XML Dialplan

About

The XML dialplan is the default dialplan used by FreeSwitch. XML is easily edited by hand without requiring special tools, other than a text editor.

In general, dialplans are used to route a call to an endpoint, which can be a traditional extension, voicemail, interactive voice response (IVR) menu or other compatible application. Dialplans are extremely flexible.

Dialplans can be separated into contexts, allowing calls to follow different pathways for different kinds of calls. Calls can be handed-off to other contexts as well. For example you might configure two dialplans: one that handles calls originating from the public phone network (PSTN) and one that handles calls originating from internal extensions. The sample dialplans in the FreeSWITCH default (or vanilla) configuration are following the exact same principle: forcing an incoming external PSTN call through some additional scrutiny (via the public context in conf/dialplan/public.xml) before being handed off to the internal dial plan (the default context in conf/dialplan/default.xml).

Dialplan contexts also allow you to share a single PBX with multiple tenants in an office building. Since each tenant will likely have their own set of (and often conflicting) extensions, voice menus, etc., it makes sense to separate tenants into their own independent dialplans to ease configuration and maintenance.

Learning Prerequisites

Understanding this documentation requires an understanding of regular expressions. XML dialplans use the common Perl Compatible Regular Expression (PCRE) matching syntax on fields, which decreases the "learning curve" when creating and maintaining dialplans.

Simple dialplans can be created by anyone with a working knowledge of PCRE syntax.

Authors of moderately complex XML dialplans would benefit from in-depth PCRE experience along with a working knowledge of variables and flow-control used in scripting or programming languages.

Learning Goals

After reading this document, you should be able to:

- Add a configuration statement to the Sofia SIP profile to call an XML dialplan. (See Configuring FreeSWITCH#SIPProfiles, Sofia Configuration Files, Sofia SIP Stack, mod_sofia pages for more information on SIP profiles.)
- Create a moderately complex XML dialplan using regular expressions to reduce a 1000 extension dialplan to one extension definition.
- Understand the flow-altering capabilities of conditions and nested conditions, and why they might be useful in a dialplan.

The 10,000 Foot View

This overview uses the Sofia SIP driver, mod_sofia, as the source and destination (i.e., an endpoint) for calls handled by a dialplan. Other drivers have similar mechanisms.

When a call arrives at the FreeSWITCH™ PBX, Sofia is the first responder. She gathers information about the call, and decides which dialplan to invoke. Sofia passes information about the call to your dialplan inside channel variables, which your dialplan accesses to make decisions about what to do with the call. Channel variables contain a wealth of information about the call being processed. For example, the destination_number variable contains the digits dialed by the caller. Other variables contain the caller-ID information for the call, the source IP address of the caller, etc.

The XML dialplan is organized as a series of extension definitions (called extensions). FreeSWITCH™ steps through each extension definition until it finds one that matches.

Matching is performed by evaluating condition definitions (called conditions) inside each extension. FreeSWITCH™ steps through each condition inside the extension.

When condition(s) for an extension are met, the extension's action definitions (called actions) are executed. In the simplest case, the action might be to connect the call to the physical extension. Of course there are a wide range of actions you can perform in a dialplan. For the purposes of this overview, we'll just use a few of them.

If the conditions for your extension are not met, there are optional action definitions (called "anti-actions") that can be executed.
Of course these are just the basic concepts needed to start working with the XML dialplan. If you are familiar with other programming or scripting languages, XML dialplans offer logical operations and control-flow options allowing you to create amazingly complex dialplans. But don’t think the XML dialplan is the answer to all your problems. Complex XML dialplans are sometimes difficult to read and understand, which is why other dialplan processors are available. That said, the XML dialplan is a good place to start, and usually does the bulk of the work when FreeSWITCH™ processes a call.

**Introductory Example**

This following example shows a dialplan that understands how to route calls to only two extensions, 500 and 501. But before it can be used, you’ll need to tell the SIP driver the name of the dialplan (called the **context**) to use when a call needs processing.

```xml
<profile name="example_sip_profile">
  <param name="context" value="example"/>
  ...other configuration statements...
</profile>
```

The sample dialplan is shown below.

```xml
<context name="example">
  <extension name="500">
    <condition field="destination_number" expression="^500$">
      <action application="bridge" data="user/500"/>
    </condition>
  </extension>

  <extension name="501">
    <condition field="destination_number" expression="^501$">
      <action application="bridge" data="user/501"/>
      <action application="answer"/>
      <action application="sleep" data="1000"/>
      <action application="bridge" data="loopback/app=voicemail:default ${domain_name} ${dialed_extension}"/>
    </condition>
  </extension>
</context>
```

When a call needs processing, FreeSWITCH™ evaluates each extension in the dialplan until it finds a match.

The first line in our example defines the context block. All items inside this block belong to the "example" dialplan context. Nested inside the context block are extension blocks, containing matching "conditions" and their corresponding "action" rules. Extensions are processed in the order that they appear in the configuration file.

In the example, Sofia would start the dialplan after she receives an incoming SIP call. Here’s what happens next:

- The first extension is block is evaluated, and since the destination number does not exactly match "500", the rest of the block is skipped.
- The second extension block is evaluated next. The destination matches "501" exactly, which causes the actions under that condition to be executed. In this example, extension 501 rings. If the extension is not answered, the second action answers the call, and the following actions delay for 1000 milliseconds (1 second) and then connect the call to the voicemail system.
Dialplan Configuration Directory Structure

Find under: ${conf_dir}/dialplan/

The FreeSWITCH "Vanilla" configuration stores dialplans in the ${conf_dir}/dialplans directory, with each context stored in a subdirectory beneath it. This is the recommended configuration.

For example, this sample structure houses a dialplan for a simple PBX:

**Dialplan hierarchy**

```
freeswitch/
  conf/
    dialplans/
      public.xml
      public/
        00_security_screen.xml
        10_inbound_sip-bandwidth-r-us.xml
        29_inbound_sip-super-call.xml
        ...
      default.xml
      default/
        00_feature_codes.xml
        05_voicemail_access.xml
        20_extension_2001.xml
        20_extension_2002.xml
```

**Things to notice**

- This dialplan structure is similar to the example configuration that ships with FreeSWITCH™. It is a good starting point to customize your own PBX, as we have done here.
- Any `.xml` file placed in the `dialplans` directory will be loaded when FreeSWITCH starts. `public.xml` contains all the common configuration information for the `public` context, then includes all the XML files in the `public` directory.
- `default.xml` contains all the common configuration information for the `default` context, then includes all the XML files in the `default` directory.
- Files are loaded in alphabetical order, so beginning file names with a number is a good practice to ensure the bits and pieces of your dialplan are loaded in the proper sequence, if needed.

**Beginning Concepts**

**Context**

Contexts are a logical grouping of extensions.

