

The OpenR2 Guide – January 2009 - Version 0.1

Moisés Silva <moy@sangoma.com>

Alexandre Alencar <alexandre.alencar@gmail.com>

<http://www.libopenr2.org/>

<http://code.google.com/p/openr2/>

1. About this guide.

OpenR2 is a library that implements the MFC/R2 signaling over E1 lines using the Zapata Telephony Interface, DAHDI and in the near future the OpenZAP TDM abstraction library. This document DOES NOT describe the programming interfaces provided by OpenR2, please refer to the code for that :-), and don't hesitate on asking me any question you may have.

This document describes the installation of the library to be used by Asterisk and/or FreeSwitch.

For questions regarding installation or deployment issues, **please don't contact me directly**, unless you are willing to pay me to give you support :-). Instead, use the Digium's asterisk-r2 mailing list at: <http://lists.digium.com/mailman/listinfo/asterisk-r2>, or join IRC in [irc.freenode.org](irc:freenode.org) in the #openr2 channel. I usually hang out there with the nick 'moy'.

If there is an MFC/R2 variant that is not supported and you need (or is not working as expected), send me an e-mail. I also will appreciate any contribution or correction to this document and/or the library code itself.

2. What is MFC/R2?

MFC/R2 is a telephony signaling widely used in Mexico, Brazil and other countries in Latin America and Asia. MFC/R2 stands for Multi Frequency Compelled R2. Compared to more recent signaling protocols like ISDN PRI/BRI or SS7, R2 offers a limited set of functionality. The signaling is only used to setup the call and bring it down. Some MFC/R2 variants may send billing pulses during the call though, but those are rarely used.

There is analog and digital MFC/R2 versions, any reference to MFC/R2 or R2 in this document refers to the digital version that uses E1 facilities. Most readers will not likely be interested in the analog version because is not widely used anymore. MFC/R2 is defined by the ITU, however most countries using MFC/R2 do not follow the ITU specification and implement their own national variant.

3. How MFC/R2 works?

Whether you are a programmer that is going to use the library or system's administrator that is going to install it along with a telephony engine (ie Asterisk or FreeSwitch), it is a good idea to have a basic knowledge of how the signaling works, that will allow you to troubleshoot issues

easily. If you are not interested in this and just want a quick guide to install OpenR2 and having it working with your telephony engine of choice, you can skip this section.

MFC/R2 is a peer-to-peer signaling protocol, which means there are just 2 parties involved in an R2 E1 link and both parties behave in the same way, unlike PRI where you have the “NET” and “CPE” sides of the link.

As mentioned before, MFC/R2 stands for Multi Frequency Compelled R2. The name describes the nature of this signaling, where you have 2 types of signals:

Line signals are used to monitor the state of the call and MF signals are used to transmit information of the call during the call setup (DNIS, ANI, Calling Party Category). Line signals are sent using CAS signals which travel using the channel 16 of the E1 link. All CAS signaling for each channel of the E1 is multiplexed through this channel. Each 2ms each side of the link update their 4 CAS signal bits known as the ABCD bits.

MFC/R2 uses 2 of those 4 bits to send the following signals: *Idle, Block, Seize, Seize Ack, Clear Back, Forced Release, Clear Forward, Answer*. Just 2 bits for 7 possible signals is not possible without repeating a bit pattern, therefore some of this 7 R2 signals have the same bit pattern, but that is not a problem considering that you cannot go from *Idle* to *Forced Release* for example, therefore, even when the bit pattern for *Forced Release* and *Seize* are the same, the protocol stack knows what the other end of the link wants. The reason to use just 2 bits having 4 available is historical and comes from the times when the analog version of MFC/R2 was ported to work in a digital world.

The following table describe the bit patterns used in R2 signaling via the CAS ABCD bits.

Circuit State	Forward AB	Backward AB
Idle/Released	1 0	1 0
Seized	0 0	1 0
Seizure Acknowledged	0 0	1 1
Answered	0 0	0 1
Clear Back	0 0	1 1
Clear Forward (before Clear Back)	1 0	0 1
Clear Forward (after Clear Back)	1 0	1 1
Blocked	1 0	1 1

Address signals, in the other hand, are 15 different MF signals, which are audible tones composed of 2 frequencies that travel using the audio channel itself, that's why an audio detector is an important component of an R2 stack, OpenR2 by default uses its own R2 MF detector,

borrowed from Steve Underwood's SpanDSP library. You don't need to install SpanDSP in order to use OpenR2 though, because the detector it is “built-in”.

