BARIX
THE VOICE OF SIMPLICITY

ABCL FirmwareMy First Day With BCL

Basic guide for beginning BCL programming

Programming Tutorial

Version V1.02

Released 03. Feb. 2011

Supports:

- Barionet 50, 100 (LX & EX XPort)
- Annuncicom 100, 155, 200,1000
- Exstreamer 100, 110, 120, 500,1000
- Instreamer 100
- IPAM 100, 200, 300

Table of Contents

1 Introduction

The Barix Control Language (BCL) is a simple interpreted high-level language to customize the behavior of certain Barix devices. This small tutorial serves as an introduction to BCL programming, by describing the development of a simple custom BCL application, the related web interface, and the final installation of the application in the device.

For this purpose, the tutorial will guide through the programming of a simple BCL script that collects some messages from a microphone connected to a Barix Annuncicom 100, recording them to a USB stick.

1.1 Requirements

- The ABCL Kit, can be downloaded from http://www.barix.com
- Barix Annuncicom 100

1.2 Other Useful References

Some other useful technical reference documents to keep on hand are:

- ABCL Technical Documentation
- BCL Programmers Manual

They can be downloaded from http://www.barix.com.

2 Environment Setup

As a first step, the ABCL Kit must be downloaded and decompressed inside a preferred directory. The kit consists of a set of tools and scripts running on both Windows and Linux platforms.

However, to work on Linux platform, the external programs“dosemu”, “atftp” and “wine” are required.

2.1 ABCL Firmware

ABCL Firmware is the device platform that allows BCL programming on the Barix Audio products family.

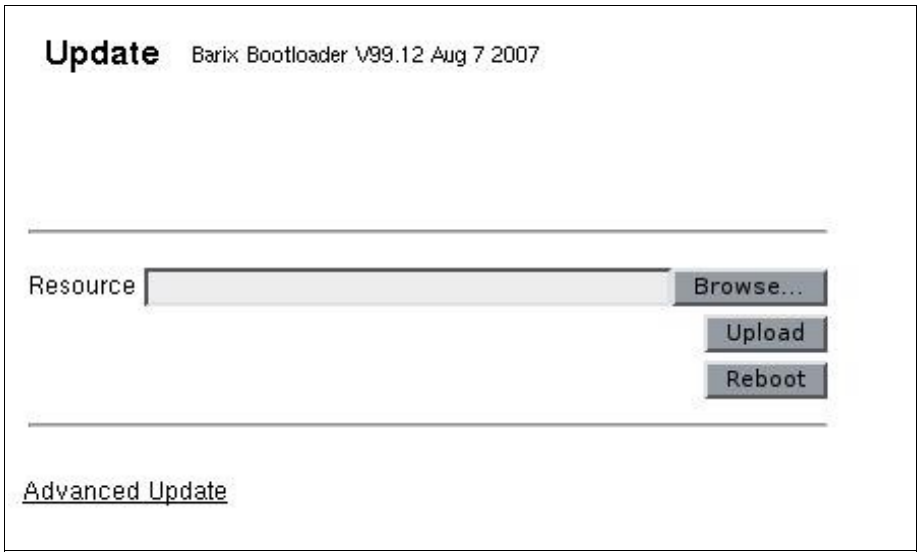
Barix Audio devices such as the “IP Audio Module”, “Exstreamer 100/200”, ”Instreamer 100” and “Annuncicom 100” are shipped with a standard firmware which has no BCL support.

The next section explains the procedure to replace the standard firmware with the ABCL Firmware.

2.2 ABCL Firmware upload

From the device home page panel:

1. Click on “Configuration” button
2. Click on “Update” button. An update page will appear
3. Click on the link “Please click here to continue”, wait until the countdown end, then the following update page will appear:



4. Click on “Browse...” button and select the binary file “compound.bin” from the ABCL Kit “update_rescue” subdirectory.
5. Once selected, click on "Upload".

This process can take a few minutes. After a successful upload click on the "update" link and when the update window reappears click the "Reboot" button.

The device will reboot with the new ABCL firmware.

3 The Web User Interface

The ABCL Firmware, after a successful installation into the device, offers a set of web pages for the remote user control and configuration.

3.1 ABCL Firmware home page

The main home page displays some setting sections to configure the device. The first section, called “APPLICATION” allows a BCL application to be selected and run pressing the “Apply” button at the bottom of the page.

