

# BARIONET

TM



## BARIONET

Universal network-enabled  
automation interface in  
home automation, commercial  
control and monitoring applications



## Barix Control Language (BCL) Programmers Manual

Revision 1.46\_20041012



---

# Table of Contents

---

<b>Introduction.....</b>	<b>1</b>
--------------------------	----------

<b>Development Tools .....</b>	<b>2</b>
Editor.....	2
Tokenizer.....	2
Web2cob.....	3
TFTP upload.....	3
Batch files.....	4
WEB Usage Table.....	4
Debugging.....	5

---

<b>General information.....</b>	<b>6</b>
Execution speed.....	6
Runtime environment limitations.....	6
Floating point type variables.....	6
Variable names.....	6
String memory usage.....	7
Temporary string usage.....	7
Storing multiple values in a string.....	7
String constants.....	7
Notations.....	8
Priority of Operations.....	9
Structure of a BCL program.....	9

---

<b>Control statements.....</b>	<b>10</b>
Statement delimiters.....	10
Line Numbers / Labels.....	10
' (apostrophe).....	10
= (equal).....	10
GOTO.....	11
IF .. THEN .. ELSE .. ENDIF.....	11

Multi line IF.....	11
Single line IF.....	12
ELSE.....	12
ENDIF.....	12
FOR .. NEXT.....	12
GOSUB .. RETURN.....	13
DIM.....	14
ON ERROR GOTO.....	14
ON .. GOSUB.....	15
TIMER.....	16
_TMR_ array.....	16
DELAY.....	17
IOCTL.....	17
SYSLOG.....	18
MIDSET.....	19
MIDCPY.....	19

---

**Functions.....20**

LEN.....	20
INSTR.....	20
ISEQV replaced by “=”.....	20
MID\$.....	21
MIDGET.....	21
ASC.....	21
CHR\$.....	21
VAL.....	21
STR\$.....	21
SPRINTF\$.....	22
LCASE\$.....	23
UCASE\$.....	23
SYSTIME replaced by _TMR_(0).....	24
IOSTATE.....	24
PING.....	24
NOT.....	25
AND.....	25
OR.....	25
XOR.....	25
SHL.....	25

SHR.....	25
----------	----

---

**File and stream support.....26**

OPEN .....	26
OPEN TCP.....	26
OPEN UDP.....	26
OPEN COM.....	27
OPEN for CIFS.....	27
OPEN for Flash File System.....	28
OPEN for SETUP.....	28
CLOSE.....	28
READ.....	29
WRITE.....	30
CONNECTED.....	30
MEDIATYPE.....	30
LASTLEN.....	30
PLAY.....	31
RMTPORT.....	31
RMTHOST.....	31
FILESIZE.....	31

---

**WEB interface.....32**

DHTML tags.....	32
Variable setting by CGI.....	32
CGI handling in BCL.....	33

---

**Appendix A - IP Address, Netmask etc.....34**

IP Addressing.....	34
Network Address.....	35
Broadcast Address.....	35
IP Netmask.....	35
Private IP networks and the Internet.....	36
Network RFC's.....	36

---

**Appendix B – BIN / DEC / HEX conversion. 37**



## **Introduction**

---

The Barix Control Language (further referred to as “BCL”) is a high level, interpreted control language, used to program the Barix Barionet.

The aim of BCL is to allow system integrators, OEM developers and certain end users to customize the Barix Barionet to a very high degree by using essentially the syntax of the well-known BASIC language, with significant enhancements to allow for I/O as well as network access (UDP, TCP, CGI etc).

BCL is very easy to learn and allows instant results for most people fluent in a higher level programming language.

## Development Tools

---

### Editor

Development of BCL programs is done in any text editor of the programmers choice – as long as it supports standard ASCII files. Modern development tools – like the free Eclipse development system – allow comfortable editing with syntax highlighting, the use of such tools however is optional. The only requirements are that the editor used can produce an ASCII file containing the BCL program code.

Procedure with Notepad:

- open notepad.exe
- type in your BCL program code
- save as barionet.bas

### Tokenizer

The tool tokenizer is used to compact the ASCII BCL program code and to convert it into the Barix TOK format.

This reduces the memory usage of a BCL program and speeds up the execution significantly.

Command prompt call:

```
tokenizer BCL\barionet.bas
```

BCL\ is the folder that holds the source file barionet.bas

#### **NOTE:**

The tokenizer tool only supports file names obeying the DOS rules (8 characters length, extension .BAS).

If the tokenizer is called from outside a folder (just as in the example above) and there is no **ERRORS.HLP** present in that folder then the tokenizer will create this error help file for you. The content of **ERRORS.HLP** is used to issue syslog errors in clear text.

The interpreter expects a file “barionet.tok” to be present, if the file is not found, it will issue a syslog warning “No barionet.tok” and terminate.

### Web2cob

The resulting `.tok` file must then be wrapped in a `.cob` file (for debugging and / or documentation reasons together with the `.bas` source file) plus files needed by the project (HTML, graphics etc). The tool `web2cob` wraps these files into a single COB file which can be directly loaded into the flash memory of a Barix product.

Command prompt call:

```
web2cob /o barionetbcl.cob /d BCL
```

`/o` defines the name of the created cob file  
`/d` defines which folder should be packed

A created cob file exceeding the size of 64 Kilobytes will use up 2 flash memory pages. This has to be taken in account when uploading to prevent overwriting (see web usage on next page)

### TFTP upload

The above mentioned `cob` file can be uploaded into a flash memory page of the target hardware using the TFTP command.

