BARIX
THE VOICE OF SIMPLICITY

# BCL Barix Control Language

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

# Programmers Manual

**Version** 1.20
**Released** 6th January 2016
**Supports**:

- **Barionet 50, 100 (LX & EX XPort)**
- **Annuncicom family**
- **Exstreamer family**
- **Instreamer 100**
- **IPAM family**

# Revision History

| Version | Date | Initials | Notes |
|---|---|---|---|
| 1.10 | 30/03/10 | PK | Described changes implied by frame based buffering. |
| 1.10 | 05/05/10 | KK | Do not write into _TMR_ |
| 1.11 | 02/06/10 | PK | Link between file and audio |
| | 03/06/10 | JP/PK | Added the target parameter for Tokenizer<br>New Barix logo |
| 1.12 | 08/05/10 | KK | Corrected explanation of TCP, COM READ timeouts |
| | 17.08.10 | JP | BCL.1.5 parameter changes incorporated. |
| 1.13 | 10/11/10 | PK | Programmable 1-wire interface on Barionet 50 |
| | 03/12/10 | PK | Changed front page graphics |
| | 21.12.10 | JP | No of supported Handles(16) updated. |
| | 08/02/11 | PK | Corrected %H printf parameter. |
| 1.14 | 22/02/11 | PK | Added new audio parameters: bass/treble frequency setting |
| 1.15 | 02/09/11 | PK | More details about Setup reading/writing.<br>Example extended.<br>More detailed description of system variables and INSTR.<br>Function FIND described. |
| 1.16 | 19/01/12 | PK | Added G.722 support |
| 1.17 | 18.05.12 | PK | Added 5-band parametric equalizer |
| | 30.05.12 | PK | Added AEC control bit into audio “quality” parameter |
| | 10.10.12 | PK | Note on using LINK with raw UDP |
| 1.18 | 25.02.13 | PK | Audio delay described in full-duplex (#47.83)<br>Last packet timestamp audio status parameter for end of stream detection with LINK command.<br>MP3 encoding with bitrate: CBR, VBR, ABR (#48.31)<br>Min and max jitter added to audio status: (#48.32)<br>Microphone gain from 12dB up to 43.5dB on VLSI (#53.82) |
| | 09.09.13 | PK | Removed TCP handle limitation to 5 handles |
| | 29.04.14 | PK | Updated: Jitter is in milliseconds |
| 1.19 | 21.08.14 | PK | Added new RTP payload type 113<br>Added long directory listing |
| 1.20 | 06.01.16 | PK | Updated the SNMP features for the Barionet 50 |

# References

| | Document | Date | Author |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

# Table of Contents

# 1 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.

## 1.1 Notation

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

| | |
|---|---|
| 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 &HFFFFFFFF) |
| L | Line number. Line numbers are unsigned integers from 1 to 32767 |
| 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) |
| A | Integer array |
| V | integer variable or array element V(E [, E] ) |
| H | file handle (integer in range 0..15) |
| 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 |

## 1.2 Supported devices

BCL is currently supported by the automation controller Barix Barionet and all Barix Audio products including the IP Audio Module family.

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

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

| | | | | | | |
|---|---|---|---|---|---|---|
| TCP/UDP networking | ● | ● | ● | ● | ● | ● |
| Serial port(s) | ● | ● | ●[1] | ● | ● | ●[2] |
| Web interface | ● | ● | ● | ● | ● | ● |
| Audio output | | | ● | ● | ● | ● |
| Audio input | | | ● | | ● | ● |
| USB filesystem | | | ● | ●[3] | ●[4] | ●[5] |
| One-wire sensors | ● | ● | | | | |
| Programmable one-wire interface | | ● | | | | |
| TFTP | ● | | | | | |
| Wiegand reader | ● | | | | | |
| Flash write | ● | | | | | |

1 Two serial interfaces available
2 Two serial interfaces available on Annuncicom 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 Instreamer100
5 Not available on older hardware versions prior to Annuncicom100

# 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 newlines[1]. 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 , page 78).

## 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 target program.bas
```

where `program.bas` is the name of the the source file and `target` is the target platform for the BCL program. Supported targets are:

| | |
|---|---|
| barionet | Barionet 100 |
| barionet50 | Barionet 50 |
| phoenix | FL COMSERVER PRO |
| audio | All Barix audio products and IPAM based products |

**Note:** Make sure that you provide the proper target for your application. TOK file build for a different platform might not run properly.

The tokenizer will tokenize the program and create `program.tok` file, it also creates `ERRORS.HLP` file 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

1 As common in DOS and Windows operating systems

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.

## 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 BCL basics

## 3.1 Starting with BCL

### 3.1.1 Simple program

Here is a simple program to test that tokenizer, tftp uploading, BCL interpreter in the BCL device and syslog daemon[1] 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, V1.5
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, V1.5
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 4.1 Interger Arrays on page 12) 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 &
```

1 For more information about syslog, see section , page

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"
```

### 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 Syntax overview

This section is provides an introduction to the BCL syntax. It is not intended to provide a complete syntax of BCL but an overview of most important language elements and constructs. For more details refer to section 4 and following. The BCL grammar can be found in chapter 15.

### 3.2.1 Data types and variables

There are two basic types of constants in BCL: **integers** and **strings**. Integer constants can be written in decimal notation (signed), e.g. `537`, `+26` or `-17`; or in hexadecimal notation (unsigned) starting with prefix `&H`, e.g. `&H2FA` (762 decimal).

String costants are quoted using the ' " ' character (ASCII 0x22), e.g. "hello".

Three types of data structures are supported by BCL: **integer variables**, **integer arrays** and **strings**. Every variable has a unique name composed of ASCII letters, digits and underscore ('_', ASCII 0x5F). Variable names must not start with a digit. Variable names are case insensitive and only the first five characters are significant.

Variables are **declared** with the `DIM` command:

```
DIM int                 ' integer variable
DIM array(10,10)        ' integer array
DIM str$                ' string
```

All variables are **initialised** to 0, strings are initialised to an empty string.

Variables and constants can be combined in **expressions**, however, only elements of the same type can be combined:

```
3*b+(26*a-7)/c              ' integer expression
"hello "+a$                 ' string expression
```

A value is assigned to a variable in an **assignment**, e.g.:

```
a=10
b$="hello" + " " + "world"
d=(a+b)/2
```

Read more about types and variables in sections 11 and 14.

### 3.2.2 Procedures and functions

Built-in BCL **procedures** are called by the procedure name optionally followed by comma-separated procedure parameters. E.g.:

```
WRITE 1,a$,10
```

Procedures do not return any value.

Built-in **functions** as well as user defined functions are called by their name, followed by the opening bracket '(' (ASCII 0x28), an optional comma-separated list of function parameters and closed by the closing bracket ')' (ASCII 0x29).

**Functions always return a value** (a string or an integer) and therefore can be called in an expression. Functions can be called as a statement, in which case the return value is discarded.

Examples:

```
l=LEN("Hello world!")          ' function call in an assignment
a$="Total sum is "+STR$(a)     ' function call in an expression
my_function()                  ' user function called as a statement
```

Read more about user defined functions in chapter , more about integer functions in chapter 4 and more about string functions in chapter 5.6.

### 3.2.3 Conditional statements

Conditional program execution is achieved by using `IF-THEN-ELSE` construct. For example:

```
IF a<b THEN min=a ELSE min=b
```

The program part after `THEN` is executed only if the condition after `IF` is true. The `ELSE` branch is executed only if the condition is false.

Conditional expressions can be spread over several lines or the `ELSE` branch can be omitted.

Example:

```
IF LEN(a$)>100 THEN
      SYSLOG "String too large"
      GOSUB 2000
ENDIF
```

Read more about conditional statements in chapter 6.5

### 3.2.4 Program flow control

Program flow can be controlled by the following statements: `GOTO`, `GOSUB-RETURN`, `FOR-NEXT`.

`GOTO` is an **unconditional jump** to a specific **label.** Label is a unique numeric mark in the program .

Example:

```
10
      SYSLOG "Current time is "+SPRINTF$("%1t",0)
      DELAY 1000
      GOTO 10
```

GOSUB is a **subroutine call** and is similar to GOTO. The difference is that GOSUB uses a stack to remember the original place where the subroutine was called from. RETURN statement is then used to return from the subroutine and continue in the original code.

Example:

```
...
      IF time>1000 THEN
            GOSUB 1000
      ELSE
            WRITE 1,a$,10
      ENDIF

...


1000
      SYSLOG "Connection timed out"
      CLOSE 1
      RETURN
```

The FOR-NEXT construct allows to repeat a specific part of the program (called **loop**) several times.

Example:

```
      a$=""
      FOR i=1 TO 10
            a$=a$+" "+STR$(i)
      NEXT i
      SYSLOG a$
```

Read more about program flow control in chapter 6.

# 4 Integers

## 4.1 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`.

## 4.1 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 varibles 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
```

## 4.2 Integer expressions

Integer expressions can be formed using the following operators

| | |
|---|---|
| () | brackets |
| +,- | unary sign operator |
| ^ | exponentiation |
| *,/, % | multiplication, division, remainder(modulo) |
| +,- | addition, subtraction |

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

## 4.3 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 , page 29.

## 4.4 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 5.6.2 on page 17).

## 4.1 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 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)
```

### 4.1.1 Array search

`FIND( E, A, [E0] )`
Searches a one- or two-dimensional integer array (table) `A` for a key `E`. Returns the row of the first occurrence of the key `E` in array `A` (return value counts from 0, i.e. return value 0 means first row in the table), or -1 if not found. The optional parameter `E0` specifies the column to search, 0 stands for the first column. If not provided the first column is searched.

This function is useful for implementation of various lookup tables since it provides a fast search in oppose to conventional FOR loop. Example: address lookup in a table of pairs <IP address,port>

## 4.2 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.

# 5 Strings

In BCL strings are NULL terminated and indexed from 1. (Future releases: please note change 2, chapter 1, page 100)

## 5.1 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
```

## 5.2 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.

| | |
|---|---|
| \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"
```

## 5.3 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
```

## 5.4 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 then 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.

## 5.5 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.

## 5.6 String functions

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 substring E2$ in a string E1$ starting from the position indexed by E up to the end of the string (the first character \0 in the string). E counts from 1. On success INSTR returns the absolute position of E2$ in E1$, counting from 1 (for E2$ being a prefix of E1$). If E2$ is not found 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 1, page 100.