The context tag has only one required parameter called name. This is the name that incoming call handlers (like the [Sofia] SIP driver) use when they need to route a call. There is often more than one context in a typical PBX dialplan. There is one reserved name, any which matches any context.
Extensions

Inside each context are one or more extensions. Extensions are destinations for a call. In FreeSWITCH™, extensions are the primary way calls are routed. Don't be fooled by the name, since an extension does not always have a real phone attached. An extension can be used to process call routing logic, security restrictions, outgoing trunk selection and more.

The extension tag has one required parameter name, and it must be unique. Other dialplans can transfer control to your dialplan using this name.

Inside an extension are one or more conditions. If the conditions are met, the actions belonging to that condition are executed. If the conditions are not met, optional anti-actions are executed.

For example:

```xml
<extension name="Your extension name here">
  <condition(s>...
    <action(s) .../>
    <anti-action(s) .../>
  </condition>
</extension>
```

Typically when an extension is matched in your dialplan, the corresponding actions are performed and dialplan processing stops. An optional continue parameter allows your dialplan to continue evaluating the remaining extensions:

```xml
<extension name="500" continue="true">
```

Conditions

Conditions do the hard work in a dial-plan. They are extremely flexible, and as such, are sometimes challenging to construct. This document will start with simple examples and progress forward to more complex examples.

Conditions allow you to match regular expressions against channel variables associated with each call. The first example in this document did just that, matching the destination number against "500", as shown below.

```xml
<extension name="500">
  <condition field="destination_number" expression="^500$">
    <action application="bridge" data="user/500"/>
  </condition>
</extension>
```

Each condition is parsed using the Perl Compatible Regular Expression library. (See the Perl Compatible Regular Expression documentation to learn more.)

Example 1: Capturing Digits

The next example will use a feature of PCRE to show how to route a local call to the local Telco/PTT. (Local providers often insist that local calls be stripped of their country/area/city code.)

In the USA, the last 7 digits of the full 11 digit phone number are the local part. So, if our PBX is located in Los Angeles, the first four digits will always be 1213 for local calls, so we can check for that:

```xml
<extension name="500">
  <condition field="destination_number" expression="^1213\d\d\d\d\d\d\d\d$">
    <action application="bridge" data="user/500"/>
  </condition>
</extension>
```

The condition will match any 11 digit number beginning with "1213". To capture the final seven digits, we write it this way, enclosing the last seven digits in parentheses.

```xml
<extension name="500">
  <condition field="destination_number" expression="^1213(\d\d\d\d\d\d\d\d)$">
    <action application="bridge" data="user/500"/>
  </condition>
</extension>
```
The condition will still match any 11 digit number beginning with 1213. The added feature is that the final seven digits of the number are now available in the temporary variable $1$. Now we can make a complete dialplan extension that passes all local calls to the local phone company.

```xml
<extension name="local_calls">
    <condition field="destination_number" expression="^1213\d\d\d\d\d\d\d\d$">
        <action application="bridge" data="sofia/gateway/LocalTelco/$1"/>
    </condition>
</extension>
```

A destination number of 12135551212 would set $1 to 5551212 and then send the call out to the local phone company.

---

**Example 2: Logical AND**

Suppose you wanted to redirect a call for extension 500 to extension 531, but only on Sunday.

In English you would say “if destination number is 500 and today is Sunday, send the call to ext. 531, otherwise send it to ext. 500”

In the XML dialplan, you can place more than one condition in an extension, and all conditions must match before the final condition’s actions will be executed.

This does the trick:

```xml
<condition field="destination_number" expression="^500$"/>
<condition wday="1">
    <action application="bridge" data="sofia/internal/531@example.com"/> # It's Sunday: Forward the call.
    <anti-action application="bridge" data="sofia/internal/500@example.com"/> # It's not Sunday: Act normally.
</condition>
```

---

**Things to Notice**

- We use the “anti-action” here to ensure that if the final condition fails, the call will go to the proper place.
- Don’t be fooled: FreeSWITCH™ does not evaluate all the conditions to make its decision. This design pattern only works because FreeSWITCH™ stops processing an extension after the first condition failure.
- Keep in mind that you must observe correct XML syntax when using this structure. Be sure to close all conditions except the last one with `/>`. The last condition contains the final actions to be run, and is closed on the line following the last action.

**Example 3: Logical OR**

It is possible to emulate the logical OR operation available in many programming languages, using multiple conditions. In this situation, if one of the conditions matches, the actions are executed.

There are a few ways to do this. Use the one that meets your needs.

The simplest version using a single regular expression that matches the destination number against 501 OR 502:

```xml
<condition field="destination_number" expression="^501|502$">
    action(s)...
</condition>
```

This method works well if your OR condition uses the same field. If you need to use two or more different fields then the extended regular expression syntax will help:
Notice the regex="any" in the condition. This says that if any of the following regex statements match, then execute the actions. Of course if none of the regular expressions match, the anti-actions (if any) will be executed instead. Here's an example that executes the extensions actions if the caller's name is "Some User" OR the extension being called is 1001.

```xml
<extension name="Regex OR Example">
  <condition regex="any">
    <!-- If either of these is true then the subsequent actions are added to execute list -->
    <regex field="caller_id_name" expression="Some User"/>
    <regex field="caller_id_number" expression="^1001$"/>
    <action application="log" data="INFO At least one of the conditions matched!"/>
    <!-- If *none* of the regexes is true then the anti-actions are added to the execute list -->
    <anti-action application="log" data="WARNING None of the conditions matched!"/>
  </condition>
</extension>
```

The `condition regex="any"` has two other options: all and xor. So the full compliment of options looks like this:

- `regex="any"` — We used this above. Any of the regular expressions contained in the condition can match for the actions to be taken.
- `regex="all"` — is equivalent to a logical AND operation. All of the regular expressions contained in the condition must match for the actions to be taken. This is another way to perform a logical AND, like we did in the previous section. Use the syntax that makes your dialplan clearer and easier to read.
- `regex="xor"` — is the equivalent to a logical XOR operation. XOR is commonly known as "eXclusive-OR", because exactly one of the regular expressions must match for the actions to be taken.

Actions and anti-actions are executed like they would be in a standard condition block.

This method makes it easier to match the caller's name OR caller ID number and execute actions when either is true.