This windows command can be executed in a command window or an optional TFTP client (available as free ware) can be used to upload comfortably using a graphical user interface.

Command prompt call:

```
tftp -i 192.168.0.10 PUT basictest.cob WEB4
```

To upload files into the Barix Barionet the TFTP protocol can be used but a short pause of approximately 3 seconds after each upload has to be allowed for in order for the Barix Barionet to store the file internally.

#### WARNING:

If this timing is not obeyed, wrong or corrupt files are uploaded to wrong destinations using TFTP unpredictable behaviour can be observed, which might render a Barix Barionet unusable.

Barix recommends the supplied batch files (see next page) and to leave the usage of TFTP to advanced programmers only.

**Batch files**

To facilitate the tokenizing, web2cob and the tftp upload Barix provides the following batch file:

```
bcl <IP address>
```

where <IP address> is the IP address of your Barionet.

This will create a COB file barionetbcl.cob and transfers it to your Barionet.

Here the bcl.bat content:

```
@echo off
echo # Tokenizing BCL...
tokenizer BCL\barionet.bas
echo # Creating BCL COB file...
web2cob /o barionetbcl.cob /d BCL
echo # Transferring to Barionet...
tftp -i %I put barionetbcl.cob WEB4
```

You can also use

```
web <IP address>
```

to update the web UI application separately.

Here the web.bat content:

```
@echo off
echo # Creating Web COB file...
web2cob /o barionetweb.cob /d stuff
echo # Transferring to Barionet...
tftp -i %I put barionetweb.cob WEB1
```

**WEB Usage Table**

WEB usage table (for programmers using tftp as in the batch files)

- X1        Firmware
- WEB1..2 Web UI application, Help, PIC Firmware
- WEB3    Reserved for future increased Web UI application
- WEB4    Sample BCL application Digital I/O Tunnel  
          (including Web setup)
- WEB5    Unused 64 KB (free for customization)
- WEB6    Unused 64 KB (free for customization)
- WEB7    Unused 64 KB (free for customization)

### Debugging

Debugging of BCL programs is facilitated by the device at runtime using syslog messages issued via the network interface. Syslog is a well known reporting application usually using UDP port 514. Check the Internet for a free Syslog daemon. Alternatively a universal logging tool, provided free of charge by Barix, can be used.

Warnings and errors will be issued automatically.

Code example and Syslog warning:

```
DIM TEST$ ( 4096 )
```

```
2004-04-01 11:18:48 192.168.2.70 Warning! Temporary  
string size was increased up to 4096 bytes!
```

To add own messages one can use the SYSLOG statement:

Code example:

```
SYSLOG "TESTING SYSLOG OUTPUT"
```

## General information

---

### Execution speed

The execution speed of BCL programs depends on the specific hardware and firmware used, but is generally very high, typically 5400 instructions per second (1900 in previous versions). In other words: each instruction needs 0.184 milliseconds to execute.

### Runtime environment limitations

The runtime environment clearly has size constraints which should be considered when writing BCL code.

In the current version of BCL the following limitations exist:

- maximal amount of numeric labels **100/1000**
- range of numeric labels **1-65535**
- maximal FOR-NEXT nesting **25**
- maximal GOSUB-RETURN nesting **25**
- maximal recursive nesting (amount of brackets) **10**
- maximal amount of variables of type integer **64**
- maximal amount of variables of type string **64**
- length of variable's names **unlimited**
- significant number of characters in variable's name **5**
- maximal dimensions of arrays of type integer **2**
- size of integer variable **32 bit**
- default size of variable of type string (if not changed by DIM statement) **256 bytes**
- maximal number of files/streams **5**
- default size of buffers for files/streams **512 bytes**

### Floating point type variables

BCL does not support floating point types. For most applications, floating point like operations can easily be done by scaling values.

Suggestion:

to use temperatures values from -99.99 to 199.99 use integers -9999 to 19999 and just convert for input and output.

### Variable names

As the significant number of characters in variable's name is 5 the tokenizer will issue a warning when variables are defined using the same first five characters.

**String memory usage**

When writing a more complex BCL program using a great amount of string variables we advise to decrease memory usage by dimensioning the string variables to their respective need. This can be accomplished with the DIM statement.

**Temporary string usage**

For string calculations BCL uses temporary strings with size of maximal declared string variable (if it exceeds 256 bytes, a warning will be output to syslog). If that string will not be used for calculations (usually if it is a binary buffer assigned only using MIDSET/MIDCPY instructions), the “**\_M**” prefix for the string name can be used to avoid changing the temporary string size.

**Storing multiple values in a string**

Strings can be used to store “bit” or “byte” Variables/Values, so if you interface to a security system with 300 room states, just put these 300 states in a standard string variable (DIM it with a length of 300 bytes) – that way you save on variables and memory space.

Code example: `DIM RSTAT$( 300 )`

Syslog warning: 2004-04-01 11:18:48 192.168.2.70  
Warning! Temporary string size was increased to 300 bytes!

To prevent the warning one can use the “**\_M**” prefix as stated in Temporary strings usage (see above)

Code example: `DIM _MRSTAT$( 300 )`

**String constants**

Commonly used strings (like CR/LF) can be defined in a string, which improves code space and execution speed. However, these strings should be dimensioned by the DIM statement before assigning them to avoid excessive memory usage. Strings are delimited by a Null character so one has to add \ character to the used characters for correct dimensioning.

