

PROGRAMMING



BCL Barix Control Language

Basic like programming language for Barix Automation products and Audio products running the ABCL Virtual Machine



Programmers Manual

Version VI.03

Released 14. Sep. 2007

Supports:

Supports:

- Barionet (LX & EX XPort)
- Annunicom 100, 1000, IC
- Exstreamer 100, 1000, Red, Digital, Gold
- Instreamer 100
- IPAM 100, 200



Table of Contents

1 Introduction.....	1
1.1 Notation	1
1.2 Supported devices.....	2

2 Development Tools	3
2.1 Editor.....	3
2.2 Tokenizer.....	3
2.3 Web2cob.....	3
2.4 Program upload.....	4
2.5 Batch files.....	4

3 The BCL language.....	5
3.1 Syntax of the BCL language.....	5
3.1.1 Simple program.....	5
3.1.2 Comments.....	5
3.1.3 Command delimiters.....	5
3.1.4 Multi-line commands.....	5
3.1.5 Recommended structure of BCL programs.....	6
3.2 Integers.....	6
3.2.1 Integer constants.....	6
3.2.2 Integer variables.....	6
3.2.3 Integer expressions.....	6
3.2.4 Integer functions.....	7
3.2.5 Real numbers.....	7
3.2.6 Integer Arrays.....	7
3.2.7 Bit operations.....	8
3.3 Strings.....	8
3.3.1 String constants.....	8
3.3.2 Escape sequences.....	8
3.3.3 String expressions.....	9
3.3.4 String variables.....	9
3.3.5 Binary arrays.....	10
3.4 Execution flow control commands.....	10
3.4.1 The END command.....	10
3.4.2 Labels.....	10
3.4.3 Unconditional jump.....	10
3.4.4 The FOR-NEXT loop.....	10
3.4.5 Subroutines.....	11
3.4.6 Conditional statements.....	12
3.4.7 Time.....	13
3.4.8 Events.....	13
3.4.9 The LOCK command.....	15
3.5 Functions working with strings.....	16
3.5.1 String/Integer conversions.....	17
3.6 Network functions.....	19
3.7 Miscellaneous functions.....	19
3.8 Binary array functions.....	20
3.9 User defined functions.....	20
3.10 I/O stream functions.....	21
3.10.1 The UDP network protocol.....	23
3.10.2 The TCP network protocol.....	25
3.10.3 Audio interface (audio devices only).....	26
3.10.4 Serial port.....	31

3.10.5	SETUP.....	31
3.10.6	The USB filesystem (not supported on Barionet).....	32
3.10.7	The local flash filesystem.....	34
3.10.8	The Wiegand reader (Barionet only).....	34
3.11	Direct hardware access.....	36
3.12	Diagnostic functions.....	36
<hr/>		
4	Interpreter information.....	38
4.1	Execution speed.....	38
4.2	Runtime environment limitations.....	38
4.3	System variables.....	38
<hr/>		
5	WEB interface	39
5.1	HTML tags.....	39
5.1.1	Displaying variables in webpages.....	39
5.1.2	Calling a subroutine from a webpage.....	40
5.2	Variable setting by CGI.....	40
5.3	CGI handling in the BCL.....	41
<hr/>		
6	Preprocessor.....	43
6.1	Preprocessor directives.....	43
6.2	Using the preprocessor.....	43
<hr/>		
7	Debugging.....	45
7.1	Error messages.....	45
<hr/>		
8	Example programs.....	49
8.1	Playing an MP3 file from the USB filesystem.....	49
8.2	Record audio into an MP3 file on the USB filesystem.....	49
8.3	Sending an email.....	49
8.4	Streaming MP3 over RTP.....	50
8.5	TCP serial gateway	50
8.6	The Wiegand reader.....	52
8.7	Simple internet radio player	53
<hr/>		
9	Syntax summary.....	55
9.1	Variables, Constants, Expressions.....	55
9.2	Declarations.....	56
9.3	Statements and functions.....	57
<hr/>		
10	Appendix A - obsolete or unimplemented functions.....	60
<hr/>		
11	Appendix B - BIN / DEC / HEX conversion.....	62
<hr/>		
12	Appendix C - BCL version 2.....	63
<hr/>		
	Alphabetical Index.....	64
<hr/>		
	Legal Information.....	67

I Introduction

The Barix Control Language (further referred to as "BCL") is a high level, interpreted control language, used to program certain Barix devices (further referred to as "BCL devices") .

The aim of BCL is to allow system integrators, OEM developers and skilled end users to customize Barix BCL devices to a very high degree by using essentially the syntax of the well-known BASIC language. It has built-in support for various input/output interfaces and for various network protocols.

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

I.1 Notation

When introducing command/function syntax, the following notation is used in this manual:

Notation symbol	Meaning
N	Integer constant, value between -2.147.483.648 and +2.147.483.647 can be also written in hexadecimal notation (corresponding range is from &H00 to &HFFFFFFF)
L	Line number. Line numbers are unsigned integers from 1 to 65535
Q\$	Quoted string constant of length up to 255 characters
S\$	string variable (zero terminated). With some restrictions, string variables can be used to hold binary data (all possible 8-bit values, including 0)
V	integer variable or array element V(E [, E])
H	file handle (integer in range 0..7)
F()	function returning integer
F\$()	function returning string
E	expression of type integer, typically a result of arithmetic operations with N, V, and F()
E\$	expression of type string, the result of concatenating Q\$, S\$, and F\$ ()
bE	boolean expression
[...]	square brackets are used to indicate that the bracketed part is optional
{...}	curly brackets are used together with vertical bars to list possible options

I.2 Supported devices

BCL is currently supported by the automation controller Barix Barionet and all Barix Audio products including the Audio Modules IPAM 100 and IPAM 200.

Furthermore supported are all legacy Audio products except for the Exstreamer Wireless.

See table below for I/O protocols supported on above mentioned devices.

I/O protocol	Barionet	IP Audio Module	Exstreamer family	Instreamer family	Annunicom family
TCP/UDP networking	•	•	•	•	•
Serial port(s)	•	• ¹	•	•	• ²
Web interface	•	•	•	•	•
Audio output		•	•	•	•
Audio input		•		•	•
USB filesystem		•	• ³	• ⁴	• ⁵
One-wire sensors	•				
TFTP	•				
Wiegand reader	•				
Flash write	•				

1 Two serial interfaces available

2 Two serial interfaces available on Annunicom 100, the second one being RS485

3 Not available on older hardware versions prior to Exstreamer 100

4 Not available on older hardware versions prior to Instreamer 100

5 Not available on older hardware versions prior to Annunicom 100

2 Development Tools

This section describes usage of tools required for development of a BCL program. Development tools are also described in detail in the Barionet Development Kit Manual document, which is available from the Barix website.

2.1 Editor

BCL programs can be developed with any text editor – as long as the editor supports standard ASCII files with CRLF newlines¹. An example of such an editor is the Notepad application shipped with the Microsoft Windows operating system. Modern development tools – like the free Eclipse development system – allow comfortable editing with syntax highlighting, the use of such tools is however optional.

BCL source program files are expected to have `.bas` extension with the exception of files to be preprocessed (for details, see section 6, page 43).

2.2 Tokenizer

The BCL language interpreter can run programs in Barix TOK format. In this format, individual tokens (atomic part of the source code – operators, function and variable names, constants,...) of the source BCL file are encoded in a space efficient way in order to improve execution speed.

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

Command prompt call:

```
tokenizer program.bas
```

where `program.bas` is the name of the the source file, will tokenize the program and create `program.tok` file.

The tokenizer also creates a file called `ERRORS.HLP` if it doesn't exist. The `ERRORS.HLP` file is used for generating syslog messages in clear text and therefore it is recommended to include this file in the `.cob` file (see the next section).

2.3 Web2cob

The resulting `.tok` file generated by the tokenizer must be stored in a `.cob` file (for debugging and/or documentation reasons together with the `.bas` source file) plus any files needed by the project (HTML, graphics etc). The tool `web2cob` can be used to wrap the contents of a directory into a single COB file which can be directly loaded on a Barix device.

Command prompt call:

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

`/o` defines the name of the output cob file

`/d` defines which directory should be packed

Note: A cob file exceeding 64 Kilobytes will use two or more flash memory pages. This has to be taken into account when uploading - to prevent unintended overwriting of other flash content.

Note: Maximal allowed size of files in a cob file is 64kB. Larger files are not supported.

¹ As common in DOS and Windows operating systems

2.4 Program upload

The above mentioned `cob` file can be uploaded into a flash memory page on the target hardware using the TFTP protocol (Barionet) or via the web interface.

A comfortable graphical client or a command-line TFTP tool can be used. For example a command-line utility called `tftp` shipped with the Microsoft operating system can be used in the following way:

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

(In this example COB file `basictest.cob` is uploaded in the flash page `WEB4` of the device with the IP address `192.168.0.10`)

There should be a short pause of approximately 3 seconds after each upload in order to allow the Barix BCL device to store the file internally.

WARNING: Incorrect timing may result in corrupted files.

Note: Tftp uploading of BCL `.cob` files to certain Barix BCL devices is possible only when these devices are in the bootloader mode (the IP Audio Module is an example of such a device).

Barix recommends using the supplied batch files (see the next section).

2.5 Batch files

To make the tokenizing, `web2cob` and the `tftp` upload easier, Barix provides the `bcl` batch file that should be used in the following way:

```
bcl <name> <IP address>
```

where `<IP address>` is the IP address of your BCL device and `<name>` is a name of a subdirectory containing the BCL source files. The source file has to have the same name as the subdirectory. In the following example program `myprog` is loaded into flash page 4 of the device with IP address `192.168.2.145`. The directory `myprog` contains BCL source `myprog.bas`.

```
bcl myprog 192.168.2.145
```

Content of the `bcl.bat` follows:

```
cd %1
del *.bak
..\tokenizer %1.bas
if errorlevel 1 goto quit
cd ..
web2cob /o %1.cob /d %1
tftp -i %2 put %1.cob WEB4
goto endit
:quit
echo "ERROR - TOKENIZER REPORTS FAILURE"
cd ..
:endit
```

Note: The `bcl.bat` file can be modified to upload `.cob` files to another WEB page. Consult the documentation of the BCL device for the list of the WEB pages available for user programs.

3 The BCL language

3.1 Syntax of the BCL language

3.1.1 Simple program

Here is a simple program to test that tokenizer, tftp uploading, BCL interpreter in the BCL device and syslog daemon¹ are all working well:

```
SYSLOG "Hi, everything is OK"
END
```

After the program is uploaded to the device and interpreted, messages similar to the following should appear in the syslog:

```
Dec  2 15:53:53 192.168.2.145 BARIX BCL Interpreter, V1.2
Dec  2 15:53:53 192.168.2.145 Hi, everything is OK.
Dec  2 15:53:53 192.168.2.145 BCL end
```

BCL keywords are case insensitive. Parameters should be separated by a comma (',', ASCII character 44). For functions, the parameters should be in parenthesis, for example:

```
I=PING("192.168.2.18", 50)
```

even if the value is not used:

```
PING("192.168.2.18", 50)
```

3.1.2 Comments

It is possible to have useful comments inside the source BCL file. The ' (apostrophe, ASCII code 39) character is used for commenting. All text after the apostrophe sign is ignored till the end of the line (CRLF).

```
'This is our first program!
SYSLOG "Hi, everything is OK"      'send message using syslog
command
'end of our first program!
END
```

Functionally this program is exactly the same as the first program so the syslog output is identical:

```
Dec  2 15:53:53 192.168.2.145 BARIX BCL Interpreter, V1.2
Dec  2 15:53:53 192.168.2.145 Hi, everything is OK.
Dec  2 15:53:53 192.168.2.145 BCL end
```

3.1.3 Command delimiters

Most BCL statements can be delimited with new line (CRLF, ASCII codes 13,10) or ':' (colon, ASCII code 58) characters.² Comments and DIM statements (see section 3.2.6 Interger Arrays on page 7) must be terminated with CRLF.

3.1.4 Multi-line commands

It is possible to write multi-line commands by putting an '&' character (ampersand) at the end of the line to be continued. An example follows:

```
SYSLOG "1":SYSLOG"2":SYSLOG &
"3"
```

¹ For more information about syslog, see section 7, page 45

² Using space (' ', ASCII code 32) as separator, which was possible in previous versions, is considered deprecated and will not be supported in future versions of BCL.

3.1.5 Recommended structure of BCL programs

The BCL program code should start with the definitions and dimensioning of the variables used and end with the `END` command or with a carriage return/line feed (CRLF) when using the `GOTO` or `RETURN` statement.

Code examples:

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

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

3.2 Integers

3.2.1 Integer constants

Integer constants (denoted `N` in this document) can be written as ordinary signed integers. They must be in the range from `-2147483648` to `+2147483647`. They can be also written in hexadecimal notation using `&H`, eg. `&H1A` instead of `26`.

3.2.2 Integer variables

Integer variables are identified by their name (case insensitive), of which only the first five characters are significant. Variable names must begin with a letter and can consist only of alphanumerical characters and underscores. Integer variables can hold integers in the range allowed for integer constants.

Integer variables can be assigned values using the assign operator `=` with syntax

```
V=E
```

where `V` is the name of the variable and `E` is an integer expression.

Integer variables should be declared with the `DIM` command at the beginning of the program for better code legibility. If `DIM` is omitted, variables are declared implicitly.

Integer variables are always initialized to `0` at startup (no matter if `DIM` is used or not).

It's possible to declare multiple variables with one `DIM` command. Syntax of `DIM` is the following:

```
DIM NAME1 [, NAME2 [, NAME3 ...] ...]
```

Example:

```
DIM a, b, c

a=17
b=3*a
c=b+5
```

3.2.3 Integer expressions

Integer expressions can be formed using the following operators

Integer operators (descending priority of evaluation)	
()	brackets
+,-	unary sign operator
^	exponentiation
*,/,%	multiplication, division, remainder(modulo)
+, -	addition, subtraction

Operators can be applied to integer constants, integer variables and integer functions.

3.2.4 Integer functions

Functions returning integer values are called integer functions. Several built-in functions are available and it is also possible to create user defined functions, see section 3.9, page 20.

