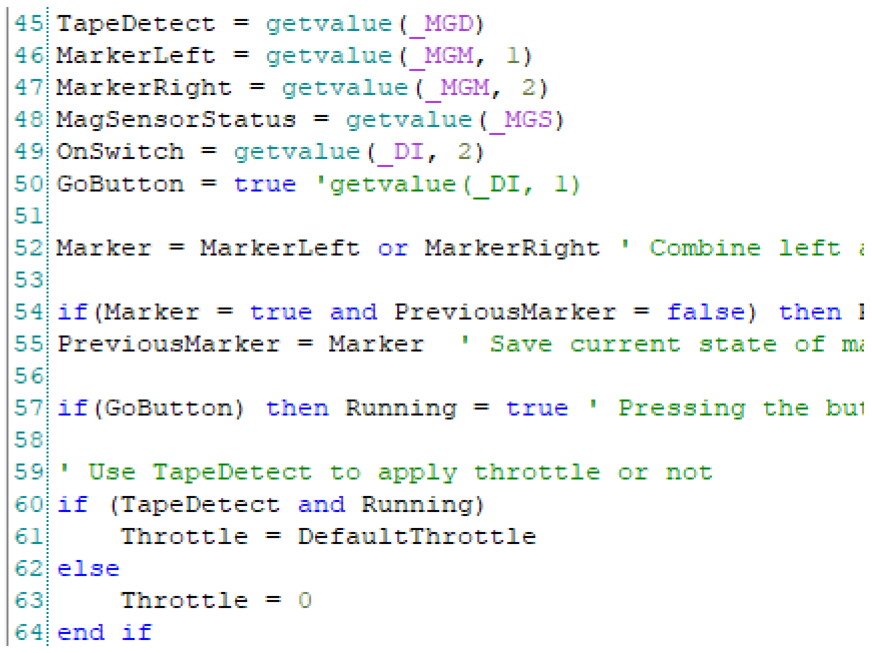
RoboteQ

# MicroBasic Scripting Language



```
TapeDetect = getvalue(_MGD)
MarkerLeft = getvalue(_MGM, 1)
MarkerRight = getvalue(_MGM, 2)
MagSensorStatus = getvalue(_MGS)
OnSwitch = getvalue(_DI, 2)
GoButton = true 'getvalue(_DI, 1)

Marker = MarkerLeft or MarkerRight ' Combine left

if(Marker = true and PreviousMarker = false) then
PreviousMarker = Marker  ' Save current state of m

if(GoButton) then Running = true ' Pressing the bu

' Use TapeDetect to apply throttle or not
if (TapeDetect and Running)
    Throttle = DefaultThrottle
else
    Throttle = 0
end if
```

## User & Reference Manual

V2.0, July 8, 2019

visit www.roboteq.com to download the latest revision of this manual

## Revision History

| Date | Version | Changes |
|---|---|---|
| July 8, 2019 | 2.0 | Extracted from main User Manual |

# Introduction

## Refer to the Datasheet for Hardware-Specific Issues

This manual is the companion to your controller's datasheet. All information that is specific to a particular controller model is found in the datasheet. These include:

- Number and types of I/O
- Connectors pin-out
- Wiring diagrams
- Maximum voltage and operating voltage
- Thermal and environmental specifications
- Mechanical drawings and characteristics
- Available storage for scripting
- Battery or/and Motor Amps sensing
- Storage size of user variables to Flash or Battery-backed RAM

## User Manual Structure and Use

The user manual discusses issues that are common to all controllers inside a given product family. Except for a few exceptions, the information contained in the manual does not repeat the data that is provided in the datasheets.

## MicroBasic Scripting

This section describes the MicroBasic scripting language that is built into the controller. It describes the features and capabilities of the language and how to write custom scripts. A Language Reference is provided.

# MicroBasic Scripting

One of the Roboteq products' most powerful and innovative features is their ability for the user to write programs that are permanently saved into, and run from the device's Flash Memory. This capability is the equivalent, for example, of combining the motor controller functionality and this of a PLC or Single Board Computer directly into the controller. Script can be simple or elaborate, and can be used for various purposes:

- **Complex sequences:**
  MicroBasic Scripts can be written to chain motion sequences based on the status of analog/digital inputs, motor position, or other measured parameters. For example, motors can be made to move to different count values based on the status of pushbuttons and the reaching of switches on the path.
- **Adapt parameters at runtime**
  MicroBasic Scripts can read and write most of the controller's configuration settings at runtime. For example, the Amps limit can be made to change during operation based on the measured heatsink temperature.
- **Create new functions**
  Scripting can be used for adding functions or operating modes that may be needed for a given application. For example, a script can compute the motor power by multiplying the measured Amps by the measured battery Voltage, and regularly send the result via the serial port for Telemetry purposes.
- **Autonomous operation**
  MicroBasic Scripts can be written to perform fully autonomous operations. For example the complete functionality of a line following robot can easily be written and fitted into the controller.

## Script Structure and Possibilities

Scripts are written in a Basic-Like computer language. Because of its literal syntax that is very close to the every-day written English, this language is very easy to learn and simple scripts can be written in minutes. The MicroBasic scripting language also includes support for structured programming, allowing fairly sophisticated programs to be written. Several shortcuts borrowed from the C-language (++, +=, …) are also included in the scripting language and may be optionally used to write shorter programs.

The complete details on the language can be found in the MicroBasic Language Reference on page 15.

## Source Program and Bytecodes

Programs written in this Basic-like language are interpreted into an intermediate string of Bytecode instructions that are then downloaded and executed in the controller. This two-step structure ensures that only useful information is stored in the controller and results in significantly higher performance execution over systems that interpret Basic code directly. This structure is for the most part entirely invisible to the programmer as the source editing is the only thing that is visible on the PC, and the translation is done in the background just prior to downloading to the controller.

Depending on the product, programs can be from 8192 to 32768 Bytecodes long. This translates to approximately 1500 to 6000 lines of MicroBasic source.

## Variables Types and Storage

Scripts can store signed 32-bit integer variables and Boolean variable. Integer variables can handle values up to +/– 2,147,483,647. Boolean variables only contain a True or False state. The language also supports single dimensional arrays of integers and Boolean variables.