Code example: `DIM CR$( 3 )`  
`CR$=CHR$( 13 )+CHR$( 10 )`

**Notations**

The following notations are used in this manual:

- N** – numeric constant  
(integer value between  $-2^{147}483^{648}$   
and  $+2^{147}483^{647}$ , inclusive)  
Can also be written Hexadecimal (**&H**nnnnnnnn)
  
- L** – line number/label.  
Labels are unsigned integers from 1 to 65535
  
- “..”** – quoted string constant  
(must be less than 127 characters)
  
- S\$** – string variable (zero terminated by default). With  
some restrictions, string variables can be used to hold  
binary data (all possible 8bit values, including 0)
  
- V** – integer variable or array element  $V(E [ , E ] )$
  
- H** – file handle (integer, 0..4)
  
- F()** – function returning integer
  
- F\$( )** – function returning string
  
- E** – expression of type integer, typically the result of  
arithmetic operations with N, V, and F( )
  
- E\$** – expression of type string, a result of a concatenation  
of “..”, S\$, and F\$( )

**Priority of Operations**

BCL operators have the following priority of execution (from highest to lowest):

- ( ) – brackets
  
- +/- – unary operators
  
- ^ – power (exponentiation)
  
- \* / % – multiplication, division, modulo (remainder of division)
  
- + - – addition or string concatenation, subtraction
  
- = > < – compare operators, equal, greater, less  
the equal sign is also used for assignment operation.

**Structure of a BCL program**

The BCL program preferably starts with the definition and dimensioning of used variables but must end with the END statement or a carriage return/ line feed (CRLF) when using the GOTO or RETURN statement.

Code example 1:

```
DIM CR$ ( 3 )  
.  
.  
END [ EOF ]
```

Code example 2:

```
DIM CR$ ( 3 )  
.  
10 A$ = . . .  
.  
GOTO 10 ( or RETURN )  
[ EOF ]
```

## Control statements

---

**Statement delimiters** Most BCL statements can be delimited with space, tab, line feed (LF) or new line (CRLF) characters.

Comments and **DIM** statements must be terminated with CRLF.

A statement can consist of several *physical* lines concatenated with “&” symbol. The “:” symbol can also be used as a statement delimiter.

**Line Numbers / Labels** Line numbers are optional in BCL. If a line number is used, it must be at the beginning of a line. There is no requirement that line numbers are used in ascending order. Line numbers can be used as labels for the GOTO and the RETURN statements.

**END** End of the BCL program. The runtime interpreter is stopped and the device issues the syslog message:

```
no Barionet.tok
```

**' (apostrophe)** Used for commenting. All text after apostrophe sign is ignored, till the end of current line (CRLF).

Code Example:

```
DIM CR$(3)'dimensioning CRLF string constant
DIM SBUF$(64) ' buffer size of 64 bytes
```

**= (equal)** **V=E** or **S\$=E\$**

Assigns a new value to integer (V) or string (S\$) variable. If that variable does not exist, it is automatically created. In case of a string the default size of 256 Bytes is used.

Code Example:

```
CNT=12 MSG$="Hello world"
```

**GOTO****GOTO L**

Unconditional jump to line number (label) L

Code Example:

```
10 CNT=CNT+1
   A=B*CNT
   GOTO 10
```

**IF .. THEN ..  
ELSE .. ENDIF**

Condition evaluation and branch statement.

**IF** is followed by a (boolean or integer) expression:

```
{ E0 | E1 { = | > | < | >= | <= | <> } E2 }
```

if the logical expression is true or the integer result is nonzero the commands following the **THEN** statement are executed.

Two forms/versions of the IF statement exist.

**Multi line IF**

If **THEN** is the last statement in the line (directly followed by CR/LF), a *multiline IF* statement is assumed and all following lines up to an **ELSE** or **ENDIF** statement are executed. In that case, **ELSE** must be the last statement in a line as well. The **ELSE** statement is optional.

Code Example:

```
10 CNT=0 B=2
   CNT=CNT+1
   A=B*CNT
   IF A < 500 THEN
       MSG$="not yet there"
       C=0
   ELSE
       MSG$="there now"
       C=1
   ENDIF
   GOTO 10
```

**Single line IF**

If **THEN** is followed by one or more statements, these are executed when the expression is true, up to an **ELSE** statement or the end of the line (CR/LF). CR/LF is implicitly treated as **ENDIF**.

Code Example:

```
10 CNT=0
20 CNT=CNT+1
   IF CNT < 500 THEN GOTO 20 ELSE GOTO 10
```

If the expression/integer result is false (zero), execution continues after either the first unmatched **ELSE** statement or **ENDIF** (multiline IF) or CR/LF (single line IF).

**ELSE**

part of the **IF .. THEN .. ELSE .. ENDIF** functionality, see above.

**ENDIF**

part of the **IF .. THEN ... ELSE .. ENDIF** functionality, see above. Only used with multi line IF.

**FOR .. NEXT**

**FOR V=E1 TO E2  
NEXT [V]**

Loop statement. First, V is assigned result of expression E1. Then all statements up to the first unmatched NEXT (or the NEXT statement with the correct variable) are executed. When the NEXT statement is reached, V is incremented and compared with E2. The execution restarts at the FOR statement as long as V is less or equal than E2. If V is larger than E2, the loop is terminated and the statement following NEXT is executed. V can be modified in the loop.

