can have unpredictable results. Constants can be combined using the standard arithmetic operators (+, -, *, /, %, <<, >>, &, |, ^, ~). Precedence and associativity are the same as C. Parentheses can be used to group lower precedence operations together.

The RCX provides 32 global variables. In order to use variables in your program, you must first declare them:

```
int foo;
```

All of the variables are global (even across multiple programs), hence they must be decalred outside the scope of any tasks and subroutines. Variables can be assigned values using the following assigment operators:

```
var = value;
var += value;
var -= value;
var *= value;
var /= value;
```

Note that general expressions (e.g. a = b + c) are only supported for compile time constants and not for runtime evaluation:

```
int foo,bar;

task main
{
    foo = 2 + 3;  // legal
    foo = bar + 2; // illegal
}
```

Other value types inclue:

`Input(n)`	- current value of sensor 'n', where n=0-2
`Timer(n)`	- current value of timer 'n', where n=0-2
`Random(n)`	- random number between 0 and n-1? (not sure about the range)
`Message()`	- value of the last received IR message
`Watch()`	- value of the system clock

Note: the rxc.nqh file defines macros IN_1, IN_2, and IN_3 as Input(0), Input(1), and Input(2), respectively.

Control Structures

Conditions can be either a comparison beteen two values, or one of the predefined constants "true" and "false". There are six relations (<, >, <=, >=, ==, !=). Conditions may be combined using && and ||, or negated with '!".

Here are the control structures:

```
if (condition) statement
if (condition) statement else statement
while(condition) statement
do statement while(condition)
repeat(value) statement
wait (condition) statement
```

The first four structures are identical to their counterparts in C.

The "repeat" structure causes a statement to be executed a number of times. This number does not have to be a constant, for example it could be the value of a global variable, however the number is only evaluated prior to starting the loop (as opposed to the "for" structure in C which evaluates the condition before each iteration). The RCX allows "repeat" loops to be nested (maintaining its own "stack" of look counters). The "break" statement is also legal within repeat loops, although it has a rather unexpected result when used within a nested loop. Although the control flow will be broken, there is no way to pop the old counter off the internal stack, so the enclosing loop will be using the wrong counter when it starts repeating. This cannot be reliably checked at compile time since the internal repeat stack is shared across subroutine calls.

The "wait" structure executes the body until the condition becomes true. It is actually implemented as a macro (so be careful of side effects). It it handy when waiting for something to happen:

```
wait(IN_3 == 1); // wait for sensor 3 to have value 1
```

Several other control flow statements are supported:

```
break;
continue;
return;
```

"break" and "continue" can be used within "do" and "while" loops.
"return" can be used to exit a subroutine before getting to its end.

Sensors

The constants IN_1, IN_2, and IN_3 can be used to either read an input sensor, or to specify a sensor. Sensor types are defined by IN_SWITCH, IN_LIGHT, IN_ANGLE, IN_PULSE, and IN_EDGE. The Sensor() function is used to initialize the sensor:

```
Sensor(value sensor, const type);
```

Some sensor types (such as the IN_ANGLE type) allow you to "reset" the sensors internal counter with the following command:

```
ClearSensor(value sensor);
```

The only supported value types for "sensor" in the above two examples are numeric constants and Input(n) - all other types have undefined behavior.

Here's a brief example that configures sensor 1 as a switch, then waits for it to be pressed before playing a sound.

```
Sensor(IN_1, IN_SWITCH);
wait (IN_1 == 1);
PlaySound(0);
```

The macros IN_1, IN_2, and IN_3 are equivalent to Input(0), Input(1), and Input(2) and can be used anywhere that a runtime value is legal (such as setting a variable, checking a condition, etc).

Motors

The constants OUT_A, OUT_B, and OUT_C refer to the three outputs. They can be combined (using + or |) in order to specify multiple outputs. OUT_FULL is a constants that represents full speed for a motor. Here's how to turn on all three motors for 1 second:

```
Fwd(OUT_A + OUT_B + OUT_C, OUT_FULL);
Sleep(100);
Off(OUT_A + OUT_B + OUT_C);
```

These commands control motors:

```
Fwd(const outs, value speed) // turn motor(s) on forwards
Rev(const outs, value speed) // turn motor(s) on backwards
Off(const outs) // stop motor(s)
```

Other Commands

```
PlaySound(const n); // play system sound #n
PlayNote(const freq, const duration); // play a note

Sleep(value v) // sleep for v ticks (100 ticks per second)

Display(value v) // set the dispaly mode as follows:
```