In total, up to 1024 or 4096 (depending on the product) Integer variables and up to 1024 Boolean variables can be stored in the controller. An array of n variables will take the storage space of n variables.

The language only works with Integer or Boolean values. It is not possible to store or manipulate decimal values. This constraint results in more efficient memory usage and faster script execution. This constraint is usually not a limitation as it is generally sufficient to work with smaller units (e.g. millivolts instead of Volts, or milliamps instead of Amps) to achieve the same precision as when working with decimals.

The language does not support String variables and does not have string manipulation functions. Basic string support is provided for the Print command.

## Variable content after Reset

All integer variables are reset to 0 and all Boolean variables are reset to False after the controller is powered up or reset. When using a variable for the first time in a script, its value can be considered as 0 without the need to initialize it. Integer and Boolean variables are also reset whenever a new script is loaded.

When pausing and resuming a script, all variables keep the values they had at the time the script was paused.

## Controller Hardware Read and Write Functions

The MicroBasic scripting language includes special functions for reading and writing configuration parameters. Most configuration parameters that can be read and changed using the Configuration Tab in the Roborun PC utility or using the Configuration serial commands, can be read and changed from within a script. The GetConfig and SetConfig functions are used for this purpose.

The GetValue function is available for reading real-time operating parameters such as Analog/Digital input status, Amps, Speed or Temperature.

The SetCommand function is used to send motor commands or to activate the Digital Outputs. Practically all controller parameters can be access using these 4 commands, typically by adding the command name as defined in Section "Serial (RS232/RS485/TCP/USB) Operation" found in Roboteq Controllers User Manual v2.0 preceded with the "_" character. For example, reading the Amps limit configuration for channel 1 is done using getvalue(_ALIM, 1).

See the MicroBasic Language Reference on page 15 for details on these functions and how to use them.

## Timers and Wait

The language supports four 32-bit Timer registers. Timers are counters that can be loaded with a value using a script command. The timers are then counting down every millisecond independently of the script execution status. Functions are included in the language to load a timer, read its current count value, pause/resume count, and check if it has reached 0. Timers are very useful for implementing time-based motion sequences.

A wait function is implemented for suspending script execution for a set amount of time. When such an instruction is encountered, script execution immediately stops and no more time is allocated to script execution until the specified amounts of milliseconds have elapsed. Script execution resumes at the instruction that follows the wait.

## Execution Time Slot and Execution Speed

MicroBasic scripts are executed in the free time that is available every 1ms, after the controller has completed all its motion control processing. The available time can therefore vary depending on the functions that are enabled or disabled in the controller configuration. For example more time is available for scripting if the controller is handling a single motor in open loop than if two motors are operated in closed loop with encoders. At the end of the allocated time, the script execution is suspended, motor control functions are performed, and scripts resumed. An execution speed averaging 50,000 lines of MicroBasic code, or higher, per second can be expected in most cases.

## Protections

No protection against user error is performed at execution time. For example, writing or reading in an array variable with an index value that is beyond the 1024 or 4096 variables available in the controller may cause malfunction or system crash. Nesting more than 64 levels of subroutines (i.e. subroutines called from subroutines, …) will also cause potential problems. It is up to the programmer to carefully check the script's behavior in all conditions.

## Print Command Restrictions

A print function is available in the language for outputting script results onto the serial or USB port. Since script execution is very fast, it is easy to send more data to the serial or USB port than can actually be output physically by these ports. The print command is therefore limited to 32 characters per 1ms time slot. Printing longer strings will force a 1ms pause to be inserted in the program execution every 32 characters and/or loss of characters.

## Editing, Building, Simulating and Executing Scripts

### Editing Scripts

An editor is available for scripting in the RoborunPlus PC utility. See Scripting Tab on page 368 (Roborun scripting) for details on how to launch and operate the editor.

The edit window resembles this of a typical IDE editor with, most noticeably, changes in the fonts and colors depending on the type of entry that is recognized as it is entered. This capability makes code easier to read and provides a first level of error checking.

Code is entered as free-form text with no restriction in term of length, indents use, or other.

### Building Scripts

Building is the process of converting the Basic source code in the intermediate Bytecode language that is understood by the controller. This step is nearly instantaneous and normally transparent to the user, unless errors are detected in the program.

Build is called automatically when clicking on the "Download to Device" or "Simulate" buttons.

Building can be called independently by clicking on the "Build" button. This step is normally not necessary but it can be useful in order to compare the memory usage efficiency of one script compared to another.

### Simulating Scripts

Scripts can be ran directly on the PC in simulation mode. Clicking on the Simulate button will cause the script to be built and launch a simulator in which the code is executed. This feature is useful for writing, testing and debugging scripts. The simulator works exactly the same way as the controller with the following exceptions.

- Execution speed is different.
- Controller configurations and operating parameters are not accessible from the simulator
- Controller commands cannot be sent from the simulator
- The four Timers operate differently in the simulator
- RoboCAN commands and queries have no effect

In the simulator, any attempt to read a Controller configuration (example Amps limit) or a Controller Runtime parameter (e.g. Volts, Temperature) will cause a prompt to be displayed for the user to enter a value. Entering no value and hitting Enter, will cause the same value that was entered last for the same parameter to be used. If this is the first time the user is prompted for a given parameter, 0 will be entered if hitting Enter with no data.

When a function in the simulator attempts to write a configuration or a command, then the console displays the parameter name and parameter value in the console.

Script execution in the simulator starts immediately after clicking on the Simulate button and the console window opens.

Simulated scripts are stopped when closing the simulator console.

## Downloading MicroBasic Scripts to the controller

The Download to Device button will cause the MicroBasic script to be built and then transferred into the controller's flash memory where it will remain permanently unless overwritten by a new script.

The download process requires no other particular attention. There is no warning that a script may already be present in Flash. A progress bar will appear for the duration of the transfer which can be from a fraction of a second to a few seconds. When the download is completed successfully, no message is displayed, and control is returned to the editor A downloaded script cannot be read out.

An error message will appear only if the controller is not ready to receive or if an error occurred during the download phase.