3.2.5 Real numbers

BCL does not support floating point types. For most applications floating-point-like operations can be easily created by scaling values as in the following example.

```
SYSLOG "Computing the circumference and the area of a circle"
radius = 13          'set radius
phi = 314           'set approximately the value of phi times 100
circum = 2*phi*radius 'compute circumference
area = phi*radius^2 'compute the area
SYSLOG "Given circle of radius"+STR$(radius)+ &
      ", the circumference is " &
      + STR$(circum/100)+"."+STR$(circum%100)+" and the area is "+ &
      STR$(area/100)+"."+STR$(area%100)+" ."
```

To print numbers in fixed decimal point format use `SPRINTF$` with the `%F` flag (see section 3.5.1.1 on page 17).

3.2.6 Integer Arrays

It's possible to use one dimensional or two dimensional arrays of integers. Such arrays are declared by the `DIM` command with the following syntax:

```
DIM NAME (INDEX)          - for a one dimensional array
or
DIM NAME (INDEX1, INDEX2) - for a two dimensional array
```

where `NAME` is the name of the array and `INDEX`, `INDEX1`, `INDEX2` are the highest possible indices. As indexing of array elements starts from zero, an array declared with the highest possible index `INDEX` will be of size `INDEX+1`. For example an array declared as `DIM NUM(5)` has 6 elements numbered from 0 to 5. Elements of the array can be accessed using the syntax `NAME (INDEX)` or `NAME (IND1, IND2)`. Arrays are initialized to 0 at startup.

Code Example:

```
DIM OLD(2) 'declare array of size 3
DIM NEW(2) 'declare array of size 3
DIM DIFF(2) 'declare array of size 3

....

DIFF(0) = NEW(0)-OLD(0)
DIFF(1) = NEW(1)-OLD(1)
DIFF(2) = NEW(2)-OLD(2)
```

3.2.7 Bit operations

Bit operations are implemented with the syntax of integer functions. The following bit operations are available:

NOT(E)

Bitwise NOT operation.

AND(E [, E1 [, ...]])

Bitwise AND operation. If only E is given, returns E.

OR(E [, E1 [, ...]])

Bitwise OR operation. If only E is given, returns E.

XOR(E [, E1 [, ...]])

Bitwise XOR operation. If only E is given, returns E.

SHL(E, E0)

Bitwise shift left of E by E0 bits.

SHR(E, E0)

Bitwise shift right of E by E0 bits.

Note: Some bitwise operations have the same names as logical operations and similar syntax, but they can be distinguished by the type of their parameters.

3.3 Strings

In BCL strings are NULL terminated and indexed from 1. (Future releases: please note change 2, chapter 12, page 63)

3.3.1 String constants

String constants are always quoted and their maximum length is 255 characters.

An example of such a constant is the "Hi, everything is OK" in the following program:

```
SYSLOG "Hi, everything is OK"
END
```

3.3.2 Escape sequences

Escape sequences can be used to include special ASCII characters in a string constant. Escape sequence counts as a single character when calculating string length.

Escape sequences	
\a	ASCII character 7
\b	ASCII character 8
\t	ASCII character 9 (horizontal tab)
\n	ASCII character 10 (LF – line feed)
\v	ASCII character 11 (vertical tab)
\f	ASCII character 12
\r	ASCII character 13 (CR - carriage return)
\\	ASCII character 92 (backslash)
\"	ASCII character 34 (")
\xhh	ASCII character with hexadecimal index equal to hh

Code example:

```
V="This string contains CRLF\r\n"
```

3.3.3 String expressions

String expressions can be formed by concatenating string constants, string variables and string functions using the + operator. One simple example is the following modification of the first program:

```
SYSLOG "Hi, "+" everything is OK"           'string expression example
```

3.3.4 String variables

String variables are identified by their names (case insensitive), of which only the first five characters are significant. String variable name must begin with a letter, can consist only of alphanumerical characters and underscores, but the last character has to be '\$'. The tokenizer generates a warning message when variables are defined using the same first five characters.

String variables can be assigned values using the assign operator = with syntax

```
S$=E$
```

where S\$ is the name of the variable and E\$ is a string expression.

Example:

```
FIRST$ = "Hi, "           'assign value to the FIRST$ variable
SECOND$ = " everything OK!" 'assign value to the SECOND$ variable
CONCAT$ = FIRST$+SECOND$
SYSLOG CONCAT$           'syslog concatenation
```

Syslog output will be the same as in the previous example.

String variables should be declared with the DIM command at the beginning of the program for better code legibility. If DIM is omitted, variables are declared implicitly.

At startup string variables are initialized to an empty string.

By default the maximum length of string variables is 256 characters. String variables longer than 256 characters must be declared using the DIM command, with syntax DIM NAME\$(SIZE), as in the following example:

```
DIM LONG$(600)           'LONG$ can hold 599 characters
LONG$="....."          'assign 8 dots
LONG$=LONG$+LONG$+LONG$+LONG$ 'assign 32 dots
LONG$=LONG$+LONG$+LONG$+LONG$ 'assign 128 dots
LONG$=LONG$+LONG$+LONG$+LONG$ 'assign 512 dots
```

This program creates a string consisting of 512 dots to syslog (useful probably only as an example). For normal use, string variables are terminated with a trailing zero character, so a variable dimensioned to a size of 600 can hold a string of maximum 599 characters.

Commonly used string constants (like the CR/LF newline sequence) can be defined in a string, which can save code space. However, these strings should be dimensioned with the DIM command before assigning them to avoid excessive memory usage.

```
DIM CR$(3)
CR$="\r\n"              'newline sequence
```

Note: String array is not available in BCL. If needed, it can be simulated with one long string using string functions to access substrings.

3.3.5 Binary arrays

Strings can be used as binary buffers (e.g. when reading/writing files) or as bit or byte arrays. E.g. when interfacing to a security system with 300 rooms where there is an 8-bit state for each room, it is better to store the states into a string variable (DIM it with a length of 300 bytes) instead of an integer array. This way memory can be saved because an integer array of the same size would need four times more memory (integers are 32-bit).

When storing binary data into a string, the string concatenation operation can not be used, since binary data may contain the 0 character which is a string terminator in text mode. Therefore it's always necessary to work with the string and its length (in separate variable) and concatenate strings with MIDCPY. To access elements of a binary array use MIDSET/MIDGET commands.

For string calculations BCL uses a temporary buffer with a size of the largest string variable declared (if it exceeds 256 bytes a warning will be issued to the tokenizer console). If the string is not going to be used for calculations (typically if it is a binary working buffer for MIDSET/MIDCPY/MIDGET commands), the string name should start with the "_M" prefix to avoid changing of the internal string buffer. The "_M" prefix counts as two of the five significant variable name characters.

3.4 Execution flow control commands

3.4.1 The END command

END command stops the interpreter. It has the following syntax

```
END [E$]
```

where the optional parameter E\$ can be used to start another BCL program. In that case, E\$ should contain the name of the program to be executed.

END statement can be used anywhere in the program.

3.4.2 Labels

Line numbers are optional in BCL, but they are essential for jumping/subroutine calls. If a line number is used, it must be placed at the beginning of the line. Line numbers can be used in any order, but they must be used uniquely.

3.4.3 Unconditional jump

Unconditional jump to label L has the syntax

```
GOTO L
```

After interpreting this command the BCL interpreter continues the execution starting from the line labeled L.

Code example:

```
10 SYSLOG "Neverending loop"
   GOTO 10
```

3.4.4 The FOR-NEXT loop

The syntax of the FOR-NEXT loop is

```
FOR V=E1 TO E2
.....
NEXT [V]
```

First, V is assigned the result of the expression E1. Then all statements up to the matching NEXT statement are executed. When the NEXT statement is reached, V is incremented and compared with

E2. The execution restarts at the the FOR statement as long as V is less than or equal to E2. If V is larger than E2, the loop is terminated and the execution continues after the NEXT statement.

Code example:

```
DIM OLD(25)      'declare one dimensional array, size 26
DIM NEW(25)     'declare one dimensional array, size 26
DIM DIFF(25)    'declare one dimensional array, size 26

....
FOR V=0 TO 25
    DIFF(V)=NEW(V)-OLD(V)      'calculate differences
NEXT V
```

NEXT is the closing statement of the FOR-NEXT loop and there's only one NEXT allowed per loop. The following example is illegal:

```
FOR i=1 TO 10
    if ... THEN NEXT i      ' ILLEGAL !!! - use GOTO instead
    ...
NEXT i
```

Note: V can be modified in the loop, which can be used for early loop termination.

Note: The programmer is strongly discouraged from using GOTO to jump into FOR..NEXT loops. Jumping out of the loops using GOTO is possible. Another way to leave a FOR..NEXT loop is to set the loop variable to E2.

Nested FOR loops are allowed but correct order of FOR and NEXT must be kept:

```
FOR A=1 TO 10
FOR B=1 TO 10
...
NEXT A      ' This is WRONG!
NEXT B

FOR A=1 TO 10
FOR B=1 TO 10
...
NEXT B      ' This is CORRECT
NEXT A
```

3.4.5 Subroutines

```
GOSUB L
...
L ...
....
RETURN [L1]
```

When a GOSUB is found the interpreter remembers the actual code position and starts interpreting with the statement at line/label L.

When a RETURN command is found the execution is resumed at the first statement after the calling GOSUB instruction. If optional parameter L1 is used the execution is resumed at line L1. Only lines to which GOTO jump from the original return point would be allowed can be used for L1.

WARNING: The use of the GOTO statement to jump into or out of a sub-routine is prohibited!

To end a subroutine, the RETURN command must be used, otherwise the calling stack of the interpreter is not cleared which may result in a stack overflow and a program termination with an error message.

3.4.6 Conditional statements

Condition evaluation and code branching are possible using the `IF` statement. `IF` is followed by a boolean or integer expression:

If the logical expression is true or the integer result is non-zero the commands following the `THEN` statement are executed. Two syntax forms of the `IF` statement exist:

3.4.6.1 Multiline IF

If the expression is true and `THEN` is the last statement on the line (excluding comments), a multiline `IF` statement is assumed and all following lines up to the first unmatched `ELSE` or `ENDIF` statement are executed.

In that case the optional `ELSE` must be the last statement on the line as well and if the expression result is false (zero), execution continues after either the first unmatched `ELSE` statement or an `ENDIF`.

Code Example:

```
IF A < 500 THEN
    MSG$="A"
ELSE
    MSG$="there now"
ENDIF
SYSLOG MSG$
```

Note: In the current version of the BCL interpreter, due to the execution speed it's recommended to use single-line `IF` where possible or `GOTO/GOSUB` instead of long `IF` branches.

3.4.6.2 Single line IF

In the case that the expression is true and `THEN` is followed by one or more statements these statements are executed up to the first unmatched `ELSE` statement or an end of the line (CR/LF). A CR/LF is implicitly treated as an `ENDIF`.

Code Example:

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

If the expression result is false (zero), execution continues after either the first unmatched `ELSE` statement or a CR/LF.

3.4.6.3 Boolean expressions

Simple boolean expressions made of integer expressions have the following syntax:

```
E1 = E2, E1 > E2, E1 < E2, E1 >= E2, E1 <= E2, E1 <> E2
```

Simple comparison of strings is also possible:

```
S1$ = S2$, S1$<>S2$
```

Logical/boolean expressions (`bE`) in the BCL can have a value of logical constant `TRUE` (-1) or `FALSE` (0). Complex logical expressions can be built using the following logical functions:

`NOT(bE)`
logical NOT operation.

`AND(bE [, bE2 [, ...]])`
logical AND operation.

`OR(bE [, bE2 [, ...]])`

logical OR operation.

`XOR(bE [, be2 [, . . .]])`

logical XOR operation.

Code Example:

```
IF AND(A>5,B<7) THEN SYSLOG "A is greater than 5 and B is less than 7"
```

Note: AND (bE), OR (bE), XOR (bE) with one argument return the value of expression bE.

3.4.6.4 Multiple branching depending on an integer value

Multiple branching depending on an integer value is possible with the following syntax:

`ON E {GOSUB | GOTO } L1, [L2, [L3, . . .]]`

If E equals 1, then GOSUB/GOTO to label L1 is executed.

If E equals 2 and L2 is given, then GOSUB/GOTO to the label L2 is executed.

If E equals 3 and L3 is given, then GOSUB/GOTO to the label L3 is executed.

etc.

If E is less than 1 or greater than the number of given labels, no action is taken.

Note: Since it is possible to use complex expressions as E, jumping can take place for various values. For example, by shifting the value of integer variable V by 499 we can use multiple branching to jump in the cases 500, 501, 502:

```
ON (V-499) GOTO 6010,6020,6030
```

3.4.7 Time

System variable `_DTS_` is initialized during boot time and then incremented every second. On devices supporting an RTC (realtime clock chip) `_DTS_` is initialized to the current time read from the RTC (number of seconds since 1/1/1970), on other devices `_DTS_` is initialized to zero or set via NTP protocol.

`_DTS_` can be used when programming time dependent programs.

Any value can be assigned to `DTS` using the following code sequence:

```
_DTS_ =0
DELAY 0
_DTS_ =value
```

If RTC is supported by the device, it's value is updated as well.

`DELAY E`

Delays the execution of the program for E milliseconds (maximum possible delay is 65535 ms).

Code example:

```
DELAY 500
SYSLOG "DONE"
```

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

Note: DELAY is ignored in ON-call subroutine during the ON-call event handling.

3.4.8 Events

A program can have a limited event-driven structure using the `ON . . GOSUB` construct.

3.4.8.1 Timers

Four independent software timers (resolution in milliseconds) can be used to trigger the call of a subroutine. Timers must be set up using the `TIMER` statement

`TIMER E0, E`

Set the timer `E0` to trigger every `E` milliseconds. The timer is reset with this statement so it will be first triggered after `E` milliseconds.

Valid timers are 1, 2, 3, 4.

Code example: `TIMER 1, 100`

defines the timer to go off every 100 milliseconds

A parameter of 0 disables the timer.

The actual value of all timers (counting up from 0 to value set using the `TIMER` statement) can be read from the special variable array variable `_TMR_`. Besides, `_TMR_(0)` returns the number of milliseconds since the last hardware restart.

Handling of time events is defined using the following statement:

```
ON TIMER{1|2|3|4} GOSUB L
```

When the `ON . . . GOSUB` construct is interpreted, an event handler subroutine (indicated with a label/line number `L`) is entered in a table. Then if the matching event is triggered, the interpreter executes the registered subroutine.