Inside the “Application” drop-down list, many application entries are visible. Every entry of the list shows a long descriptive name and a short program identification name between parenthesis. There is a placeholder usable for user-modifiable applications called “custom1”.
If we want that our sample application could be selectable from this list by a descriptive name of our choice, we need to modify this html page.

If we choose for example “My First BCL Application” and “mfbclapp” respectively as the descriptive name and the short program id name, the following modifications are needed:

Open the file “uisettings.html” on the PC with your preferred editor. You can find it in the relative path “webuidevkit/abclapp”, starting from the root of the ABCL Kit directory. Doing a backup copy of this file can be useful to restore the original home page.

Now let's modify the html line

```
<option value="custom1" &Lsetup(3,"%s",180,S,"custom1",
            "selected");>Custom Application 1 (custom1)</option>
```

with the line

```
<option value="mfbclapp" &Lsetup(3,"%s",180,S,"mfbclapp",
            "selected");>My First BCL Application (mfbclapp)</option>
```

So, now we can save and close the file since our web home page is ready to be uploaded. We have to remember that our custom application BCL program is not yet developed and uploaded, so after the new web interface has been uploaded, we still cannot select it for running.

3.2 Updating the Web

Next step is to replace the home page on the device with our modified one. To do this, we need to prepare a package with all the original ABCL Web content but containing our modified home page.
For this purpose open the ABCL Kit root directory and, moving to the “webuidevkit” subdirectory, execute the tool “abclapp.bat” on Windows or “abclapp.sh” on Linux.

A file named “abclapp.cob” will be generated. To upload it into the device some steps are required:

1. Select “Update” button. An update info page will appear.
2. Click on “Please click here to continue”, and wait for the countdown end.
3. Click on “Advanced Update”, a more detailed update page will appear
4. Enter WEB3 in the Target field, click on “Browse...” and select “abclapp.cob” from the “webuidevkit” subdirectory.
5. Once selected, click on "Upload" and wait for the “successfully loaded” message,

now click on “update” and then “Reboot”.

1. Reset the device
2.

Now this “My First BCL Application” choice is selectable, next step is to prepare the Web part of our custom application.

3.3 Custom application Web interface

In the “bcldevkit” subdirectory of the ABCL Kit some demo BCL applications are available.
For every application there is a subdirectory with the respective name.

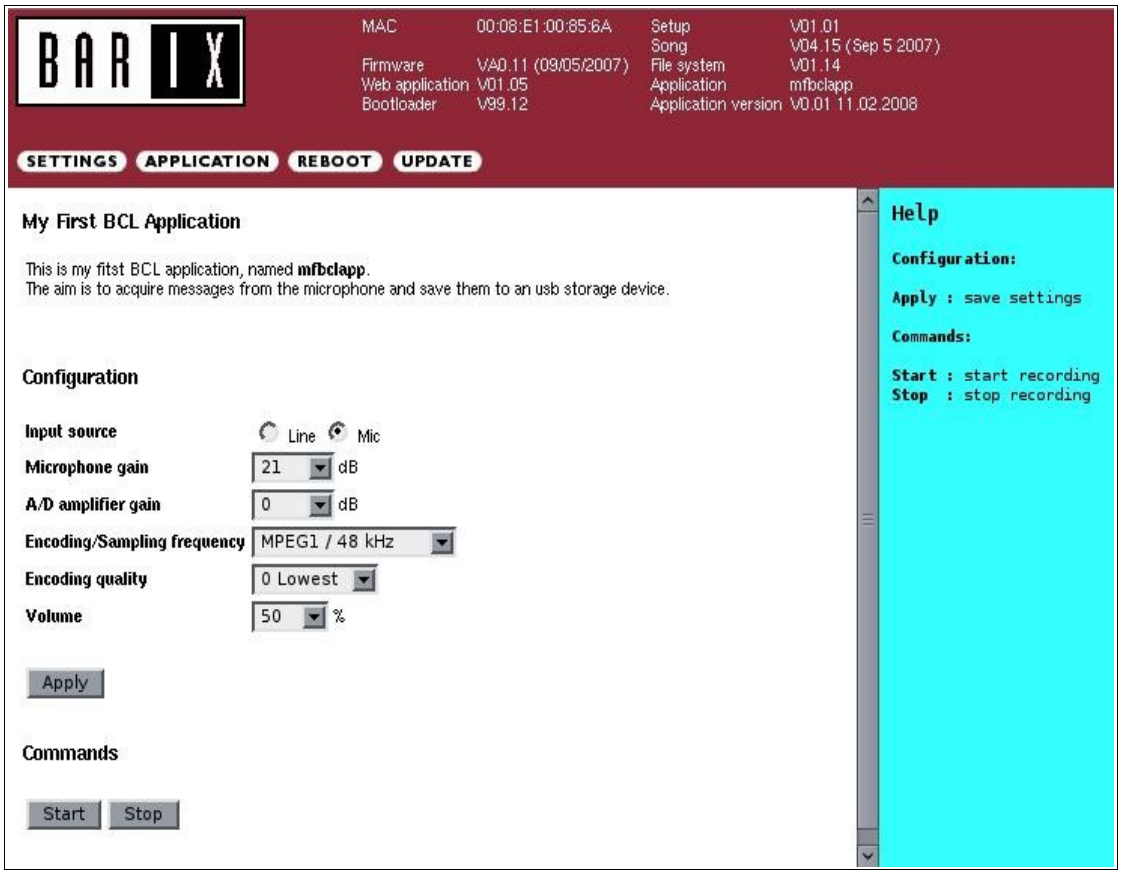
The directory named “custom1” is a placeholder for a user-modifiable application. We can start copying this folder and renaming it with our custom application name “mfbclapp”.

