

Enhanced Firmware

Description of New Variables and Opcodes

Jan. 03, 2005

Table of Contents

<u>Document Change History.....</u>	<u>4</u>
<u>Overview.....</u>	<u>5</u>
<u>Interpreter Variables.....</u>	<u>6</u>
Quick Start.....	6
Efficient Array Access Source.....	10
System Parameter Types.....	11
UART Setup Parameter Types.....	12
Sound Parameter Types.....	13
Task Priority Expansion.....	13
Expanding The Number of Variables.....	13
<u>Interpreter Opcode Overview.....</u>	<u>15</u>
Changes in Opcode Sizes.....	15
Changes in Direct Opcodes.....	15
Existing 16-Bit Integer Arithmetic Operations.....	16
<u>New Arithmetic Opcode Descriptions.....</u>	<u>17</u>
Generic 16-Bit Integer Arithmetic Operations.....	17
16-Bit Integer Arithmetic Operations, Memory Parameters Only.....	20
Optimized Opcodes for Global Variables.....	20
32-Bit Integer Arithmetic.....	23
Floating Point Arithmetic.....	25
Other Arithmetic Opcode Sets.....	26
Optimized Two Byte Result Variables.....	26
32-bit Arithmetic with Long Constant Opcodes.....	27
Generic Absolute Value Opcode.....	27
<u>Conditional Branch Opcodes.....</u>	<u>28</u>
<u>“Switch” Opcodes.....</u>	<u>31</u>
rcxSwitch and rcxSwitchByteCase	31
rcxSwitchIndexTableNear and rcxSwitchIndexTableFar	32
<u>Unconditional Branch Opcodes.....</u>	<u>33</u>
<u>Expanded User Infrared Messaging Support.....</u>	<u>34</u>
Standard Firmware	34
Enhanced Firmware	34
<u>Miscellaneous New Opcodes.....</u>	<u>36</u>
Array Bounds Limit Checking.....	36
Useful “Extra” Arithmetic Functions.....	36
<u>Program Memory Map and Allocation.....</u>	<u>37</u>

<u>Standard Firmware</u>	<u>37</u>
<u>Enhanced Firmware</u>	<u>37</u>
<u>What's Not Documented Yet.....</u>	<u>39</u>

Document Change History

Date/Version	Contents
18 Nov 2004	Initial Release
28 Nov 2004	Added description of Program Memory Map. Added description of opcodes for implementing 'switch' structures in one opcode.
06 Dec 2004	Added description of 'rcxBranchNearByte' and 'rcxBranchFarWord'
10 Dec 2004	Added description of 'rcxAbsoluteValueSourceValue' opcode
11 Dec 2004	Added description of opcodes for inter-RCX messaging. Removed the new and undocumented 'rcxWaitMessage' opcode. Equivalent functionality can be provided with an 'event' definition. Added description of new opcode for run-time checking of array bounds. Added description of capability for assigning CPU scheduling priority. Added description of new source types that allow very efficient generation of array accessing code.
03 Jan 2005	Added description of new "generic parm" opcode set. Added description of rationale behind the multiple opcode sets. Corrected the description of the "bit complement" opcodes. They are unary and not binary opcodes. Added description of changes for "flipped bit 3" in direct opcodes. Added description of new opcodes for shift left/right, Xor, etc. Source types 'system' and 'sound' are incorrect against the current implementation. Documentation not fixed. Refer to header file.

Overview

The new enhanced firmware expands the byte code interpreter to support 256 global variables. It also includes support for long and floating-point numbers. It has also implemented a number of new intrinsic variables to gain access to new functionality. This document contains a description of these items.

The first chapter describes the new and expanded intrinsic variables. The format is similar to LEGO's LASM byte code description manual.

The next chapter describes the new arithmetic opcodes (add, minus, etc). Many of these are related to long and float operation.

The next chapter describes the new conditional branch opcodes.

There is a large set of new opcodes. However, only a handful needs to be implemented to provide the core functionality. The remainder provides real time speed up and code efficiency; for example, constants are frequent arithmetic operands so there is a group of opcodes especially for one byte opcodes.

Interpreter Variables

The byte code interpreter stores variables in three bytes. The first byte indicates the source of the variable and the next two bytes contain an index value. The sources are described in the table on the following page.

In many cases, the index variable fits in a single byte. Some opcodes within the interpreter take advantage of this and only use two bytes to represent the variable. A further code size optimization used by some opcodes is that they implicitly only work on a particular source type and don't include a byte for the source type.

Quick Start

The table on the next page can be somewhat overwhelming. For a quick start on using the expanded variable range, simply consider the following.

In the standard firmware, variables are represented within an opcode by two bytes. The first byte indicates that it is a variable (value is 0) instead of an intrinsic. The second byte contains the index or "address" of the variable. An address of 0 to 31 represents the 32 global variables; an address of 32 to 47 represents the 16 task variables. There are 10 tasks in the interpreter, each with 16 variables. The interpreter uses the current task number coupled with index (normalized to 0 – 15) to index its internal array of task variables. The two bytes representing a variable are thus:

0	0 to 47
---	---------

In the enhanced firmware, there are 256 global variables. These cannot be accessed with the existing variable type. So a new type (37) was defined specifically for global variables. A global variable is represented by the two bytes

37	0 to 255
----	----------

The opcodes in the interpreter internally uses one, two or three bytes to store variables. Three bytes is the generic case: one byte for the type and two bytes for the index/address. In all but a few cases, the upper byte of index/address field is zero; only one of the 30+ types (i.e. a 16-bit constant) can have a non-zero value. So some opcodes take a shortcut and only use two bytes for the variable type. Using global variables instead of the combined global/task variable type will work fine in the new interpreter.

A few opcodes implicitly know that an operand is a variable and only store a single index/address byte for an operand. This does present a problem if you want to refer to a global variable higher than 32. The solution was to define new opcodes that "match" the existing ones except they implicitly refer to global variables.