Downloading a new script while a script is already running will cause the running script to stop execution. All variables will also be cleared when a new script is downloaded. When using multiple controllers over a CAN network with the RoboCAN protocol, it is possible to download the script into any node. Use the Download to Device button from the Scripting tab of the PC Utility.

In networked systems using the RoboCAN protocol, scripts can be loaded in any active node by using the Download to Remote button in the PC utility.

## Saving and Loading Scripts in Hex Format

Compiled scripts can be saved as a .hex format file from the PC Utility. The bytecodes can then be loaded into the controller using the "Update Script" button on the Console tab. Using this technique is a good way of keeping the source code secret and/or safe while allowing field updates.

The bytecodes in the .hex file can also be loaded in the controller by any microcomputer using the following sequence:

Send the string:

%sld 321654987

the controller will reply with

HLD

to indicate it is waiting for data

then send the hex file, one line at a time. At the end of each line received, the controller will send a +

Beware that if no data is received for more than 1s, the controller will exit the HLD mode.

## Executing MicroBasic Scripts

Once stored in the Controller's Flash memory, scripts can be executed either "Manually" or automatically every time the controller is started.

Manual launch is done by sending commands via the Serial or USB port. When connected to the PC running the PC utility, the launch command can be entered from the Console tab. The commands for running as stopping scripts are:

- **!r** : Start or Resume Script
- **!r 0**: Pause Script execution
- **!r 1**: Resume Script from pause point. All integer and Boolean variables have values they had at the time the script was paused.
- **!r 2**: Restarts Script from start. Set all integer variables to 0, sets all Boolean variables to False. Clears and stops the 4 timers.

On CAN networks running the RoboCAN protocol, a script on a remote node can be launched or stopped by using the above commands with the RoboCAN prefix. For example

- **@06!r** : Start or Resume Script on device at RoboCAN node 6

If the controller is connected to a microcomputer, it is best to have the microcomputer start script execution by sending the !r command via the serial port or USB.

Scripts can be launched automatically after controller power up or after reset by setting the Auto Script configuration to Enable in the controller configuration memory. When enabled, if a script is detected in Flash memory after reset, script execution will be enabled and the script will run as when the !r command is manually entered. Once running, scripts can be paused and resumed using the commands above.

## Important Warning

**Prior to set a script to run automatically at start-up, make sure that your script will not include errors that can make the controller processor crash. Once set to automatically start, a script will always start running shortly after power up. If a script contains code that causes system crash, the controller will never reach a state where it will be possible to communicate with it to stop the script and/or load a new one. If this ever happens, the only possible recovery is to connect the controller to a PC via the serial port and run a terminal emulation software. Immediately after receiving the Firmware ID, type and send !r 0 to stop the script before it is actually launched. Alternatively, you may reload the controller's firmware.**

## Debugging Microbasic Scripts

While running a script with the source code visible in the Scripting tab, it is possible to view the state of all variable in real time. Click on the "Inspect Variable" buttons and over the variable in the source code. The variable value will appear near the mouse. To see changes to the variable, move the mouse away and then back on the variable.

Using print statements in questionable parts of the code is also a very effective debug tool. Run script from the console in order to be able to view the script output.

## Script Command Priorities

When sending a Motor or Digital Output command from the script, it will be interpreted by the controller the same way as a serial command (RS232, RS485, TCP or USB). The watchdog timer will trigger in if no commands are sent from the script within the watchdog timeout.

Script commands have the highest priority from all other interfaces.

## MicroBasic Scripting Techniques

Writing scripts for the Roboteq controllers is similar to writing programs for any other computer. Scripts can be called to run once and terminate when done. Alternatively, scripts can be written so that they run continuously.

### Single Execution Scripts

These scripts are programs that perform a number of functions and eventually terminate. These kind of scripts can be summarized in the flow chart below. The amount of processing can be simple or very complex but the script has a clear begin and end.

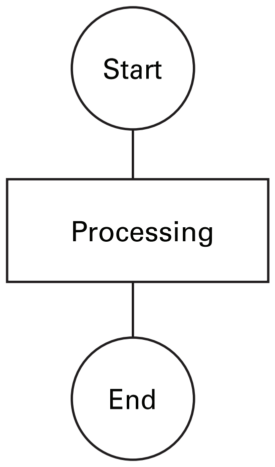


FIGURE 1-1. Single execution scripts

### Continuous Scripts

More often, scripts will be active permanently, reacting differently based on the status of analog/ digital inputs, or operating parameters (Amps, Volts, Temperature, Speed, Count, …), and continuously updating the motor power and/or digital outputs. These scripts have a beginning but no end as they continuously loop back to the top. A typical loop construction is shown in the flow chart below.

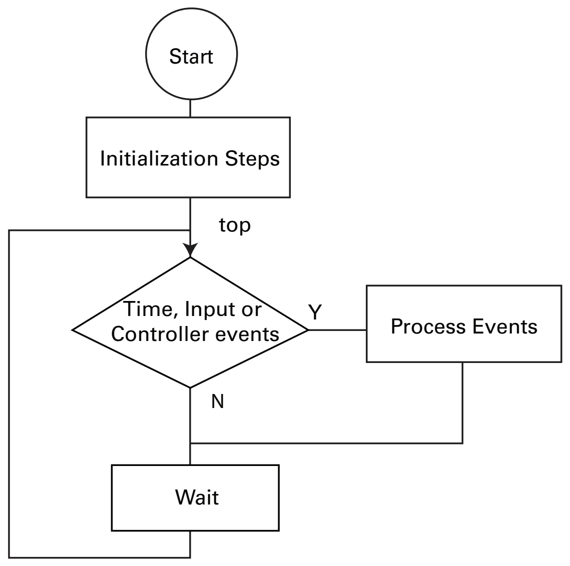


FIGURE 1-2. Continuous execution scripts

Often, some actions must be done only once when script starts running. This could be setting the controller in an initial configuration or computing constants that will then be used in the script's main execution loop.

The main element of a continuous script is the scanning of the input ports, timers, or controller operating parameters. If specific events are detected, then the script jumps to steps related to these events. Otherwise, no action is taken.