**NOTE:** the programmer is strongly discouraged to use **GOTO** to jump into or out of **FOR .. NEXT** loops.  
To end a **FOR .. NEXT** loop, set the loop variable to E2+1.

Code Example:     **CNT=4**  
                  **FOR I=1 TO 100**  
                          **CNT=CNT\*I**  
                  **NEXT I**

**GOSUB ..  
RETURN****GOSUB L****RETURN [LI]**

Subroutine call and return.

The interpreter remembers the actual code position and starts interpreting with the statement following line/label L.

When a **RETURN** statement is found execution is resumed on the statement following the calling **GOSUB** instruction.

```
Code Example:  10 ' main loop
                GOSUB 20
                GOSUB 30
                GOTO 10

                20 FOR I=1 TO 100
                  CNT=CNT*I
                NEXT I
                RETURN
```

If parameter **LI** is given in the **RETURN** statement, execution is resumed at that label.

```
Code Example:  10 ' main loop
                GOSUB 20
                GOSUB 30
                GOTO 10

                20 IF R=1 THEN RETURN 10
                  A=5
                RETURN
```

**WARNING:**

The use of the **GOTO** statement to jump into or out of a subroutine is prohibited!

To end a subroutine, the **RETURN** statement must be used, otherwise the calling stack of the interpreter is not cleared and can cause erratic behavior up to restarting the device.

**DIM**

**DIM { V(N [,N] ) | S\$(N) } [ , { V(N [,N] ) | S\$(N) } ] ...**

The DIM statement is used for the purpose of defining and initializing variables. Although optional for integer and string variables, it is mandatory for integer arrays and strings with nonstandard size ( $\neq$  256 bytes).

With the execution of the DIM statement, the necessary space for the variable or array is acquired from the heap and the variable or array is initialized to 0 or in case of string to an empty string.

Code Example:    **DIM X(10,10)**

defines two-dimensional integer array (10x10)

Code Example:    **DIM BUF\$(1024)**

defines string variable with a max. size of 1024 bytes

**Note:** for normal (non-binary) use, string variables are terminated by a trailing zero character, so a variable dimensioned to a size of 1024 can hold a string of maximal 1023 characters.

**ON ERROR  
GOTO****ON ERROR GOTO L**

With this command, a line number is defined to which the interpreter will branch in case of an (recoverable) error to line/label **L**.

This allows the BCL program to catch certain runtime errors and handle them appropriately.

If the line number given is 0, errors will be handled by the BCL interpreter's default error handler, usually terminating the program with an error message to syslog.

The error handler has access to the error occurred and line number by reading integer variables **\_ERR\_** and **\_ERL\_**.

**ON .. GOSUB****ON { TIMER{1|2|3|4} | CGI | UDP } GOSUB L**

The **ON ... GOSUB** constructs allow limited event-driven structuring of a program.

When the **ON ... GOSUB** construct is interpreted, an event handler subroutine (indicated by label/line number **L**) is entered in a table, and in case the matching event is triggered, the interpreter will branch to that subroutine.

Obviously, this subroutine should return as soon as possible with a **RETURN** statement to allow continuation of normal program operation.

Code example:    **ON CGI GOSUB 100**

A label/statement number of 0 disables the function for that specific event.

Code example:    **ON UDP GOSUB 0**

The following events can be used to trigger the call of a subroutine:

- 4 independent software timers (resolution in milliseconds). Timers must be set up using the **TIMER** statement
- incoming UDP packet
- incoming CGI request to special handler

Digital and analog inputs cannot be used to trigger directly events but when using a **ON TIMERx GOSUB** statement then the input states can be checked in the called subroutine using the interval of the used timer.

## TIMER

### TIMER E0, E

Set timer **E0** to trigger every **E** milliseconds. Timer is reset with this statement so it will not trigger before the **E** milliseconds.

Valid timers are 1, 2, 3, 4 and 0.

Code example: `TIMER1,100`

defines the timer to go off every 100 milliseconds

A parameter of 0 disables the timer.

Code example: `TIMER1,0`

Timer 0 is a special timer. If set, it can be used as a “software watchdog”, and once the count of this timer expires the BCL program will be terminated. To prevent this it has to be reset using the statement:

### TIMER 0, E

Code example: `TIMER0,5000`

resets the watch dog timer for the next 5 seconds

## `_TMR_` array

### `_TMR_(E)`

The actual value of all timers (counting up from 0..value set using this statement) can be read from the special variable array

Code example:

```

t1=_TMR_(0)
10 i=i+1
a$=STR$(i)
IF i<=1000 THEN GOTO 10
t2=_TMR_(0)
SYSLOG "Tested 1000 loops in"
SYSLOG STR$(t2-t1)+" msec."
    
```

**DELAY**

**DELAY E**

Delays execution of the program for E milliseconds.

Code example:    **DELAY 500**  
                  **SYSLOG "DONE"**

waits half a second and then sends syslog message "DONE"

Event handling is unaffected by the delay and continues to operate. A value of "0" is understood as "forever" - indefinitely.

Code example:    **DELAY 0**

waits forever

**NOTE:**

DELAY is ignored in ON-call subroutine during ON-call event.

**IOCTL**

**IOCTL E0, E**

Sets I/O variable E0 or port to value E.

This function is hardware dependent.