Source	Symbolic Name	Name	Access	Version	Index	Comments
0	rcxParmVar	Variable	R/W	2.0	0 - 47	Global variables <0 - 31>. Task variables <32 - 47>
1	rcxParm100MsecTimer	Timer	R/W	2.0	0 - 3	Timer in 100 millisecond units. Can only write value to zero.
2	rcxParmConstant	Constant	R/O	2.0	int	Constant
3	rcxParmMotorState	Motor Status	R/W	2.0 / New	0 - 2	motor state <0,1,2> <0x07 power 0x08 fwd 0x40 off 0x80 on 0x00 float>. R/O variable in Lego firmware. R/W in new firmware.
4	rcxParmRandom	Random	R/W	2.0 / New	int	random number. Return value is between zero and the value of the 'index'. Lego firmware uses a pseudo random number generator algorithm. New firmware calculates random number by periodically sampling the least significant 4-bits of the "free running counter" (operating at 2 MHz with values in range of 0..500) and add to (lastRandomNumber << 4). The value is writeable in the new firmware but has no effect.
5	rcxParmCyberTachoCounts	N/A		2.0		Cybermaster only
6	rcxParmCyberTachoSpeed					
7	rcxParmCyberTachoCurrent					
8	rcxParmProgramNumber	Program Slot	R/W	2.0	0 - 4	Current program number
9	rcxParmSensor	Sensor Value	R/W	2.0	0 - 2	Sensor value. A write operation sets the value to zero and is the equivalent of the "ClearSensor" opcode (0xD1). This only has meaning with rotation counter sensors.
10	rcxParmSensorType	Sensor Type	R/W	2.0	0 - 2	Sensor type
11	rcxParmSensorMode	Sensor Mode	R/W	2.0	0 - 2	Sensor mode
12	rcxParmSensorRaw	Sensor Raw	R/O	2.0	0 - 2	Sensor raw value
13	rcxParmSensorBoolean	Sensor Boolean	R/O	2.0	0 - 2	Sensor Boolean value
14	rcxParmClockMinutes	Clock	R/W	2.0		Minutes on clock. Minutes are reset to zero on power up. Wraps back to zero every 24 hours.
15	rcxParmMessage	Message	R/W	2.0 / New	0 - 1	Message and optional parameter. Index 0 access the message and index 1 accesses the message parameter.
16	rcxCyberAGC	N/A		2.0		Cybermaster only.

Source	Symbolic Name	Name	Access	Version	Index	Comments
17	rcxGlobalMotorStatus	Global Motor Status	R/W	2.0		bit 0-2: global max power; 3-3: global direction; 7-7: global on-off Currently not implemented in new firmware and returns zero.
18	rcxParmTaskStackVar Byte	Task Stack	R/W	New	0 - 255	Indexed by bytes from stack top. Future implementation
19	rcxParmTaskStackVar Word					
20	rcxParmTaskStackVar Long					
21	rcxParmCounter	Counter	R/W	2.0	0 - 2	Same as global variables 0-2. Used to generate counter events.
22	rcxParm1MsecTimer	Timer	R/W	New	0 - 3	Timer in 1-millisecond units. Can only write value to zero.
23	rcxParmTaskEvents	Task Events	R/O	2.0	0 - 9, 10	Bit mask showing which events alerted the task. Index value of 10 indicates current running task.
24	rcxParmSystem	System	R/W	New	0 - N	Contains many indexes to customize RCX firmware. See following table for meaning of specific index values.
25	rcxParmEventState	Event State	R/O		0 - 15	Current event state. Low, normal, or high
26	rcxParm10MsecTimer	Timer	R/W	2.0	0 - 3	Timer in 1-millisecond units. Can only write value to zero.
27	rcxParmClickCounter	Event Clicks	R/W	2.0	0 - 15	Event. Numb of clicks detected for event sensor
28	rcxParmUpperThresho ld	Event Upper Threshold	R/W	2.0	0 - 15	Event. Upper threshold for event generation
29	rcxParmLowerThresh old	Event Lower Threshold	R/W	2.0	0 - 15	Event. Lower threshold for event generation
30	rcxParmHysteresis	Event Hysteresis	R/W	2.0	0 - 15	Event. Hysteresis value for event generation
31	rcxParmDuration	Event Duration	R/W	2.0	0 - 15	Event. Blink time. Measured in 10 msec units. Minimum value is 5
32	rcxParmMotorPower1 28	Motor Power Level	R/W	New	0 - 2	Motors. Power levels using 0..127 range
33	rcxParmUARTSetup	UART Setup	R/W	2.0 / New		Used to set up data and serial link parameters for a custom message on the infrared serial link. In "new" firmware, can also configure the serial link parameters for permanent use. See detailed documentation.

Source	Symbolic Name	Name	Access	Version	Index	Comments
34	rcxParmAvgBatteryLevel	Average Battery Level	R/O	New	ignored	Battery voltage level averaged over the past 32 samples.
35	rcxParmFirmwareVersion	Firmware Version	R/O	2.0	ignored	Firmware version.
36	rcxParmIndirectVarInt	Indirect Variable	R/W	2.0	0 - 47	Indirect variable. Treat index as a variable (source == 0) and look up value. Then use this value as index into variable (source == 0) for read or write access.
37	rcxParmGlobalVar	Global Variable	R/W	New	0 - 255	Global variable. Expands the range of global variables to 255 from 32. Global variables can be 16-bit integers, 32-bit integers or 32-bit float values. The 32-bit variables are stored in two consecutive 16-bit values; first 16-bits is high word.
38	rcxParmIndirectGlobalInt	Indirect Variable	R/W	New		Similar to source == 36, but treats indices as global variables. Depending on source, result address is treated as 16-bit int, 32-bit int or 32-bit float.
39	rcxParmIndirectGlobalLong					
40	rcxParmIndirectGlobalFloat					
41	rcxParmIndexedGlobalAndConstant	Indexed Variable	R/W	New		Used to provide efficient array access into the array of global variables. For example, accessing 'array[i]' where 'array' starts at global address 32 (or hex 0x20) and 'i' is stored in global variable 15 (0x0F0 would have an index value of 0x200F. The upper byte contains the base address and the lower byte contains the index (global) variable. Indexing into both 16-bit and 32-bit arrays are supported within firmware using the two source value. Upper index byte cannot be zero (i.e. don't allocate arrays starting at variable zero). This restriction enables run-time firmware efficiency.
42	rcxParmIndexedGlobalLongAndConstant					
43	rcxParmStackVar	Future	R/W	New	0 - 255	Placeholder for future implementation
44	rcxParmConstantVar					
45	rcxParmVarByte					
46	rcxParmVarWord					
47	rcxParmVarLong					
48	rcxParmEventType	Event Type	R/W		0 - 15	Type of event being detected
49	rcxParmTaskVar	Task Variable	R/W	New	0 - 15	Provides direct indexing into task variables.
50	rcxParmMotorPower8	Motor	R/W	New	0 - 2	New. Power levels 0 - 7 range

Source	Symbolic Name	Name	Access	Version	Index	Comments
		Power				
51	rcxParmEvent	Event Sensor Type			0 - 15	Type of sensor for event
52	rcxParmSoundInfo	Sound Variables	R/O and R/W	New		Provides access to information about speaker and sound status.
53	rcxParmEventCounts	Event Counts	R/W	New	0 - 15	Counts the number of times an event has occurred.
54	rcxParmTaskStackSize	Task Stack Size	R/O	New	0 - 9, 10	Retrieves the size of the stack for the selected task. Future. Not yet implemented.
55	rcxParmTaskStackAddress	Task Stack Address	R/O	New	0 - 9, 10	Used by PC debugging routines to retrieve the address of a task stack.
56	rcxParmFunctionReturnValueWord	Function Return Value	R/W	New	ignored	Used to set a return value for a subroutine. And to retrieve the last return value from subroutine. Useful for implementation of recursive subroutines.
57	rcxParmFunctionReturnValueLong					
58	rcxParmFunctionReturnValueFloat					

Efficient Array Access Source

Source types ‘rcxParmIndexedGlobalAndConstant’ and ‘rcxParmIndexedGlobalLongAndConstant’ are designed for efficient generation of array access. The upper byte of the index is the base variable “address” of the array; the lower byte specifies a global variable that contains the array index. The effective address of the variable accessed is the base address + contents of the index variable.

