BARIX IPAM-400 OEM
SW SDK Documentation

SW Development Kit User Manual v1.10

Related IPAM-400 SW Development Kit Release: v1.05

Document Revision Table

Date	Version	Who	Change
12th March 2018	1.00	ASI	First version
18th April 2018	1.01	ASI	Added Yocto layer configuration and usage information
20th April 2018	1.02	ASI	Cleaned-up chapters 5 and 7
23rd April 2018	1.03	ASI	Minor fixes and removing irrelevant information
15th May 2018	1.04	ASI	Added u-boot-tools and chrpath to the list of needed Linux packages Added instruction to remount the shadow partition as RW
28th May 2018	1.05	ASI	Added info about using OPKG Fixed bits of incomplete information and some typos Removed the command creating the rescue image

11th Jul 2018	1.06	ASI	Updated the OEM devkit version to 1.02 Fixed some typos Updated section 7 Added info about burning the SD card outside the Yocto environment
7th Sep 2018	1.07	ASI	Added section for manual FW image update usingthe qiba-update-client in rescue mode. Fixed some types, and added couple of references.
25th Mar 2019	1.08	ASI	Updated for v1.05. New SIP Client cd features since 1.02: • Support for OPUS codec • "Auth user configuration field on the webUI • Added "autoanswer" configuration option • Support for MA400 HW type
17th Dec 2020	1.09	ASI	Added "Terms of Use" section
17th June 2021	1.10	MBA	Adapted to login changes in Bitbucket, changed username.

TABLE OF CONTENTS

1 About Barix SW Development Kit

The Barix SW Development kit allows Barix OEM customers to develop their own applications for the Barix IPAM-400 platform. The Barix OEM SDK kit is delivered as a bunch of BitBucket repositories, containing the necessary Yocto layers to be added to the standard Yocto environment. Only authorized Barix OEM users have access to these repos. If you are interested to develop your products based on the Barix IPAM-400 platform, please contact Barix Customer support.

The Barix OEM SDK kit uses Yocto Linux as a development environment and is created in a way to keep the customized scripts and packages in additional layers, which reduces the need to change the Yocto Linux system files to the minimum. For this purpose it includes scripts and configuration files to integrate it in the Yocto environment, and as well as Barix specific packages and configuration scripts that do create the Barix specific ecosystem on the target device.

The current release of the SW Development kit is optimized to run on a Barix Annuncicom 60 motherboard with IPAM-400 module installed, but could be easily adapted to any OEM board design.

For better understanding this manual, a good knowledge of Yocto Linux and the Kconfig framework is required, so please refer to the **Yocto Reference Manual** *first before reading the rest of this document.*

Barix recommends using Ubuntu Linux 16.04 for best development results.

2 Terms of Use

1.1 Purpose of the development environment

The only use allowed for this development environment is for developing firmware for the Barix IPAM400 module.

1.2 Licensing

1. *You are responsible for assuring that you have licenses and permission to use the firmware components included as well as added by you for your products. (For example the license to use AAC+ support or other components that require licensing)*
2. *It is in your responsibility to evaluate what functionality that you implement requires licensing und to assure you have permission to use it.*
3. *The IPAM400 Evaluation kit is licensed for AAC+ (simple player). IPAM400 modules bought from Barix or its partners are NOT licensed for any firmware component such as AAC+.*
4. *Before installing your firmware on Barix IPAM400 modules, you need to assure that you have registered and paid for all the required licenses. (e.g. should you use simple player you have to register with www.via-corp.com and pay the required licenses. This extends to all components used in your development that requires licensing.*

1.3 Limitation of Liability

1. *Neither Party shall be liable to the other Party for any special, direct, indirect or consequential loss or damages, including lost profits or costs of procurement of substitute goods.*
2. *In any case, the loss or damage is limited to the Purchase Price of the respective failing Device.*
3. *This limitation of liability shall not apply if the respective loss or damage is caused by fraud or intent.*

1.4 Barix Terms & Conditions

This software also applies under the general Barix Terms & Conditions at: https://www.barix.com/i/terms-conditions/

You acknowledge that it will not acquire any intellectual property rights under this Agreement in the Devices, Firmware, Software or other products or associated materials of the other Party, and that all rights herein are strictly reserved.

Barix AG, Zürich

25.4.2020

3 Configuring the Yocto Linux Development Environment

Contact Barix Support to receive a Virtual Development environment that is ready to go.

This chapter explains how to install Yocto Linux on UBUNTU 16.04 and lists all the required packages that need to be installed in order the installation to be successful.
Barix supports Ubuntu, and the example commands below are tested on Ubuntu 16.04. While the Development kit may run on any Linux distribution, they are not tested by Barix and you may need to find the way to install the required packages on your own.
A Virtual Box Appliance with everything already installed and preconfigured is also under development. Please contact Barix Customer Support for indications how to get it.

3.1 Configuring your Linux distribution

a) First install the Ubuntu compile tools

```
sudo apt-get install build-essential
```

b) Next, add i386 architecture libraries and update the package DB:

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

c) Last, install the packages recommended (and required) by Yocto for essential development on a headless work station:

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib
chrpath socat u-boot-tools chrpath python-minimal libssl-dev
```

Now your Linux system is ready to install and compile the Development Kit with Yocto

3.2 Installing the Yocto layers of the Barix SW Development Kit

3.2.1 Prerequisites

In order to be able to fetch the IPAM-400 SDK, you need:

A) A Linux Build PC, preferably local, not remotely hosted because the script creating SD card requires physicl access to the SD card interface;

B) Read access to the following BitBucket repos (see chapter 49 for the BitBucket credentials):

- ***qiba-oem_bsp-platform**: This repository contains the manifest files needed by the repo tool to initialize the Yocto configuration for compiling the Barix SDK image.*
- ***meta-barix-sdk:** This repo contains the SIP Demo application, and the binary and include files of some Barix libraries, needed for the SIP demo application to run. The sources of the SIP Demo application are provided as a tarball, with the needed fixes to run on the IPAM-400. If you need the sources directly from the relevant GIT repo, you need access to **barix_oem_sample_apps** (see below)*
- ***meta-qiba:** This repo provides support for the Allwinner Linux modules used by Barix / Qibixx (kernel, devicetree, system utilities, FW updater backend, etc). It contains also some tools for manipulating the factory-info partition, generate web update image out of the generated by Yocto system image, and a SD-card burner tool.*

C) Optional read access to the following repos (see chapter 49 for the BitBucket credentials):

- *Barix_oem_sample_apps: This repo contains the sources of some demo applications:*
  1. ***pjsua:** Barix modified PJSUA client with added button handling, status reporting, and DTMF relay control.*
  2. ***simple player:** A simple application using the Barix proprietary Player library that plays a stream from URL*
  3. ***sip_cgi:** A CGI command handler backend to receive commands from the webUI, and pass them over to the modified pjsua SIP client.*

3.2.2 Barix oe-core Setup

Once you got the user name and password from Barix Support, perform the following steps:

1. *Install the repo bootstrap binary in your home folder:*

```
mkdir ~/bin
PATH=~/bin:$PATH
curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
```

2. *Setup the repository credentials for ipam400-oem bitbucket account:*

```
git config --global user.name "ipam400-oem"
git config --global user.email "ipam400-oem@barix.com"
git config --global credential.helper cache
```

3. *Create a directory for your oe-core setup to live in and clone the meta information. During the following steps the user is prompted for the ipam400-oem Bitbucket credentials.*

```
mkdir oe-core
cd oe-core
repo init -u https://bitbucket.org/kibix/qiba-oem-bsp-platform -b master -m barix-sdk-v1.05.xml
repo sync
```

***NOTE:** While you can use the develop (barix-sdk-develop.xml), or the master (barix-sdk-master.xml) manifest files, Barix recommends that you use the manifest for the latest officially released version of the SDK (v1.05 as of the time of writing this document). It guarantees that the dependency layers are being checked out with the same SHA1 hash tags as of the release date, and not taking the master/develop HEAD which might have changed in the meantime.*

4. *Source the export script to setup the environment. On first invocation this also copies a sample configuration to build/conf/*.conf.*

```
source export
```

With this your Yocto build environment is properly setup, all the sorces fetched, and you can start building images

3.2.3 Compiling images and burning SD card

1 *To compile the Barix SDK image, type the following command line from the build folder:*

```
bitbake core-image-barix-sdk
```

All output images for the specified MACHINE (barix-ipam400 in our case) are located in:

```
./tmp-glibc/deploy/images/barix-ipam400/
```

2 *To burn the SD card (dev/sdb in the examples below) with the generated core-image-barix-sdk use the "create-qiba-sd.sh" script located in "stuff/meta-qiba/tools" folder. For example:*

```
sudo ./create-qiba-sd.sh barix-ipam400 core-image-barix-sdk /dev/sdb
```

In the above case the U-Boot is not added to the SD-Card. To program the SD card with the U-Boot use:

```
sudo ./create-qiba-sd.sh barix-ipam400 core-image-barix-sdk /dev/sdb y
```

3 *To create an image, that can be uploaded via web update, run the command:*

```
. /create-qiba-update.sh --machine=barix-ipam400 --root=core-image-barix-sdk
```

3.2.4 Deploying the generated images

Barix supports several methods for uploading images to the target. For the purpose of this manual we can use 2 methods: webUI update, and SD card update.

Separate binaries and/or files can be also transferred diretly on a booted device using SCP.

3.2.4.1 Updating the SD card image from the Yocto environment

This is the easiest and the preferred way to run the newly generated image. It has the following advantages:

- Works always-if for any reason the u-boot and/or the rescue image in the flash are broken, the device will successfully boot off the SD card (assuming that the u-boot has been added to the image as described above)
- Saves time and effort-the SD card is prepared in less than 2 min
- This is the only way to setup initially your image

The disadvantage of this method is that you will lose the data in the data partition if you reuse the same SD card.

To burn the SD card, just follow the steps above, insert the SD card in the IPAM-400 module, plug it back in its socket, then power on the device.

3.2.4.2 Updating the SD card without recompiling the complete Yocto environment.

Starting from v1.02, Barix distibites the following images:

File name	***Description***
core-image-barix-sdk-barix-ipam400.tar.gz	The image to be burned in the root partition of the SD card
u-boot-sunxi-with-spl.bin-barix-ipam400	Bootloader to be stored in the boot partition of the SD card
qiba-update-core-image-barix-sdk-barix-ipam400-xxxxxxxxx.tar	The timestamped webupdate file
create-qiba-sd-extern.sh	Script to create the SD card

If all you want is to give the current version of the SDK kit a try without all the hastle of recompiling the complete Yocto environment from scratch, then copy all these files in a folder on your Linux PC, and just run the provided create-qiba-sd-extern.sh file like this:

