## Table of Contents

### 1 Introduction

1.1 Notation

### 2 Development Tools

2.1 Editor
2.2 Tokenizer
2.3 Web2cob
2.4 TFTP upload
2.5 Batch files
2.6 Debugging

### 3 The BCL language

3.1 Syntax of the BCL language

3.1.1 Simple program
3.1.2 Comments
3.1.3 Command delimiters
3.1.4 Multi-line commands
3.1.5 Recommended structure of BCL programs

3.2 Integers

3.2.1 Integer constants
3.2.2 Integer variables
3.2.3 Integer expressions
3.2.4 Integer functions
3.2.5 Real numbers
3.2.6 Arrays of integers
3.2.7 Bit operations

3.3 Strings

3.3.1 String constants
3.3.2 Escape sequences
3.3.3 String expressions
3.3.4 String variables
3.3.5 Binary arrays

3.4 Execution flow control commands

3.4.1 The END command
3.4.2 Labels
3.4.3 Unconditional jump
3.4.4 The FOR-NEXT cycle
3.4.5 Subroutines
3.4.6 Conditional statements
3.4.7 Events and time
3.4.8 The LOCK command

3.5 Functions working with strings

3.5.1 String/Integer conversions

3.6 Binary array functions

3.7 User defined functions

3.8 I/O stream functions

3.8.1 The UDP network protocol
3.8.2 The TCP network protocol
3.8.3 Audio (audio devices only)
3.8.4 Serial port
3.8.5 SETUP
3.8.6 The CIFFS filesystem and the local USB filesystem (not supported on Barionet)
3.8.7 The local flash filesystem
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 certain 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 accessories 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:

<table>
<thead>
<tr>
<th>Notation symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Integer constant, value between -2147483648 and +2147483647 can be also written in hexadecimal notation (corresponding range is from &amp;H00 to &amp;HFFFFFFFF)</td>
</tr>
<tr>
<td>L</td>
<td>Line number. Line numbers are unsigned integers from 1 to 65535</td>
</tr>
<tr>
<td>Q</td>
<td>Quoted string constant of length up to 127 characters</td>
</tr>
<tr>
<td>S$</td>
<td>String variable (zero terminated by default). With some restrictions, string variables can be used to hold binary data (all possible 8-bit values, including 0)</td>
</tr>
<tr>
<td>V</td>
<td>Integer variable or array element V[E [, E] ]</td>
</tr>
<tr>
<td>H</td>
<td>File handle (integer in range 0..7)</td>
</tr>
<tr>
<td>F( )</td>
<td>Function returning integer</td>
</tr>
<tr>
<td>F$()</td>
<td>Function returning string</td>
</tr>
<tr>
<td>E</td>
<td>Expression of type integer, typically a result of arithmetic operations with N, V, and F( )</td>
</tr>
<tr>
<td>E$</td>
<td>Expression of type string, a result of a concatenation of Q, S$, and F$( )</td>
</tr>
<tr>
<td>bE</td>
<td>Boolean expression</td>
</tr>
<tr>
<td>[...]</td>
<td>Square brackets are used to indicate that the bracketed part is optional</td>
</tr>
<tr>
<td>{...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

Development of BCL programs can be done in any text editor of the programmers choice – as long as the editor supports standard ASCII files with CRLF newlines\(^1\), an example of such editor is the Notepad application shipped with the Microsoft Windows operating system. Modern development tools – like the free Eclipse development system – allow comfortable editing with syntax highlighting, the use of such tools is however optional.

BCL source program files are expected to have `.bas` extension with the exception of files to be preprocessed (for details, see section 6, page 26).

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 compile the ASCII BCL program code and to convert it into the Barix TOK format.

Command prompt call:

```
tokenizer program.bas
```

where `program.bas` is the name of the the source file, will result in creation of the tokenized program file `program.bas.tok`.

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

2.3 Web2cob

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

Command prompt call:

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

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

`/d` defines which folder should be packed

A cob file exceeding the size of 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.

2.4 TFTP upload

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

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

\[^1\] As common in DOS and Windows operating systems
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 that 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).
Direct usage of TFTP is recommended only for advanced programmers.

### 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 files:

```
    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 in an obvious way in order 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.

### 2.6 Debugging

The Barix BCL interpreter allows debugging of programs using the syslog protocol, 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"
```

---

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 provided free of charge by Barix can be used.
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.
3 The BCL language

3.1 Syntax of the BCL language

3.1.1 Simple program

We can write a simple one-liner and use it just to test that tokenizer, tftp uploading and BCL interpreter in the BCL
device are all working well:

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

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

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

3.1.2 Comments

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

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

When comes to the functionality this program is exactly the same as our first program. Of course the syslog output
will not differ:

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

3.1.3 Command delimiters

Most BCL statements can be delimited with space, tab, new line (CRLF) or ':' (colon) characters. Comments and
DIM statements must be terminated with CRLF.

3.1.4 Multi-line commands

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

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

3.1.5 Recommended structure of BCL programs

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

Code examples:

```bcl
DIM CR$(3)
.
END(EOF)
DIM CR$(3)
```
3.2 Integers

3.2.1 Integer constants
Integer constants (denoted N in this documentation) can be written as ordinary integers. They must be in the range from -2,147,483,648 to +2,147,483,647. They can be also written in hexadecimal notation using &H, eg. &H1A instead of 26.

3.2.2 Integer variables
Integer variables are identified by their name, of which only first five characters are significant. They can hold integers in the range allowed for integer constants.

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

\[ V = E \]

where \( V \) is the name of the variable and \( E \) is integer expression.

3.2.3 Integer expressions
Integer expressions can be formed using the following operators:

<table>
<thead>
<tr>
<th>Integer operators (descending priority of execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
</tr>
<tr>
<td>+, -</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td>(^*/,%)</td>
</tr>
<tr>
<td>+, -</td>
</tr>
</tbody>
</table>

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

3.2.4 Integer functions
Functions returning integer values are called integer functions. Several built-in functions are available and it is also possible to create user defined functions, see section 3.7, page 16.

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

SYSLOG "Computing the circumference and the area of a circle"

\[
\begin{align*}
\text{radius} & = 13 & \text{set radius} \\
\text{phi} & = 3.14 & \text{set approximately the value of phi} \\
\text{circum} & = 2 * \text{phi} * \text{radius} & \text{compute circumference} \\
\text{area} & = \phi * \text{radius}^2 & \text{compute the area}
\end{align*}
\]

SYSLOG "Given circle of radius "+STR(radius)+", the circumference is " & 
" + STR(circum)=" & STRS(circum\%100)+" and the area is " & 
" + STRS(area\%100)=" & STRS(area\%100)+" ."