- ‘rcxParmIndexedGlobalAndConstant’ is used for accessing word (16-bit variables) arrays.
- ‘rcxParmIndexedGlobalLongAndConstant’ is used for accessing long and float arrays. In this case the interpreter internally multiplies the index variable by two to get the address offset by the base since each long/float takes two words.

System Parameter Types

Description of index variables and their meanings for source type 24 (rcxParmSystem).

Index	Symbolic Name	Access	Comments
0	kSystemImmediateBatteryLevel	R/O	Battery voltage level from the last A/D sample
1	kSystemDebugTaskMode	R/O	
2	kSystemMemoryMapAddress	R/O	Start address in RAM of the interpreter's memory map.
3	kSystemCurrentTask	R/O	Number of currently executing task.
4	kSystemSerialLinkStatus	R/O	Status on the infrared serial link.
5	kSystemOpcodesPerTimeslice	R/W	Number of opcodes to execute per interpreter time slice. [Only applies to the 100X firmware].
6	kSystemMotorTransition	R/W	Delay period, in milliseconds, when flipping direction of motor movement. Brake is applied during this delay.
7	kSystemSensorRefreshRate	R/W	Refresh rate, in milliseconds, for scanning sensors. Default is 3. Value of 0 indicates 0.5 msec rate should be used.
8	kSystemExpandedRemoteControlMessages	R/W	
9	kSystemLCDRefreshRate	R/W	Refresh rate, in 100 msec ticks, for the LCD display. Value of zero indicates no refresh.
10	kSystemNoPowerDownOnACA daptor	R/W	Boolean flag to indicate no power down of RCX when operating on AC adaptor.
11	kSystemDefaultTaskStackSize	R/W	Used to set the size of a task stack. Applies to the next allocation of task stacks. Task stacks are allocated or resized when a task is started. Memory allocated to task stack remains allocated until a task is deleted.
12	kSystemTaskPriority	R/W	Priority of current task.
13	kSystemTransmitterRange	R/W	Short or long range on the RCX infrared transmitter.
14	kSystemFloatDuringInactiveMotorPWM	R/W	Boolean flag to indicate whether 'float' (default) or 'brake' should be used during inactive motor PWM cycles.
15	kSystemRotationErrorsCount	R/W	
16	kSystemRotationDebouncedGlitches	R/W	Number of glitches detected on rotation counter that have been 'debounced'. Primarily a debugging aid and may not persist in the firmware.

Index	Symbolic Name	Access	Comments
17	kSystemPreambleSize	R/W	Size of preamble to use
18	kSystemUnsolicitedMessages	R/W	Boolean flag to indicate whether message should be sent on infrared link when exception occurs. A PC development system could listen for these messages.
19	kSystemExpandedNumOfSubs	R/W	Boolean flag to indicate whether 8 or 40 subroutines should be supported.
20	kSystemPowerDownDelay	R/W	Minutes to delay before power down.
21	kSystemWatchFormat	R/W	Display format for the watch display on the LCD.
22	kSystemSensorMissedConversions	R/W	Number of times the A/D conversion routine had inadequate real time to complete. Primarily a debugging aid and may not persist in the firmware.
23	kSystemMotorControlRegiser	R/W	Debugging aid.

UART Setup Parameter Types

Description of index variables and their meanings for source type 33 (rcxParmUARTSetup).

Index	Symbolic Name	Access	Comments
0 - 15	setupMessageByteXX	R/W	Setup data bytes for user generated message
16	setupUserMessageHeaders	R/W	Setup message header parameters for user generated message. Format as defined below: <pre>typedef enum { UARTmsgIncludeNone = 0x00, UARTmsgIncludePreamble = 0x01, UARTmsgIncludeComplements = 0x02, UARTmsgIncludeChecksum = 0x04 } TUARTpreamble;</pre>
17	setupUserBaudRate	R/W	Setup baud rate for user generated message. Format as defined below. <pre>typedef enum { kUARTdefaultSpeed= 0x00, //2400 38kHz, 50% duty cycle UART4800baud = 0x01, //zero in bitfield is 2400 baud UART76KHz = 0x02, //zero in bitfield is 38kHz UART25dutyCycle = 0x04 //zero in bitfield is 50% duty //cycle UART9600baud = 0x01, //doesn't work } TUARTspeed;</pre>
18	setupNormalMessageHeaders		Setup message header parameters for normal messaging
19	setupNormalBaudRate		Setup baud rate for normal messaging

Sound Parameter Types

Description of index variables and their meanings for source type 33 (`rcxParmUARTSetup`).

Index	Symbolic Name	Access	Comments
0	<code>rcxParmVolume</code>	R/W	Control the “volume” of the speaker.
1	<code>rcxParmBOOLSOUNDPlaying</code>	R/O	Boolean flag to indicate whether sound is currently playing. True if sound is active.
2	<code>rcxParmPlaySounds</code>	R/W	Boolean flag to play or mute sound on the speaker.
3	<code>rcxParmQueuedSoundCount</code>	R/O	Count of the number of playing and queued sounds.

Task Priority Expansion

The standard firmware stores a one-byte priority field for each task. The default value is 7. The only use of task priority is for allocation/reservation of devices (sensors, timers) as part of the device management routines. The standard firmware has an opcode, `rcxSetPriority`, to set the task device allocation priority.

In the expanded firmware, task priority has been expanded to control both devices and CPU time. An additional priority variable was defined to specify the CPU scheduling priority.

Priority management for devices continues as standard firmware implementation.

The expanded firmware contains two new intrinsic variables to read/write both priority fields. This variable is the only way to access the scheduling priority; a new opcode was not defined for this purpose.

The CPU scheduler selects the highest priority runnable tasks for execution. All tasks at this highest priority are then given time slices in a round robin scheduling fashion. Lower priority tasks will not receive any CPU time. Non-runnable tasks, i.e. those that are waiting for a timer or are stopped, are not assigned time slices. The only way for a lower priority task to receive CPU time is for all higher priority tasks to become non-runnable.

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.

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 (0-470. These opcodes have identical format and parameters as existing arithmetic opcodes to simplify conversion of programming systems.