```
sudo ./create-qiba-sd-extern.sh barix-ipam400  core-image-barix-sdk /dev/sdc y
```

3.2.4.3 Doing web update

In order to be able to use the Web Update, we need to have the device preloaded with an image that provides the minimal webUI to allow flashing the FW. Beware that empty devices (i.e devices without SD card image, that have only the rescue image burned in the flash) do not have this functionality. To ensure this functionality, the needed binaries and scripts are already included in the meta-qiba Yocto layer.

To upload a new image, just open your browser to the URL:

http://192.168.11.166/uifloader.html

replacing the IP address with your device/board IP address, then follow the instructions on the screen.

The device will transfer the rootfs on the SD card, leaving the data partition intact, and will reboot automatically when the update is finished

3.2.4.4 Updating the FW image manually in rescue mode

In some cases the developers may want to have a quick update without all the hastle of compiling the whole Yocto environment and/or burning the SD card, by using directly the provided update images. In this case, performing webUI update is the easiest way to go.

Unfortunately, the webUI update option is available only if we have previously burned an SD card, which has the webUI update scripts available.

*If the IPAM-400 module has no SD card with valid Yocto image, then it will boot from the flash the rescue image. Due to the limited flash memory, the rescue image has limited number of utilities preinstalled, one of which is the **qiba-update-client**, which ensures the remote update functionality.*

So, in order to update the SD card, do the following steps:

1. *Plug in an empty SD card into the IPAM-400 module, connect the device to the network via a LAN cable, and power it on.*
2. *The device will boot in rescue mode. You can use the serial console, connected to the debugging serial port as explained in the Connecting serial terminal chapter, to observe the boot messages.*
3. *Once the device boots, you can login with user root, and no password, then get the IP address of the device:*

```
root@barix-ipam400:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:08:E1:06:A4:FB
          inet addr:192.168.11.135  Bcast:192.168.11.255  Mask:255.255.255.0
          inet6 addr: fe80::208:e1ff:fe06:a4fb%lo/64 Scope:Link
          inet6 addr: fd5d:12c9:2201:1:208:e1ff:fe06:a4fb%1/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8579 errors:0 dropped:34 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:905071 (883.8 KiB)  TX bytes:1286 (1.2 KiB)
```

4. *Go in the folder of your PC where you have stored your web update image, and copy it over to the device using scp, for example:*

```
scp qiba-update-core-image-barix-sdk-barix-ipam400-20180706124013.tar
root@192.168.11.135:/tmp
```

5. *Back in the device terminal, start the qiba-update-client manually:*

```
root@barix-ipam400:~# qiba-update-client -f /tmp/qiba-update-core-image-barix-sd
k-barix-ipam400-20180706124013.tar
```

The device will reformat the SD card, will create all the needed partitions, copy the image to it, and then reboot with the new image

3.2.4.5 Deploying a binary using SCP

The easiest, and the quickest way to test a newly copiled binary or script, is to copy it directly on to a working device and run it from there.

To do that, locate the binary you want to copy, and the desired destination, then use the SCP command. For example, to copy the barix-pjsua binary (assuming we are in the Yocto build folder), we do:

1. *Login to the device (ether by serial terminal, or ssh). For example, using SSH:*

```
$ ssh root@192.168.11.218
root@192.168.11.218's password:
```

2. *Type "`oem_devkit_17`" for password. Next, you need to stop the running application:*

```
root@barix-ipam400:~# /etc/init.d/pjsua stop
root@barix-ipam400:~#
```

3. *From your Linux development PC, copy the new binary:*

```
scp tmp-glibc/work/cortexa7hf-neon-vfpv4-oe-linux-gnueabi/pjsua/local-r0/barix-
pjsua root@192.168.11.218:/usr/bin/
root@192.168.11.218's password:
barix-pjsua
100% 7948KB   7.8MB/s   00:01
alex@kubuntu-vm:~/oe-core/build$
```

4. *Last, restart again the application:*

```
root@barix-ipam400:~# /etc/init.d/pjsua start
```

3.2.4.6 Remote update

The Barix rescue image is built with a remote update client in it. If the boot from the SD card fails, the device will attempt to load the rescue image from the flash. If you have the serial port connected, you will be able to see the console prompt. You can enter into rescue image also if you keep pressed the reset button while powering on the device.

The remote update may not, or may not be activated on the rescue image at production time. If the `/mnt/shadow/update_servers.txt` file exists, and contains download URL, the device will try to fetch the new image from that address.

***NOTE1:** Be aware that during this process the SD card will be completely erased, formatted, and the image burned to it. If you have any data on it, they will be lost.*

***NOTE2:** The shadow partition might be mounted read only. If this is the case, use the following command to mount it for read/write in order to change the update_servers.txt:*

```
mount -o rw,remount /mnt/shadow
```

4 A detailed look at the SIP DEMO application

In this we will discuss the SIP Demo application. We will go through the process of creating/understanding the application start script, and the automatic configuration generation of the pjsua.conf file. So, in fact we have:

- *A SIP client, based on the PJSUA library*
- *A wrapper application to interact with the webUI and the SIP client (sip_cgi)*

So, let's start reviewing all components one by one ….

4.1 Modifying the PJSIP package

Barix has made changes to the PJSIP and PJSUA libraries in order to add some hooks in to the source code to be able to get status information from the application and send some commands to it. These changes have been split in two parts:

4.1.1 Compiling PJSIP Library for the needs of the SIP demo application

The recipe of the PJSIP package is located in `meta-barix-sdk/recipes-external/pjsip` folder. The recipe compiles also the standard pjsip client binaries, but since we are only interested using the colpiled library, we are not copying them on the device.

4.1.2 Barix custom changes of the PJSUA client

In order to be able to communicate with the CGI backend (sip_cgi) that reads the PJSIP status and sends it to the webUI, and receives some commands form the web page, Barix has modified a bit the PJSUA client provided with the PJSIP library by adding an UDP socket class, that is used to communicate with the CGI backend. The source code of the modified PJSUA client is provided as tarball file, located in the `meta-barix-sdk/recipes-apps/sip-demo/pjsua` folder. If the `pjsua_git.bb` recipe is used intead of the default `pjsua_local.bb`, then the source will be fetched from the **barix_oem_demo_apps repository on BitBucket**.

The source code si compiled, then links it against the PJSIP library that is already preinstalled in the Yocto staging directory. The resulting `barix_pjsua` binary is then copied in the `/usr/local/bin` folder on the target image.

4.2 Understanding the sip_demo package

The sip demo package has been created with the following functionality:

- *The main application that will be executed at device startup*
- *Compile and install the demo applications and helpers – PJSUA based SIP client and SIP-CGI agent/backend*
- *Install all the relevant configuration templates and the webUI files*

Below we explain shortly about each component:

4.2.1 The webUI interface files

The webUI files are available as a tarball in the files section of the `meta-barix-sdk/recipes-apps/sip-demo/sip-demo-web-ui/` recipe.

Ince decompressed, you can find the CGI scripts, that generate dynamic HTML content on demand when processed by the haserl script, located in the cgi-bin folder.

The frame files (named `uif*.html`) and the help pages (named `uih*.html`) are in the main webUI folder. The only exception is uifhome, which has been modified to load the home page of the currently active demo application. To do this, it is renamed to .cgi and moved to the cg-bin folder. The code is pretty simple to

understand and is a good example to follow:

```
<%
function get_home_page {
      homepage=$(/sbin/uci get -q application.main_config.active_app)
      echo -n "/cgi-bin/"
      echo -n "$homepage"
      echo -n "_uihome.cgi"
}

function get_help_page {
      helppage=$(/sbin/uci get -q application.main_config.active_app)
      echo -n "/$helppage"
      echo -n "_help.html"
}

echo -n '
<html>
<frameset  id="uifhome" cols="650,350,*" frameborder=no border=0>
      <frame src='
get_home_page
echo -n ' noresize name="m" marginwidth=0 marginheight=0>
      <frame src='
get_help_page
echo ' noresize marginwidth=0 marginheight=0>
</frameset><noframes>Please use a frame enabled browser</noframes>
</html>'
%>
```

The tags "<%" and "%>"instruct the lighthttpd server to ignore everything enclosed between them, and pass it to the haserl binary. The enclosed code is executed like a bash script, and the echo –n ' … ' statements generate the real HTML content. The frame source link is generated by calling the get_home_page() and get_help_page() functions that properly detect the active application, and redirect to the correct cgi home and help pages. Below you find a list of the dynamic web pages in the cgi-bin folder:

File Name	*Purpose*
command.cgi	*A CGI script to provide status of the Simple Player to the webUI, and receive some commands from it. Shows how the POST parameters are being fetched and parsed, and can be easily extended by adding other commands, or easily adapted for another projects.*
config.cgi	*A CGI script that is being called when the user clicks on the Submit button to send a POST request. This script then takes care to apply the settings, and display the right message when the IP settings do change. Please do not modify this script unless you get deep understanding about the Barix Linux*

	Ecosystem, just use it as it is.
download_applog .cgi	*A CGI script to download /var/log/messages, including the logrotated gzipped files too*
download_weblog.cgi	*A script to download the lighttpd log files*
format.cgi	*A script to format the SD card. It is started by a button on the DEFAULTS menu tab*
menu.cgi	*A script implementing the navigation menu of the home page*
pjsua_uihome.cgi	*SIP client home page. Shows the status of the SIP client and allows sending commands to it*
Sip.cgi	*This is the sip_cgi application from the sip_demo package that provides the status of the PJSUA client, and sends commands to it.*
uidefaults.cgi	*Factory defaults page implementation*
uilogs.cgi	*Logs web page*
uinetwork.cgi	*The SIP applications and system settings page*
uireboot.cgi	*A script implementing the Reboot page*
uistatus.cgi	*A script implementing the Status page*

4.2.2 The SIP CGI handler

The SIP CGI handler is the "man-in-the-middle" between the webUI, and the modified PJSUA application. It is an example use the CUdpSocket class from the Barix proprietary utility_lib. When the browser sends request to `cgi-bin/sip.cgi`, *the* `lighttpd` *executes it, passing the request parameters to it. The SIP CGI agent then forwards the request to UDP port 5555, where the barix-pjsua is listening for commands. The source code can be found in the files section of the* `barix-sdk/recipes-apps/sip-demo/sip-demo-config` *recipe.*

4.2.3 The UCI default settings

The default settings and configuration templates are provided as a tarball in the files section of the `meta-barix-sdk/recipes-apps/sip-demo/sip-demo-config/` *recipe. It contains not only SIP deno configuration files, but also the UCI configuration files of the depending services. All of them can be found in the* `barix/config/defaults/` *once the tarball is decompressed.*

Below we list only the options in the files of our interest:

4.2.3.1 SIP Application defaults (application)

Contains settings that are common for all SIP Demo applications:

Option name and default value	***Purpose***
application.main_config.active_app=pjsua	Defines which one of the SIP demo apps to be started at boot. Currently selects between "SIP Client" and "Simple Player"
application.audio.amplifier=on	Switches ON/OFF the amplifier (Speaker OUT)
application.audio.mic_linein=mic	Switches between MIc/Line In input
application.audio.volume=50	Sets the default volume in %
application.audio.mic_gain	Sets the Microphone gain in dB
application.audio.mic_boost	Switches ON/OFF the microphone boost
application.audio.ad_gain	Sets the AD gain in dB
application.audio.silence_playback	

4.2.3.2 PJSUA default settings (pjsua)

Contains pjsua specific defaults:

```
package 'pjsua'

config section 'sip_account'
      option registrar  'change_me.server.com'
      option username         'change_me'
      option password         'change_me'
      option reg_to           '600'

config section 'aec'
      option no_vad           'y'
      option ec_tail          '250'
      option ec_opt           'disabled'

config section 'misc'
      option autoanswer 'n'
      option cmd_port         '52221'
      option capture_lat      '100'
      option playback_lat     '100'
      option quick_dial_num   'change_me'
      option dtmf_pattern     '1234'
```

5 Using the SIP Demo Application

5.1 Making/Receiving Call

After to have set your own SIP credentials on Settings page, to place calls put on home page extension@server after sip: and press the DIAL button.

e.g.