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"

## 5.6.1 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.

## 5.6.2 Formatted conversions - SPRINTF$

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$

### *5.6.2.1 Integer to string conversions*

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"

### *5.6.2.2 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)

### *5.6.2.3 Network information*

%H MAC address without separators (e.g. 00204A804087)
%1H MAC address with colon separators (e.g. 00:20:4A:80:40:87)

The parameter E must be set to 0.

%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 1 "DNS 1" (e. g. 192.168.0.1)
5 Broadcast address (e. g. 192.168.0.255)

%lA 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>.

%llA same with leading zeroes

### *5.6.2.4 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"`

## 5.7 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!!"

# 6 Execution flow control commands

## 6.1 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.

## 6.1 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.

## 6.2 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
```

## 6.3 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
```

## 6.4 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.

## 6.5 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:

### 6.5.1 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.

### 6.5.2 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.

### 6.5.3 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.

### 6.5.4 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
```

## 6.6 Time

System variable _DTS_ is inicialized 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 inicialized 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.

## 6.7 Events

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

### 6.7.1 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. Do not write into _TMR_ variables.

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.

### 6.7.2 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 a pending received 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 7.2, page 33.

### 6.7.3 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 12, page 74.

### 6.7.4 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).

### 6.7.5 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.

## 6.8 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)

| | |
|---|---|
| 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 |
| 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.

# 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.

# 7 I/O stream functions

## 7.1 Function overview

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.

### 7.1.1 Open and close

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,..,15 are allowed. Handle numbers are common for all protocols and the same handle cannot be opened for two different streams at the same time.
   - Whether open is blocking or non-blocking depends on the particular protocol, see protocol specific sections below.
1. 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.
2. 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`.

### 7.1.1 Write

`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: Unless explicitly set in the open statement (see non-blocking TCP below) the write call is blocking and it does not return before the data is written to the output (or the output buffer).

### 7.1.2 Read

`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, unless there is a received UDP packet pending on the handle H.

#### *7.1.2.1 Line read*

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 the end-of-line character(s) being stripped off.

The program must provide large enough buffer (S$ string), so that the longest possible line of the input fits into the buffer. If a line longer than the string size appears on the input, the READ function reads the first N-1 characters of the input and returns, where N is the size of the provided S$

The supported end-of-line markers are CR only (modbus), LF only (Unix) and CRLF (DOS).

Line read can be combined with binary read, however if line read is used, the only allowed combination of CR+LF is as the EOL marker. The input **must not** contain binary LF (ASCII 0x0A) after text CR (ASCII 0x0D).

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
```

#### *7.1.2.2 Read timeout*

For streams (COM, TCP) E0 can be positive integer timeout in milliseconds:

- If no new data on the stream, returns immediately
- As soon as whole S$ can be filled, fills it and returns.
- If time out, reads available data and returns.

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

#### *7.1.2.3 Pattern search*

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, unless there is a received UDP packet pending on the handle.

### 7.1.3 Stream types

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.

| | |
|---|---|
| 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 |
| 19 | One wire bus |

### 7.1.4 Other functions

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.

## 7.2 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 packed received from stream H or 0 if not applicable.

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

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

### 7.2.1 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$, -l
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 14.4, page 86.

### 7.2.2 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$=...
LEN=...
RECIP(0,0)=RESOLVE("192.168.15.3")
RECIP(0,1)=200
RECIP(1,0)=RESOLVE("192.168.13.2")
RECIP(1,1)=300
RECIP(2,0)=RESOLVE("192.168.13.2")
RECIP(2,1)=400
OPEN "UDP:" AS 1
WRITE 1,BUF$,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.

### 7.2.3 Multicast

Multicast IP addresses can be used for UDP reception as well as for UDP sending. The device subscribes to a multicast group at the moment of calling UDP OPEN with a multicast address (listener). De-registration happens upon calling CLOSE on the handle. Up to 16 different multicast groups can be registered at the same time.

Note: Sending a UDP frame to a multicast address does not register to the group. Subscription is not needed to send to a multicast group.

## 7.3 The TCP network protocol

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

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

Note that for TCP connections associated with the handle number 0 a large (4KiB) receiving buffer is used. Otherwise only a 1024B receiving buffer is used.

### 7.3.1 Listening socket

To open a listening socket, the IP address should be 0.0.0.0:

```
OPEN "TCP:0.0.0.0:<port number>" AS H
```

The OPEN function returns immediately in that case and after returning the socket is ready and waiting for incoming connections.

Example:

```
OPEN "TCP:0.0.0.0:1000"  AS 3
```
opens listening TCP connection on port 1000 as handle 3

## 7.3.2 Blocking TCP connection

If an active connection is open, the OPEN function is **blocking** and waits up to 10 seconds for the connection to be established. After it returns, the connection is either established or closed (open failed), the state can (and should) be checked by calling the CONNECTED function. In case of failure BCL does not specify the cause of the error, it can be one of the following reasons:

- no ARP entry found
- DNS resolution failed
- connection timed out
- port blocked or closed.

If the OPEN fails and the connection has not been established, the file handle is already closed and it is not necessary to call CLOSE.

Example:
```
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.

## 7.3.1 Non-blocking TCP connection

To open an active connection without blocking, use the following OPEN sequence:
```
OPEN "TCN:<IP address>:<port number>" AS H
```

OPEN returns immediately and it is up to the application to check the state of the socket; the state can be checked by calling the CONNECTED function. After the connection has been established, the socket can be used for reading and writing.

If the connection has not been established within a reasonable amount of time, it's up to the application to handle the error condition; in that case the handle is still open and CLOSE **must** be called to release it for further use.

Non-blocking open can be also called with a DNS address instead of an IP address, in that case the OPEN function returns after the address has been resolved, which may take significant time. Therefore to avoid this delay the usage of DNS addresses with non-blocking TCP is not recommended; the RESOLVE function should be used instead. Example:

```
DIM ip
ip=RESOLVE("my.address.com")              ' resolve once at startup
...
OPEN SPRINTF$("TCN:%lA:1000",ip) as 3     ' use the resolved IP
                                          ' address in program core
```

### *7.3.1.1 Non-blocking TCP write*

WRITE H, E$, -E0

A negative length defines a non-blocking TCP write. In this case the WRITE statement returns immediately, writing **up to** E0 bytes from E$ into the TCP socket H. A consecutive LASTLEN call returns the number of bytes actually written to the socket. For proper operation the application **should** check the length returned by LASTLEN and handle potential retries.

## 7.3.2 TCP close

CLOSE H [ ,E ]

TCP close can be called in two ways. If CLOSE is called without the optional parameter E, the system waits until the partner acknowledges the close, which may take up to 10 seconds. The execution of the program is blocked for that time.

In some situations a faster reaction time is required. The optional parameter E defines the maximum time in milliseconds (non-negative integer), how long the system waits for the other party to acknowledge the close. After this timeout, if no acknowledge is received, the connection is dropped.

In the case, that E is zero, the system will wait for unlimited time.

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

For the special CGI handle -1 (see chapter 12.2) it returns the originating port of the connection, which can be used e.g. for authentication.

RMTHOST$( H )
Returns the remote host IP of the stream H, or an empty string.

For the special CGI handle -1 (see chapter 12.2) RMTHOST$ returns the remote address of the client, which can be used e.g. for authentication.

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 incomming FIFO of the stream H available for reading.

See also example 14.3 on page 85.

## 7.4 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

| | |
|---|---|
| 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 14.6, page 86.

## 7.5 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 don't move the current position pointer. That means subsequent reads or writes will always be from the same position where the file was opened.

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.

`READ H, S$`
Reads 256 bytes from Setup handle `H` into `S$`.

`WRITE H, S$, 256`
Writes 256 bytes from `S$` into Setup handle `H`.

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$
' perform modifications on set$
WRITE 3,set$,256
CLOSE 3
OPEN "STP:256"  AS 3
```

```
READ 3,set$
' perform modifications on set$
WRITE 3,set$,256
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.

## 7.6 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.

### 7.6.1 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.

`RENAME OLD$, NEW$`
Renames the file OLD$ to NEW$. OLD$ is a full path including the device identifier and possibly directory names. NEW$ is just a filename and must not contain any device identifier or slashes. Therefore it cannot move the file into another directory. The OLD$ and NEW$ names must be short ones in the 8.3 format.
Example:

```
RENAME "usb:///dir/readme.txt", "readme"
```

Renames the file `readme.txt` in directory `dir` to `readme`.

`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 14.1 on page 85 how to play a file.

### 7.6.2 Directory access

*Short filenames*

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
```

*Long filenames, extended listing*

If a larger buffer of at least 298 bytes (i.e. 299 bytes with terminating zero) is provided, the directory read function returns for each directory entry a complete information including the long filename.