3.2.6 Arrays of integers
It is possible to use one dimensional or two dimensional arrays of integers. Such arrays are declared using the DIM command with the following syntax:

\[
\begin{align*}
\text{DIM NAME(INDEX)} & \quad \text{for one dimensional array} \\
\text{DIM NAME(INDEX1, INDEX2)} & \quad \text{for two dimensional array}
\end{align*}
\]
where \texttt{NAME} is the name of the array and \texttt{INDEX}, \texttt{INDEX1}, \texttt{INDEX2} are the highest possible indices. As indexing of array elements starts from zero, an array declared with highest possible index \texttt{INDEX} will be of size \texttt{INDEX+1}. Elements of the array can be accessed using the syntax \texttt{NAME(INDEX)}, or \texttt{NAME(INDEX1, INDEX2)}. For example an array declared as \texttt{DIM NUM(5)}, has 6 elements numbered from 0 to 5.

Code Example:

```bcl
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) = NBW(0) - OLD(0)
DIFF(1) = NBW(1) - OLD(1)
DIFF(2) = NBW(2) - OLD(2)
```

### 3.2.7 Bit operations

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

- \texttt{NOT(\ldots)}
  Bitwise NOT operation.
- \texttt{AND(\ldots, \ldots \ldots \ldots, \ldots)}
  Bitwise AND operation.
- \texttt{OR(\ldots, \ldots \ldots \ldots, \ldots)}
  Bitwise OR operation.
- \texttt{XOR(\ldots, \ldots \ldots \ldots, \ldots)}
  Bitwise XOR operation.
- \texttt{SHL(\ldots, \ldots)}
  Bitwise shift left of \ldots by \ldots bits.
- \texttt{SHR(\ldots, \ldots)}
  Bitwise shift right of \ldots by \ldots bits.

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

### 3.3 Strings

#### 3.3.1 String constants

String constants are always quoted and their maximum length is 127 characters. An example of such constant is the “Hi, everything is OK” in the following program:

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

#### 3.3.2 Escape sequences

*Escape sequences* can be used to include special ASCII characters in a string constant.

<table>
<thead>
<tr>
<th>Escape sequences</th>
<th>ASCII character</th>
</tr>
</thead>
<tbody>
<tr>
<td>`a`</td>
<td>ASCII character 7</td>
</tr>
<tr>
<td>`b`</td>
<td>ASCII character 8</td>
</tr>
<tr>
<td>`\n`</td>
<td>ASCII character 10</td>
</tr>
<tr>
<td>`\f`</td>
<td>ASCII character 12</td>
</tr>
</tbody>
</table>
### Escape sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>ASCII Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r</td>
<td>13</td>
</tr>
<tr>
<td>\t</td>
<td>92 (backslash)</td>
</tr>
<tr>
<td>&quot;</td>
<td>34 (&quot;</td>
</tr>
<tr>
<td>\xhh</td>
<td>ASCII character with hexadecimal index equal to hh</td>
</tr>
</tbody>
</table>

#### 3.3.3 String expressions

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

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

#### 3.3.4 String variables

String variables are identified by their names (case insensitive), of which only first five characters are significant, the last character must be `$`. The tokenizer generates a warning message when variables are defined using the same first five characters. They can be assigned values using the assign operator `=` with syntax

```
$v$ = $E$
```

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

Example:

```
FIRSTs = "Hi,"
SECONDS = " everything OK!"
CONCATs = FIRSTs+SECONDS
SYSLOG CONCATs    // syslog concatenation
```

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

By default the maximum length of string variables is 256 characters. String variables longer than 256 characters must be declared using `DIM` command as in the following example:

```
DIM LONGs(600)       // LONGs can hold 599 characters
LONGs="............." // assign 8 dots
LONGs+LONGs+LONGs+LONGs+LONGs  // assign 32 dots
LONGs+LONGs+LONGs+LONGs+LONGs+LONGs+LONGs+LONGs+LONGs  // assign 128 dots
LONGs+LONGs+LONGs+LONGs+LONGs  // assign 512 dots
SYSLOG LONGs         // syslog 512 dots
```

This program sends a message 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 CR/LF) 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 CRs(3)
CRs="\n"
```

#### 3.3.5 Binary arrays

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

For string calculations BCL uses temporary buffer with 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 instructions), the string name should start with the "_M" prefix to avoid changing of the internal string buffer.

### 3.4 Execution flow control commands
3.4.1 The END command

The 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.

3.4.2 Labels

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

3.4.3 Unconditional jump

Unconditional jump to label L has the syntax:

```
GOTO L
```

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

Code Example:

```
10   "   
10   SYSLOG "Neverending cycle"
   GOTO 10
```

3.4.4 The FOR-NEXT cycle

The syntax of the FOR-NEXT cycle is:

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

First, V is assigned the result of the expression $E_1$. Then all statements up to the the first unmatched NEXT (or the NEXT statement with the correct variable) are executed. When the NEXT statement is reached, $V$ is incremented and compared with $E_2$. The execution restarts at the FOR statement as long as $V$ is less or equal than $E_2$. If $V$ is larger than $E_2$, the loop is terminated and the statement after the NEXT statement is executed.

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 COUNT=0 TO 25
   DIFF(COUNT)+NEW(COUNT)-OLD(COUNT) 'calculate differences
NEXT COUNT
```

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

NOTE: the programmer is strongly discouraged to use 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 $E_2 + 1$.

3.4.5 Subroutines

```
GOSUB L
```

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.

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

To end a subroutine, the \texttt{RETURN} command must be used, otherwise the calling stack of the interpreter is not cleared which may result in unpredictable behavior or device reset.

\subsection*{3.4.6 Conditional statements}

Condition evaluation and code branching are possible using the \texttt{IF} statement. \texttt{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 \texttt{THEN} statement are executed. Two syntax forms of the \texttt{IF} statement exist:

\subsubsection*{3.4.6.1 Multiline IF}

If \texttt{THEN} is the last statement on the line (excluding comments), a multiline \texttt{IF} statement is assumed and all following lines up to the first unmatched \texttt{ELSE} or \texttt{ENDIF} statement are executed. In that case the optional \texttt{ELSE} must be the last statement on the line as well.

Code Example:

\begin{verbatim}
10  CNT=0
20  CNT=CNT+1
< 500    20    10
 IF CNT < 500 THEN GOTO ELSE GOTO
ENDIF

SYSLOG MSG
\end{verbatim}

\subsubsection*{3.4.6.2 Single line IF}

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

Code Example:

\begin{verbatim}
10  CNT=0
20  CNT=CNT+1
< 500    20    10
 IF CNT < 500 THEN GOTO ELSE GOTO 10
\end{verbatim}

If the expression result is false (zero), execution continues after either the first unmatched \texttt{ELSE} statement or an \texttt{ENDIF} (multiline \texttt{IF}) or a CR/LF (single line \texttt{IF}).