Global Variables	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

Interpreter Opcode Overview

Changes in Opcode Sizes

The existing Lego standard firmware uses a simple scheme for determining the actual size of an opcode. The size of an opcode can be calculated by looking at the lower three bits of the opcode. The size is determined by the following calculation.

```
nBytesInOpcode = (ubyte) (opcode & 0x07);
if (nBytesInOpcode > 5)
    nBytesInOpcode -= 5;
else
    ++nBytesInOpcode;
```

This “rule” limits the flexibility in defining new opcodes because there may not be an undefined opcode that matches the size criteria of the new opcode. The new firmware uses a table lookup to determine the size so any opcode can be any size.

Another “rule” in the existing firmware restricted the system to a total of 128 opcodes (i.e. 0x10 and 0x18 are the same opcode). This rule is important for the “direct control” opcodes that can be executed from messages sent from the PC. When two consecutive messages from the PC are sent, the firmware discards second and subsequent messages that have the same opcode; this is to eliminate retransmission of messages on errors where the firmware received the opcode correctly but the PC did not get the reply. The PC program needs to “flip” the opcode so that firmware can distinguish between retransmission and new message. Since the new opcodes are, in general, not the “direct control” type, this opens up an additional 128-opcode slots.

Changes in Direct Opcodes

Direct opcodes are those that can be sent as a single infrared message. When a direct opcode message is received by the RCX it is immediately executed. The enhanced firmware continues to support all direct opcodes with one significant improvement.

The existing Lego standard firmware flips bit 3 between successive direct opcode messages with the same opcode. The intent is to distinguish between retransmission of a failed message and a new message. If two messages are for the same opcode and bit 3 is not flipped, then the standard firmware simply repeats the previous reply message and does not ‘execute’ the second message. The enhanced firmware eliminates the need for flipping bit three for all but a handful of opcodes.

- A flipped bit three is not required for direct opcodes that have a benign impact on the state of the RCX interpreter. Opcodes with a benign impact are those like ‘send value of a variable’, ‘set sensor type or sensor mode’ or even ‘start task’.
- Flipped bit 3 is still required for those opcodes where a repetition would have an undesirable effect. This only impacts the 11 opcodes (‘addTo’, ‘minusTo’, ‘divideTo’, other similar arithmetic opcodes and ‘datalogNext’) where a repetition would adversely impact the value of variables.

Existing programs will find the enhanced firmware implementation transparent. A significant benefit of the enhanced firmware implementation is that it now enables successful allow recover from messaging errors during program download. If a program download message is corrupted and the PC program retransmits the message in a recovery attempt, then the standard firmware will simply repeat the previous error message. The enhanced firmware will reprocess the repeated message and can respond with a success message if the message was received OK.

Existing 16-Bit Integer Arithmetic Operations

The existing interpreter has six opcodes for arithmetic operations. These opcodes are all five bytes in length and have the format described in the following table. These opcodes operate on variables of source type 0 (rcxParmVar) only. The operand can be any source type. The format and appropriate opcodes are.

Opcode	Result Index Low	Operand		
		Type	Index Low	Index High

Opcode	Symbolic Name	Description
0x14	rcxAssign	result = operand;
0x24	rcxAddTo	result += operand;
0x34	rcxMinusTo	result -= operand;
0x54	rcxTimesTo	result *= operand;
0x44	rcxDivideTo	result /= operand;
0x84	rcxAndTo	result &= operand;
0x94	rcxOrTo	result = operand;
0x2E	rcxBitComplement (new)	result = ~operand;
0xF8	rcxModuloTo (new)	result %= operand;

The last two are new additions to the interpreter to round out the suite of arithmetic operations.

The interpreter also supports a “generic” assignment opcode. This opcode is used to assign values to any valid interpreter variable or intrinsic.

Opcode	Result		Operand		
	Type	Index Low	Type	Index Low	Index High

Opcode	Symbolic Name	Description
0x05	rcxSetSourceValue	result = operand;

New Arithmetic Opcode Descriptions

Generic 16-Bit Integer Arithmetic Operations

The new firmware expands upon the existing arithmetic opcodes to provide a set of opcodes that operate on “generic” values using three byte formats for the result and operands. These opcodes accept any type of source for both the result and operand values.

The common format for these generic arithmetic opcodes and a table of the opcodes is below.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High
Opcode	Symbolic Name			Description		
0x2F	rcxAssignSourceValue			result = operand;		
0xF8	rcxAddToGenericParm			result += operand;		
0xF9	rcxMinusToGenericParm			result -= operand;		
0xFA	rcxTimesToGenericParm			result *= operand;		
0xFB	rcxDivideToGenericParm			result /= operand;		
0xFC	rcxAndToGenericParm			result &= operand;		
0xFD	rcxOrToGenericParm			result = operand;		
0xFE	rcxBitComplementGenericParm			result = ~operand;		
0xFF	rcxModuloToGenericParm			result %= operand;		

These opcodes provide complete flexibility on ‘result’ and ‘operand’ values for 16-bit integer arithmetic. However, they do not provide the most efficient real time speed because of the overhead of examining both operand types at run time to determine the variable type. As well, they are bigger at seven bytes in length.

Pseudo code for the internal implementation of these opcodes is given below.

```
{
  word nResult;
  word nOperand;

  nResult = getCommonParameterValue012();
  nOperand = getCommonParameterValue345();
  stdArithmeticOp(&nResult, nOperand);
  setValue012(nResult);
}
```

There are four subroutine calls involved in the pseudo code as shown in the following table.

Subroutine	Description
getCommonParameterValue012	Retrieves the value of the result source stored in opcode parameter bytes 0, 1 and 2. This is a ‘relatively’ expensive routine to perform as it must first do a ‘switch’ statement on the source type and then perform type specific lookup/calculations

Subroutine	Description
	to determine the value.
getCommonParameterValue345	Retrieves the value of the result source stored in opcode parameter bytes 0, 1 and 2.
stdArithmeticOp	Performs the appropriate arithmetic operation. Internally it knows that the arithmetic operation is specified by the lower three bits of the opcode.
setValue012	Stores the value into the source specified opcode parameter bytes 0, 1 and 2.