This subroutine should return as soon as possible with a `RETURN` statement because handling of other events is not possible until then. A label/statement number of 0 disables this function.

3.4.8.2 UDP event

Incoming UDP packet can be used to trigger the call of a subroutine.

Handling of incoming UDP blocks can be defined using following statement:

```
ON UDP GOSUB L
```

When the `ON UDP GOSUB` construct is interpreted, an event handler subroutine (indicated with a label/line number `L`) is entered in a table. The interpreter then executes the registered subroutine every time one or more UDP handles receive an incoming block.

The handling subroutine has to check all receiving UDP handles for an incoming block (`LASTLEN` returns negative value) and process all blocks before it returns. Otherwise a block may be lost. It may also happen, that the handler subroutine is called while all incoming blocks have already been processed.

In that case the handling subroutine should not perform any action.

This subroutine should return as soon as possible with a `RETURN` statement because handling of other events is not possible until then. A label/statement number of 0 disables this function.

For example see section 3.10.1, page 23.

3.4.8.3 CGI event

Handling of CGI requests can be defined using the following statement:

```
ON CGI GOSUB L
```

When the `ON CGI GOSUB L` construct is interpreted, an event handler subroutine (indicated with a label/line number `L`) is entered in a table. The interpreter then executes the registered subroutine in the case of CGI request.

This subroutine should return as soon as possible with a `RETURN` statement because handling of other events is not possible until then. A label/statement number of 0 disables this function.

See also section 5, page 39.

3.4.8.4 Handling I/O events

Digital and analog inputs can not be used to trigger events directly, they have to be polled. Typically a subroutine is registered with the `ON TIMERx GOSUB` statement and the input states are polled by this routine in a defined time interval (depending on the timer used).

3.4.8.5 Error Handling

Error handling routine can be set using:

```
ON ERROR GOSUB L
```

This command stores the line number/label of the error handling subroutine. In case of a (recoverable) error the interpreter executes the subroutine at line/label `L`. This allows the BCL programmer to catch certain runtime errors and handle them appropriately.

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

The error code and the line number where the error has occurred are stored in system variables `_ERR_` and `_ERL_` respectively.

Note: The error handler is not triggered by warnings.

3.4.9 The LOCK command

The `LOCK` command is multipurpose. With only one parameter it locks (`LOCK 1`) or unlocks (`LOCK 0`) the BCL interpreter into memory, which means that no task switching will occur and the BCL interpreter will be the only application running (i.e. web server, audio handling, etc. are stopped). This is useful only in very specific, time critical situations.

`LOCK 2` reboots the device

`LOCK 3` reboots into the bootloader mode (this function is supported only by certain BCL devices).

With `LOCK x, y` certain services can be disabled in runtime.

`LOCK 0, x` enables services masked with bit-mask `x` (see the table and examples below)

`LOCK 1, x` disables services masked with bit-mask `x` (see the table and examples below)

Bit index	Service
0	snmp write
1	snmp read
2	modbus/tcp write
3	modbus/tcp read
4..7	reserved
8	rc.cgi
9	i/o dynamic tags
10	setup.cgi
11	setup dynamic tags
12	BAS.cgi
13	basic variable dynamic tags

Bit index	Service
14	Basic.cgi
15	tftp

Examples:

```
Lock 1, 32768
```

disables the TFTP upload function, all other functions are enabled.

```
Lock 0, &H0C00
```

enables the setup functions (cgi and dynamic tags), all other functions are enabled.

3.5 Functions working with strings

The BCL language provides a variety of functions for working with strings. Note that in all the following examples strings are indexed from 1.

LEN(E\$)

Returns the length of the string E\$ as an integer (excluding the terminating NULL character).

Example:

```
A=LEN("SHORT")
```

will store 5 in variable A

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

Searches for E2\$ in E1\$ starting from the position indexed by E up to the end of the string (the first character \0 in the string). On success it returns the position of the E2\$ in E1\$, counting from 1 (for E2\$ at the beginning of E1\$). Otherwise it returns 0. Search for an empty string E2\$ returns 0.

If E is negative, searches from index -E up to the next character \0. This can be used for searching in binary arrays or in a concatenation of multiple \0 terminated ASCII strings.

Example:

```
A$= "is it here?"
B$= "i"
POS=INSTR(2, A$, B$)
```

will store 4 in variable POS as we start the search from "s" on.

Future releases: please note change 1, chapter 12, page 63.

MID\$(E\$, E1 [, E2])

Returns the sub-string of E\$ consisting of E2 characters starting from the position E1. E1 counts from 1. In the case that E2 is omitted, returns all characters from position E1 to the end of E\$.

Example:

```
A$= "is it here?"
B$=MID$(A$, 4, 2)
```

will store "it" in variable B\$.

If a string variable is used as a binary array MID\$ accepts a string variable S\$ instead of a string expression E\$.

LCASE\$(E\$)

Returns a string produced by converting all characters of E\$ to lower case.

For example, after executing

```
OUT$=LCASE$ ("LoWeR")
```

the value of OUT\$ will be "lower"

UCASE\$ (E\$)

Returns a string produced by converting all characters of E\$ to upper case.

For example, after executing

```
OUT$=UCASE$ ("Upper")
```

the value of OUT\$ will be "UPPER"

3.5.1 String/Integer conversions

ASC (E\$)

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

Example: Value 32 is stored into variable N after execution of

```
N=ASC(" there is a space at the beginning of this string")
```

CHR\$ (E)

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

Example: S\$ equals to " " after execution of

```
S$=CHR$ (32)
```

VAL (E\$)

Converts the initial portion of the string E\$ to an integer and returns the value. E\$ must be decimal.

Examples:

```
a=VAL ("123")
```

returns 123 .

```
a=VAL ("09BA")
```

return 9 .

```
a=VAL ("Fred")
```

returns 0 .

STR\$ (E)

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

STIME (E\$)

returns the time E\$ converted to seconds since 1/1/1970. Format of the E\$ string is "YYMMDDhhmmss" . See also SPRINTF\$ below.

3.5.1.1 Integer to string conversions

SPRINTF\$ (E\$, E)

Converts the integer value E into a string using C-style formatting specified in the format string E\$ and returns the result. The format string uses the common "C" notation but only one parameter is allowed

Code example:

```
A$=SPRINTF$ ("the value is %u",1922)
```

will store the string "the value is: 1922" in the variable A\$

The following formats are supported:

- %[[-|0]n]u unsigned 16 bit integer
- %[[-|0]n]lu unsigned 32 bit integer
- %[[-|0]n]d signed 16 bit integer
- %[[-|0]n]ld signed 32 bit integer
- %[[-|0]n]x 16 bit hex value
- %[[-|0]n]lx 32 bit hex value
- %c as character in ASCII

where:

- "-" aligns to the left side,
- "0" adds the leading zeros,
- "n" number of character positions for the output

%0.xF can be used to print integers in fixed decimal point format, the decimal point is moved to the left by x places to divide the number by 10^x. This feature is available only on the Barionet.

Code example:

```
A$=SPRINTF$ ("%0.2F", 123)
A$:"1.23"
```

3.5.1.2 Printing version information

- %V firmware version (e.g. B1.29)
- %1V the same as the above including "-" (underscore) and the build date YYYYMMDD (e.g. B1.29_20040514)

3.5.1.3 Printing network information

- %H MAC address without separators (e.g. 00204A804087)
- %1H MAC address with colon separators (e.g. 00:20:4A:80:40:87)
- %A access to current network variables (e. g. 192.168.0.2) (see below)
- %1A same with leading zeroes (e. g. 192.168.000.002)

variable returned depends on the parameter E:

- 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 I "DNS I" (e. g. 192.168.0.1)
- 5 Broadcast address (e. g. 192.168.0.255)

%1A takes a parameter E encoding an IP address in a 32-bit signed integer and outputs it in the dotted quad notation. Outputs the bits in the following order, 0 being the least significant bit, 31 the most significant: <0-7>.<8-15>.<16-23>.<24-31>.

- %11A same with leading zeroes

3.5.1.4 Time to string conversion

%xt converts either the system time or the provided argument (see the bit 4 below) into a time string. The value x is bitwise OR of any combination of the below bits.

The full time format is: [v] [yy]YYMMDDhhmm[ss] [w]

By default (if x is 0) prints the system time in format YYMMDDhhmm (e.g. 0405140914).

bit function

- 0 including seconds *ss* (e.g. 040514091459)
- 1 including the leading century *yyyy* (e.g. 200405140914)
- 2 adjust for a local time zone and DST (in future releases)
- 3 leading character for time valid ("2" invalid time, "3" valid time)
- 4 use 32-bit parameter *E* as time source (number of seconds since 1/1/1970) instead of the system time
- 5 including *w* - one-digit week-day number in range 1-7, 1 is Sunday (e.g. 04051409145)

Code example:

```
A$=SPRINTF$ ("%1t", 0)
```

Result depends on system time, possible output for example

```
A$ : "00490606000002"
```

3.6 Network functions

RESOLVE (E\$)

Resolves a string address to an IP address stored in integer. *E\$* can be either a numeric IP address in the dot notation or a DNS address. If *E\$* is a DNS address, tries to resolve it asking the configured DNS servers. Then returns the IP address as an integer in the range -2147483648...2147483647. IP address written as A.B.C.D is put into a signed 32-bit number, A into LSB and D into MSB. If $D < 128$ then the resulting number is $A + 256 * B + 65536 * C + 16777216 * D$, if $D \geq 128$ then the resulting number is $A + 256 * B + 65536 * C + 16777216 * D - 4294967296$. Example: 192.168.2.3 results in $192 + 256 * 168 + 65536 * 2 + 16777216 * 3 = 50505920$. If the DNS resolution fails or *E\$* is not a valid IP address, the function returns 0.

Note: For converting addresses the other way around use `SPRINTF$ ("%1A", E)` (see above).

3.7 Miscellaneous functions

RANDOM([E])

interfaces to the built-in non-linear additive feedback 16-bit pseudo-random number generator.

Called without parameters returns the next pseudo-random number in the sequence as a positive value between 0 and 65535.

When called with parameter *E*, sets *E* as the seed for a new sequence of pseudo-random numbers and returns 0.

Note: When initializing the seed, RANDOM has to be called in an assignment, e.g.:

```
dummy=RANDOM(123)
SYSLOG "random number "+STR$(RANDOM())
```

MD5\$(S\$, E, [E0])

(Audio only)

calculates the MD5 sum of the first *E* bytes of *S\$*. The optional parameter *E0* defines the return format. If omitted or set to 0 the function returns 16 binary characters. If set to 1 returns the MD5 sum in hexadecimal ASCII notation (with capital letters), e.g.:

```
SYSLOG MD5$ ("hello", 5, 1)
```

sends a message "5D41402ABC4B2A76B9719D911017C592"

3.8 Binary array functions

MIDSET S\$,E0,E1,E

stores the E as a byte (E1=1), a word (E1=2), or a double word (E1=4) at position E0 (starting from 1) of the string variable S\$ (binary array).

Words and double words are written in the little endian (Intel) format by default. If E1 is negative (-1, -2, -4) the value is written in the big endian format.

Code example:

```
BA$ = "      " ' hex 2020202000
MIDSET BA$,2,1,64
```

will result in BA\$ (in hex): 2040202000

MIDGET (S\$,E0,E1)

Returns a byte, word, or double word (E1=1, 2, and 4 respectively) at position E0 (starting from 1) of the string variable S\$ (binary array).

The value is read in the little endian (Intel) format by default. If E1 is negative (-1,-2,-4) the value is read in the big endian format.

MIDCPY S\$,E0,E1,S1\$[,E2]

replaces E1 bytes at position E0 (starting from 1) of the string variable S\$ with E1 bytes from the beginning of the string variable S1\$. If optional parameter E2 is used, replaces with E1 bytes of string variable S1\$ starting from offset E2.

Code example:

```
A$= "Come here!"
B$= "Look there!"
MIDCPY A$,1,5,B$
```

will result in A\$ containing "Look here!"

Another code example:

```
A$= "Come here!"
B$= "Look there!"
MIDCPY B$,6,5,A$,6
```

will result in B\$ containing "Look here!!"

3.9 User defined functions

User defined functions in BCL are implemented as subroutine (GOSUB-RETURN) calls and the return value is stored in an integer variable of the same name as the function. Functions are declared with the DIM statement:

```
DIM funct <GOSUB 1010>
...
1010
    ...
    funct=...
    RETURN
```

As an option functions can be called with a list of parameters enclosed within parenthesis. When called the associated subroutine (at line 1010 in the above example) is executed. The return value is assigned (by the subroutine) to the associated integer variable and can be later used in any integer expression.

Code examples:

```
var=funct()      ' execute function funct() and then assign
                 ' the returned value to variable var
```

```
var=funct(5*3)  ' execute function funct() with parameter 15 and
                 ' then assign the returned value to variable var
```

```
var=funct      ' assign the last returned value of funct() to var
```

All function parameters and local (temporary) variables are declared with the LOCAL statement at the beginning of the associated subroutine. The LOCAL statement must be the first statement of the subroutine and may not be used anywhere else in the subroutine.

```
LOCAL {V|S$} [, {V2|S2$} [, ...]]
```

The LOCAL statement has the same syntax as the DIM statement with the exception that only simple integer variables (not arrays) or default size string variables (without a size specification) may be declared. The function arguments are listed first in the declaration followed by the local variables.

WARNING: The local variable names are unique within the program scope and should not be used outside the subroutine. Value of a local variable is not defined outside the subroutine.

Local variables count to the total number of variables.

When calling a function with N arguments the function arguments are stored into the first N local variables declared with LOCAL, the remaining variables are initialized to null values. If the LOCAL statement declares less than N variables, only the first arguments are stored and the remaining arguments are discarded. Any expressions can be passed to a function as arguments and the number of arguments is not limited. The number of arguments can be retrieved from the system variable _ARG.

Code example:

```
DIM circum <GOSUB 1010>      'function computing circumference
                             ' given radius
....
c1=circum(3)
....
....
1010
LOCAL radius
IF _ARG <>1 THEN SYSLOG "Bad number of arguments for function
circum!"
circum = (2*314*radius)/100 'compute circumference
RETURN
....
```

Note: During a user defined function execution no events can be captured and handled, therefore the user function subroutines should be kept short. If this is not possible GOSUB-RETURN should be used instead of a function.

