

Firmware Features:

The following is a description of the addition capabilities of the enhanced firmware over the standard Lego supplied firmware

- Up to 100 times faster than standard Lego™ supplied firmware.
- Backwards compatible with Lego™ standard firmware
- One millisecond resolution on timers.
- “Perfect” rotation counter support. No spurious counts. Support for up to 1,300 ticks per second vs. standard firmware operation limitation of 220.
- 256 total global variables (16 bit)
- 32-bit integer variables.
- Floating-point variables.
- Support for powerful debugging functions. Includes breakpoints, single and multi statement and instruction step mode, and ASSERT opcode.
- Enhanced infrared messaging support during normal byte code operation
 - 2400 or 4800 baud operation
 - Programmable message preamble size
 - Programmable “complement” byte disable
 - Four times faster messaging than standard Lego supplied firmware.

End user program control of all options.

- Fine motor control: 128 power levels, optional brake (vs. float) during inactive PCM, etc.
- Memory stacks for each task to hold return addresses and local procedure variables.
- Nested subroutine calls. Return addresses stored on task stack.
- Native support for function / procedure return values.
- Expanded sound features. Volume control, count of queued sounds, etc.
- New variables for tuning operating system performance: time slice size, sensor polling interval, LCD refresh interval, optional no power off when AC powered, immediate battery level,
- Exception support. For over 25 different exceptions including capabilities like: array bounds out of range, invalid opcode, and range check on intrinsic variable parameters, stack overflow, etc.
- Messages expanded from 8-bits to 16-bits. Support for both 8-bit legacy messages and two new opcodes to send 16-bit messages. Useful for
- Expanded number of subroutines to 40. Support for nested subroutine calls.
- New real-time optimized opcodes

Details on Firmware Features:

100 times faster than standard Lego™ supplied firmware

Replacement firmware is up to 100 times faster than the Lego supplied firmware. Lego firmware typically takes 3.2 milliseconds per opcode. In enhanced firmware (100X version), a no-op (opcode 0x10) opcode takes less than 10 microseconds to execute! New firmware interprets 16-bit add or subtract opcodes in under 30 microseconds. 16-bit conditional branch opcodes take 50 –

80 microseconds. Real-life measurements taken from live firmware. Time for opcode execution includes all operating system overhead (scheduler, sensor handling, LCD updates, interrupts, etc).

Typical performance appears to be 20,000 – 35,000 interpreter opcodes executed per second; these are actual measurements from a variety of sample programs. Standard Lego firmware does just over 300.

The performance improvements come from:

- Standard Lego firmware appears to use multiple levels of data abstraction to “hide” data between different modules. Overkill for this small application and lots of real time loss through procedure calls to get at data that could be stored in an “extern” variable.
- Selective coding of a few high-runner routines in hand-optimized assembler to get the best performance. The critical routine is the one that converts interpreter variables into actual values.
- Extremely efficient task scheduler compared to standard firmware. Standard firmware interpreter executes one opcode and then relinquishes control to other OS tasks. When opcode interpreter is given next time slice, it moves to next task and tries to execute one opcode; it ‘burns’ a time slice to every task even if that task is inactive. I’m guessing over 2/3 of the time spent executing an opcode is in the scheduling overhead. Enhanced firmware (10X) can be configured to interpret several opcodes in a single task timeslice; it does not ‘waste’ timeslices on interpreter tasks not able to run.
- Use of the Hitachi / Renesas tool chain for development. I found these tools to be far more efficient than GCC and others in both the size of generated code and its real time performance.

32-bit and floating opcodes are much slower. Floating point opcodes typically take 200+ microseconds each. This is because the H8 CPU is only an 8/16 bit CPU; arithmetic operations on ‘floats’ and ‘longs’ are done by software subroutines.

Backwards compatible with Lego™ standard firmware

The objective was to provide backwards compatibility with standard Lego firmware. This has been substantially achieved. See separate section providing more details.