\subsubsection*{3.4.6.3 Boolean expressions}

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

\begin{verbatim}
  E1 = E2, E1 > E2, E1 < E2, E1 = E2, E1 <> E2, E1 && E2, E1 || E2
\end{verbatim}

Simple comparison of strings is also possible:

\begin{verbatim}
  S1 = S2, S1 <> S2
\end{verbatim}

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

\begin{verbatim}
  NOT (    )
  \texttt{logical NOT operation}
  AND (., ., bE2 \ldots)
  \texttt{logical AND operation}
\end{verbatim}
**OR** \( \text{bE} \) \( \text{be} \) \( \text{ez} \) \( \text{e} \)

logical OR operation.

**XOR** \( \text{bE} \) \( \text{be} \) \( \text{ez} \) \( \text{e} \)

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: \( \text{AND(bE)}, \text{OR(bE)}, \text{XOR(bE)} \) with one argument return the value of expression \( \text{bE} \).

### 3.4.7 Events and time

Program can have limited event-driven structure using the \( \text{ON...GOSUB} \) construct.

```
ON ... TIMER 1 | 2 | 3 | 4 | CGI | UDP | GOSUB L
```

When the \( \text{ON...GOSUB} \) construct is interpreted, an event handler subroutine (indicated with a label/line number \( \text{L} \)) is entered in a table. Then if the matching event is triggered (e.g., a UDP block is received), the interpreter executes the registered subroutine.

Obviously, this subroutine should return as soon as possible with a \( \text{RETURN} \) statement to allow continuation of the normal program operation.

Code example: \( \text{ON CGI GOSUB 100} \).

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

Code example: \( \text{ON UDP GOSUB 0} \).

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

- 4 independent software timers (resolution in milliseconds). Timers must be set up using the \( \text{TIMER} \) statement
- incoming UDP packet (see section 3.8.1, page 18)
- incoming CGI request to a special handler

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

```
TIMER E0, E
```

Set the timer \( \text{E0} \) to trigger every \( \text{E} \) milliseconds. Timer is reset with this statement so it will be first triggered after \( \text{E} \) milliseconds.

Valid timers are 1, 2, 3, 4.

Code example: \( \text{TIMER 2,100} \),

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 \( \text{TIMER} \) statement) can be read from the special variable array variable _TMR_. Besides, _TMR(0) returns number of milliseconds since last hardware restart.

```
DELAY E
```

delays the execution of the program for \( \text{E} \) milliseconds (maximum possible delay is 65535 ms).
Code example:

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

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

Event handling is not affected by the delay.

**NOTE:**

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

System variable _DTS_, which is incremented every second, can be also used when writing time dependent operations.

**ON ERROR GOTO L**

This command stores the line number/label of the error handling subroutine. In case of an (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 is stored in system variables _ERR_ and _ERL_ respectively.

Note: The error handler is not triggered by warnings.

### 3.4.8 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 happen and the BCL interpreter will be the only running application. This is useful only in very specific 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)

<table>
<thead>
<tr>
<th>Bit index</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>snmp write</td>
</tr>
<tr>
<td>1</td>
<td>snmp read</td>
</tr>
<tr>
<td>2</td>
<td>modbus/tcp write</td>
</tr>
<tr>
<td>3</td>
<td>modbus/tcp read</td>
</tr>
<tr>
<td>4,7</td>
<td>reserved</td>
</tr>
<tr>
<td>8</td>
<td>rc.cgi</td>
</tr>
<tr>
<td>9</td>
<td>i/o dynamic tags</td>
</tr>
<tr>
<td>10</td>
<td>setup.cgi</td>
</tr>
<tr>
<td>11</td>
<td>setup dynamic tags</td>
</tr>
<tr>
<td>12</td>
<td>BAS.cgi</td>
</tr>
<tr>
<td>13</td>
<td>basi variable dynamic tags</td>
</tr>
<tr>
<td>14</td>
<td>Basic.cgi</td>
</tr>
</tbody>
</table>
Examples:

<table>
<thead>
<tr>
<th>Bit index</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>tftp</td>
</tr>
</tbody>
</table>

Lock 1,32768

disables the TFTP upload function, all other functions remain unchanged.

Lock 0, &HCO0

enables the setup functions (cgi and dynamic tags), all other functions remain untouched.

3.5 Functions working with strings

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

\[\text{LEN}(\ E_5\ )\]

Returns the length of the string \(E_5\), as an integer (excluding the terminating NULL character).

Example:

\[A+\text{LEN}(\ "\SHORT\")\]

will store 5 in variable \(A\).

\[\text{INSTR}(\ E_1, \ E_5, \ E_2)\]

Searches for \(E_2\) in \(E_5\) starting from the position indexed by \(E_1\). On success returns offset of the \(E_2\) in \(E_5\). Otherwise returns 0.

Example:

\[\text{}A=\text{"is it here!"}\]
\[\text{}B=\text{"i"}\]
\[\text{}\text{POS=INSTR}(2, A, B)\]

will store 4 in variable \(\text{POS}\) as we start the search from “s” on.

\[\text{MID}(\ E_5, \ E_1, \ E_2)\]

Returns the sub-string of \(E_5\) consisting of \(E_2\) characters starting from the position \(E_1\). In the case that \(E_2\) is 0, returns all characters from position \(E_1\) to the end of \(E_5\).

If a string variable is used as a binary array (in which case a string variable \(S_5\) must be used instead of a string expression \(E_5\)), it may contain several bytes with value 0, which can be located by using the \(\text{MIDS}(S_5, \text{index})\) function.

\[\text{LCASE}(\ E_5)\]

Returns a string produced by converting all characters of \(E_5\) to lower case.

For example, after executing

\[\text{OUTs=LCASEs(\"LoWeR\")}\]

the value of \(\text{OUTs}\) will be “lower”.

\[\text{UCASE}(\ E_5)\]

Returns a string produced by converting all characters of \(E_5\) to upper case.

Code example:

\[\text{OUTs=UCASEs(\"Upper\")}\]

Result: \(\text{OUTs = \"UPPER\"}\)

When writing a more complex BCL program using plenty of string variables it is recommended to decrease memory usage by dimensioning the string variables to their particular needs. This can be accomplished with the \(\text{DIM}\) statement.
For string calculations the BCL uses temporary buffers with size of the largest string variable declared (if it exceeds 256 bytes a warning will be sent to syslog). If a string is not going to be used for calculations, the string name should start with the "_M" prefix to avoid changing of the internal string buffer.