The structure is described in the following table (for Midget add 1 to the offset):

| | | | |
|---|---|---|---|
| 0 | 13 | Short name | Short filename in form 8.3 plus terminating zero |
| 13 | 2 | Flags | 1 = file<br>2 = directory |
| 15 | 1 | Attributes | Copy of FAT16 attributes:<br>bitwise: 00ARSHDV |
| 16 | 2 | Extended attributes | Copy of VFAT extended attributes |
| 18 | 4 | Creation time | 1 byte hour<br>1 byte minute<br>1 byte second<br>1 byte centiseconds (10ms unit) |
| 22 | 4 | Creation date | 1 byte day (1-31)<br>1 byte month (1-12)<br>1 byte year since 1980 (0=1980)<br>1 byte reserved |
| 26 | 4 | Last access date | 1 byte day (1-31)<br>1 byte month (1-12)<br>1 byte year since 1980 (0=1980)<br>1 byte reserved |
| 30 | 4 | Last modification time | 1 byte hour<br>1 byte minute<br>1 byte second<br>1 byte centiseconds (10ms unit) |
| 34 | 4 | Last modification date | 1 byte day (1-31)<br>1 byte month (1-12)<br>1 byte year since 1980 (0=1980)<br>1 byte reserved |
| 38 | 4 | File size | File size in bytes (32-bit unsigned integer) |
| 42 | 256 | Long file name | 255 characters + terminating zero |

# 7.7 The local flash filesystem

## 7.7.1 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 1

```
OPEN "F_R:testfile.txt" AS 1
```

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.

### 7.7.2 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.

## 7.8 Keyboard and display interface (audio devices only)

On hardware featuring a keyboard and/or a text display, these can be accessed through the file interface as well. For that purpose a special handle -2 is defined. This handle is always open and can not be closed. The following IO functions are supported on handle -2.

### 7.8.1 Display

FILESIZE ( -2 )
Returns the size of the display as a 16-bit word: 256* height + width. Height defines the number of lines of the display whereas width the number of characters per line.
If the hardware does not feature a display FILESIZE returns 0.

WRITE -2, S$, E
Writes E bytes from S$ to handle the display. If E is 0 then the whole string up to the null-terminator is written.

S$ can contain both text and control characters. Text is written from the current cursor position up to the end of the line. Characters with codes 32 to 127 are supported, US ASCII encoding is used.

Standard ANSI escape sequences are used to control the display. Each control sequence starts with the “escape” character (ASCII code `27`, hexadecimal `0x1B`) followed by the '[' character (left square bracket, ASCII code `91`, hexadecimal `0x5B`). The following sequences are supported.

| | |
|---|---|
| `ESC [ 2 J` | **Display Clear**<br>clears the display and moves cursor to the upper left corner of the display (position 0,0) |
| `ESC [ Pn A` | **Cursor Up**<br>Moves cursor up by the given specified of lines. If the cursor is already at the top line ignores this sequence. |
| `ESC [ Pn B` | **Cursor Down**<br>Moves cursor down by the given specified of lines. If the cursor is already at the bottom line ignores this sequence. |
| `ESC [ Pn C` | **Cursor Forward**<br>Moves cursor right by the given specified of lines. If the cursor is already in the rightmost column ignores this sequence. |
| `ESC [ Pn D` | **Cursor Backward**<br>Moves cursor left by the given specified of lines. If the cursor is already in the leftmost column ignores this sequence. |
| `ESC [ PL ; Pc H` | **Cursor Postition**<br>Moves the cursor to the specified position (coordinates). If the position is not specified moves the cursor to the upper left corner. If the coordinates are out of the screen they are clipped to the display size. |
| `ESC [ PL ; Pc f` | Same as the previous sequence. |
| The following abbreviations are used:<br>*Pn* – stands for a decimal number<br>*PL* – stands for a line number, line 0 is the topmost line<br>*Pc* – stands for a column number, 0 is the leftmost column | |

In addition the character with ASCII code `10` (line feed) is used as the end-of-line terminator (moves cursor to the beginning of the next line). If already at the last line of the display the sequence is ignored.

Note: to include ESC in a string use the `\x1b` sequence

```
WRITE -2, "\x1b[2J",0                                    ' clear display
WRITE -2, "\x1b[0;"+str$((FILESIZE(-2)/2)-6)+"H",0   ' cursor pos
WRITE -2, "Hello  World",0
```

The above example clears the screen and writes “Hello  World” in the middle of the first line.

## 7.8.2 Keyboard

The built-in keyboard driver captures key events (key presses and key releases) and stores them in an internal event queue. If the queue is full new key events are lost.

Each key event is represented as a 16-bit word containing the 7-bit key number 0 to 127 and the pressed/released information in the bit 7. If the key is pressed the bit 7 is set, if the key is released the bit 7 is zero. The bits 8 to 15 contain the event source.

| b15 | b14 | b13 | b12 | b11 | b10 | b9 | b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Event souce | | | | | | | | Press ed | Key number | | | | | | |

Key mapping as well as the event source numbering are hardware dependent, see the respective hardware documentation.

Each key press and key release generates an event, e.g. if a key number 10 is pressed and released two events are generated, the first event with number `138` and the second event with number `10`.

The key-event queue is accessible to the application via the file handle `-2` interface using the following calls:

`LASTLEN ( -2 )`
Returns the number of key-events in the queue.

`READ -2, S$, E`
Removes the topmost key-event from the queue and returns it in `S$`. If `E` is non-zero and the queue is empty blocks until the next event arrives (wait for a key-press or key-release). Otherwise returns immediately.

## 7.8.3 IR interface (audio devices only)

An external infrared remote-control receiver attached to the first serial port can be read via the special handle -3. This handle is always present, no need to call open or close.

The commands sent by the remote-control are decoded on a high level and, the format is autodetected. See the Exstreamer Technical Documentation for more detailed description of the format.

If a file `remote.ini` is present in the device's FLASH memory, the received commands are automatically searched in this file and the appropriate line-number (counting from 1) is returned as well. The structure of the file is the same as for the Exstreamer (a text file with one command per line, the lines start with the command code; see the Exstreamer Technical Documentation for more details). If the received code is not found in the `remote.ini` the returned line number is 0.

A repetitive command is signalised with a star ( '`*`'), the line number is zero in that case.

`READ -3, S$`
Returns a formatted string: `<3-digit line number>,<decoded command>`
E.g.

```
024,NE: 00FED02F
006,NE: 00FE6897
```

If no command has been received or the infrared receiver is not available returns an empty string (without blocking).

## 7.9 The Wiegand reader (Barionet 100 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
```

### 7.9.1 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 14.7 on page 88.

## 7.10 1-wire interface (Barionet 50 only)

An interface to the 1-wire bus on Barionet 50 allows to access any connected 1-wire device from the BCL, besides temperature sensors and real-time clock, which are handled by the firmware.

### 7.10.1 Device addresses

Every 1-wire device has a unique 64-bit address assigned in the factory. At Barionet startup the bus is scanned for all connected devices and their addresses are stored in a table in memory.

This table is mapped to the IO space from offset 1001 to 1200 and is accessible from the BCL application via `IOSTATE` commands as follows:

1001 – address of the first 1-wire device: low 32-bits
1002 – address of the first 1-wire device: high 32-bits
1003 – address of the second 1-wire device: low 32-bits
1004 – address of the second 1-wire device: high 32-bits
...

Device addresses are stored in little-endian format. Unused entries in the table have value 0.

**Example:**
To read the address of the fifth device use:

```
low=IOSTATE(1009)
high=IOSTATE(1010)
```

A temperature sensor with address 10-51-8B-FF-00-08-00-C4 will read as:

```
low=0xFF8B5110
high=0xC4000800
```

## 7.10.2 File interface

The 1-wire bus is accessed by sending transactions through a file-like interface. Transactions are queued and atomically executed by the 1-wire bus scheduler in the firmware. This way transactions from the BCL application and from the firmware (temperature sensors and real-time clock) can be multiplexed.

A transaction is a complete access to one or more devices on the bus, which cannot be interrupted by any other request. For instance a transaction to measure temperature is the following sequence of commands:

- bus reset
- select a device by address (MATCH ROM)
- start temperature measurement (CONVERT TEMPERATURE command)
- apply power for 1 second
- bus reset
- select a device by address (MATCH ROM)
- read the measured value (READ SCRATCHPAD)
- bus reset

`OPEN "OWR:" as H`
Open the 1-wire bus interface.

`WRITE H , S$`
Send a 1-wire transaction `S$` to the bus. Returns immediately. `LASTLEN` returns >0 on success or 0 if the 1-wire interface is busy. On success transaction is queued and executed asynchronously.

Only one transaction can be submitted at a time.

`FILESIZE ( H )`
Is used to test whether the last 1-wire transaction has been processed (returns value >=0) or still in processing (returns <0).

`READ H, S$`
Reads the response to the last 1-wire transaction. `LASTLEN` returns the number of bytes read (>=0).

`LASTLEN ( H )`
Returns the length of the last 1-wire response when called after `READ`, or error status of a `WRITE` command.

## 7.10.3 Bus transactions

A 1-wire transaction is a binary string consisting of one or several commands. Each command has zero or more parameters. Commands are not separated from each other by any special character, neither are commands separated from their parameters. A command sequence is terminated by binary zero.

**Example:**
***<cmd 1><par1><par2><cmd2><command 3><parameter 1><zero>***

16-bit parameters are stored in little-endian (low byte first) format.

| | | | |
|---|---|---|---|
| 'X' (0x58) | Bus reset | None | Issue bus reset |
| '?' (0x3F) | Enumeration | None | Rescan the bus for new devices. Please note that this command may change the order the devices appear in the address table, if devices were attached or removed. |
| 'T' (0x54) | Transmit | 1-byte `length` `length` bytes data | Send `length` bytes of data to the bus. Data follow after the `length` parameter. |
| 'R' (0x52) | Receive | 1-byte `length` | Receives `length` bytes of data from the bus and append to the receiving buffer. |
| 'A' (0x41) | Address | 2-byte `index` | Send 64-bit address of the `index`-th device to the bus. `Index` counts from 0 and is an index to the device address table. E.g. `index=1` is the device, whose address can be read with `IOCTL(1002)` and `IOCTL(1003)`. |
| 'F' (0x46) | Address family | 2-byte `index` zero terminated list of 1-byte `family codes` | Similar to the 'A' command. Sends 64-bit address of the `index`-th device from the listed device families. `Family code` is the first byte of device's address. The device is searched as by the 'A' command, but devices, which do not match any of the listed family codes are ignored (skipped).<br>E.g. if there are devices with family codes 0x10, 0x28, 0x10, 0x10, 0x22, 0x28 in the table, the 'F' command issued with `index=2` and `family codes=0x22,0x28` sends the address of the last device. |
| 'D' (0x44) | Delay µs | 2-byte `delay` | Waits `delay` microseconds. This command is useful when waiting for the 1-wire device to finish an operation (e.g. temperature measurement). |
| 'M' (0x4D) | Delay ms | 2-byte `delay` | Waits `delay` milliseconds. This command is useful when waiting for the 1-wire device to finish an operation (e.g. temperature measurement). |
| 'H' (0x48) | High power | None | Apply high power to the bus. This might be necessary for bus powered devices to perform certain operations (e.g. during temperature measurement). Please note that during high-power it is not possible to communicate on the bus. Therefore high power must be removed using the 'L' command, typically after certain time ('M' or 'D' commands). |

| | | | |
|---|---|---|---|
| 'L' (0x4C) | Low power | None | Remove high power from the bus (apply low power). Low power is the default bus state and allows data communication.<br>This command is used in combination with the 'H' command. |

**Important:**
A 1-wire transaction is a binary string and special attention must be taken when creating or manipulating it. Use `MIDSET` or `MIDCPY` to create a transaction.
For simplicity you can also create a transaction by assigning a text string into a variable, use the `\x` escape sequence to insert non-zero binary characters and use a padding character (e.g. dot or minus) where binary zero will be in the sequence. Then replace the padding characters with binary zero using `MIDSET`. See the following example.

## 7.10.4 Example

The following example reads the first DS18B20 temperature sensor and prints the current temperature to syslog.

```
      DIM response$(10)
      DIM req$(50)

      req$="X"                ' bus reset
      req$=req$+"T\x01\x55"   ' send "MATCH ROM" command
      req$=req$+"F--\x28-"    ' send address of the N-th sensor
                              ' from family 0x28(DS18B20)
      req$=req$+"T\x01\x44"   ' send "CONVERT TEMPERATURE" command
      req$=req$+"H"           ' apply strong power on the bus
      req$=req$+"M\xe8\x03"   ' wait 1000ms (0x3e8)
      req$=req$+"L"           ' stop strong power
      req$=req$+"X"           ' bus reset
      req$=req$+"T\x01\x55"   ' send "MATCH ROM" command
      req$=req$+"F--\x28-"    ' send address of the N-th sensor
                              ' from family 0x28(DS18B20)
      req$=req$+"T\x01\xbe"   ' send "READ SCRATCHPAD" command
      req$=req$+"R\x09"       ' read 9 bytes
      req$=req$+"X"           ' bus reset to clean up
      MIDSET req$,6,2,0       ' set family index for the first
                              ' address command
      MIDSET req$,9,1,0       ' terminate the first family command
      MIDSET req$,23,2,0      ' set family index for the second
                              ' address command
      MIDSET req$,26,1,0      ' terminate the second family command

      OPEN "OWR:" as 1        ' open 1-wire handle
2
      WRITE 1,req$            ' send request
1
      ' wait until the request is processed
      IF FILESIZE(1)<0 THEN GOTO 1
      READ 1,response$        ' read the response
      IF LASTLEN(1)=9 THEN    ' received 9 bytes of data -> success
            temp=MIDGET(response$,1,2)    ' raw temperature
            temp=temp*100/16        ' convert from the default
                                    ' resolution 1/16 deg C
```

```
      SYSLOG "the temperature is "+SPRINTF$("%0.2F",temp)+" C"
ENDIF

GOTO 2
' this part of the code is never executed
CLOSE 1
END
```

# 8 Audio interface (audio devices only)

BCL provides a high-level interface for decoding and encoding[1] 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.

Generally, the following audio formats are supported in encoding, decoding, RTP and raw format: MP3, PCM, µ-Law and A-Law (G711). See the product specific documentation which formats are supported by your hardware.

## 8.1 Opening audio

Audio input/output can be opened with the following syntax

