



# Call Flow Development – Programmer's Guide

(version 2.1)

**Disclaimer**

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR WYDE VOICE REPRESENTATIVE FOR A COPY.

IN NO EVENT SHALL WYDE VOICE OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF WYDE OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**Copyright**

Except where expressly stated otherwise, the Product is protected by copyright and other laws respecting proprietary rights. Unauthorized reproduction, transfer, and or use can be a criminal, as well as civil, offense under the applicable law.

WYDE Voice and the WYDE Voice logo are registered trademarks of WYDE Voice LLC in the United States of America and other jurisdictions. Unless otherwise provided in this Documentation, marks identified with “R” / ®, “TM” / ™ and “SM” are registered marks; trademarks are the property of their respective owners.

For the most current versions of documentation, go to the WYDE support Web site:

<http://www.wydevoice.com/support>

July 24, 2010

## Symbols and Notations in this Manual

The following notations and symbols can be found in this manual.



Denotes any item that requires special attention or care. Damage to the equipment or the operator may result from failure to take note of the noted instructions

**Figure** Denotes any illustration

**Table** Denotes any table

**Text** Denotes any text output

*Folder/File* Denotes any folders (paths) or files names

`commands` Denotes any callback handlers, DTMF commands, attributes and parameters

## Table of Contents

Symbols and Notations in this Manual .....	3
Table of Contents .....	4
Tables List .....	6
Figures List .....	7
Chapter 1: Introduction .....	8
Assumed Skills .....	8
Asterisk Extension Language .....	8
Assumptions .....	9
Definitions .....	10
Chapter 2: Samples of Call Flows .....	12
Sample 1 – Simple Call Flow without Authorization .....	14
Sample 2 – Call Flow with Authorization .....	16
Sample 3 – Call Flow with DTMF Processing .....	20
Sample 4 – Call Flow with Custom Handlers .....	24
Chapter 3: Function Reference .....	30
Callback Handlers .....	30
entry_handler .....	30
fastjoin_handler .....	30
waitmoderator_handler .....	31
holdline_handler .....	31
welcome_handler .....	31
gotomp_handler .....	31
recstop_handler .....	31
terminate_handler .....	32
hangupt_handler .....	32
DTMF commands .....	32
call_participantsnumber .....	32
call_exit .....	33
call_instructions .....	33
call_mute_switch .....	33
conference_mute_switch .....	33
conference_lock_switch .....	34
conference_entryexittones_switch .....	34
conference_qa_moderator .....	34
recording_switch .....	34
Dialplan commands and functions .....	34
WYDE_Playback .....	35
WYDE_Input .....	35
WYDE_Choice .....	35
WYDE_AGIRrequest .....	36
WYDE_IVRStat .....	36
WYDE_IVRConfStat .....	37
WYDE_IVRVar .....	37
WYDE_IVRCheckRole .....	37

Appendix A: Call Flow Library .....	38
/usr/local/DNCA/lib/CallFlows/functions.ael .....	38
Appendix B: Support Resources .....	43
Support Documentation.....	43
Web Support.....	43
Telephone Support.....	43
Email Support.....	43

***Tables List***

Table 1: SAMPLE Call Flow Folder Structure and Contents ..... 12

**Figures List**

Figure 1: Register New SAMPLE Call Flow on the Bridge .....	13
Figure 2: New SAMPLE Call Flow .....	13
Figure 3: New DNIS 8665089020 for SAMPLE Call Flow .....	14
Figure 4: Calls Screen for the Conference – Sample 1 Call Flow (without Authorization). 16	
Figure 5: Two Created Conference Accounts with Host and Participants Roles.....	17
Figure 6: Calls Screen for the Conference – Sample 2 Call Flow (with Authorization).....	20
Figure 7: Calls Report – Sample 2 Call Flow (with Authorization).....	20
Figure 8: SAMPLE Call Flow with New Lock and Mute Attributes.....	23
Figure 9: Calls Screen for the Conference – Sample 3 Call Flow (with DTMF Processing) .....	24
Figure 10: SAMPLE Call Flow with New Custom Attributes.....	29

## Chapter 1: Introduction

Call flow is a unique conference service setup, the logic that is used to process the conference calls. This is the process a call goes through from call setup to, to processing, to call tear down. It is possible to create own call flow and customize existing call flow. This guide describes how to do it.