A slightly more advanced use of this method is demonstrated below. If the caller's name is "Michael S Collins" OR the caller ID is 1002, 3757, or 2816, the variable calling_user is set to "mercutioviz". If neither is true, then the calling_user variable is set to "loser". After playing the welcome message, a custom message is played based on the calling_user variable.

```xml
<extension name="Regex OR example 2" continue="true">
  <condition regex="any" break="never">
    <regex field="caller_id_name" expression="^Michael\s*S\s*Collins$/"/>
    <regex field="caller_id_number" expression="^1001|3757|2816$"/>
    <action application="set" data="calling_user=mercutioviz" inline="true"/>
    <anti-action application="set" data="calling_user=loser" inline="true"/>
  </condition>
</extension>
```

```xml
<extension>
  <condition application="answer"/>
  <action application="sleep" data="500"/>
  <action application="playback" data="ivr/ivr-welcome_to_freeswitch.wav"/>
  <action application="sleep" data="500"/>
</condition>
```

```xml
<condition field="$\{calling_user\}" expression="^loser$">
  <action application="playback" data="ivr/ivr-dude_you_suck.wav"/>
  <anti-action application="playback" data="ivr/ivr-dude_you_rock.wav"/>
</condition>
```

```xml
</extension>
```
Example 4: Logical XOR

We've seen how the regex= clause can be used to implement AND and OR. For completeness, here is a final example showing how to use XOR. The actions are executed if either the first or second regular expression matches, but not if both match.

```xml
<extension name="Regex XOR example 3" continue="true">
  <condition regex="xor">
    <!-- If only one of these is true then the subsequent actions are added to execute list -->
    <regex field="caller_id_name" expression="Some User"/>
    <regex field="caller_id_number" expression="^1001$"/>
    <action application="log" data="INFO Only one of the conditions matched!"/>
    <!-- If *none* of the regexes is true then the anti-actions are added to the execute list -->
    <anti-action application="log" data="WARNING None of the conditions matched!"/>
  </condition>
</extension>
```

Actions and Anti-Actions

The final parts of an extension definition are its actions. These cause FreeSWITCH™ to do things on your behalf, like connect a call or provide other functions. The list of available actions is long and varied, and can be found [here](#). Some of the more useful actions are:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>answer</td>
<td>Answer the call</td>
</tr>
<tr>
<td>bridge</td>
<td>Bridge the call to another session</td>
</tr>
<tr>
<td>log</td>
<td>Write a message in the log file</td>
</tr>
<tr>
<td>hangup</td>
<td>Disconnect the call</td>
</tr>
<tr>
<td>playback</td>
<td>Pay an audio file or tone stream</td>
</tr>
<tr>
<td>set</td>
<td>Set a channel variable</td>
</tr>
<tr>
<td>transfer</td>
<td>Transfer the call the another extension</td>
</tr>
</tbody>
</table>

It bears repeating: Actions are not executed "in-line" with the conditions they are bundled with. FreeSWITCH™ first checks all the conditions in an extension before it executes any actions. In most cases you cannot rely on an action being performed and having its result be available for checking (or using in) a `<condition...` or `<action...` that follows it.

1In some cases, the action may support the inline="true" option which causes the action to execute along with its matching condition. Not all dialplan applications support this. The set application is probably the most likely place you would use this feature, as we showed in [this example](#).

Anti-Actions

In a few of our introductory examples, we showed the use of the anti-action expression. As its name implies, anti-actions are executed when the condition they are contained in evaluate to false.

For every condition, if the condition's actions aren't executed, the anti-actions will be executed, and vice-versa.

Dialplan Variables

So far you've seen the basic pieces of the dialplan and its structure. The examples have shown the basic use of:

- Extensions
Conditions
Simple Regular Expressions
Actions and Anti-Actions

We have only touched upon dialplan variables. They are essential to do anything useful inside a FreeSWITCH™ dialplan.

Variables are named bits of information that are stored for later use. When a dialplan starts, a wealth of information comes pre-loaded in the form of variables. You can use regular expressions to match against a variable and control your dialplan execution as a result.

We will touch on a few common variables here to solidify their proper use and demonstrate some features of variable substitution in FreeSWITCH™.

Accessing Variables

There are three kinds of variables available for you to access in your XML dialplan:

- Channel variables
- Global variables
- Built-in variables (time, date, etc.)

Channel Variables

Channel variables can be used to get information about a call and control its behavior as it progresses through your dialplan. Channel variables contain a wealth of information about the call being processed and an amazing array of options that can be set. You access the contents of a channel variable by surrounding its name with ${ }. For example, to append the default SIP domain to the end of a dial string, you can just reference \${domain_name} in the dial string text, and its value will be substituted automatically:

```
<action application="bridge" data="user/1000@$\{domain_name}"
```

You set the value of a channel variable like this:

```
<action application="set" data="call_timeout=30"/>
```

For the complete list of Channel Variables and alternate ways to use them in dial strings, see: Channel Variables. We'll present a short list here to give you a sense of what's possible.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller_id_name</td>
<td>The name of the calling party.</td>
</tr>
<tr>
<td>destination_number</td>
<td>The number that the calling party dialed.</td>
</tr>
<tr>
<td>direction</td>
<td>Whether this call leg is inbound or outbound</td>
</tr>
<tr>
<td>channel_name</td>
<td>The name of this call's inbound channel, for example: sofia/sales/John_Smith@192.168.1.1</td>
</tr>
<tr>
<td>call_timeout</td>
<td>Set to the number of seconds to wait for a call to be answered before giving up.</td>
</tr>
<tr>
<td>state</td>
<td>The state of this channel, for example CS_EXECUTE or CS_HANGUP</td>
</tr>
<tr>
<td>bridge_hangup_cause</td>
<td>The reason a call ended, for example NO_ANSWER, NORMAL_CLEARING or USER_BUSY</td>
</tr>
</tbody>
</table>

Built-In Variables

Built-in variables are accessed by name, without using the customary $ prefix.

The built-in variables are: (updated 29-July-2014, switch_xml.c, switch_xml_std_datetime_check())

<table>
<thead>
<tr>
<th>Built-In Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Date and Time</td>
<td></td>
</tr>
</tbody>
</table>
date-time  The current data and time: YYYY-MM-DD HH:MM:SS  
For example: 2014-08-12 15:34:59

time-of-day  The current local time: HH:MM:SS  
For example: 15:34:59

Data and Time Parts (all based on local timezone)

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>The current year through. (1970 through 9999)</td>
</tr>
<tr>
<td>mon</td>
<td>The current month. (1 through 12)</td>
</tr>
<tr>
<td>mday</td>
<td>The day of the current month. (1 through 31)</td>
</tr>
<tr>
<td>hour</td>
<td>The hour of the current day. (0 through 23)</td>
</tr>
<tr>
<td>minute</td>
<td>The minute of the current hour. (0 through 59)</td>
</tr>
</tbody>
</table>

Advanced Data and Time Parts

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wday</td>
<td>The day of the current week. (Sunday = 1 through 7)</td>
</tr>
<tr>
<td>week</td>
<td>The current week since January 1st. (1 through 53)</td>
</tr>
<tr>
<td>mweek</td>
<td>The week of the current month. (1 through 6)</td>
</tr>
<tr>
<td>yday</td>
<td>The day of the current year (1 through 366)</td>
</tr>
<tr>
<td>minday</td>
<td>The current minute of the day starting at midnight. (0 through 1440)</td>
</tr>
<tr>
<td>tz-offset</td>
<td>Timezone offset in hours from GMT. (-11 to +11)</td>
</tr>
<tr>
<td>dst</td>
<td>Returns 1 if daylight savings time is in effect, or 0 if not.</td>
</tr>
</tbody>
</table>

In condition statements, you can compare against more than one built-in variable. All comparisons must match (logical AND) before its actions will be executed.

