Syntax Description of nqBASIC Language

An nqBASIC program has the following structure:

constant, type and variable definitions first (CONST, TYPE and DIM),
functions and tasks next (SUB and TASK)
main-function at end (MAIN).

These definitions may be spread over several source-files (".nqb" files, but should follow the order as listed above). I.e. after function-definitions NO variable definitions may follow.

All code is case-INsensitive

Comment started by " **//** ". The rest of the line is skipped by the compiler.
Comment between "**/*….*/**". Several of these comment sections can be on a single line or the comment section can include several lines.

Constant number formats recognized are:

decimal: e.g. 255,
hex (preceded by "0x"). e.g. 0xFF
binary (preceded by "0b"). e.g. 0b11111111
character constant (enclosed in " ' "). Single characters or special characters preceded by "\". See table below.
string constant (enclosed in " " "). e.g. "Press a key to continue…"

Special Characters		Ascii-value (decimal)
\t	Horizontal Tab	9
\n	Line Feed	10
\r	Carriage Return	13
\v	Vertical Tab	11
\\	Back Slash	92
\"	Double quote	34
\f	Form Feed	12
\b	BackSpace	8
\a	Alert	7
other without "\"	any readable char. e.g. 'A'	e.g. 'A' = ascii 65

The constant identifier is defined with a CONST statement:

Const <ident> **=** <value>

Example: Const PIN5 = 5

NOTES: constant identifiers cannot start with a numeric digit.

<value> may be another constant definition, but NOT an expression.

Dim <ident> **as new** <type or class-name> [(<pins>)]
<type> can be: BYTE, WORD
<class> see chapter 6 and appendix A for classes. Classes are followed by "(…)" and may require PIN-numbers. (Pin-numbers are 0-based, where pin-0 is the first "CONTROL PIN" line in the trg-file in your ".ncp" project-file). NOTE: trg-file must support pin-protocol for that object. (I.e. ASYNC_RX for RX-pin of SCI).

Note all variables are globals (except for function parameters).

Dim <ident>**[**<constant size>**] as new** [BYTE | WORD]
As "Dim" but generates array with <constant size> elements. Elements can be accessed by: **<ident>[<index>]**
The <index> can be a constant, an (indexed array-) variable or an expression. If you use just the identifier <ident> without "[<index>]", the compiler will interpret this as if index "0" was specified. Note: the maximum array-size is 0xFFF right now.

Note all arrays are globals.

Type <type-name> **User defined type**
1{byte|word <field-name> [<size>]}n //1+ byte, word and array fields
End Type
Use DIM to create variable of type <type-name>. Fields can be accessed by using the variable-name, a dot and the field-name.

Example:

```
Type MyType
    byte field1
    word field2
    byte field3[5]          //This field contains array of 5 bytes
    word field4[8]          //This field contains array of 8 words
End type

Dim MyVar as new MyType
MyVar.field1 = 3
MyVar.field3[3]= 8
```

Expressions: operators: +,-,*,/,%,==,!=,AND,OR,&,|,^,>,<,>=,<=,>>,<<

NOTE: left-to-right evaluation, UNLESS "**(**" and "**)**" used to indicate priority. Note that indexing an array with a non-constant value is treated as an expression. Expressions can be used in:

assignments
conditions (IF, WHILE, UNTIL,SELECT)
parameters of function-calls only for IN-parameters (since OUT-parameters require variables in which results are to be stored)

Operators which do an equality compare (like ==, !=, <, <=, >, >=) result in 0 (false) or 1 (true). Compile-time calculations support is included. e.g. “index < (10 – 1)” will result in “index < 9” being processed by the compiler. Note that a Const-definition cannot have compile-time expressions!

Assignment: <ident> = <expression> [;]   ";" is optional.

If (<expression>) **then** <BODY> **else** <BODY> **end if**

Note: currently *elseif* is not supported. However nesting of *if-then-else* gives the same result. Only difference is that you would get multiple sets of *end if* following the outer *if-then*.

Example:
```
If (x > 10)
        S.SER_Put_string (“x-value was >10“)
else
        S.SER_Put_string (“x-value was <=10“)
End if
```

While (<expression>) <BODY> **end while**
NOTE: while (1) means "loop forever". See constant FOREVER in stdlib.nqb.

Do <BODY> until (<expression>)

For-next-step: for <variable> = <start> to <end> [step [-]<constant>]
```
                                   <BODY>            //Empty body NOT allowed!
```
next

For-next is a short-form of a while-loop:
```
          <variable> = <start>
          while (<variable>  <   <end>)                //”>  <end>” when step is negative.
                    <BODY>
                   <variable> = <variable> + step    //Note if step negative then decreases
          end while
```

Step is optional and defaults to 1. If <start> greater then <end> then “step” MUST be included to indicate negative stepping of iterator <variable>.
<start> and <end> can be expressions. Loop-condition “<variable> < <end>” is evaluated every iteration. This is also true, when <end> contains an expression, which includes other variables. (Expression will be evaluated every iteration). If the <end> expression does not change, it is wise to assign the expression to a variable before the for-loop and use the variable as <end>. This speeds up the loop.
See example “fordip40” for several for-next-step constructions!