AEL2 language (Asterisk Extension Language v.2) is used to write call flow scenarios. When you write own call flows you can either use standard applications and functions available in AEL2 or use procedures and functions provided by WYDE bridge environment.

### *Assumed Skills*

This call flow development programmer's guide assumes you have a working knowledge of the following technologies and skills:

- PC usage
- System administration
- Asterisk administration and configuration
- AEL basics
- VOIP basics
- TCP/IP networking
- Web Administration Interface – User Guide

### *Asterisk Extension Language*

Asterisk is software that turns an ordinary computer into a voice communications server. Asterisk is the worlds most powerful and popular telephony development tool-kit. It is used by small businesses, large businesses, call centers, carriers and governments worldwide. Asterisk is open source and is available free to all under the terms of the GPL.

The Asterisk software includes many features available in proprietary PBX systems: voice mail, conference calling, interactive voice response (phone menus), and automatic call distribution. Users can create new functionality by writing dial plan scripts in several of Asterisk's own extensions languages, by adding custom loadable modules written in C, or by implementing Asterisk Gateway Interface (AGI) programs using any programming language capable of communicating via the standard streams system (stdin and stdout) or by network TCP sockets.

Perhaps one of more interest to many deployers today, Asterisk also supports a wide range of Video and Voice over IP protocols, including SIP, MGCP and H.323. Asterisk can interoperate with most SIP telephones, acting both as registrar and as a gateway between IP phones and the PSTN. Asterisk developers have also designed a new protocol, Inter-Asterisk eXchange (IAX2), for efficient trunking of calls among Asterisk PBXes, and to VoIP service providers who support it. Some telephones support the IAX2 protocol directly.



AEL v.2 is intended to provide an actual programming language that can be used to write an Asterisk dialplan. It further extends AEL, and provides more flexible syntax, better error messages, and some missing functionality.

AEL v.2 is a new version of the AEL compiler. It was originally introduced as a large asterisk patch in the Asterisk bug database.

AEL is really the merger of 4 different 'languages', or syntaxes:

- The first and most obvious is the AEL v.2 syntax itself. A BNF is provided near the end of this document.
- The second syntax is the Expression Syntax, which is normally handled by Asterisk extension engine, as expressions enclosed in `$[...]`. The right hand side of assignments are wrapped in `$[ ... ]` by AEL, and so are the if and while expressions, among others.
- The third syntax is the Variable Reference Syntax, the stuff enclosed in `${..}` curly braces. It's a bit more involved than just putting a variable name in there. You can include one of dozens of 'functions', and their arguments, and there are even some string manipulation notations in there.
- The last syntax that underlies AEL/AEL2, and is not used directly in AEL/AEL2, is the Extension Language Syntax. The extension language is what you see in `extensions.conf`, and AEL2 compiles the higher level AEL2 language into extensions and priorities, and passes them via function calls into Asterisk. Embedded in this language is the Application/AGI commands, of which one application call per step, or priority can be made. You can think of this as a "macro assembler" language that AEL2 will compile into.

Any programmer of AEL should be familiar with its syntax, of course, as well as the Expression syntax, and the Variable syntax.

The detail information about AEL can be read in the following articles:

- The Open Source Telephony Project Asterisk – <http://www.asterisk.org/>
- Asterisk Extension Language on Voip-Info.org – <http://www.voip-info.org/wiki/view/Asterisk+AEL2>
- Asterisk on WikipediA.org – [http://en.wikipedia.org/wiki/Asterisk\\_PBX](http://en.wikipedia.org/wiki/Asterisk_PBX)

### ***Assumptions***

Each call flow scenario should be placed in its own folder – the subfolder of `/usr/local/DNCA/callflows` folder that is root folder for call flow scenarios.

This folder should contain the following files:

- *script.ael* – Asterisk dialplan on AEL language;
  - *callflow.spec* – specification of the scenario;
- and this folder also should contain subfolder:
- *sounds* – the set of voice files in the *ulaw* format.