WARNING: The maximal nesting for recursive calculations is only 10. Therefore use of recursive functions is not recommended.

3.10 I/O stream functions

The BCL language supports a variety of real world interfaces and protocols for input and output. The same function set is used throughout but the functionality differs slightly depending on the protocol.

Simplified procedure for an I/O stream operation consists of three phases:

1. Opening of the I/O stream using the `OPEN` function with the syntax: `OPEN S$ AS H` where
 - `E$` is a string expression which determines the protocol and sets appropriate parameters (for details see descriptions of individual protocols below)
 - `H` is the handle number (integer). For most protocols the numbers 0,...,7 are allowed. An exception is the TCP protocol where only numbers 0,...,5 are allowed. Handle numbers are common for all protocols and the same handle cannot be opened for two different streams at the same time.
2. Using the stream with `WRITE`, `READ`, `SEEK` etc. (list of available functions depends on particular protocol) Handle number of the stream is given to functions to determine the stream. Multiple I/O commands can be used in this phase before closing the stream.
3. Closing the stream using `CLOSE H` command, where `H` is the handle number.

After the `CLOSE` the handle is available again for use with any I/O stream.

Not all peripherals/protocols mentioned in this chapter are supported on all BCL devices. Check the specific device documentation for more information about the protocols supported.

Besides `OPEN`, the following commands are common for all I/O operations:

`CLOSE H`

Closes the file or stream with handle `H`.

`WRITE H, E$, E0`

Writes `E0` bytes from `E$` into the stream `H`.

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

To write a binary zero, use an empty string `S$` and `E0 = 1`.

Note: The write call is blocking and it does not return before the data is written to the output (or the output buffer).

`READ H, S$ [, E0 [, E$]]`

Reads from the stream `H` into the string variable `S$`.

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

Without the optional parameters, files are read in "binary" mode. The read command reads all currently available bytes up to the size of the destination variable. The number of bytes read is returned by the `LASTLEN(H)` function.

If the optional parameter `E0` is 0, the file (flash or USB file) or TCP/serial stream is read in "line" mode, every read returns either nothing or a complete text line of the input with CR/LF being stripped off. If an empty line is on input, `LASTLEN` returns 0 and `READ` returns an empty string in `S$`. If there's an incomplete line or no data on input, `LASTLEN` returns 0 and `READ` does not read anything and **does not change** `S$`. This can be easily used e.g. for parsing HTTP headers. See the below example:

```

10
   s$=""
   READ 0, s$, 0
   IF LASTLEN(0)>0 THEN
       SYSLOG s$           ' line read
   ELSE
       IF LEN(s$)=0 THEN SYSLOG "--- empty line ---"
   ENDIF
   GOTO 10
    
```

For streams (COM, TCP) the parameter `E0` can be also a positive integer. In that case the `READ` function either returns immediately with no data if there is not enough data in the receive buffers, or returns the exact number of bytes required if the given buffer can be completely filled, or no additional data has been received from the stream for at least `E0` milliseconds.

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

The second optional parameter `E$` defines a "match" string. With the `E$` the `READ` function skips all input data until an exact match with `E$` is found and then starts reading from the very first character after the matching string. This functionality is ideally suited to reading data after a certain tag in XML or web data.

If the `E$` is given but not found, the function returns immediately and all subsequent calls to `LASTLEN(H)` return 0.

MEDIATYPE(H)

Returns the media type number if the stream `H` has been opened, or 0 if `OPEN` has failed or the file or stream is already closed.

Value	Media type
3	USB - reading
4	USB - writing
6	TCP
7	UDP
8	Serial port (COM)
9	Flash read
10	Flash write
11	Flash append
13	Setup
14	Wiegand protocol
17	Audio
18	USB - append

LASTLEN(H)

Returns the number of bytes transferred in the last read/write operation on the stream `H`.

It is also used as an error code or EOF (End Of File) mark, in that case `LASTLEN` returns -1.

For UDP reception negative return value means new data available for reading (new packet has arrived). The absolute value gives the number of bytes available.

FILESIZE(H)

Returns the file size of file/stream `HANDLE` or returns the number of entries in a USB directory.

For serial streams (COM, TCP) returns number of bytes available for reading in the incoming FIFO.

For audio streams (AUD) returns the number of free bytes in the audio-decoding buffer.

3.10.1 The UDP network protocol

A UDP stream for both sending and receiving can be opened using:

```
OPEN "UDP:<IP address or DNS address>:<port number>" AS H
```

The given IP address should be 0.0.0.0 for a listening socket.

The following example opens a listening UDP socket on port 1000:

```
OPEN "UDP:0.0.0.0:1000" AS 3
```

The IP address can be omitted:

```
OPEN "UDP::1000" AS 3
```

RMTPORT(H)

Returns the source port of the last UDP packet received from stream H or 0 if not applicable.

RMTHOST\$(H)

Returns the source IP address (as a string) of the last UDP packet received from the stream H, or an empty string.

WARNING: `RMTPORT` and `RMTHOST$` functions have to be called **before** `READ` is called (see below).

3.10.1.1 Receiving UDP packets

When in receiving mode, the `LASTLEN(H)` may return both positive or negative value. A negative return value indicates there is new data available for reading. The absolute value is the number of bytes available for reading. After reading, the return value of `LASTLEN` is positive (number of bytes read) unless new data have arrived.

When reading UDP packets, `LASTLEN`, `RMTHOST` and `RMTPORT$` should be used before `READ` in order to determine packet size, like in the following example:

```
...
OPEN "UDP:0.0.0.0:1234" AS 1
OPEN "COM:..." AS 2
10
l=LASTLEN(1)
port=RMTPORT(1)
ip$=RMTHOST$(1)
IF l<0 THEN
    READ 1,buf$
    WRITE 2, buf$, -1
ENDIF
GOTO 10
...
```

The reason is that a new packet may arrive between `READ` and `LASTLEN` and the information about data length could be lost. See following modification of previous example:

```
...
READ 1, buf$
' at this moment, new packet can arrive
' in such case lastlen(1) will return negated size of the new
packet
' and the information about the size of the previous packet is lost!!
len = LASTLEN(1)
IF len>0 THEN WRITE 2, buf$, len
...
```

See also example 8.4, page 50.

3.10.1.2 Sending UDP packets

The syntax of the `WRITE` command is extended for the UDP protocol:

WRITE H, E\$, E0, E2\$, E1

Sends E0 bytes from E\$ to the destination address E2\$ (an IP address or a DNS address) port E1.

Example:

```
OPEN "UDP:0.0.0.0:5555" as 4
WRITE 4, "hello", 5, "192.168.2.255", 5555
CLOSE 4
```

It is also possible to send a UDP packet to multiple addresses with one command using a two dimensional integer array of size (number_of_addresses-1,1) filled with pairs of IP addresses and port numbers.

```
OPEN "UDP:..." AS H
WRITE H, BUF$, LEN, ADR
```

```
DIM RECIP(2,1)      'two dimensional field of recipients
...
BUF$=...
LEN=...
RECIP(0,0)=VAL("192.168.15.3")
RECIP(0,1)=200
RECIP(1,0)=VAL("192.168.13.2")
RECIP(1,1)=300
RECIP(2,0)=VAL("192.168.13.2")
RECIP(2,1)=400
OPEN "UDP:" AS 1
WRITE 1, BUF$, LEN, RECIP 'Sends data to 192.168.15.3 port 200,
                           '192.168.13.2 port 300, 192.168.13.2 port 400
```

If port equals 0 for some IP address, nothing is sent to this IP address, if IP address equals 0, broadcast is sent.

3.10.2 The TCP network protocol

A TCP socket, both passive and active connections, can be opened using

```
OPEN "TCP:<IP address or DNS address>:<port number>" AS H
```

The given IP address should be 0.0.0.0 for a listening socket. Note that for TCP connections associated with the handle number 0 a large (4KiB) receiving buffer is used. Otherwise only a 512B receiving buffer is used.

Examples:

```
OPEN "TCP:0.0.0.0:1000" AS 3
```

opens listening TCP connection on port 1000 as handle 3

```
OPEN "TCP:192.168.11.99:1000" AS 3
```

opens an active connection (session will be established) to IP address 192.168.11.99, port 1000, as handle 3

RMTPORT(H)

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

RMTHOST\$(H)

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

CONNECTED(H)

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

FILESIZE(H)

Returns number of bytes in the incoming FIFO of the stream H available for reading.

See also example 8.3 on page 49.

3.10.3 Audio interface (audio devices only)

BCL provides a high-level interface for decoding and encoding¹ audio in variety of audio formats. The audio interface is accessed through a file handle using the `OPEN`, `READ`, `WRITE`, `CLOSE`, `LASTLEN` and `FILESIZE` calls. The audio mode, the quality and the data format are specified within the string passed to the `OPEN` call.

3.10.3.1 Opening audio

Audio input/output can be opened with the following syntax
`OPEN "AUD:MODE, FLAGS, QUALITY, DELAY" as H`

MODE value	mode
1	MP3 decoding
2	MP3 encoding
3	full-duplex PCM (16-bit signed, mono) see the FLAG bits and chapter 3.10.3.3 below
4	full-duplex μ -Law
5	full-duplex A-Law
6-8	reserved
9	PCM stereo decoding (16-bit signed, stereo, left channel first) see the FLAG bits and chapter 3.10.3.3 below

FLAGS bit	value/meaning	
0	0	read/write raw data
	1	read/write RTP frames
1	0	start playback immediately
	1	delayed playback
2	0	delay in milliseconds
	1	delay in bytes
3	reserved, always 0	
4	0	no rebuffering
	1	automatic rebuffering on buffer underrun (MP3 decoding only) in RTP mode resync to a new sequence number on buffer underrun
5	0	PCM data in big-endian format (Msb first)
	1	PCM data in little-endian format (Lsb first)

QUALITY :

Is ignored for MP3 decoding.

For full-duplex modes (modes 3, 4 and 5) specifies sampling rate in kHz - 8 or 24

The values for MP3 encoding are documented in the following table:

¹ Not available on Exstreamer devices.

Bit	Meaning												
0..3	Interpreted as a 4-bit integer <table border="1" style="margin-left: 20px;"> <tr><td>0</td><td>MPEG2/22.05kHz</td></tr> <tr><td>1</td><td>MPEG1/44.1kHz (MP3)</td></tr> <tr><td>2</td><td>MPEG2/24kHz</td></tr> <tr><td>3</td><td>MPEG1/48kHz (MP3)</td></tr> <tr><td>4</td><td>MPEG2/16kHz</td></tr> <tr><td>5</td><td>MPEG1/32kHz (MP3)</td></tr> </table>	0	MPEG2/22.05kHz	1	MPEG1/44.1kHz (MP3)	2	MPEG2/24kHz	3	MPEG1/48kHz (MP3)	4	MPEG2/16kHz	5	MPEG1/32kHz (MP3)
0	MPEG2/22.05kHz												
1	MPEG1/44.1kHz (MP3)												
2	MPEG2/24kHz												
3	MPEG1/48kHz (MP3)												
4	MPEG2/16kHz												
5	MPEG1/32kHz (MP3)												
4..7	encoding quality 0 to 7 (0 lowest, 7 highest)												
8	1= disable CRC, 0= enable CRC												
9	MS-Stereo enc 1=disable empty, 0=enable												
10	bitreservoir 1=kept, 0=used												
12-13	Emphasis 0=none, 1=50/15us, 3=CCITT J.17												
14	0=stream is copy, 1=stream is original												
15	stream is copyright protected 0=yes, 1=no												

3.10.3.2 Raw data mode

Format of the data read or written from the audio interface depends on the audio mode and flags. In raw mode data are transferred to or from the codec as they are and the application must care about the buffer handling and correct data formatting (e.g. avoid sending broken MP3 frames to the codec).

READ H, S\$ [, E]

When reading from the audio interface (audio encoding) an optional size parameter *E* can be specified. Then the READ function returns either *E* bytes, or 0 if there's not enough data available. This can be used for constant bitrate data stream of uncompressed audio.

If *E* is not specified, the READ function reads all available data, up to the size of *S\$*.

FILESIZE(H)

Returns the number of free bytes (free space for writing) in the audio-decoding buffer.

See also example 8.1 on page 49 and example 8.2 on page 49.

3.10.3.3 PCM audio data

By default, 16-bit PCM data (modes 3 and 9 in `OPEN`) are expected to be in the **big-endian** (Motorola, Msb first) format. To swap the endianness to the **little-endian** (Intel, Lsb first) format, set the bit 5 of the `FLAGS` parameter you pass to the `OPEN` function.

Encoding (reading from the audio device) and decoding (writing into the audio device) always uses the same endianness. It's not possible e.g. to encode in big-endian and decode in little-endian. If this is required, an extra processing must be performed by the application.

Note: The endianness bit also affects the RTP payload type accepted/generated by the audio interface. By default all PCM RTP payload types are big-endian. With the `FLAGS` bit 5 set, a different set of payload types is accepted/generated in PCM modes. See section 3.10.3.4 below for more details.

3.10.3.4 RTP data mode

READ H, SS

In RTP mode the audio interface provides complete RTP frames to the application including correct time stamp and sequence number. The `READ` function returns either one complete RTP frame or no data (`LASTLEN` returns 0). The application then just sends the data to the network without any further processing.

The same mechanism is used in writing, the BCL application reads an RTP frame from a UDP socket and writes it as it is to the audio interface. No additional processing is needed. The audio interface automatically handles buffer overruns and underruns, duplicates the data if necessary etc.

This significantly reduces the complexity of the application and increases the processing speed.

The audio interface can be open for **one RTP stream at a time**. If more complex application receiving multiple RTP streams (e.g. with a different payload type or on a different port) is required, non-relevant frames **must be filtered** out by the application.

In RTP mode the `WRITE` function ignores frames with payload type not matching the audio type passed to the `OPEN` function. E.g. if audio was open to receive MP3, only payload type 14 (MPA) will be accepted and all other payload types (e.g. 4 for PCMA) will be ignored. `LASTLEN` returns a negative value in that case to signalize an error condition.

See also example 8.4 on page 50.

3.10.3.5 RTP payload types

The following table shows the defined RTP payload types and the according audio mode, quality and flags to be used with the `OPEN` function.