There are a number of ways match against a built-in variable. You can:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>&lt;condition wday=&quot;1&quot;&gt;</td>
<td>Is it Sunday?</td>
</tr>
<tr>
<td>Range</td>
<td>&lt;condition wday=&quot;2-4&quot;&gt;</td>
<td>Is it Monday, Tuesday, or Wednesday?</td>
</tr>
<tr>
<td>List</td>
<td>&lt;condition wday=&quot;1,4&quot;&gt;</td>
<td>Is it Sunday or Wednesday?</td>
</tr>
<tr>
<td>Combination</td>
<td>&lt;condition wday=&quot;1-3,7&quot;&gt;</td>
<td>Is it Sunday, Monday, Tuesday, or Saturday?</td>
</tr>
</tbody>
</table>

The range operation can be used on times and dates as well:

<condition time-of-day="08:00:00-09:00:00">  
<condition date-time="2010-10-01 00:00:01-2010-10-15 23:59:59">  

⚠️ The range operator for dates is "~", not "-".

Advanced Concepts

Nested Conditions

Conditions can be nested one inside each other. Nested conditions is hard, very very hard to get right. You may want to use a script, instead (yes, yes, you want to use a script). See mod_lua for Lua scripting.
When a (main) condition has nested conditions, first its (main) expression is evaluated, and its (main) actions are pushed in TODO list, then nested conditions are evaluated, and their actions pushed.

Each condition has an implied parameter "require-nested", that is by default set to "true".

When we have a condition with require-nested parameter set to "true", then ALL its nested conditions must evaluate to true (AND its expression must evaluate to true) for the condition to be evaluated to true.

If "require-nested" is set to false, then only the expression must evaluate to true for the condition to be true, whatever the nested conditions are.

Conditions are checked the usual way (respecting the "break" parameter), and actions inside conditions added to TODO list.

Let's see an example extension with nested conditions you can play with, checking all possible cases and corner cases.

```xml
<extension name="nested_example">
  <condition field="destination_number" expression="29010" require-nested="false">
    <action application="log" data="ERR 00 CIDnum is ${caller_id_number} CIDname is ${caller_id_name}"/>
    <action application="set" data="var_01=N/A" inline="true"/>
    <action application="set" data="var_02=N/A" inline="true"/>
    <action application="set" data="var_03=N/A" inline="true"/>
    <action application="set" data="var_04=N/A" inline="true"/>
    <action application="set" data="var_05=N/A" inline="true"/>
    <action application="log" data="ERR 01 I'm before..."/>
    <action application="log" data="ERR 02 I'm before ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
    <condition field="caller_id_number" expression="1011" break="on-false">
      <action application="log" data="ERR 03 I'm the first..."/>
      <action application="log" data="ERR 04 I'm the first CIDnum is ${caller_id_number}"/>
      <action application="set" data="var_02=02" inline="true"/>
      <action application="log" data="ERR 05 I'm the first ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
      <condition field="${caller_id_name}" expression="Giovanni" break="on-false">
        <action application="log" data="ERR 08 I'm the second..."/>
        <action application="log" data="ERR 09 I'm the second CIDname is ${caller_id_name}"/>
        <action application="set" data="var_04=04" inline="true"/>
        <action application="log" data="ERR 10 I'm the second ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
      </condition>
      <action application="log" data="ERR 11 I'm after..."/>
      <action application="set" data="var_05=05" inline="true"/>
      <action application="log" data="ERR 12 I'm after ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
    </condition>
  </condition>
  <action application="log" data="ERR 06 I'm in between..."/>
  <action application="set" data="var_03=03" inline="true"/>
  <action application="log" data="ERR 07 I'm in between ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
  <condition field="${caller_id_name}" expression="Giovanni" break="on-false">
    <action application="log" data="ERR 08 I'm the second..."/>
    <action application="log" data="ERR 09 I'm the second CIDname is ${caller_id_name}"/>
    <action application="set" data="var_04=04" inline="true"/>
    <action application="log" data="ERR 10 I'm the second ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
    <condition field="${caller_id_name}" expression="Giovanni" break="on-false">
      <action application="log" data="ERR 08 I'm the second..."/>
      <action application="log" data="ERR 09 I'm the second CIDname is ${caller_id_name}"/>
      <action application="set" data="var_04=04" inline="true"/>
      <action application="log" data="ERR 10 I'm the second ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
      <condition field="${caller_id_name}" expression="Giovanni" break="on-false">
        <action application="log" data="ERR 08 I'm the second..."/>
        <action application="log" data="ERR 09 I'm the second CIDname is ${caller_id_name}"/>
        <action application="set" data="var_04=04" inline="true"/>
        <action application="log" data="ERR 10 I'm the second ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
      </condition>
      <action application="log" data="ERR 11 I'm after..."/>
      <action application="set" data="var_05=05" inline="true"/>
      <action application="log" data="ERR 12 I'm after ${var_01} ${var_02} ${var_03} ${var_04} ${var_05}"/>
    </condition>
  </condition>
  <extension name="call_has_not_stopped_before_here">
    <condition field="destination_number" expression=".*">
      <action application="log" data="ERR NOT STOPPED BEFORE HERE"/>
    </condition>
  </extension>
</extension>
```

There are a lot of things you can change to test this example:

- Obviously, all the "expression": 2901, 1011, Giovanni
- in first condition, require-nested can be set to "true" or "false"
- the "inline" parameter of the "set" actions can be set to "true" or "false"
- the "break" parameter of the nested conditions can be set to "on-false", "on-true", "never", "always"
Depending on how you nest conditions within conditions, you will get a nested order result, or a line order result. Please observe the following two examples and the execution order of each.

