



Stefan Wintermeyer
Philipp Kempgen

Asterisk 1.4+1.6

Installation, Programmierung und Betrieb



3 Dialplan – die Grundlagen

In diesem Kapitel beschäftigen wir uns mit den Grundlagen des Dialplans. Wir benutzen das englische Wort *Dialplan* und nicht das deutsche Wort *Wählplan*, weil es auch Asterisk-intern immer wieder benutzt wird.¹

Im Verzeichnis `/etc/asterisk` gibt es zwei für den Dialplan wichtige Dateien: zum einen die `extensions.conf`, die den Dialplan noch in dem für die Asterisk-Versionen 1.2, 1.4 und 1.6 von Digium empfohlenen Prioritäten-Modell abbildet, und zum anderen die `extensions.ael`, die einen neuen Weg zur Beschreibung des Dialplans geht. AEL wird in einem gesonderten Kapitel in diesem Buch besprochen (siehe Kapitel 7, *AEL (Asterisk Extension Language)*). Wir beschäftigen uns hier mit dem traditionellen Prioritäten-Modell, weil auch in der Version 1.4 und 1.6 bei jedem Starten von Asterisk die `extensions.ael` automatisch (intern) in die alte Schreibweise konvertiert wird.



Hinweis

Auf die Frage »Soll ich lieber die `.conf` oder die `.ael` benutzen?« gibt es aktuell von Digium die Antwort: »`.conf` ist der Default-Weg.«

3.1 Context

Der Dialplan wird in verschiedene Abschnitte unterteilt. Diese Abschnitte heißen im Asterisk-Jargon »Contexte«. Am Anfang eines Dialplans muss es immer einen `[general]`-Context für allgemeine Konfigurationen geben, die danach folgenden Contexte können beliebig benannt werden. Die Contexte bilden das Verbindungsstück zwischen der Definition eines Telefons (z. B. SIP oder ISDN) und dem Dialplan. Für ein Telefon wird immer ein Context definiert. Hier sehen Sie ein Beispiel aus einer `sip.conf`:

¹ Zur Definition des Begriffes *Dialplan* gibt es in der Telefoniewelt immer wieder beinahe religiös anmutende Diskussionen. In diesem Buch wird das Wort *Dialplan* in der im Asterisk-Umfeld üblichen Bedeutung verwendet.

```
[2000]
type=friend
context=interne-telefone
secret=1234
host=dynamic
```

Das SIP-Telefon mit der Nummer 2000 ruft in dieser Konfiguration immer den Context `interne-telefone` auf. Wenn also ein Benutzer mit dem Telefon 2000 eine bestimmte Nummer wählt, dann sucht Asterisk im Context `interne-telefone` nach der entsprechenden Extension (also der dazu passenden Regel). Ist diese Extension nicht vorhanden, passiert auch nichts.



Achtung

Zu verstehen, wie man Contexte benutzt, ist essenziell für Programmierung und Administration einer Asterisk-Telefonanlage. Es ist für Anfänger nicht immer ersichtlich, warum die korrekte Benutzung von Contexten so wichtig ist. Sollten Sie sich jetzt nicht ganz sicher sein, dann gehen Sie bitte das Beispiel für eine minimale Telefonanlage, das in Kapitel 2, »Hello World«, beschrieben ist, Schritt für Schritt durch.

Über diese Hürde stolpert am Anfang fast jeder, und es ist sinnvoll, sich die Zeit zu nehmen, die man braucht, um das Konzept der Contexte zu verinnerlichen!

3.1.1 Syntax

Ein Context selbst wird durch Text in eckigen Klammern eingeleitet. »Text« ist hierbei ein sinnvoller Name, der den Context benennt und als spätere Referenz für denselben verwendet wird. Alle Zeilen nach einer solchen Einleitung bis zum nächsten Context werden als Bestandteil (Regeln, Anweisungen) dieses Contextes behandelt:

```
[general]
```

```
[interne-telefone]
Regeln, Anweisungen, ...
```

```
[apfelmus]
Regeln, Anweisungen, ...
```

3.2 Extension

Die einzelnen Dialplan-Programme werden Asterisk-intern Extensions genannt. Eine Extension wird nicht kompiliert, sondern bei jedem Durchlauf von Asterisk interpretiert. Das Einlesen erfolgt einmalig automatisch während des Startens des

Asterisk-Daemons.² Das erneute Einlesen des Dialplans kann aber auch im laufenden Betrieb im CLI (Command Line Interface) durch den Befehl `reload now` bzw. `dialplan reload` forciert werden.