```
OPEN "AUD:MODE,FLAGS,QUALITY,DELAY,FRAME_DURATION,SSRC" as H
```

### 8.1.1 The MODE parameter – audio format

| | |
|---|---|
| 1 | MP3 decoding (MPEG 1, 2 and 2.5, layer III decoding) |
| 2 | MP3 encoding (MPEG 1 and 2, layer III encoding) |
| 11 | Uncompressed audio (PCM, G711 and G722 encoding and/or decoding) |
| 13 | MP3 encoding with bit rate (MPEG 1, MPEG 2 and MPEG 2.5, layer III encoding) |

Configuration of parameters (e.g. sampling rate, number of channels, format, etc.) of each of the above audio modes are defined by the `QUALITY` field, see below.

NOTE: Please note that the audio modes supported depend on the hardware device.

### 8.1.2 The FLAGS parameter – open options

1 Not available on Exstreamer devices.

| | | |
|---|---|---|
| 0 | 0 | read/write raw data |
| | 1 | read/write RTP frames |
| 1 | 0 | start playback immediately |
| | 1 | delayed playback<br>not used in encoding and in RTP |
| 2 | 0 | delay in milliseconds<br>valid only if delayed playback is set |
| | 1 | delay in bytes<br>valid only if delayed playback is set |
| 3 | reserved, always 0 | |
| 4 | 0 | no rebuffering |
| | 1 | automatic rebuffering on buffer underrun (MP3 decoding only; ignored in RTP) |
| 5 | 0 | PCM data in big-endian format (Msb first) |
| | 1 | PCM data in little-endian format (Lsb first) |
| 7 | Valid only in RTP mode. Allows to use the RTP buffering algorithm with raw UDP. If set, then raw UDP (frames without the RTP header) are expected on the input. | |

## 8.1.3 The QUALITY parameter – sampling rate, etc.

### *8.1.3.1 MP3 decoding*

For MP3 decoding the QUALITY parameter is ignored.

### *8.1.3.2 MP3 encoding*

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

| | |
|---|---|
| 0..3 | Interpreted as a 4-bit integer<br>0 MPEG2/22.05kHz<br>1 MPEG1/44.1kHz (MP3)<br>2 MPEG2/24kHz<br>3 MPEG1/48kHz (MP3)<br>4 MPEG2/16kHz<br>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 , 0=enable |
| 10 | bitreservoir[1] 1=kept empty, 0=used |
| 11 | 0=stereo, 1=mono encoding |
| 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 |

### *8.1.3.3 MP3 encoding with bitrate*

MP3 encoding with bitrate allows to encode MP3 in VBR, CBR or ABR mode with a configured bitrate rather than quality as above. The following table describes configuration bits of the `QUALITY` parameter.

| | |
|---|---|
| 0..3 | Interpreted as a 4-bit integer<br>0 MPEG2/22.05kHz<br>1 MPEG1/44.1kHz (MP3)<br>2 MPEG2/24kHz<br>3 MPEG1/48kHz (MP3)<br>4 MPEG2/16kHz<br>5 MPEG1/32kHz (MP3) |
| 4..9 | Bit rate in kbps/8 |
| 10 | bitreservoir[2] 1=kept empty, 0=used |
| 11 | 0=stereo, 1=mono encoding |
| 12..13 | Bit rate mode:<br>0=reserved, 1=VBR, 2=ABR, 3=CBR |
| 14..15 | reserved |

NOTE: Please note that MP3 encoding with bitrate is available only on VS1063 codec based hardware (IPAM 102, IPAM 302).

### *8.1.3.4 Uncompressed modes*

In Uncompressed mode encoder and decoder can run independently up to the sampling rate of 32kHz. At higher sampling rates (44.1kHz and 48kHz) only uni-directional audio is supported. The encoder and the decoder use the same settings of the audio format, number of channels and sampling rate.
The configuration is listed in the following table:

1 In RTP encoding make sure that bitreservoir is kept empty otherwise audio glitches may appear on frame loss.
2 In RTP encoding make sure that bitreservoir is kept empty otherwise audio glitches may appear on frame loss.

<table>
<tr><td></td><td></td></tr>
<tr><td>0..7</td><td>Sampling rate in kHz:<table>
<tr><td>8</td><td>8000 Hz</td></tr>
<tr><td>11</td><td>11025 Hz</td></tr>
<tr><td>12</td><td>12000 Hz</td></tr>
<tr><td>16</td><td>16000 Hz</td></tr>
<tr><td>22</td><td>22050 Hz</td></tr>
<tr><td>24</td><td>24000 Hz</td></tr>
<tr><td>32</td><td>32000 Hz</td></tr>
<tr><td>44</td><td>44100 Hz</td></tr>
<tr><td>48</td><td>48000Hz</td></tr>
</table></td></tr>
<tr><td>8..9</td><td>Audio format:<table>
<tr><td>0</td><td>µ-Law</td></tr>
<tr><td>1</td><td>A-Law</td></tr>
<tr><td>2</td><td>PCM</td></tr>
<tr><td>3</td><td>G.722</td></tr>
</table></td></tr>
<tr><td>10</td><td>Enable encoder: 1= enable, 0= disable</td></tr>
<tr><td>11</td><td>Enable decoder: 1= enable, 0= disable</td></tr>
<tr><td>12</td><td>Stereo: 0= mono, 1=stereo</td></tr>
<tr><td>13</td><td>Acoustic Echo Cancellation: 1= enable, 0= disable<br><br>AEC available only in full-duplex mode (both encoder and decoder active) and on hardware that supports AEC.</td></tr>
<tr><td>14..15</td><td>Reserved, set to 0</td></tr>
</table>

### 8.1.4 The DELAY parameter – delayed playback

In raw decoding-only mode, the start of decoding can be delayed if the delay bit is set in `FLAGS`. The `DELAY` parameter contains the delay in bytes or milliseconds, depending on the flags used.

This feature is not available in raw full-duplex mode.

In RTP mode (decoding-only and full-duplex) `DELAY` is used to set the initial buffer level, regardless of the `FLAGS` value. Read more in section 8.2.3 on page 55.

### 8.1.5 RTP encoder parameters FRAME_DURATION and SSRC

The last two parameters are valid only for RTP encoder. In other cases they can be omitted (the open string ends with the `DELAY` parameter). Read more in section 8.2.3.2 on page 56 and section 8.2.3.3 on page 56.

## 8.2 Data formats

### 8.2.1 PCM audio data

By default, 16-bit PCM data (Uncompressed mode) are expected to be in the **big-endian** (Motorola, Msb first) format. To swap the endianity to the **little-endian**

(Intel, Lsb first) format, set the bit 5 of the FLAGS parameter passed to the OPEN function.

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

Note: The endianity 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 8.2.3 below for more details.

### 8.2.2 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 14.1 on page 85 and example 14.2 on page 85.

### 8.2.3 RTP data mode

In RTP mode the decoder automatically manages the decoding buffer using the additional information provided within the stream. The decoder performs the following:

- Corrects the network jitter and recovers lost frames
- Corrects long term clock drift between the source and the decoder
- Manages constant decoding delay within one frame and this way allows synchronisation of multiple devices
- Synchronises to one stream using SSRC, sequence numbers and time stamps in the RTP header
- In case the audio buffer reaches zero level (no data from the source) or if the time stamp sequence suddenly significantly changes (e.g. the source resets) the decoder automatically restarts and re-synchronises to the new packet sequence.

#### *8.2.3.1 Initial delay*

The RTP decoder keeps a constant delay with an accuracy of one RTP frame (the frame size depends on the audio source and format used). The initial delay in milliseconds is specified by the DELAY parameter upon OPEN. The delay bits (bits 1 and 2) in the FLAGS parameter are ignored.

For correct operation, the initial delay must be set to at least the codec internal buffer size plus the time of the network jitter and possible lost frames. The following table lists the sizes of the codec internal buffers:

| | | |
|---|---|---|
| Micronas | MP3 | 1780 |
| Micronas | PCM | 320 |
| VLSI (non AAC+) | MP3 | 2048 |
| VLSI (non AAC+) | PCM | 2048 |
| VLSI (AAC+) | MP3 | 2048 |
| VLSI (AAC+) | PCM | 2048 |

#### *8.2.3.2 Frame duration*

The encoder automatically builds RTP header for every outgoing frame.

In MP3 mode every RTP packet carries one MP3 frame. The frame duration is given by the duration of the MPEG frame and depends on the MPEG format and sampling frequency used.

In PCM mode the frame duration is configurable by the optional `FRAME_DURATION` parameter to audio open. Select the frame duration in samples. To convert a value in milliseconds to samples use the following formula:

$$\frac{milliseconds \cdot samplerate}{1000}$$

Where *`samplerate`* is the sampling rate in Hertz.

By default, if not specified or set to 0, the frame duration of 20 milliseconds is used.

The frame duration is limited by the maximum UDP packet size of 1300 bytes. If `FRAME_DURATION` is set to a value, which exceeds the maximum packet size, it is automatically clipped to the maximum size.

The maximum payload size in bytes for the encoder can be calculated from the following formula:

$$min(1300, \frac{20 \cdot chans \cdot Bps \cdot samplerate}{1000})$$

Where *`chans`* is the number of channels, *`Bps`* is the number of bytes per sample and *`samplerate`* is the sampling rate in Hertz.

The maximum payload size for MP3 is 1400 bytes.

#### *8.2.3.3 SSRC*

An RTP stream is identified at the source (encoder) by a 32-bit SSRC identifier, which is transferred in every RTP frame. This helps to identify multiple streams coming from the same IP address. The SSRC field is configurable by the optional `SSRC` parameter at open.

`READ H, S$`

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 examples 14.4 and 14.5 on page 86.

### 8.2.1 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.

| | | | | |
|---|---|---|---|---|
| 0 | µ-Law, 8bit, mono, 8kHz | 11 | 0x0008 | - |
| 8 | A-Law, 8bit, mono, 8kHz | 11 | 0x0108 | - |
| 9 | G.722, mono, 16kHz | 11 | 0x0310 | |
| 10 | PCM 16bit, MSB first, signed, 44.1kHz stereo, left channel first | 11 | 0x122C | bit5=0 |
| 11 | PCM 16bit, MSB first, signed, 44.1kHz mono | 11 | 0x022C | bit5=0 |
| 14 | MPEG audio | 1,2 | - | - |
| 96 | PCM, 16bit, MSB first, signed, 8kHz mono | 11 | 0x0208 | bit5=0 |
| 97 | µ-Law, 8bit, mono, 24kHz | 11 | 0x0018 | - |
| 98 | A-Law, 8bit, mono, 24kHz | 11 | 0x0118 | - |
| 99 | PCM, 16bit, MSB first, signed, 24kHz mono | 11 | 0x0218 | bit5=0 |
| 100 | µ-Law, 8bit, mono, 32kHz | 11 | 0x0020 | - |
| 101 | A-Law, 8bit, mono, 32kHz | 11 | 0x0120 | - |
| 102 | PCM, 16bit, MSB first, signed, 32kHz mono | 11 | 0x0220 | bit5=0 |
| 103 | PCM 16bit, MSB first, signed, 48kHz stereo, left channel first | 11 | 0x1230 | bit5=0 |
| 104 | PCM, 16bit, LSB first, signed, 8kHz mono | 11 | 0x0208 | bit5=1 |
| 105 | PCM, 16bit, LSB first, signed, 24kHz mono | 11 | 0x0218 | bit5=1 |
| 106 | PCM, 16bit, LSB first, signed, 32kHz mono | 11 | 0x0220 | bit5=1 |
| 107 | PCM 16bit, LSB first, signed, 44.1kHz stereo, left channel first | 11 | 0x122C | bit5=1 |
| 108 | PCM 16bit, LSB first, signed, 48kHz stereo, left channel first | 11 | 0x1230 | bit5=1 |
| 109 | µ-Law, 8bit, mono, 12kHz | 11 | 0x000C | - |
| 110 | A-Law, 8bit, mono, 12kHz | 11 | 0x010C | - |
| 111 | PCM, 16bit, MSB first, signed, 12kHz mono | 11 | 0x020C | bit5=0 |
| 112 | PCM, 16bit, LSB first, signed, 12kHz mono | 11 | 0x020C | bit5=1 |
| 113 | PCM, 16bit, MSB first, signed, 24kHz stereo, left channel first | 11 | 0x1218 | bit5=0 |

| | | | | |
|---|---|---|---|---|
| 127 | Generic (see below) | - | - | - |

(*) Quality does not reflect bits 10 and 11 (encoder and decoder settings), these can be set independently.

Payload types 0, 8, 10, 11 and 14 are defined by the RTP standard. Barix defines assignment for payload types 96 to 112 (dynamic payload types) in the above tables.

## 8.3 Reading audio status

Various audio status information 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
```

Table 1: Reading the audio device status

| | | | |
|---|---|---|---|
| 1 | IN peak left | left channel | input linear audio peak level (see Note1 below) |
| 2 | IN peak right | right channel | |
| 3 | OUT peak left | left channel | output linear audio peak level (see Note1 below) |
| 4 | OUT peak right | right channel | |
| 5 | bitrate | bitrate in kbits/s | |
| 6 | cur buf level | current output buffer level in bytes | |
| 7 | avg buf level | average output buffer level in bytes<br>(since last cca. 5 seconds) | |
| 8 | input MUX status | current setting of the input MUX (see Input MUX control Error: Reference source not found) | |
| 9 | zero count | The decoder's audio buffer is monitored in 100ms intervals for empty buffer condition. If the buffer level drops to zero within the 100ms interval, zero count is increased by one. This counter is reset with every open.<br>Zero count is updated even if playback active is zero. | |
| 10 | RTP lost frames | Number of lost frames during RTP decoding. The value is reset with every new RTP sequence | |
| 11 | RTP dupl frames | RTP decoder only: number of duplicated frames due to the buffer management (frames are duplicated if encoder is slower than decoder) | |
| 12 | RTP drop frames | RTP decoder only: number of dropped frames due to the buffer management (frames are dropped if encoder is faster than decoder) | |
| 13 | IN peak log left | left channel | input audio peak level in dbFS (see Note2 below) |
| 14 | IN peak log right | right channel | |
| 15 | OUT peak log left | left channel | output audio peak level in dbFS (see Note2 below) |
| 16 | OUT peak log right | right channel | |
| 17 | playback active | If audio is open for decoding (or simultaneous decoding and encoding), typically audio buffer is first filled to a certain level and first then the playback starts. This status variable shows (playback_active is non-zero) if the playback has already started (data are being transmitted to the codec).<br>This status variable is independent on the encoder.<br>Please note that in some cases playback_active can change to 0 even during already active playback. This can happen if rebuffering on buffer underrun is selected in audio flags or if a new RTP sequence arrives. | |