256 total global variables (16 bit)

Standard Lego firmware supports 32 global variables and 16 variables per task. Replacement firmware expands support to 256 global variables.

Task variables are overlaid on the global variables 32 to 191.

New opcodes and intrinsic variables have been added to the firmware to support the expanded variable address space.

32-bit integer variables

New opcodes and intrinsic variables have been added to firmware to support 32-bit long integers. Each long integer occupies the space of two consecutive 16-bit integers.

Floating point variables

New opcodes and intrinsic variables have been added to firmware to support 32-bit float variables. Each long integer occupies the space of two consecutive 16-bit integers.

One millisecond resolution on timers

New firmware implements a single high-resolution (32-bit) clock variable with a one-millisecond period. Four Lego firmware compatible timers are then implemented using this clock variable.

New opcodes and intrinsic variables are implemented to provide full access to these timers. This includes variables that provide 1, 10 and 100-millisecond resolution; and new wait1Msec opcode to perform program waits using one millisecond resolution.

“Perfect” rotation counter support

The Lego rotation counters are wonderful sensors. Internally they have a fan blade interrupting two optical transmitter/receiver pairs. This provides 16 counts per revolution; there are four states for the optical T/R as each of the four fan blades rotates through them. However the sensor frequently provide spurious readings as each fan blade transitions under a optical T/R. Depending on rotation speed, error rates range from 0.05% (very fast speed) to over 60% (very slow).

New firmware provides a powerful software state machine to (hopefully) provide perfect rotation counter support. The state machine filters and eliminates transient readings.

Standard Lego firmware scans rotation sensor every three milliseconds leading to a theoretical maximum speed of 333 counts per second. However, the asymmetrical nature of the fan blade – the gap is different size from the blade – reduces the maximum speed to about 220 counts. RobotC supports maximum speed of ~1,300 counts per second; six times the capacity of the standard Lego firmware.

New firmware allows customization of the sensor scan rate through a new intrinsic variable. A one-byte value holds the number of milliseconds in the scan interval. A value of zero indicates a 0.5 millisecond scan rate. Of course, as the scan rate is reduced, less CPU time is available for user programs; at 0.5 millisecond rate, the byte code interpreter is about twice as slow as the ‘standard’ 3 millisecond rate.

Firmware resets the scan rate to 3 milliseconds on RCX power up.

Powerful debugging functions

New firmware has been designed for tight integration with a new IDE providing powerful debugging capabilities from the PC. New opcodes and capabilities have been implemented for debugging support including:

- The byte code interpreter can be suspended or resumed under PC IDE control.
- Opcodes are provided to allow the IDE to single step the byte code interpreter. Single step is provided for either a single instruction or single ‘C’ statement level (i.e. step for ‘n’ opcodes or until a instruction branch occurs).
- An ‘ASSERT’ opcode is provided to check for user programmed exceptions.
- Opcodes that provide very efficient upload of data to an IDE’s debugging windows (variable watchers, call stack display, task status, etc). Current firmware provides upload of some of these variables, but it is one variable at a time leading to slow updates and lots of IR messaging. New opcodes are provided to upload a block of variables in one message.
- Exception handling support in the interpreter (e.g. for stack underflow/overflow). Interpreter halts on exception with audible sound and LCD display. Optionally, an unsolicited message can be sent to the PC detailing the exception.

Current programming environments do not support these new capabilities. A new IDE is required and one has been written that is currently in Alpha testing.

Enhanced infrared messaging support

New firmware allows infrared messaging to be optionally configured to operate over four times faster than the standard Lego supplied firmware. This has value in two applications: [1] faster IR communication in multi-RCX applications and [2] faster communication between RCX and PC development environments.