3.2.1 Syntax

Eine Extension besteht immer aus folgenden Teilen:

- Extension (Nummer oder Name)
- Priorität (also der Programmzeilenzähler)
- Applikation – das ist die Anweisung, die Asterisk ausführen soll.

`exten => Extension,Priorität,Applikation()`

z. B.:

`exten => 123,1,Answer()`



Achtung

Die erste Priorität in einer Extension muss immer eine 1 (eins) sein. Ansonsten wird Asterisk diese Extension nicht aufrufen. Die nächsten Prioritäten müssen danach immer um +1 erhöht werden. Größere Sprünge werden von Asterisk nicht erkannt.

3.2.2 Grundlegende Applikationen

Um die Programmierbeispiele in diesem Kapitel halbwegs sinnvoll zu gestalten³, benötigen wir folgende Applikationen (alle diese Applikationen werden später noch in Anhang C, *Applikationen im Dialplan*, genau erklärt):

- `Answer()`

Die `Answer()`-Applikation dient dazu, einen Verbindungsversuch zu akzeptieren. Wenn ein Channel klingelt, dann kann `Answer()` den virtuellen Hörer abnehmen (siehe auch Abschnitt C.10, *Answer()*).

- 2 Eine Ausnahme stellt hier die *Asterisk RealTime Architecture* (ARA) dar. In einem ARA-System wird der Dialplan in einer Datenbank (z. B. MySQL) abgespeichert und dort von Asterisk bei jedem Anruf neu eingelesen (also nicht nur einmal beim Starten von Asterisk). So können Dialpläne auch im laufenden Betrieb ständig geändert werden. Allerdings hat diese Variante viele Nachteile. Nähere Informationen zu ARA finden Sie unter <http://www.voip-info.org/wiki/view/Asterisk+RealTime>.
- 3 Ein typisches Henne-Ei-Problem. Man kann eine Applikation nur verstehen, wenn man die Programmierung eines Dialplans versteht, und umgekehrt.

- `Hangup()`

`Hangup()` ist das Gegenstück zu `Answer()`. Die Verbindung wird getrennt, und der virtuelle Hörer wird aufgelegt (siehe auch Abschnitt C.68, *Hangup()*).
- `Playback(Soundfile)`

Mit `Playback()` kann man Sounddateien abspielen. Diese finden sich, wenn kein anderes Verzeichnis angegeben worden ist, im Verzeichnis `/var/lib/asterisk/sounds/`. Die Dateierdung wird dabei nicht angegeben (Asterisk sucht sich den optimalen Codec selbstständig heraus; siehe auch Abschnitt C.117, *Playback()*).
- `Wait(Zahl)`

Mit `Wait()` kann man eine Pause abrufen. Die Zahl in der Klammer gibt die Anzahl der zu wartenden Sekunden an (siehe auch Abschnitt C.188, *Wait()*).
- `NoOp(String)`

Die Applikation `NoOp()` macht nichts. »NoOp« steht für *No Operation*. Sie ist aber ein praktisches Tool, um Dialpläne zu debuggen. Der Inhalt des übergebenen Strings wird auf dem CLI ausgegeben. Im CLI muss dafür aber der Verbose-Level auf mindestens 3 eingestellt sein. (Geben Sie einfach im CLI `core set verbose 3` ein; siehe auch Abschnitt C.107, *NoOp()*.)
- `VoiceMail(Voicemailbox,u)`

Die Applikation `VoiceMail()` gibt dem Anrufer die Möglichkeit, eine Sprachnachricht auf der Voicemailbox zu hinterlassen, die als erster Parameter bestimmt wird (siehe auch Abschnitt C.186, *VoiceMail()*).
- `VoiceMailMain()`

Die Applikation `VoiceMailMain()` gibt dem Anrufer Zugang zum Voicemail-System. Wer über eine Voicemailbox verfügt, kann diese dort abhören (siehe auch Abschnitt C.187, *VoiceMailMain()*).

3.2.3 Priorität

Eine typische Extension besteht aus mehreren Schritten. Damit Asterisk diese Schritte in der richtigen Reihenfolge ausführen kann, ist eine Art Zähler nötig. Das erinnert ein wenig an frühe BASIC-Programme, die auch am Anfang einer jeden Zeile einen solchen Zähler hatten. Dieser Zähler heißt bei Asterisk *Priorität*. Prioritäten werden der Reihenfolge nach abgearbeitet (es wird immer +1 gezählt). Wenn die nächste logische Priorität (Lücken sind nicht zulässig!) nicht definiert ist, bricht Asterisk ab – gibt aber leider keine Fehlermeldung auf dem CLI aus.