| | | |
|---|---|---|
| 18 | resync count | RTP decoder only: this counter increments every time the RTP decoder synchronises to a new stream. This can be due to a buffer underrun or if a frame with a timestamp significantly out of sync is received from the same source.<br>This counter together with “zero count” can be used to detect errors in the stream.<br>The counter resets with every audio open. |
| 19 | cur buf ms | RTP decoder only: current duration of the buffered data in milliseconds |
| 20 | avg buf ms | RTP decoder only: average duration of the buffered data in milliseconds |
| 21 | Last packet timestamp | Time stamp in milliseconds (value of `_TMR_(0)` ) of the last successful write into audio handle (i.e. decoding).<br>The time stamp is initialized on opening the audio handle and updated with every successful data write into audio handle or with every audio data block transfer via the `LINK` command.<br>For more details see chapter 8.8.3. |
| 22 | Min jitter | Minimum and maximum jitter of incoming RTP packets in milliseconds.<br><br>The jitter is calculated as a difference of the RTP packet delivery time to the nominal clock given by the timestamp in the RTP packet. Lost packets don't have any influence on the calculation as soon as the sender properly sets the timestamp.<br><br>Min and max values are measured for 4 seconds and then buffered. The last measurement is returned (i.e. the values change every 4 seconds). |
| 23 | Max jitter | |

NOTE 1:
Linear audio peak levels are in the range 0 – 0x7FFF, linear scale:
0000 = 0%
2000 = 25% (-12dBFS)
4000 = 50% (-6dBFS)
7FFF = 100% (0dBFS)
NOTE 2:
Logarithmic audio peak levels are in dbFS units (dB full-scale). Value 0dBFS refers to the full-scale value of the AD/DA converter. The peak value is always <=0.

See also example 14.2 on page 90.

## 8.4 Setting audio parameters

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

`INDEX` determines the parameter to be set and `VALUE` is the value to be set.
The below table lists all available parameters. Parameters not supported by the hardware are ignored.

| | | | | |
|---|---|---|---|---|
| 1 | mic gain | Microphone input gain settings | 0..15 (in 1.5dB steps, starting at 21dB)<br>default: 0 (21dB)<br><br>On VLSI (IPAM 102) based hardware the following values are valid:<br>-6...+15 (in 1.5dB steps from 12dB up to 43.5dB) | |
| 2 | AD gain | A/D converter gain settings. Please note that the A/D gain applies for any analog input including the microphone input. Therefore for microphone input both gains are added up | 0..15 (in 1.5dB steps, starting at -3dB)<br>default: 0 (-3dB) | |
| 3 | input source | Input source switch | 1 | line in (default) |
| | | | 2 | microphone<br>(*) see Note 2 below |
| | | | 4 | SPDIF optical |
| | | | 5 | SPDIF coaxial |
| 4 | analog input mode | Analog input stereo/mono settings.<br>(*) see Note1 below | 0 | mono (left channel only) |
| | | | 1 | stereo (default) |
| 5 | output mode | Analog output settings.<br>If mono is selected both channels are mixed together and the signal is output on both channels.<br>Bridge is the same as mono with the difference that the polarity on one channel is inverted | 0 | stereo (default) |
| | | | 1 | mono |
| | | | 2 | bridge |
| 6 | output volume gain | Output gain adjustment<br>This parameter offsets the linear volume by the given amount of dB. It is used only if volume_type is 0 (linear volume). If logarithmic volume is selected this parameter is ignored. | in dB<br>default: 0dB | |
| 7 | loudness | Loudness adjustment on analog output. Linear from 0 (off) to 20 (maximum) | 0..20<br>default: 0 (off) | |
| 8 | balance | Balance control on analog output. | -10..10 (-10=left, 10=right)<br>default: 0 (center) | |
| 9 | treble | Treble adjustment on analog output. | -10..10<br>default: 0 (no correction) | |
| 10 | bass | Bass adjustment on analog output | -10..10<br>default: 0 (no correction) | |

| | | | | |
|---|---|---|---|---|
| 11 | volume type | Volume unit selection. Volume can be set either in dB or in percent<br>(*) see Note3 below | 0 | 5% steps (default) |
| | | | 1 | dB |
| 12 | volume | Volume (amplification) settings for analog output<br>(*) see Note3 below | (see the previous row)<br>default: 10 (50%) | |
| 13 | output mixer | Analog output mixer settings | bits 0..6: Rec. input | linear scale<br>0:off, 64: 100%, 127:200% |
| | | | bits 8-14: Playback | linear scale<br>0:off, 64:100%, 127:200% |
| | | | Default: 0x4000<br>(100% playback, rec input off) | |
| 14 | 1st microphone control | Microphone type settings for the first microphone | 0 | off |
| | | | 1 | dynamic (phantom power off) - default |
| | | | 2 | electret (phantom power on) |
| 15 | external amplifier control | External amplifier control | 0 | enabled (default) |
| | | | 1 | disabled |
| 16 | reserved | reserved | reserved | reserved |
| 17 | AUX 1 control | First auxilliary audio port control | 0 | disabled (default) |
| | | | 1 | half-duplex input |
| | | | 2 | half-duplex output |
| 18 | Input MUX control | Certain hardware devices feature an input multiplexer, where audio connections are shared between input and output.<br>This setting enables to control the multiplexer.<br>The actual state of the multiplexer can be read from the status parameter Input MUX status. The value may differ from the one set if the multiplexer is forced by a hardware switch. | 0 | input only (default) |
| | | | 1 | full- duplex |
| | | | 2 | output only |

| | | | |
|---|---|---|---|
| 19 | I/O mask send | For RTP with LINK command only.<br>Bitmask of digital inputs to be sent in RTP header extension. Bit 0 corresponds to the first digital input, bit 1 to the second digital input, etc. | Binary 1 in the mask enables sending of the respective input, whereas binary 0 disables sending of the input.<br>To completely disable sending header extensions, set all bits in the mask to 0 (default). |
| 20 | I/O mask receive | For RTP with LINK command only.<br>Bitmask of relays to be controlled by the remote party through RTP header extension. Bit 0 corresponds to the first relay, bit 1 to the second relay, etc. | Binary 1 in the mask enables remote control of the respective relay, whereas binary 0 disables remote control of the relay.<br>It is still possible to control relays with IOCTL, however relays with 1 in the mask will change according to the stream, as soon as the next RTP frame is received.<br>To completely disable remote control of relays, set all bits in the mask to 0 (default). |
| 21 | Tone Control: treble freq | Tone control setting. Limiting frequency for the treble high-shelf filter. Use together with the “treble” setting above.<br>(*) see Note4 below | Value in Hz. Operating range 1000-15000Hz, 1000Hz resolution.<br>Default 4000Hz. |
| 22 | Tone Control: bass freq | Tone control setting. Limiting frequency for the bass low-shelf filter. Use together with the “bass” setting above.<br>(*) see Note4 below | Value in Hz. Operating range 20-150Hz, 10Hz resolution.<br>Default 100Hz. |
| 23 | 5-Band Equalizer Enable | Selects the tone control either by bass/treble settings or using 5-band parametric equalizer.<br>(*) see Note5 below | 0 = disable 5-band equalizer and enable Bass/Treble control<br>1 = enable 5-band equalizer and disable Bass/Treble control<br>Default: 0 |
| 24-28 | 5-Band Equalizer Level | Amplification/attenuation level in 0.5 dB step for each band of the 5-band parametric equalizer<br>(*) see Note5 below | Level from -32 (-16dB; maximum attenuation) to +32 (+16dB; maximum amplification). Value 0 means no change.<br>Default: 0 for all parameters<br>(**) see Note6 below |

| | | | |
|---|---|---|---|
| 29-32 | 5-Band Equalizer Frequency | Cut-off frequency in Hz for bands 1 to 4 of the 5-band parametric equalizer. The last band does not have a cut-off frequency.<br><br>(*) see Note5 below | The frequency is limited for each band as follows:<br><br>Bass (29): 20-150Hz<br>Mid-bass (30): 50-1000Hz<br>Mid (31): 1000-15000Hz<br>Mid-high (32): 2000-15000Hz<br><br>The frequencies must be strictly ascending; e.g. combination 80, 50 is not allowed.<br><br>Default: 0 for all parameters |

NOTE 1: The `analog input mode` affects only the analog input switch and not the number of channels in the encoded stream. It is still possible to encode stereo stream from a mono input (then both channels will contain the same data and double bandwidth will be used). To change the number of encoded channels, see the `QUALITY` parameter of the `OPEN` call.

If `mono` input is selected then only the signal from the left channel is sent to the encoder (left channel is copied into the right channel).

SPDIF inputs are always stereo and are not affected by the `analog input mode` setting.

NOTE 2: If `microphone` source is selected `analog input mode` setting is ignored and mode is forced to mono.

NOTE 3: Logarithmic volume (amplification) is set in 1dB steps from -127dB to (theoretical maximum) +127dB. The value -127dB mutes the output. The maximum volume depends on the hardware type and is typically 0dB, however, on some devices the maximum volume can be up to +48dB. Values above the maximum volume given by the hardware are clipped to he maximum volume.

NOTE 4: Available only on VLSI based devices.