Prior to looping back to the top of the loop, it is highly recommended to insert a wait time. The wait period should be only as short as it needs to be in order to avoid using processing resources unnecessarily. For example, a script that monitors the battery and triggers an output when the battery is low does not need to run every millisecond. A wait time of 100ms would be adequate and keep the controller from allocating unnecessary time to script execution.

## Optimizing Scripts for Integer Math

Scripts only use integer values as variables and for all internal calculation. This leads to very fast execution and lower computing resource usage. However, it does also cause limitation. These can easily be overcome with the following techniques.

First, if precision is needed, work with smaller units. In this simple Ohm-law example, whereas 10V divided by 3A results in 3 Ohm, the same calculation using different units will give a higher precision result: 10000mV divided by 3A results in 3333 mOhm

Second, the order in which terms are evaluated in an expression can make a very big difference. For example (10 / 20) * 1000 will produce a result of 0 while (10 * 1000)/20 produces 5000. The two expressions are mathematically equivalent but not if numbers can only be integers.

## Script Examples

Several sample scripts are available from the download page on Roboteq's web site.

Below is an example of a script that continuously checks the heat sink temperature at both sides of the controller enclosure and lowers the amps limit to 50A when the average temperature exceeds 50oC. Amps limit is set at 100A when temperature is below 50o. Notice that as temperature is changing slowly, the loop update rate has been set at a relatively slow 100ms rate.

```
' This script regularity reads the current temperature at both sides
' of the heat sink and changes the Amps limit for both motors to 50A
' when the average temperature is above 50oC. Amps limit is set to
' 100A when temperature is below or equal to 50oC.
' Since temperature changes slowly, the script is repeated every 100ms

' This script is distributed "AS IS"; there is no maintenance
' and no warranty is made pertaining to its performance or applicability

top: ' Label marking the beginning of the script.

' Read the actual command value
Temperature1 = getvalue(_TEMP,1)
Temperature2 = getvalue(_TEMP,2)
TempAvg = (Temperature1 + Temperature2) / 2

' If command value is higher than 500 then configure
' acceleration and deceleration values for channel 1 to 200

if TempAvg > 50 then
        setconfig(_ALIM, 1, 500)
        setconfig(_ALIM, 2, 500)
else
' If command value is lower than or equal to 500 then configure
' acceleration and deceleration values for channel 1 to 5000
        setconfig(_ALIM, 1, 1000)
        setconfig(_ALIM, 2, 1000)
end if

' Pause the script for 100ms
wait(100)
' Repeat the script from the start
goto top
```

# MicroBasic Language Reference

## Introduction

The **Roboteq Micro Basic** is high level language that is used to generate programs that runs on Roboteq motor controllers. It uses syntax nearly like Basic syntax with some adjustments to speed program execution in the controller and make it easier to use.

## Comments

A comment is a piece of code that is excluded from the compilation process. A comment begins with a single-quote character. Comments can begin anywhere on a source line, and the end of the physical line ends the comment. The compiler ignores the characters between the beginning of the comment and the line terminator. Consequently, comments cannot extend across multiple lines.

```
'Comment goes here till the end of the line.
```

## Boolean

`True` and `False` are literals of the Boolean type that map to the true and false state, respectively.

## Numbers

Micro Basic supports only integer values ranged from -2,147,483,648 (0x80000000) to 2,147,483,647 (0x7FFFFFFF).

Numbers can be preceded with a sign (+ or -), and can be written in one of the following formats:

- **Decimal Representation**

  Number is represented in a set of decimal digits (0-9).

  ```
  120                 5622                504635
  ```

  Are all valid decimal numbers.

- **Hexadecimal Representation**

  Number is represented in a set of hexadecimal digits (0-9, A-F) preceded by 0x.

  ```
  0xA1                0x4C2               0xFFFF
  ```

  Are all valid hexadecimal numbers representing decimal values 161, 1218 and 65535 respectively.

- **Binary Representation**

  Number is represented in a set of binary digits (0-1) preceded by 0b.

  ```
  0b101               0b1110011           0b111001010
  ```

  Are all valid binary numbers representing decimal values 5, 115 and 458 respectively.

## Strings

Strings are any string of printable characters enclosed in a pair of quotation marks. Non printing characters may be represented by simple or hexadecimal escape sequence. Micro Basic only handles strings using the Print command. Strings cannot be stored in variable and no string handling instructions exist.

- **Simple Escape Sequence**

  The following escape sequences can be used to print non-visible or characters:

| Sequence | Description |
|---|---|
| \' | Single quote |
| \" | Double quote |
| \\ | |
| | Null |
| \a | Alert |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |

- **Hexadecimal Escape Sequence**
  Hexadecimal escape sequence is represented in a set of hexadecimal digits (0-9, A-F) preceded by \x in the string (such as \x10 for character with ASCII 16).

  Since a hexadecimal escape sequence can have a variable number of hex digits, the string literal "`\x123`" contains a single character with hex value 123. To create a string containing the character with hex value 12 followed by the character 3, one could write "`\x00123`".

So, to represent a string with the statement "Hello World!" followed by a new line, you may use the following syntax:

```
"Hello World!\n"
```

## Blocks and Labels

A group of executable statements is called a statement block. Execution of a statement block begins with the first statement in the block. Once a statement has been executed, the next statement in lexical order is executed, unless a statement transfers execution elsewhere.

A label is an identifier that identifies a particular position within the statement block that can be used as the target of a branch statement such as `GoTo`, `GoSub` or `Return`.

Label declaration statements must appear at the beginning of a line. Label declaration statements must always be followed by a colon (:) as the following:

```
Print_Label:
    Print("Hello World!")
```

Label name should start with alphabetical character and followed by zero or more alphanumeric characters or underscore. Label names cannot start with underscore. Labels names cannot match any of Micro Basic reserved words.

Label names are case insensitive that is `PrintLabel` is identical to `printLabel`.

The scope of a label extends whole the program. Labels cannot be declared more than once in the program.