Ein hello-world-Beispiel

Die folgende Extension wird immer ausgelöst, wenn ein Telefon mit dem Context `apfelmus` die Nummer 8888 anruft. Asterisk nimmt dann ab, spielt den Sprachbaustein `hello-world` ab und legt auf.

```
[apfelmus]
exten => 8888,1,Answer()
exten => 8888,2,Playback(hello-world)
exten => 8888,3,Hangup()
```

n-Priorität

Seit der Asterisk-Version 1.2 ist es möglich, Prioritäten nicht nur streng mit Zahlen, sondern auch mit dem Platzhalter `n` zu belegen. Der `n`-Zähler fungiert hierbei als ein automatischer Programmzeilenzähler. Jedes Mal, wenn die Programmsteuerung auf die `n`-Priorität stößt, addiert sie 1 zum letzten Wert der Priorität. Dies ist dann hilfreich, falls Sie viele aufeinanderfolgende Regeln definiert haben und eine weitere Regel einfügen möchten, denn dann müssen Sie nicht mehr die Zähler der nachfolgenden Regeln neu nummerieren. Wenn eine normale Extension wie folgt aussieht

```
exten => 1234,1,Answer()
exten => 1234,2,Wait(2)
exten => 1234,3,Playback(hello-world)
exten => 1234,4,Wait(2)
exten => 1234,5,Hangup()
```

kann man die gleiche Extension auch mit der `n`-Priorität definieren:

```
exten => 1234,1,Answer()
exten => 1234,n,Wait(2)
exten => 1234,n,Play(hello-world)
exten => 1234,n,Wait(2)
exten => 1234,n,Hangup()
```

Dies kann nicht nur an der zweiten Priorität, sondern an einer beliebigen Stelle passieren:

```
exten => 1234,1,Answer()
exten => 1234,2,Wait(2)
exten => 1234,3,Play(hello-world)
exten => 1234,n,Wait(2)
exten => 1234,n,Hangup()
```

3.3 Pattern Matching

Mit unserem bisher erworbenen Wissen müssen wir pro möglicher Rufnummer immer eine eigene Extension schreiben. Dies würde schon nach kurzer Zeit sehr lange und fehleranfällige Dialpläne nach sich ziehen. Sollen z. B. die Rufnummern 100 bis 109 jeweils immer den Sprachbaustein `hello-world` abspielen, so würde die `extensions.conf` wie folgt aussehen:

[general]

[apfelmus]

```
exten => 100,1,Answer()  
exten => 100,2,Playback(hello-world)  
exten => 100,3,Hangup()
```

```
exten => 101,1,Answer()  
exten => 101,2,Playback(hello-world)  
exten => 101,3,Hangup()
```

```
exten => 102,1,Answer()  
exten => 102,2,Playback(hello-world)  
exten => 102,3,Hangup()
```

```
exten => 103,1,Answer()  
exten => 103,2,Playback(hello-world)  
exten => 103,3,Hangup()
```

```
exten => 104,1,Answer()  
exten => 104,2,Playback(hello-world)  
exten => 104,3,Hangup()
```

```
exten => 105,1,Answer()  
exten => 105,2,Playback(hello-world)  
exten => 105,3,Hangup()
```

```
exten => 106,1,Answer()  
exten => 106,2,Playback(hello-world)  
exten => 106,3,Hangup()
```

```
exten => 107,1,Answer()  
exten => 107,2,Playback(hello-world)  
exten => 107,3,Hangup()
```

```
exten => 108,1,Answer()  
exten => 108,2,Playback(hello-world)  
exten => 108,3,Hangup()
```

```
exten => 109,1,Answer()  
exten => 109,2,Playback(hello-world)  
exten => 109,3,Hangup()
```

**Tipp**

Definition *Regular Expression*:

»Reguläre Ausdrücke (Abkürzung: *RegExp* oder *Regex*, engl. *regular expressions*) dienen zur Beschreibung von (Unter-)Mengen von Zeichenketten mithilfe syntaktischer Regeln. Sie finden vor allem in der Softwareentwicklung Verwendung. Für fast alle Programmiersprachen existieren Implementierungen.« (zitiert aus http://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck)

Wenn wir ein Pattern in Form einer Regular Expression (auch Regex genannt) verwenden, sieht der gleiche Dialplan gleich viel handlicher aus:

```
[general]
```

```
[apfelmus]
```

```
exten => _10X,1,Answer()
```

```
exten => _10X,2,Playback(hello-world)
```

```
exten => _10X,3,Hangup()
```

Das Pattern `_10X` beschreibt den Zahlenraum von 100 bis 109.