NOTE 5: **5-Band Parametric Equalizer**
Allows to adjust the audio sound in 5 configurable frequency bands: bass, mid-bass, mid, mid-high, treble.
The equalizer is available only on VS1063 based devices. On other devices settings 23-32 are ignored.
Either bass/treble control or 5-band equalizer can be active at a time, not both at the same time. The selection is done by setting parameter 23. Default is bass/treble control.

NOTE 6: In order to avoid clipping the equalizer does not amplify over 0dB volume level. I.e. the total amplification of `volume+level` for each band cannot exceed 0dB. Levels that would result in amplification above 0dB are clipped to 0dB. Therefore the perceived effect of the equalization can be different at full volume than at lower volume.
If you intend to boost certain frequency bands use the `output volume gain`

(parameter 6) to reduce the overall amplification by the maximum boost level to achieve the same equalization independent on the volume setting.
**Example:** Equalizer level vector +12,+4,-5,-3,+4 boosts bass and treble and attenuates the mids if volume is lower than -6dB. However at volume level 0dB (maximum volume) it only attenuates the mids, since bass and treble cannot be boosted over the 0dB level. In order to have the same effect independent on the volume set `output volume gain` to -6dB.

## 8.5 Flushing decode buffer

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

## 8.6 Flushing encode buffer

The encoding buffer can be flushed by calling SEEK on the audio handle, i.e.:
`SEEK H,0`

## 8.7 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.

## 8.8 Audio tunelling (audio devices only)

To increase the performance in audio encoding and decoding, a direct link between a UDP handle or a file and the audio can be established using the `LINK` command. The audio data is then copied by the interpreter without assistance of the BCL program. This feature is useful especially for encoding and decoding high-quality uncompressed audio transferred via RTP.

The `LINK` command has the following syntax:

`LINK H1, H2 [ ,A ]`
Creates a tunnel between handles `H1` and `H2`. Both handles must be open, one as UDP or as USB file and one as audio. The order of the handles in the command is not significant. For a link between audio and UDP an additional parameter `A` containing a table of destinations is required.

For link between audio and file the audio handle should be open for decoding (raw or MP3).

For link between audio and UDP the audio handle can be open as an encoder, decoder or in full-duplex. The audio format can be raw or RTP.

NOTE: In order to use LINK with raw audio data open the audio interface with flags 129 (RTP buffering algorithm, headerless).

### 8.8.1 File playback

Once the LINK command is executed a background process starts reading the USB file and copying the data into audio. The playback stops at the end of the file and the file position is equal to the file size.

During the playback audio parameters (e.g. volume) can be set, however no data should be written to the audio handle using `WRITE`.

An example file player:

```
OPEN "C_R:usb:///file.mp3" AS 1     ' open USB file
size=FILESIZE(1)                    ' get the file size
OPEN "AUD:1,6,0,32000" AS 2         ' open audio for MP3 playback
WRITE 2,"7",-12                     ' set volume

LINK 1,2                            ' start playback

1 IF FILEPOS(1)<>size THEN GOTO 1   ' wait for the end of the file

CLOSE 1                             ' clean up and exit
CLOSE 2
END
```

### 8.8.2 Decoder

All incoming UDP traffic on the UDP handle is passed directly to the audio device, bypassing the BCL program. UDP frames not accepted by the audio device (if open in RTP mode all non-RTP frames and frames with non-coresponding payload type) are passed to the BCL program and can be read using UDP `READ`, either by polling or in an `ON UDP` handler.

### 8.8.3 Detecting end of stream

An audio stream transferred via RTP or UDP is often terminated by a timeout, unless special signaling between the two parties is used. If `LINK` is used the audio reception is completely handled by the firmware without assistance of the BCL application. In order to detect an end of stream by timeout the time stamp of the last packet received is remembered and reported in audio status, parameter 21 (see Last packet timestamp parameter on page 61).

Stream time out can be then detected by the application by reading `Last packet timestamp` parameter and comparing with the value of `_TMR_(0)`, even if `LINK` is used for audio transfer.

The following example demonstrates how to detect end of stream with 500ms timeout.

```
DIM dst(1,1)                                ' 2 destinations
                                            ' dummy for decoder only
                                            ' initialised to 0
DIM dummy$(1)
OPEN "UDP:0.0.0.0:3030" AS 1                    ' open UDP port
OPEN "AUD:11,17,"+STR$(&H1A30)+",16384" AS 2    ' open audio at 48kHz
WRITE 2,"7",-12                                 ' set volume

LINK 1,2,dst                                ' start receiving RTP audio

1
READ 2,dummy$,-21                           ' read last pkt. timestamp
timestamp = LASTLEN(2)

IF _TMR_(0)-timestamp>500 THEN              ' 500ms timeout detected
      SYSLOG "stream time out 500ms"        ' close handles and end
      CLOSE 1
      CLOSE 2
ELSE
      GOTO 1                                ' no timeout → read again
ENDIF

END
```

### 8.8.4 Encoder

Encoded audio is sent either in raw format or formatted as RTP (depending on the open parameters of the the audio open command) to the provided list of destinations `A`. `A` is a two-dimensional integer array of pairs: `(IP address, port)`. Entries with IP address equal to 0 are ignored (i.e. no data is transferred). Broadcast is achieved by setting the IP address to the broadcast address of the network (e.g. 192.168.0.255) .

The destinations can be altered in runtime by writing to the array `A`, however, the IP address and port should be changed atomically. This can be achieved by using the LOCK command.

The number of destinations is virtually not limited, the only limit is the performance of the hardware. The number of outgoing streams also depends on the stream bitrate (higher bitrate means higher system utilisation).

Standard audio `READ` can not be used with the tunnel feature as no data will be returned. Audio `WRITE` can be used in parallel with the tunnel. Audio status can be read and audio parameters can be set independently.

WRITE to the UDP handle can be used independently on the tunnel.

### 8.8.5 Examples

See also example 14.2 on page 90.

48kHz stereo PCM RTP decoder:

```
DIM dst(1,1)                                ' 2 destinations
                                            ' dummy for decoder only
                                            ' initialised to 0

OPEN "UDP:0.0.0.0:3030" AS 1                    ' open UDP port
OPEN "AUD:11,17,"+STR$(&H1A30)+",16384" AS 2    ' open audio
WRITE 2,"7",-12                                 ' set volume

LINK 1,2,dst
1 GOTO 1
```

32kHz full-duplex stereo PCM RTP encoder and decoder:

```
DIM dst(2,1)                                ' 3 destinations

OPEN "UDP:0.0.0.0:3030" AS 1                    ' open UDP port
OPEN "AUD:11,17,"+STR$(&H1E20)+",16384" AS 2    ' open audio
WRITE 2,"7",-12                                 ' set volume

' initialise destinations
dst(0,0)=RESOLVE("10.0.0.7")                ' IP address
dst(0,1)=3030                               ' port
dst(1,0)=RESOLVE("my.server.com")           ' IP address
dst(1,1)=10000                              ' port
dst(2,0)=RESOLVE("another.server.com")      ' IP address
dst(2,1)=1234                               ' port

LINK 1,2,dst
```

# 9 Miscellaneous functions

## 9.1 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>=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$("%lA", E)` (see above).

## 9.2 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

SYSLOG E$ [,E]

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

Code example:

```
SYSLOG "ALARM"
```

## 9.3 Cryptographic 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`"

# 10 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 technical 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 1 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 1 on the Barionet into the variable INP1.

# 11 SNMP Interface

SNMP walk and traps are supported in BCL. On audio platforms and Barionet 50 also a set of strings is supported. See the Barionet MIB for a complete list of objects.

## 11.1 Integers

The IO map is exported to the SNMP interface as a table of integers. See the product specific technical documentation and the Barix MIB for more details.

## 11.2 Text strings (audio devices and Barionet 50)

A set of user definable strings is accessible via the SNMP interface. The string variables are defined as `_Sxx$` where `xx` is a number from 1 to 64 providing a maximum number of 64 strings. The strings are stored in the table:
`iso(1).org(3).dod(6).internet(1).private(4).enterprises(1).barix(17491).products(1).abcl(5).gpt(1).gptTable(1)`

The maximum string length is 127, however if not specified otherwise, the string is allocated to the default size of 256 bytes (only the first 127 bytes are available to the SNMP interface).

Only the strings defined by the BCL program (allocated using the `DIM` statement) are available, the remaining strings are empty and do not consume any space in memory.

The `_Sxx$` strings can be accessed via SNMP walk, sent in a trap, but also used in string expressions in the BCL program as any other strings.

## 11.3 Traps

A trap can be sent by the BCL program:

### 11.3.1 Barionet 100

`TRAP E$, N1, [ N, ....]`
Sends an enterprise trap number `N1` to IP or DNS address `E$`. On the Barionet only the integers from the IO table can be sent in a trap. The trap can contain virtually any number of entries, the only restriction is that the expanded trap must fit into one UDP packet.

The objects of the integer IO table can be sent in a trap:
`iso(1).org(3).dod(6).internet(1).private(4).enterprises(1).barix(17491).products(1).barionet(2).bariInputTable(2).bariInputEntry(1)`

Two objects of the table can be addressed: `bariInputIndex(1)` and `bariInputValue(2)`

The list of integer parameters `N` of the `TRAP` command following the IP address E$ are:

- `bariInputIndex(1)` – for N>10000, index N-10000 is addressed
- `bariInputValue(2)` – otherwise

**Example:**
To send a trap 3 containing the digital input 1 (IO 201) and digital input 2 (IO 202) to 192.168.0.254.

```
TRAP "192.168.0.254",3,10201,201,10202,202
```

The following objects will be sent in the trap:

- `enterprises.17491.1.2.2.1.1.201`
- `enterprises.17491.1.2.2.1.2.201`
- `enterprises.17491.1.2.2.1.1.202`

- enterprises.17491.1.2.2.1.2.202

### 11.3.1 Audio devices and Barionet 50

The trap interface on audio devices is more general.