Inside this directory there is a custom application BCL skeleton script (custom1.bas) and three html
files composing the custom application web interface.
The file “custom1.html” will be the custom application main page, it contains the reference to other configuration and help html frames. They are contained respectively in custom1_config.html and custom1_help.html files.

We need to rename the files, respecting the naming conventions defined in the ABCL Firmware manual, as indicated below:

custom1.bas	to	mfbclapp.bas
custom1.html	to	mfbclapp.html
custom1_config.html	to	mfbclapp_config.html
custom1_help.html	to	mfbclapp_help.html

Now we can modify these files to fit our application. After some standard html work, this is how the custom application web page will appear:



There are some selectable setup options for the input volume, some commands to start stop recording and an “Update” command to apply configuration changes.

An important thing to focus on is the way the drop-down lists and buttons are connected with the BCL application. The link is done through some special fields called “dynamic marks”. Their types and syntax details are explained in section 4.2 of the ABCL technical documentation manual.

To use “dynamic marks”, web pages or frames must include, in the first 500 bytes and before any other dynamic mark, the special initial dynamic mark

&L(0,"*");

As shown in the html code below, dark gray “&Lsetup(...); ” dynamic marks allow the retrieval and display of the application setup variables values.
Light gray are necessary parts to submit selected configuration values, saving the changes in the setup memory.

```
<form action=setup.cgi method=get><input type=hidden name=L value=rebooting.html>
<table>
<tr>
  <td><font size=1><b>Input source</td>
  <td><font size=1>
    <input type=radio name=B810 value=129 &LSetup(3,"%s",810,B,129,"checked");> Line
    <input type=radio name=B810 value=130 &LSetup(3,"%s",810,B,130,"checked");> Mic
  </td>
```

```
</tr>
<tr>
  <td><font size=1><b>Encoding/Sampling frequency</td>
  <td><font size=1>
    <select size=1 name=B811>
      <option value=3 &LSetup(3,"%s",811,B,3,"selected");>MPEG1 / 48 kHz</option>
      <option value=1 &LSetup(3,"%s",811,B,1,"selected");>MPEG1 / 44.1 kHz</option>
      <option value=5 &LSetup(3,"%s",811,B,5,"selected");>MPEG1 / 32 kHz</option>
      <option value=2 &LSetup(3,"%s",811,B,2,"selected");>MPEG2 / 24 kHz</option>
      <option value=0 &LSetup(3,"%s",811,B,0,"selected");>MPEG2 / 22.05 kHz</option>
      <option value=4 &LSetup(3,"%s",811,B,4,"selected");>MPEG2 / 16 kHz</option>
    </select>
  </td>
</tr>
<tr>
  <td><br><input type=submit value=" Apply "></td>
  <td> </td>
</tr>
</table>
</form>
```

Looking for example at the html “select” tag name: the “B811” mean that the selected value will be stored/read at address 811 of the setup memory, where “B” means that we are storing a single byte integer.