Most I/O variables, are bit (0/1) or 16 bit variables, however, 32 bit variables might also exist (counters for example).

Please refer to the Barionet manual for details.

Code example:    **IOCTL 101,1**  
                  **IOCTL 2,999**

activates first the digital output 1 and then toggles the 2<sup>nd</sup> relay

**SYSLOG****SYSLOG E\$ [,E]**

Sends **E\$** as a UDP message to Syslog (port 514).

Code example: `SYSLOG "ALARM"`

If given, the optional parameter **E** specifies a debugging level.

The output can be controlled by two special variables, `_DBG_` and `_SIP_$`.

Messages with **E** given are only sent if  $E \geq \_DBG\_$ .  
The default value of `_DBG_` is 0, so all messages are sent.

Code example: `_DBG_=4`  
`SYSLOG "ALARM",4`  
`SYSLOG "NEVER",5`

will only send the message "ALARM", as the syslog debug level in variable `_DBG_` is set to 4 the second syslog will not be issued

**SYSLOG** messages are sent as broadcast unless the variable `_SIP_$` is defined.

If it is defined, it must contain a valid IP address to which Syslog messages are then sent.

Code example: `_SIP_$="192.168.2.13"`  
`SYSLOG "ALARM"`

will send the syslog message "ALARM" only to the host on IP address 192.168.2.13

**MIDSET**

**MIDSET S\$,E0,EI,E**

inserts **E** as byte (**EI=1**), word(**EI=2**), or double word (**EI=4**) into position **E0** of string variable **S\$** (binary array).

Word and double word storage is done in the little endian (Intel) format.

Code example: `BA$= " " ' hex 20202020  
MIDSET BA$,2,1,64`

will result in BA\$ looking like this in hex: 2020642020

**MIDCPY**

**MIDCPY S\$,E0,EI,SI\$**

inserts **EI** bytes into position **E0** of string variable **S\$** from the beginning of string variable **SI\$**.

Code example: `A$= "is great!"  
B$= "This was"  
MIDCPY A$,0,5,B$`

will result in A\$ containing "This is great!"

**NOTE!**

For string calculations BCL uses temporary strings with size of maximal declared string variable (if it exceeds 256 bytes, a warning will be output to syslog).

If that string will not be used for calculations (usually if it is a binary buffer assigned only using MIDSET/MIDCPY instructions), the “\_M” prefix for the string name can be used to avoid changing the temporary string size.

## Functions

---

### LEN

#### LEN( E\$ )

Returns the length of string **E\$** as an integer.

Code example: `A=LEN( "SHORT" )`

will store 5 in variable A

### INSTR

#### INSTR( E, E1\$, E2\$ )

Returns position of **E2\$** in the **E1\$** (search is performed from position **E**).

Code example: `A$= "is it here?"`  
`B$= "i"`  
`POS=INSTR(1,A$,B$)`

will store 3 in variable A as we start the search from "s" on

### ISEQV replaced by "="

#### ISEQV( E1\$, E2\$ )

This function is replaced by the equal sign (=) as strings can now be matched directly :

#### E1\$=E2\$

Code example: `IF A$=B$ THEN`  
`SYSLOG "same"`  
`ELSE`  
`SYSLOG "different"`  
`ENDIF`

**MID\$**

**MID\$( E\$, EI, E2 )**

Returns sub-string of E\$ (E2 chars starting from position EI).

If a string variable is used as a binary array (in which case a string variable S\$ must be used instead of a string expression E\$), it may contain several bytes with value 0, which can be located by using the **MID\$(S\$, index, 1)** function.

**MIDGET**

**MIDGET(S\$,E0,EI)**

Extracts integer variable as byte, word, or double word (EI=1, 2, and 4 respectively) from position E0 of string variable S\$ (binary array).

**NOTE!** The variable is retrieved in little endian (Intel) format.

**ASC**

**ASC( E\$ )**

Returns ASCII code of first character in the string E\$.

**CHR\$**

**CHR\$( E )**

Returns character (string of length one) having ASCII code E.

**VAL**

**VAL( E\$ )**

Interprets string E\$ as an ASCII representation of an integer, and returns the value.

**STR\$**

**STR\$( E )**

Returns a string containing the ASCII representation of integer value E.

**SPRINTF\$**

**SPRINTF\$( E\$, E )**

Converts integer value E to a string, using C-style formatting specified by E\$, and returns the string. The format string uses common “C” notation, and a maximum of one parameter is allowed. The parameter can be either an 32bit integer or a string.

Code example: `A$=PRINTF$ ( "%s" ,1922 )`

will store the string “1922” in the variable A\$

The following formats are supported:

**Standard “C” functions:**

- %u unsigned 16 bit integer
- %lu unsigned 32 bit integer
- %x 16 bit hex value
- %lx 32 bit hex value
- %c character
- %s string

**Special Barionet functions:**

Version related:

- %V firmware version (e.g. B1.29)
- %1V incl. \_build date YYYYMMDD (e.g. B1.29\_20040514)

Network related:

- %1H MAC address (e.g. 00:20:4A:80:40:87)
- %xA access to current network variables (e. g. 192.168.0.2)
- %0xA same with leading zeroes (e. g. 192.168.000.002)

<b>x</b>	<b>function</b>
1	IP address (e. g. 192.168.0.2)
2	Netmask (e. g. 255.255.255.0)
3	Default Gateway (e. g. 192.168.0.1)
4	Domain Name Server   “DNS I” (e. g. 192.168.0.1)
5	Broadcast address (e. g. 192.168.0.255)