**Achtung**

Man benutzt für die Beschreibung dieses Prozesses häufig das englische Verb *match* und das Substantiv *Pattern*. »Pattern« kann mit »Suchmuster« übersetzt werden. »match« lässt sich in etwa mit »zutreffen« übersetzen und die Verwendung dieser Begriffe ist am einfachsten mit einem Beispiel zu beschreiben: Ein Pattern ist »_10X«, und dieses Pattern »matcht« auf den Zahlenraum 100 bis 109. Es trifft also nicht auf die Zahl 110 zu.

**Hinweis**

Die Begriffe *Pattern* und *Regular Expression* werden in vielen Dokumentationen sehr beliebig eingesetzt. Auch dieses Buch leidet unter diesem Problem. Formal korrekt ist sicherlich der Begriff *Pattern*, aber die meisten Programmierer werden den Begriff *Regular Expression* benutzen.

3.3.1 Syntax

Ein Pattern wird immer mit einem Unterstrich (_) vor dem eigentlichen Suchmuster eingeleitet:

exten => _Regular Expression,Prioritaet,Applikation

Eine Regular Expression kann in Asterisk aus den folgenden Elementen⁴ bestehen:

- [ABC]

Die Ziffern A, B und C. Beispiel für die Zahlen 34, 37 und 38:

exten => _3[478],1,NoOp(Test)

- [A-B]

Beliebige Ziffer von A bis B. Beispiel für alle Zahlen von 31 bis 35:

exten => _3[1-5],1,NoOp(Test)

(Zum Beispiel wäre auch [25-8] für folgende Ziffern möglich: 2,5,6,7,8.)

- X

Beliebige Ziffer von 0 bis 9. Beispiel für alle Zahlen von 300 bis 399:

exten => _3XX,1,NoOp(Test)

- Z

Beliebige Ziffer von 1 bis 9. Beispiel für alle Zahlen von 31 bis 39:

exten => _3Z,1,NoOp(Test)

- N

Beliebige Ziffer von 2 bis 9. Beispiel für alle Zahlen von 32 bis 39:

exten => _3N,1,NoOp(Test)

- .

Eine oder mehrere beliebige Ziffer(n). Beispiel für alle Nummern, die mit einer 0 beginnen:

exten => _0.,1,NoOp(Test)

⁴ Es gibt noch weitere Elemente, die im deutschen Sprachraum aber im Allgemeinen wenig Sinn machen. Aus diesem Grund werden sie hier nicht aufgeführt.

**Hinweis**

Das Pattern `_` sollten Sie nicht verwenden! Es trifft auch auf besondere Extensions wie `i`, `t` oder `h` zu. Benutzen Sie stattdessen `_X` oder `_X`, falls nötig.



Eine oder mehrere beliebige Ziffer(n) – ab Asterisk 1.4. Dieser besondere Platzhalter trifft zu, sobald unzweifelhaft nicht eine andere explizite Nummer im Dialplan gewählt wird. Dann hebt sofort die Leitung für »overlap dialing« ab. Dieses Element wird hier nur der Vollständigkeit halber erwähnt.

**Achtung**

Ein beliebter Fehler ist es, am Anfang einer Regular Expression das Underscore-Zeichen »_« zu vergessen. Für Asterisk ist aber die Extension `XXX` ebenfalls eine vollkommen sinnvolle Extension (da SIP ja nicht nur Zahlen, sondern auch Buchstaben als Zieladresse kennt). Entsprechend wird es auch keine Fehlermeldung geben. Dummerweise wird das Pattern aber auch nie matchen, weil es nicht als Pattern (also mit dem `_`) eingegeben wurde.