**Line order vs nested order during execution**

```xml
<extension name="nested example">

  <condition field="destination_number" expression="^29015$" require-nested="false">
    <action application="log" data="ERR 1 Nest Level 1a"/>
    <action application="log" data="ERR 2 Nest Level 1b"/>
    <condition field="direction" expression="inbound" break="never">
      <action application="log" data="ERR 3 Nest Level 2a"/>
      <action application="log" data="ERR 4 Nest Level 2b"/>
      <condition field="direction" expression="inbound" break="never">
        <action application="log" data="ERR 5 Nest Level 3a"/>
        <action application="log" data="ERR 6 Nest Level 3b"/>
      </condition>
    </condition>
    <action application="log" data="ERR 7 Nest Level 2c"/>
    <action application="log" data="ERR 8 Nest Level 2d"/>
  </condition>

  <action application="log" data="ERR 9 Nest Level 1c"/>
  <action application="log" data="ERR 10 Nest Level 1d"/>
</extension>

EXECUTE verto.rtc/2901 log(ERR 1 Nest Level 1a)
EXECUTE verto.rtc/2901 log(ERR 2 Nest Level 1b)
EXECUTE verto.rtc/2901 log(ERR 9 Nest Level 1c)
EXECUTE verto.rtc/2901 log(ERR 10 Nest Level 1d)
EXECUTE verto.rtc/2901 log(ERR 3 Nest Level 2a)
EXECUTE verto.rtc/2901 log(ERR 4 Nest Level 2b)
EXECUTE verto.rtc/2901 log(ERR 7 Nest Level 2c)
EXECUTE verto.rtc/2901 log(ERR 8 Nest Level 2d)
EXECUTE verto.rtc/2901 log(ERR 5 Nest Level 3a)
EXECUTE verto.rtc/2901 log(ERR 6 Nest Level 3b)
```
Here is a more complex example, using the `break="on-true"` and `break="on-false"` statements to perform time-based routing for a support organization. The user dials extension 1100. The actual support extension is 1105 and is staffed every day from 8am to 10pm, except Friday, when it is staffed between 8am and 1pm. At all other times, calls to 1100 are sent to the support “after-hours” mailbox.

```xml
<extension name="Time-of-day">
  <condition field="destination_number" expression="^1100$" break="on-false"/>
    <!--Don’t bother evaluating the next condition set if this is true.-->  
  <condition wday="6" hour="8-12" break="on-true"/>  
    <!--Fri, 8am-12:59pm-->  
    <action application="transfer" data="1105 XML default"/>
  </condition>
  <condition wday="1-5" hour="8-21" break="on-true"/>
    <!--Sunday-Thursday, 8am-9:59pm-->  
    <action application="transfer" data="1105 XML default"/>
  </condition>
  <condition>
    <!--this is a catch all, sending the call to voicemail at all other times.-->  
    <action application="voicemail" data="default ${domain} 1105"/>
  </condition>
</extension>
```
**Thing to Notice**

The break="on-true" statement causes the extension to halt processing when the condition its used in matches. In the example above, if the call arrives in the designated support hours, the call is transferred and the rest of the conditions are not processed.

By default, break="on-false" behavior is assumed if it is not specified.

**break="never"**

In this example, we use the break="never" statement to cause the first condition to always "fall through" to the next condition even if the first condition is false. This is useful to set various flags as part of extension processing. This example sets the variable begins_with_one if the destination number begins with 1.

```xml
<extension name="break-demo">
  <condition field="destination_number" expression="^1\d+" break="never">
    <action application="set" data="begins_with_one=true"/>
  </condition>
  <condition field="destination_number" expression="^\d+$">
    ...other actions that may query begins_with_one...
  </condition>
</extension>
```

**Asterisk Pattern Matching**

In addition to PCRE FreeSWITCH also supports Asterisk patterns. Any expression starting with an underscore (_) will be processed using asterisk pattern matching. See mod_dialplan_asterisk.

Since * is normally reserved in regular expressions, its use in Asterisk patterns must be escaped with \, as shown in the second part of the example below.

```xml
<extension name="US-Domestic">
  <condition field="destination_number" expression="_NXXXXXXXXX">
    <action application="bridge" data="sofia/internal/$1@example.com"/>
  </condition>
</extension>

<extension name="star-code-using-escape">
  <condition field="destination_number" expression="_\*XX(.*)">
    <action application="log" data="ERR captured $1 ~~~ $2"/>
    <action application="answer"/>
    <action application="playback" data="tone_stream://path=${base_dir}/conf/tetris.ttml;loops=10"/>
  </condition>
</extension>
```

**Example Library**

This example shows how to ensure that all expressions match before executing actions, otherwise the anti-actions will be executed. In this case, the SIP gateway must be the default provider, and it must be an emergency call, and the auto-answer option must be enabled and stored in the database:
Caller Profile Fields vs. Channel Variables

One thing that may seem confusing is the distinction between a **caller profile field** (the built-in variables) and a channel variable.

Caller profile fields are accessed like this:

```xml
<condition field="destination_number" attributes...>
```

While channel variables are accessed like this:

```xml
<condition field="${sip_has_crypto}" attributes...>
```

Please take note of the `$variable_name` syntax. Channel variables may also be used in action statements.

In addition, API functions can be called from inside a condition statement to provide dynamic data.

For example, you can use the **cond** API:

```xml
<condition field="$\{cond($my_var > 12 ? YES : NO)\}" expression="^YES$">
    <action application="log" data="INFO $\{my_var\} is indeed greater than 12"/>
</condition>
```

This example tests `$\{my_var\}`. If it is more than 12, "YES" is returned. Otherwise "NO" is returned. The condition tests the results for "YES" and logs the resulting message to the FreeSWITCH log.

Availability of Variables

*Asterisk users must read!*
The XML Dialplan has the ability to test a number of conditions based upon variables with expressions; however, it needs to be understood that some variables may not be available for conditional testing until the first transfer or execute_extension is performed (see workarounds below).

Why

In essence the XML Dialplan is to be used for Call Routing rather than for complex or extensive conditional tests and evaluations. This is why FreeSWITCH makes Lua, JavaScript, Perl, Python and other APIs available since they are far better alternatives than coming up with a convoluted XML solution, or worse yet some arcane and convoluted acronym such as "AEL".

This may be confusing to former Asterisk users since the info application such as <action application="info"/> will in fact display the variables as if they are available for a conditional test when in fact they may not.

The reason for this is that FreeSWITCH does the hunting and the executing in two separate steps. First - based on conditions, actions and anti-actions - all applications that need to be executed are gathered. Second, that sequence of applications is executed. This means that channel variables set by the executed applications won't be available to conditions at hunting time.

This is why you may find that your XML condition is failing even though the variable and its value are displayed with the <action application="info"/>.

Workarounds

The workaround for this is to either implement the vast majority of your dialplan logic within Lua, JavaScript or one of the other Dialplan scripting languages, OR execute an extension which will make those variables you seek to do conditional evaluations on available for parsing within your XML Dialplan condition.

NOTE: Since f21b4a21374 it is possible for certain applications to be run inline. This means that they are executed at hunting time which has the effect that channel variables set by these applications are available to the following conditions at hunting time.

Actions and Anti-Actions

So far, we've seen example dialplan entries that contain conditions along with the actions that run when the conditions match.

You can also specify 'anti-actions' that run if the conditions for the extension 'are not met'.

In this example, the value of ${my_var} is compared with 12, and a message is logged for either result.

```
<condition field="${cond(${my_var} > 12 ? YES : NO})" expression="^YES$">
    <action application="log" data="INFO ${my_var} is indeed greater than 12"/>
    <anti-action application="log" data="INFO ${my_var} is not greater than 12"/>
</condition>
```

Available Actions

See Modules and dialplan functions

Inline Actions

You may set an extra attribute inline="true" on an action so that it will be executed during the hunting phase of the dialplan:

```
<action inline="true" application="set" data="some_var=some_val"/>
```

This makes it possible to have a condition in the following extension, that matches on the ${some_var} field.