About the dynamic mark

```
&LSetup(3,"%s",811,B,1,"selected");
```

The meaning of every single parameter is explained in details in the ABCL technical documentation manual.
In this particular case, this syntax means “if the variable stored at setup memory address 811 is equal to the value “1”, print out a “selected” string”. Regarding the html “select” tag, the produced output
will set the active selection to reflect the value stored in the setup memory.

Application setup memory addresses must be selected starting from offset 500.
To avoid the overwriting of other application settings, these addresses need to be chosen according to the “eeprom setup record” document in the subfolder “bcldevkit” of the ABCL Kit.

Actually, our custom applications can use free addresses starting from offset 800.
To transfer a command to our running BCL application, the CGI web script support is used.
The following html code shows how this is done:

```
<form action="BAS.cgi" method="GET" target="empty">
  <input type="hidden" name="cmd$" value="start">
  <input type="submit" value="Start">
</form>
```

When the “Start” button is clicked, the tag

```
<input type=”hidden” name=”cmd$” value=”start”>
```

will allow the transfer of the string value “start” inside the BCL application to string variable “cmd$”.

4 The BCL code

Finally we can proceed to write down our BCL code.

4.1 The complete application code

Before looking at each section of the program in detail, this is how the final BCL application program appears:

```
'-------------------------------------------------------------------------------------------
' "mfbclapp.bas", My First BCL Application
'-------------------------------------------------------------------------------------------
' V00.01  11.02.08    - initial version
'-------------------------------------------------------------------------------------------

        ' strings declaration
        dim     ver$    (18)
        dim     m_in$   (5000)
        dim     cmd$    (64)
        dim     app$, set$

        ' integer variables declaration
        dim     msgn,inlen
        dim     vol,inp,enc,qlt,mcg,adg

        ' variables initialization
        app$    = "My First BCL Application"
        ver$    = "V0.01 11.02.2008"

        ' opening serial port for debug tracing
        open "COM:9600,N,8,1,NON:1" as 2

        ' start message on serial port
        write 2, "\r\n" + app$ + " started\r\n"

        ' debug message, using syslog protocol
        syslog app$ + " started"

        ' load default configuration from setup memory
        gosub 300

100     ' "start" command clicked from web ?
        if cmd$ = "start" then
                ' cleaning cmd variable for next check
                cmd$ = ""
                ' debug trace
                syslog "start recording..."
                ' incrementing msg number, for file name
                msgn = msgn + 1
                ' opening usb as output
                open "C_W:usb:///msg_" + str$(msgn) + ".mp3" as 4
                ' mass storage is inserted ?
                if mediatype(4) = 0 then goto 100
                ' opening audio as input
```

```
                gosub 200
110         ' reading the message
                read 7, m_in$, 4096 : inlen = lastlen(7)
                ' if something is read, add the chunk to the USB file
                if inlen > 0 then write 4, m_in$, inlen
                ' "stop" command clicked from web ?
                if cmd$ = "stop" then
                        ' cleaning cmd variable for next check
                        cmd$ = ""
                        ' debug trace
                        syslog   "stop recording..."
                        ' closing both audio input and usb output
close 7
                        close 4
                        ' exit read loop
                        goto     100
                endif
                ' continued read
                goto 110
        endif

        ' back to command check loop
        goto 100

'-------------------------------------------------------------------------------------------
' subroutines
'-------------------------------------------------------------------------------------------


200     ' opening audio input

        open "AUD:2,0,"+str$(1024+qlt*16+enc) as 7          ' mp3 encoding
        write 7,str$(mcg),-1                                ' set MIC Gain
        write 7,str$(adg),-2                                ' set A/D Gain
        write 7,str$(inp),-3                                ' set input source (1-line,2-mic)
        write 7,str$(vol),-12                               ' set Volume (*5%)
        return

300     ' getting setup memory values

        open "STP:800" as 3 : read 3, set$ : close 3
        vol   = midget(set$,1,1)                            ' get volume
        adg   = midget(set$,2,1)                            ' get AD gain
        mcg   = midget(set$,3,1)                            ' get mic gain
        inp   = midget(set$,11,1)-128                       ' get input (129=line, 130=mic)
        qlt   = midget(set$,13,1)                           ' get quality
        enc   = midget(set$,12,1)                           ' get encoding
        if enc > 5 then enc = 3                             ' default encoding is
MPEG1/48kHz
        return

        end
```

First part is about dimensioning, declaration and initialization of variables, then follows a main loop, a nested read/write loop, and finally there are two subroutines.