When you write call flow scenarios you should follow to the following assumptions and regulations:

- to customize the call flow scenario the callback procedures are used, these procedures are written on AEL language and these procedures are being called by WYDE bridge environment in all key-points of the processing of the calls;
- each callback procedure is being defined in the *script.ael* file as separate AEL scope (context);
- the callback procedure either returns processing to call flows execution environment using `Return ()` command or interrupts the call using `Hangup ()` command;
- the binding of the callback procedures and descriptors is being made in the *callflow.spec* file in the *handlers* section;
- the *script.ael* file should contain the definition for at least one callback procedure – this callback procedure should be bound with the call entry point (`entry_handler`);
- the names of the procedures (scopes/context) defined in the *script.ael* file should be unique and should not match to the names defined in other call flow scenarios and should not match to the names of the call flows execution environment core.

### **Definitions**

In order to discuss the WYDE call flows development effectively, we need to have a common set of terminology. For this purpose, we should definite the dictionary for the terms you will see throughout this programmer's guide:

- **Call Flow** – A unique conference service setup, the logic that is used to process the conference calls. This is the process a call goes through from call setup to, to processing, to call tear down. It includes the logic, DTMF key-presses used, functions, and the recorded prompts. There are two basic call flow categories: call flows without authentication and call flows with authentication.
- **Attribute** – In terms of WYDE web services API, a data structure is used to carry attributes for call flow, DNIS and conference account (user). The attributes skeleton is defined by call flow; other attributes can only override some of them, so for instance when a user called in to the conference DNIS it gets attributes exposed by the call flow, but some of these attributes can be already altered by the DNIS. Each attribute has name, type, value, and role.
- **Subscriber** – A real person, he has a name, phone number, e-mail address, etc. The subscriber can have conference accounts, he does not have access codes, but access codes are properties of conference accounts that have subscribers. Note that non-admin (non-operator) subscribers can see only “own” information, i.e. his information and information that belongs to subscribers created by him, he can see only their calls, conferences, the reports will show only their data, etc.
- **PIN** – The login ID for the subscriber (must be unique). It can be used either as login in Web Administration Interface (in this case it can be either number or alpha-numeric) or as login for some call flows (in this case must be numeric) for participants authorization.
- **Conference Account** – The element of subscriber conferences configuration. Conference accounts always belong to subscriber. It is being used to define a person in a conference with a particular role (e.g. host, participant, listener, etc.), the DNIS number that should be used to call to the conference, and the access code that should be entered by the user that called to the conference DNIS to determine his role. A

subscriber could be a host user in one conference and a listener in another. Conference accounts with the same conference number represent single conference setup.













- **DNIS** – A unique set of numbers that is outputted by a phone carrier that indicates the intended destination for a particular call. It can be any length digits (although usually 10 digits). DNIS is the property of the conference account, but different DNIS numbers can be used to connect to the same conference.
- **Access Code** – A numeric code unique for DNIS that allows a host or participant or listener access to a conference call. When users call to DNIS number they being asked to enter their access code. The access code determines the conference and the user role in the conference. Different access codes can determine the same conference, for instance one access code can determine the connected user has host role, another access code can determine that connected user has participant role, and another access code can determine that connected user has listener role.
- **Host** – A user in the conference call that can make changes to the system while the conference call is in progress. Like change the security setting, change who can talk or answer, etc. Sometimes the host user is called moderator. This user role is defined in conference account.
- **Participant** – A person in the conference who can actively participate in a call by both talking and listening. This user role is defined in conference account.
- **Listener** – A person in the conference who can hear the conference call, but cannot speak. Their audio path is one way only (receive). This user role is defined in conference account.
- **Conference Number** – A unique external conference number. Conference number is the property of conference account. If the conference accounts have the same conference number all these accounts determine one single conference. For instance the user can create one conference account record that determine host role, another conference account record that determine participant role, and another conference account record that determine listener role – all these records should have the same conference number to determine one unique conference.
- **Conference ID** – A unique conference ID that represents the instance of a conference. When any conference is being started it receives unique conference ID, and all calls to this conference have the same conference ID; if this conference has been completed and another conference is being started that conference will receive another conference ID. Conference ID is normally not exposed to users, unless on the reports.
- **Conference** – A data structure is used to describe ongoing conference on the bridge. Objects of this type are only created by server. User may fetch these objects by calling appropriate function. When conference is over the conference object is deleted by the server.
- **Session** – A data structure represents a single ongoing call on the server. User can not directly create this object. When the call is over server automatically deletes this object. Normally this data structure is used to get information about call attributes like calling/called number etc., or do something with the call, for instance mute, hang, hold etc.