The ITU defines which frequencies can be mixed to compose the MF tones and assign meanings to this tones. However, some countries assign different meanings to this MF tones. The MF tones are identified in the OpenR2 library using the numbers from 1 to 0 (0 being 10) and letters B to F (A is not used, 0 is used instead). If you enable verbose logging for the OpenR2 stack you will get the details of which tones are sent and received during each call setup. More information on this in the “Troubleshooting” section or the installation section for your telephony engine of choice.

The MF tones are used to transmit ANI (Automatic Number Identification, commonly known by some users as Caller ID), DNIS (Dialed Number Identification Service, that is, the dialed number or destiny number) and the Calling Party Category. Probably other information is exchanged as well in international R2 links (however I don't have experience with those and are probably no longer used). As soon as the call is either accepted or rejected, the MF detector is turned off and no other MF signals will be exchanged, the audio channel can then be used for voice or early media.

4. Zaptel/DAHDI installation.

OpenR2 installation is pretty straight forward, however there are some pre-requisites you should fulfill. As of now, the library requires Zaptel 1.2/1.4 or DAHDI in order to compile. The configure script will detect whether you have Zaptel 1.2, Zaptel 1.4 or DAHDI installed. You may run into troubles if you have more than one of those packages installed because OpenR2 will compile using the first that it finds.

Zaptel 1.2 installs the zaptel.h header in /usr/include/linux/
Zaptel 1.4 installs the zaptel.h header in /usr/include/zaptel/
DAHDI installs the user.h header in /usr/include/DAHDI/

The installation paths may vary, but it's a good idea to make sure you **only** have **one** of those. If you don't have Zaptel or DAHDI installed, go to <http://downloads.digium.com/> to get it.

If you plan to install Asterisk 1.2, you will need Zaptel 1.2

If you plan to install Asterisk 1.4, you will need Zaptel 1.4 or DAHDI.

If you plan to install Asterisk 1.6, you will need DAHDI.

== Zaptel 1.2 ==

```
# wget http://downloads.digium.com/<zaptel-1.2-release-path-to-file>.tar.gz  
# tar -xvpzf zaptel-<release>tar.gz  
# cd zaptel-<release>  
# make  
# make install
```

== Zaptel 1.4 ==

```
# wget http://downloads.digium.com/<zaptel-1.2-release-path-to-file>.tar.gz
# tar -xvpzf zaptel-<release>.tar.gz
# cd zaptel-<release>
# make
# make install
```

== DAHDI ==

Starting with the DAHDI release, Digium splitted the user space tools and the kernel drivers in 2 packages: dahdi-tools and dahdi-linux.

You must install first dahdi-linux, which contains the actual drivers, before dahdi-tools.

```
# wget http://downloads.digium.com/<dahdi-linux-path-to-file>.tar.gz
# tar -xvpzf dahdi-linux-<version>.tar.gz
# cd dahdi-linux-<version>
# make
# make install
```

Now you must download and install dahdi-tools. OpenR2 does not strictly require dahdi-tools, however dahdi-tools provides several binaries that are handy for troubleshooting problems (like the replacement for zttool named dahdi-tool).

```
# wget http://downloads.digium.com/<dahdi-tools-path-to-file>.tar.gz
# tar -xvpzf dahdi-tools-<version>.tar.gz
# cd dahdi-tools-<version>
# ./configure --prefix=/usr
# make
# make install
```

Now, if you have a Sangoma card, you need to install wanpipe. Sangoma drivers attach themselves to Zaptel/DAHDI, but we need to install them first. Sangoma drivers are included in a package known as wanpipe. Refer to the Sangoma wiki (<http://wiki.sangoma.com/>) for information about installation.

Now that you have installed Zaptel or DAHDI, you need to configure your hardware. DAHDI is configured in `/etc/dahdi/system.conf` and Zaptel is configured in `/etc/zaptel.conf`. Fortunately for us, `/etc/dahdi/system.conf` is backwards compatible with the syntax (but supports new options). Remember that in the R2 world, the physical link is E1 not T1 (at least I have not seen an R2 installation using something else than an E1), if you received your boards from USA or Canada probably they sent them configured for T1. Sangoma boards can change this from software, however Digium boards T1/E1 configuration needs to be changed using jumpers in the board.