```
200@192.168.0.243 or 200@myserver.com
```

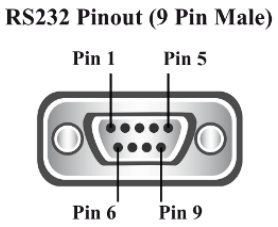
To answer a call, press the button "ANSWER", and to close a call-press the button "HANGUP".

5.2 Placing a Call with digital input

To place a call with digital input, shorten the IN0 contacts on rear connector. A call to the preset ***Call On Input ID*** *will be started.*

5.3 Trigger the RS232 RTS pin with DTMF code1

The Annuncicom 60 does not have a digital, or relay output. However it is possible to toggle the RTS pin (RTS is pin 7, GND is pin 5; see picture below) of the RS232 port using DTMF code. Configure the trigger code in the DTMF Pattern field (5 digits maximum) on the web UI. The RTS output will be unconditionally set to "Off" when the call is closed



The state of the pin is reported on home page (green/active, +6V; gray/not active, -6V).

5.4 Understanding the PJSUA configuration file

To add/change the configuration parameters not accessible via WEB interface, it's possible to edit the `pjsua` *file in* `/barix/config/templates/templates/.`

```
#
# PJSUA config options coming from the template
# To modify them edit the /barix/config/templates/templates/pjsua file
# for more PJSUA config options check the PJSIP documentation:
# http://www.pjsip.org/pjsua.htm#reference
#
```

[1] This option is not yet functional-portingthis feature form IPAM-390 to the new IPAM-400 is still in progress.

```
# Optional features. Uncomment if needed
#--log-file=/var/log/pjsua.log
#--log-level=5
#--app-log-level=5
#--log-append
#--dis-codec=g711
#--auto-loop
#--auto-rec

# Barix custom default settings:
# Disable console. It must be disabled or we could not run it as a
# service in background
--no-cli-console
--use-cli
--cli-telnet-port=52221

# disable TCP. Uncomment if TCP/TLS is required
--no-tcp

# Use the default soundcard. Change it here if you are using another
# (ex. USB sound card)
# Note 1: PJSIP device numbering starts from 1 (0 means "use default"),
#         ALSA-from 0! (ex. USB card would be 2, in ALSA 1)
# Note 2: For USB cards to work the kernel needs be compiled with
#         USB soundcard support!
--playback-dev=0
--capture-dev=0

# Use the standard RTP port
--rtp-port=5004

# use any realm
--realm=*

# limit the calls to 1 only
--max-calls=1
```

To have the new configuration running, the pjsua application must be restarted with the command:

```
/etc/init.d/pjsua restart
```

For example, to add audio loop (the incoming audio from remote peer is looped internally and sent back) and auto-answer functions, append:

```
# loop audio
--auto-loop
```

NOTE 1: *For testing/debugging purpose it's useful to start with auto-loop configured; if the audio loop works properly, but in normal call there is not incoming/outgoing audio, the cause could be an improper audio parameters configuration (see also section 8.4, Using the ALSA mixer) or an incorrect wiring.*

See below the complete list of pjsua configuration options.

```
Usage:
  pjsua [options] [SIP URL to call]

General options:
  --config-file=file  Read the config/arguments from file.
  --help              Display this help screen
  --version           Display version info

Logging options:
  --log-file=fname    Log to filename (default stderr)
  --log-level=N       Set log max level to N (0(none) to 6(trace)) (default=5)
  --app-log-level=N   Set log max level for stdout display (default=4)
  --log-append        Append instead of overwrite existing log file.

  --color             Use colorful logging (default yes on Win32)
  --no-color          Disable colorful logging
  --light-bg          Use dark colors for light background (default is dark bg)
  --no-stderr         Disable stderr

SIP Account options:
  --registrar=url     Set the URL of registrar server
  --id=url            Set the URL of local ID (used in From header)
  --realm=string      Set realm
  --username=string   Set authentication username
  --password=string   Set authentication password
  --contact=url       Optionally override the Contact information
  --contact-params=S  Append the specified parameters S in Contact header
  --contact-uri-params=S  Append the specified parameters S in Contact URI
  --proxy=url         Optional URL of proxy server to visit
```

```
                        May be specified multiple times
  --reg-timeout=SEC     Optional registration interval (default 300)
  --rereg-delay=SEC     Optional auto retry registration interval (default 300)
  --reg-use-proxy=N     Control the use of proxy settings in REGISTER.
                        0=no proxy, 1=outbound only, 2=acc only, 3=all (default)
  --publish             Send presence PUBLISH for this account
  --mwi                 Subscribe to message summary/waiting indication
  --use-ims             Enable 3GPP/IMS related settings on this account
  --use-srtp=N          Use SRTP?  0:disabled, 1:optional, 2:mandatory,
                        3:optional by duplicating media offer (def:0)
  --srtp-secure=N       SRTP require secure SIP? 0:no, 1:tls, 2:sips (def:1)
  --use-100rel          Require reliable provisional response (100rel)
  --use-timer=N         Use SIP session timers? (default=1)
                        0:inactive, 1:optional, 2:mandatory, 3:always
  --timer-se=N          Session timers expiration period, in secs (def:1800)
  --timer-min-se=N      Session timers minimum expiration period, in secs (def:90)
  --outb-rid=string     Set SIP outbound reg-id (default:1)
  --auto-update-nat=N Where N is 0 or 1 to enable/disable SIP traversal behind
                        symmetric NAT (default 1)
  --disable-stun        Disable STUN for this account
  --next-cred           Add another credentials

SIP Account Control:
  --next-account        Add more account

Transport Options:
  --set-qos             Enable QoS tagging for SIP and media.
  --local-port=port     Set TCP/UDP port. This implicitly enables both
                        TCP and UDP transports on the specified port, unless
                        if TCP or UDP is disabled.
  --ip-addr=IP          Use the specifed address as SIP and RTP addresses.
                        (Hint: the IP may be the public IP of the NAT/router)
  --bound-addr=IP       Bind transports to this IP interface
  --no-tcp              Disable TCP transport.
  --no-udp              Disable UDP transport.
  --nameserver=NS       Add the specified nameserver to enable SRV resolution
                        This option can be specified multiple times.
  --outbound=url        Set the URL of global outbound proxy server
                        May be specified multiple times
  --stun-srv=FORMAT     Set STUN server host or domain. This option may be
```

```
                      specified more than once. FORMAT is hostdom[:PORT]

Audio Options:
  --add-codec=name    Manually add codec (default is to enable all)
  --dis-codec=name    Disable codec (can be specified multiple times)
  --clock-rate=N      Override conference bridge clock rate
  --snd-clock-rate=N  Override sound device clock rate
  --stereo            Audio device and conference bridge opened in stereo mode
  --null-audio        Use NULL audio device
  --play-file=file    Register WAV file in conference bridge.
                      This can be specified multiple times.
  --play-tone=FORMAT  Register tone to the conference bridge.
                      FORMAT is 'F1,F2,ON,OFF', where F1,F2 are
                      frequencies, and ON,OFF=on/off duration in msec.
                      This can be specified multiple times.
  --auto-play         Automatically play the file (to incoming calls only)
  --auto-loop         Automatically loop incoming RTP to outgoing RTP
  --auto-conf         Automatically put calls in conference with others
  --rec-file=file     Open file recorder (extension can be .wav or .mp3
  --auto-rec          Automatically record conversation
  --quality=N         Specify media quality (0-10, default=6)
  --ptime=MSEC        Override codec ptime to MSEC (default=specific)
  --no-vad            Disable VAD/silence detector (default=vad enabled)
  --ec-tail=MSEC      Set echo canceller tail length (default=256)
  --ec-opt=OPT        Select echo canceller algorithm (0=default,
                        1=speex, 2=suppressor)
  --ilbc-mode=MODE    Set iLBC codec mode (20 or 30, default is 30)
  --capture-dev=id    Audio capture device ID (default=-1)
  --playback-dev=id   Audio playback device ID (default=-1)
  --capture-lat=N     Audio capture latency, in ms (default=100)
  --playback-lat=N    Audio playback latency, in ms (default=100)
  --snd-auto-close=N  Auto close audio device when idle for N secs (default=1)
                      Specify N=-1 to disable this feature.
                      Specify N=0 for instant close when unused.
  --no-tones          Disable audible tones
  --jb-max-size       Specify jitter buffer maximum size, in frames (default=-1)
  --extra-audio       Add one more audio stream

Media Transport Options:
  --use-ice           Enable ICE (default:no)
```

```
  --ice-regular       Use ICE regular nomination (default: aggressive)
  --ice-max-hosts=N   Set maximum number of ICE host candidates
  --ice-no-rtcp       Disable RTCP component in ICE (default: no)
  --rtp-port=N        Base port to try for RTP (default=4000)
  --rx-drop-pct=PCT   Drop PCT percent of RX RTP (for pkt lost sim, default: 0)
  --tx-drop-pct=PCT   Drop PCT percent of TX RTP (for pkt lost sim, default: 0)
  --use-turn          Enable TURN relay with ICE (default:no)
  --turn-srv          Domain or host name of TURN server ("NAME:PORT" format)
  --turn-tcp          Use TCP connection to TURN server (default no)
  --turn-user         TURN username
  --turn-passwd       TURN password

Buddy List (can be more than one):
  --add-buddy url     Add the specified URL to the buddy list.

User Agent options:
  --auto-answer=code  Automatically answer incoming calls with code (e.g. 200)
  --max-calls=N       Maximum number of concurrent calls (default:4, max:255)
  --thread-cnt=N      Number of worker threads (default:1)
  --duration=SEC      Set maximum call duration (default:no limit)
  --norefersub        Suppress event subscription when transferring calls
  --use-compact-form  Minimize SIP message size
  --no-force-lr       Allow strict-route to be used (i.e. do not force lr)
  --accept-redirect=N Specify how to handle call redirect (3xx) response.
                      0: reject, 1: follow automatically,
                      2: follow + replace To header (default), 3: ask

CLI options:
  --use-cli           Use CLI as user interface
  --cli-telnet-port=N CLI telnet port
  --no-cli-console    Disable CLI console
```

5.4.1 The “hidden” SIP Client telnet interface

The SIP client is listening for telnet connections at port 52221. You can manually control a large number of parameters of the PJSUA based SIP client. For more information and the list of supported commands take a look at the PJSUA user manual. Below is a command to get the detailed status:

```
macmini: asi$ telnet 192.168.11.159 52221
Trying 192.168.11.159...
Will map carriage return on output.
Will send carriage returns as telnet <CR><LF>.
Connected to 192.168.11.159.
```