## Variables

Micro Basic contains only two types of variable (`Integer` and `Boolean`) in addition to arrays of these types. Boolean and arrays must be declared before use, but `Integer` variables may not be declared unless you use the `Option Explicit` compiler directive.

```
Option Explicit
```

Variables can be declared using DIM keyword (see Dim (Variable Declaration) on page 20).

Variable name should start with alphabetical character and followed by zero or more alphanumeric characters or underscore. Variable names cannot start with underscore. Variable names cannot match any of Micro Basic reserved words.

Variable names are case insensitive, that is `VAR` is identical to `var`.

The scope of a variable extends whole the program. Variables cannot be declared more than once in the program.

## Arrays

Arrays is special variables that holds a set of values of the variable type. Arrays are declared using DIM command (see Dim (Variable Declaration) on page 20).

To access specific element in the array you can use the indexer [] (square brackets). Arrays indices are zero based, so index of 5 refer to the 6th element of the array.

```
arr[0] = 10  'Set the value of the first element in the array to 10.

a = arr[5]   'Store the 6th element of the array into variable a.
```

## Terminology

In the following sections we will introduce Micro Basic commands and how it is used, and here is the list of terminology used in the following sections:

- Micro Basic commands and functions will be marked in blue and cyan respectively.
- Anything enclosed in < > is mandatory and must be supplied.
- Anything enclosed in [ ] is optional, except for arrays where the square brackets is used as indexers.
- Anything enclosed in { } and separated by | characters are multi choice options.
- Any items followed by an ellipsis, ... , may be repeated any number of times.
- Any punctuation and symbols, except those above, are part of the structure and must be included.

| | |
|---|---|
| var | is any valid variable name including arrays. |
| arr | is any valid array name. |
| expression | is any expression returning a result. |
| condition | is any expression returning a boolean result. |

| | |
|---|---|
| stmt | is single Micro Basic statement. |
| block | is zero or more Micro Basic statements. |
| label | is any valid label name. |
| n | is a positive integer value. |
| str | is a valid string literal. |

## Keywords

A keyword is a word that has special meaning in a language construct. All keywords are reserved by the language and may not be used as variables or label names. Below is a list of all Micro Basic keywords:

| | | | | |
|---|---|---|---|---|
| #define | And | AndWhile | As | Boolean |
| Continue | Dim | Do | Else | ElseIf |
| End | Evaluate | Exit | Explicit | False |
| For | GoSub | GoTo | If | Integer |
| Loop | Mod | Next | Not | Option |
| Or | Print | Return | Step | Terminate |
| Then | To | ToBool | True | Until |
| While | XOr | | | |

## Operators

Micro Basic provides a large set of operators, which are symbols or keywords that specify which operations to perform in an expression. Micro Basic predefines the usual arithmetic and logical operators, as well as a variety of others as shown in the following table.

| **Category** | **Operators** | | | | | | |
|---|---|---|---|---|---|---|---|
| Arithmetic | `+` | `-` | `*` | `/` | `Mod` | | |
| Logical (boolean and bitwise) | `And` | `Or` | `XOr` | `Not` | `True` | `False` | |
| Increment, decrement | `++` | `--` | | | | | |
| Shift | `<<` | `>>` | | | | | |
| Relational | `=` | `<>` | `<` | `>` | `<=` | `>=` | |
| Assignment | `=` | `+=` | `-=` | `*=` | `/=` | `<<=` | `>>=` |
| Indexing | `[]` | | | | | | |

## Micro Basic Functions

The following is a set of Micro Basic functions

| | |
|---|---|
| Abs | Returns the absolute value of a given number. |
| Atan | Returns the angle whose arc tangent is the specified number. |
| Cos | Returns the cosine of the specified angle. |
| Sin | Returns the sine of the specified angle. |
| Sqrt | Returns the square root of a specified number. |

## Controller Configuration and Commands

The following is a set of device functions for interacting with the Controller:

| | |
|---|---|
| SetConfig | Set a configuration parameter |
| SetCommand | Send a Real Time command |
| GetConfig | |
| | Read an operating value |

A set of similar commands are available for accessing/changing configurations, sending commands and reading operating values of remote nodes on CAN networks using the RoboCAN protocol.

## Timers Commands

The following is a set of functions for interacting with the timers:

| | |
|---|---|
| SetTimerCount | Set number of milliseconds for timer to count. |
| SetTimerState | Set state of a specific timer. |
| GetTimerCount | Read timer count. |
| GetTimerState | Read state of a specific timer. |

## Pre-Processor Directives (#define)

The #define creates a macro, which is the association of an identifier with a token expression. After the macro is defined, the compiler can substitute the token expression for each occurrence of the identifier in the source file.

```
#define <var> <expression>
```

The following example illustrates how use pre-processor directive:

```
#define CommandID _GO + 5

Print(CommandID)
```

## Option (Compilation Options)

Micro Basic by default treats undeclared identifiers as integer variables. If you want the compilers checks that every variable used in the program is declared and generate compilation error if a variable is not previously declared, you may use Option explicit compiler option by pacing the following at the beginning of the program:

```
Option Explicit
```

## Dim (Variable Declaration)

Micro Basic contains only two types of variable (Integer and Boolean) in addition to arrays of these types. Boolean and arrays must be declared before use, but Integer variables may not be declared unless you use the Option Explicit compiler directive.

```
Dim var As { Integer | Boolean }
```

The following example illustrates how to declare Integer variable:

```
Dim intVar As Integer
```

Arrays declaration uses a different syntax, where you should specify the array length between square brackets []. Array length should be integer value greater than 1.

```
Dim arr[n] As { Integer | Boolean }
```

The following example illustrates how to declare array of 10 integers:

```
Dim arr[10] As Integer
```

To access array elements (get/set), you may need to take a look to Arrays section (see Arrays on page 18).

Variable and arrays names should follow specification stated in the Variables section (see Variables on page 18).

## If...Then Statement

- Line If

```
If <condition> Then <stmt> [Else <stmt>]
```

- Block If

```
If <condition> [Then]
      <block>
[ElseIf <condition> [Then]
      <block>]
[ElseIf <condition> [Then]
      <block>]
...
[Else
      <block>]
End If
```

An `If`...Then statement is the basic conditional statement. If the expression in the If statement is true, the statements enclosed by the `If` block are executed. If the expression is false, each of the `ElseIf` expressions is evaluated. If one of the `ElseIf` expressions evaluates to true, the corresponding block is executed. If no expression evaluates to true and there is an `Else` block, the `Else` block is executed. Once a block finishes executing, execution passes to the end of the `If...`Then statement.