3.5.1 String/Integer conversions

**ASC(E) \text{\(E\)**}

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

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

\[N = \text{ASC(" there is a space at the beginning of this string")}\]

**CHR(E) \text{\(E\)**}

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

Example: \(S\) equals to " " after execution of

\[S = \text{CHR}(32)\]

**VAL(E) \text{\(E\)**}

Interprets the string \(E\) as the ASCII representation of an integer and returns the value. If \(E\) includes "." (dot char), \(VAL\) interprets \(E\) as an IP address and converts it into an integer value in the little-endian (Intel) format. For converting the other way around you can use \(SPRINTF\) (see below).

**STR(E) \text{\(E\)**}

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

**SPRINTF(E) \text{\(E\)} \text{\(E\)**}

Converts the integer value \(E\) into a string using C-style formatting specified with \(E\) and returns the string. The format string uses the common "C" notation but only one parameter is allowed. The parameter can be either an integer or a string.

Code example:

\[A = \text{SPRINTF("the value is ", 1922)}\]

will store the string "the value is: 1922" in the variable \(A\).

The following formats are supported:

- %[\([-|0\]n\)]u unsigned 16 bit integer
- %[\([-|0\]n\)]u unsigned 32 bit integer
- %[\([-|0\]n\)]d signed 16 bit integer
- %[\([-|0\]n\)]d signed 32 bit integer
- %[\([-|0\]n\)]x 16 bit hex value
- %[\([-|0\]n\)]x 32 bit hex value
- %c as character in ASCII
- %s string

where:

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

Version related:

- %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)

Network related:

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

\(x\) function

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)

Time related:
%xt formats a time string, the value x is bitwise OR of any combination of the below bits.

The full time format is: [yy]YYMMDDhhmmss w

x function
0 default time YYMMDDhhmm (e.g. 0405140914)
1 including seconds ss (e.g. 040514091459)
2 including the leading century yyYY (e.g. 200405140914)
4 adjust for a local time zone and DST (in future releases)
8 leading character for time valid ("2" not set, "3" set)
16 prints a 32-bit parameter (secs since 1/1/1970) instead of the system time
32 including w - one-digit week-day number in range 1-7 (e.g. 04051409145)

Code example:
```c
$ = $ ( "%3t", 0 )
```
Result: $ : "00490 6 0 6 0 0 0 0 0 0 2 ..."

```
STIME( $ )
```
returns the $ converted to seconds since 1/1/1970. Format of the $ string is "YYMMDDhhmmss". See also SPRINTFS.

### 3.6 Binary array functions

**MIDSET $s, 0, 1, E** replaces the E as a byte (E=1), a word (E=2), or a double word (E=4) at position E0 of the string variable $s (binary array).

Word and double word storage is done in the little endian (Intel) format by default. If E is negative (-1,-2,-4) the big endian format is used.

Code example:
```
BA$ = " " / hex 2020202000
MIDSET BA$,2,1,64
```
will result in $ (in hex): 2040202000

**MIDGET $s, 0, 1** Extracts an integer variable as byte, word, or double word (E=1, 2, and 4 respectively) from position E0 of the string variable $s (binary array).

The variable is retrieved in the little endian (Intel) format by default. If E is negative (-1,-2,-4) the big endian format is used.

**MIDCPY $s, 0, 1, S** replaces E bytes at position E0 of the string variable $s from the beginning of the string variable S.

Code example:
```
As= "Come here!"
Bs= "Look there!"
MIDCPY As,1,s,Bs
```
will result in As containing "Look here!"
### 3.7 User defined functions

Users can define their own functions in the BCL. First an integer variable with the same name as the function must be declared using the `DIM` statement. This variable is used to store a return value of the user-defined function. For example:

```
DIM fct \( \langle \text{GOSUB 1010} \rangle \)
```

where `fct` (factorial) is the name of the global integer variable, that can be used as a usual variable, but if it has parenthesis with or without a list of argument expressions, the subroutine at label 1010 will be called first in order to renew the `fct` value. For example the code:

```
fact=fct(6)
```

can be used to call and calculate the recursive factorial function:

```
1010 LOCAL y
    if y > 1 then fct = y⇒fct(y-1) else fct = y
    return
```

Really, this function calculates a factorial. Unfortunately only up to \( i=10 \) because the maximal nesting for recursive calculations is only 10. Therefore we do not recommend to use recursive functions.

The statement `LOCAL` may be used only once in the subroutine as the first statement. It declares the list of temporary variables and has the same syntax as the `DIM` command. The local variables which are used as function arguments must be listed first in this list. Only simple integer variables (not arrays) or string variables of the default size (without a size specification) can be used for arguments.

Any expressions can be passed to a function as arguments. The number of arguments is not limited and can be retrieved from the system variable `ARG`. Variables in the list which are not used to store any arguments may be of any type and will be initialized to null values.

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.

**Note:** During a user function execution no events can be captured and handled, therefore usually the user function subroutines should be rather short. If this is not possible the usual subroutine `GOSUB` should be used.

### 3.8 I/O stream functions

The BCL language supports variety of accessories and protocols for input and output. The same function set is used for different protocols but the functionality slightly differs depending on the protocol.

Simplified procedure for a 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
   - `S` is a string (not a general string expression) which determines the protocol and sets appropriate parameters (for details see descriptions of individual protocols below)
   - `H` is the handle number (integer). For most protocols the numbers 0,...,7 are allowed. An exception is the TCP protocol where only numbers 0,...,6 are allowed. Handle numbers are common for all protocols and the same handle cannot be opened for two different streams at the same time.
2. Using the stream with `WRITE`, `READ`, `SEEK` etc. (list of available functions depends on particular protocols) Handle number of the stream is given to functions to determine the stream. Multiple I/O commands can be used in this phase before closing the stream.
3. Closing the stream using `CLOSE(H)` function, where `H` is handle number.

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

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

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

```
CLOSE H
```

Closes the file or stream with handle `H`.
**WRITE** \( H, S, E \)

Writes \( E \) bytes from the \( S \) string variable into the stream \( H \).

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

To write a character with code zero, use an empty string \( S \) and \( E = 1 \).

**READ** \( H, S, I, E \)

Reads from the stream \( H \) into the string variable \( S \).

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

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

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

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

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

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

The second optional parameter \( E \) if given 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 read 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 -1.