4.2 Comments

BCL commenting is done putting an apostrophe character (') before the comment line. In this simple application comments are used frequently remarking a detailed explicative purpose. Comments can be added everywhere, in a new line, or to comment some code appending them in the same line.
Every word after the apostrophe character is intended as a comment for that line.

4.3 Variable declaration and dimensioning

Starting a BCL program, a good practice is to declare at program begin the variables that we are going to use later. The declaration is done through the “DIM” command. For string variables, a “$” at the end of the variable name must be appended.

The default space in memory reserved for a string is 256 bytes. We can do a different dimensioning of a string variable following the name with the desired length value inside parenthesis.

It is still possible to define and initialize variables elsewhere in the code, even without a declaration at the program begin.

```
' strings declaration
dim     ver$     (18)
dim     m_in$   (5000)
dim     cmd$    (64)
dim     app$, set$

' integer variables declaration
dim     msgn,inlen
dim     vol,inp,enc,qlt,mcg,adg
```

In these lines we are declaring:

- “ver$”, a version string with a length of 18 bytes
- “m_in$”, a buffer of 5000 bytes for the microphone audio input data
- “cmd$”, a string of 64 bytes to capture web interface commands
- “app$”, a string for the application title
- “set$”, a string to read setup variable values from memory

4.4 Variables initialization

Just after the variables declaration, follows some code to initialize some of the declared variables:

```
'variables initialization
app$        = "My First BCL Application"
ver$        = "V001 11.02.2008"
```

- “app$” is initialized to the application name string, useful later
- “ver$” is initialized to a version string useful for the application history

We don't have to worry about setting integers to 0, all integer variables are guaranteed to be "0" after start-up.

4.5 I/O streams

The sample program sends a start up message over the serial port.

```
' opening serial port for debug tracing
open "COM:9600,N,8,1,NON:1" as 2
```

The “open” function opens an I/O stream channel.

The string

“COM:9600,N,8,1,NON:1”

means that we are opening a serial port (COM) at a speed of 9600 baud, none (N) as parity, 8 data bits, 1 stop bit, none (NON) as flow control, 1 as the port number we want to open.
The port number is usually 1 or 2, this number is related to the specific product documentation.

To read and write to the stream a numeric identification is needed: the “as 2” means that the stream
will have a “handle” value of 2 to be used for consecutive read/write accesses and to close the stream when all operations with it are terminated.

```
' debug start message
write     2, “\r\n” + app$ + " started\r\n"
```

With “write” we send a first startup message through the serial port, to inform that the application is running.

Just after the “write” follows the “2” handle value. It indicates we are writing to the serial port stream that we have previously opened with the handle value 2. If we need to send a specific number of characters, we can specify a third integer parameter with the string length. For zero-terminated strings as above, no other parameters are needed.

4.6 Syslog

The “syslog” command is the main BCL tool for debug purposes. It sends text debug messages through the network using the “Syslog” protocol specifications.

```
' debug message, using syslog protocol
        syslog app$ + " started"
```

A Syslog server PC-application is necessary to receive these messages saving them on a log file.

There is no need to specify a server IP address, the messages are delivered in broadcast fashion, so the server application can detect and accept them.

On Unix, “syslogd” is a common daemon that can be launched with the “-r” option to receive syslog messages, saved then in “/var/log/user.log”.
On Windows, there are some freeware applications as “Kiwi Syslog Daemon”, or “3COM Free Syslog Daemon” and some other.

4.7 Subroutines

BCL code is executed as a sequence of lines, but we could need to execute a block of code for more than one time. Or, as in this sample, to organize the code in some functions, as one to read the setup variables from non-volatile memory and another one to open the audio input stream.

```
' load default configuration from the setup memory
gosub 300
```

The statement “gosub” directs the program execution to a particular label. Here, “gosub 300” calls a subroutine that loads configuration variables from the setup memory. “300” is a numeric label used to indicate the point of the program where the routine begins.