The line version of the `If` statement has a single statement to be executed if the `If` expression is true and an optional statement to be executed if the expression is false. For example:

```
Dim a As Integer
Dim b As Integer

a = 10
b = 20
' Block If statement.
If a < b Then
    a = b
```

```
Else
    b = a
End If
```

```
' Line If statement
If a < b Then a = b Else b = a
```

Below is an example where `ElseIf` takes place:

```
If score >= 90 Then
    grade = 1
ElseIf score >= 80 Then
    grade = 2
ElseIf score >= 70 Then
    grade = 3
Else
    grade = 4
End If
```

## For...Next Statement

Micro Basic contains two types of For...Next loops:

- **Traditional `For...Next`:**
  Traditional For...Next exists for backward compatibility with Basic, but it is not recommended due to its inefficient execution.

  Traditional For...Next is the same syntax as Basic For...Next statement.

- **C-Style `For...Next`:**
  This is a new style of For...Next statement optimized to work with Roboteq controllers and it is recommended to be used. It is the same semantics as C++ for loop, but with a different syntax.

  ```
  For <var> = <expression> AndWhile <condition> [Evaluate
  <stmt>]
      <block>
  Next
  ```

  The c-style for loop is executed by initialize the loop variable, then the loop continues while the condition is true and after execution of single loop the evaluate statement is executed then continues to next loop.

  ```
  Dim arr[10] As Integer
  For i = 0 AndWhile i < 10
      arr[i] = -1
  Next
  ```

  The previous example illustrates how to initialize array elements to -1.

  The following example illustrates how to use Evaluate to print even values from 0-10 inclusive:

  ```
  For i = 0 AndWhile i <= 10 Evaluate i += 2
      Print(i, "\n")
  Next
  ```

## While/Do Statements

- **While...End While Statement**

```
While <condition>
    <block>
End While
```

Example:

```
a = 10
While a > 0
    Print("a = ", a, "\n")
    a--
End While
Print("Loop ended with a = ", a, "\n")
```

- **Do While...Loop Statement**

```
Do While <condition>
    <block>
Loop
```

The `Do While...Loop` statement is the same as functionality of the `While...End While` statement but uses a different syntax.

```
a = 10
Do While a > 0
    Print("a = ", a, "\n")
    a--
Loop
Print("Loop ended with a = ", a, "\n")
```

- **Do Until...Loop Statement**

```
Do Until <condition>
    <block>
Loop
```

**Unlike** `Do While...Loop` statement, Do `Until...Loop` statement exist the loop when the expression evaluates to true.

```
a = 10
Do Until a = 0
    Print("a = ", a, "\n")
    a--
Loop
Print("Loop ended with a = ", a, "\n")
```

- **Do...Loop While Statement**

```
Do
    <block>
Loop While <condition>
```

`Do...Loop While` statement grantees that the loop block will be executed at least once as the condition is evaluated and checked after executing the block.

```
a = 10
Do
    Print("a = ", a, "\n")
    a--
Loop While a > 0
Print("Loop ended with a = ", a, "\n")
```

- **`Do...Loop Until Statement`**

```
Do
        <block>
Loop Until <condition>
```

**Unlike** Do...Loop While statement, Do...Loop Until statement exist the loop when the expression evaluates to true.

```
a = 10
   Do
       Print("a = ", a, "\n")
       a--
   Loop Until a = 0
   Print("Loop ended with a = ", a, "\n")
```

## Terminate Statement

The Terminate statement ends the execution of the program.

```
Terminate
```

## Exit Statement

The following is the syntax of Exit statement:

```
Exit { For | While | Do }
```

An `Exit` statement transfers execution to the next statement to immediately containing block statement of the specified kind. If the `Exit` statement is not contained within the kind of block specified in the statement, a compile-time error occurs.

The following is an example of how to use Exit statement in While loop:

```
While a > 0
    If b = 0 Then Exit While
End While
```

## Continue Statement

The following is the syntax of Continue statement:

```
Continue { For | While | Do }
```

A Continue statement transfers execution to the beginning of immediately containing block statement of the specified kind. If the `Continue` statement is not contained within the kind of block specified in the statement, a compile-time error occurs.

The following is an example of how to use `Continue` statement in `While` loop:

```
While a > 0
    If b = 0 Then Continue While
End While
```

## GoTo Statement

A GoTo statement causes execution to transfer to the specified label. `GoTo` keyword should be followed by the label name.

```
GoTo <label>
```

The following example illustrates how to use `GoTo` statement:

```
GoTo Target_Label
Print("This will not be printed.\n")
Target_Label:
    Print("This will be printed.\n")
```

## GoSub/Return Statements

`GoSub` used to call a subroutine at specific label. Program execution is transferred to the specified label. Unlike the `GoTo` statement, `GoSub` remembers the calling point. Upon encountering a Return statement the execution will continue the next statement after the `GoSub` statement.

```
GoSub <label>
```

```
Return
```

Consider the following example:

```
Print("The first line.")
GoSub PrintLine
Print("The second line.")
GoSub PrintLine
Terminate

PrintLine:
    Print("\n")
    Return
```

The program will begin with executing the first print statement. Upon encountering the `GoSub` statement, the execution will be transferred to the given **PrintLine** label. The program prints the new line and upon encountering the Return statement the execution will be returning back to the second print statement and so on.

## ToBool Statement

Converts the given expression into boolean value. It will be return `False` if expression evaluates to zero, True otherwise.

```
ToBool(<expression>)

Consider the following example:

Print(ToBool(a), "\n")
```

The previous example will output `False` if value of a equals to zero, `True` otherwise.