Note: display modes for "Display" are as follows: 0 = system clock, 1-3 = input channel, 4-6 = output channel. To show the value of the third input, use "Display(3)", not "Display(IN_3)". The later command will read the value of sensor 3, and use that value as the argument to the display command (e.g. if the sensor's value is 0, then the system clock will be displayed). Yeah, this is pretty stupid, but its how the RCX works.

```
SendMessage(value m);   // send low 8 bits of m as an IR message
ClearMessage();   // clear the last received IR message

ClearTimer(const n);    // clear timer #n
```

Data Logging

The RCX supports a Datalog feature. Using this feature requires three steps:

1) Set the datalog size
2) Add data points to the datalog
3) Upload the datalog to a host

You can set the datalog size with the following command

```
SetDatalog(const n); // create datalog for n points
```

```
Point can be added with

Datalog(value v); // add v to the datalog
```

Only certain types of values may be added (variables, timers, sensor readings, the system clock).

Some (or all) of the datalog may be uploaded at any time as follows:

```
UploadDatalog(const index, count count);
```

"index" is the first data point to upload, and "count" is the number of points to upload.

Each entry in the datalog uses three bytes - the first byte is the type of entry, the next two are a 16 bit value in little endian format. If you set the size of the datalog to N, then there are actually N+1 entries - the first one (at index 0) is the number of data points collected so far (type = 0xff), while the remaining ones (index 1 - N) are the points themselves. Here are the type codes for normal data points:

0x00 - value of a variable
0x20 - value of a timer
0x40 - value of a sensor
0x80 - value of the clock

Keywords

The following keywords are used in nqcc:

```
asm         if          stop
break       int         sub
continue    repeat      task
do          return      true
else        start       while
false
```

Grammar

program : unit_list
;

unit_list: unit_list unit
|
;

unit : **int** id_list **;**
| **task id** block
| **sub id** block
;

id_list : id_list **, id**
| **id**
;

block : **{** stmt_list **}**
;

stmt_list : stmt_list stmt
|
;

stmt : **;**
| block
| **asm {** bytes **} ;**
| **while (** condition **)** stmt
| **do** stmt **while (** condition **) ;**
| **repeat (** value **)** stmt
| **break ;**
| **continue ;**
| **if (** condition **)** stmt
| **if (** condition**)** stmt **else** stmt
| **id** assign value **;**
| **start id ;**

```
        | stop id ;
        | id ( ) ;
        | return ;

bytes   : bytes , expr
        | expr
        ;

value   : expr
        | id
        | @ expr
        ;

assign  : = | += | -= | *= | /=
        ;

expr    : number
        | expr binop expr
        | - expr
        | ~ expr
        | ( expr )
        | [ value ]
        ;

binop   : + | - | * | / | % | & | | | ^ | << | >>
        ;

condition : value relop value
        | ! condition
        | condition && condition
        | condition || condition
        | ( condition )
        | true
        | false
        ;

relop   : == | != | < | > | <= | >=
        ;
```

For Hackers Only

NQC contains very little in the way of RCX specific functions. Support for controlling motors, reading sensors, etc. is provided in the rcx.nqh file. This section explains some of the low level constructs used within these definitions.

Internally, many of the RCX operations use "values", which are essentially a form of typed data. Each value has a "type" and a "index", which together specify a sort of effective address for the runtime value. Of course, one of the types is "constant", in which case the effective value is equal to the index itself. Other types include reading an input sensor or generating a random number.

The compiler can perform arithmetic operations only on compile time constants. Certain statement (such as conditional clauses) require values, and constants are automatically promoted to values in these situations.

However, sometimes it is desireable to specify the type/index pair directly. This can be done by using the @ operator, which converts a 24 bit number directly to a value. The lower 16 bits are the index, the next 8 are the type code. For example, since the type code for "constants" is 2, the following value is equal to 0x123:

```
@0x20123
```

The type code for variables is 0, so the following refers to variable 7:

```
@0x00007
```

This is very different from 0x00007, which equals the number 7. It would've been much nicer if LEGO had made the type code for constants equal to 0, but we have to live with this hack.

Sometimes it is necessary to convert a value back into its "effective address". This is done by enclosing the value in brackets:

```
[ Random(7) ]
```

The effective address is constant, thus can be manipulated with the compiler's arithmetic operators.

Finally, the "asm" statement allows most primitives to be implemented:

```
asm { data };
```

"data" is one or more numbers, separated by commas. The asm statement emits the literal data contained within its braces. The data bytes for an asm statement must be compile time contsatnts, not runtime values.