TRAP E$, N1, N2, E1$, [ E2$ .... ]
Sends an SNMP trap to IP or DNS address E$. N1 is the trap type: public (0) or enterprise (6) and N2 is the trap number. Object IDs follow as E1$, E2$, etc. The trap can contain virtually any number of objects, the only restriction is that the expanded trap must fit into one UDP packet.

The format of the OIDs is the following: <type><OID subset string>
Where type denotes the public MIB branch (type is "P") or Barix enterprise MIB branch ("E"). The OID_subset_string is a user friendly ASCII formatted subset of the OID.

**Type:**
P is iso(1).org(3).dod(6).internet(1).mgmt(2).mib-2(1)
E is iso(1).org(3).dod(6).internet(1).private(4).enterprises(1).barix(17491)

**Example:**
To send enterprise trap 3 containing SysUpTime and text_entry 4 to "myhost.mydomain.com"
SysUpTime is P.system(1).sysUpTime(3).0
text_entry 4
E.products(1).abcl(5).gpt(1).gptTable(1).gptEntry(1).gptValue(2).4
The BCL variable for the text string 4 is _S04$.

```
DIM _S04$

_S04$="Hello World"

TRAP "myhost.mydomain.com",6,3,"P1.3.0","E1.5.1.1.1.2.4"
```

# 12 WEB interface

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

## 12.1 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);
```

### 12.1.1 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` is an integer variable or a cell of an integer array and `S$` is a string variable.

Syntax of the LBAS tag is the following:

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

where `format_string` coresponds to `SPRINTF` formatting (see chapter 5.6.2 on page 17) and variable is the name of the variable to be printed, or a cell in an integer array.

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 or an array subscript is out of range.

Following example displays the current uptime in seconds (the `_DTS_` variable) on the web page and some other variables:
HTML file:

```
&L(0,"*",1);
<html>
<head>
</head>
<body>
Time since reboot: &LBAS(1,"%lu",_DTS_);seconds.
A=&LBAS(1,”%ld”,A);
B(4)=&LBAS(1,”%ld”,B(4));
C$=&LBAS(1,”%fs”,C$);
D(3,7)=&LBAS(1,”%ld”,D(3,7));
</body>
```

```
</html>
```

### 12.1.2 Calling a subroutine from a webpage

Fore 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.

## 12.1 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.

## 12.2 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
```

# 14 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.

## 12.1 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. |

## 12.2 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
```

### 12.3 Predefined macros

The preprocessor directive `_COMPILETIME_` is replaced by the date and time of the tokenisation of the BCL source. It can be used in a `#define` statement to define a macro containing the build date of the program. E.g.

```
#define BUILD_DATE _COMPILETIME_

syslog "Starting program X (BUILD_DATE)"
```

Will print a message similar to:

```
Starting program X (2009/01/22 10:02:38)
```

# 13 Interpreter information

## 13.1 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.

## 13.2 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 **1-32767**
- maximal FOR-NEXT nesting **25**
- maximal GOSUB-RETURN nesting **23**
- maximal recursive nesting (amount of brackets) **10**
- maximum number of variables of type integer **255**
- maximum number of variables of type string **255**
- length of variable's names **unlimited**
- number of significant characters in variable names **32**
- 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 **16 ( max 6 TCP stream)**
- default size of buffers for files/streams **1024 bytes**

As the significant number of characters in a variable's name is 32 the tokenizer will issue an error when variables are defined using the same thirty two characters.

## 13.1 System variables

Several system variables interfacing various system parameters are predefined by the interpreter. These do not have to be declared by a DIM statement and can be used directly in the program. Calling DIM on system variables results in no operation.

| | |
|---|---|
| _ARG_ | holds the number of arguments given to the function, see section on page 29 |
| _CGI_ | used for CGI request handling, see section 12.2 on page 76 |
| _DTS_ | time counter, see section 6.7 on page 24 |
| _ERL_ | line number of the last error occurred, see section 6.7 on page 24 |
| _ERR_ | error code of the last error, see section 6.7 on page 24 |
| _TMR_ | array of time counters, see section 6.7 on page 24 |

# 14 Debugging

The Barix BCL interpreter allows debugging of programs using the syslog protocol[1], 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.

## 14.1 Error messages

| | |
|---|---|
| 0 | BCL file not exisiting 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...)<br>➔ TIMER 0 is obsolete, see section , page 97 |
| 7 | Wrong element is used in this string/numeric expression, maybe a type mismatch |
| 8 | Divided by Zero<br>➔ division by zero, for example: Var1=Var2/0 |
| 9 | Wrong label is used in GOTO/GOSUB statement (allowed only a numeric constant) |
| 10 | Wrong symbol is used in source code, syntax error, tokenization is impossible<br>➔ 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) |

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.

| | |
|---|---|
| 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<br>➔ 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<br>➔ 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 |
| 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<br>➔ can be caused by too long quoted string constant |
| 54 | Run-time warning: Lost data during UDP READ -- Please, increase string size<br>➔ 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)<br>➔ target string size (either set using DIM or the default) is insufficient |
| 58 | Run-time warning: Lost data during assignment of string -- Please, increase target string size (DIM statement)<br>➔ 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 |

# 17 Example programs

## 14.1 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
```

## 14.2 Record audio into an MP3 file

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 8 on page 51. For details about `_DTS_` variable see section 6.7 on page 24.

```
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
```

## 14.3 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: <joey@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$)=0 THEN GOTO 5
WRITE 0, "SUBJECT:Greetings\r\n"+&
      "From:joey@example.com\r\n"+&
      "To:agnes@example.com\r\n"+&
      "from joey\r\n"+&
      ".\r\n",0
6 READ 0,BUFFER$,0
IF LEN(BUFFER$)=0 THEN GOTO 6
WRITE 0,"QUIT",0
CLOSE 0
END
```

## 14.4 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$,-l
    GOTO 1099
```

## 14.5 RTP Sender

Encode MP3 from stereo analog input and broadcast as RTP on UDP port 5555.

```
    DIM _Mb$(4097)
    DIM rho$(50)

    rho$=SPRINTF$("%A",5)     ' local broadcast address

    OPEN "UDP:0.0.0.0:5555" AS 4
    OPEN "AUD:2,7,"+str$(&H71)+",0" as 7
    IF MEDIATYPE(7)=0 THEN SYSLOG "ERROR: Audio open failed"
    WRITE 7,"1",-4          ' stereo
    WRITE 7,"1",-3          ' source analog input
    i=1

10  READ 7,_Mb$
    l=LASTLEN(7)
    IF l>0 THEN WRITE 4,_Mb$,l,rho$,5555
    GOTO 10
```

## 14.6 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
```

## 14.7 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+MIDGET(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
```

## 14.1 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
```

## 14.2 RTP player with statistics

The below example demonstrates the usage of the LINK command in a simple RTP MP3 receiver. The program first opens UDP port 10000 and establishes a link with the audio interface. Then it periodically reads audio status and reports to syslog.

```
    DIM dst(0,1)                ' dummy destination for LINK
    DIM a$

    OPEN "UDP:0.0.0.0:10000" AS 1  ' open UDP receiver at port 10000
    OPEN "AUD:1,1,0,10000" AS 7    ' open audio for RTP

    WRITE 7,"7",-12             ' set volume

    LINK 1,7,dst                ' establish receiver link

2
    READ 7,a$,-5                ' bitrate
    br=LASTLEN(7)

    READ 7,a$,-6                ' current buffer level
    lev=LASTLEN(7)

    READ 7,a$,-9                ' zero count (buffer underrun)
    zero=LASTLEN(7)

    READ 7,a$,-10               ' lost frames
    lost=LASTLEN(7)

    READ 7,a$,-11               ' duplicated frames
    dupl=LASTLEN(7)

    READ 7,a$,-12               ' dropped frames
    drop=LASTLEN(7)

    SYSLOG
```

```
        "RTP statistics: bitrate="+STR$(br)+ &
        ", buffer level="+STR$(lev)+ &
        ", zero count="+STR$(zero)+ &
        ", lost frames="+STR$(lost)+ &
        ", duplicated frames="+STR$(dupl)+ &
        ", dropped frames="+STR$(drop)
    DELAY 1000
    GOTO 2
```

# 15 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**

## 15.1 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**

## 15.1 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 ")"

## 15.1 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", "RENAME", "SEEK", "SYSLOG", "TIMER", "TRAP", "WRITE", "LINK"
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", "FIND", "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**

# 19 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 6.5.3 on page 23.

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
```

## 19.1 Audio interface

The following audio modes are obsolete and no longer supported.

MODE:

| | |
|---|---|
| 3 | full-duplex PCM (16-bit signed, mono)<br>see the FLAG bits and chapter 8.1.3.4 above |
| 4 | full-duplex µ-Law |
| 5 | full-duplex A-Law |
| 6-8 | reserved |
| 9 | PCM stereo decoding (16-bit signed, stereo, left channel first)<br>see the FLAG bits and chapter 8.1.3.4 above |

QUALITY:
For full-duplex modes (modes 3, 4 and 5) specifies sampling rate in kHz – 8, 12, 24 or 32
For PCM stereo decoding the quality is either 44 for 44.1kHz sampling rate or 48 for 48kHz sampling rate.

# 20 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**

| | | |
|---|---|---|
| **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.

# 1 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

# Legal Information



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