```
Escape character is '^]'.
barix> dd

12:30:21.761   pjsua_core.c !Start dumping application states:
PJLIB (c)2008-2009 Teluu Inc.
Dumping configurations:
 PJ_VERSION                : 2.4.5
 PJ_M_NAME                 : arm
 PJ_HAS_PENTIUM            : 0
 PJ_OS_NAME                : arm-buildroot-linux-gnueabi
 PJ_CC_NAME/VER_(1,2,3)    : gcc-4.8.3
 PJ_IS_(BIG/LITTLE)_ENDIAN : little-endian
 PJ_HAS_INT64              : 1
 PJ_HAS_FLOATING_POINT     : 0
 PJ_DEBUG                  : 1
 PJ_FUNCTIONS_ARE_INLINED  : 0
 PJ_LOG_MAX_LEVEL          : 5
 PJ_LOG_MAX_SIZE           : 4000
 PJ_LOG_USE_STACK_BUFFER   : 1
 PJ_POOL_DEBUG             : 0
 PJ_HAS_POOL_ALT_API       : 0
 PJ_HAS_TCP                : 1
 PJ_MAX_HOSTNAME           : 128
 ioqueue type              : select
 PJ_IOQUEUE_MAX_HANDLES    : 64
 PJ_IOQUEUE_HAS_SAFE_UNREG : 1
 PJ_HAS_THREADS            : 1
 PJ_LOG_USE_STACK_BUFFER   : 1
 PJ_HAS_SEMAPHORE          : 1
 PJ_HAS_EVENT_OBJ          : 1
 PJ_ENABLE_EXTRA_CHECK     : 1
 PJ_HAS_EXCEPTION_NAMES    : 1
 PJ_MAX_EXCEPTION_ID       : 16
 PJ_EXCEPTION_USE_WIN32_SEH: 0
 PJ_TIMESTAMP_USE_RDTSC:   : 0
 PJ_OS_HAS_CHECK_STACK     : 0
 PJ_HAS_HIGH_RES_TIMER     : 1

Dumping endpoint 0x209164:
```

```
 Dumping caching pool:
   Capacity=0, max_capacity=0, used_cnt=21
  Dumping all active pools:
              pjsua:     5984 of     9024 (66%) used
       pept0x209100:    49076 of    52096 (94%) used
          pjsua-app:     7644 of    10024 (76%) used
           tsxlayer:     4332 of     5120 (84%) used
         ua0x216e50:     2284 of     3072 (74%) used
            med-ept:    24436 of    26112 (93%) used
      alsa_aud_base:    13200 of    13568 (97%) used
           alsa_aud:      100 of      256 (39%) used
          codec-mgr:      196 of      256 (76%) used
              speex:      196 of     4096 (4%) used
                gsm:      196 of     4096 (4%) used
               g711:      196 of     4096 (4%) used
               g722:      196 of     1024 (19%) used
                l16:      196 of     4096 (4%) used
              evsub:     1564 of     2048 (76%) used
        udp0x22b3e0:      824 of     1024 (80%) used
       glck0x22b7f0:      408 of      512 (79%) used
        rtd0x22b9f8:     4592 of    12096 (37%) used
        acc0x22ca00:      364 of      512 (71%) used
        acc0x22cc08:      660 of      768 (85%) used
       regc0x22ce10:     2400 of     3072 (78%) used

  Total    119044 of    156968 (75 %) used!
 Endpoint pool capacity=52096, used_size=49076
 Outstanding transmit buffers: 0
 Dumping listeners:
 Dumping transports:
  udp0x22b3e0 udp 0.0.0.0:5060 [published as 192.168.11.159:5060] (refcnt=3)
 Timer heap has 3 entries
Dumping PJMEDIA capabilities:
  Total number of installed codecs: 14
   Audio codec # 0: pt=98 (speex @16KHz/1, 27.8Kbps, 20ms vad cng plc penh)
   Audio codec # 1: pt=97 (speex @8KHz/1, 15.0Kbps, 20ms vad cng plc penh)
   Audio codec # 2: pt=99 (speex @32KHz/1, 29.6Kbps, 20ms vad cng plc penh)
   Audio codec # 3: pt=104 (iLBC @8KHz/1, 13.3Kbps, 30ms vad plc penh)
   Audio codec # 4: pt=3 (GSM @8KHz/1, 13.2Kbps, 20ms vad plc)
   Audio codec # 5: pt=0 (PCMU @8KHz/1, 64.0Kbps, 20ms vad plc)
```

```
   Audio codec # 6: pt=8 (PCMA @8KHz/1, 64.0Kbps, 20ms vad plc)
   Audio codec # 7: pt=9 (G722 @16KHz/1, 64.0Kbps, 20ms vad plc)
   Audio codec # 8: pt=11 (L16 @44KHz/1, 705.6Kbps, 10ms vad plc disabled)
   Audio codec # 9: pt=10 (L16 @44KHz/2, 1.41Mbps, 10ms vad plc disabled)
   Audio codec #10: pt=120 (L16 @8KHz/1, 128.0Kbps, 20ms vad plc disabled)
   Audio codec #11: pt=121 (L16 @8KHz/2, 256.0Kbps, 20ms vad plc disabled)
   Audio codec #12: pt=122 (L16 @16KHz/1, 256.0Kbps, 20ms vad plc disabled)
   Audio codec #13: pt=123 (L16 @16KHz/2, 512.0Kbps, 20ms vad plc disabled)

Dumping media transports:

Dumping transaction table:

 Total 0 transactions
 - none -

Number of dialog sets: 0

Dumping pjsua server subscriptions:
<sip:192.168.11.159:5060>
  - none -
  sip:change_me@change_me.server.com
  - none -

Dumping pjsua client subscriptions:
- no buddy list -

12:30:21.881   pjsua_core.c  Dump complete

barix>
```

NOTE 1: *To Enable the webUI control for changing the telnet port open the* `/usr/local/www/current/cgi-bin/uinetwork.cgi` *file on the device, and uncomment the command port settings.*

NOTE 2: *There are commands to add another SIP account, rearrange the order of codec, etc. While they may work at runtime, all changes will be lost when the application is restarted since the application start script regenerates the* `/etc/pjsua.conf` *file everytime. To make your changes permanent, change the pjsus template in the* `/baric/config/templates/templates` *folder.*

6 Barix Linux Ecosystem

This chapter gives a brief overview of the Barix Linux System from the user space point of view.

6.1 SPI Flash Partitioning

The IPAM-400 module has 16 MB SPI flash, which is partitioned according to the following layout:

No	*Partition Name*	*Size*	*Purpose*	*FS type*
0	*U-boot*	*1 MB*	*The U-Boot partition contains the SPL U-Boot. This partition starts at offset 0x000000 and ends at 0x100000.*	*raw*
1	*Rescue Image*	*13 MB*	*The Rescue image is a FIT image containing the Linux Kernel, the Kernel DTB, and the RAM disk with the rescue root fs. This partition starts at offset 0x100000 and ends at 0xE00000.*	*raw*
2	*Shadow parameters*	*1MB*	*The Shadow parameters partition contains a set of parameters that can be used by the Rescue image and the Firmware Upgrade Client. This partition starts at offset 0xE00000 and ends at 0xF00000. At boot it gets mounted to /mnt/shadow*	*vfat*
3, 4	*Factory parameters*	*4 MB*	*Factory parameters are stored in the Production parameters area that starts at the beginning of the last megabyte of the SPI flash.* *The MAC addresses of the Ethernet and of the WiFi card are stored at the beginning of the Production parameters partition as hexadecimal values. Each MAC is followed by the CRC8 of the MAC addresses itself. These addressed are read by the U-Boot and they are made available to the user space. Up to 4 MAC addresses can be stored at this location.* *Hardware information, image information and test parameters are stored in a JSON file starting at an offset of 64Kbyte of the Production parameters area. The JSON file is stored with a small header of 6 bytes containing the file size (4 bytes) and the file CRC16. A redundant copy of this JSON file is stored at on offset of 64Kbyte from the primary file and it's used in case the primary file is invalid.*	*raw*

You can view the current SPI Flash layout with the following command:

```
root@barix-ipam400:~ # cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00001000 "uboot"
mtd1: 00d00000 00001000 "fit"
mtd2: 00100000 00001000 "shadow"
mtd3: 00010000 00001000 "production-ro"
mtd4: 000f0000 00001000 "production"
root@barix-ipam400~#
```

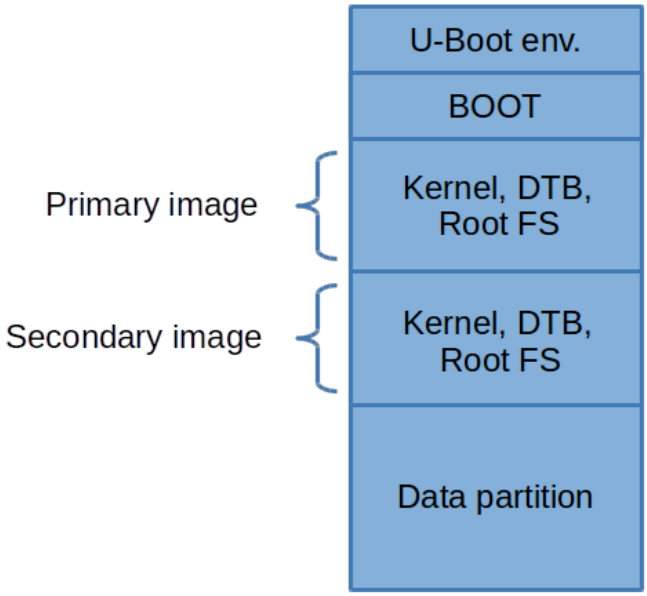
6.2 SD Card layout

The Micro-SD will be formatted so that it contains partitions for two Linux images plus a large user data space.

One image will be defined the primary image, the other is the secondary image and is used in case a newly

downloaded image does not work. The function of the two images is swapped after a successful firmware upgrade is executed.

The Micro-SD layout is shown in the following picture:



The Linux Kernel and the kernel device tree (DTB) are embedded in the Root FS. This reduces the total number of needed partitions and simplifies the handling of the updates.

6.3 System V Init

For system initialization the standard System V Init is used. Init is started after kernel initialization, by executing the `/sbin/init`. It is a direct or indirect ancestor of all processes, maintains orphaned processes, and starts or restarts processes after they end (e.g. the login console after a user logs out).

*Init employs a concept of **runlevels**, which are the states of the system. The runlevels are defined in the file `/etc/inittab`; for each runlevel a set of actions is defined. There are 8 runlevels: 0 to 6 and S (or s); three of the runlevels are reserved for special action: 0 = halt, 1 = single user, 6 = reboot. Other runlevels are defined by the system.*

*For initialization System V Init calls **rc** scripts, which are a set of shell scripts located in directories `/etc/init.d`, `/etc/rc.0` to `/etc/rc.6` and `/etc/rc.S`.*

For each runlevel a set of actions on entering and leaving is defined. Each directory contains scripts (or symbolic links to scripts in `/etc/init.d`) that start or stop the specific service. The scripts are ordered by name, each name is prefixed with a number giving the order of execution. Start scripts are prefixed with "S", stop scripts with "K" (kill). When Init changes the runlevel it first executes the "kill" scripts for the previous runlevel and then the "start" scripts for the new runlevel.

6.4 Run levels

Table 1 lists the system run levels.

The system boots in run-level S and then enters run level 2, which is the default run level for system start up, the system starts the default application and provides a login prompt on the serial console.

On system shut-down run level 0 is entered, on reboot, run level 6. Run levels 3 to 4 are not used and they

link to run level 2.

Run level 5 is dedicated for production testing.

Run level 1 is dedicated to system administration, it does not start the application.

Run level	Description
0	Halt
S	Start-up; executed at boot time before entering any other runlevel
1	Reserved for single-user mode without application start
2	Not used, identical to level 5
3	Not used, identical to level 5
4	Not used, identical to level 5
5 (default)	Full Application functionality with serial console login
6	Reboot
c	Pseudo run level for system re-configuration. Does not change the current run level, just executes the respective **rc** script.

6.5 Configuration run level

The Init used in Linux offers several pseudo run levels (a,b,c) to invoke certain system wide actions. Barix uses the pseudo runlevel c ("c" for configuration) for the purpose of system reconfiguration via the web UI.

When Init with runlevel c is invoked, it does not change the current runlevel (in normal operation it stays at 2), but just executes once the respective script `/etc/init.d/rcC` *which performs the reconfiguration.*

6.6 Configuration Framework

The ARM based IPAM 400 platform has a software architecture that is common to Unix systems, utilizing many software components (services, background tasks, scripts) that cooperate and communicate with each other.

The large variety of system components and services, each with its own configuration file syntax, typically located in the /etc directory of the root file system, raised the need for Barix to create a unified configuration interface for all system components. This interface is the Configuration Manager, which is accessed using

the web UI.

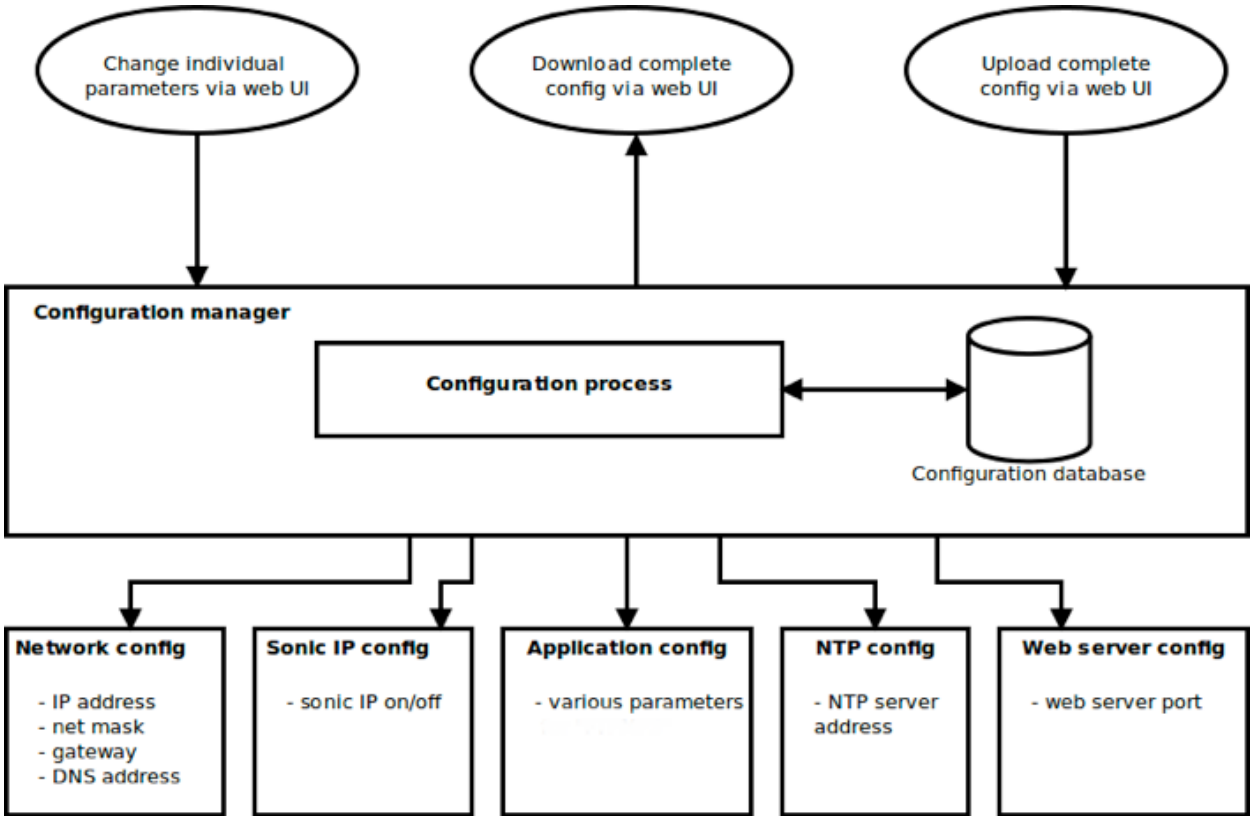
6.6.1 Configuration Manager

*The architecture of the configuration system is depicted in the picture below. The configuration manager consists of a database holding all system settings (**Configuration database**) and the **Configuration process** providing interface to the web UI on one side and passing the configuration to the system components in the right format on the other side. The configuration database is stored in the root file system, effectively in the NAND flash memory.*

The functions of the configuration process are the following:

- *Maintain the configuration database (create, store, update)*
- *Provide functions to the web UI (and possibly to "remote update"): changing individual parameters, downloading of the complete configuration, uploading of the complete configuration*
- *Create individual configuration files for each system component: extract relevant configuration parameters, save in the right format*
- *Restart individual system services, if needed, in order to apply the new configuration*

The benefit of this approach is that a new system configuration can be applied without a system reset, and at the same time the new configuration is stored permanently.



6.6.2 Configuration Framework implementation

The implementation of the Configuration Manager is based on OpenWRT's program UCI. UCI maintains a text-based database of configuration parameters in the "key=value" form and provides a simple hierarchy of 2 levels. A change-commit access to the database is managed. Modified parameters need to be committed for permanent storage. UCI natively supports the following functions:

- *parameter read and write*

- *add and delete parameters*
- *list parameters, list modified (uncommitted) parameters*
- *configuration database dump*
- *configuration import*

Additional scripts have been built around UCI to:

- *integrate with the WEB UI*
- *automatically generate configuration files for system programs and the application*
- *automatic restart of system services after configuration change*

6.6.3 Configuration Database

6.6.3.1 Folder structure

The configuration database is stored in the /barix/config folder. The following structure is used:

Directory	Description
/current	Current device configuration
/defaults	The default configuration
/templates	Templates for automatic generation of configuration files in /etc

6.6.3.2 UCI internal configuration file format

UCI stores configuration in multiple text-based files. The structure is described in detail in the OpenWRT documentation.2 Barix uses the following conventions:

- *a three level hierarchy is used: package, section, parameters*
- *parameters related to a single package are stored in a file with the "package" name, e.g. the parameters belonging to the package httpd are stored in the file /barix/config/current/httpd*
- *the parameters are referenced as package.section.parameter*
- *each package corresponds to a subsystem, e.g. network, ntp, rtc, timezone, etc.*
- *each file can be further broken down into section, e.g. network.eth0, network.sonic_ip*
- *sections contain the actual parameters*

Example of a configuration file "network":

```
package 'network'

config interface 'eth0'
      option proto 'dhcp'
```

[2] http://wiki.openwrt.org/doc/uci

```
        option ipaddr '192.168.1.100'
        option netmask '255.255.255.0'
        option gateway '192.168.1.1'

config sonic_ip 'sonic_ip'
        option enabled 'true'
          option volume '50%'
```

The structure described above is used for internal representation and for the default configuration. Configuration dump (upload and download the complete configuration to/from the device) also uses the same syntax, however all packages are listed in a single file - the internal files are simply concatenated.

6.6.3.3 Current device configuration

The device current configuration is located in the `/barix/config/current` folder.

In order to prevent configuration loss on system update (root filesystem re-flashed), the configuration parameters are located on a dedicated NAND partition, which is not erased during update. This partition is mounted to `/barix/local` and the `/barix/config/current` is just a symbolic link to `/barix/local/config`. The mounting is handled by the `/etc/init.d/mount_config` startup script.

A new device straight from the factory contains an empty configuration partition. This is detected during the first start-up and in such a case, the default parameters are simply copied into the current configuration folder.

6.6.3.4 Default configuration

The default settings are located in the `/barix/config/defaults` folder. If factory defaults are applied the files from the defaults configuration folder "defaults" are copied over to the runtime configuration folder "current".

A selective copy can be done to omit certain settings, e.g. as the network settings are not changed, if the "factory defaults" option was selected over the web interface.

The default parameters can be altered simply by modifying the appropriate files in the "defaults" folder (i.e.. in the root filesystem image).

6.6.3.5 Runtime configuration

UCI maintains a copy of the runtime configuration in a temporary folder `/var/run/.uci` This folder contains the uncommitted changes.

6.6.4 Interfaces

6.6.4.1 Binaries

Configuration subsystem binaries and scripts are stored in the following files and locations:

File	Description
/sbin/uci	The main command line interface to UCI
/lib/config/functions.sh	Functions for WEB UI integration

File	Description
/lib/config/gen_config.sh	Script for the generation of configuration files in /etc for system components

6.6.4.2 UCI command line interface

The UCI command line program is described in detail in the OpenWRT documentation.3 This chapter lists only the most common use cases:

- *set parameter value:*

```
uci set  package.section.parameter=value
```

- *get a single parameter value:*

```
uci get  package.section.parameter
```

- *list all parameters:*

```
uci show
```

- *commit changes in order to be permanently stored:*

```
uci commit
```

6.6.4.3 WEB UI integration

In /lib/config/functions.sh high level Bash script functions are defined for easy use in shell scripts like init scripts, CGI scripts, etc. For forward compatibility reasons it is recommended to use the functions defined below instead of calling UCI directly.

These functions can be included using the Bash syntax:

```
. /lib/config/functions.sh
```

The following functions are defined:

Name	Parameters	Description
cfg_print_param	1. full parameter name	Prints a value of a parameter to stdout. The value is printed without end of line. Example: *cfg_print_param network.eth0.proto*

[3] http://wiki.openwrt.org/doc/uci

Name	Parameters	Description
cfg_bool_is_true	1. full parameter name	Assumes a bool parameter, returns true or false according to the bool parameter value. To be used in expressions. Example: *if cfg_bool_is_true network.sonic_ip.enabled ; then /usr/local/sbin/sonic_ip eth0; fi*
cfg_string_compare	1. full parameter name 2. string to compare with	Compares parameter value with a string and returns true if the strings are equal, false otherwise. Example: *if cfg_string_compare network.eth0.proto "dhcp" ; then /usr/sbin/dhcpcd; fi*
cfg_set_param	1. full parameter name 2. value	Sets parameter to a value Example: *cfg_set_param network.sonic_ip.enabled false*
cfg_save	None	commits configuration parameters and restarts the respective services
cfg_has_changed	1. full parameter name	Returns true if the configuration parameter in argument has changed, false otherwise
cfg_restarting_services	None	Returns true if service restarting is in progress, false otherwise (i.e. system again fully functional).
cfg_dump_database	None	Dumps the complete configuration to standard output

Name	Parameters	Description
cfg_restore_database	1. File with configuration dump	Imports configuration from a file. Restarts the affected system services.
cfg_restore_defaults	None	Restore factory defaults and restart all affected services
cfg_restore_soft_defaults	None	Restore factory defaults without network settings and restart all affected services

6.6.4.4 Automatic system service restarting

After the configuration parameters are applied, the affected system services need to be restarted. This is automatically done by the above-mentioned functions.

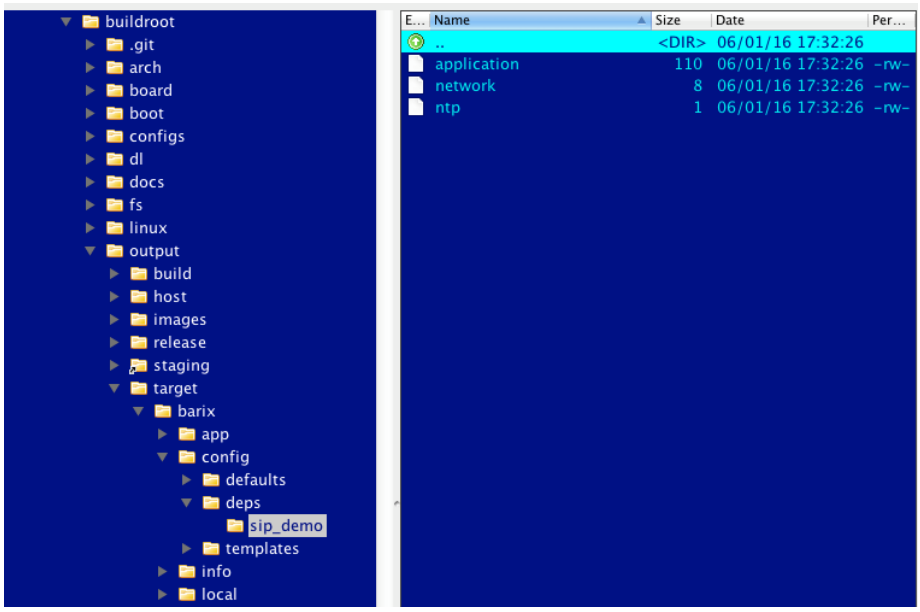
When the configuration parameters are changed e.g. by calling cfg_set_param and the configuration is saved by calling cfg_save, then the following sequence is executed:

1. *Parse all the UCI dependencies (see section 34) created for the given application, then a list of affected packages is created and stored in /var/run/.service_restart_list*
2. *UCI configuration is committed*
3. *Runlevel "C" is triggered by calling "init c"*
4. *The caller (e.g. CGI script calling save_cfg) returns*
5. *Init calls /etc/init.d/rcC, which restarts the services*
6. *The status of the services restarting can be polled by calling cfg_restarting_services*

The folder `/barix/config/templates/rc.d/` contains a symbolic link for each service to the appropriate init script in `/etc/init.d`. The same naming convention as in `/etc/rcN.d` is used (only "start" scripts); the numbering defines the order in which the affected services are restarted. The restart is executed by calling the respective init script with the "restart" parameter. If ypu like to have your application or service automatically restarted, you have to add it there, and the best way to do that is to include commands to install the corresponding symlinks in your project

6.6.4.5 UCI services dependency system

In some cases you may need to restart a specific service or application when you change a specific configuration option from the webUI. The UCI framework does that by checking the dependencies defined for the specific application in the /barix/config/deps folder. For example for the sip_demo application we have:



We see that we have dependencies for three services/applications: application, network, and ntp. These are just text files that contain the list of configuration options on the change of which the given service needs to restart. For example for the application service (which is the /etc/init.d/application service that is controlling the start/stop of our sip_demo application) we have:

```
application
# restart application on the followig uci settings change
network.eth0.proto
pjsua
simple_player
```

This means thaw the application service will restart on:

a) change of the protocol option for the network service
b) any change in the pjsua options
c) any change in the simple_player options

We can specify also a specific section, for example, if we replace “pjsua” with “pjsua.sip_account” which would mean: “Restart the application if any parameter in the sip_account section of pjsua changes.
To view all the pjsua parameters available in UCI we can just type from the serial terminal:

```
[root@barix ~]# uci show pjsua
pjsua.sip_account=section
pjsua.sip_account.reg_to=600
pjsua.sip_account.username=9245
pjsua.sip_account.registrar=sip99.barix.com
pjsua.sip_account.password=my_test_passwd

pjsua.aec=section
pjsua.aec.no_vad=y
pjsua.aec.ec_tail=250
```

```
pjsua.aec.ec_opt=speex
pjsua.misc=section
pjsua.misc.autoanswer=n
pjsua.misc.cmd_port=52221
pjsua.misc.capture_lat=200
pjsua.misc.playback_lat=200
pjsua.misc.quick_dial_num=change_me
pjsua.misc.dtmf_pattern=1234
[root@barix ~]#
```

6.6.4.6 Configuration files for system components

The device configuration parameters are stored in the UCI database, however most of the system components use their own configuration files and not UCI. Therefore a thin intermediate layer has been implemented in a form of shell script `/lib/config/gen_config.sh`.

This script can generate any text-based configuration file and be called from any of the init scripts in `/etc/init.d/` *using the following syntax:*

```
/lib/config/gen_config.sh <package>
```

this creates the configuration files for the given package. For example, the following call creates configuration files for the network subsystem.

```
 /lib/config/gen_config.sh network
```

For the successful configuration file generation a template must be present in `/barix/config/templates` *folder. It is a Bash file, which is included by the gen_config.sh and contains the following elements:*

Name	Type	Description
DST_FILE	array	Absolute path to the target generated configuration file.
TEMPLATE_FILE	array	Optional template file, which is prepended to the auto-generated configuration. Template files are located in */barix/config/templates/templates*
COMMENT_PREFIX	array	Character used to indicate comment. Typically hash “#”
DYNAMIC_CONTENT_FN	array	Function to create the dynamic content. Typically “create_dynamic_config”
create_dynamic_config	function	Shell function to print the dynamic content. This function reads the UCI configuration parameters and prints them in the appropriate format. The output is redirected to the target file

If only a single configuration file needs to be created the elements are initialized without index as in the

following example:

```
DST_FILE=/etc/ntp.conf
```

If multiple files are to be generated for each above element an array is created as:

```
DST_FILE[0]=
DST_FILE[1]=
```

and the corresponding functions for dynamic content are defined.

```
# Barix configuration interface
# (c) 2018 Barix AG
#
# meta-file for automatic config-file generation
# destination file for the configuration (absolute path)
DST_FILE=/etc/ntp.conf

# template file located in /barix/config/templates/templates (no template)
TEMPLATE_FILE=

# comments are prefixed with this character
COMMENT_PREFIX="#"

# function to create the dynamic content
DYNAMIC_CONTENT_FN=create_dynamic_config

# function to create dynamic content
function create_dynamic_config()
{
      # servers
      owner=`cfg_print_param ntp.source.owner`
      if [ "system" = "$owner" ]; then
          for nr in 1 2 3 ; do
            server=`cfg_print_param ntp.source.server$nr`
            if [ $server ] ; then  echo "server $server iburst" ; fi
          done
        elif [ "application" = "$owner" ]; then
            for nr in 1 2 3 ; do
                 server=`cfg_print_param ntp.source.server_app$nr`
                 if [ $server ] ; then  echo "server $server iburst" ; fi
            done
        fi
      # access restrictions
      echo "
```

```
# By default, exchange time with everybody, but don't allow configuration.
restrict -4 default kod notrap nomodify nopeer #noquery
restrict -6 default kod notrap nomodify nopeer #noquery

# Local users may interrogate the ntp server more closely.
restrict 127.0.0.1
restrict ::1
      "
}
```

More advanced example with multiple target configuration files:

```
# Barix configuration interface
# (c) 2012 Barix AG
#
# meta-file for automatic config-file generation

# -------- create /etc/network/interfaces

# destination file for the configuration (absolute path)
DST_FILE[0]=/etc/network/interfaces

# template file located in /barix/config/templates/templates
TEMPLATE_FILE[0]=network.interfaces

# comments are prefixed with this character
COMMENT_PREFIX[0]="#"

# function to create dynamic content
DYNAMIC_CONTENT_FN[0]=create_network_interfaces

# function to create dynamic content
function create_network_interfaces()
{
      if cfg_string_compare network.eth0.proto "dhcp" ; then
            # DHCP configuration, get all auto
            echo "iface eth0 inet dhcp"
            if cfg_string_compare network.eth0.dhcpname "" ; then
                  true
            else
                  echo -n "   hostname '"
                  cfg_print_param network.eth0.dhcpname
```

```
                echo -n "'"
            fi
      else
            # DHCP configuration, set all static
            ipaddr=`cfg_print_param network.eth0.ipaddr`
            netmask=`cfg_print_param network.eth0.netmask`
            gateway=`cfg_print_param network.eth0.gateway`

            echo "iface eth0 inet static"
            echo "      address $ipaddr"
            echo "      netmask $netmask"
            if [ -n "$gateway" ]; then
                  echo "      gateway $gateway"
            fi

      fi
}


# -------- create /etc/resolv.conf

# destination file for the configuration (absolute path)
DST_FILE[1]=/etc/resolv.conf

# template file located in /barix/config/templates/templates
TEMPLATE_FILE[1]=

# comments are prefixed with this character
COMMENT_PREFIX[1]="#"

# function to create dynamic content
DYNAMIC_CONTENT_FN[1]=create_resolv_conf

# function to create dynamic content
function create_resolv_conf()
{
      if cfg_string_compare network.eth0.proto "static" ; then
            dns1=`cfg_print_param network.eth0.dns1`
            dns2=`cfg_print_param network.eth0.dns2`
```

```
            if [ "X$dns1" != "X" ] ; then echo "nameserver $dns1" ; fi
            if [ "X$dns2" != "X" ] ; then echo "nameserver $dns2" ; fi
      fi

      # no action for DHCP
}


# -------- create /etc/sonicip.conf
# destination file for the configuration (absolute path)
DST_FILE[2]=/etc/sonicip.conf