The best way to determine if the span is in E1 or T1 mode is reading the contents of file `/proc/zaptel/<spanno>` or `/proc/dahdi/<spanno>` with the 'cat' command, just like this:

```
# cat /proc/zaptel/1
```

or if you have DAHDI.

```
# cat /proc/dahdi/1
```

That will list the configuration for span 1, for an E1 that will list 31 channels.

A DAHDI or Zaptel E1 span is configured like this:

```
span=1,1,0,cas,hdb3
```

```
cas=1-15:1101
```

```
dchan=16
```

```
cas=17-31:1101
```

It's important for you to know the meaning of each parameter. I describe briefly each one, but you are encouraged to read `zaptel.conf.sample` or `system.conf.sample` in your Zaptel/DAHDI distribution.

The “span” lines determines, of course, the span configuration. You can see an span as a group of channels. You can see how many spans are registered by listing the contents of directory `/proc/zaptel` or `/proc/dahdi`. Each file listed in that directory represents a span, whether or not this span represents a hardware group of channels or a software emulation channel depends on the type of the span. Kernel modules like `ztdummy` register a “dummy” span just to provide a clock (using kernel timer facilities). Ok, enough about spans, just remember that if you want to configure one of the spans listed in `/proc/zaptel` or `/proc/dahdi` you will have to do it in `/etc/zaptel.conf` or `/etc/dahdi/system.conf` using the span parameter. The span parameter is composed of the following elements:

```
span=<span num>,<timing source>,<line build out (LBO)>,<framing>,<coding>[,crc4|yellow[,yellow]]
```

For the purposes of R2 lines, feel free to ignore anything beyond the `<coding>` value. Let's describe the others.

`` refers to an integer number that represents the span number. In our example we were configuring span 1.

`<timing source>` refers to an integer that determines if the far end (whatever you have connected on the other side) will provide the clock signal and used as master source of clock timing. If you are connected to the PSTN (your telco) most likely you need to set this to a number greater than 0, depending on how many spans you have. For example, if you have 2 spans connected to the

PSTN one of those spans should be set with <timing source> to 1 and the other to 2, marking them as the first master clock source and second master clock source respectively. If you have the span connected to some legacy PBX, probably the legacy PBX expects to receive clocking from the PSTN, but since is not connected to the PSTN but to your box, you must provide clock of the PBX, in this case you set this value to 0.

```
span=1,0,0,cas,hdb3
```

That means span 1 will provide the clock to whatever is connected to the other end. Be aware that the `zaptel.conf.sample` file clearly states that incorrect timing sync may cause clicks/noise in the audio, poor quality or failed faxes and unreliable modem operation among other nasty issues.

<line build out (LBO)> is an integer between 0 and 7 which determines the transmission level of the span, you can choose that value depending on the table provided in the sample `Zaptel/DAHDI` configuration file:

```
# 0: 0 db (CSU) / 0-133 feet (DSX-1)
# 1: 133-266 feet (DSX-1)
# 2: 266-399 feet (DSX-1)
# 3: 399-533 feet (DSX-1)
# 4: 533-655 feet (DSX-1)
# 5: -7.5db (CSU)
# 6: -15db (CSU)
# 7: -22.5db (CSU)
```

<framing> and <coding> determine low level signaling and formatting of E1 frames, for the sake of R2 signaling you will use `cas` and `hdb3` respectively.

That's it for the "span". Now, let's see what the "cas" lines mean. After configuring the span, we need to configure each channel of the span with the proper signaling. In this case, the `cas` span will have all of its channels with `cas` signaling (also known as user signaling).

```
cas=x-y:1101
```

That means the range of channels from `x` to `y` will report CAS bits. The `:1101` is the initial position of the CAS bits (also known as ABCD bits) which are used by R2 to signal line state (answer, hangup etc). The `1101` signal means "Blocked" in R2, which means that when the hardware wakes up for the first time is not able to receive calls (Asterisk or any other software using R2 will later change this bits to `1001`, which means IDLE, or ready to make calls).