3.3.2 Testen mit `dialplan show`

Nehmen wir einmal an, dass in unserer `extensions.conf` der folgende Dialplan steht:

```
[general]

[meine-telefone]
exten => 23,1,Answer()
exten => 23,2,Playback(hello-world)
exten => 23,3,Hangup()
```

Dann können wir im CLI von Asterisk (das ist das Interface, das bei einem bereits laufenden Asterisk mit `asterisk -r` gestartet werden kann) mit dem Befehl `dialplan show` (auf Asterisk 1.2: `show dialplan`) den aktuellen Dialplan anzeigen:

```
*CLI> dialplan show
[ Context 'default' created by 'pbx_config' ]
```

```
[ Context 'meine-telefone' created by 'pbx_config' ]
'23' =>          1. Answer()                [pbx_config]
                2. Playback(hello-world)   [pbx_config]
                3. Hangup()                [pbx_config]

[ Context 'parkedcalls' created by 'res_features' ]
'700' =>         1. Park()                  [res_features]

-- 2 extensions (4 priorities) in 3 contexts. --
*CLI>
```

Das CLI zeigt jetzt alle Wählregeln an, die Asterisk bekannt sind. Deshalb gibt es auch noch einen Context »parkedcalls«, den wir gar nicht wissentlich aktiviert haben (dieser wird standardmäßig in der `features.conf` aktiviert und stört uns jetzt nicht weiter). Wenn wir uns nur für den Dialplan für den Context `meine-telefone` interessieren, so können wir diesen mit `dialplan show meine-telefone` abrufen:

```
*CLI> dialplan show meine-telefone
[ Context 'meine-telefone' created by 'pbx_config' ]
'23' =>          1. Answer()                [pbx_config]
                2. Playback(hello-world)   [pbx_config]
                3. Hangup()                [pbx_config]

-- 1 extension (3 priorities) in 1 context. --
*CLI>
```

Der Befehl `dialplan show` kann aber nicht nur ganze Contexte anzeigen, sondern auch sagen, was passiert, wenn ich eine bestimmte Nummer wähle. Wenn ich mit einem Telefon, das im Context `meine-telefone` ist, die Nummer 25 anrufe, dann kann ich mit `dialplan show 25@meine-telefone` anzeigen, was passiert:

```
*CLI> dialplan show 25@meine-telefone
There is no existence of 25@meine-telefone extension
*CLI>
```

Es wird also nichts passieren, weil es keinen Match für die von mir gewählte Extension 25 gibt. Wenn ich das Gleiche für die 23 mache, dann gibt es folgende Ausgabe:

```
*CLI> dialplan show 23@meine-telefone
[ Context 'meine-telefone' created by 'pbx_config' ]
'23' =>          1. Answer()                [pbx_config]
                2. Playback(hello-world)   [pbx_config]
                3. Hangup()                [pbx_config]

-- 1 extension (3 priorities) in 1 context. --
*CLI>
```

Wenn ich in allen verfügbaren Contexten nach einem Match für die 23 suchen möchte, so geht das mit `dialplan show 23@`:

```
*CLI> dialplan show 23@
[ Context 'meine-telefone' created by 'pbx_config' ]
  '23' =>          1. Answer()                [pbx_config]
                  2. Playback(hello-world)   [pbx_config]
                  3. Hangup()                 [pbx_config]

-= 1 extension (3 priorities) in 1 context. -=
*CLI>
```

Erweitern wir unseren Dialplan einmal um einen weiteren Context:

```
[general]

[meine-telefone]
exten => 23,1,Answer()
exten => 23,2,Playback(hello-world)
exten => 23,3,Hangup()

[abteilung-z]
exten => _2X,1,Answer()
exten => _2X,2,Playback(hello-world)
exten => _2X,3,Hangup()
```

Und jetzt führen wir noch einmal `dialplan show 23@` aus (vorher müssen wir natürlich Asterisk mit `reload` im CLI sagen, dass es den neuen Dialplan einlesen soll):

```
*CLI> dialplan show 23@
[ Context 'abteilung-z' created by 'pbx_config' ]
  '_2X' =>         1. Answer()                [pbx_config]
                  2. Playback(hello-world)   [pbx_config]
                  3. Hangup()                 [pbx_config]

[ Context 'meine-telefone' created by 'pbx_config' ]
  '23' =>          1. Answer()                [pbx_config]
                  2. Playback(hello-world)   [pbx_config]
                  3. Hangup()                 [pbx_config]

-= 2 extensions (6 priorities) in 2 contexts. -=
*CLI>
```

Es werden also alle matchenden Extensions angezeigt. Um im obigen Beispiel zu bleiben, probieren wir das jetzt auch noch einmal mit `dialplan show 25@` aus:

```
*CLI> dialplan show 25@
[ Context 'abteilung-z' created by 'pbx_config' ]
  '_2X' =>         1. Answer()                [pbx_config]
```

```

2. Playback(hello-world)           [pbx_config]
3. Hangup()                         [pbx_config]

```

```

== 1 extension (3 priorities) in 1 context. ==
*CLI>

```

Logischerweise gibt es dabei nur einen Treffer, und der ist im Context `abteilung-z`. Sollten Sie also mit einem Telefon, das im Context `meine-telefone` arbeitet, die 25 wählen, so werden Sie trotzdem kein `hello-world` hören, denn dies funktioniert nur bei Telefonen, die auch im Context `abteilung-z` arbeiten.

