Program Structure
declarations

Declarations
variable_declaration
task_declaration
function_declaration
subroutine_declaration

Variable Declaration
int *variables_list*;

Variables List is comma separated list of any of
variable_name
variable_name=*constant_expression*
array_name[*constant_expression*]

Task Declaration
```
task task_name() {
  // task body
}
```

Function Declaration
```
void function_name(argument_list) {
  // function body
}
```

Subroutines
```
sub subroutine_name() {
  // subroutine body
}
```

Statements
variable_declaration
assignment
compound_statement
if (*condition*) *statement*
if (*condition*) *statement* else *statement*
while (*condition*) *statement*
until (*condition*) *statement*
do *statement* while (*condition*);
for (*statement*; *condition*; *statement*) *statement*
repeat (*expression*) *statement*
switch (*expression*) *statement*
acquire (*resources*) *statement*
acquire (*resources*) *statement* catch *statement*
monitor (*events*) *statement*
monitor (*events*) *statement* catch *statement*
function_name(*argument_list*);
subroutine_name();
start *task_name*;
stop *task_name*;
break;
continue;
return;
expression;
;

Acquire and Monitor Statements
```
acquire (resources) {
  // action while task has ownership
  // higher priority task can pre-empt
} catch {
  // action if acquire fails or ownership lost
  // while executing (catch is optional)
}
monitor (events) {
  // action while monitoring events
} catch {
  // action if event occurs while monitoring
  // (catch is optional)
}
```

Program Entry Point (Compulsory Task)
```
task main() {
  // task body
}
```

If Statements
```
if (expression) {
  // consequence if expression true
} else {
  // alternative if expression false
}
```

Switch Statements
```
switch (expression) {
  case constant_expression:
    // action if (expression == constant_expression)
    break;
  default:
    // default action if no cases match
    break;
}
```

Break, Continue, Return
```
break;     // break out of switch or loop
continue; // skip to end of current iteration of loop
return;    // explicit or early exit from function/subroutine
```

Function Argument Types

Type	Meaning	Restriction	Example
int	pass by value	none	void foo(int x)
const int	pass by value	constants only	void foo(const int x)
int&	pass by reference	variables only	void foo(int& x)
const int&	pass by reference	no modification	void foo(const int& x)

Expressions - Any or combinations of
numeric_constants
variables
operators

Assignment
variable assignment_operator expression;

Examples
```
x = 1;
y += 2;
z *= (x + y);
a[3] -= a[4];
```

Assignment Operators

Operator	%
=	assign expression
+=	add expression
-=	subtract expression
*=	multiply by expression
/=	divide by expression
&=	bitwise AND with expression
|=	bitwise OR with expression
||=	assign absolute value of expression
+-=	assign sign of expression (-1, 0 or 1)

Other Statements
```
function_name(argument_list);    // inline invocation
subroutine_name();    // subroutine call
```

Preprocessor
```
#include "filename"
#define macro_name macro_text
#define macro_name(identifier) macro_text
#if condition    // constant expression inc. defined()
```

#undef	#endif
#ifdef *macro_name*	#pragma noinit
#ifndef *macro_name*	#pragma init *function*
#else	#pragma reserve *start*
#elif *condition*	#pragma reserve *start end*

Numeric Constants
decimal e.g. 1234
hexadecimal e.g. 0xABC

Operator Precedence

Operator	Description	Assoc	Restriction
abs()	absolute value	n/a	
sign()	sign of	n/a	
++	increment	left	variables only
--	decrement	left	variables only
-	unary minus	right	
~	bitwise negation	right	constants only
*	multiply	left	
/	divide	left	
%	modulo	left	
+	add	left	
-	subtract	left	
<<	left shift	left	constants only
>>	right shift	left	constants only
&	bitwise AND	left	
^	bitwise XOR	left	constants only
|	bitwise OR	left	
&&	boolean AND	left	constants only
||	boolean OR	left	constants only

Conditions

Condition	True if
true	always
false	never
expression	expression non-zero
expr1 == expr2	expr1 equal to expr2
expr1 != expr2	expr1 not equal to expr2
expr1 < expr2	expr1 less than expr2
expr1 <= expr2	expr1 less than or equal to expr2
expr1 > expr2	expr1 greater than expr2
expr1 >= expr2	expr1 greater than or equal to expr2
!condition	condition is false
cond1 && cond2	both cond1 and cond2 true
cond1 || cond2	either cond1 or cond2 true

Notes
1. These additional keywords are reserved, __event_src, __sensor, __type, asm.
2. RCX2 supports a maximum of 10 tasks, 8 subroutines, 32 global variable locations, 16 local variable locations.
3. Integers and array elements are 16 bit signed integers.
4. Constants are evaluated using 32 bit signed arithmetic before conversion to 16 bit signed constants.
5. NQC functions are always expanded to inline code. Subroutines cannot be nested.
6. Arrays cannot be used as arguments, only elements. Elements cannot use ++ or -- operators or any assignment other than =. Elements cannot be initialized.
7. const int& arguments cannot be modified by the called function but can pass anything (constants, variables, sensors, etc) and are read every time an expression is evaluated unlike int where the expression is evaluated only when the function is called.