# template file located in /barix/config/templates/templates
TEMPLATE_FILE[2]=

# comments are prefixed with this character
COMMENT_PREFIX[2]="#"

# function to create dynamic content
DYNAMIC_CONTENT_FN[2]=create_sonicip_conf

# function to create dynamic content
function create_sonicip_conf()
{
      sonic_vol=`cfg_print_param network.sonic_ip.volume`
      if [ "X$sonic_vol" != "X" ] ; then echo "SONICIP_VOLUME=$sonic_vol" ; fi
}
```

6.7 Web Interface

6.7.1 Functions

The web interface is the major user interface of Barix devices and the only interactive way to configure a Barix unit. It has the following functions:

- *display runtime device status*
- *provide a configuration interface*
- *allow control of the device over the network*
- *maintenance interface to: update firmware, reboot the unit and revert the settings to factory defaults*

6.7.2 Web interface components

6.7.2.1 Web server

The standard web server on Linux server systems is Apache. Due to its size, process-based architecture,

resource demands and speed it is not suitable for embedded systems. An event driven single process server seems to be more suitable for embedded systems. Therefore a lightweight alternative to Apache has been selected – Lighttpd.

6.7.2.2 CGI and dynamic page content

For CGI and dynamic page content a lightweight and simple scripting language is needed; the Haserl4 module for Lighttpd is used, together with Bash scripting.

6.7.2.3 Web Configuration

The webserver configuration is located in /etc/lighttpd. The following features are configured:

- *no virtual hosts*
- *HTTP only*
- *web server port is configurable, default 80*
- *web server running under user:group www-data:www-data*
- *maximum 2 simultaneous worker threads*
- *automatic indexes*
- *CGI executed via Haserl*
- *digest authentication support using /etc/lighttpd/.passwd file*

6.7.3 Web Folders

6.7.3.1 Web server folders

The web server uses the following folders:

Directory	Description
/etc/lighttpd	Web server configuration
/var/log	Log files
/var/run	Runtime temporary files
/usr/local/www	Web content

6.7.3.2 Web content structure

The web UI content is split in to a common system part and an application specific part. This way multiple applications can have their web UI, sharing a common structure and common system-specific pages. Each application's specific web UI files are stored in a dedicated directory. The symbolic link "current" points to the current application's web UI.

The following folder convention is used in the web content folder /usr/local/www:

[4] http://haserl.sourceforge.net/

Directory	Description
/application1	Web UI files for application 1
/application2	Web UI files for application 2, etc.
/current	Symbolic link pointing to the current application
/sys	Common system web UI files containing the frame-work, look and feel, styles, menu, common pages for status, reboot, update, factory defaults, etc. Linked with a symbolic link from each application folder
/template	Example application web UI folder

6.7.3.3 Application specific files

Within the application folder the following convention is used:

Name	Description
cgi-bin	Folder for CGI scripts and all dynamic pages
images	Folder for images
js	Folder for Java Script
sys	Symbolic link to the system “sys” folder
index.html	The index file - entry to the web UI

6.7.3.4 CGI scripts

Files containing dynamic content as well as CGI files to receive user actions with GET or POST are implemented using Bash scripting and Haserl. Haserl uses a similar syntax to PHP: the CGI file contains static content (HTML), if dynamic content – the Bash script – needs to be added, it is enclosed within <% and %> tags.

Overview of operation

See the Haserl documentation5 for a detailed description, what follows is an overview:

[5] http://haserl.sourceforge.net/

- *The environment is scanned for HTTP_COOKIE, which may have been set by the web server. If it exists, the parsed contents are placed in the local environment.*
- *Script parameters received via HTTP GET or POST are placed in the local environment.*
- *The script is tokenized, parsing haserl code blocks from raw text. Raw text is converted into "echo" statements, and then all tokens are sent to the sub-shell.*
- *Haserl forks and a sub-shell (typically /bin/sh) is started.*
- *All tokens are sent to the STDIN of the sub-shell, with a trailing exit command.*
- *When the sub-shell terminates, the haserl interpreter performs final cleanup and then terminates.*
- *The STDOUT of the script is sent raw to the web browser. Please note that the HTTP header must be sent for proper operation.*

6.7.3.5 CGI functions

Several Bash functions are provided to ease working with device configuration and status, they are stored in /usr/local/lib/cgi.

Naturally, the configuration functions described above can be used as well.

Do not forget to include the respective shell file to use the function.

Function	Parameters	File	Description
form_print_radio	1. configuration parameter full name 2. true label 3. false label	*config.sh*	Assumes a boolean configuration parameter. Prints 2 radio buttons, one for the true value, one for the false value and selects the currently selected value. The corresponding labels are provided as function parameters 2 and 3.
print_http_hdr	1. optional content type	*generic.sh*	Typically call this function at the beginning of your script. Prints HTTP with content type. Unless specified the *text/html* content type is used.
print_datetime	None	*status.sh*	Print device's date and time
print_hw_type_id	None	*status.sh*	Print device's hardware ID as integer
print_device_type	None	*status.sh*	Print device's hardware type as text
print_module_type_id	None	*status.sh*	Print device's IPAM module ID as integer
print_ipam_type	None	*status.sh*	Print device's IPAM module type as text

Function	Parameters	File	Description
print_fw_version	None	*status.sh*	Print application version
print_kernel_version	None	*status.sh*	Print kernel version
print_uptime	None	*status.sh*	Print system uptime in a nice form
print_mac_addr	None	*status.sh*	Print device's MAC address
print_ip_addr	1. optional interface	*status.sh*	Print current device IP address. By default refers to eth0 status. Optionally the interface name can be provided as a parameter.
print_netmask	1. optional interface	*status.sh*	Print current device netmask By default refers to eth0 status. Optionally the interface name can be provided as a parameter.
print_default_gw	1. optional interface	*status.sh*	Print current device default gateway By default refers to eth0 status. Optionally the interface name can be provided as a parameter.
print_dhcp_name	None	*status.sh*	Print device DHCP name
print_dns_servers	None	*status.sh*	Print current DNS servers, one per line
print_mount_table	1. parameters to the <*table*> element 2. “no devices” text	*status.sh*	Prints an HTML table with removable storage device information. The output is similar to “du -sh”. The 1 parameter is printed as options to the <*table*> element. The second parameter is printed as a message if no media is found (typically use “no media found”)

Function	Parameters	File	Description
show_device_config	None	*status.sh*	List the current device configuration.

CGI variables – HTML forms

HTML forms sent to the device via GET or POST method are captured by Haserl and the keys and values are stored in the Bash script environment variables as:

Variable	Description
FORM_key=value	All form variables
GET_key=value	Keys sent via the GET method
POST_key=value	Keys sent via the POST method

7 Miscellaneous

In this section we will mention some tips and tricks that will make more easy the development with the IPAM-400 SDK

7.1 Connecting serial terminal

Using the serial terminal is an important part of the Linux development process, especially for viewing the boot-up messages and doing some debugging on the device if the network is not properly set up, or broken.

IPAM-400 serial is connected to J7. Since they are 3.3V compatible, make sure that you use the correct USB to serial adapter. The connections are shown in the photo and table below:

J7 Pin No	*Wire Colour*	*Function*	*Connection*
1	*White*	*Tx*	*Connected*
2	*Green*	*Rx*	*Connected*
3	*Black*	*Gnd*	*Connected*
-	*Red*	*3.3V*	*Not used*

NOTE 1: *The latest IPAM-400 modules do come from the factory without the 3 pin header soldered. You may need to solder it yourself.*

NOTE 2: *Never use/connect serial interfaces with 5V or 12V, because this may damage the IPAM-400 module!*

NOTE 3: *In order to powercycle the device, you need also to disconnect temporary the serial interface.*

Next, after booting the device, and connecting your USB-to-serial adapter to the device, you will need to open a serial terminal on your development PC using a standard terminal emulator with the following parameters: 115200,8,0,N.

You can use the following emulators:

- *Linux: minicom, CoolTerm, GTK Term*
- *MacOS: Zterm, minicom*
- *Windows: Putty*

For example, to open the terminal on mac using minicom

```
MacBook-Pro-2:~ asi$ minicom -D /dev/cu.usbserial
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on May 17 2017, 04:52:30.
Port /dev/cu.usbserial, 13:48:33
```

```
Press CTRL-A Z for help on special keys
```

Once you see the minicom greeting screen above, then press Ctrl+Z,A,O to enter in the minicom configuration menu, and set the serial port communixation parameters accordingly

***NOTE 1:** Use the correct device name for your own controller. You can find it on MAC with this command:*

```
MacBook-Pro-2:~ asi$ ls -la /dev/tty.*
crw-rw-rw-  1 root  wheel   17,   0 Apr 18 08:31 /dev/tty.Bluetooth-Incoming-Port
crw-rw-rw-  1 root  wheel   17,   2 Apr 18 08:31 /dev/tty.W810i-SerialPort
crw-rw-rw-  1 root  wheel   17,   4 Apr 18 14:10 /dev/tty.usbserial
MacBook-Pro-2:~ asi$
```

On Linux system the name of the serial port will be most likely /dev/ttyUSB0

7.2 Useful Yocto commands

7.2.1 Recompiling specific package

Sometimes is necessary to test a small change in the code, and recompile the package, without the need to go through the same whole process of committing the chages to the repo, fetching the new one on jocto, and recompiling again. Instead, just modify the file you need directly in the build folder of the package, then use the following command:

```
bitbake -f -c compile pjsua (use your own package name here)
```

7.2.2 Cleaning a package

To clean the working folder of a package, use the following command

```
bitbake -c cleanall -f pjsua (use your own package name here)
```

Be aware that his will clean everything so the sources will be fetche either from a GIT repo (for git recipes), or decompressed from the tarball (for local recipes), so take care not to lose any modifications you have done

7.2.3 Generate compiling tool chain

Sometimes it might be useful to generate a toolchain with which you can develop externally your package, without the hastle to use the whole infrastructure needed by Yocto. Use the command:

```
bitbake -c populate_sdk core-image-barix-sdk
```

The result will be a selfextracting script file in the `build/tmp-glibc/deploy/sdk/` *folder, containing the cross-toolchain and all the needed libraries to compile your application without all the hastle of using the complete Yocto infrastructure.*

```
work/oe-core/build$ ls tmp-glibc/deploy/sdk/
oecore-i686-cortexa7hf-neon-vfpv4-toolchain-nodistro.0.host.manifest
oecore-i686-cortexa7hf-neon-vfpv4-toolchain-nodistro.0.sh
oecore-i686-cortexa7hf-neon-vfpv4-toolchain-nodistro.0.target.manifest
oecore-i686-cortexa7hf-neon-vfpv4-toolchain-nodistro.0.testdata.json
~/work/oe-core/build$
```

You can develop and compile on a standard PC, and just deploy manually to the target device the compiled binary. Once the development phase is completed, you can then spend time to add the corresponding

recipes in Yocto.

To use the generated Yocto SDK, just run the script, and answer to the question where you would like to have it installed. In the example below we have opted for the home folder, since this would not require root access:

```
$ ./oecore-i686-cortexa7hf-neon-vfpv4-toolchain-nodistro.0.sh
OpenEmbedded SDK installer version nodistro.0
==========================================
Enter target directory for SDK (default: /usr/local/oecore-i686): ~/oecore-i686
You are about to install the SDK to "/home/test_oem/oecore-i686". Proceed[Y/n]?
Y
Extracting SDK.....................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the
environment setup script e.g.
 $ . /home/test_oem/oecore-i686/environment-setup-cortexa7hf-neon-vfpv4-oe-
linux-gnueabi
```

Next, when you want to develop, open a terminal, source the specified file above, and use the standard `make` *as you usually do.*

7.3 Development Environment Credentials

7.3.1 Device SSH credentials

The device default credentials are:

```
User: root
Password: oem_devkit_17
```

You can use these to login to the device either via the serial interface, or SSH:

```
macmini:~ asi$ ssh root@192.168.11.159
root@192.168.11.159's password:
[root@barix ~]#
```

7.3.1.1 Changing the device password

To change the password on the device, just type the "passwd" command in the serial or SSH terminal after login. Type the new password two times to confirm.

Please have in mind that the password will be reset to the default one in the rootfs image after a FW update.

7.3.1.2 Changing the default root password in the build

To change the default password in the build you will need to open the recipe for the image. Open `recipes-core/images/core-image-barix-sdk.bb` *from the meta-barix-sdk layer folder in some text editor, and find the following line:*

```
EXTRA_USERS_PARAMS_append="\
      usermod -P oem_devkit_17 root; \
"
```

and change the oem_devkit_17 password to your preference, ex:

```
EXTRA_USERS_PARAMS_append="\
      usermod -P my_new_password root; \
"
```

and recompile again the image running the command from the Yocto build folder:

```
bitbake core-image-barix-sdk
```

7.3.2 Bitbucket credentials

You can fetch the distribution of the OEM Development Kit from BitBucket using these credentials:

Email: ipam400-oem@barix.com

User: ipam400-oem

Password: eCn-U3o-tXg-4rF

For now it is a common login for all OEM customers, but in the future it will be converted to a group, and every OEM customer will get his own login ID and password.

7.3.3 Adding package manager to the generated image

In some cases it might be useful to have a package manager preinstalled on the device, so that it can be possible to install a package with all its dependencies with a single command. In this way, the Yocto environment could be configured to compile as many packages as desired (and that may not be included in the provided image), which can be exported to a web server, from wfich the devices can easily fetch them.

To do that, we need to do the following three subtasks:

7.3.3.1 Configuring the Yocto environment to include the OPKG manager

To add support for package management, you need to edit meta-barix-sdk/recipes-core/images/core-image-barix-sdk.bb file, and add the package management dependencies, marked with red below:

```
    IMAGE_INSTALL= "\
         ${CORE_QIBA_IMAGE_BASE_INSTALL} \
         ${IMAGE_INSTALL_BARIX_COMMON} \
         ${IMAGE_INSTALL_BARIX_TOOLS} \
         lighttpd \
         ntp \
         pjsua \
         sip-demo-web-ui \
         opkg \

    EXTRA_IMAGE_FEATURES= "\
         debug-tweaks \
         package-management \
```

Next, rebuild your image as specified in section 9:

```
bitbake core-image-barix-sdk
```

When the build finishes you will have an image, containing the opkg package manager, and the package feeds generated `in build/ tmp-glibc/deploy/ipk/` folder:

```
~/work/oe-core/build$ ls -la  tmp-glibc/deploy/ipk/
total 552
drwxr-xr-x 6 test_oem test_oem   4096 May 28 16:29 .
drwxr-xr-x 6 test_oem test_oem   4096 May 28 16:30 ..
drwxr-xr-x 2 test_oem test_oem   4096 May 28 16:29 all
drwxr-xr-x 2 test_oem test_oem   4096 May 28 16:29 barix_ipam400
drwxr-xr-x 2 test_oem test_oem 376832 May 28 16:29 cortexa7hf-neon-vfpv4
drwxr-xr-x 2 test_oem test_oem 163840 May 28 16:29 i686-nativesdk
-rw-r--r-- 1 test_oem test_oem      0 Apr 23 15:27 Packages
```

7.3.3.2 Setting a server with the generated package feeds

The next step is to export all the folders, listed above, to your web server folder. This could be done either by manually copyinf them, ar by just creating a symlink, pointing to the location where your Yocto scripts are generating them. Note that setting up a web server is out of the scope of this document. Please check the corresponding manuals wor the web server you plan to use (ex. Apache, Lighttpd, buildt in MacOS HTTP server, etc)

7.3.3.3 Configuring the device to use the package feeds

The last step is to configure the device so that ther local opkg manager so tha it nows where to fetch the package feeds from. So, login to the device via serial interface or SSH, and create the `/etc/opkg.conf` file with the following contents:

```
src all http:// your.server.com/ipam400/repository/ipk/all/
src barix_ipam400 http:// your.server.com /ipam400/repository/ipk/barix_ipam400/
src cortexa7hf-neon-vfpv4
http://your.server.com/ipam400/repository/ipk/cortexa7hf-neon-vfpv4/


# Default destination for installed packages
dest root /
option lists_dir /var/lib/opkg/lists
```

Please note the following:

- *The name of the package feeds mut be tha same as the directory names listed above*
- *You have to replace the `http://your.server.com` with the real name or IP address of your web server*

Now it should be possible to install any package just by typin for example:

```
opkg install python
```

7.4 Listing all factory defaults

As explained in previous sections, the factory defaults are defined in the UCI defaults for every package. If you want to have a full list of parameters that will be reset when you restore to factory defaults, then you have two options:

7.4.1 Checking all defaults files in the /barix/config/defaults/ folder of the device.

Login to the device via the serial or SSH console, and type:

```
[root@barix ~]# cd /barix/config/defaults/
[root@barix defaults]# ls -la
total 44
drwxr-xr-x    2 root     root           904 Oct 13 08:57 ./
drwxr-xr-x    5 root     root           432 Oct 13 08:57 ../
-rw-r--r--    1 root     root           247 Oct 13 08:57 application
-rw-r--r--    1 root     root           610 Oct 13 08:37 dropbear
-rw-r--r--    1 root     root            63 Oct 13 08:52 httpd
-rw-r--r--    1 root     root           236 Oct 13 08:52 network
-rw-r--r--    1 root     root           193 Oct 13 08:48 ntp
-rw-r--r--    1 root     root           462 Oct 13 08:57 pjsua
-rw-r--r--    1 root     root            60 Oct 13 08:52 rtc
-rw-r--r--    1 root     root           162 Oct 13 08:52 security
-rw-r--r--    1 root     root           243 Oct 13 08:57 simple_player
-rw-r--r--    1 root     root            62 Oct 13 08:36 syslogd
-rw-r--r--    1 root     root            77 Oct 13 08:52 timezone
[root@barix defaults]# cat /barix/config/defaults/*
package 'application'

config section 'main_config'
      option active_app 'pjsua'

config section 'audio'
      option amplifier  'on'
      option mic_linein 'line'
      option volume           '50'
      option mic_gain         '0dB'
      option mic_boost  'on'
      option ad_gain          '0dB'

package 'dropbear'

config section 'SSH'
      option Port '22'
      option RootLogin  '1'
      option RootPasswdAuth   '1'
      option DisablePasswdLogins    '0'
```

```
	option SSHKeepAlive	'300'
	option IdleTimeout	'0'
	option WindowBuffer	'24576'
	option KeepAlive  '0'
	option DisableLocalPortFwd    '1'
	option DisableRemotePortFwd   '1'
	option AllowRemoteHosts '0'

……

package 'network'

config interface 'eth0'
	option proto 'dhcp'
	option ipaddr '192.168.1.100'
	option netmask '255.255.255.0'
	option gateway '192.168.1.1'

config sonic_ip 'sonic_ip'
	option enabled 'true'
	  option volume '50%'
package 'ntp'

……

package 'syslogd'

config section 'remote'
	option ipaddr ''

package 'timezone'

config section 'timezone'
	option description 'UTC*UTC'

[root@barix defaults]#
```

7.4.2 Listing all defaults with UCI command

While the first method is easy and straightforward, the information is shown in not so optimal way. To have a more synthesized information you can do the following:

1) *Reset to factory defaults from the web UI by going to DEFAULTS → Reset Factory Defaults.*

2) *Login to the device (either via serial or SSH console) and type:*

```
[root@barix ~]# uci show
application.main_config=section
application.main_config.active_app=pjsua
application.audio=section
application.audio.amplifier=on
application.audio.mic_linein=line
application.audio.volume=50
application.audio.mic_gain=0dB
application.audio.mic_boost=on
application.audio.ad_gain=0dB
dropbear.SSH=section
dropbear.SSH.Port=22
dropbear.SSH.RootLogin=1
dropbear.SSH.RootPasswdAuth=1
dropbear.SSH.DisablePasswdLogins=0
dropbear.SSH.SSHKeepAlive=300
dropbear.SSH.IdleTimeout=0
dropbear.SSH.WindowBuffer=24576
dropbear.SSH.KeepAlive=0
dropbear.SSH.DisableLocalPortFwd=1
dropbear.SSH.DisableRemotePortFwd=1
dropbear.SSH.AllowRemoteHosts=0
dropbear.FilePaths=section
dropbear.FilePaths.dropbear_folder=/barix/local/app-data/dropbear
dropbear.FilePaths.rsakey=dropbear_rsa_host_key
dropbear.FilePaths.dsskey=dropbear_dsa_host_key
dropbear.FilePaths.banner=banner
dropbear.FilePaths.PID=/var/run/dropbear.pid
dropbear.RunCtl=section
dropbear.RunCtl.enable=1
httpd.webserver=section
httpd.webserver.port=80
network.eth0=interface
network.eth0.proto=dhcp
network.eth0.ipaddr=192.168.1.100
network.eth0.netmask=255.255.255.0
network.eth0.gateway=192.168.1.1
network.sonic_ip=sonic_ip
```

```
network.sonic_ip.enabled=true
network.sonic_ip.volume=50%
ntp.source=section
ntp.source.owner=system
ntp.source.server1=1.barix.pool.ntp.org
ntp.source.server2=2.barix.pool.ntp.org
ntp.source.server3=3.barix.pool.ntp.org
pjsua.sip_account=section
pjsua.sip_account.registrar=change_me.some_sip_server.com
pjsua.sip_account.username=change_me
pjsua.sip_account.password=change_me
pjsua.sip_account.reg_to=600
pjsua.aec=section
pjsua.aec.no_vad=y
pjsua.aec.ec_tail=250
pjsua.aec.ec_opt=disabled
pjsua.misc=section
pjsua.misc.autoanswer=n
pjsua.misc.cmd_port=52221
pjsua.misc.capture_lat=100
pjsua.misc.playback_lat=100
pjsua.misc.quick_dial_num=change_me
pjsua.misc.dtmf_pattern=1234
rtc.rtc=section
rtc.rtc.enabled=true
security.reset=reset
security.reset.enabled=true
security.defaults=defaults
security.defaults.enabled=true
security.update=update
security.update.enabled=true
simple_player.common=section
simple_player.common.media_type=stream
simple_player.streaming=section
simple_player.streaming.url=http://www.barix.com/radio.m3u
simple_player.streaming.buffer_ms=300
simple_player.files=section
simple_player.files.media_folder=simple_player
syslogd.remote=section
timezone.timezone=section
```

```
timezone.timezone.description=UTC*UTC
[root@barix ~]#
```

In this way you have a quick and condensed overview of all configuration parameters, and their defaults.

8 Tips, Known Issues and Work in Progress

As of the date of writing this manual, the following issues are known:

8.1 “Relay” control via the RTS pin of the serial port

In IPAM-390 Dev kit it was possible to use the RTS pin of the serial port to control a LED or to drive an external relay. This feature is available also on the IPAM-400 SDK starting from v1.02

8.2 SIP Rebroadcast application

SIP Rebroadcast application hasn’t been ported yet from the IPAM-390 Development Kit to the new IPAM-400 SDK. For this reason, the selection of the active application on the SETTINGS page allows switching only between the SIP client, and the Simple player applications.

8.3 Yocto generated external toolchain

The toolchain that is generated by Yocto to enable the developers develop their own applications externally, without the need to use the Yocto environment, currently does not contain the Barix proprietary libraries-utility_lib and player_lib, so the developers will not be able to link against them externally.

There is a work in progress preparing the recipes and the scripts these libraries to be included automatically when the toolchain for the Barix SDK image is generated.

As a work around, the precompiled object code, and the needed include files can be copied from the Yocto build folder to the place where the Yocto SDK tarball is being decompressed

For any other issues and questions please contact Barix Customer Support. All suggestions for improving this manual are welcome.

9 Links, References and Used Document Sources

No	*Document Title*	*Document file name*	*Document Location*	*Author*
1	*Boot and Update strategy ARM platform*	*DevWorkEnvironment_IPAM390_v0.2.odt*	*Conlfuence*	*JR, AB*
2	*Combined HW/SW Architecture-Concept*	*Combined Hardware Software Architecture.odt*	*IPAM-390 Doc repo on ford*	*PK*
3	*Service Restart Dependencies*	*Restart_deps.odt*	*IPAM-390 Doc repo on ford*	*PK*
4	*New Platform User Space*	*New platform user space.odt*	*IPAM-390 Doc repo on ford*	*PK*

10 Legal Information

Barix AG
Ringstrasse 15a
8600 Dübendorf

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**