## Chapter 2: Samples of Call Flows

In this section we will explain how to write call flows and we will give few samples of call flow scripts.

Let's assume that for samples purposes we create call flow with name `SAMPLE`. To do that we need to create directory `/usr/local/DNCA/callflows/SAMPLE`, it will be the working directory for the call flow `SAMPLE`. This folder will contain two files: `callflow.spec` and `script.ael`; the content of these files will be different for each sample and this content will define the call flow behavior. Also we need to create subdirectory `sounds` (`/usr/local/DNCA/callflows/SAMPLE/sounds`) to put audio (voice) files in the `ulaw` format into this folder. In addition if we need to pronounce numbers we need to create subdirectory `digits` in directory `sounds` (`/usr/local/DNCA/callflows/SAMPLE/sounds/digits`) and put the files `0.ul`, `1.ul`, `2.ul`, etc. into it. `SAMPLE` call flow folder structure is shown in Table 1.

[Click here to download all sounds files that are necessary to implement these samples.](#)

Folder or File Name	Description
 <code>SAMPLE</code>	call flow root folder
 <code>callflow.spec</code>	Asterisk dialplan on AEL language file
 <code>script.ael</code>	specification of the scenario file
 <code>sounds</code>	audio (voice) files folder
 <code>accesscode_accepted.ul</code>	audio file that pronounces "access code accepted" message
 <code>callers.ul</code>	audio file that pronounces "callers" message
 <code>enter_accesscode.ul</code>	audio file that pronounces "enter access code" message
 <code>incorrect_accesscode.ul</code>	audio file that pronounces "incorrect access code" message
 <code>thereare.ul</code>	audio file that pronounces "there are" message
 <code>welcome.ul</code>	audio file that pronounces "welcome" message
 <code>digits</code>	digits audio files folder (to pronounce numbers)
 <code>0.ul</code> <code>1.ul</code> <code>2.ul</code> <code>3.ul</code>	audio files that pronounce numbers (i.e. "zero", "one", ..., "ten", "eleven", "twelve", ..., "twenty", "thirty", ..., "ninety", "hundred", "thousand")
<code>4.ul</code> <code>5.ul</code> <code>6.ul</code> <code>7.ul</code> <code>8.ul</code>	
<code>9.ul</code> <code>10.ul</code> <code>11.ul</code> <code>12.ul</code>	
<code>13.ul</code> <code>14.ul</code> <code>15.ul</code> <code>16.ul</code>	
<code>17.ul</code> <code>18.ul</code> <code>19.ul</code> <code>20.ul</code>	
<code>30.ul</code> <code>40.ul</code> <code>50.ul</code> <code>60.ul</code>	
<code>70.ul</code> <code>80.ul</code> <code>90.ul</code>	
<code>hundred.ul</code> <code>thousand.ul</code>	

**Table 1: SAMPLE Call Flow Folder Structure and Contents**

Let's assume that DID number for new call flow is `8665089020`. To register new `SAMPLE` call flow and assign it to the DID the following commands should be executed:

```
cd /usr/local/DNCA/callflows/SAMPLE
wyde callflow-add name SAMPLE
wyde did-add number 8665089020 callflow SAMPLE
```

```

root@testbox:/usr/local/DNCA/callflows/SAMPLE
[root@testbox ~]# cd /usr/local/DNCA/callflows/SAMPLE
[root@testbox SAMPLE]# wyde callflow-add name SAMPLE
[root@testbox SAMPLE]# wyde did-add number 100 callflow SAMPLE
[root@testbox SAMPLE]# █

```

Figure 1: Register New SAMPLE Call Flow on the Bridge

The `cd` command is being executed to make SAMPLE call flow directory the current folder; `wyde callflow-add` command creates new call flow (see Figure 2); `wyde did-add` command creates and associates new DNIS (DID) number with the specified call flow (see Figure 3).