4.8 The setup memory

So after the “gosub” statement the program execution jumps to label “300”.

```
300     ' getting setup memory values

        open "STP:800" as 3 : read 3, set$ : close 3
        vol   = midget(set$,1,1)                    ' get volume
        adg   = midget(set$,2,1)                    ' get AD gain
        mcg   = midget(set$,3,1)                    ' get mic gain
        inp   = midget(set$,11,1)-128               ' get input (129=line, 130=mic)
        qlt   = midget(set$,13,1)                   ' get quality
        enc   = midget(set$,12,1)                           ' get encoding
        if enc > 5 then enc = 3                     ' default encoding is
MPEG1/48kHz
        return
```

The routine above opens a particular stream called “STP”. This is the name of the stream to access the
non-volatile setup memory. The value 800 is the memory offset, starting from 0, from which we are going to read.

The BCL interpreter allows to read the setup memory in blocks of 256 bytes starting from the given offset.

Once the stream is open we read from it through the handle 3.

The “read” function will fill the “set$” string variable with a block of 256 bytes.

After the read operation the stream is closed, stream handle is now free to be used again.

The “set$” variable is now filled with 256 bytes of binary values. Values from the binary array can be retrieved by the “midget” command.
It needs three parameters, the first is the binary array where the values to retrieve are contained, the second is the position we want retrieve starting from 1, the third, set to 1, indicate that we want to retrieve a single byte integer type of value.

There is then a “return” statement where the subroutine ends. It means that execution must jump back to the line that come just after the “gosub 300” line.

4.9 The main application loop

Now follow the part of the code that does the job of reading from the microphone, recording audio messages to the USB memory stick.

```
100
        ' "start" command clicked from web ?
        if cmd$ = "start" then
                ' cleaning cmd variable for next check
                cmd$ = ""
                ' debug trace
                syslog "start recording..."
                ' incrementing msg number, for file name
                msgn = msgn + 1
                ' opening usb as output
                open "C_W:usb:///msg_" + str$(msgn) + ".mp3" as 4
                ' mass storage is inserted ?
                if mediatype(4) = 0 then
                        ' informing we had some trouble and restart
                        write     2,"usb file open failed\r\n", 0
                        goto 100
                endif
                ' opening audio as input
                gosub 200

110
                ' reading the message
                read 7, m_in$, 4096 : inlen = lastlen(7)
                ' if something is read, add the chunk to the USB file
                if inlen > 0 then write 4, m_in$, inlen
                ' "stop" command clicked from web ?
                if cmd$ = "stop" then
                        ' cleaning cmd variable for next check
                        cmd$ = ""
                        ' debug trace
                        syslog   "stop recording..."
                        ' closing both audio input and usb output
                        close 7
                        close 4
                        ' exit read loop
                        goto      100
                endif
                ' continued read
                goto 110
        endif

        ' back to command check loop
        goto 100
```

The program runs in a loop, starting from the “100” label, and jumping back to this label every time a “goto 100” line is encountered. In this continuous cycle, commands sent from the application web page can be checked and some action can be taken for everyone of them.

First of all, by the “if” statement we check the “$cmd” string variable content. It is filled directly from the web through the CGI support. If the variable is empty or different form “start”, the condition is

not met, so the program execution will move to the related “endif” statement, to jump back to label 100 for another check by the “goto 100”.

```
' "start" command clicked from web ?
                if cmd$ = "start" then
                        ' cleaning cmd variable for next check
                        cmd$ = ""
```

If the condition is met, it means that the “Start” button has been clicked in the application web page, so we then need to execute a sequence of operations.
The first operation to be done is to clean the cmd$ variable. This is necessary to avoid the detection of a new false “start” command.

Now some other lines follow:

```
'serial dbg trace
write       2,"start reconrding...\r\n", 0
'incrementing msg number, for file name
msgn = msgn + 1
```

The debug string “start recording ...“ is sent to the serial port. To create the mp3 file name, the “msgn” integer variable is incremented.

```
'opening usb as output
open "C_W:usb:///msg_" + str$(msgn) + ".mp3" as 4
'mass storage is inserted ?
if mediatype(4) = 0 then
        'informing we had some trouble and restart
        write       2,"usb file open failed\r\n", 0
        goto 100
endif
```

The string

“C_W:usb:///”

means that we are creating a file for write operations (C_W) on the USB memory stick.