The `dchan=16` parameter has been a bit controversial since some people do not set it while others do. The channel 16 (probably better said, timeslot 16) is used to carry the multiplexed CAS bits for the other E1 channels. In my experience the bits are well received whether or not you set `dchan=16`, but setting it helps to clear any previous signaling the channel could have been on. I recommend setting it unless you have problems when executing "`ztcfg`" or "`dahdi_cfg`". Some versions of `ztcfg` will not accept a `cas` span with `dchan=16` and will refuse to configure the channels, just remove that parameter to fix it.

4.1 Sangoma Wanpipe Installation.

This step is only required if you have Sangoma hardware. Because Sangoma hardware works in more different scenarios, it requires some easy extra installation steps to fully integrate Sangoma boards functionality with the Zaptel/DAHDI kernel infrastructure.

Information about Sangoma Wanpipe installation may be found here:

<http://wiki.sangoma.com/wanpipe-linux-asterisk-install>

5. OpenR2.

Now that we have the hardware configuration in place, we can proceed to download and install OpenR2. The official site for openr2 is <http://www.libopenr2.org/>, however, the source code is hosted in google code site at <http://code.google.com/p/openr2/> and the downloads section is at <http://code.google.com/p/openr2/downloads/list>

1. Download OpenR2 from google code web page.

```
# wget http://openr2.googlecode.com/files/openr2-1.0.0.tar.gz
# tar -xzf openr2-1.0.0.tar.gz
# cd openr2
```

2. Run the 'configure' script.

```
# ./configure --prefix=/usr
```

The --prefix=/usr option will install the library in /usr/lib/, but if you don't specify a "--prefix" option it will be installed in /usr/local/, I recommend installing with --prefix=/usr, believe me, is gonna save you from running into problems (most of the time anyway). If the configure script does not found the Zaptel or DAHDI headers then it will fail with an error. Be sure to search for configure errors and install any dependency you might be missing. In general, just Zaptel or DAHDI is required.

3. Compile.

```
# make
```

If there are no errors on this step, continue. An error in this stage most likely means a bug or that you are compiling in an unsupported platform. You can ask for support in the lists or IRC (see 1. About this guide). But be sure to look for a descriptive error in the output and provide it.

4. Install

```
# make install
```

That's it, you should be ready to go. Remember that usually you need to be root to execute this last step, unless you specified a "--prefix" option to the script pointing to a directory that is writable for your user.

Now that the library is installed, you can proceed to install Asterisk. However, you can also test your R2 lines without Asterisk being involved. Read the section 6 "OpenR2 test" to learn how to test without Asterisk. If you are not interested in testing without Asterisk, you can proceed to section number 7 "OpenR2 in Asterisk".

6. OpenR2 test

The OpenR2 library provides you with a small test program called "r2test" (unless you specified the option --without-r2test). Is a small application that uses a configuration file to "listen" for calls into the Zaptel/DAHDI devices you specify and answer calls and either put them in an echo-like application or plays a file (in alaw format). The r2test program not just wait for calls, it can also be configured to make calls.

In order to use r2test you need first to create a configuration file. A sample configuration file is included in OpenR2 in the doc/r2test.conf, the format for the file is:

```
# this is a comment  
parameter1=value  
parameter2=value  
channel=start-end
```

When r2test finds a "channel" parameter, it will create a new block of DAHDI/Zaptel channels starting with "start" and ending with "end" and configure them with any preceding parameters. You should make sure "start" and "end" are valid DAHDI/Zaptel devices. For example, this is a minimal r2test.conf configuration file:

```
variant=mx  
caller=no  
maxani=10  
maxdnis=4  
channel=1-15
```

An r2test.conf file with such contents would open DAHDI/Zaptel channels from 1 to 15, and configure them with caller=no, maxani=10 and maxdnis=4. If you get "No such file or directory" errors, it might be because you have not loaded the DAHDI/Zaptel drivers. When DAHDI/Zaptel drivers are loaded, they create devices in /dev/dahdi or /dev/zap, in the previous configuration example devices /dev/zap/1 to /dev/zap/15 or /dev/dahdi/1 to /dev/dahdi/15 are required.

The “variant” parameter determines which R2 variant will be used. You can list the supported variants with the command:

```
# r2test -l
```

For openr2 1.0.0 the list is:

```
Variant Code Country
AR Argentina
BR Brazil
CN China
CZ Czech Republic
CO Colombia
EC Ecuador
ITU International Telecommunication Union
MX Mexico
PH Philippines
VE Venezuela
```

The “caller” parameter determines whether that group of channels will make calls or wait for calls, “caller=yes” means the channels will make calls, in that case the “dnid” parameter must be specified.