- The standard Lego firmware uses a fixed 2,400-baud infrared messaging. Every message data byte is also followed with a complement byte. Standard firmware allows modification of transmit parameters for a one-time user program generated message. In addition to this functionality, new firmware allows complete customization of the IR messaging parameters (2,400 or 4,800 baud; enable/disable complement byte) for both transmit and receive directions for normal operation.
- There are many fast firmware downloading programs that operate at quad speed (4,800 baud and no complement bytes). However, end user byte code programs are always downloaded at the slower standard rate. The expanded infrared messaging in the new firmware enables programming environments to provide quad speed downloading of end user programs.
- Standard Lego firmware prefixes every message with a three byte (0x55, 0xFF, 0x00) preamble. The preamble has minimal benefit; if a corrupted value of one of the preamble bytes is received, then the complete message will be incorrectly received. The new firmware allows customization of the preamble size from zero to three bytes. RCX message reception is flexible and will always accommodate any size preamble.
- A quirk in the standard Lego firmware appears to require a 30-millisecond delay between transmission of last byte of a reply message from the RCX and reception of the first byte of a new message. The new firmware does not emulate this quirk!

On RCX power up, the messaging is always reset to the “standard” 2,400-baud values.

The new IDE currently in Alpha testing takes full advantage of the enhanced IR messaging. Features include:

- RCX is configured to use quad speed communication with the PC enabling faster download of user programs.
- An incremental downloading function where only the tasks and subroutines that have changed since the last download are retransmitted to the RCX.
- Quad speed and bulk variable upload are used to provide rapid update of PC debugging windows.

Note: The Lego remote control only operates at the slow 2,400-baud speed. It is incompatible with the optional ability to configure RCX to operate at faster IR rates.

Fine motor control

Three new enhancements to the motor control.

- Up to 128 power levels.
- Programmable delay period when switching direction. Motor is 'braked' during the delay. Standard firmware inserts a fixed 100-millisecond delay.
- Option to replace un-powered PWM cycles with ‘brake’ instead of ‘float’ for finer control over motor power and speed.

Standard Lego firmware provides eight levels of motor power levels. New firmware provides support for 128 levels of control. New opcodes and intrinsic variables are provided to provide full access to this expanded range. User programs can use either format at the power level is internally stored in the 128 level format and conversion to 8 level is provided when accessing this format.

Standard Lego firmware provides a 100-millisecond delay when changing powered motor direction to “prevent wear and tear on the motor gears”. The implementation appears to have a few software errors, as it doesn’t appear to always work. New firmware provides a clean implementation. The actual delay period is customizable from 0 to 255 milliseconds; a new intrinsic variable provides control over the interval.

There is a complicated relationship between power level and motor speed related to the actual load on the motor. It is not a linear relationship! Normally, the PWM (Pulse Wave Modulation) algorithm used to control motor provides periods of ‘power’ and ‘float’ to the motor; at higher power levels, there are more frequent ‘power’ intervals. A technique borrowed from Steve

Hassenplug's Legway is to replace the 'float' periods with 'brake'. This provides a more linear relationship power level and speed; however, the motors become very noisy (grinding gears?). New firmware provides a new intrinsic variable for use of 'brake' for users who want to explore this functionality and perhaps wear out their motors faster.

Memory stacks for each task

New firmware implements a stack for each task. The stack is used to store subroutine return addresses and (future) local procedure variables and parameters. This functionality enables nested subroutine calls.

A new intrinsic variable allows customization of the size of task stack. The task stack is allocated when a task is first run. The stack remains allocated until the task is deleted.

It would be a relatively easy extension in the future to provide storage for procedure variables on the task. This would eliminate restrictions on running out of variables.

Nested subroutine calls

Standard Lego firmware supports only one level of subroutine call. New firmware supports nested subroutine calls. The number of nested subroutines is limited by the size of the allocated task stack; four bytes are used for each call. Default task size is four bytes.

Native support for function return values

New firmware includes a new intrinsic variable to store function return values. Usage of this feature will require changes to existing programming environments.