3.3.3 Wann matcht welches Pattern?

Das Pattern Matching in Asterisk ist bei großen Dialplänen eine trickreiche Angelegenheit. Asterisk geht nämlich nicht, wie allgemein angenommen wird, plump von oben nach unten den Dialplan durch. Nein, es priorisiert innerhalb der Patterns!

Je exakter ein Pattern matcht, desto höher ist die Wahrscheinlichkeit, dass es matcht. Asterisk geht allerdings – bevor es eine Entscheidung trifft – den ganzen Context durch. Es könnte ja sein, dass ein anderes Pattern noch besser matcht.

Beispiel:

```

[verkauf]
exten => _12X.,1,NoOp{12X}
exten => 12345,1,NoOp{12345}
exten => _1234.,1,NoOp{1234.}

```

Um herauszufinden, welches Pattern matcht, wenn die 12345 gewählt wird, können wir dies mit `dialplan show 12345@verkauf` überprüfen:

```

*CLI> dialplan show 12345@verkauf
[ Context 'verkauf' created by 'pbx_config' ]
'12345' =>      1. NoOp{12345})           [pbx_config]
'_1234.' =>     1. NoOp{1234.})()        [pbx_config]
'_12X.' =>      1. NoOp{12X})()          [pbx_config]

```

```

== 3 extensions (3 priorities) in 1 context. ==
*CLI>

```

Asterisk zeigt alle Treffer, aber priorisiert die Zeile, in der `12345,1,NoOp{12345}` steht. Die höchste Priorität wird immer ganz oben angezeigt.

Jetzt kontrollieren wir das noch für die Nummer 12346 mit dem Befehl `dialplan show 12346@verkauf`:

```

*CLI> dialplan show 12346@verkauf
[ Context 'verkauf' created by 'pbx_config' ]
'_1234.' =>     1. NoOp{1234.})()        [pbx_config]

```

```
'_12X.' => 1. NoOp(12X)() [pbx_config]
-- 2 extensions (2 priorities) in 1 context. --
*CLI>
```

Auch hier bekommt das Pattern mit dem »besten« Treffer die höchste Priorität.



Achtung

Es ist nicht wichtig, in welcher Reihenfolge Patterns im Dialplan geschrieben werden! Es ist nur wichtig, wie genau ein Pattern matcht. Je genauer es matcht, desto höher wird es priorisiert.



Achtung

Es gibt eine Sonderregel für das Pattern `_`.
`»_«` matcht immer und hat auch immer die höchste Priorität. Es ist also egal, was Sie sonst noch in diesem Context für Regeln haben! Es wird immer nur die Regel mit dem Pattern `»_«` ausgeführt. Man sollte also lieber ein `»_X«` nehmen, außer man ist sich absolut sicher und weiß, was das Pattern `»_«` bewirkt.

Auch wenn die Reihenfolge von Pattern nicht immer ganz trivial ist, gibt es eine einfache Debugging-Möglichkeit. Mit `show dialplan 12345@verkauf` lässt sich der Dialplan für die gewählte Nummer 12345 im Context `verkauf` auflisten. So können Sie für spezielle Nummern überprüfen, ob auch die dafür vorgesehene Regel matcht.

Sonderregel für das Pattern `_` in Asterisk 1.2

Damit das Leben eines Asterisk-Administrators nicht zu einfach wird, hat sich Digium noch eine Besonderheit für das Pattern `»_«` in der Asterisk-Version 1.2 ausgedacht. Obwohl dieses Pattern das allgemeinste und damit von der Logik her das Pattern mit der geringsten Priorität sein müsste, ist es genau andersherum!



Achtung

`_` bekommt in der Asterisk-Version 1.2 immer die höchste Priorität!



Hinweis

Bitte beachten Sie, dass der CLI-Befehl `show dialplan` zwar auch noch in der Version 1.4 funktioniert, aber unerwünscht ist. Deshalb lauten die Aufrufe in der Version 1.2 `show dialplan` und in der Version 1.4 `dialplan show`.