Select <expression> **case** <constant>: <BODY> **break default**: <BODY> **End Select**
Duplicate case labels can be combined (leading to the same code-block). Multiple case-constant-values are not reported right now. Only the block of the first case-value will be executed. Case-labels do not fall through. The **break** statement MUST be included (not optional).

Example:
```
select X
        case 3:
        case ‘A’:
              S.SER_Put_string (“X equals 3 or 65”)
              break            //break is NOT optional!
       default:
              S.SER_Put_string (“X was something else…”)
              break
end select
```

Sub <name> (<in|out> <byte|word> <arg-name>) <function-BODY> **end sub**

Example:
```
Sub MyFunction (in byte x)
         S.SER_Put_string (“X-value was: “)
         S.SER_Put_decimal (x)
End sub
```
Arguments can be of type BYTE or WORD. When calling a function, IN-parameters can be either a constant, variable or expression. OUT-parameters can only take variables as parameter.
Task <task-body> **end task**
The <task-body> is similar to <function-body>. It may contain any number of statements. However, task-bodies can also include calls to the WAIT-function. After the main-function has been executed, each task will execute –in the order as listed in the source-files- until a WAIT-function is encountered. A task can either finish, when it reaches “end task”, after which it will never execute again OR is paused when the WAIT-function is called. NOTE: Currently max. 10 tasks are supported.

Typically a task looks like this:
```
               Task
                        While (FOREVER)
                                       //do something…
                                      Wait (<an event>)    //allows other tasks to run
                        End while
               End task
```

Wait (<event or expression>)

The WAIT-function is a special nqBASIC function, which makes multi-tasking possible. Compiled nqBASIC code includes a lean event-driven multi-tasking engine, which can execute multiple tasks. When a task calls the WAIT-function, the nqBASIC scheduler will pause the current task and check if other tasks are ready to run. This happens when the event passed to the WAIT-function has occurred OR when an expression passed to the WAIT-function evaluates to TRUE (non-zero). If a task loops, without calling the WAIT-function, other tasks will NEVER run. (The nqBASIC scheduler is non-preemptive). If you are using multiple tasks, while one of the tasks does not wait for a particular event, you will have to call the WAIT-function with EVENT_IDLE as parameter regularly, to prevent other tasks being deprived of execution.

Currently the following parameters –defined in stdlib.nqb- can be passed to the WAIT-function:

Events	**Description**

EVENT_IRQ	IRQ pin (constant IRQ or PIN27) generated interrupt
EVENT_XIRQ	IRQ pin (constant XIRQ or PIN28) generated interrupt
EVENT_RTI	Realtime Timer overflow/expiration (See RTI-object).
EVENT_SCI	Serial character received by SCI UART (See SCI-object)
EVENT_SPI	SPI interrupt
EVENT_TIM	Main timer overflow
EVENT_IOC0	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_IOC1	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_IOC2	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_IOC3	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_IOC4	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_IOC5	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_IOC6	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_IOC7	Pin-timer overflow/expiration (See TIMIO-object).
EVENT_CAN	CAN frame received (See CAN-object)
EVENT_IDLE	Resume when no other tasks are ready for execution
<EXPRESSION>	When the expression evaluates to a non-zero value, the task will resume execution. This way tasks can communicate and/or wake each other up when an event occurs. e.g. one task can assign a value to a global variable, which is used in an expression that is passed to a WAIT-function in another task.

Main <function-body> **end main**
The main-function of an nqBASIC program is the first part of your program that will execute. Use it to setup global objects and data structures used elsewhere in the program (tasks and functions). Interrupts are not enabled when the MAIN-function is called. Do NOT use an endless loop in the main-function if you are using multiple tasks since, in that case, these tasks will never be able to execute. If you ARE NOT using tasks, while you use objects that need interrupts (like SCI or CAN), then you probably want to enable interrupts by calling System.INTS_On().

Objects (see API Reference) give access to internal peripherals, I/O pins, and software libraries. Objects can be instantiated by using a DIM statement with a variable (e.g. when I/O pins are used), but can also have class-functions, which can be called regardless of an instantiated object, by using the class-name.

Class functions are called with: "<class-name>." in front of the function-name.

Example:

```
System.Delay (100)    ‘100microsecond delay
```

Object functions are called with: "<object-variable name>." in front of the function-name. (The dim statement must have been used to create the object with the variable name).

Example:

```
Const PIN1 = 1
Const HIGH = 1
Dim D as new DIO (PIN1)
Main
    D.PIN_Out (PIN1, HIGH)
End Main
```