**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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Media type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>CIFS - reading</td>
</tr>
<tr>
<td>4</td>
<td>CIFS - writing</td>
</tr>
<tr>
<td>5</td>
<td>CIFS – directory reading</td>
</tr>
<tr>
<td>6</td>
<td>TCP</td>
</tr>
<tr>
<td>7</td>
<td>UDP</td>
</tr>
<tr>
<td>8</td>
<td>Serial port (COM)</td>
</tr>
<tr>
<td>9</td>
<td>Flash read</td>
</tr>
<tr>
<td>10</td>
<td>Flash write</td>
</tr>
<tr>
<td>11</td>
<td>Flash append</td>
</tr>
<tr>
<td>13</td>
<td>Setup</td>
</tr>
<tr>
<td>14</td>
<td>Wiegand protocol</td>
</tr>
<tr>
<td>17</td>
<td>Audio</td>
</tr>
<tr>
<td>18</td>
<td>CIFS - append</td>
</tr>
</tbody>
</table>

**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.

FILESIZE (HANDLE)
Returns the file size (or -1 for directory) of file/stream HANDLE.
Not applicable for serial streams. For audio streams (AUD) returns the number of free bytes in the audio-decoding buffer.

3.8.1 The UDP network protocol

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

OPEN "UDP:<IP_address>:<port_number>" AS HANDLE

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

The following example command results in opening handle number 3 for listening to a UDP socket on port 1000:

OPEN "UDP:0.0.0.0:1000" AS 3

Syntax of the WRITE command is extended for the UDP protocol:
WRITE HANDLE, $, 0, $, 1

Additional parameter $ specifies the destination IP address and the parameter 1 specifies the destination port number.

When in receiving mode, the LASTLEN (HANDLE) returns a negative value if any new data are available for reading. After reading the value is positive unless new data have been received.

RMTPORT (HANDLE)
Returns the remote port of the stream HANDLE or 0 if not applicable. It can be used for a UDP reply to sender’s port number.

RMTHOST (HANDLE)
Returns the remote host IP for the stream HANDLE or an empty string. It can be used for a UDP reply to determine the IP address of the sender.

3.8.2 The TCP network protocol

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

OPEN "TCP:<IP_address>:<port_number>" AS HANDLE

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

Examples:

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

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

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

RMTHOST (HANDLE)
Returns the remote host IP of the stream HANDLE or an empty string.

CONNECTED (HANDLE)
Returns TRUE if the connection has been established for TCP-based stream HANDLE, or FALSE otherwise.

See also example 7.3 on page 27.
3.8.3 Audio (audio devices only)

Audio input/output can be opened with the following syntax:

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

<table>
<thead>
<tr>
<th>MODE value</th>
<th>mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MP3 decoding</td>
</tr>
<tr>
<td>2</td>
<td>MP3 encoding</td>
</tr>
<tr>
<td>3</td>
<td>full-duplex PCM</td>
</tr>
<tr>
<td>4</td>
<td>full-duplex μ-Law</td>
</tr>
<tr>
<td>5</td>
<td>full-duplex A-Law</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FLAGS bit</th>
<th>value/meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0: read/write raw data</td>
</tr>
<tr>
<td></td>
<td>1: read/write MP3 RTP frames</td>
</tr>
<tr>
<td>1</td>
<td>0: start playback immediately</td>
</tr>
<tr>
<td></td>
<td>1: delayed playback</td>
</tr>
<tr>
<td>2</td>
<td>0: delay in milliseconds</td>
</tr>
<tr>
<td></td>
<td>1: delay in bytes</td>
</tr>
</tbody>
</table>

QUALITY:

Is ignored for MP3 decoding.
For full-duplex modes (modes 3, 4 and 5) specifies sampling rate in kHz: 8 or 24
Meaning for MP3 encoding is documented in the following table:

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

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

Syntax:

```
READ H, BUFFER, -INDEX
```
The `BUFFER` parameter is ignored in this case.

### Table 1: Reading the audio device status

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

Audio device parameters can be set using the `WRITE` command with the following syntax:

```
WRITE H VALUE PARAM_INDEX
```

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

<table>
<thead>
<tr>
<th>PARAM_INDEX</th>
<th>PARAMETER</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mic gain</td>
<td>0..15 (in 1.5dB steps, starting at 21dB)</td>
</tr>
<tr>
<td>2</td>
<td>AD gain</td>
<td>0..15 (in 1.5dB steps, starting at -3dB)</td>
</tr>
<tr>
<td>3</td>
<td>input source</td>
<td>1 line in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 microphone</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 autodetect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 SPDIF optical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 SPDIF coaxial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 mix line in with playback</td>
</tr>
<tr>
<td>4</td>
<td>input mode</td>
<td>0 mono</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 stereo</td>
</tr>
<tr>
<td>5</td>
<td>output mode</td>
<td>0 stereo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 mono</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 bridge</td>
</tr>
<tr>
<td>6</td>
<td>output volume gain</td>
<td>in dB</td>
</tr>
<tr>
<td>7</td>
<td>loudness</td>
<td>0..20</td>
</tr>
<tr>
<td>8</td>
<td>balance</td>
<td>-10..10 (-10=left, 10=right)</td>
</tr>
<tr>
<td>9</td>
<td>treble</td>
<td>-10..10</td>
</tr>
<tr>
<td>10</td>
<td>bass</td>
<td>-10..10</td>
</tr>
<tr>
<td>11</td>
<td>volume type</td>
<td>0 5% steps (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 dB</td>
</tr>
<tr>
<td>12</td>
<td>volume</td>
<td>(see the previous row)</td>
</tr>
</tbody>
</table>

See also example 7.1 on page 27.

#### 3.8.4 Serial port

First serial port can be opened using:
OPEN "COM:Baudrate,Parity,Data,Stopbits,FlowControl;PortNumber:" AS 3

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

Possible values for Baudrate are:
230400,115200,76800,38400,19200,9600,4800,2400,1200,600,300

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Type of flow control</th>
</tr>
</thead>
<tbody>
<tr>
<td>NON</td>
<td>none</td>
</tr>
<tr>
<td>SFW</td>
<td>Software flow control (XON,XOFF)</td>
</tr>
<tr>
<td>HDW</td>
<td>Hardware flow control (RTS,CTS)</td>
</tr>
<tr>
<td>422</td>
<td>RS422</td>
</tr>
<tr>
<td>485</td>
<td>RS485</td>
</tr>
</tbody>
</table>

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

3.8.5 SETUP

The EEPROM of the device can be accessed opening stream in the following way:

OPEN "STP:offset:" AS 3

where parameter offset is used to specify an offset in the setup EEPROM.

Only ONE such file can be opened at a time.

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.

The read and write operations use strings for binary operations, so with a read call the complete string will be filled with data from the EEPROM.

The one parameter after the identifier is used to specify an offset in the setup EEPROM (starting from 0).

If the WRITE operator has been used on the file, the new modified Setup will be saved into EEPROM upon CLOSE.