From a CPU overhead perspective, each of these four different routines has approximately the same execution time. For purposes of comparison consider the relative CPU overhead of these opcodes as '4'. With some simple enhancements, it is very easy to define additional "opcode sets" that reduce this overhead to '2' or even '1'. The types of optimizations and their impact are:

Pseudo Code	Description
<pre>{ word *pResult; word nOperand; pResult = getResultParameterAddress(); nOperand = getOperandParameterValue(); stdArithmeticOp(pResult, nOperand); }</pre>	<p>Usually the source types will be a RAM memory variable. The 'get' and 'set' routines can be replaced with a single call to a 'getAddress' routine.</p> <p>Relative weight of 3.</p>
<pre>{ word *pResult; word nOperand; pResult = &globalVariable[index]; nOperand = getOperandParameterValue(); stdArithmeticOp(pResult, nOperand); }</pre>	<p>Frequently the result is one of the 256 global variables. The address can be immediately determined without a procedure call.</p> <p>Relative weight of 2.</p>
<pre>{ word nResult; nResult = getCommonParameterValue012(); stdArithmeticOp(&nResult, constantParmValue); setValue012(nResult); }</pre>	<p>Often the operand is a constant which can be directly retrieved from the opcode with.</p> <p>Relative weight of 3.</p>
<pre>{ word *pResult; pResult = &globalVariable[index]; stdArithmeticOp(pResult, constantParmValue); }</pre>	<p>Combines both optimizations</p> <p>Relative weight of 1.</p>

Several new sets of opcodes have been designed to take advantage of the above optimizations to reduce the execution speed of an opcode. With the 100X firmware version, the execution speed difference between the generic "do it all" format and a optimized opcode can be significant; for example, 85 vs 20 microseconds for "nIndex += 3".

A similar type of optimization is possible to reduce the size of an opcode. The generic opcodes are seven bytes in length; one byte for the opcode and three bytes for each of the two parameters.

Parameter Format			Description
Source Type	Index Low	Index High	Fully generic version. Supports all source types.
Source Type	Index Low		Only a few source types actually require a 16-bit index (e.g. a 16-bit constant value or some of the indirect access source types). For the vast majority of cases, the source type fits in two bytes.
	Index Low	Index High	The source type can be implied from the opcode. For example, for a constant parameter as in the “playTone” opcode.
	Index Low		Implied source type and it fits in one byte. For example, a global variable reference.

New opcode sets have been designed for the high runner cases to take advantage of the smaller opcode sizes possible. For example “nIndex += 3” fits in three bytes.

The remainder of this chapter describes the new arithmetic opcode sets that have been designed to take advantage of these speed and space savings.

16-Bit Integer Arithmetic Operations, Memory Parameters Only

Same format as the generic opcodes but result operand must be a RAM memory variable.

The common format for these generic arithmetic opcodes and a table of the opcodes is below.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High
Opcode	Symbolic Name			Description		
0x2F	rcxAssignSourceValue			result = operand;		
0x08	rcxAddToSourceValue			result += operand;		
0x09	rcxMinusToSourceValue			result -= operand;		
0x0A	rcxTimesToSourceValue			result *= operand;		
0x0B	rcxDivideToSourceValue			result /= operand;		
0x0C	rcxAndToSourceValue			result &= operand;		
0x0D	rcxOrToSourceValue			result = operand;		
0x0E	rcxBitComplementSourceValue			result = ~operand;		
0x0F	rcxModuloToSourceValue			result %= operand;		

One assignment optimization for single byte operands is the following.

Opcode	Result			Unsigned byte
	Type	Index Low	Index High	
Opcode	Symbolic Name			Description
0x2F	rcxSetSourceValueByteConst			result = const;

Optimized Opcodes for Global Variables

These opcodes are designed to work with the new expanded range of global variables. They implicitly know the result is a global variable so that only one (vs. three) byte is required to specify the result parameter.

Global variables 0 to 2 have special meaning within the interpreter. They are counter variables, which can be used in event definitions. Checking for “counter” variable is not performed on the assignment results of these opcodes. In fact, counter variable checking is only performed when the original Lego defined opcodes are used.

Opcode	Result	Operand	
	Index Low	Type	Low
Opcode	Symbolic Name		Description
0x2D	rcxAssignGlobalVariable		result = operand;
0xC8	rcxAddToGlobal		result += operand;

Opcode	Symbolic Name	Description
0xC9	rcxMinusToGlobal	result -= operand;
0xCA	rcxTimesToGlobal	result *= operand;
0xCB	rcxDivideToGlobal	result /= operand;
0xCC	rcxAndToGlobal	result &= operand;
0xCD	rcxOrToGlobal	result = operand;
0xCE	rcxBitComplementGlobal	result = ~operand;
0xCF	rcxModuloToGlobal	result %= operand;

This suite of opcodes provide operation on global variables when the operand is an unsigned constant that fits in a single byte.

Opcode	Result	Constant unsigned byte
	Index Low	

Opcode	Symbolic Name	Description
0x2B	rcxAssignGlobalConstantByte	result = operand;
0x48	rcxAddToGlobalUbyte	result += operand;
0x49	rcxMinusToGlobalUbyte	result -= operand;
0x4A	rcxTimesToGlobalUbyte	result *= operand;
0x4B	rcxDivideToGlobalUbyte	result /= operand;
0x4C	rcxAndToGlobalUbyte	result &= operand;
0x4D	rcxOrToGlobalUbyte	result = operand;
0x4E	rcxBitComplementGlobalUbyte	result = ~operand;
0x4F	rcxModuloToGlobalUbyte	result %= operand;

This suite of opcodes provide operation on global variables when the operand is a signed 16-bit constant.

Note: The stored format of the constant is the reverse of the low / high order used in the existing opcodes. The C compiler used to compile the firmware generates more efficient code for this situation.

Opcode	Result	Constant ¹	
	Index Low	High	Low

Opcode	Symbolic Name	Description
0x2C	rcxAssignGlobalConstant	result = operand;

¹ The stored format of the constant is the reverse of the low / high order used in the existing opcodes. The C compiler used to compile the firmware generates more efficient code for this situation.

Opcode	Symbolic Name	Description
0x58	rcxAddToGlobalConstant	result += operand;
0x59	rcxMinusToGlobalConstant	result -= operand;
0x5A	rcxTimesToGlobalConstant	result *= operand;
0x5B	rcxDivideToGlobalConstant	result /= operand;
0x5C	rcxAndToGlobalConstant	result &= operand;
0x5D	rcxOrToGlobalConstant	result = operand;
0x5E	rcxBitComplementGlobalConstant	result = ~operand;
0x5F	rcxModuloToGlobalConstant	result %= operand;

32-Bit Integer Arithmetic

The new firmware (optionally) provides support for 32-bit signed integer arithmetic.

Opcodes in the following arithmetic set assume both result and operand are 32-bit integer memory¹ variables and are not intrinsic variables.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High
Opcode	Symbolic Name			Description		
0x26	rcxAssignLong			result = operand;		
0x18	rcxAddToLong			result += operand;		
0x19	rcxMinusToLong			result -= operand;		
0x1A	rcxTimesToLong			result *= operand;		
0x1B	rcxDivideToLong			result /= operand;		
0x1C	rcxAndToLong			result &= operand;		
0x1D	rcxOrToLong			result = operand;		
0x1E	rcxBitComplementLong			result = ~operand;		
0x1F	rcxModuloToLong			result %= operand;		

Type conversion from 16-bit to 32-bit integer is performed with the following opcode.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High
Opcode	Symbolic Name			Description		
0x8D	rcxIntegerToLong			result = operand;		

Type conversion from 32-bit to 16-bit integer is performed with the following opcode.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High
Opcode	Symbolic Name			Description		
0xAF	rcxLongToInteger			result = operand;		

Assignment of constant values to 32-bit integer variables has the following several opcodes dependent on the size of the constant.