When writing to the audio interface (decoding audio) only the payload type associated with the particular mode is accepted, RTP frames with a different payload type are ignored.

RTP payload type	Audio Format	Mode	Quality	Flags
0	μ-Law, 8bit, mono, 8kHz	4	8	-
8	A-Law, 8bit, mono, 8kHz	5	8	-
10	PCM 16bit, MSB first, signed, 44.1kHz stereo, left channel first	9	44	bit5=0
14	MPEG audio	1,2	-	-
96	PCM, 16bit, MSB first, signed, 8kHz mono	3	8	bit5=0
97	μ-Law, 8bit, mono, 24kHz	4	24	-
98	A-Law, 8bit, mono, 24kHz	5	24	-
99	PCM, 16bit, MSB first, signed, 24kHz mono	3	24	bit5=0
100	reserved			
101	reserved			
102	reserved			
103	PCM 16bit, MSB first, signed, 48kHz stereo, left channel first	9	48	bit5=0
104	PCM, 16bit, LSB first, signed, 8kHz mono	3	8	bit5=1
105	PCM, 16bit, LSB first, signed, 24kHz mono	3	24	bit5=1
106	reserved			
107	PCM 16bit, LSB first, signed, 44.1 kHz stereo, left channel first	9	44	bit5=1
108	PCM 16bit, LSB first, signed, 48kHz stereo, left channel first	9	48	bit5=1

3.10.3.6 Reading audio status

Status values for an audio device can be obtained by reading from a "negative offset" and then using the LASTLEN(H) function. INDEX values and corresponding return values for LASTLEN are listed in the table below.

Syntax:

READ H, BUFFER, -INDEX

The BUFFER parameter is ignored in this case.

Table 1: Reading the audio device status

INDEX	LASTLEN RETURNS
1	left channel input audio peak level
2	right channel input audio peak level
3	left channel output audio peak level
4	right channel output audio peak level
5	bitrate in kbits/s
6	current output buffer level
7	average output buffer level (since last cca. 5 seconds)

3.10.3.7 Setting audio parameters

Audio device parameters can be set using the `WRITE` command with the following syntax
`WRITE H, "VALUE", -PARAM_INDEX`

`PARAM_INDEX` determines the parameter to be set and `VALUE` is the value to be set. See the table below.

PARAM_INDEX	PARAMETER	Possible values	
1	mic gain	0..15 (in 1.5dB steps, starting at 21dB)	
2	AD gain	0..15 (in 1.5dB steps, starting at -3dB)	
3	input source	1	line in
		2	microphone
		3	autodetect
		4	SPDIF optical
		5	SPDIF coaxial
		6	mix line in with playback
4	input mode	0	mono
		1	stereo
5	output mode	0	stereo
		1	mono
		2	bridge
6	output volume gain	in dB	
7	loudness	0..20	
8	balance	-10..10 (-10=left, 10=right)	
9	treble	-10..10	
10	bass	-10..10	
11	volume type	0	5% steps (default)
		1	dB
12	volume	(see the previous row)	
13	output mixer	bits 0..6: Rec. input	linear scale 0:off, 64: 100%, 127:200%
		bits 8-14: Playback	linear scale 0:off, 64:100%, 127:200%

3.10.3.8 Flushing decode buffer

Decode buffer can be flushed by "writing" zero bytes, i.e.:
`WRITE H, "", 0`

3.10.3.9 Closing audio

An optional parameter is available to the `CLOSE` command.

`CLOSE H [,E]`

By default `CLOSE` causes an immediate close of the audio device discarding the content of the internal playback buffer. If called with an optional non-zero parameter `E`, the program is blocked until the whole content of the playback buffer is played and then the audio device is closed.

3.10.4 Serial port

Serial ports can be opened using:

```
OPEN "COM:Baudrate,Parity,Data,Stopbits,FlowControl:PortNumber" AS X
```

where `Baudrate`, `Parity`, `Data`, `Stopbits`, `FlowControl` and `PortNumber` are integer parameters.

Possible values for `Baudrate` are:

230400, 115200, 76800, 57600, 38400, 19200, 9600, 4800, 2400, 1200, 600, 300

Possible values for `Parity` are:

N, O, E

Possible values for `Data` are:

7, 8

Possible values for `Stopbits` are

1, 2

Possible values for `FlowControl` are

Value	Type of flow control
NON	none
SFW	Software flow control (XON, XOFF)
HDW	Hardware flow control (RTS,CTS)
422	RS422
485	RS485

`PortNumber` is the number of the serial port, usually 1 or 2. Depending on the hardware configuration, various ports are supported. Please refer to the specific product documentation for details.

Serial configuration on the Barionet is ignored and the configuration is taken from the system configuration. To open serial port on the Barionet use the following `OPEN` command:

```
OPEN "COM::1" AS X
```

`FILESIZE(H)`

Returns number of bytes available for reading in the incoming FIFO.

See also example 8.5, page 50.

3.10.5 SETUP

Non volatile parameters (e.g. configuration) are held in EEPROM. When the device starts up these values are automatically copied from EEPROM and stored in RAM in the Setup table. The Setup table can be accessed by opening a special file in the following way:

```
OPEN "STP:<offset>" AS 3
```

where parameter `offset` specifies an offset (starting from 0) in the Setup table.

The BCL interpreter allows to read the Setup table in 256 byte blocks (starting from the given offset). Configuration larger than 256 bytes can be read/written by subsequent accessing 256 byte blocks. But only one such file can be opened at a time.

The read and write operations use strings for binary operations, so a full 256 bytes is read from or written to the Setup table for each read or write operation.

If the WRITE call has been used on the file, the new modified Setup will be saved into EEPROM upon CLOSE. Note that the complete Setup table is saved into EEPROM not just the 256 byte block currently open.

Example for accessing a 512 byte Setup table using handle 3

```
DIM set$
OPEN "STP:0" AS 3
READ 3,set$
.. read and write functions
CLOSE 3
OPEN "STP:256" AS 3
READ 3,set$
.. read and write functions
CLOSE 3
```

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 parameters.

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

3.10.6 The USB filesystem (not supported on Barionet)

Files and directories on the local USB disk can be accessed . FAT12 and FAT16 filesystems with long filename extension are supported. Elements of directory paths are separated with forward slashes ("/"). A full path starts with a slash.

3.10.6.1 File access

To open a file, use the following OPEN command:

```
OPEN "C_R:usb://<filename>" AS H – open a file in read mode
OPEN "C_W:usb://<filename>" AS H – open a file in write mode
OPEN "C_A:usb://<filename>" AS H – open a file in append mode
```

Where filename is the full path name of the file to be accessed (starting with slash).

Example:

```
OPEN "C_R:usb:///music/Rolling_Stones/song01.mp3" AS 1
```

If a file is open for writing and it does not exist, it is automatically created. Already existing files are truncated to zero (C_W write mode) or newly written data are appended to the end of the file (C_A append mode).

Once the file is open it can be read and written using READ and WRITE calls. Besides them the following calls are available.

FILEPOS (H)

Returns the current file position for the file handle H (offset in bytes from beginning of the file).

SEEK H, E

Sets the current file position of the file H to the position E (in bytes from the beginning of the file).

FILESIZE (H)

Returns the size of the file `H` in bytes.

`DELETE S$`

Deletes a file (not directory!) with the name `S$` on the USB filesystem.

Example:

```
DELETE "usb:///dir/filename.ext"
```

Deletes the file `filename.ext` in directory `dir`.

WARNING: The file has to be closed before deleting.

See also a program example 8.1 on page 49 how to play a file.

3.10.6.2 Directory access

Use the following command to read a directory:

```
OPEN "C_R:usb://<filename>" AS H - open a directory in read mode
```

Where `filename` is the full path name of the file to be accessed (starting with a slash).

In the directory mode each `READ` call returns a descriptor of the next directory entry in `S$`. The descriptor starts with the filename in 8.3 format followed by 16-bit flag. The appropriate short name is returned for files with long names. The first two entries returned are `."` for the current directory and `.."` for the parent directory.

The directory entry flags can be obtained with the following `MIDGET` command:

```
flag=MIDGET(S$,14,2)
```

The value of `flag` is 1 for files and 2 for directories.

If the directory listing is already at the end, `LASTLEN` returns -1 (EOF condition).

The `SEEK` call sets the current directory pointer to point to the given entry (starting from 0).
E.g. `SEEK H,0` rewinds the directory read.

The `FILEPOS` function returns the current position of the directory pointer.

The `FILESIZE` function returns the number of directory entries including the `."` and `.."`

A directory listing example:

```

DIM dir$
DIM _Mb$(20)
DIM fl

dir$="/music"
OPEN "C_R:usb:///"+dir$ AS 1
SYSLOG "Directory listing of "+dir$
100
READ 1,_Mb$
IF LASTLEN(1)=-1 THEN GOTO 200
fl=MIDGET(_Mb$,14,2)
IF fl=1 THEN type$="file" ELSE type$="directory"
SYSLOG _Mb$+" "+type$
GOTO 100
200
CLOSE 1
END
    
```

3.10.7 The local flash filesystem

3.10.7.1 Reading files

Files in FLASH memory of the device (stored in `.cob` files) can be read using the following `OPEN` command:

```
OPEN "F_R:<filename>" AS H
```

Example:

open file "testfile.txt" for reading with handle I

```
OPEN "F_R:testfile.txt" AS I
```

Except the `READ` command, the following commands are available:

FILEPOS (H)

Returns the current file position for the FFS file `H`.

SEEK H, E

Sets the current file position of the FFS file `H` to the position `E` (in bytes from the beginning of the file).

FILESIZE (H)

Returns the size of flash file `H` in bytes.

3.10.7.2 Writing files (Barionet only)

In addition to the read FLASH-file access the Barionet offers a write support with the following limitations. The file has to exist in a `.cob` file in FLASH memory of the Barionet and start with a special header "`<*>CRLF`" (ASCII: 60, 42, 62, 13,10). The size of the file is fixed and can not be changed. Only text data can be stored in the file since the NULL character (`\0`) is recognized as an end-of-file mark.

The file header and the EOF mark are transparent for `READ` operations as well as for the built in webserver. This way the HTML/DHTML pages can be modified on the fly or e.g. system logs be written.

`OPEN "F_W:<filename>" AS H` – opens file in write mode

`OPEN "F_A:<filename>" AS H` – opens file in append mode

In write mode the file size is truncated to zero (EOF marker is moved to the start of the file) and the file position pointer is set to the beginning of the file. In append mode the original content of the file is preserved and the file position pointer is set to the end of the file.

Each `WRITE` call moves the EOF mark appropriately, but maximum number of bytes of the original file size can be written (the file can not be enlarged within the `.cob` file).

The `FILEPOS`, `SEEK` and `FILESIZE` functions can be used to seek within the file and to obtain the current file size.

3.10.8 The Wiegand reader (Barionet only)

It is possible to access up to two connected Wiegand readers with the `RDR` filetype:

```
OPEN "RDR:" AS H
```

The Wiegand interface is read-only. The `READ` command reads raw bit output of the Wiegand reader and returns the data in the following format:

B0		B1	B2	B3	B4	...
b7	b6..b0	b7...b0	b7...b0	b7...b0	b7...b0	...
adr	len	Data b0...b7	Data b8...b15	Data b16...b23	Data b24...b31	...

The byte 0 contains address `adr` of the reader in the highest bit (0-first reader, 1-second reader) and length `len` in **bits** of the raw data returned by the reader (the lower 7 bits).

Byte 1 and following contain the raw bit output of the reader in the big-endian format (see the above figure). Bit 0 of the raw output is in bit 7 of the first data byte and bit 7 of the raw output is stored in bit 0 of the first data byte.

The `READ` call is non-blocking and returns either the complete data from the reader or no data if there is no input. The `LASTLEN` function returns 0 in the latter case. It's recommended to use similar command sequence as in the following example:

```

DIM rdr$(10)                                     ' for 26-bit Wiegand
DIM bits
...
OPEN "RDR:" AS 1
...
10                                               ' the main loop
READ 1, rdr$
IF LASTLEN(1)=0 THEN GOTO 10                     ' no input, wait for data
bits=AND(127, MIDGET(rdr$,1,1))                 ' Wiegand type
IF AND(128,MIDGET(rdr$,1,1)) THEN
    ' the second reader
ELSE
    ' the first reader
ENDIF
' process the data
GOTO 10                                         ' jump to the main loop
    
```

3.10.8.1 26-bit Wiegand reader

For example if a 26-bit Wiegand reader is connected, the first byte of the data returned by the `READ` call will be `0x1A` for the first reader (26 plus the top most bit not set) and `0x9A` for the the second reader (26 plus the top most bit set).

The 26-bit Wiegand reader sends first the parity bit for the first 12 bits, followed by the second 12 bits and a parity bit for the latter 12 bits.

B1		B2		B3	B4	
b7	b6..b0	b7...b3	b2...b0	b7...b0	b7	b6
parity	12 bits		12 bits		parity	

An example how to read a 26-bit Wiegand reader:

```

DIM b1,b2,b3,id
DIM rdr$(10)                                     ' for 26-bit Wiegand

OPEN "RDR:" AS 1

10
READ 1, rdr$
IF LASTLEN(1)=0 THEN GOTO 10                     ' no input, wait for data
    
```

```

bits=AND(127, MIDGET(rdr$,1,1))      ' Wiegand type
if bits<>26 THEN
    SYSLOG "Unsupported card, "+STR$(bits)+"bits"
    GOTO 10
ENDIF

b1=AND(255,SHL(MIDGET(rdr$,2,1),1))+SHR(MIDGET(rdr$,3,1),7)
b2=AND(255,SHL(MIDGET(rdr$,3,1),1))+SHR(MIDGET(rdr$,4,1),7)
b3=AND(255,SHL(MIDGET(rdr$,4,1),1))+SHR(MIDGET(rdr$,5,1),7)
id=b1*65536+b2*256+b3                ' store 24bit wiegand ID

IF AND(128,MIDGET(rdr$,1,1)) THEN
    SYSLOG "Second reader: ID="+STR$(id)
ELSE
    SYSLOG "First reader: ID="+STR$(id)
ENDIF
GOTO 10                                ' jump to the main loop
    
```

See also the example program 8.6 on page 52.

