AR8x application note

Getting started: creating simple application zip for ar8x web configuration

1. Scope

AR8x is equipped with a web configuration interface which has also a feature to install additional software to the platform. The applications need to packed to a zip-file as described in this documentation.

1. Application zip file contents

At minimum, the application zip-file contents should atleast be as described below:

/bin/

/bin/run

/properties/

/properties/manifest

* 1. RUN SCRIPT

The bin-folder contains bash-script named “run”, which is executed when the application is to be started or stopped from the web UI. The bin-folder may also contain any other binaries required for executing your application. These binaries however need to started from the run-script. The script itself will be executed in the background and any standard output is forwarded to /var/log/<applicationame>.log. The contents of the script should be as follows:

#!/bin/sh

if [ "$1" = "start" ]; then

# Signal start to your process(es)

echo "started"

else

# Signal stop to your process(es)

echo "exit"

fi

The above example is the minimal content for the script i.e. place everything you need for executing/signaling your process to start to the “start”-block of the if-else-clause, and everything required to signal your process to stop to the “stop”-block. When actually executing something from the script, you will need to have the full path to the executable, for example when running the example Ping-webservice example with the script, the contents should be:

mono /mnt/userdata/apps/appexample/bin/ThirdpartyBackendExtension.exe

or

mono ~/ThirdpartyBackendExtension.exe

The home folder for each "application"-user is /mnt/userdata/apps/<applicationname>/bin). So the full content of the run-script would look something like this:

#!/bin/sh

if [ "$1" = "start" ];

then

# Signal start to your process(es)

mono ~/ThirdpartyBackendExtension.exe

else

# Signal stop to your process(es)

echo "exit"

fi

And the bin-folder would then also contain the “ThirdPartyBackendExtension.exe”-file.

If your application needs to be automatically restarted if it exits(or crashes), you should add its execution in to a loop to the script. Or optionally start another script file which runs this. For example:

**/mnt/userdata/apps/<applicationname>/bin/run**

#!/bin/sh

if [ "$1" = "start" ];

then

# Signal start to your process(es). Here we start it to the background

~/appscript.sh &

else

# Signal stop to your process(es)

echo "exit"

fi

**/mnt/userdata/apps/<applicationname>/bin/appscript.sh**

#!/bin/sh

while :

do

# Execute your process

echo “Hello from appscript.sh”

# Sleep for 1 second when the process exits before starting it again

sleep 1

done

* 1. MANIFEST

In addition to the script found from the bin-folder, another mandatory item to be found is the manifest-file located in the properties-folder. The file should contain a keyvaluepair for each setting as shown below:

header=ar8x\_application

name=application\_name

version=1.0.0

autostart=false

The line containing the header-key is used to recognize the contents of the manifest and the line with the name-key is used as the name of the application to be installed. Note that the value for the name-key is also used as the “user” in the system when starting the process and other related. The version-key is used for the version number of your software and only used for informational purposes. The last line containing autostart-key is used to define if the application is to be started when the CoreService is started. If this setting is set to true, the CoreService will execute the “run”-script of your application package when it gets started. If for some reason the CoreService requires a restart, it will also check whether your application is already running, and will not start it if determines that it already is. Note that changing this setting does not start the application automatically, but instead it will started on next system restart and/or Coreservice restart.

Keep in mind that the above is the minimum required content for the file, and additional settings can be stored in it as well if the application sandbox gets extended in the future.

1. APP WEB FRONTEND

The application package can also contain a web frontend for the application. It can be used just for informational purposes, as also for configuring your application if you require a user interface for it. On the previous chapter we described the minimum content for the zip-file. To add also the web frontend to the package the zip-file should contain a file structure similar to the below example:

/bin/

/bin/run

/properties/

/properties/manifest

**/frontend/**

**/frontend/index.htm**

The frontend-folder is used for storing any web-UI related parts of your solution, for example if you wish to configure the application through the AR8x-web configuration. The CoreService on AR8x will check whether this folder and the index.htm file exists in the zip-file and if they do, it will place an alias for nginx-configuration so that requests to https://ar8x\_address/app/frontend/<applicationame> will get their response from /mnt/userdata/apps/<applicationname>/frontend

The index.htm should then contain atleast:

<!--#include file="$ar8xroot/app\_header.htm" -->

<body>

</body>

<!--#include file="$ar8xroot/app\_footer.htm" -->

This is due to the reason that the content is opened in to an iframe via the web-ui and the parent window waits for this iframe to finish "loading" i.e. the app\_footer.htm calls a script which writes a simple div-element to the body of the frame to inform the the parent windows that it has completed loading. The nginx-configuration adds server side includes for the application specific “frontend”, so including the mentioned files is possible for your frontend as well. Optionally you can skip both app\_header.htm and app\_footer.htm and simply place <div style="display:none" id="doneID">Application title</div> within the body-tags in your index.htm if you do not see any use for the scripts included in the app\_header.htm.