Parameters “maxani” and “maxdnis” just determine what is the maximum number of digits that will be received for ANI and DNIS.

The “channel” parameter ***must*** be in the format “x-y” to specify the range of channels for which the previous configuration parameters will be applied to (just like zapata.conf or chan_dahdi.conf). If you just need 1 channel you must specify it like “channel=1-1” (yeah, it sucks but is the way currently works, patches accepted).

Once the r2test.conf file was created you can start r2test like this:

```
# r2test -c r2test.conf
```

The default behavior after completing a call is to put the audio in an echo-like application where everything said is repeated immediately. However, you can also specify the options:

```
playaudio=yes
audiofile=doc/intro-openr2-es.alaw
```

Assuming you are running r2test from the same folder where you have the sources. OpenR2 provides 3 sample audio files in ALAW format, intro-openr2-xx.alaw in es (spanish), br (Portuguese) and en (english). The ALAW format is the only one openr2 understands.

There is just a couple of options more I want to discuss. Debugging options. These are useful when you are trying to get help in mailing lists or IRC. Debugging output may not mean anything to you but will greatly help to anyone who understands the R2 signaling protocol.

The option “loglevel” is quite powerful for filtering messages. The following logging values are available:

Levels 'error', 'warning', 'notice' and 'debug' are self-descriptive, will log any critical messages, warning messages, notice messages and general debug messages.

Level 'cas' is for logging CAS tx and rx signals.

Level 'mf' is for logging the multi frequency tones tx and rx.

Level 'all' is a special value that enables all the debugging messages.

Level 'nothing' is another special value to not log anything.

When requesting support make sure that your debugging level is set to 'all'.

You can mix up values like this: “loglevel=warning,error,notice”.

There is another interesting option for debugging that is independent of the loglevel option, this is “callfiles”. This option accepts “yes” or “no”. If you chose “yes” a special “.call” file will be dropped after each call in the directory where you run r2test. These “.call” files contain valuable debugging information.

The previously described options are the most important ones, but, not the only ones. You can find more extensive (and probably more accurate) documentation about all the options in the sample r2test.conf file provided with openr2 in the doc/ folder.

There is just one final option for r2test, that is “-v” to display the openr2 version installed.

```
# r2test -v
```

For more information about r2test you can also try “man r2test” to see the help.

5. OpenR2 in Asterisk.

Asterisk does not understand R2 signaling by itself, it requires the help of the OpenR2 library to do it. But you also need to make Asterisk aware of the presence of the OpenR2 library so the proper code inside Asterisk can be compiled to make use of OpenR2. There are currently 2 ways to get an R2-enabled Asterisk version. You can either download an Asterisk SVN branch that is already patched and ready to play with OpenR2, or you can download a normal Asterisk .tar.gz and then download the OpenR2-Asterisk patches and apply them yourself.

Be aware, however, that if you download a normal Asterisk .tar.gz and then try to apply the OpenR2-Asterisk patch, the patch may fail to apply if the Asterisk .tar.gz is old too new. I try to have patches to most Asterisk major versions, but minor versions change very often and I cannot always keep up with those changes quickly.

Asterisk 1.6 is an exception, it does not have a stable branch as 1.2 and 1.4 do. The 1.6 branch is a copy of the trunk branch and is therefore a development branch.

Given the Asterisk development cycles, Asterisk 1.2 and Asterisk 1.4 will never include MFC/R2 support in official releases because those 2 versions are already frozen for features, which means Digium, the company leading the Asterisk development, will NOT accept big feature changes like MFC/R2 support. However, Asterisk 1.6 has very good chances of being the first Asterisk version ever including native MFC/R2 support, and it will do so using OpenR2 library. More information about the progress of this inclusion can be found here:

<http://bugs.digium.com/view.php?id=12509>

If you don't want to install Asterisk from an SVN branch and you would rather apply a patch to a specific Asterisk version you can skip this section.