Probieren wir noch einmal unseren obigen Dialplan mit einer zusätzlichen Extension »_.« aus:

```
[verkauf]
exten => _12X.,1,NoOp{12X}
exten => 12345,1,NoOp{12345}
exten => _1234.,1,NoOp{1234.}

exten => _.,1,NoOp{Bingo}
```

Wenn wir jetzt die Rufnummer 12346 ausprobieren wollen, bekommen wir mit dem Befehl `dialplan show 12346@verkauf` in der Version 1.4 folgende Ausgabe:

```
*CLI> dialplan show 12346@verkauf
[ Context 'verkauf' created by 'pbx_config' ]
'_1234.' => 1. NoOp{1234.}() [pbx_config]
'_12X.' => 1. NoOp{12X}() [pbx_config]
'_.' => 1. NoOp{Bingo}() [pbx_config]

-- 3 extensions (3 priorities) in 1 context. --
*CLI>
```

In Asterisk 1.2 bekommt der Befehl `show dialplan 12346@verkauf` aber folgende Ausgabe:

```
*CLI> dialplan show 12346@verkauf
[ Context 'verkauf' created by 'pbx_config' ]
'_.' => 1. NoOp{Bingo}() [pbx_config]
'_1234.' => 1. NoOp{1234.}() [pbx_config]
'_12X.' => 1. NoOp{12X}() [pbx_config]

-- 3 extensions (3 priorities) in 1 context. --
*CLI>
```

Deshalb sollte man als »Restesammler« (wenn überhaupt) nur das Pattern `_X.` benutzen. Der folgende Dialplan wird in den Asterisk-Versionen 1.2 und 1.4 gleich behandelt:

```
[verkauf]
exten => _12X.,1,NoOp{12X}
exten => 12345,1,NoOp{12345}
exten => _1234.,1,NoOp{1234.}

exten => _X.,1,NoOp{Bingo}
```

Die Prioritäten sind in beiden Asterisk-Versionen wie folgt:

```
*CLI> dialplan show 12346@verkauf
[ Context 'verkauf' created by 'pbx_config' ]
'_1234.' =>      1. NoOp{1234.}()           [pbx_config]
'_12X.' =>      1. NoOp{12X}()           [pbx_config]
'_X.' =>       1. NoOp{Bingo}()         [pbx_config]

-= 3 extensions (3 priorities) in 1 context. -=
*CLI>
```

3.4 Includes im Dialplan

Includes bringen Struktur und Ordnung in große Dialpläne. Mit einem Include können andere Contexte in den aktuellen Context eingebaut (also »included«) werden.

3.4.1 Syntax

```
include => Name-des-anderen-Contextes
```

3.4.2 Beispiel

```
[general]

[verkauf]
include => intern
include => extern

[intern]
exten => 2000,1,Dial(SIP/2000)

[extern]
exten => 03012345678,1,Dial(SIP/03012345678)
```

3.4.3 Die Reihenfolge beim Include

Asterisk sucht, bevor es einen anderen Context einbindet (include), immer erst im aktuellen Context nach einem Treffer (einem Match). Gibt es einen Treffer, wird dieser benutzt. Gibt es keinen Treffer, wird das erste Include aufgerufen und dort nach einem Treffer gesucht. Dies funktioniert rekursiv nach unten – auch verschachtelt: Es können also auch Includes innerhalb von Includes abgearbeitet werden.

Im Zweifelsfall können Sie zum Debuggen auch hier die Applikation `dialplan show nummer@name-des-contextes` benutzen, um herauszufinden, welche Regel von Asterisk angewendet wird.



Hinweis

Benutzer von Asterisk 1.2 müssen anstatt `dialplan show immer show dialplan` eingeben.

Ein paar Beispiele:

```
[general]
```

```
[verkauf]
include => intern
include => extern
```

```
[intern]
exten => 2000,1,Dial(SIP/2000)
```

```
[extern]
exten => 03012345678,1,Dial(SIP/03012345678)
```

Wenn mit diesem Dialplan im Context `verkauf` die Nummer 2000 angerufen wird, dann können wir mit dem CLI-Befehl `dialplan show 2000@verkauf` den Ablauf analysieren:

```
*CLI> dialplan show 2000@verkauf
[ Included context 'intern' created by 'pbx_config' ]
  '2000' =>          1. Dial(SIP/2000)                                [pbx_config]

-- 1 extension (1 priority) in 1 context. --
*CLI>
```