You can find an example of the web frontend from the application sample package: appimportsample\zipcontents\frontend

If the application web frontend is to be used for configuring your application, you’ll need to enable the backend for it too. This is explained with more details in the next chapter. See also chapter 4.1 for an example.

1. APP BACKEND

Unless the application frontend is meant to be just informational, i.e. display some versioning info of the application in question, you’ll need to add a “backend” for the web config too. The backend itself can just an addition to your currently running application which listens for the RESTful requests, or other similar, but since the platform does not allow any kind of configuration, the communication will require some settings.

So to achieve proper configuration for the backend, the zip-file should be at minimum as follows:

/bin/

/bin/run

/properties/

/properties/manifest

**/backend/**

**/backend/configuration**

The configuration-file contains the port to used for forwarding the communication and the contents of the file should be similar to this:

header=ar8x\_backend

port=10001

Again, the header-key is to identify the file and port-key contains the port number. Valid port numbers for this configuration are from 10000 to 11000. If the backend service determines that the file and the port is valid, it will configure nginx so that any communication to https://ar8x\_address/app/backend/<applicationame> is forwarded to http://127.0.0.1:<portnumber>. Then any webrequest(or any other approach used) should communicate with the application in question using [https://ar8x\_address/app/backend/<applicationame](https://ar8x_address/app/backend/%3capplicationame)>

* 1. EXAMPLE

On the sources of ThirdPartyAppPing\_src-project(found from the sample package), you should be able to find these lines. In simplified terms, the ThirdPartAppPing waits for request coming to http://127.0.0.1/app/backend/appexample path, and the lines below introduce the Ping-webservice method for the path. This can be then called from clients. The example package uses jquery to request it.

[OperationContract,

WebInvoke(Method = WebRequestMethods.Http.Get,

UriTemplate = "/Ping",

RequestFormat = WebMessageFormat.Json, ResponseFormat = WebMessageFormat.Json)

]

String Ping();

public String Ping()

{

Console.WriteLine("Ponging");

return "Pong";

}

So if our “ThirdpartyBackendExtension.exe” found in **AppImportSample** listens to port 10001 for JSON formatted HTTP requests and it is running after being successfully installed, we can send a simple HTTP GET request to it like this:

<script type="text/javascript">

function Ping() {

var fullurl = 'https://ar8x**567890**.local/app/backend/appexample/Ping';

$.ajax({

url: fullurl,

type: "GET",

cache: false,

processData: false,

success: function (data) {

alert(data); // Pong

},

error: function (data) {

alert("backend service connection error " + data);

}

});

}

</script>

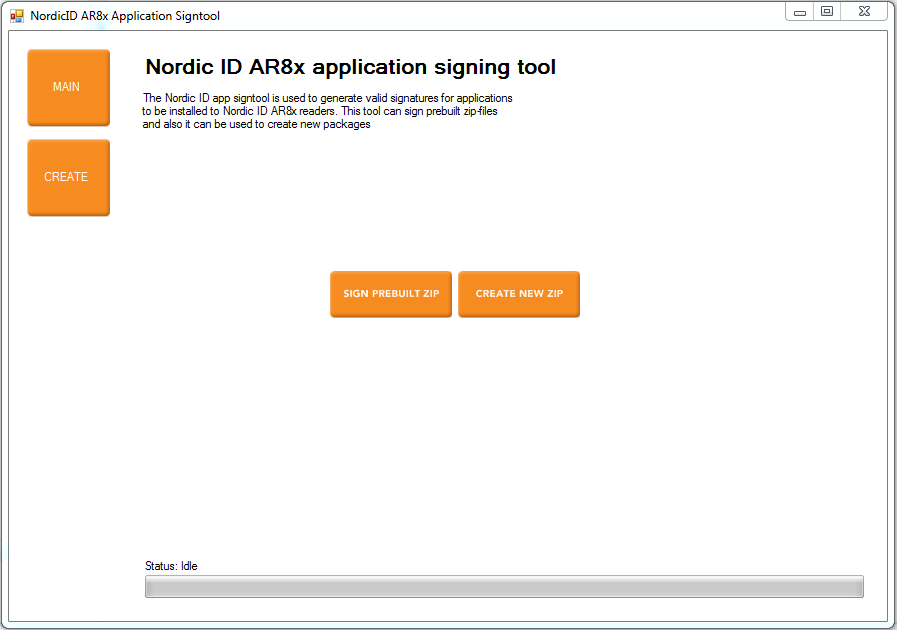
1. SIGNING

To provide more security to the platform, the application zip-files need to be signed with Nordic ID provided signing tool. The public key generated to the zip-file will be then verified against the list of files when installing the zip-file to your AR8x device. This makes sure that only valid content from the zip-file can be installed.