Key - Document description - *user supplied syntax* - NQC language - comments in NQC syntax - user supplied examples - API definitions - *user supplied API arguments*

ActiveEvents(*task*)
AddToDatalog(*expression*)
BatteryLevel()
CalibrateEvent(*event_number, low, upper, hyst*)
ClearAllEvents()
ClearCounter(*counter_number*)
ClearEvent(*event_number*)
ClearMessage()
ClearSensor(*sensor*)
ClearSound()
ClearTimer(*timer_number*)
ClickCounter(*event_number*)
ClickTime(*event_number*)
Counter(*counter_number*)
CreateDatalog(*datalog_size*)
DecCounter(*counter_number*)
Event(*events*)
EventState(*event_number*)
FastTimer(*timer_number*)
FirmwareVersion()
Float(*outputs*)
Fwd(*outputs*)
GlobalOutputStatus(*output_number*)
Hysteresis(*event_number*)
IncCounter(*counter_number*)
LowerLimit(*event_number*)
MuteSound()
Off(*outputs*)
On(*outputs*)
OnFor(*outputs, time*)
OnFwd(*outputs*)
OnRev(*outputs*)
OutputStatus(*output_number*)
PlaySound(*sound*)
PlayTone(*frequency, duration*)
Program()
Random(*random_limit*)
Rev(*outputs*)
SetDisplay(*display_mode*)
SelectProgram(*program_number*)
SendMessage(*message*)
SendSerial(*buffer_index, byte_count*)
SensorMode(*sensor_number*)
SensorType(*sensor_number*)
SensorValue(*sensor_number*)
SensorValueBool(*sensor_number*)
SensorValueRaw(*sensor_number*)
SerialData(*buffer_index*)
SetClickTime(*event_number, time*)
SetClickCounter(*event_number, expression*)
SetDirection(*outputs, output_direction*)
SetEvent(*event_number, source, event_type*)
SetGlobalDirection(*outputs, output_direction*)
SetGlobalOutput(*outputs, output_mode*)
SetHysteresis(*event_number, expression*)
SetLowerLimit(*event_number, expression*)
SetMaxPower(*outputs, output_power*)
SetOutput(*outputs, output_mode*)
SetPower(*outputs, output_power*)
SetPriority(*priority*)
SetRandomSeed(*expression*)
SetSensor(*sensor, sensor_configuration*)
SetSensorMode(*sensor, sensor_mode*)
SetSensorType(*sensor, sensor_type*)
SetSerialCom(*serial_settings*)
SetSerialData(*buffer_index, expression*)
SetSerialPacket(*packet_settings*)
SetSleepTime(*minutes*)
SetTimer(*timer_number, expression*)
SetTxPower(*tx_power*)
SetUpperLimit(*event_number, expression*)
SetUserDisplay(*value, precision*)
SetWatch(*hours, minutes*)
SleepNow()
StopAllTasks()
Timer(*timer_number*)
Toggle(*outputs*)
UnmuteSound()
UploadDatalog(*datalog_index, count*)
UpperLimit(*event_number*)
Wait(*time*)
Watch()

display_mode	
DISPLAY_WATCH	(default)
DISPLAY_SENSOR_1	
DISPLAY_SENSOR_2	
DISPLAY_SENSOR_3	
DISPLAY_OUT_A	
DISPLAY_OUT_B	
DISPLAY_OUT_C	
DISPLAY_USER	

event_type	restrictions
EVENT_TYPE_PRESSED	sensors
EVENT_TYPE_RELEASED	sensors
EVENT_TYPE_PULSE	sensors
EVENT_TYPE_EDGE	sensors
EVENT_TYPE_FASTCHANGE	sensors
EVENT_TYPE_LOW	any
EVENT_TYPE_NORMAL	any
EVENT_TYPE_HIGH	any
EVENT_TYPE_CLICK	any
EVENT_TYPE_DOUBLECLICK	any
EVENT_TYPE_MESSAGE	messages