Expanded sound features

RobotC firmware provides expanded sound features.

- Standard firmware provides no way to determine how many sound items are currently queued. If the queue is full, new sound items are simply discarded. The enhanced firmware provides a new intrinsic variable to interrogate the size of the queue. It's now easy to implement a wait loop until there is space in queue before a call to *PlayTone* or *PlaySound*.
- Standard firmware plays tones with a square wave using a 50/50 on/off duty cycle. New firmware provides end user control of the duty cycle with 16 steps. This provides a primitive volume control. Unfortunately the mechanical characteristics of the speaker are such that the results are not linear and do not provide a consistent volume control.

Tunable Operating System Parameters

New firmware provides several new intrinsic variables to tune the performance of the operating system. These include parameters like:

- Time slice size. The time slice size defines the number of byte code opcodes that are executed before switching to a different task.
- Sensor refresh rate can be specified in one-millisecond increments with a value range of 1 to 255. A value of zero is a special case indicating a 0.5 millisecond scan rate. Sensor refresh in the standard firmware is fixed at a three-millisecond scan rate.
- LCD refresh interval can be specified in 100 millisecond increments. Refreshing the RCX's LCD takes about two milliseconds of CPU time. For real time critical user applications the refresh rate can be set to zero which will disable refresh when any task is running

- No power down when powered by AC adaptor. The RCX 1.0 includes a AC adaptor jack. When this Boolean variable is set, firmware will not power off the RCX if the voltage level is above 9.8 volts.
- Several parameters to control the default IR messaging.
- Programmable transition delay period when switching powered motor direction.
- Size of task stack allowing nested subroutine calls.
- Control over sound “volume”.
- A few new formats of the LCD watch display. Current firmware format is HH:MM. New formats include MM:SS and M:SS.S
- Use ‘brake’ instead of ‘float’ during non-powered PWM pulses.
- Expand from 8 to 40 subroutines.
- Interrogate the immediate unfiltered battery level. Current firmware provides a battery level that is an average of 32 recent voltage samples.

Exception Support

Enhanced firmware provides support for over 25 different exceptions. When an exception occurs, the byte code interpreter is suspended and a special display is posted to the RCX LCD.

Optionally the RCX can be configured to generate an autonomous exception report message that can be received by a PC-based IDE. When the new IDE is in debug mode it is continuously scanning for these exception messages and will pop up a window display the exception details.

Some of the different exceptions include:

- Range check on array bounds.
- Task stack overflow / underflow.
- User program generated ASSERT failure.
- Invalid opcodes.
- Range checking on intrinsic variable parameters.
- Invalid program counter value (on branch or subroutine return)
- Attempting to write a read only intrinsic variable
- ‘long’ or ‘float’ not supported in current firmware version.

Messages expanded to 16-bits

Standard firmware only supports 8-bit message. The enhanced firmware has opcodes to support both 8-bit and 16-bit messages. 16-bit messages are useful in multi-RCX applications where 8-bit messages have inadequate bandwidth.

Subroutine limit expanded to 40

Standard Lego firmware supports eight subroutines for each of the five program slots. The lack of nested subroutines, return values and procedure parameters make it cumbersome to use more than eight slots with standard firmware. The RobotC overcomes these limitations facilitating extensive use of more subroutines as good programming practice.

RobotC firmware operates in two modes: [1] compatibility with standard firmware and [2] making all 40 subroutines available to a single program slot.

New Real Time Optimized Opcodes

Enhanced firmware implements existing opcodes for backwards compatibility. Over 100 new opcodes have been implemented to provide enhanced functionality and improved performance. These opcodes provide the functionality described in the preceding text (256 variables, ‘long’ variables, ‘float’ variables, debugging support, expanded messaging, etc).

The best real time performance (100X and above) comes from compilers that support the use of the new real time optimized opcodes; using these opcodes probably contributes an additional 30% performance factor.