The tool in question is called Nordic ID AR8x Application Signtool. The tool can be used to sign prebuilt zip-files, as also to create new zip-files from scratch. We will go through shortly both approaches in this document. If you have not yet installed the tool, please download it from <http://www.nordicid.com/en/downloads/> and install it.

* 1. SIGNING PRE-BUILT ZIP-FILES

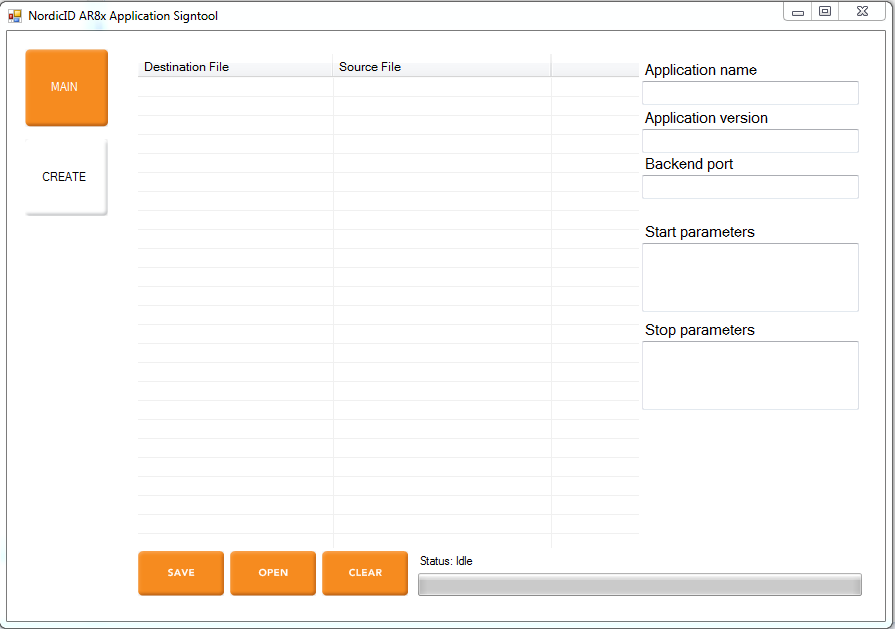
When opening the Nordic ID AR8x Application Signtool, you’ll have 2 buttons on the main page. By clicking the “Sign prebuilt zip” opens up a dialog where you can enter your zip-file which you have generated. After clicking OK, the tool will sign the file and place the publickey and the list of files in to the zip-file and save it as <zipname>-signed.zip



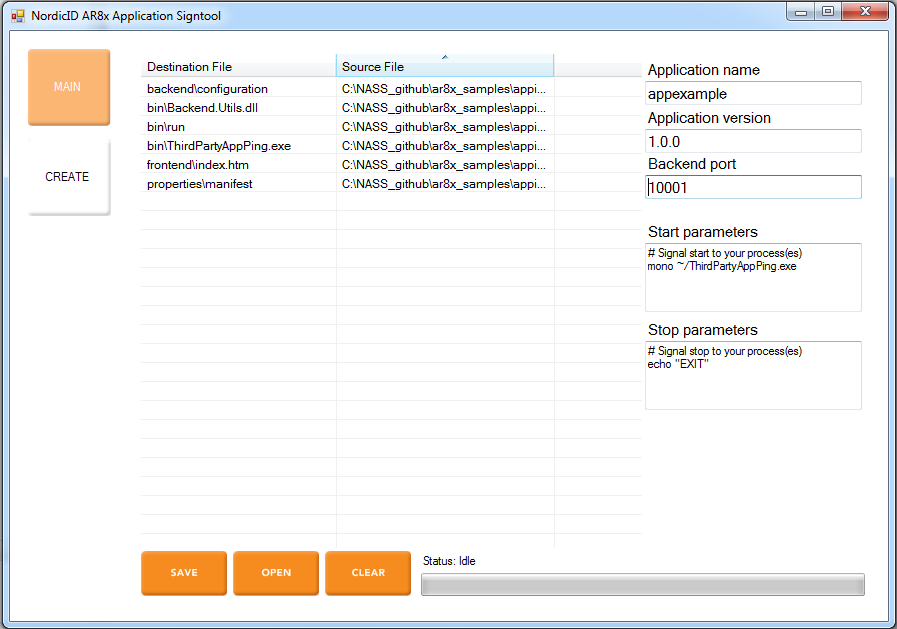
When the zip-file has been signed, the tool will display the above i.e. Status will be set back to idle. At this point you should be able to install the zip-file in question to your AR8x device using the Web UI.