3.8.6 The CIFS filesystem and the local USB filesystem (not supported on Barionet)

Files on a CIFS (MS Windows file sharing protocol) shared filesystem can be opened for reading, writing or append:

OPEN "C_R:\filename" AS H – open file in read mode
OPEN "C_W:\filename" AS H – open file in write mode
OPEN "C_A:\filename" AS H – open file in append mode

The filename parameter is the IP/share of the file to be accessed.

Example:

OPEN "C_R:\192.168.5.1/Share/Directories/Data/File.txt" AS 1

The "C_D:\..." values are used for directory listing. In the directory mode the $S$ is modified by the OPEN call to contain the first directory name after OPEN $S$ AS H.
User ID/Password information can be attached to the file name preceded by a “@” character. If no user id is given, the “GUEST” user id will be used.

Example:

```
OPEN «C_R://192.168.5.1/Share/File.txt@hello;password» AS 1
```

Directory search on the CIFS filesystem is also possible using

```
OPEN «C_D:<filename>» AS H — open directory for listing
```

In the directory mode each `READ` command returns the name of the next directory entry in the `$S$`.

A file named `filename.ext` on the local USB filesystem can be accessed as “C_R:usb://filename.ext”.

Files on the local USB filesystem can also be deleted using the syntax:

```
DELETE «usb://filename.ext»
```

For CIFS, see also program example 7.1 on page 27.

### 3.8.7 The local flash filesystem

The local flash file system can be open using the `OPEN` command:

```
OPEN «F_R:<filename>» AS H — open file in read mode
OPEN «F_W:<filename>» AS H — open file in write mode
OPEN «F_A:<filename>» AS H — open file in append mode
```

Open file on the local flash file system in the device in read (F_R), write (F_W) or append (F_A) mode.

Example:

```
open file “testfile.txt” for reading with handle 1
```

When using `WRITE`, an EOF zero byte terminator will be automatically written with the data.

```
FILEPOS(   )
```

Returns the current file position for the FFS file H.

```
SEEK(   )
```

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

### 3.8.8 The Wiegand reader

It's possible to access a connected Wiegand reader with the following statement:

```
OPEN «RDR:» AS H
```

See example 7.5 on page 29.

### 3.9 Direct hardware access

```
IOCTL(   )
```

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

This function is hardware dependent.

Most I/O variables are bit variables (0/1) or 16 bit variables, however, 32 bit variables may also exist (for example counters). Please refer to the BCL device manual for details.

Code example:
activates the first digital output 1 and then toggles the second relay on the Barionet.

**IOSTATE**(*Eo*)

Returns the state of I/O variable or port *Eo*.

Most I/O variables are bit variables (0/1) or 16 bit variables, however, 32 bit variables may also exist (for example counters). Please refer to the BCL device manual for details.

Code example:

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

stores the state of the digital input 1 into the variable INP1.

### 3.10 Diagnostic functions

**PING**(*E5*, *E*)

Returns the time period (in milliseconds) the device with IP address *E5* needed to respond to a PING (ICMP echo) request, or 0 if no reply has been received within *E* milliseconds (timeout).

Code example:

```
IP5 := "192.168.2.18"
rtme := PING(IP5, 50)
```

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

**TRAP**(*E5*, *N1*, ...)

Send *Ns* as SNMP trap to IP address *E5*.

**SYSLOG**(*E5*, *E*)

Sends the *E* as a UDP message to Syslog (port 514). The optional parameter *E* specifies a debugging level.

Code example:

```
SYSLOG "ALARM"
```
4 Interpreter information

4.1 Execution speed

The execution speed of BCL programs depends on the specific hardware and firmware used, but in general it is very high, typically more than 5000 instructions per second. In other words: each instruction needs 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. During a LOCK period about 11800 commands are executed per second.

4.2 Runtime environment limitations

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

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

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

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

4.3 System variables

Several predefined values are available:

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ARG</em></td>
<td>holds the number of arguments given to the function, see section 3.7 on page 16</td>
</tr>
<tr>
<td><em>CGI</em></td>
<td>used for CGI request handling, see section 5.3 on page 25</td>
</tr>
<tr>
<td><em>DTS</em></td>
<td>time counter, see section 3.4.7 on page 11</td>
</tr>
<tr>
<td><em>ERL</em></td>
<td>line number of the last error, see section 3.4.7 on page 11</td>
</tr>
<tr>
<td><em>ERR</em></td>
<td>error code of the last error, see section 3.4.7 on page 11</td>
</tr>
<tr>
<td><em>TMR</em></td>
<td>array of time counters, see section 3.4.7 on page 11</td>
</tr>
</tbody>
</table>
5 WEB interface

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

### 5.1 DHTML tags

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

\[
\begin{align*}
\&LBAS(1, "\%2\%ld", V) \\
\&LBAS(1, "\%2\%fs", S$)
\end{align*}
\]

where \( V \) and \( S$ \) are integer and string variables. The functions will return "[NO_VAR]" if the variable is not defined.

Please note that for integer variables the format string must be "\%2\%ld", and for string variables the format string must be "\%2\%fs".

### 5.2 Variable setting by CGI

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

```html
<a href="/BAS.cgi\S\&S\&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:

```html
<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>
```

### 5.3 CGI handling in the BCL

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

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

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

The reply to the browser can be sent back using the special handle -1:

```
WRITE(-1,Es).
```

Instead of "closing" the handle, set the \_\_CGI\_\_ variable back to "" (an empty string) in order to finish sending the reply.

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

Preprocessor gives additional possibilities for the BCL code handling. In order to use the preprocessor the subdirectory “BCL” should be created in the main project directory containing the BCL programs with the .bcl extension instead of the .bas extension.

Then the tokenizer should be used:

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

with a start file “file_name.bcl” for your project. The tokenizer will automatically search files in the BCL subdirectory, collect them from separate “.bcl” files of the project according to #include directives, handle macros by #define directives, write the final “source” “file_name.bas” file into the main directory, and convert it to “file_name.tok” file. Thus, the original project files will NOT be included in the COB-file by default.

```
#include file_name.bcl
```

adds 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.

```
#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
```

Preprocessor directives must be terminated with CRLF or a comment. Macro handling is case sensitive and is not recursive.
7 Example programs

7.1 Playing an MP3 file from the USB filesystem

```plaintext
DIM _Ms(1537)
OPEN "AUD:1,0,4000" AS 7
SYSLOG "playback"
OPEN "C: Auswahl\file.mp3" AS 2
READ 2,_Ms
l=LASTLEN(2)
IF l<0 THEN
    SYSLOG "end of file"
    CLOSE 2
    GOTO 11
ENDIF
IF filesize(7)<1 THEN GOTO 102
WRITE 7,_Ms,l
GOTO 101
END
```

7.2 Record audio into an MP3 file on the USB filesystem

Audio line input is encoded as MP3 and recorded for 60 seconds into file.mp3 on the local USB filesystem. For details about setting the audio parameters see section 3.8.3 on page 19. For details about _DTS_ variable see section 3.4.7 on page 11.