¹ Memory variables refer to those that end up as global or task variables. This includes the indirect and indexed source types. It excludes intrinsic variables (e.g. sensor values).

Assignment of 8-bit signed constant uses the following.

Opcode	Result			Signed byte
	Type	Index Low	Index High	

Opcode	Symbolic Name	Description
0x76	rcxAssignLongConstantByte	result = constant;

Assignment of 16-bit constant uses the following.

Opcode	Result			Constant ¹	
	Type	Index Low	Index High	Index High	Index Low

Opcode	Symbolic Name	Description
0x28	rcxAssignLongConstantWord	result = constant;

Assignment of 32-bit constant uses the following. Byte 3 is the high byte of the constant. This opcode can also be used to assign to float results.

Opcode	Result			Constant			
	Type	Index Low	Index High	Byte 3	Byte 2	Byte 1	Byte 0

Opcode	Symbolic Name	Description
0x29	rcxAssignLongConstant	result = constant;

¹ The stored format of the constant is the reverse of the low / high order used in the existing opcodes. The C compiler used to compile the firmware generates more efficient code for this situation.

Floating Point Arithmetic

The new firmware (optionally) provides support for 32-bit floating-point arithmetic. All float opcodes have the following format

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High

Opcodes in the following arithmetic set assume both result and operand are floating variables and are not intrinsic variables.

Opcode	Symbolic Name	Description
0x26	rcxAssignLong	result = operand;
0x38	rcxAddToFloat	result += operand;
0x39	rcxMinusToFloat	result -= operand;
0x3A	rcxTimesToFloat	result *= operand;
0x3B	rcxDivideToFloat	result /= operand;
n/a	undefined	result &= operand;
		result = operand;
		result = ~operand;
		result %= operand;

Type conversion from 16-bit to 32-bit integer is performed with the following opcodes.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High

Opcode	Symbolic Name	Description
0xAC	rcxIntegerToFloat	(float)result = (int) operand;
0xAD	rcxLongToFloat	(float)result = (long) operand;
0xAE	rcxFloatToInteger	(int)result = (float)operand;
0x8C	rcxFloatToLong	(long)result = (float) operand;

Constants may be assigned to floating point variables using the same opcodes as for 32-bit integers. It is the “raw” constant that is assigned to the variable, so it should already be in floating point format.

Other Arithmetic Opcode Sets

There's a couple of additional sets of arithmetic opcodes that have been implemented. Probably a little aggressive in defining new opcodes. They may be overkill in terms of code efficiency and may not survive in ongoing evolution. These include.

Optimized Two Byte Result Variables

Opcode	Result		Operand	
	Type	Index Low	Type	Index Low

Opcode	Symbolic Name	Description
0x36	rcxSetSourceValueShortVariable	result = operand;
0xB8	rcxAddToShortVar	result += operand;
0xB9	rcxMinusToShortVar	result -= operand;
0xBA	rcxTimesToShortVar	result *= operand;
0xBB	rcxDivideToShortVar	result /= operand;
0xBC	rcxAndToShortVar	result &= operand;
0xBD	rcxOrToShortVar	result = operand;
0xBE	rcxBitComplementShortVar	result = ~operand;
0xBF	rcxModuloToShortVar	result %= operand;

Opcode	Result		Constant	
	Type	Index Low	High Byte	Low Byte

Opcode	Symbolic Name	Description
xx	xx	result = operand;
0xD8	rcxAddToShortVarConstant	result += operand;
0xD9	rcxMinusToShortVarConstant	result -= operand;
0xDA	rcxTimesToShortVarConstant	result *= operand;
0xDB	rcxDivideToShortVarConstant	result /= operand;
0xDC	rcxAndToShortVarConstant	result &= operand;
0xDD	rcxOrToShortVarConstant	result = operand;
0xDE	rcxBitComplementShortVarConstant	result = ~operand;
0xDF	rcxModuloToShortVarConstant	result %= operand;

32-bit Arithmetic with Long Constant Opcodes

.Opcode	Result			Constant			
	Type	Index Low	Index High	Byte 3	Byte 2	Byte 1	Byte 0

Opcode	Symbolic Name	Description
0x29	rcxAssignLongConstant	result = constant;
0x68	rcxAddToLongConstant	result += operand;
0x69	rcxMinusToLongConstant	result -= operand;
0x6A	rcxTimesToLongConstant	result *= operand;
0x6B	rcxDivideToLongConstant	result /= operand;
0x6C	rcxAndToLongConstant	result &= operand;
0x6D	rcxOrToLongConstant	result = operand;
0x6E	rcxBitComplementLongConstant	result = ~operand;
0x6F	rcxModuloToLongConstant	result %= operand;

Generic Absolute Value Opcode

Generic 16-bit integer opcode for absolute value. Opcode is rcxAbsoluteValueSourceValue.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High

Conditional Branch Opcodes

Naturally new conditional branch instructions are required to support the long and float variables. There are two conditional branch opcodes in the existing firmware; the first (or left) operand is two bytes and the second (or right) operand is three bytes. These are capable of supporting 16-bit integer arithmetic with the expanded address range although they cause some problems to a compiler; for example large constants can only fit in the right operand.

The following three opcodes are all that is required to fully support int, long and conditional branch within the interpreter. They provide complete flexibility for conditional branching. All others are variants that reduce the size of the opcode or provide real time efficiency.

Opcode	Left Side			Right Side			Branch Offset	
	Type*	Index Low	Index High	Type	Index Low	Index High	High#	Low

Opcode	Symbolic Name	Description
0x3E	rcxTestAndBranchIntBigFar	16-bit signed comparison
0x3D	rcxTestAndBranchLongBigFar	32-bit signed comparison
0x3F	rcxTestAndBranchFloatBigFar	Floating point comparison

As above, but with a single byte offset.

Opcode	Left Side			Right Side			Offset
	Type*	Index Low	Index High	Type	Index Low	Index High	Low

Opcode	Symbolic Name	Description
0x8F	rcxTestAndBranchIntBigNear	16-bit signed comparison
0x3C	rcxTestAndBranchLongBigNear	32-bit signed comparison
0x89	rcxTestAndBranchFloatBigNear	Floating point comparison

Next the two operands both fit in two bytes so the opcode can be smaller.

Opcode	Left Side		Right Side		Branch Offset	
	Type*	Index Low	Type	Index Low	High#	Low

Opcode	Symbolic Name	Description
0xD5	rcxTestAndBranchIntShortFar	16-bit signed comparison
0x96	rcxTestAndBranchLongShortFar	32-bit signed comparison
0x86	rcxTestAndBranchFloatShortFar	Floating point comparison

Format is different from that used in the Lego firmware. Lego firmware uses low/high byte order.