Note that the only applications that may be run inline are the ones that quickly get- or set some variable(s) and that don't access or modify the state of the current session.

Applications that are allowed to be run inline are:

check_acl
Also keep in mind that inline executed applications don't show up in your call detail records like normally run applications do.

**Complete Syntax**

```xml
<!-- For enumerated attributes the first value on the list is the default -->
<extension name="unique_extension_name" continue="[false|true]">
  <condition field="[field_name|${variable_name}|${api_func(api_args ${var_name})}]" expression="regular expression" break="[on-false|on-true|always|never]" require-nested="[true|false]">
  </condition>
  <condition ...><!-- Conditions can be nested with great care --> ...
  </condition>
  <action application="app name" data="app arg"/>
  <anti-action application="app name" data="app arg"/>
</extension>
```

**Summary**

Extensions hunting stops at the first extension where conditions evaluate to true and continue="false" (the default).

Conditions evaluate to true if the last checked condition is true.

Conditions are checked depth-first until:

- break="on-false" and the condition is false
- or break="on-true" and the condition is true
- or break="always".
The condition is true if expression matches and either there are no nested conditions or require-nested="false" or nested conditions evaluate to true.

If condition is true, its <action>s are collected for execution, otherwise its <anti-action>s are collected for execution.

If require-nested="true" (the default) and nested conditions evaluate to false, no <anti-action>s are collected. (Is this intentional?)

Other Dialplan Pearls of Wisdom

The dialplan is parsed once when the call hits the dialplan parser in the ROUTING state. With one pass across the XML the result will be a complete list of instructions installed into the channel based on parsed <action> or <anti-action> tags.

Those accustomed to Asterisk may expect the call to follow the dialplan by executing the applications as it parses them allowing data obtained from one action to influence the next action. This is not the case with the exception being the ${api func(api arg ${var_name})} field type where a pluggable api call from a module may be executed as the parsing occurs. This is meant to be used to draw real-time information such as date and time or other quickly accessible information and should not be abused.

Auto Hunt

You may turn on auto_hunt and then if the Extension name precisely equals the dialed number, FreeSWITCH will jump to this extension to begin the searching. It may or may not match the conditions, though.

Dialing through gateways

"gateway" is treated as a keyword by mod_sofia, it obviously means the call will be placed through a configured gateway. This is an exception for the pattern sofia/profilename/extension@ip-address.

If a gateway, for instance, is named "gw", the bridge string for sending a call to gw's extension 100 would be:

```xml
<extension name="testing">
    <condition field="destination_number" expression="^(100)$">
        <action application="bridge" data="sofia/gateway/gw/$1"/>
    </condition>
</extension>
```

destination_number is a FreeSWITCH variable; it shouldn't be changed.

Examples

**NOTE:** if you plan to include your extension in a separated .XML file:

please disable or change enum extension if you don't need it

add the tag <include> and close it with </include>

**Example 1: Matching a condition**

The incoming call will be bridged only if it comes from 192.168.1.1. If it does, the destination number will be captured in $1, and the call will be bridged to the same number, at 192.168.2.2.

This is a bizarre example. We can do better.
Note that this example is not the same as doing this:

```xml
<extension name="Test1Wrong">
  <condition field="destination_number" expression="^\d+$"/>
  <condition field="network_addr" expression="^192\.168\.1\.1$">
    <action application="bridge" data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

The call will not be routed properly because the captured destination number in $1 is not available outside of the condition that created it. This is a peculiarity of how captured values work. You can work around this by storing the captured value in a standard variable, like this:

```xml
<extension name="Test1_2">
  <condition field="destination_number" expression="^\d+$">
    <action application="set" data="dialed_number=$1"/>
  </condition>
  <condition field="network_addr" expression="^192\.168\.1\.1$">
    <action application="bridge" data="sofia/profilename/${dialed_number}@192.168.2.2"/>
  </condition>
</extension>
```

**Example 2: Matching multiple conditions (AND)**

In this example we need to match a called number beginning with the prefix 1 AND match the incoming IP address at the same time.

```xml
<extension name="Test2">
  <condition field="network_addr" expression="^192\.168\.1\.1$"/>
  <condition field="destination_number" expression="^1\d+$">
    <action application="bridge" data="sofia/profilename/$0@192.168.2.2"/>
  </condition>
</extension>
```

Notice that although we match with the rule 1\d+$ we don't use the variable $1 which would contain only the rest of the dialed number with the leading 1 stripped off, we use the variable $0 which contains the original destination number.

**Example 3: Stripping leading digits**

In this example we need to match a called number beginning with 00 but we also need to strip the leading digits. Assuming that FreeSWITCH™ receives the number 00123456789 and we need to strip the leading 00 digits, then we can use the following extension:

```xml
<extension name="Test3.1">
  <condition field="destination_number" expression="^00\d+$">
    <action application="bridge" data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

If you anticipate receiving non-digits, or you want to match on more than just digits, you can use ".+" instead of \d+.

⚠️ Exercise caution when using regular expressions containing (.+) or \d+\d+. The more specific the regular expression, the less chance that your system can be exploited by an unauthorized user.
Example 4: Adding a prefix

In this example, numbers beginning with 0 are transformed into the same number, only beginning with 011 instead. If FreeSWITCH™ receives 00123456789, we should dial 011123456789:

```
<extension name="Test4">
  <condition field="destination_number" expression="^00(\d+)$">
    <action application="bridge" data="sofia/profilename/011$1@x.x.x.x"/>
  </condition>
</extension>
```

Example 5: SIP Profiles (dialing with different configurations)

In this example we will demonstrate the use of profiles when using a FreeSWITCH endpoint that supports profiles, like mod_sofia. Assuming that we want to use different call settings (codecs, DTMF modes, etc) for sending the calls to different IP addresses, we can create different profiles. For example, in the configuration of sofia.conf, we see an example profile named "test", which we rename to profile1 for this example, and add a profile2 for comparison:

```
<profile name="profile1">
  <param name="debug" value="1"/>
  <param name="rfc2833-pt" value="101"/>
  <param name="sip-port" value="5060"/>
  <param name="dialplan" value="XML"/>
  <param name="dtmf-duration" value="100"/>
  <param name="codec-prefs" value="PCMU@20i"/>
  <param name="codec-ms" value="20"/>
  <param name="use-rtp-timer" value="true"/>
</profile>
<profile name="profile2">
  <param name="debug" value="1"/>
  <param name="rfc2833-pt" value="101"/>
  <param name="sip-port" value="5070"/>
  <param name="dialplan" value="XML"/>
  <param name="dtmf-duration" value="100"/>
  <param name="codec-prefs" value="PCMA@20i"/>
  <param name="codec-ms" value="20"/>
  <param name="use-rtp-timer" value="true"/>
</profile>
```

The difference between the two profiles are in the codecs. The first uses G.711 u-law and the second G.711 A-law.