```plaintext
DIM _M_r(5000)
OPEN "C: Auswahl\file.mp3" AS 4
OPEN "AUD:2,0,4strs(1024*7*16+1)" AS 7
WRITE 7,"10",-1
WRITE 7,"10",-2
WRITE 7,"1",-3
WRITE 7,"20",-12
READ _M_r ftp_l LASTLEN
>0
IF ftp_l THEN WRITE _M_r ftp_l
(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
```

7.3 Sending an email

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

```plaintext
DELAY 3000
DIM BUFFERS(200)
dim CRs(3)
CRs = CHR$(13)+CHR$(10)
BUFFERS=""
OPEN "TCP:192.168.2.130:25" AS 0
10
IF NOT(CONNECTED(0)) THEN GOTO 10
1 READ 0,BUFFERS,0
IF LEN(BUFFERS)>0 THEN GOTO 1
WRITE 0,"HELO BCLdevice"+CRs,0
2 READ 0,BUFFERS,0
IF LEN(BUFFERS)>0 THEN GOTO 2
```
7.4 Streaming MP3 over RTP

The following program plays an RTP MP3 stream incoming on the UDP port 5555.

```plaintext
DIM _Mss(1537)
Open "UDP:0.0.0.0:5555" as 4
Open "AUD:1,7,0,4000" as 7
Write 7,"6",-2
Write 7,"18",-12

if lastlen(4) < 0 then
    read 4._Mss
    Write 7._Mss,lastlen(4)
    syslog =read:"+str(lastlen(4))"
endif
GOTO 1099
end
```
# 7.5 The Wiegand reader

Wiegand reader access with queuing and socket to listen to

```vba
Dim Coms(24) ' file name for
open function
Dim ss(256), ps(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 one word)
Dim quin, quout ' in and out
pointer to reading queue

COMS = "RDR:" ' open reader
interface
Open COMs as 4
COMS = "TCP:0.0.0.0:10009" ' tcp listening socket
Open COMs as 1
quin+1 quout+1
syslog "Wiegand reader demo 2.1", 2

100    read 4, ss, 0
If lastlen(4) > 0 Then GoSub 1000 ' ID read, get data and
store in queue
If and(connected(1), quin<>quout) Then

ps= strs(qu(quout, 1)) + sprintfs("", %04x, qu(quout, 2)) + sprintfs("", %06x, qu(quout, 3))

Write 1, ps, 0
ps = from qu entry " + strs(quout) + " sent: " + ps
syslog ps, 6
quout+quout+1 If quout>201 Then quout+1 ' next storage
space, wrap to 1
endif
GoTo 100