The first file name saved will be “msg_1.mp3” where the value 1 is “msgn” value converted to string by the “str$()” function.

The function mediatype(4) checks if the file is really open. If the return value is 0 the “open” command has failed, so execution jumps back to the start of the loop by the “goto 100” statement.

```
'opening audio as input
gosub 200
```

With “gosub 200”, a subroutine that open the audio input stream is called.
The subroutine will perform the opening operation:

```
200     'opening audio input

        open "AUD:2,0,"+str$(1024+qlt*16+enc) as 7          'mp3 encoding
        write 7,str$(mcg),-1                                'set MIC Gain
        write 7,str$(adg),-2                                'set A/D Gain
```

```
write 7,str$(inp),-3                    'set input source (1-line,2-microphone)
write 7,str$(vol),-12                   'set Volume (*5%)
return
```

The string
“AUD:2,0”

means “audio interface”, the value 2 is the mode which we are opening it, and means “mp3 encoding”.

The value 0 are the flags for the stream and mean read/write.
The last value is the quality, and will be the string representation of the value obtained by the equation “1024+qlt*16+enc”, where “qlt” and “enc” variable contents are read from the setup memory at the program startup. More details on audio configuration values are documented in the BCL Programmers manual.

After we have opened the stream, we want to set some audio device parameter. This is done by using “write” with a negative value as the last parameter. Every negative value determines the parameter to be set: -1 is the microphone gain, -2 the A/D gain, -3 the input source and -12 the volume.
We set this parameters with our application configuration values, read from the setup memory at application startup.

4.10 The read/write nested loop

From label 110, a read/write nested cycle start.

```
110
    'reading the message
    read 7, m_in$, 4096 : inlen = lastlen(7)
    'if something is read, add the chunk to the USB file
    if inlen > 0 then write 4, m_in$, inlen
```

Reading from stream handle 7 we are trying to get some audio information from the microphone, putting received bytes in the m_in$ array.
By the function “lastlen” we can check if we have read some data. If something has been read, the “inlen” variable will contain a value greater than 0. At this point everything read is written into the USB file stream, the one opened before with the handle 4. We write all the bytes we read “as they are” into the mp3 file.

In the nested loop is also checked if the user need to stop the recording process:

```
    '"stop" command clicked from web ?
    if cmd$ = "stop" then
        'cleaning cmd variable for next check
        cmd$ = ""
        'serial dbg trace
        write       2,"stop reconrding...\r\n",0
        'closing both audio input and usb output
        close 7
        close 4
        'exit read loop
        goto        100
    endif
    'continued read
```

```
goto 110
```

If the “Stop” command is detected, a “stop recording...” message is sent through the serial port, then both the input audio and the mp3 streams are closed. Statement “goto 100” jumps to the beginning of the main loop, to wait again for another “Start” command from the web.

If no “Stop” command has been clicked, the recording process continues, the statement “goto 110” jumps back to label 110, to read again some other data chunk from the audio input.

5 Final steps

Now is time to create a final package, ready to be uploaded into the device.

5.1 Tokenizer, cob files and application upload

The last part to have our application working on the Barix Annuncicom device is to convert the code we have written into a format that the BCL interpreter on the device can understand and execute.

For this purpose, the tool supplied called “tokenizer.exe” will convert the .bas source file into a .tok executable file. On the newer BCL/tokenizer version 1.5 (and newer) also a target parameter (“audio” , barionet50 or barionet) must be defined, e.g.

tokenizer audio mfbclapp.bas

To be uploaded on the device, the generated .tok file must be packed together with other files, as the custom application web interface html files.

Barix supply a single script tool, “bcl.bat” on Windows or “bcl.sh” on Linux to perform together the .tok generation, the .cob packaging and the final upload on the device.

The procedure is the following

1. Select “Update” button. An update page will appear.
2. Click on “Please click here to continue”, and wait for the countdown end.
3. From a command prompt, move to the “bcldevkit” directory.
4. Type “bcl.bat mfbclapp DEVICE_IP”.
5. Click on “Reboot”.

5.2 Running the custom application program

After the device has restarted, in the main web page we can select “My First BCL Application”. Clicking apply we select our application to be executed. Wait for the countdown, click on the APPLICATION button and our custom application is ready to be used.

6 Legal Information

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