Continuing the examples above, we have:

```
<extension name="Test5ulaw">
  <condition field="network_addr" expression="^192.168.1.10$"/>
  <condition field="destination_number" expression="^1(\d+)$">
    <action application="bridge" data="sofia/profile1/$0@192.168.2.2"/>
  </condition>
</extension>
```

to send the call in G.711 uLaw and

```
<extension name="Test5alaw">
  <condition field="network_addr" expression="^192.168.1.10$"/>
  <condition field="destination_number" expression="^1(\d+)$">
    <action application="bridge" data="sofia/profile2/$0@192.168.2.2"/>
  </condition>
</extension>
```

Example 6: Calling registered user
This example shows how to bridge to devices that have registered with your FreeSWITCH box. In this example we assume that you have setup a sofia profile called 'local_profile' and your phones are registering with the domain example.com.

The % instead of @ in the dial string indicates an endpoint that is registered to FreeSWITCH.

Example 7: Action failover on failed action

The following example shows how it is possible to call another action if the first action fails.

If the first action is successful the call is bridged to 1111@example1.company.com and will exist until one of the parties hangs up. After this, no other processing will be done because the caller's channel is closed. (I.e. 1111@example2.company.com is not called)

If the initial call to 1111@example1.company.com was not successful the channel will not be closed and the second action will be called.

Note: If you have more than one action and the application of the first action

DOES hangup the channel, the second action will NOT be called.

DOES NOT hangup the channel, the second action will be called.

Example 8: Check user is authenticated

The following example requires that a caller be authenticated before passing through. It was yanked from a mailing list post.

Example 9: Routing DID to an extension

To route incoming calls which come in to a certain DID to a fixed extension 1001, do something LIKE the following (from a mailing list post) (where XXXxxxxxxx is the phone number of your incoming DID)
In **public.xml**:

```xml
<extension name="test_did">
    <condition field="destination_number" expression="^(XXXxxxxxxx)$">
        <action application="transfer" data="$1 XML default"/>
    </condition>
</extension>
```

and then in **default.xml** have something like this in the default context:

```xml
<extension name="Local_Extension">
    <condition field="destination_number" expression="^(XXXxxxxxxx)$">
        <action application="set" data="dialed_ext=$1"/>
        <condition field="destination_number" expression="^${caller_id_number}$">
            <action application="set" data="voicemail_authorized=${sip_authorized}"/>
            <action application="answer"/>
            <action application="sleep" data="1000"/>
            <action application="voicemail" data="check default $${domain} $${dialed_ext}"/>
            <anti-action application="ring_ready"/>
            <anti-action application="set" data="call_timeout=10"/>
            <anti-action application="set" data="hangup_after_bridge=true"/>
            <anti-action application="set" data="continue_on_fail=true"/>
            <anti-action application="bridge" data="USER/1001@$${domain}"/>
            <anti-action application="answer"/>
            <anti-action application="sleep" data="1000"/>
            <anti-action application="voicemail" data="default $${domain} $${dialed_ext}"/>
        </condition>
    </condition>
</extension>
```

(1001 in the "bridge" line is the extension we're ringing)

FYI, calls from the "public" go into the public context where they then need to be transferred to another more friendly context for processing, like "default". That is why you add the entry to public and the 'data="$1 XML default"' says to transfer called number $1 to the context "default" using XML dialplan. In the "default" context is where the call is actually bridged to the desired phone.

$$ ${domain} a variable that is set in vars.xml to the domain, most likely your IP or the host name.

**Example 10: Route to a gateway extension with custom caller id**

In this example we demonstrate an outgoing call with 10 digits from extension 1000 then route it to the asterlink.com gateway. This examples shows how to route for a specific extension and allows custom caller id for that extension.

```xml
<extension name="asterlink.com">
    <condition field="caller_id_number" expression="^1000$"/>
    <condition field="destination_number" expression="^\d{10}$">
        <action application="set" data="effective_caller_id_number=8001231234"/>
        <action application="set" data="effective_caller_id_name=800 Number"/>
        <action application="bridge" data="sofia/gateway/asterlink.com/1208$1"/>
    </condition>
</extension>
```

**Example 11: Route based on number prefix**

In this example we demonstrate routing to different destination based on NPANXX. Also how to respond to the calling party with a different failure message than the destination sends to FreeSWITCH.
Example 12: Handle calls which match no extension

In this example we demonstrate how to catch invalid extensions/Destinations.

You need to add this extension at the bottom of your dialplan before ENUM can get included.

See mod_enum.

```xml
<extension name="catchall">
  <condition field="destination_number" expression=".*" continue="true">
    <action application="playback" data="misc/invalid_extension.wav"/>
  </condition>
</extension>
```

Example 13: Call Screening

In this example, we ask the caller for a name, connect to the called party and announce that name. The called party may then press 1 to accept the call, or hang up. If the called party hangs up, the caller is connected with voicemail.
Example 14: Media recording

This extension is used to play/record media in audio (wav) format recording / playback extension

Thanks to rupa for the help.

Example 15: Speaking Clock

This example will speak time using the Flite text to speech engine.

See: mod_flite
Example 16: Block certain codes

This extension example is to demonstrate how to block certain NPAs that you do not want to terminate based on caller id area codes and respond with SIP: 503 to your origination so that they can route advance if they have another carrier to terminate the call.

```xml
<extension name="blocked_cid_npa">
  <condition field="caller_id_number" expression="^\+1?((876|809)\d{7})$">
    <action application="respond" data="503"/>
    <action application="hangup"/>
  </condition>
</extension>
```

Example 17: Receive fax from inbound did

To use the predefined fax_receive extension in freeswitch/conf/dialplan/default.xml for inbound calls, put this in freeswitch/conf/dialplan/public/fax.xml:

```xml
<include>
  <extension name="incoming-fax">
    <condition field="destination_number" expression="^$${local_fax_number}$">
      <action application="set" data="domain_name=$${domain}"/>
      <action application="transfer" data="9178 XML default"/>
    </condition>
  </extension>
</include>
```

Then in freeswitch/conf/vars.xml you set your fax number to 1234 or whatever:

```xml
<X-PRE-PROCESS cmd="set" data="local_fax_number=1234"/>
```

Example 18: Add international call prefix to effective_caller_id_number on incoming BRI calls

When using FreeTDM with zaphfc on a BRI line, the incoming calls received will not contain the international call prefix in the caller_id_number. This extension adds it to the effective_caller_id_number.


In Germany, the international call prefix is called "Verkehrsausscheidungsziffer" (VAZ), see [http://de.wikipedia.org/wiki/Verkehrsausscheidungsziffer](http://de.wikipedia.org/wiki/Verkehrsausscheidungsziffer).