Time related:

`%xt, p` access to time function where x values can be added up for concatenated time string

<b>x</b>	<b>function</b>
0	default time YYMMDDHHMM (e.g. 0405140914)
1	include seconds SS (e.g. 040514091459)
2	include leading century yyYY (e.g. 200405140914)
4	adjust for local time zone and DST (in future release)
8	leading character for time valid ("2" not set, "3" set)
16	16bit parameter p in sec. since 1/1/1970 (0=SYSTIME)

Code example: `A$=SPRINTF$ ("%3t", 0)`

Result: `A$ : "20041012034759"`

## **LCASE\$**

### **LCASE\$( E\$ )**

Returns a string produced by converting all characters in E\$ into lower case.

Code example: `OUT$=LCASE$( "LOWER" )`

Result: `OUT$ : "lower"`

## **UCASE\$**

### **UCASE\$( E\$ )**

Returns a string produced by converting all characters in E\$ into upper case.

Code example: `OUT$=UCASE$( "Upper" )`

Result: `OUT$ : "UPPER"`

**SYSTIME** replaced  
by **\_TMR\_(0)**

**SYSTIME**

Returns time in milliseconds since last boot/startup.

This function supported in earlier versions is now replaced by the direct access to the special variable **\_TMR\_(0)** which holds the content of the timer 0 running in milliseconds.

```
Code example: 10 STIME=_TMR_(0)
               SYSLOG STR$(STIME)
               DELAY 1000
               GOTO 10
```

See also control statement **TIMER E0, E** and variable array **\_TMR\_(E0)** when custom timing (<>msec) is needed. Furthermore check the function **SPRINTF\$("%!t",0)** to access time in a string with the format "YYMMDDHHMMSS".

**IOWSTATE**

**IOWSTATE( E0 )**

Returns state of I/O variable or port E0.

This function is hardware dependent. Most I/O variables, are bit (0/1) or 16 bit variables, however, 32 bit variables might also exist (counters in the Barionet, for example).

```
Example:      INP1=IOWSTATE( 201 )
```

stores the state of digital input 1 in variable INP1

**PING**

**PING( E\$, E )**

Returns the time period (in milliseconds) the device with IP address specified by E\$ needed to respond to a PING (icmp echo) request, or 0 if no reply was received within E milliseconds (timeout).

```
Example:      IP$="192.168.2.18"
               rtime=PING( IP$, 50 )
```

stores the time period needed to receive the PING reply from the host on IP address 192.168.2.18

**Logical operations**

Logical/boolean expressions (**bE**) in BCL can have the value of TRUE (-1) or FALSE (0).

All logical functions described below are either bitwise (perform bit calculations) if integer variable (E) or constant was specified, or logical if logical expression or constant was used.

**AND(bE), OR(bE), XOR(bE)** with one argument returns TRUE or FALSE for logical expression **bE**.

**NOT****NOT( { E | bE } )**

Bitwise/logical **NOT** operation.

**AND****AND( { E | bE } [ , ... ] )**

Bitwise/logical **AND** operation.

**OR****OR( { E | bE } [ , ... ] )**

Bitwise/logical **OR** operation.

**XOR****XOR( { E | bE } [ , ... ] )**

Bitwise/logical **XOR** operation.

**SHL****SHL( E, E0 )**

Bitwise shift left of E by E0 bits.

**SHR****SHR( E, E0 )**

Bitwise shift right of E by E0 bits.

## File and stream support

---

BCL programs can access a variety of I/O resources. This is universally done via file handles. Up to 5 such file handles are available and can be opened at the same time, but only the first three handles (0,1,2) can be used for TCP connections. File handle #0 has a large receiving buffer if used with TCP/IP (4kByte).

### OPEN

#### OPEN S\$ AS H

Open function, file handle H will be assigned to a specific I/O source.

S\$ must be a STRING (not a string expression !), and starts with a three-letter I/O identifier, followed by a colon and parameters specific to that I/O method.

The following identifiers are possible:

### OPEN TCP

#### TCP:

TCP socket, both passive and active connections. Next parameter is the IP address (0.0.0.0 for a listening connection), followed by a colon and the port number.

Examples:

“TCP:0.0.0.0:1000” - listening TCP connection on port 1000

“TCP:192.168.11.99:1000” - active connection open (session will be established) to IP address 192.168.11.99 Port 1000

### OPEN UDP

#### UDP:

UDP socket, used for both receiving and sending. Next parameter is the IP address (0.0.0.0 for a listening socket), followed by a colon and the port number.

Examples:

“UDP:0.0.0.0:1000” - listening UDP socket for packets arriving on port 1000

**OPEN COM**

**COM:**

Opens serial port(s).

Depending on Hardware configuration, various ports are supported.

Please refer to the specific product manual for details.

Example:

“COM:,,,;:1” - open port 1 on device

Settings for Baudrate, parity, 7 or 8 bits, 1 or 2 stop bits are taken from the configuration settings and are accessible via STP read and write (see further below).

**OPEN for CIFS**

**C\_R:, C\_W:, C\_D:**

CIFS (Windows file sharing protocol), open file for reading, writing or directory search. Not supported on Barionet.

The parameter is the IP/share of the file to access.