== Asterisk with MFCR2 SVN branch installation ==

Having said that, I have 3 Asterisk branches in order to support R2 in Asterisk 1.2, 1.4 and 1.6. The branches for Asterisk 1.2 and 1.4 are a copy of the latest stable Asterisk version, I just copied the code using SVN and added the code necessary to make Asterisk use OpenR2. The 1.6 branch is a copy of the “trunk” development branch, and is therefore considered unstable, you can help to speed up the development process by testing this development branch.

Asterisk-OpenR2 1.2 branch: <http://svn.digium.com/svn/asterisk/team/moy/mfcr2-1.2>

Asterisk-OpenR2 1.4 branch: <http://svn.digium.com/svn/asterisk/team/moy/mfcr2-1.4>

Asterisk-OpenR2 1.6 branch: <http://svn.digium.com/svn/asterisk/team/moy/mfcr2>

The steps for installation are the same that one usually follows to install any Asterisk version, the only difference is that this time you download Asterisk using SVN. Here are some brief instructions depending on the Asterisk version you want.

== Asterisk 1.2 ==

```
# svn checkout http://svn.digium.com/svn/asterisk/team/moy/mfcr2-1.2
```

```
# cd mfcr2-1.2
```

```
# make
```

```
# make install
```

When you type “make”, the install script will detect if you already have OpenR2 installed, if so, it will compile Asterisk with chan_zap.so ready to be used with MFC/R2. If you don't have OpenR2 installed, chan_zap.so may still be compiled (if Zaptel 1.2 is present) but **without** MFC/R2 support, which defeats the whole purpose of the branch anyway.

== Asterisk 1.4 ==

```
# svn checkout http://svn.digium.com/svn/asterisk/team/moy/mfcr2-1.4
```

```
# cd mfc2-1.4
# ./configure --prefix=/usr
# make
# make install
```

Starting with Asterisk 1.4, it is required to run the configure script to install Asterisk. This configure script will take care of detecting the libraries you have installed on your system, including OpenR2. If OpenR2 is found, Asterisk will be installed with chan_zap.so enabled for MFC/R2 signaling. If you don't have OpenR2 installed, chan_zap.so may still be compiled (if Zaptel 1.4 is present) but **without** MFC/R2 support, which defeats the whole purpose of the branch anyway.

== Asterisk 1.6 ==

```
# svn checkout http://svn.digium.com/svn/asterisk/team/moy/mfc2
# cd mfc2
# ./configure --prefix=/usr
# make menuselect
```

Starting with Asterisk 1.4, it is required to run the configure script to install Asterisk. This configure script will take care of detecting the libraries you have installed on your system, including OpenR2. If OpenR2 is found, Asterisk will be installed with chan_dahdi.so enabled for MFC/R2 signaling. If you don't have OpenR2 installed chan_dahdi.so may still be compiled (if DAHDI is present) but **without** MFC/R2 support, which defeats the whole purpose of the branch anyway.

This is a development branch, which means things may change frequently in different parts of Asterisk, not just the R2 code. If you wish to help to support R2 officially into Asterisk, please test this branch and let us know your results at <http://bugs.digium.com/view.php?id=12509>, this will help to have R2 support in official Asterisk releases sooner. In order to install this branch **you will need DAHDI** installed. This branch **does not work with any version of Zaptel**, just with DAHDI, this is because Asterisk 1.6 has removed Zaptel support completely.

In order to verify if chan_zap.so or chan_dahdi.so are properly compiled and linked with OpenR2, you can execute this command:

== Asterisk 1.2 and 1.4 ==

```
# ldd channels/chan_zap.so | grep openr2
```

== Asterisk 1.6 ==

```
# ldd channels/chan_dahdi.so | grep openr2
```

That command should show something like this:

libopenr2.so.1 => /usr/lib/libopenr2.so.1 (0x000000000589000)

The load address (0x000000000589000) and library version (.1) can be different. However, if you don't see any output then OpenR2 is not enabled in Asterisk, therefore is very likely you did not install openr2 properly or installed it AFTER having installed Asterisk.

== Asterisk with OpenR2 patches ==

A set of patches for different Asterisk versions is available at:

<http://code.google.com/p/openr2/downloads/list>

The patches include in the name the lowest Asterisk version they are known to work on. For example, a patch named openr2-asterisk-1.4.18 is known to work with at least Asterisk 1.4.18, it may or not work with Asterisk 1.4.18.1 or higher versions. If a patch applies without being rejected by the “patch” command, has very good chances of working fine without run time errors. The patches do not include sample configuration documentation, you need to download the chan_dahdi_or_zapata.conf.sample file in order to read the sample documentation.