The international call prefix is never transmitted. It can be predicted by looking at the ToN information.

```xml
<extension name="Add-VAZ" continue="true">
  <!-- On incoming BRI calls, the Verkehrsausscheidungsziffer (VAZ) is dropped. This extension adds it again to the caller_id_number. TODO: add support for international numbers -->
  <condition field="source" expression="^mod_freetdm$">
    <action application="set" data="effective_caller_id_number=0$\{caller_id_number\}"/>
  </condition>
</extension>
```

Example 19: DISA

Be able to dial into FS box and get a dialtone to dial again, just like in Asterisk's DISA()
In FS/conf/dialplan/public/*.xml

```xml
<extension name="incoming-bri-wor">
  <condition field="destination_number" expression="^(disa_target)$">
    <action application="answer"/>
    <action application="start_dtmf"/>
    <action application="play_and_get_digits" data="2 5 3 37000 # ${base_dir}/sounds/en/us/callie/ivr/8000/ivr-please_enter_pin_followed_by_pound.wav ${base_dir}/sounds/en/us/callie/ivr/8000/ivr-pin_or_extension_is_invalid.wav digits ^${DISA_PASSWORD}$"/>
    <action application="transfer" data="$1 XML default"/>
  </condition>
</extension>
```

In FS/conf/dialplan/default/03_DISA.xml

```xml
<!-- DISA - allow to dial into the box and get a dialtone like new trunk -->
<include>
  <extension name="DISA for FS">
    <condition field="destination_number" expression="^(disa_target)$">
      <action application="answer"/>
      <action application="read" data="2 15 'tone_stream://%(10000,0,350,440)' digits 30000 #"/>
      <action application="execute_extension" data="${digits}"/>
      <action application="transfer" data="disa_target XML default"/>
    </condition>
  </extension>
</include>
```

Please replace "disa_target" to your extension number.

You can use this technique to dial into your box from a pstn / mobile phone and get a dialtone to do anything with.

**Example 20: Fix invalid caller ID**

If you run into a problem where your B-Leg do not like the invalid caller ID: for instance, you have an INVITE message with From: header as From: <sip: Unavailable@Unavailable.invalid:5060> You can force the invalid number to be replaced by a fixed caller ID number. The following example checks for a valid NANPA CLID:
Example 21: Block outbound caller ID
To have caller ID block for calling party by dialing "*67 follows by the dial number, you can do the following:

```xml
<extension name="block_caller_id">
  <condition field="destination_number" expression="^\*67(\d+)$">
    <action application="privacy" data="full"/>
    <action application="set" data="sip_h_Privacy=id"/>
    <action application="set" data="privacy=yes"/>
    <action application="transfer" data="$1 XML default"/>
  </condition>
</extension>
```

Example 22: Play MOH while doing a database lookup
If you want to play MOH while doing a data dip that takes a long time, the non-ESL way to do this by making the dialplan use the FSAPI via variable expansion to call luarun on the script. This way a new thread will be launched to execute the Lua script.

```xml
<extension name="Get_Data">
  <condition field="destination_number" expression="^(Get_Data)$">
    <action application="play_and_get_digits" data="4 16 3 7000 # phrase:Enter_Case_Number phrase:Invalid_entry case_number \d+"/>
    <action application="set" data="x=${expand(luarun GetDataFromLDAP.lua $case_number ${uuid})}"/>
    <action application="playback" data="/tmp/LongMusicFile.wav"/>
  </condition>
</extension>
```

In the Lua script you can use the "uuid" argument passed, to break the MOH or transfer to another extension

```lua
--GetDataFromLDAP.lua
key = argv[1] sessionId = argv[2] api = freeswitch.API() -- Do your data dips here -- Once the database
operations are done you can simply stop the MOH or transfer to another extension
api:execute("uuid_break", sessionId) -- break the MOH
```

SIP-Specific Dialstrings
SIP dialing has several options. Here are some aspects of what you might call the anatomy of a SIP dialstring.

Dialing A SIP URI
Basic syntax is: sofia/my_profile/user@host Host can be a name or an IP address, for example:

```
sofia/my_profile/12340192.168.0.1
```
This would dial 1234 at host 192.168.0.1 via the profile "my_profile". If you use a name instead of an IP address, Sofia will try to resolve the name as a NAPTR or SRV record before trying it as a standard A record.

**Dialing A Registered User**

There are two options depending upon whether or not there is an alias for the domain. Without an alias you can do this:

```
sofia/my_profile/1234@mydomain.com
```

If you have an alias for the domain then this syntax is valid:

```
sofia/mydomain.com/1234
```

Note how the profile does not need to be explicitly supplied in the dialstring.

Also you can do it this way for users defined in the directory:

```
user/1234@mydomain.com
```

**Dialing Through A Gateway (SIP Provider)**

A gateway is a means for making outbound calls through a SIP provider. For example:

```
sofia/gateway/mygateway.com/1234
```

This will dial through the gateway named mygateway.com to user 1234.

Note how there is no need to append anything after the user "1234"
This is an example of how NOT to do it:

```
sofia/gateway/mygateway.com/1234@mygateway.com <==== WRONG WRONG WRONG
```

**Dialing With A Specific Transport**

Sometimes you will need to specify the transport, for example TCP, UDP, TLS, or SCTP. This can be done by appending a semicolon and the transport method. For example:

```
sofia/my_profile/1234@192.168.0.1;transport=tcp
```

**Specifying The Codec**

Occasionally you may want to force the system to use a specific codec. This syntax will accomplish that:

```
{absolute_codec_string=XXXX}sofia/my_profile/user@your.domain.com
```
In this example, XXXX represents the codec to be used. The possible codec values are listed [here](#).

Additional dialstring examples from [Absolute Codec String variable](#).

### Getting Fancy With PortAudio

If you have PortAudio running and would like to specify the codec you need to originate first and bridge second:

```plaintext
originate (absolute_codec_string=XXXX)sofia/default/foo@bar.com bridge:portaudio/auto_answer inline
```

### Changing the SIP Contact user

FreeSWITCH normally uses mod_sofia@ip:port for the internal SIP contact. To change this to foo@ip:port, there is a variable, sip_contact_user:

```plaintext
{sip_contact_user=foo}sofia/my_profile/12340192.168.0.1;transport=tcp
```

### Using a Custom SIP URI

FreeSWITCH allows you to specify custom URI's as needed. For example, you may need to interoperate with equipment that accepts a URI only if it is formatted in a particular way. The key is to prefix your SIP URI with "sip:" in the dialstring. For example:

```plaintext
sofia/my_profile/sip:xxxx;phone-context=cdp.udp@somedomain.com;user=phone
```

The above example will send the URI exactly as specified after the "sip:" prefix.

### Testing the dialplan with a command line

```plaintext
originate loopback/<destination number>/<mycontext> hangup inline
```

- Note: You can also set your variables to match your dialplan requirement. See below example:

```plaintext
originate (toll_allow=international)loopback/0116628888888/default hangup inline
```

### Setting up SIP Diversion Header for call forward

```plaintext
<action application="export" data="sip_h_Diversion=<sip:2134445555@1.2.3.4>;reason=unavailable"/>
```