1000   If ANDmidget(ss, 1, 1), 128 Then rdr=2 Else rdr=1 ' get
reader ID (1 or 2)
rlen=AND(127, midget(ss, 1, 1)) ' get read
length (how many bits)
ps=sprintfs("Wiegand read from %u, "%rdr) + sprintfs("Wiegand read from %u", %02u bits: %r, rlen)
For i=2 to lastlen(4)
ps=ps+sprintfs("%02x", midget(ss, i, 1))
Next i
syslog ps, 5 ' if 26 bits, then
declare to "real" 3 bytes
if rlen=26 Then
b1=and(255, shl(midget(ss, 2, 1), 7)) + shr(midget(ss, 3, 1), 7)
b2=and(255, shl(midget(ss, 3, 1), 7)) + shr(midget(ss, 4, 1), 7)
b3=and(255, shl(midget(ss, 4, 1), 7)) + shr(midget(ss, 5, 1), 7)
v1=0
v2=b1+65536+b2+256+b3 ' store 24bit
wiegand ID
GoTo 1100
endif
If rlen=44 Then
v1=midget(ss, 2, 1) * 256 + midget(ss, 3, 1)
v2=(midget(ss, 4, 1) * 256 + midget(ss, 5, 1)) * 256 + midget(ss, 6, 1)
GoTo 1100
endif
```

WEB interface
Return  

\begin{verbatim}
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
    If i<quout Then quin=i                   ' only store if this does
not overrun queue :
        Return
End
\end{verbatim}

\subsection*{7.6 TCP serial gateway}

\begin{verbatim}
DIM S$(12)
tcps= "TCP:0.0.0.0:10001"                    'TCP waiting for connection to
10001
    OPEN tcp AS 1                              'Open serial terminal
    OPEN "COM:9600.N,8,1,NON;1" AS 2          'Open serial terminal
    write 2, "Wait for connection...",0
101
    IF CONNECTED(1) = 0 THEN GOTO 101         'waits and checks TCP connection
    write 2, "Terminal now CONNECTED with TCP ",
    tcp = "Host: " + RMTHOST$(1) + ", Port: " + STR$(RMTPORT(1)) PRINT tcp
    tcp = tcp + 111
111
    read 2,S$,-1                                 'gets chars from keyboard
    WRITE 1, S$, 0                               'sends chars to TCP
    IF ASC(S$) = 27 THEN GOTO 112                'ESC code to Break
    READ 1, S$,0 write 2,S$,0                   'reads chars from TCP
    IF CONNECTED(1) THEN GOTO 111                'checks TCP connection
112
    write 2, "Terminal finished",0
    close 1                                     '
    close 2                                     '
END
\end{verbatim}
### 8 Appendix B – BIN / DEC / HEX conversion

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

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

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.
Appendix C – Obsolete or unimplemented functions

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

**SEQV ( 1$, 2$ )**
This function has been used for string comparison in boolean expressions. It has been replaced by the equal sign (=) as strings now can be matched directly, see section 3.4.6.3 on page 10.

**SYS_TIME**
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  = (0 ) _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 input buffer, or an empty string if there were no characters on input. Same functionality can be achieved using the standard I/O functions for the serial port.

**INPUT [ { 0$ | } , ] { | $ }**
Original function: Prints $0$ 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 [ [ $ | ] [ , | ; ] ] . . .**
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**
Original function: Timer 0 is a special timer. If set, it can be used as a "software watchdog", and once the count of this timer expires the BCL program will be terminated. To prevent this it has to be reset using the statement: `TIMER 0; ... E`

**Code example:**
```
  TIMER 0,5000
```
resets the watch dog timer for the next 5 seconds.
Alphabetical Index

<table>
<thead>
<tr>
<th>audio</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td>12</td>
</tr>
<tr>
<td>DELAY</td>
<td>17</td>
</tr>
<tr>
<td>DIM</td>
<td>5, 7p</td>
</tr>
<tr>
<td>ELSE</td>
<td>10</td>
</tr>
<tr>
<td>END</td>
<td>9</td>
</tr>
<tr>
<td>ENDIF</td>
<td>10</td>
</tr>
<tr>
<td>FOR</td>
<td>9</td>
</tr>
<tr>
<td>GOSUB</td>
<td>10</td>
</tr>
<tr>
<td>GOTO</td>
<td>9p</td>
</tr>
<tr>
<td>IF 10</td>
<td></td>
</tr>
<tr>
<td>LOCAL</td>
<td>17</td>
</tr>
<tr>
<td>LOCK</td>
<td>13, 26</td>
</tr>
<tr>
<td>NEXT</td>
<td>9</td>
</tr>
<tr>
<td>RETURN</td>
<td>10</td>
</tr>
<tr>
<td>The END</td>
<td></td>
</tr>
<tr>
<td>The LOCK</td>
<td>13</td>
</tr>
<tr>
<td>THEN</td>
<td>10</td>
</tr>
<tr>
<td>Timer</td>
<td>34</td>
</tr>
<tr>
<td>TO?</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>comments</td>
<td>5</td>
</tr>
<tr>
<td>delimiters</td>
<td>5</td>
</tr>
<tr>
<td>escape sequences</td>
<td>8</td>
</tr>
<tr>
<td>events</td>
<td>11, 17</td>
</tr>
<tr>
<td>expression</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>11</td>
</tr>
<tr>
<td>integer</td>
<td>6</td>
</tr>
<tr>
<td>string</td>
<td>8</td>
</tr>
<tr>
<td>function</td>
<td></td>
</tr>
<tr>
<td>ASC</td>
<td>14</td>
</tr>
<tr>
<td>CHR$</td>
<td>14</td>
</tr>
<tr>
<td>CLOSE</td>
<td>17</td>
</tr>
<tr>
<td>CONNECTED</td>
<td>19</td>
</tr>
<tr>
<td>DELETE</td>
<td>22</td>
</tr>
<tr>
<td>EXEC</td>
<td>34</td>
</tr>
<tr>
<td>FILEPOS</td>
<td>23</td>
</tr>
<tr>
<td>FILESIZE</td>
<td>18</td>
</tr>
<tr>
<td>INKEY$</td>
<td>34</td>
</tr>
<tr>
<td>INPUT</td>
<td>34</td>
</tr>
<tr>
<td>INSTR</td>
<td>13</td>
</tr>
<tr>
<td>IOCTL</td>
<td>24</td>
</tr>
<tr>
<td>IOSTATE</td>
<td>24</td>
</tr>
<tr>
<td>ISEQV</td>
<td>34</td>
</tr>
<tr>
<td>LASTLEN</td>
<td>18p</td>
</tr>
<tr>
<td>LCASE$</td>
<td>14</td>
</tr>
<tr>
<td>LEN</td>
<td>13</td>
</tr>
<tr>
<td>MEDIATYPE</td>
<td>18</td>
</tr>
<tr>
<td>MIDS</td>
<td>14</td>
</tr>
<tr>
<td>MIDCPI</td>
<td>16</td>
</tr>
<tr>
<td>MIDGET</td>
<td>16</td>
</tr>
<tr>
<td>MIDSET</td>
<td>16</td>
</tr>
<tr>
<td>OPEN</td>
<td>17</td>
</tr>
<tr>
<td>PING</td>
<td>24</td>
</tr>
<tr>
<td>PRINT</td>
<td>34</td>
</tr>
<tr>
<td>READ</td>
<td>17</td>
</tr>
<tr>
<td>RMTHOST$</td>
<td>19</td>
</tr>
<tr>
<td>RMTPORT</td>
<td>19</td>
</tr>
<tr>
<td>SEEK</td>
<td>23</td>
</tr>
<tr>
<td>SPRINTF$</td>
<td>14</td>
</tr>
<tr>
<td>SYM</td>
<td>16</td>
</tr>
<tr>
<td>SYSLOG</td>
<td>24</td>
</tr>
<tr>
<td>SYSTIME</td>
<td>34</td>
</tr>
<tr>
<td>TRAP</td>
<td>24</td>
</tr>
<tr>
<td>UCASES$</td>
<td>14</td>
</tr>
<tr>
<td>VAL</td>
<td>14</td>
</tr>
<tr>
<td>WRITE</td>
<td>17, 19</td>
</tr>
<tr>
<td>18p</td>
<td></td>
</tr>
<tr>
<td>GOTO</td>
<td>10</td>
</tr>
<tr>
<td>labels</td>
<td>9</td>
</tr>
<tr>
<td>operation</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>7, 11</td>
</tr>
<tr>
<td>assign</td>
<td>6, 8</td>
</tr>
<tr>
<td>integer</td>
<td>6</td>
</tr>
<tr>
<td>NOT</td>
<td>7, 11</td>
</tr>
<tr>
<td>OR</td>
<td>11</td>
</tr>
<tr>
<td>OR7</td>
<td>7</td>
</tr>
<tr>
<td>SHR</td>
<td>7</td>
</tr>
<tr>
<td>string</td>
<td>9</td>
</tr>
<tr>
<td>XOR</td>
<td>7, 11</td>
</tr>
<tr>
<td>= 6, 8</td>
<td></td>
</tr>
<tr>
<td>protocol</td>
<td></td>
</tr>
<tr>
<td>The CIFS</td>
<td>22</td>
</tr>
<tr>
<td>The TCP</td>
<td>19</td>
</tr>
<tr>
<td>The UDP</td>
<td>19</td>
</tr>
<tr>
<td>The Wiegand</td>
<td>23</td>
</tr>
<tr>
<td>strings</td>
<td>7</td>
</tr>
<tr>
<td>subroutines</td>
<td>10</td>
</tr>
<tr>
<td>tokenizer</td>
<td>12</td>
</tr>
<tr>
<td>variable</td>
<td></td>
</tr>
<tr>
<td>ARG</td>
<td>17</td>
</tr>
<tr>
<td>array of integers</td>
<td>6</td>
</tr>
<tr>
<td>DTS</td>
<td>12</td>
</tr>
<tr>
<td>integer</td>
<td>6</td>
</tr>
<tr>
<td>string</td>
<td>8</td>
</tr>
<tr>
<td>_ARG</td>
<td>16</td>
</tr>
<tr>
<td>_ARG</td>
<td>17</td>
</tr>
<tr>
<td>_CGL$</td>
<td>27</td>
</tr>
<tr>
<td>_CMD$</td>
<td>34</td>
</tr>
<tr>
<td>_ERR</td>
<td>12</td>
</tr>
<tr>
<td>_M9, 14</td>
<td>12</td>
</tr>
<tr>
<td>_TMR</td>
<td>12</td>
</tr>
</tbody>
</table>

WEB interface  33