* Upper two bits contain condition type (==, !=, <=, >=) and lower six bits contain the source type.

As above, but with a single byte branch offset.

Opcode	Left Side		Right Side		Offset
	Type*	Index Low	Type	Index Low	Low

Opcode	Symbolic Name	Description
0x8E	rcxTestAndBranchIntShortNear	16-bit signed comparison
0x8B	rcxTestAndBranchLongShortNear	32-bit signed comparison
0x8A	rcxTestAndBranchFloatShortNear	Floating point comparison

The following four opcodes are purely for faster real time performance improvement. They are the same format as the generic “big format” (three bytes for each parameter) 16-bit conditional branch except either the left side or the right side is a constant. The interpreter is able to gain faster access to the constant than if it had to do a run-time lookup; new opcodes are simply defined so that the interpreter knows which operand is a constant. These opcodes are about 40% faster during execution than the generic case.

Opcode	Constant Byte Parm		Right Side		Branch Offset
	Type*	Low Byte	Type	Index Low	Low

Opcode	Symbolic Name	Description
0x56	rcxTestAndBranchIntConstShortNear	16-bit signed comparison

Opcode	Left Side		Constant Byte Parm		Branch Offset
	Type*	Index Low	02	Low Byte	Low

Opcode	Symbolic Name	Description
0x78	rcxTestAndBranchIntShortConstNear	16-bit signed comparison

Opcode	Left Side			Right Side Constant			Offset
	Type*	Index Low	Index High	02	Low Byte	High Byte	Low

Opcode	Symbolic Name	Description
0x79	rcxTestAndBranchIntBigConstNear	16-bit signed comparison

Opcode	Left Side Constant			Right Side			Offset
	Type*	Low Byte	High Byte	Unused	Index Low	Index High	Low

Opcode	Symbolic Name	Description
0x57	rcxTestAndBranchIntConstBigNear	16-bit signed comparison

‘Switch’ Opcodes

There are four new opcodes for extremely efficient implementation of ‘C’ language ‘switch’ statements in a single opcode. The size of these opcodes is dynamic based on the number of case labels. Without these opcodes, a compiler implements a “switch” statement as a group of consecutive “conditional branch” opcodes, one condition test for each “case” value; this can result in slow execution especially when the switch variable matches a case at the end of the list.

rcxSwitch and rcxSwitchByteCase

These two opcodes can handle any type of switch. The opcodes contains two tables; one contains the case label values and the second table contains the branch offset. Opcode processing searches the case value table for a match with the selection variable and then uses the appropriate index into the relative branch offset table.

The format of these opcodes is in the following table. The first byte is the opcode followed by the table size (i.e. number of case labels), then the switch selection variable (two bytes), then a table of case labels and finally a table of branch offsets.

Opcode	
Table Size	
Arg Type	Arg Index
Case Value 0	
Case Value 0	
• • •	
Case Value “size – 1”	
Offset. Case 0	
Offset. Case 1	
• • •	
Offset. Case “size – 1”	
Default Case Offset	

rcxSwitch stores the case values as two bytes. **rcxSwitchByteCase** uses a single byte. The table of branch offsets always uses two bytes per entry.

rcxSwitchIndexTableNear and rcxSwitchIndexTableFar

Often the case labels in a switch statement are almost consecutive linear values. Optimized opcodes are provided for this. The case value table is eliminated and replaced with simply a minimum value; the index into the relative branch table is the selection value minus the minimum value.

Consider the example where case labels were 5, 6, 7, 9, 10. The opcode contains the number of explicit case labels (i.e. $6 = \max - \min + 1$), the selection variable, the minimum case label and a table of six branch offsets followed by the default case branch offset. Case label 8 will also point the default case offset. Two forms of the opcode are used depending on whether one or two bytes are needed for the branch offset table.

Opcode	
Table Size	
Arg Type	Arg Index
Min High	Min Low
Offset. Case 0	
Offset. Case 1	
• • •	
Offset. Case "size - 2"	
Offset. Case "size - 1"	
Default Case Offset	

Unconditional Branch Opcodes

Opcode	Offset Bits 0..6 and Direction		RcxBranchNear
Opcode	Offset Bits 0..6 and Direction	Offset High bits 7..14	RcxBranchFar

The existing firmware uses a complex format to represent the offset of a branch instruction. For small one-byte branches, bit 7 contains the direction (0 = forward, 1 = backwards) of the branch and bits 0..6 contain a positive number representing the relative offset of the branch. For two-byte branches the first byte contains the direction (sign) and least significant seven bits of the branch and the second byte contains the most significant bits 7..14. This leads to a complicated calculation that must be made at run time for the branch location. This is shown below for the ‘far’ branch format.

```

uword calculateFarBranchRelativeOffsetLegoFormat(ubyte nOffsetInOpcode)
{
    register pUbyte pOffsetLocation;
    register ubyte lowByte;
    register uword offset;

    pOffsetLocation = ((pUbyte) pCurrOp) + nOffsetInOpcode;
    {
        register pUbyte pTemp;

        pTemp = pOffsetLocation;
        lowByte = *pTemp++;
        offset = *pTemp; // high byte
    }
    offset <<= 7; // 14 instructions on the H8 CPU! Two instructions to shift 1-bit.
    offset += (ubyte) (lowByte & 0x7F);

    if (lowByte & 0x80)
        return -offset;
    else
        return offset;
}

```

Opcode	Signed Char Offset		RcxBranchNearByte
Opcode	Signed Unpacked Word Offset		RcxBranchFarWord
	High Byte	Low Byte	

Two new opcodes were defined to utilize the “native” numeric format of the H8 CPU. Small branches are stored in a signed character format and large branches in a unpacked (i.e. not word aligned) signed word format. The code for relative branch offset calculations becomes

```

uword calculateFarBranchRelativeOffsetNativeFormat(ubyte nOffsetInOpcode)
{
    register pPackedWord pOffsetLocation;

    pOffsetLocation = ((pUbyte) pCurrOp) + nOffsetInOpcode;
    return *pOffsetLocation;
}

```

Expanded User Infrared Messaging Support

Enhanced firmware has expanded support for user infrared messages to include a one word parameter.

Standard Firmware

The existing firmware has two messages for sending and receiving messages. The **rcxSendMessage** opcode causes a **rcxSetMessage** opcode message to be sent over the infrared link. The “mailbox ID” parameter is obtained by evaluating the argument source. Receiving RCXs execute the message by setting the argument value (one byte) into the **message** mailbox.

Opcode	Argument		rcxSendMessage
	Type	Index Low	
Opcode	Mailbox ID		rcxSetMessage
Opcode			rcxClearMessage

User programs can emulate reception of a mailbox message by directly executing inline the **rcxSetMessage** opcode.