outputs
OUT_A
OUT_B
OUT_C

output_direction	
OUT_FWD	
OUT_REV	
OUT_TOGGLE	
OUT_FLIP	(SetGlobalDirection)

output_mode	brake
OUT_OFF	on
OUT_ON	n/a
OUT_FLOAT	off

output_power
OUT_LOW
OUT_HALF
OUT_FULL

packet_settings
SERIAL_PACKET_DEFAULT
SERIAL_PACKET_PREAMBLE
SERIAL_PACKET_NEGATED
SERIAL_PACKET_CHECKSUM
SERIAL_PACKET_RCX

resources
ACQUIRE_OUT_A
ACQUIRE_OUT_B
ACQUIRE_OUT_C
ACQUIRE_SOUND
ACQUIRE_USER_1
ACQUIRE_USER_2
ACQUIRE_USER_3
ACQUIRE_USER_4

sensor
SENSOR_1
SENSOR_2
SENSOR_3

sensor_configuration	type/mode
SENSOR_TOUCH	touch/bool
SENSOR_LIGHT	light/percent
SENSOR_ROTATION	rotation/rotation
SENSOR_CELCIUS	temperature/celcius
SENSOR_FAHRENHEIT	temperature/fahrenheit
SENSOR_PULSE	touch/pulse
SENSOR_EDGE	touch/edge

sensor_mode	reading
SENSOR_MODE_RAW	0 to 1023
SENSOR_MODE_BOOL	0 or 1
SENSOR_MODE_EDGE	counts edges
SENSOR_MODE_PULSE	counts pulses
SENSOR_MODE_PERCENT	0 to 100
SENSOR_MODE_FAHRENHEIT	degrees F
SENSOR_MODE_CELCIUS	degrees C
SENSOR_MODE_ROTATION	16 per rotation

sensor_type
SENSOR_TYPE_NONE
SENSOR_TYPE_TOUCH
SENSOR_TYPE_TEMPERATURE
SENSOR_TYPE_LIGHT
SENSOR_TYPE_ROTATION

serial_settings	default
SERIAL_COMM_DEFAULT	
SERIAL_COMM_4800	2400 Baud
SERIAL_COMM_DUTY25	50% duty cycle
SERIAL_COMM_76KHZ	38kHz

sound
SOUND_CLICK
SOUND_DOUBLE_BEEP
SOUND_DOWN
SOUND_UP
SOUND_LOW_BEEP
SOUND_FAST_UP

tx_power
TX_POWER_LO
TX_POWER_HI

arguments	range	restrictions
buffer_index	0 to 15	constants only
byte_count	1 to 15	constants only
counter_number	0 to 2	constants only
datalog_size	0 to ?	constants only
event_number	0 to 15	constants only
message	0 to 255	
output_number	0 to 2	constants only
power	0 to 7	
precision	0 to 4?	constants only
priority	0 to 255	constants only
program_number	0 to 4	constants only
random_limit	0 to desired max.	constants only
sensor_number	0 to 3	constants only
task	0 to 9?	constants only
timer_number	0 to 3	constants only

arguments	units	restrictions
duration	10ms	constants only
frequency	Hz	
time	10ms	

event states	description
0	low (between min and lower_limit)
1	normal (between lower and upper limits)
2	high (between upper_limit and max)
3	undefined
4	start calibrating
5	calibration in progress (takes approx. 50ms)

Notes

1. The SENSOR_1, etc. macros are not equivalent to *sensor_number* 0, 1 or 2. The macros can be used in expressions to return a sensor reading or as arguments in functions that expect *sensor*.

2. *outputs* can be combined. E.g. OUT_A + OUT_B;

3. User display shows dynamic contents of a timer, counter, sensor or variable in a global location. Precision is decimal places to right.

4. RCX2 supports a mximum of 4 timers and 3 counters.

5. Timers increment at 100ms intervals (10ms for fast timers) and count from 0 to 32767.

6. Counters overlap with global locations so e.g.
#pragma reserve 1 // reserve counter 1

7. A *slope* can be added to *sensor_mode* when *sensor_mode* is expected.

8. EVENT_TYPE_FASTCHANGE should only be used when *slope* parameter given. Triggers when change exceeds *slope* within 3ms.

9. For functions that expect *events* convert *event_number* using EVENT_MASK(*event_number*) macro.

Key - Document description - *user supplied syntax* - NQC language - comments in NQC syntax - user supplied examples - API definitions - *user supplied API arguments*