Wenn wir jetzt im Context `verkauf` wie folgt erweitern, ...

```
[general]

[verkauf]
include => intern
include => extern

exten => 2000,1,Answer()
exten => 2000,2,Playback(hello-world)
exten => 2000,3,Hangup()

[intern]
exten => 2000,1,Dial(SIP/2000)

[extern]
exten => 03012345678,1,Dial(SIP/03012345678)
```

... dann bekommen wir folgende Analyse des Dialplans angezeigt:

```
*CLI> dialplan show 2000@verkauf
[ Context 'verkauf' created by 'pbx_config' ]
  '2000' =>      1. Answer()                [pbx_config]
                2. Playback(hello-world)   [pbx_config]
                3. Hangup()                 [pbx_config]
[ Included context 'intern' created by 'pbx_config' ]
  '2000' =>      1. Dial(SIP/2000)          [pbx_config]

-= 2 extensions (4 priorities) in 2 contexts. -=
*CLI>
```

Asterisk wird also den Sprachbaustein `hello-world` abspielen und nicht zum Telefon 2000 durchstellen, und das, obwohl das Include vorher im Dialplan auftaucht. Das liegt daran, dass erst alle Möglichkeiten innerhalb eines Contextes und dann erst die Includes abgearbeitet werden.

3.4.4 Includes zeitgesteuert

Durch die Fähigkeit, Includes auch zeitgesteuert durchzuführen, kann man mit diesem Mechanismus sehr leicht Tag- und Nachtschaltungen durchführen.

Syntax

```
include => context|<uhrzeit>|<wochentag>|<tag-des-monats>|<monat>
```

Die Wochentage und Monate werden immer durch die ersten drei Buchstaben des entsprechenden englischen Begriffs bestimmt. Die Wochentage heißen also: `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `sun`.

Beispiel

Wenn eine Firma an Wochentagen von 9:00 bis 17:00 Uhr und samstags von 9:00 bis 14:00 Uhr geöffnet hat, kann ein Dialplan für sie wie folgt aussehen:

```
; Tag

include => tagschaltung|09:00-17:00|mon-fri|*|*

include => tagschaltung|09:00-14:00|sat|*|*
include => anrufbeantworter

[tagschaltung]
exten => 2000,1,Dial(SIP/2000)

[anrufbeantworter]
exten => 2000,1,VoiceMail(2000,u)
```

3.5 Die Variable `${EXTEN}` und die Funktion `${CALLERID(num)}`

Obwohl wir im Buch erst später über Variablen (siehe Abschnitt 6.1.2, *Variablen*), und Funktionen (siehe Anhang D, *Funktionen im Dialplan*) sprechen, möchte ich zwei sehr einfache und intuitiv zu benutzende Elemente schon hier vorstellen. Es handelt sich um die Systemvariable `${EXTEN}` und die Funktion `${CALLERID(num)}`.

3.5.1 `${EXTEN}`

In der Systemvariable `${EXTEN}` wird von Asterisk automatisch die gewählte Rufnummer (also die Extension) gespeichert. Man kann also in der `extensions.conf` anstatt

```
exten => 2000,1,Dial(SIP/2000)
```

auch einfach

```
exten => 2000,1,Dial(SIP/${EXTEN})
```

schreiben. Bei einer Zeile ist das natürlich noch wenig sinnvoll, aber wenn man diese Funktionalität mit Pattern Matching (siehe Abschnitt 3.3.1, *Syntax*), kombiniert, dann kann man sehr viel Zeit und Aufwand sparen und bekommt zusätzlich auch noch eine viel übersichtlichere Konfiguration.

Um somit alle SIP-Telefone mit den Durchwahlen 2000 bis 2999 in der `extensions.conf` anwählbar zu machen, reicht folgende Zeile:

```
exten => _2XXX,1,Dial(SIP/${EXTEN})
```



Tipp

Eine ausführliche Beschreibung zum Thema Variablen finden Sie unter Abschnitt 6.2, *Variablen*.

3.5.2 `#{CALLERID(num)}`

Der Funktionsaufruf `#{CALLERID(num)}` gibt als Ergebnis die Nummer des Anrufers aus. Dies ist besonders bei der Applikation `VoiceMailMain()` praktisch, da man dort als ersten Parameter die gewünschte Mailboxnummer angeben kann. So könnte man mit folgendem Dialplan-Eintrag eine komfortable Abfrage der Voicemailbox realisieren:

```
exten => 99,1,VoiceMailMain(#{CALLERID(num)})
```



Tipp

Eine ausführliche Beschreibung zur Funktion `#{CALLERID(num)}` finden Sie unter Anhang D.8, *CALLERID()*.