NQC Compatibility

Seamless Support for Current Programs

The enhanced firmware is believed to be compatible with existing NQC compiler. Existing NQC programs should run on the new firmware without change. They should take full advantage of the increased real time performance.

The new header file *NewFirmware.nqh* provides additional #defines to enable the incremental functionality of the enhanced firmware. The header file contains new intrinsic variables and opcode function definitions.. The file is well commented and a quick read will give a basic understanding of this functionality and how to use it.

There are a few issues that arise when trying to use these new variables in a NQC program

- Read access to these variables works fine. However, hard-coded in the NQC compiler is a table of the intrinsic variables that support write access. This table does not include the new variables and the compiler will give "expression is not a legal target for assignment" error message.
- A work-around is provided in the header file. For example use the following “SetSourceValue(motorPower128(1), 93);” instead of “motorPower128(1) = 93;” to set the power level for motor B.
- NQC has an internal character string table containing names for intrinsic variable types. It is used in generating symbolic names in the assembler listing. The output will be confused in listings for new intrinsic variables. It may contain a name that was valid in one of the other products (e.g. Spybotics) or it may show up as “?”. However, the generated code appears accurate.

Accessing Enhanced Features

Seamless support by NQC (or other development systems and compiler) to the enhanced features will require.

- Enabling new intrinsic variable in both r/w and not just r/o mode. Add new entries to internal tables for the new intrinsic variables.
- Support for the expanded number of global variables. Current firmware only supports 32 global variables and enhanced firmware supports 256. The effort required for this support can be estimated by reading the following section describing the implementation (new variable types and a handful of new opcodes) of this expansion.
- Prevent error messages with the use of nested subroutines. NQC compiler generates an error message when this is attempted.
- The enhanced firmware supports long and floats variables. Internally they are stored as two consecutive 16-bit variables. It’s likely a big effort to provide full support for these requiring additions of type checking to compiler, automatic type conversion and use of a different group of opcodes for these new types.

Expanding The Number of Variables

One benefit of the enhanced firmware is more variables to a total of 256 global variables. The existing firmware has 32 global variables and 16 local variables for each of the ten tasks.


Task variables are currently overlaid with the global variables as shown in the accompanying table. Note: it would be an easy change to use unique address spaces if this has an advantage.

Memory variables in Lego’s firmware are accessed with a single address in the range 0-47. 0-31 is for global variables and 32-47 corresponds to task variables 0-15 for the currently running task. Local variables can only be accessed during execution of a task.

Interpreter variables are defined within the interpreter with a three byte “address”. The first byte is the type of variable (global/task variable = 0, sensor = 1, constant = 2 and so on); the remaining two bytes are the index/address within that type of variable. A new type (37) was defined to represent global variables.

Many opcodes use only one or two bytes to address a variable. One byte is used to implicitly reference a global/task variable; two bytes are used when the index/address byte is only a single byte. In particular, the arithmetic opcodes (+=, -=, *=, /=, ...) always assume destination is a global/task variable. There is no way within existing opcodes to access the expanded range of global variables.

Eight arithmetic opcodes were defined in the enhanced firmware to support the expanded global variable range. The destination parameter of these opcodes is a global variable (0-255) rather than the combined global/task var (0-47). These opcodes have identical format and parameters as existing arithmetic opcodes to simplify conversion of programming systems.

Text Box: GlobalVariables Overlaid

Contents	0 - 31	Global variables	0 - 31
	32 - 47	Task 0 variables	48 - 63
		Task 1 variables	64 - 79
		Task 2 variables	80 - 95
		Task 3 variables	96 - 111
		Task 4 variables	112 - 127
		Task 5 variables	128 - 143
		Task 6 variables	144 - 159
		Task 7 variables	160 - 175
		Task 8 variables	176 - 191
		Task 9 variables	192 - 255
		Global variables	192 - 255