In this case we assume we certain Asterisk versions, but you can change the version to whatever you want as long as you keep in mind what I have mentioned regarding versions of the patches.

== Asterisk 1.2 ==

```
# wget http://downloads.digium.com/pub/asterisk/asterisk-1.2.31.tar.gz
# tar -xzf asterisk-1.2.31.tar.gz
# cd asterisk-1.2.31
# wget http://openr2.googlecode.com/files/openr2-asterisk-1.2.30.3.patch
# patch -p0 < openr2-asterisk-1.2.30.3.patch
# make
# make install
```

== Asterisk 1.4 ==

```
# wget http://downloads.digium.com/pub/asterisk/asterisk-1.4.22.1.tar.gz
# tar -xzf asterisk-1.4.22.1.tar.gz
# cd asterisk-1.4.22.1
# wget http://openr2.googlecode.com/files/openr2-asterisk-1.4.22.patch
# patch -p0 < openr2-asterisk-1.4.22.patch
```

```
# ./bootstrap.sh
# ./configure --prefix=/usr
# make
# make install
```

Please note that running the “./bootstrap.sh” Asterisk script is very important. The bootstrap script generates a new configure script that will properly detect the presence of OpenR2 and create a Makefile that compiles chan_dahdi or chan_zap with OpenR2 support. The bootstrap script seems to be very sensible to the “autoconf” version. If you don't want to run into problems is recommended that you use autoconf version 2.60 exactly, not newer, not older. If you use a different autoconf version then the bootstrap script may or may not work. To check the autoconf version you can execute:

```
# autoconf --version
```

== Asterisk 1.6 ==

```
# wget http://downloads.digium.com/pub/asterisk/asterisk-1.6.0.3.tar.gz
# tar -xzf asterisk-1.4.22.1.tar.gz
# cd asterisk-1.4.22.1
# wget http://openr2.googlecode.com/files/openr2-asterisk-1.6.0.patch
# patch -p0 < openr2-asterisk-1.6.0.patch
# ./bootstrap.sh
# ./configure --prefix=/usr
# make
# make install
```

See the notes on Asterisk 1.4 for more information on the ./bootstrap.sh script requirements.

At this point you should have Asterisk ready for MFC/R2 support with OpenR2. In order to verify if chan_zap.so or chan_dahdi.so are properly compiled and linked with OpenR2, you can execute this command:

== Asterisk 1.2 and 1.4 ==

```
# ldd channels/chan_zap.so | grep openr2
```

== Asterisk 1.6 ==

```
# ldd channels/chan_dahdi.so | grep openr2
```

That command should show something like this:

libopenr2.so.1 => /usr/lib/libopenr2.so.1 (0x0000000000589000)

The load address (0x0000000000589000) and library version (.1) can be different. However, if you don't see any output then OpenR2 is not enabled in Asterisk, therefore is very likely you did not install openr2 properly, patched Asterisk incorrectly or installed openr2 AFTER having installed Asterisk.

5.1 Asterisk MFC/R2 configuration.

Whichever branch you installed, a sample configuration file is provided either in configs/chan_zap.conf.sample or configs/chan_dahdi.conf.sample, search through the parameters starting with 'mfc_r2' to see the documentation of each one of those. If installed Asterisk with OpenR2 using the provided patches then the documentation is not included and you can download a sample configuration from here:

http://openr2.googlecode.com/files/chan_dahdi_or_zapata.conf.sample

The configuration of an R2 link will vary depending on your country or the model of the PBX you are trying to connect to. The OpenR2 package includes a doc/ directory with an asterisk/ subdirectory with sample configurations for several countries. Take a look there to see if your country already includes some samples.

6. OpenR2 in FreeSwitch.

Not supported yet :(... but it's coming!

7. Building packages.

7.1 Building Debian packages

Download the openr2 .tar.gz and decompress it. Then change into the openr2 directory and type this command:

```
# dpkg-buildpackage -uc -us
```

The packages will be generated in the upper directory.

7.2 Building RPM packages.

Download the openr2 .tar.gz and type this command:

```
# rpmbuild -ta openr2-1.0.0.tar.gz
```

The packages will be generated in rpmbuilds/RPMS/<arch>