The **message** “mailbox ID” is accessed using the **rcxParmMessage** source type. This intrinsic variable contains the ID of the last received message.

rcxClearMessage sets the “mailbox ID” to zero. Alternatively, they could set the **message** intrinsic variable to zero.

Opcode	rcxClearMessage
--------	------------------------

Enhanced Firmware

Standard firmware only sends a single parameter byte in the message which is frequently too limiting in trying to implement a messaging protocol between RCXs. Several new opcodes have been introduced.

rcxSendMessageParm causes the RCX to send a mailbox message over the IR link. The RCX evaluates the “parameter argument” and determines whether it will fit in zero, one or two bytes. It then selects the appropriate **rcxSetMessageXXX** message opcode to use based on the parameter size.

Opcode	Mailbox Argument		Parameter Argument			RcxSendMessageParm
	Type	Low Byte	Type	Low Byte	High Byte	

There are three **rcxSetMessageXXX** opcodes for zero, one or two parameter bytes. All the arguments to these opcodes are constants. Execution of these opcodes sets not only the **message** variable but also the new **messageParm** variable. **message** is accessed using the

rcxParmMessage source type with an index of zero. A non-zero index accesses the **messageParm** value.

Opcode	Mailbox ID
--------	------------

RcxSetMessage
(existing opcode in standard firmware)

Opcode	Mailbox ID	Byte Parm
--------	------------	-----------

RcxSetMessageByteParm

Opcode	Mailbox ID	Constant Parameter	
		Low Byte	High Byte

RcxSetMessageWordParm

During program development, it is frequently desirable to emulate a received message. This can be done with the **rcxSetMessageVariable** opcode. It has the format given below

Opcode	Mailbox Argument		Parameter Argument		
	Type	Low Byte	Type	Low Byte	High Byte

rcxSetMessageVariable

Miscellaneous New Opcodes

Array Bounds Limit Checking

The **rcxArrayBounds** opcode is used to perform an array bounds check. It contains the specification for an array index variable (two-byte format) and the specification for the upper limit on the array index (one byte constant). An exception is raised if the index variable is greater than or equal to the upper limit.

Opcode	Array Index Variable		Constant Bounds
	Type	Low Byte	

rcxArrayBounds

Useful “Extra” Arithmetic Functions

New opcodes have been defined to provide support for functions like “abs”, “sign”, shift operations and bit testing. These have the following format.

Opcode	Result			Operand		
	Type	Index Low	Index High	Type	Index Low	Index High

Opcode	Symbolic Name	Description
0x7B	rcxAbsoluteValueSourceValue	result = abs(operand);
0x7C	rcxSignValueSourceValue	result = sign(operand);
0x98	rcxShiftLeft	result <<= operand;
0x99	rcxShiftRight	result >>= operand;
0x88	rcxXOr	result ^= operand;
0x9B	rcxNegate	result = - operand;
0x7E	rcxBitSet	result = (1 << operand);
0x7F	rcxBitClear	result &= ~(1 << operand);

Program Memory Map and Allocation

Memory for tasks, subroutines, datalog and task stacks are all allocated from a single free memory block. Subroutines and tasks are stored in ascending memory order within this block. When the size of a item is changed all of the following items are shifted up/down depending on whether the new size is smaller or larger. The datalog is stored at the end of this free space.

Standard Firmware

The following ‘C’ structure could be used in the standard firmware implementation to provide the pointers to the starting locations for tasks/subroutines/datalog.

‘pDatalogHeadBlock’ is the starting location of the datalog. ‘pFirstFreeAddress’ is the first address of the first unused/unallocated memory.

```
struct
{
    uword    pSubroutineAddress[kNumOfPrograms][kNumOfSubroutines];
    uword    pTaskAddress[kNumOfPrograms][kNumOfTasks];
    uword    pDatalogHeadBlock;
    uword    pFirstFreeAddress;
    uword    pNextDatalogAddress; // Next location for storage
    uword    pLastAvailableMemory;
} ptrs;
```

The direct opcode ‘rcxGetMemoryMap’ uploads this structure to the PC. The structure (on the RCX’s H8 using 16-bit words) is 188 bytes in length and the received reply message is 189 bytes including the reply opcode.

Enhanced Firmware

The enhanced firmware adds task stacks to the items that can be dynamically allocated. Stacks contain subroutine return addresses and (eventually) local variables. Stack support enables the enhanced firmware to support nested subroutine calls vs. the single level of nesting supported in the standard firmware. The dynamic size avoids unnecessary memory usage for unallocated tasks and size can be adjusted depending on the level of nested subroutine calls required.

The initial implementation simply added the task stacks between the tasks and the datalog. It was fully compatible with the memory allocation routines and only required a few additional code lines. Unfortunately, this changed the size of the structure and resulted in the ‘rcxGetMemoryMap’ returning a 209-byte reply. This “broke” existing development environments (Robolab, BricxCC) that used this opcode. In particular, Robolab would first upload the memory map and then would use the contents to upload the programs from the RCX to the PC.

The “solution” for backwards compatibility was to store the stacks at the very end of available memory and grow them in descending memory order. The ‘C’ structure becomes:

```
struct
{
    uword    pSubroutineAddress[kNumOfPrograms][kNumOfSubroutines];
    uword    pTaskAddress[kNumOfPrograms][kNumOfTasks];
    uword    pDatalogHeadBlock;
    uword    pFirstFreeAddress;
    uword    pNextDatalogAddress;
    uword    pTaskStackAddress[kNumOfTasks];
}
```

```
    uword    pLastAvailableMemory;  
} ptrs;
```

'rcxGetMemoryMap' will only return the first 188 bytes of this structure. Existing IDEs can treat the start address of task zero (`pTaskStackAddress[0]`) as the last available memory location

Unfortunately, this requires expanded capabilities in the memory allocation routine. Unallocated space is now a hole in the middle rather than at the very end and a new allocation routine is required for allocating stacks downwards from the end of available memory. The quick implementation was to fix the stack size at 8 (room for two subroutine return addresses) and allocate all task stacks in the one-time initialization routine called once after firmware download.

This provides complete compatibility with Robolab and BricxCC with the limitation of only two levels of subroutine calls.

The complete structure can be uploaded to the PC using the `rcxGetProgramDataBytes` (`startAddress, numberOfBytes`) opcode. This is a generic routine (also available in the standard firmware) that allows upload of a range of memory from the RCX. Of course, you can't upload the structure without knowing the appropriate start address; the system parameter `kSystemMemoryMapAddress`, provides this address.

What's Not Documented Yet

This document is not complete. Several opcodes remain to be fully documented. This includes:

- Opcode to wait for “NN” milliseconds. Similar to the existing opcode where 10-milliseconds units are used.
- Several new opcodes that can be used to implement a PC based debugger (breakpoint opcodes, single multi-step control, etc).