* 1. CREATING NEW APPLICATION ZIP-FILE

By clicking the “Create new zip” or the “Create”-button from the main-view, you’ll be forwarded to the second view of the tool.



On this view you can either manually enter each field and add your application files yourself. The application name/version and the backend port will be filled automatically if you add or drag-and-drop the files which contain these settings. For example in the picture below, we have dragged the “zipcontents”-folder from the “Appimportsample”-package to the the listview on the signtool, and the tool has then filled each field automatically based on the contents.



Since the sample contains already valid material, we can click “Save” at this step. The tool will request you for a name and a location for the zip-file to be saved at. After clicking OK the file will be generated and saved.

1. OTHER INFO
   1. ADDITIONAL LIBRARIES

If your application requires addiditional libraries included, you can add them to the zip-file under /lib/-folder. When the CoreService starts the application, it will add the /lib/-folder to LD\_LIBRARY\_PATH before executing the run-script for the app. So if all possible features and libraries are added to the zip-file, the folder-structure along the mandatory files would look like this:

/bin/

/bin/run

/properties/

/properties/manifest

/frontend/

/frontend/index.htm

/backend/

/backend/configuration

**/lib/**

* 1. INSTALLED RUNTIMEs

The platform has Python and Mono preinstalled. However any runtime should be possible to install within an application sandbox zip-file.

* + 1. ADDITIONAL RUNTIMES

To install for example Java, you can either split the files so that they are stored in the /lib/ -folder of you app zip or just place them directly to the /bin/-folder. The only thing you need to take in consideration is that the execution path for in the run-script should include the possible location of your Java VM. For example in the Java-sample <https://github.com/NordicID/ar8x_samples/tree/master/javasample> , the VM binaries are stored in the /bin/-folder and when the javasample\_ar8x-class is started, the run-script will execute the following command:

~/java/bin/java -Djava.library.path=../lib -cp ../lib/NurApi.jar:../lib/NurApiSerialTransportLinux.jar:../lib/org.eclipse.paho.client.mqttv3-1.1.0.jar:. javasample\_ar8x

Which would the same as this:

/mnt/userdata/apps/javasample/bin/java/bin/java –Djava.library.path=/mnt/userdata/apps/javasample/lib –cp **/mnt/userdata**/apps/javasample/lib/NurApi.jar: /mnt/userdata/apps/javasample/lib/ NurApiSerialTransportLinux.jar:/mnt/userdata/apps/javasample/lib/org.eclipse.paho.client.mqttv3-1.1.0.jar: /mnt/userdata/apps/javasample/bin javasample\_ar8x

Note that when starting applications with any VM or other additional binaries, they are always started with a apps-group user i.e. it will not have any root-access.

* 1. persist memory

As mentioned the home-folder for each installed application is set to /mnt/userdata/apps/<applicationname>/bin. The /mnt/userdata/ is part of the persist memory i.e. anything stored there will be untouched during system restarts.

So if you have certain files which you don’t want to lose during reboot, such as databases or log files, you should keep the data in the applications home-folder in /mnt/userdata/apps/<applicationname>/bin or under /mnt/userdata/apps/<applicationname>/.

* 1. Nurapi

The internal Nordic ID UHF reader module can be connected from the user-applications installed to the platform. The module is connected via the internal USB, but should not be connected directly through it from user applications. This is due to the reason that the platform controls certain parts of the NUR-related HW and therefore your applications should always open a TCP socket to the NurSvc-service running on the device. By default NurSvc listens at port 4333 at localhost. The NurSvc takes care of the communication to the module and reroutes any possible HW related communications to/from the OS. You can find more information about the NurSvc from **AN006\_AR8x\_NurSvc.docx**

For using the NurApi, please refer to its documentation.

1. OTHER

See “**ar8x\_samples\AppImportSample**” for a detailed example of how to create a validly formatted zip-file.