## Print Statement

Output the list of expression passed.

```
Print({str | expression | ToBool(<expression>)}[,{str | expression
| ToBool(<expression>)}]...)
```

The print statement consists of the `Print` keyword followed by a list of expressions separated by comma. You can use `ToBool` keyword to force print of expressions as `Boolean`. Strings are C++ style strings with escape characters as described in the Strings section (see Stringspage 16).

```
a = 3
b = 5
Print("a = ", a, ", b = ", b, "\n")
Print("Is a less than b = ", ToBool(a < b), "\n")
```

## + Operator

The + operator can function as either a unary or a binary operator.

```
+ expression
expression + expression
```

## - Operator

The - operator can function as either a unary or a binary operator.

```
- expression
expression - expression
```

## * Operator

The multiplication operator (*) computes the product of its operands.

```
expression * expression
```

## / Operator

The division operator (/) divides its first operand by its second.

```
expression * expression
```

## Mod Operator

The modulus operator (Mod) computes the remainder after dividing its first operand by its second.

```
expression Mod expression
```

## And Operator

The (`And`) operator functions only as a binary operator. For numbers, it computes the bitwise AND of its operands. For boolean operands, it computes the logical AND for its operands; that is the result is true if and only if both operands are true.

```
expression And expression
```

## Or Operator

The (`Or`) operator functions only as a binary operator. For numbers, it computes the bitwise OR of its operands. For boolean operands, it computes the logical OR for its operands; that is, the result is false if and only if both its operands are false.

```
expression Or expression
```

## XOr Operator

The (`XOr`) operator functions only as a binary operator. For numbers, it computes the bitwise exclusive-OR of its operands. For boolean operands, it computes the logical exclusive-OR for its operands; that is, the result is true if and only if exactly one of its operands is true.

```
expression XOr expression
```

## Not Operator

The (Not) operator functions only as a unary operator. For numbers, it performs a bitwise complement operation on its operand. For boolean operands, it negates its operand; that is, the result is true if and only if its operand is false.

```
Not expression
```

## True Literal

The `True` keyword is a literal of type `Boolean` representing the boolean value true.

## False Literal

The `False` keyword is a literal of type `Boolean` representing the boolean value false.

## ++ Operator

The increment operator (++) increments its operand by 1. The increment operator can appear before or after its operand:

```
++ var
var ++
```

The first form is a prefix increment operation. The result of the operation is the value of the operand after it has been incremented.

The second form is a postfix increment operation. The result of the operation is the value of the operand before it has been incremented.

```
a = 10
Print(a++, "\n")
Print(a, "\n")
Print(++a, "\n")
Print(a, "\n")
```

The output of previous program will be the following:

```
10
11
12
12
```

## -- Operator

The decrement operator (--) decrements its operand by 1. The decrement operator can appear before or after its operand:

```
-- var
var --
```

The first form is a prefix decrement operation. The result of the operation is the value of the operand after it has been decremented.

The second form is a postfix decrement operation. The result of the operation is the value of the operand before it has been decremented.

```
a = 10
Print(a--, "\n")
Print(a, "\n")
Print(--a, "\n")
Print(a, "\n")
```

The output of previous program will be the following:

```
10
9
8
8
```

## << Operator

The left-shift operator (<<) shifts its first operand left by the number of bits specified by its second operand.

```
expression << expression
```

The high-order bits of left operand are discarded and the low-order empty bits are zero-filled. Shift operations never cause overflows.

## >> Operator

The right-shift operator (>>) shifts its first operand right by the number of bits specified by its second operand.

```
expression >> expression
```

## <> Operator

The inequality operator (<>) returns false if its operands are equal, true otherwise.

```
expression <> expression
```

## < Operator

Less than relational operator (<) returns true if the first operand is less than the second, false otherwise.

```
expression < expression
```

## > Operator

Greater than relational operator (>) returns true if the first operand is greater than the second, false otherwise.

```
expression > expression
```

## <= Operator

Less than or equal relational operator (<=) returns true if the first operand is less than or equal to the second, false otherwise.

```
expression <= expression
```

## > Operator

Greater than relational operator (>) returns true if the first operand is greater than the second, false otherwise.

```
expression > expression
```

## >= Operator

Greater than or equal relational operator (>=) returns true if the first operand is greater than or equal to the second, false otherwise.

```
expression >= expression
```

## += Operator

The addition assignment operator.

```
var += expression
```

An expression using the += assignment operator, such as

```
x += y
```

is equivalent to

```
x = x + y
```

## -= Operator

The subtraction assignment operator.

```
var -= expression
```

An expression using the -= assignment operator, such as

```
x -= y
```

is equivalent to

```
x = x - y
```

## *= Operator

The multiplication assignment operator.

```
var *= expression
```

An expression using the *= assignment operator, such as

```
x *= y
```

is equivalent to

```
x = x * y
```

## /= Operator

The division assignment operator.

```
var /= expression
```

An expression using the /= assignment operator, such as

```
x /= y
```

is equivalent to

```
x = x / y
```

## <<= Operator

The left-shift assignment operator.

```
var <<= expression
```

An expression using the <<= assignment operator, such as

```
x <<= y
```

is equivalent to

```
x = x << y
```

## >>= Operator

The right-shift assignment operator.

```
var >>= expression
```

An expression using the >>= assignment operator, such as

x >>= y

is equivalent to

x = x >> y

## [ ] Operator

Square brackets (`[]`) are used for arrays (see Arrays on page 18).

## Abs Function

Returns the absolute value of an expression.

```
Abs(<expression>)
```

Example:

```
a = 5
b = Abs(a - 2 * 10)
```

## Atan Function

Returns the angle whose arc tangent is the specified number.

Number is devided by 1000 before applying atan.

The return value is multiplied by 10.

```
Atan(<expression>)
```

Example:

```
angle = Atan(1000) '450 = 45.0 degrees
```

## Cos Function

Returns the cosine of the specified angle.

The return value is multiplied by 1000.

```
Abs(<expression>)
```

Example:

```
value = Cos(0) '1000
[title] Sin Function
Returns the sine of the specified angle.
The return value is multiplied by 1000.
Sin(<expression>)
```

Example:

```
value = Sin(90) '1000
Sqrt Function
Returns the square root of a specified number.
The return value is multiplied by 1000.
Sqrt(<expression>)
```

Example:

```
value = Sqrt(2) '1414
```

## GetValue

This function is used to read operating parameters from the controller at runtime. The function requires an Operating Item, and an optional Index as parameters. The Operating Item can be any one from the table below. The Index is used to select one of the Value Items in multi channel configurations. When accessing a unique Operating Parameter that is not part of an array, the index may be omitted, or an index value of 0 can be used.