3.11 Direct hardware access

BCL offers a set of functions for accessing hardware dependent inputs and outputs (e.g. digital inputs and outputs, analog inputs and outputs, relays, etc.). The access is provided through a set of general registers, their meaning is platform specific. Please see the product specific documentation for more details about IO register mapping.

IOCTL E0, E

Sets the I/O register E0 to the value E.

This function is hardware dependent.

Code example:

```
IOCTL 1,1
```

activates the first digital output I on the Barionet.

IOSTATE (E0)

Returns the current state of I/O register E0.

Code example:

```
INP1=IOSTATE(201)
```

stores the state of the digital input I on the Barionet into the variable INP1.

FIXME: IO registers, modbus etc.

3.12 Diagnostic functions

PING (E\$, E)

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

Code example:

```
IP$="192.168.2.18"
```

```
rtime=PING(IP$, 50)
```

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

TRAP E\$, N,

Send Ns as SNMP trap to IP or DNS address E\$.

SYSLOG E\$ [,E]

Sends the E\$ as a UDP message to Syslog (port 514). The optional parameter E specifies a debugging level. The maximum length of the message is 255 characters.

Code example:

```
SYSLOG "ALARM"
```

4 Interpreter information

4.1 Execution speed

The execution speed of BCL programs depends on the specific hardware and firmware used, it is typically more than 5000 tokens per second. In other words: each instruction needs in average less than 0.2 milliseconds to execute. For complex or time critical applications the `LOCK` command can be used to lock (actually setting to low priority) other CPU tasks and only run the BCL interpreter.

4.2 Runtime environment limitations

The runtime environment has size constraints resulting from the hardware platform used which should be considered when writing the BCL code.

In the current version of the BCL the following limitations exist:

- maximum number of numeric labels **1000**
- range of numeric labels **1-65535**
- maximal `FOR-NEXT` nesting **25**
- maximal `GOSUB-RETURN` nesting **25**
- maximal recursive nesting (amount of brackets) **10**
- maximum number of variables of type integer **96**
- maximum number of variables of type string **64**
- length of variable's names **unlimited**
- number of significant characters in variable names **5**
- maximal dimensions of arrays of type integer **2**
- size of integer variables **32 bit**
- default size of variables of type string (if not specified by the `DIM` statement) **256 bytes**
- maximal number of open files/streams **8 (only 0-5 for TCP)**
- default size of buffers for files/streams **512 bytes**

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

4.3 System variables

Several predefined values are available:

Variable name	Value
<code>_ARG_</code>	holds the number of arguments given to the function, see section 3.9 on page 20
<code>_CGI_</code>	used for CGI request handling, see section 5.3 on page 41
<code>_DTS_</code>	time counter, see section 3.4.8 on page 13
<code>_ERL_</code>	line number of the last error occurred, see section 3.4.8 on page 13
<code>_ERR_</code>	error code of the last error, see section 3.4.8 on page 13
<code>_TMR_</code>	array of time counters, see section 3.4.8 on page 13

5 WEB interface

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

5.1 HTML tags

Special dynamic marks "&LBAS" (see the product specific documentation for more details about dynamic marks) allows the BCL program to interact with the user through web pages. It is possible to read content of a variable and to call a BCL subroutine.

While processing the request the internal webserver parses the dynamic marks and substitutes them with their values (calls a subroutine if needed) **in the order they appear in the HTML file.**

To enable dynamic tags, the following special tag must be present at the beginning of the HTML file:

```
&L(0, "*", 1);
```

5.1.1 Displaying variables in webpages

Use the following dynamic marks to insert the current value of any BCL variable in dynamic HTML pages:

```
&LBAS(1, "%ld", V);
&LBAS(1, "%lu", V);
&LBAS(1, "%fs", S$);
```

where V and S\$ are integer and string variables respectively.

Syntax of the LBAS tag is the following:

```
&LBAS(1, <format_string>, <variable>);
```

where format_string corresponds to SPRINTF formatting (see chapter 3.5.1.1 on page 17) and variable is the name of the variable to be printed.

Please note that for string variables the format string must be always "%fs".

The functions will return "[NO_VAR]" if there is not any variable of that name used in the program or the BCL interpreter has not started yet.

Following example displays the current uptime in seconds (the _DTS_ variable) on the web page:

HTML file:

```
&L(0, "*", 1);
<html>
<head>
</head>
<body>
Time since reboot: &LBAS(1, "%lu", _DTS_);seconds.
</body>
</html>
```

5.1.2 Calling a subroutine from a webpage

For more complex output the BCL program can be directly called in order to create a dynamic content of a web page. The following dynamic mark is used:

```
&LBAS(2,"<string>",0);
```

which triggers the following set of actions:

1. The `_CGI_$` variable is loaded with the value `string`, the webserver is blocked and the BCL interpreter is called.
2. The program polls the `_CGI_$` variable or contains ON CGI handler and according to the content of the `_CGI_$` performs a specific action (e.g. calls a particular subroutine).
3. The program can directly output to the HTTP stream by writing into handle `-1`.
4. When the processing of the CGI request is finished, the `_CGI_$` variable **must be set to an empty string**.
5. The webserver detects that the `_CGI_$` has been cleared and continues with further processing.

Note: When using `&LBAS(2,...)`; dynamic mark, the speed of generating the HTML page depends on the speed of the handling subroutine, therefore the subroutine should be kept as fast as possible.

Note: Other (hardware dependent) dynamic marks may be available on certain hardware. See the respective product documentation.

5.2 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, how it could be used in the HTML code:

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

where "BAS.cgi" is the name of the interface script, "&" is the 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>
```

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

5.3 CGI handling in the BCL

To handle web requests directly in the BCL program, the "basic.cgi" CGI script can be used.

All parameters passed to the script after "?" are accessible in a special string variable `_CGI_$` from the BCL program. This variable must be declared and set to the empty string before use.

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

After reading the value of `_CGI_$` and before sending the reply, clear the `_CGI_$` variable and call `DELAY 0`:

```
_CGI_$=""
DELAY 0
```

The reply to the browser can then be sent in three ways:

1. Using the special handle `-1`:

```
WRITE -1,E$,0
```

The reply should contain the HTTP header in this case.

Instead of "closing" the handle, set the `_CGI_$` variable to "*" (asterisk) in order to finish sending the reply.

Code example (assume an integer in `_CGI_$` and return its value increased by one):

```
DIM TEMP$(256)
1 ON CGI GOSUB 2
GOTO 1
END
2
TEMP$=STR$(1+VAL(_CGI_$))
_CGI_$=""
DELAY 0
WRITE -1,"HTTP/1.0 200 OK\r\nContent-type:
text/plain\r\n\r\n",0
WRITE -1,TEMP$,0
_CGI_$="*"
RETURN
```

2. Set `_CGI_$` to asterisk "*" followed by a filename of a file in the flash memory. That file will then be sent as the reply.

Code example:

```
DIM TEMP$(256)
1 ON CGI GOSUB 2
GOTO 1
END

2 _CGI_$=""
DELAY 0
_CGI_$="*index.html"
RETURN
```

3. Assign the reply to `_CGI_$` variable.

In this case, the HTTP headers are created by the BCL automatically.

Code example:

```
DIM TEMP$(256)
1 ON CGI GOSUB 2
```

```
GOTO 1
END
2 TEMP$=STR$(1+VAL(_CGI_$))
_CGI_$=""
DELAY 0
_CGI_$= TEMP$
RETURN
```

6 Preprocessor

The BCL language offers simple preprocessor directives to include files and create macros. The preprocessor built in the tokenizer processes the BCL source before the tokenization takes place.

6.1 Preprocessor directives

Preprocessor directives must be terminated with CRLF or a comment. Macro handling is case sensitive and is not recursive. The following directive are available:

#define *source target* replaces the "source" macro template with the "target" macro text. Macro parameters named from #1 to #9 can be used (see the examples below). The preprocessor scans the code line-by-line, finds the pieces of code matching the "source" template and replaces them according to the macro definition.

#include *file_name.bcl* appends the "file_name.bcl" module from the BCL subdirectory to the end of the currently collected BAS file. Usage of the same "library" modules in different projects is possible this way. However name and label conflicts must be avoided across different modules, because all names and labels will be global in the final BAS file.

6.2 Using the preprocessor

In order to use the preprocessor, the BCL source as well as all the files to be included must be placed in the "BCL" subdirectory of the main project directory. All files in the "BCL" subdirectory (including the main project file) must be named with the .bcl extension instead of the .bas extension.

To tokenize the project, use the following command:

```
tokenizer.exe <file_name.bcl> [-<debug_level>]
```

where "file_name.bcl" is the name of the main project file.

The preprocessor will read the "file_name.bcl" from the "BCL" subdirectory, process it and output the result into "file_name.bas" in the main project directory. The processing comprises substitution of macros and including other project files. Files included with the #include directive are searched in the "BCL" directory and added to the end of the currently collected "file_name.bas". This process recurses to all included files.

After preprocessing the resulting "file_name.bas" is converted into "file_name.tok"

WARNING: The original project files will NOT be included in the COB-file.

Code examples:

```

#include pr400.bcl 'module with subroutine with label
400
#include pr600.bcl 'module with subroutine with label
600
#define s#1[#2]= midset s#1,#2,1, 'macros for using
string
#define s#1[#2] midget(s#1,#2,1) 's* as array of chars
#define SYSTIME _TMR_(0) 'replaces the old SYSTIME
function
#define TRUE -1 'replaces logical constant TRUE with a
value
#define FALSE 0 'replaces logical constant FALSE with
a value
#define FACTORIAL 1010 'replaces a text LABEL with a
real number

```

7 Debugging

The Barix BCL interpreter allows debugging of programs using the syslog protocol¹, all warnings and error messages are sent to the network.

Example of an error message:

```
Oct 21 13:14:05 192.168.2.145 BCL(53): 53 General
syntax error: wrong or not allowed delimiter or
statement at this position
```

It is also possible to send custom messages using the SYSLOG statement:

Code example:

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

results in syslog message similar to this:

```
Dec 2 23:42:55 192.168.2.145 TESTING SYSLOG OUTPUT
```

Exact message format depends on the syslog daemon program used.

7.1 Error messages

Error number	Meaning
0	BCL file not existing or invalid tokencodeversion (use correct tokenizer version)
1	PRINT was not last statement in line or wrong delimiter used (allowed ',', ' or ';')
2	Wrong logical operator in IF statement (allowed '=', '>', '>=', '<', '<=', '<>')
3	ONLY String VARIABLE can be used as parameter in OPEN, READ, PLAY, MIDxxx, EXEC
4	Wrong delimiter/parameter is used in list of parameters for this statement/function
5	ON statement must be followed by GOTO/GOSUB statement
6	First parameter of TIMER statement must be 1..4 (# for ON TIMER# GOSUB...) → TIMER 0 is obsolete, see section 10, page 60
7	Wrong element is used in this string/numeric expression, maybe a type mismatch
8	Divided by Zero → division by zero, for example: Var1=Var2/0
9	Wrong label is used in GOTO/GOSUB statement (allowed only a numeric constant)

¹ Syslog is a well known reporting protocol usually using UDP port 514. Check the Internet for a free Syslog daemon.

Alternatively a universal logging tool (IP logger) available free of charge from www.barix.com can be used.

Error number	Meaning
10	Wrong symbol is used in source code, syntax error, tokenization is impossible → can be caused by too long quoted string constant (longer than 255 characters)
11	Wrong size of string/array is used in DIM (allowed only a numeric constant)
12	Wrong type in DIM statement used (only string variable or long variable/array allowed)
13	DIM was not last statement in line or wrong delimiter used (allowed only ',')
14	Missing bracket in expression or missing quote in string constant
15	Maximum nesting of calculations exceeded (too many brackets)
16	Assignment assumed (missing equal sign)
17	Wrong size of external tokenized TOK file (file might be corrupt)
18	Too many labels needed, tokenization is impossible
19	Identical labels in source code found, tokenization is impossible
20	Undefined label in GOTO/GOSUB statement found, tokenization is impossible
21	Missing THEN in IF/THEN statement
22	Missing TO in FOR/TO statement
23	Run-time warning: Possibly, maximum nesting of FOR-NEXT loops exceeded → too many nested FOR loops
24	NEXT statement without FOR statement or wrong index variable in NEXT statement
25	Maximum nesting of GOSUB-RETURN calls exceeded
26	RETURN statement without proper GOSUB statement → can be caused by improper use of GOTO statement
27	Lack of memory for temporary 1 kilobyte buffer in WRITE
28	String variable name conflict or too many string variables used
29	Long variable name conflict or too many long variables used
30	Insufficient space in far memory for temp string, variable or program allocation
31	Current Array index bigger then maximal defined index in DIM statement
32	Wrong current number of file/stream handler (allowed only 0..4)
33	Wrong file/stream type/type name or file/stream is already closed

Error number	Meaning
34	This file/stream handler is already used or file/stream already opened
35	Missing AS statement in OPEN AS statement
36	Wrong address in IOCTL or IOSTATE
37	Wrong serial port number in OPEN statement
38	Wrong baudrate parameter for serial port in OPEN statement
39	Wrong parity parameter for serial port in OPEN statement
40	Wrong data bits parameter for serial port in OPEN statement
41	Wrong stop bits parameter for serial port in OPEN statement
42	Wrong serial port type parameter in OPEN statement
43	Run-time warning: You lost data during PLAY -- Please, increase string size
44	For TCP/CIFS file/stream only handler with number 0..5 are allowed
45	Only standard size (256 bytes) string variable allowed for READ and WRITE in STP file
46	Wrong or out of string range parameters in MID\$ or MIDxxx
47	Only one STP/F_C file can be opened at a time
48	'&' can be used ONLY at the end of a line
49	Syntax error in multiline IF...ENDIF (maybe wrong nesting)
50	Length of Search Tag must not exceed size of target String Variable for READ
51	DIM string/array variable name already used
52	Wrong user function name or array declaration missing
53	General syntax error: wrong or not allowed delimiter or statement at this position ➔ can be caused by too long quoted string constant
54	Run-time warning: Lost data during UDP READ -- Please, increase string size ➔ too small buffer given to READ
55	Run-time warning: Lost data during UDP receiving -- 1k buffer limit
56	Run-time warning: Impossible to allocate 6 TCP handles, if 6 are needed free up TCP command port and/or serial
57	Run-time warning: Lost data during concatenation of strings -- Please, increase target string size (DIM statement) ➔ target string size (either set using DIM or the default) is insufficient

Error number	Meaning
58	Run-time warning: Lost data during assignment of string -- Please, increase target string size (DIM statement) → target string size (either set using DIM or the default) is insufficient
59	Indicated flash page (WEBx) is out of range for this HW
60	COB file (F_C type) exceeds 64k limit

8 Example programs

8.1 Playing an MP3 file from the USB filesystem

```

DIM _Ms$(2048)
OPEN "AUD:1,6,0,4000" AS 7
11
  SYSLOG "playback"
  OPEN "C_R:usb:///file.mp3" AS 2
101 READ 2, _Ms$
  l=LASTLEN(2)
  IF l<=0 THEN
    SYSLOG "end of file"
    CLOSE 2
    GOTO 11
  ENDIF
102 IF filesize(7)<l THEN GOTO 102 ' check if there's
enough space ' in the audio
buffer
  WRITE 7, _Ms$,l
  GOTO 101
END
    
```

8.2 Record audio into an MP3 file on the USB filesystem

Encode audio line input as MP3 and record for 60 seconds into file.mp3 on the local USB filesystem. For details about setting the audio parameters see section 3.10.3 on page 26. For details about `_DTS_` variable see section 3.4.8 on page 13.