Example:

“C\_R://192.168.5.1/Share/Directories/Data/File.txt”

For “C\_D:...” values, S\$ is changed to the first directory name after **OPEN S\$ AS H**

User ID/Password information can be attached to the file name preceded by a “@” character.

If no user id is given, the “GUEST” user id will be used.

Example:

“C\_R://192.168.5.1/Share/File.txt@hello:password”

**OPEN for Flash  
File System**

**F\_R:**

Flash file system in device.  
With this I/O method, files in the local flash can be opened for reading.

Example:

“F\_R:testfile.txt” - opens file “testfile.txt” for reading.

**OPEN for SETUP**

**STP:**

“Configuration Setup file”, access to the eeprom where the nonvolatile parameters of the device are stored.

Some of the accessible data space is used by the OS and should not be altered, some of the space is available for the BCL program to store paramers.

Details about the Setup layout are product specific, please refer to the product manual for details.

The read and write operations use strings for binary operations, so with a read the complete string will be filled with data from the eeprom (typically 256 bytes unless the string is DIM'ed otherwise).

One parameter after the identifier is used to specify an Offset in the setup eeprom.

Example:

“STP:512” - opens file access to eeprom from position 512.

**CLOSE**

**CLOSE H**

Closes file or stream with handle H  
or flushes media buffer if H = 0.

For Setup (STP) “file” saves new Setup in EEPROM, if **WRITE** operator was used on the file.

**READ****READ H, S\$ [ ,E0 [ ,E\$ ] ]**

Reads from stream H into string variable S\$.

In case of CIFS directory, S\$ should contain previous directory entry.

The EOF condition can be checked using the LASTLEN(H) function (-1 for EOF).

Without the optional parameters, files are read in “binary” mode, the maximum amount of Bytes which can be read (up to the size of the variable) are returned then.

If the optional parameter E0 is 0, the file/stream is read in “line” mode, every read returns then a complete text line of the input, with CR/LF being stripped off.

Only non-empty text lines are returned !

For streams (COM, TCP), parameter E0 can also be a positive integer > 0. In that case, READ either returns immediately with no data (if no data is in the receive buffers), or READ only returns if the given buffer can be completely filled, or no additional data is received from the stream for at least E0 milliseconds.

This allows for very simple implementation of block protocols which define the end of a message as a timeout time.

The second optional parameter, E\$, if given, defines a “match” string. With E\$, the READ function will skip all data received on the stream until an exact match with E\$ is found and then starts reading from the first character immediately after the match string.

This functionality is ideally suited to read data after a certain tag in XML or web data.

If E\$ is given but not found, the function returns and subsequent calls to LASTLEN(H) will yield -1.

**WRITE**

**WRITE H, S\$, E0 [ , E\$ , EI ]**

Writes into stream H, E0 bytes from S\$ string variable.

If E0 = 0, writes complete string (length determined by terminating 0 in string, text mode).

To write zero, use empty string S\$ and E0 = 1.

If this is a UDP stream, E\$ specifies destination IP address and EI destination port number.

**CONNECTED**

**CONNECTED( H )**

Returns TRUE if connection was established for TCP-based stream H, or FALSE otherwise.

**MEDIATYPE**

**MEDIATYPE( H )**

Returns media type number if stream H was opened, or 0 if **OPEN** has failed or file or stream is already closed.

**LASTLEN**

**LASTLEN( H )**

Returns last read/write length for stream H.

It is also used as error code or EOF (End Of File) mark.

For UDP receiving it returns a negative value if new data is available for reading. After reading, the value is positive, unless new data is received.

The function is also important if a stream is used in binary mode. In that case LASTLEN can be used to determine how much data has been read, as LEN does only determine the string length of the buffer, which is indicated by a 0 at the “end”, which is meaningless for binary data.

**RMTPORT**

**RMTPORT( H )**

Returns remote port for the stream H, or 0 if not applicable.

It can be used for a UDP reply to sender's port number.

**RMTHOST**

**RMTHOST\$( H )**

Returns remote host IP for the stream H, or an empty string.

It can be used for a UDP reply to determine the IP address of the sender.

**FILESIZE**

**FILESIZE( H )**

Returns file size (or -1 for directory) of file/stream H.

Not applicable for serial streams.

## WEB interface

---

To interact with BCL programs from web pages a simple DHTML interface is implemented.

### DHTML tags

Use the following HTML tags to insert the current value of any BCL Variable in dynamic HTML pages:

**&LBAS(I, "%2ld", V) or &LBAS(I, "%s", S\$)**

where V and S\$ are integer and string variables. The functions will return "[NO\_VAR]" if the variable is not defined).

Please note that for integer variables the format string must be "%ld", and for string variables the format string must be "%s".

### Variable setting by CGI

To set a value of a BCL variable from a web page, use the "BAS.cgi" cgi script. As parameters, variable=value pairs are given. Example, as it could be used in HTML code:

```
<a href="/BAS.cgi?S$=start&V=0" target="empty">
```

where "BAS.cgi" is name of the interface script, "&" is delimiter between variables, and each variable value is specified as "name=value".

In the above example, V and S\$ are integer and string variables, already defined in the BCL program.

**NOTE!** Do not wrap string values in quotes.

It is also possible to use the HTML form construct for the same purpose, as it is shown in the following example:

```
<form name="DT" action="BAS.cgi"
method="GET" target="empty">
  <input type="hidden" name="S$"
value="start">
  <input type="hidden" name="V" value="0">
  <input type="submit" value="Send DATA">
</form>
```

## CGI handling in BCL

To handle web requests directly in a BCL program, the “[basic.cgi](#)” CGI script can be used.