Details on the various operating parameters that can be read can be found in the Controller's User Manual. (See "Serial (RS232/USB) Operation" on page 141)

```
GetValue(OperatingItem, [Index])

Current2 = GetValue(_BATAMPS, 2) ' Read Battery Amps for Motor 2

Sensor = GetValue(_ANAIN, 6) ' Read voltage present at Analog Input 1

Counter = GetValue(_BLCOUNTER) ' Read Brushless counter
```

## SetCommand

This function is used to send operating commands to the controller at runtime. The function requires a Command Item, an optional Index and a Value as parameters. The Command Item can be any one from the table below. Details on the various commands, their effects and acceptable ranges can be found in the Controller's User Manual (See "Serial (RS232/USB) Operation" on page 141).

```
SetCommand(CommandItem, Value)

SetCommand(_GO, 1, 500) ' Set Motor 1 command level at 500

SetCommand(_DSET, 2) ' Activate Digital Output 2
```

## SetConfig / GetConfig

These two functions are used to read or/and change one of the controller's configuration parameters at runtime. The changes are made in the controller's RAM and take effect immediately. Configuration changes are not stored in EEPROM.

```
SetConfig Set a configuration parameter
GetConfig Read a configuration parameter
```

Both commands require a Configuration Item, and an optional Index as parameters. The Configuration Item can be one of the valid controller configuration commands listed in the Command Reference Section. Refer to Set/Read Configuration Commands on page 210 for syntax. Simply add the underscore character "_" to read or write this configuration from within a script. The Index is used to select one of the Configuration Item in multi channel configurations. When accessing a configuration parameter that is not part of an array, index can be omitted or an index value of 0 can be used. Details on the various configurations items, their effects and acceptable values can be found in the Controller's User Manual.

Note that most but not all configuration parameters are accessible via the SetConfig or GetConfig function. No check is performed that the value you store is valid so this function must be handled with care.

When setting a configuration parameter, the new value of the parameter must be given in addition to the Configuration Item and Index.

GetConfig(ConfigurationItem, [Index], value)
SetConfig(ConfigurationItem, [Index])

```
Accel2 = GetConfig(_MAC, 2) ' Read Acceleration parameter for Motor 2
PWMFreq = GetConfig(_PWMF) ' Read Controller's PWM frequency
SetConfig(_MAC, 2, Accel2 * 2) ' Make Motor2 acceleration twice as slow
```

## SetTimerCount/GetTimerCount

These two functions used to set/get timer count.

```
SetTimerCount(<index>, <milliseconds>)
GetTimerCount(<index>)
```

Where:

<index> : 0 - 4 for old controller models

0 - 7 for new controller models

<milliseconds> : number of milliseconds to count

## SetTimerState/GetTimerState

These two functions used to set/get timer state (started or stopped).

```
SetTimerState(<index>, <state>)
GetTimerState(<index>)
```

Replace entire sentence with:

Where:

<index> : 0 - 4 for old controller models

0 - 7 for new controller models

<state> : 0 - Timer Running

1 - Timer has completed/stopped (reached 0ms)

## Sending RoboCAN Commands and Configuration

Sending commands or configuration values to remote nodes on RoboCAN is done using the functions.

```
SetCANCommand(<id>, <cc>, <ch>, <vv>)
SetCANConfig(<id>, <cc>, <ch>, <vv>)
```

Where:
id is the remote Node Id in decimal.
cc is the Command code, eg. _G.
cc is the channel number. Put 1 for commands that do not normally require a channel number.
vv is the value.

## Reading RoboCAN Operating Values Configurations

The following functions are available in Micro Basic for requesting operating values and configurations from a remote node on RoboCAN.

```
FetchCANValue(<id>, <cc>, <ch>)
FetchCANConfig(<id>, <cc>, <ch>)
```

Where:
id is the remote Node Id in decimal
cc is the Command code, eg. _G
cc is the channel number. Put 1 for commands that do not normally require a channel number.

The following functions can be used to wait for the data to be ready for reading.

```
IsCANValueReady()
IsCANConfigReady()
```

These functions return a Boolean true/false value. They take no argument and apply to the last issued `FetchCANValue` or `FetchCANConfig` function.

The retrieved value can then be read using the following functions.

```
ReadCANValue()
ReadCANConfig()
```

These functions return an integer value. They take no argument and apply to the last issued `FetchCANValue` or `FetchCANConfig` function.

## RoboCAN Continuous Scan

A scan of a remote RoboCAN node is initiated with the function.

```
ScanCANValue(<id>, <cc>, <ch>, <tt>, <bb>)
```

Where:
id is the remote Node Id in decimal.
cc is the Query code, eg. _V.
cc is the channel number. Put 1 for commands that do not normally require a channel number.
tt is the scan rate in ms.
bb is the buffer location.
The scan rate can be up to 255ms. Setting a scan rate of 0 stops the automatic sending from this node.

Unless otherwise specified, the buffer can store up to 32 values.

The arrival of a new value is checked with the function.

```
IsScannedCANReady(<aa>)
```

Where:

aa is the location in the scan buffer.
The function returns a Boolean true/false value.

The new value is then read with the function.

```
ReadScannedCANValue(<aa>)
```

Where:
aa is the location in the scan buffer.
The function returns an integer value. If no new value was received since the previous read, the old value will be read.

## Checking the Presence of a RoboCAN Node

No error is reported in MicroBasic if an exchange is initiated with a node that does not exist. A command or configuration sent to a non-existent node will simply not be executed. A query sent to a non existing or dead node will return the value 0. A function is therefore provided for verifying the presence of a live node. A live node is one that sends the distinct RoboCAN heartbeat frame every 128ms. The function syntax is:

```
IsCANNodeAlive(<id>)
```

Where:
id is the remote Node Id in decimal
The function returns a Boolean true/false value.