```

DIM _M_r$(5000)
OPEN "C_W:usb:///file.mp3" AS 4
OPEN "AUD:2,0,"+str$(1024+7*16+1) AS 7 'mp3 encoding
WRITE 7,"10",-1 'set MIC Gain
WRITE 7,"10",-2 'set A/D Gain
WRITE 7,"1",-3 'set input
source (1-line)
WRITE 7,"20",-12 'set Volume
(100%)

time = _DTS_
1151 READ 7, _M_r$,4096 : ftp_l = LASTLEN(7)
IF ftp_l>0 THEN WRITE 4, _M_r$,ftp_l
IF ( _DTS_ -time)<60 THEN GOTO 1151
CLOSE 7
CLOSE 4
END
    
```

8.3 Sending an email

Send an e-mail assuming the correct SMTP server address is inserted and no errors occur (the error handling is not implemented in the example):

```

DIM BUFFER$(200)
BUFFER$=""
OPEN "TCP:192.168.2.130:25" AS 0
10
IF NOT(CONNECTED(0)) THEN GOTO 10
1 READ 0,BUFFER$,0
IF LEN(BUFFER$)=0 THEN GOTO 1
    
```

```

WRITE 0,"HELO example.com\r\n",0
2 READ 0,BUFFER$,0
IF LEN(BUFFER$)=0 THEN GOTO 2
WRITE 0,"MAIL FROM: <joeey@example.com>\r\n",0
3 READ 0,BUFFER$,0
IF LEN(BUFFER$)=0 THEN GOTO 3
WRITE 0,"RCPT TO: <agnes@example.com>\r\n",0
4 READ 0,BUFFER$,0
IF LEN(BUFFER$)=0 THEN GOTO 4
WRITE 0,"DATA\r\n",0
5 READ 0,BUFFER$,0
IF LEN(BUFFER$)=0 THEN GOTO 5
WRITE 0, "SUBJECT:Greetings\r\n"+&
  "From:joeey@example.com\r\n"+&
  "To:agnes@example.com\r\n"+&
  "from joeey\r\n"+&
  ".\r\n",0
6 READ 0,BUFFER$,0
IF LEN(BUFFER$)=0 THEN GOTO 6
WRITE 0,"QUIT",0
CLOSE 0
END

```

8.4 Streaming MP3 over RTP

Play an RTP MP3 stream received on UDP port 5555.

```

DIM _Ms$(2048)
OPEN "UDP:0.0.0.0:5555" AS 4
OPEN "AUD:1,7,0,20000" AS 7
WRITE 7,"18",-12 'set volume to 90%

1099
  l = LASTLEN(4)
  IF l >= 0 THEN GOTO 1099
  READ 4,_Ms$
  WRITE 7,_Ms$,-1
  GOTO 1099

```

8.5 TCP serial gateway

```

DIM S$(512)

tcp$= "TCP:0.0.0.0:10001" 'TCP listener on port
10001
OPEN tcp$ AS 1
OPEN "COM:9600,N,8,1,NON:1" AS 2 'Open serial
port
WRITE 2, "Waiting for TCP connection...\r\n",0
101 IF CONNECTED(1) = 0 THEN GOTO 101 'wait for TCP
connection

WRITE 2, "Connection established\r\n",0
tcp$ = "Host: "+RMTHOST$(1)+", Port:
"+STR$(RMTPORT(1))+"\r\n"
WRITE 2, tcp$, 0

111 READ 2,S$ 'read serial
input
  l=LASTLEN(2)
  IF l>0 THEN WRITE 1,S$,l 'send out to
TCP

```

```

        IF ASC(S$) = 27 THEN GOTO 112      'ESC code to
Break
        IF NOT(CONNECTED(1)) THEN GOTO 112  'check TCP
connection
        READ 1, S$
        l=LASTLEN(1)
        IF l>0 THEN WRITE 2,S$,l          'read chars
from TCP
        GOTO 111

112 WRITE 2,"Terminal disconnected\r\n",0
      CLOSE 1
      CLOSE 2
      END

```

8.6 The Wiegand reader

26-bit Wiegand reader access with queuing and socket to listen to

```

DIM Com$(24)          ' file name for open
function
  DIM s$(256),p$(256) ' string variables for
read and output
  DIM i,rdr,rlen      ' loop variable, reader,
read len
  DIM qu(200,3)       ' queue for reader id and
data (not              optimised,
could be 1 word)
  DIM quin,quout      ' in and out pointer
to reading queue

  COM$ = "RDR:"       ' open reader
interface
  OPEN Com$ AS 4

  COM$ = "TCP:0.0.0.0:10009" ' tcp
listening socket
  OPEN COM$ AS 1
  quin=1 quout=1
  SYSLOG "Wiegand reader demo 2.1",2

100
  READ 4,s$,0
  IF LASTLEN(4)>0 THEN GOSUB 1000 ' ID read,
get data and
store in queue
  IF AND(CONNECTED(1),quin<>quout) THEN

    p$=STR$(qu(quout,1)) &
      +SPRINTF$(",%04lx\r\n",qu(quout,2))+ &
      +SPRINTF$(",%06lx\r\n",qu(quout,3))
    WRITE 1,p$,0
    p$="from qu entry "+STR$(quout)+" sent: "+p$
    SYSLOG p$,6
    quout=quout+1
  IF quout=201 THEN quout=1 ' next storage
space, wrap to 1
  ENDIF
  GOTO 100

1000
  ' get reader ID (1 or 2)
  IF AND(MIDGET(s$,1,1),128) THEN rdr=2 ELSE
rdr=1
  ' get read length (how many bits)
  rlen=AND(127,MIDGET(s$,1,1))
  p$=SPRINTF$("Wiegand read from %u, ",rdr) &
    +SPRINTF$(" %02u bits: ",rlen)
  FOR i=2 TO LASTLEN(4)
    p$=p$+SPRINTF$("%02x ",MIDGET(s$,i,1))
  NEXT i
  SYSLOG p$,5
  ' if 26 bits, then decode to "real"
3 bytes
  IF rlen=26 THEN
    b1=AND(255,SHL(MIDGET(s$,2,1),1))+SHR(MIDGET(
s$,3,1),7)
    b2=AND(255,SHL(MIDGET(s$,3,1),1))+SHR(MIDGET(
s$,4,1),7)
    b3=AND(255,SHL(MIDGET(s$,4,1),1))+SHR(MIDGET(
s$,5,1),7)
    v1=0

```

```

        v2=b1*65536+b2*256+b3           ' store 24bit
wiegand ID
        GOTO 1100
    ENDIF
    IF rlen=44 THEN
        v1=MIDGET(s$,2,1)*256+MIDGET(s$,3,1)
        v2=(MIDGET(s$,4,1)*256+MIDGET(s$,5,1))*256+MI
DGET(s$,6,1)
        GOTO 1100
    ENDIF
    RETURN
1100                                     ' now store
in queue
    qu(quin,1)=rdr ' store reader number
    qu(quin,2)=v1  ' first part of value
    qu(quin,3)=v2  ' second part of value (typ. 24
bit wiegand id)

        SYSLOG "stored in qu entry "+STR$(quin),6
        i=quin+1
        IF i=201 THEN i=1                 ' wrap
        ' only store if this does not overrun
queue !
        IF i<>quout THEN quin=i
    RETURN
END

```

8.7 Simple internet radio player

Plays an internet radio stream from a shoutcast/icecast server.

```

DIM adr$(256)
DIM path$(256)
DIM _Mb$(4096)
DIM _l,t
DIM bufms

    adr$="vruk.sc.llnwd.net"           ' remote server
    port=12265                         ' remote port
    path$="/"                          ' remote path

    bufms=2000                         ' buffer for N ms
before playing
    vol=70                             ' volume in percent

    OPEN "AUD:1,18,0,"+str$(bufms) as 4 ' MP3
decoding+rebuffering
    WRITE 4, str$(vol/5), -12          ' set
volume

    SYSLOG "opening "+adr$+"..."
    OPEN "TCP:"+adr$+": "+str$(port) as 0 ' open TCP
connection

    SYSLOG "waiting for connection..."
    t=_TMR_(0)
1   IF CONNECTED(0)=0 THEN            ' wait for the
remote server
        IF _TMR_(0)-t>1000 THEN
            SYSLOG "server does not respond"
            GOTO 99
        ENDIF
    GOTO 1

```

```

ENDIF

SYSLOG "sending GET..."
WRITE 0, "GET "+path$+" HTTP/1.0\r\n\r\n",0      '
send HTTP GET

SYSLOG "waiting for response..."

4   t=_TMR_(0)

5   IF and(CONNECTED(0),filesize(0)=0) THEN
      IF _TMR_(0)-t>1000 THEN
          SYSLOG "connection timed out"
          GOTO 99
      ENDIF
      GOTO 5 ' wait for more data
ENDIF
t=_TMR_(0)
_Mb$="x"
READ 0, _Mb$, 0      ' read and print the
response header
IF LASTLEN(0)>0 THEN
    SYSLOG "header: "+_Mb$
ELSE
    IF LEN(_Mb$)=0 THEN GOTO 9 ' empty line ->
end of the header
ENDIF
GOTO 4

9   SYSLOG "playing..."
10  READ 0, _Mb$      ' read data from
socket
    l=LASTLEN(0)
    IF l THEN WRITE 4,_Mb$,l      ' if any data, send
them out
    IF CONNECTED(0) THEN GOTO 10

SYSLOG "playback finished"
99  SYSLOG "closing connection"
    CLOSE 0
    CLOSE 4