All parameters passed to the script after “?” are accessible as a special string Variable, `_CGI_$`, by the BCL program. This variable must be declared and set to the empty sting at program initialization by the programmer..

To receive the request, either check this variable periodically, or use the **ON CGI...** statement.

To send a reply, set the `_CGI_$` variable back to “” and assign reply string to `_CGI_$`. If first symbol of `_CGI_$` is “\*”, The interface functions assume that the rest of the string is a name of an existing file in the flash memory, and send this file as a reply.

Otherwise, the interface functions assume that the BCL program has prepared a valid HTML page, adds HTTP header and sends it as a reply. Please note, that in this case the frame indicated in **target** option will receive the data.

### NOTE:

All BCL variables are case insensitive and internally stored in uppercase format, therefore references to variables using the above interfaces must also specify variable names in upper case.

## Appendix A - IP Address, Netmask etc

---

### IP Addressing

An IP address is a 32 bit value, divided into four octets of eight bits each. The standard representation is four decimal numbers (in the range of 0..255), divided by dots.

Example: 192.2.1.123

This is called decimal-dot notation.

The IP address is divided in two parts: a network and a host part. To support different needs, three "network classes" have been defined. Depending on the network class, the last one, two or last three bytes define the host, while the remaining part defines the network. In the following text, 'x' stands for the host part of the IP address:

### Class A network

IP address 1.x.x.x to 127.x.x.x

Only 127 different networks of this class exist. These have a very large number of potential connected devices (up to 16777216)

Example: 10.0.0.1, (network 10, host 0.0.1)

### Class B network

IP address 128.0.x.x to 191.255.xxx.xxx

These networks are used for large company networks. Every network can consist of up to 65534 devices.

Example: 172.1.3.2 (network 172.1, host 3.2)

### Class C network

IP address 192.0.0.xxx to 223.255.255.xxx

These network addresses are most common. Most smaller companies networks are class C networks. These networks can consist of a maximum number of, 254 hosts.

Example: 192.7.1.9 (network 192.7.1, host 9)

The remaining addresses 224.x.x.x - 239.x.x.x is defined as "Class D" and is used as a multicast address.

Note: No addresses are allowed with the four highest-order bits set to 1-1-1-1. (240.x.x.x - 254.x.x.x) These addresses, called "class E", are reserved.

**Network Address** The host address with all host bits set to "0" is used to address the network as a whole (for example in routing entries).

**Broadcast Address** The address with the host part bits all set to "1" is the broadcast address, meaning "for every station".

Network and Broadcast addresses must not be used as a host address (e.g. 192.168.0.0 identifies the entire network, 192.168.0.255 identifies the broadcast address).

**IP Netmask** The netmask is used to divide the IP address differently from the standard defined by the classes A,B,C. Entering a netmask, it is possible to define how many bits from the IP address are to be taken as the network part and how many bits are to be taken as the host part.

**Standard IP network netmask:**

	Network bits	Host bits	Netmask
Class A	8	24	255.0.0.0
Class B	16	16	255.255.0.0
Class C	24	8	255.255.255.0

**Netmask examples**

Netmask	Host bits
255.255.255.252	2
255.255.255.248	3
255.255.255.240	4
255.255.255.224	5
255.255.255.192	6
255.255.255.128	7
255.255.255.0	8
255.255.254.0	9
255.255.252.0	10
255.255.248.0	11
.	.
.	.
255.128.0.0	23
255.0.0.0	24

**Private IP networks  
and the Internet**

If your network is not connected to the Internet and there are no plans to make such a connection you may use any IP address you wish.

However if you are not connected to the Internet and have plans to connect to the Internet or you are connected to the Internet and want to operate your Barionet on an intranet you should use one of the sub-networks below for your network. These network numbers have been reserved for such networks. If you have any questions about IP assignment ask your Network Administrator.

Class A	10.x.x.x
Class B	172.16.x.x
Class C	192.168.0.x

**Network RFC's**

For more information regarding IP addressing see the following documents. They can be found on the Internet:

RFC 950	Internet Standard Subnetting Procedure
RFC 1700	Assigned Numbers
RFC 1117	Internet Numbers
RFC 1597	Address Allocation for Private Internets

## Appendix B – BIN / DEC / HEX conversion

---

Hexadecimal digits have values from 0..15, represented as 0...9, A (for 10), B (for 11) ... F (for 15). The following table can serve as a conversion chart bin - dec. - hex:

**BIN / DEC / HEX**  
table

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

To convert a binary value in a hexadecimal representation, the upper and lower four bits are treated separately, resulting in a two-digit hexadecimal number.

© 2004 Barix AG, Zürich, Switzerland.

All rights reserved.

The newest information about our devices and the latest version of this manual is available via download from our website, [www.barix.com](http://www.barix.com).

We explicitly reserve the right to change and improve the product without notice.

All trademarks mentioned or used are belonging to their respective owners.

Barix and Barionet are registered trademarks.

**Barix AG**

Seefeldstr. 303  
8008 Zürich  
SWITZERLAND

Phone: +41 43 433 22 11  
Fax: +41 1 274 2849

Internet

web: <http://www.barix.com>

email: [sales@barix.com](mailto:sales@barix.com)