END

```

9 Syntax summary

All elements noted in bold in this summary can be surrounded by **Whitespace**.

Whitespace is any sequence consisting of:

- spaces (" ")
- tabulators (" ")
- ampersand ("&") followed by newline

BCL program code is a sequence consisting of:

- **Comments**
- **Unnumbered Lines**
- **Numbered Lines**

ending with **END** or **RETURN** and followed by the end of file

Line number is a sequence of numerical characters ("0123456789")

Comment is a sequence of characters satisfying:

- first character is an apostrophe ('')
- last two characters are CR/LF (newline)
- CR/LF is not used anywhere else in the sequence

Numbered line consists of:

1. **Line number**
2. **Unnumbered Line**

Unnumbered Line is one of the following:

- **Declaration** followed by newline
- Sequence of **Statements** separated by **Statement delimiters** and ended by newline or **Comment**

9.1 Variables, Constants, Expressions

Unsigned integer constant is:

- either: a sequence of numerical characters ("0123456789")
- or: "&H" followed by a sequence of hexadecimal characters ("0123456789ABCDEF")

Integer variable name is a string beginning with letter and otherwise containing only underscores and alphanumerical characters

String variable name is a string beginning with letter, ending with "\$" and otherwise containing only underscores and alphanumerical characters

One dimensional array element is:

1. **Integer variable name**
2. left parenthesis "("
3. **Integer expression**
4. right parenthesis ")"

Two dimensional array element is:

1. **Integer variable name**
2. left parenthesis "("
3. **Integer expression**
4. comma ","

5. Integer expression
6. right parenthesis ")"

Integer variable is one of the following:

- Integer variable name
- One dimensional array element
- Two dimensional array element

String constant is:

1. quota sign (")
2. sequence of printable other than quota sign
3. quota sign (")

Unsigned integer expression is one of the following:

- Unsigned integer constant
- Integer variable
- Integer function
- Integer expression followed by one of "+", "-", "*", "/", "%", "^" followed by Integer expression
- User function call
- Integer expression surrounded by parentheses "(, ")"

Integer expression is either Unsigned integer expression or Signed integer expression

User function call is:

1. Integer variable name
1. "("
2. Parameters (Integer expressions and String expressions) separated by comma
3. ")"

Signed integer expression is Integer unsigned expression preceded by "-" or "+"

String expression is one of the following:

- String constant
- String variable name
- String function
- String expression followed by "+" followed by String expression

9.2 Declarations

Declaration is either Parameter declaration or General declaration

General declaration consists of

1. "DIM" command
2. any number of Variable declarations separated by commas or User function declaration
3. CR/LF (newline)

Parameter declaration consists of

1. "LOCAL" command
2. any number of Local variable declaration, separated by commas
3. CR/LF (newline)

Local variable declaration is one of:

- **String variable declaration**
- **Integer variable name**

Variable declaration is one of:

- **String variable declaration**
- **Integer variable name**
- **One dimensional array declaration**
- **Two dimensional array declaration**

User function declaration is:

1. **Integer variable name**
2. **left angle bracket "<" and GOSUB**
3. **Line number**
4. **right angle bracket ">"**

String variable declaration is:

1. **String variable name**
2. **left parenthesis "("**
3. **Unsigned integer constant**
4. **right parenthesis ")"**

One dimensional array declaration is:

1. **Integer variable name**
2. **left parenthesis "("**
3. **Unsigned integer constant**
4. **right parenthesis ")"**

Two dimensional array declaration is:

1. **Integer variable name**
2. **left parenthesis "("**
3. **Unsigned integer constant**
4. **comma ","**
5. **Unsigned integer constant**
6. **right parenthesis ")"**

9.3 Statements and functions

Statement is on one of the following:

- **Conditional statement**
- **Unconditional statement**
- **FOR loop**
- **Handler setting**

Unconditional statement is one of the following:

- **Integer assignment**
- **String assignment**
- **Command call**
- **Function call**

Statement delimiter is one of the following:

- **colon ":"**
- **newline (CR/LF)**

Integer assignment consists of:

1. **Integer variable name**
2. **"="**

3. Integer expression

String assignment consists of:

1. **String variable**
2. **"="**
3. **String expression**

Command call is:

1. **Command name**, i.e. one of "CLOSE", "DELETE", "DELAY", "END", "GOTO", "IOCTL", "LOCK", "MIDCPY", "MIDSET", "OPEN", "PLAY", "READ", "SEEK", "SYSLOG", "TIMER", "TRAP", "WRITE"
2. **Respective parameters (Integer expressions and String expressions)** separated by comma

Function call is one of the following:

- **Integer function**
- **String function**

Integer function consists of:

1. **Command name**, i.e. one of "ASC", "CONNECTED", "END", "FILEPOS", "FILESIZE", "INSTR", "IOSTATE", "LASTLEN", "LEN", "MEDIATYPE", "MIDGET", "NOT", "OR", "PING", "RANDOM", "RESOLVE", "RMTPORT", "SHL", "SHR", "VAL", "XOR"
2. **"("**
3. **Respective parameters (Integer expressions and String expressions)** separated by comma
4. **)"**

String function consists of:

1. **Command name**, i.e. one of "CHR\$", "LCASE\$", "MD5\$", "MID\$", "RMTHOST\$", "SPRINTF\$", "STR\$", "UCASE\$"
2. **"("**
3. **Respective parameters (Integer expressions and String expressions)** separated by comma
4. **)"**

Conditional statement is one of the following:

- **One line IF**
- **One line IF-ELSE**
- **Multiline IF**
- **Multiline IF-ELSE**

One line IF consists of:

1. **"IF"**
2. **Integer expression or Boolean expression**
3. **"THEN"**
4. **Unnumbered line**

One line IF-ELSE consists of:

1. **"IF"**
2. **Integer expression or Boolean expression**
3. **"THEN"**
4. **sequence of Statements** separated by colons ":"
5. **"ELSE"**
6. **Unnumbered line**

Multiline IF consists of:

1. "IF"
2. **Integer expression** or **Boolean expression**
3. "THEN"
4. newline (CR/LF)
5. Sequence of **Unnumbered lines** and **Numbered lines**
6. "ENDIF"

Multiline IF-ELSE consists of:

1. "IF"
2. **Integer expression** or **Boolean expression**
3. "THEN"
4. newline (CR/LF)
5. Sequence of **Unnumbered lines** and **Numbered lines**
6. "ELSE"
7. Sequence of **Unnumbered lines** and **Numbered lines**
8. "ENDIF"

Boolean expression is one of the following:

- **Simple boolean expression**
- One of the boolean functions "NOT", "OR", "AND", "XOR" with parameters (**Boolean expressions**) in parentheses separated by commas

Simple boolean expression is one of the following:

- **String expression** followed by one of "=", "<>" followed by **String expression**
- **Integer expression** followed by one of "<",">","=","<>","<=",">=" followed by **Integer expression**

FOR loop consists of:

1. "FOR"
2. **Integer variable V**
3. "="
4. **Integer expression**
5. "TO"
6. **Integer expression**
7. newline (CR/LF)
8. Sequence of **Numbered lines** and **Unnumbered lines**
9. "NEXT" with optional **Integer variable V**
10. newline (CR/LF)

Handler setting is one of the following:

- "ON CGI GOSUB" followed by **Number line**
- "ON UDP GOSUB" followed by **Number line**
- "ON TIMER 1 GOSUB" followed by **Number line**
- "ON TIMER 2 GOSUB" followed by **Number line**
- "ON TIMER 3 GOSUB" followed by **Number line**
- "ON TIMER 4 GOSUB" followed by **Number line**

10 Appendix A – obsolete or unimplemented functions

These functions, which can be found in older BCL programs, are now considered obsolete and replaced.

ISEQV(E1\$, E2\$)

This function has been used for string comparison in boolean expressions. It has been replaced by the equal sign (=) as strings now can be matched directly, see section 3.4.6.3 on page 12.

SYSTIME

Returns time in milliseconds since the 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 system timer counting time in milliseconds.

Code example:

```
10 STIME= TMR_(0)
SYSLOG STR$(STIME)
DELAY 1000
GOTO 10
```

CMD\$ variable had been used to determine the next program to be started. This functionality is currently provided by the END command.

EXEC function had been reserved for certain hardware dependent operations not covered by the standard I/O functions. Currently, all possible I/O operations are supported by other means so the EXEC function does not implement any functionality.

INKEY\$

Used to return the last characters received from the input buffer, or an empty string if there were no characters on input. The same functionality can be achieved using the standard I/O functions for the serial port.

INPUT [{ S0\$ | Q\$ } ,] { V | S\$ }

Original function: Prints S0\$ or Q\$ as prompt (if not specified, prints the "?" mark), waits for input from user, and sets a new value for long V or string S\$ variable. If this is a new variable name, creates the new default long or string variable. Now the same functionality can be achieved using the standard I/O functions for the serial port.

PRINT [[{ S\$ | Q\$ }] [{ , | ; }]] ...

Original function: Prints a list of arguments S\$ or Q\$ with delimiters. Delimiter "," means printing the next argument from new 8-chars zone (tabulator). Delimiter ";" means printing without spaces immediately after the last value. If no end delimiters are specified, the next printing starts from a new line.

Now, the same functionality can be achieved using the standard I/O functions for the serial port.

Timer 0

In addition to timers 1-4 there used to be timer 0 which worked as a "software watchdog". Once the count of this timer expired the BCL program was terminated. To prevent a program restart, the timer had to be periodically triggered with

TIMER 0, E (e.g. TIMER 0, 5000 reset the watch dog timer for the following 5 seconds).

The same functionality can be achieved using any other timer, if the handler subroutine is programmed to terminate the program.

Code example:

```
TIMER 1, 5000
....
5000 END
```

II Appendix B – BIN / DEC / HEX conversion

Hexadecimal digits have values from 0 . . 15, represented as 0 . . 9 and as A (for 10) to F (for 15).

The following table can serve as a conversion chart:

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.

I2 Appendix C – BCL version 2

In the future release of the BCL language (version 2), the following will be changed.

Change 1:

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

Searches for E2\$ in E1\$ starting from the position indexed by E up to the end of the string. On success it returns the position of the E2\$ in E1\$, counting from 1 (for E2\$ at the beginning of E1\$). Otherwise it returns 0. Search for an empty string E2\$ returns 0.

Change 2:

In BCL strings are indexed from 1 (not NULL terminated).

Alphabetical Index

address resolution.....	19	<i>Timer</i>	56
arrays.....		<i>TO</i>	10
<i>binary</i>	10	<i>WRITE</i>	22, 25
<i>integer</i>	7	comments.....	5
audio.....		delimiters.....	5
<i>closing</i>	31	directory.....	
<i>delay</i>	26	<i>listing</i>	34
<i>device</i>	26	<i>number of entries</i>	34
<i>flags</i>	26	<i>pointer position</i>	34
<i>flushing buffer</i>	31	<i>seek</i>	33
<i>free bytes</i>	27	<i>USB filesystem</i>	33
<i>mode</i>	26	escape sequences.....	8
<i>opening</i>	26	events.....	13, 21
<i>PCM modes</i>	27	events.....	
<i>playing file (example)</i>	46	<i>ON CGI</i>	14
<i>playing RTP (example)</i>	47	<i>ON ERROR</i>	15
<i>quality</i>	26	<i>ON TIMER</i>	14
<i>raw data mode</i>	27	<i>ON UDP</i>	14
<i>reading status</i>	29	expression.....	
<i>recording (example)</i>	46	<i>boolean</i>	12
<i>RTP</i>	28	<i>integer</i>	6
<i>setting parameters</i>	30	<i>string</i>	9
CGI.....		file	
<i>BAS.cgi</i>	40	<i>deletion</i>	33
<i>basic.cgi</i>	40	<i>flash reading</i>	34
<i>event</i>	14	<i>flash writing</i>	34
<i>handling</i>	40	<i>line read</i>	22
<i>ON CGI</i>	14	<i>position</i>	33p.
<i>variable setting</i>	40	<i>seek</i>	33p.
<i>_CGI_\$ variable</i>	38	<i>size</i>	33p.
command.....		<i>USB filesystem</i>	32
<i>CLOSE</i>	22, 31	<i>USB playing (example)</i>	46
<i>DELAY</i>	13	<i>USB writing (example)</i>	46
<i>DELETE</i>	33	function.....	
<i>DIM</i>	5pp., 9	<i>ASC</i>	17
<i>ELSE</i>	12	<i>CHR\$</i>	17
<i>END</i>	6, 10	<i>CONNECTED</i>	25
<i>ENDIF</i>	12	<i>EXEC (obsolete)</i>	56
<i>FOR</i>	10	<i>FILEPOS</i>	33p.
<i>GOSUB</i>	11	<i>FILEPOS</i>	34
<i>GOTO</i>	6, 10p.	<i>FILESIZE</i>	23, 26p., 32pp.
<i>IF</i>	12	<i>INKEY\$ (obsolete)</i>	56
<i>INPUT (obsolete)</i>	56	<i>INSTR</i>	16, 59
<i>LOCK</i>	15, 38	<i>IOCTL</i>	36
<i>NEXT</i>	11	<i>IOWSTATE</i>	37
<i>ON CGI</i>	14	<i>ISEQV</i>	56
<i>ON ERROR</i>	15	<i>ISEQV (obsolete)</i>	56
<i>ON TIMER</i>	14	<i>LASTLEN</i>	23p., 29p., 33
<i>ON UDP</i>	14	<i>LCASE\$</i>	17
<i>OPEN</i>	22	<i>LEN</i>	16
<i>PRINT (obsolete)</i>	56	<i>MD5\$</i>	19
<i>RETURN</i>	6, 11	<i>MEDIATYPE</i>	23
<i>SEEK</i>	33	<i>MID\$</i>	16
<i>SYSLOG</i>	43	<i>MIDCPY</i>	20
<i>THEN</i>	12	<i>MIDGET</i>	20

MIDSET	20	<i>line read</i>	22
PING	37	<i>on the Barionet</i>	32
RANDOM	19	setup.....	32
READ	22, 27p.	string.....	
RESOLVE	19	<i>assignment</i>	53
RMTHOST\$	24p.	<i>case</i>	17
RMTPORT	24p.	<i>constant</i>	8, 52
SEEK	33p.	<i>default size</i>	38
SPRINTF\$	17	<i>expression</i>	9, 52
STIME	17	<i>formatting</i>	17
STR\$	17	<i>function</i>	16, 54
SYSLOG	37	<i>integer conversions</i>	17
SYSTIME (obsolete)	56	<i>length</i>	16
TRAP	37	<i>number of variables</i>	38
UCASE\$	17	<i>string to time conversion</i>	17
<i>user defined</i>	20	<i>substring</i>	16, 59
VAL	17	<i>variable</i>	9
28		<i>variable name</i>	51
integer.....		strings.....	8
<i>array</i>	7	subroutines.....	11
<i>assignment</i>	53	TCP	
<i>constant</i>	6	<i>client example</i>	46, 49
<i>data size</i>	38	<i>line read</i>	22
<i>expression</i>	6, 52	<i>protocol</i>	25
<i>function</i>	7, 54	<i>radio player (example)</i>	49
<i>maximal dimension of</i>		<i>receiving</i>	25
<i>arrays</i>	38	<i>sending</i>	46
<i>number of variables</i>	38	<i>serial gateway (example)</i>	47
<i>scaling</i>	7	<i>server example</i>	47
<i>variable</i>	6, 52	time.....	13
<i>variable name</i>	51	<i>conversion</i>	17
labels.....	10	<i>delay</i>	13
MD5 sum	19	RTC	13
operation.....		<i>system time</i>	13
AND	8, 13	<i>timers</i>	14
<i>assign</i>	6, 9	<i>to string conversion</i>	18
<i>integer</i>	6	tokenizer.....	3
NOT	8, 12	UDP	
OR	8, 13	<i>client example</i>	47
SHL	8	<i>event</i>	14
SHR	8	<i>protocol</i>	24
<i>string</i>	9	<i>receiving</i>	24
XOR	8, 13	<i>sending</i>	25
protocol.....		<i>sending to multiple</i>	25
<i>audio</i>	26	<i>destinations</i>	25
Audio	2, 26	variable.....	
<i>Serial</i>	2, 31	<i>array of integers</i>	7
SETUP	32	DTS_	13
TCP	2, 25	<i>integer</i>	6
UDP	24	<i>string</i>	9
Wiegand	35	_ARG	21, 38
RTP		_CGI_\$	40
<i>data mode</i>	28	_CMD_\$	56
<i>payload types</i>	28	_ERL	15
<i>player (example)</i>	47	_ERR	15
serial.....		_M	10
<i>bytes available</i>	32	_TMR	14
<i>configuration</i>	31	web page.....	
<i>gateway (example)</i>	47	BCL specific HTML tags	39

<i>calling BCL from a webpage</i>	<i>26-bit reader</i>
<i>39</i>	<i>35</i>
<i>displaying variables</i>	<i>26-bit reader (example)</i>
<i>39</i>	<i>36</i>
Wiegand reader	<i>data format</i>
	<i>35</i>
	<i>sample program</i>
	<i>48</i>

Legal Information

© 2007 Barix AG, Zurich, Switzerland.

All rights reserved.

All information is subject to change without notice.

All mentioned trademarks belong to their respective owners and are used for reference only.

Barix, Annunicom, Barionet, Exstreamer, Instreamer, SonicIP and IPzator are trademarks of Barix AG, Switzerland and are registered in certain countries.

For information about our devices and the latest version of this manual please visit www.barix.com.



Barix AG
Seefeldstrasse 303
8008 Zurich
SWITZERLAND

Phone: +41 43 433 22 11
Fax: +41 44 274 28 49

Internet

web: www.barix.com
email: sales@barix.com
support: support@barix.com