WYDE VOICE

24/7 Professional Customer Service  
TOLL FREE 866.508.9020

Subscribers | Conferences | Calls | Reports | **DNIS** | Preferences | Logout | CONFERENCE APPLIANCE MANAGER

\*Call Flow Name

\*Directory Path

Description	Name ▲	Role	Value

Update  Cancel

Figure 2: New SAMPLE Call Flow



24/7 Professional Customer Service  
TOLL FREE 866.508.9020

Subscribers | Conferences | Calls | Reports | **DNIS** | Preferences | Logout | CONFERENCE APPLIANCE MANAGER

\*DNIS

\*Call Flow Name

Description

Description	Name ▲	Role	Value	Overridden
<input type="button" value="Update"/> <input type="button" value="Cancel"/>				

Figure 3: New DNIS 8665089020 for SAMPLE Call Flow

If the call flow already exists to update its attributes in a database the following command should be executed:

```
wyde callflow-attr-update-db callflow SAMPLE
```

This command should be executed if *callflow.spec* file was updated. If the changes were made in the *script.ael* file only, it is not necessary to run this command.

To send the signal on the WYDE bridge to the call flow engine to reload the scripts the following command should be executed:

```
wyde callflow-reload
```

This command should be executed if there were changes in the *script.ael* file or sound files were updated. If the changes were made in the *callflow.spec* file only, it is not necessary to run this command.

Note this command reloads all call flows.

As soon as these commands are executed you can call to the number 8665089020; SAMPLE call flow will process these calls to 8665089020 number.

### ***Sample 1 – Simple Call Flow without Authorization***

Let's review the following call flow scenario:

- when the call is made we need to playback “welcome” prompt (message);
- after that we need to join the call to the conference with the number equal to the called number;
- the role of the caller in the conference should be “participant”.

To implement this scenario we need to create the following *callflow.spec* and *script.ael* files with the following contents:

***callflow.spec***

```
[handlers]
entry_handler = sample_entry_handler
```

***script.ael***

```
context sample_entry_handler {
    s => {
        WYDE_Playback(welcome);
        Set(conf_number=${called_number});
        Set(role=participant);
        Return();
    }
}
```

Let's consider the implemented logic in details. The *callflow.spec* file defines that `sample_entry_handler` handler should be used as `entry_handler` of the call. The *script.ael* file contains the contents (context) of `sample_entry_handler`.  
`WYDE_Playback(welcome); // Play the prompt from a welcome.ul file.`

`Set(conf_number=${called_number}); // Set the value of conf_number variable equal to called_number (8665089020 in our sample).`

`Set(role=participant); // Set the value of role variable equal to participant, i.e. the caller will be connected to the conference with "participant" role.`

`Return(); // Return processing to call flows execution environment (the call flow engine context).`

Note 1. To join the call to the conference the WYDE bridge call flows execution environment should know the conference number and the caller role. In our sample the conference number is equal to called number (8665089020) and the role is always participant.

Note 2. If it was not previously made, do not forget to place *welcome.ul* audio file into *sounds* subfolder of *SAMPLE* folder of the call flow.

If the call flow *SAMPLE* does not exist to register our new call flow and assign it to the DID 8665089020 the following commands should be executed (as it was previously described in this guide):

```
cd /usr/local/DNCA/callflows/SAMPLE
wyde callflow-add name SAMPLE
wyde did-add number 8665089020 callflow SAMPLE
```

If the call flow already was created to update its attributes in a database the following command should be executed:

```
wyde callflow-attr-update-db callflow SAMPLE
```

To send the signal on the WYDE bridge to the call flow engine to reload the scripts the following command should be executed:

```
wyde callflow-reload
```

Now if you call to the number 8665089020 you can hear “welcome” message and after that your call will be joined to the conference with number 8665089020. For instance if two callers call to 8665089020 number you will see the calls conference screen similar to shown on Figure 4.

The screenshot shows the WYDE VOICE CONFERENCE APPLIANCE MANAGER interface. At the top, there is a navigation bar with links for Subscribers, Conferences, Calls, Reports, DNIS, Preferences, and Logout. The main header displays 'CONFERENCE APPLIANCE MANAGER' and '24/7 Professional Customer Service TOLL FREE 866.508.9020'. Below the header, the conference number is 8665089020, and the status is 'open'. There are controls for secure, hold, ASN, and recording, all currently set to OFF. A 'Dialout' button is visible. A search bar is present with a 'Search' button. The number of calls is 2, and the refresh rate is set to 'No Refresh'. A table of active calls is shown below, with columns for Calling Number, Called Number, User Name, Access Code, Call Begin, Duration, Status, Role, Mute, Hold, and Q&A. Two calls are listed: one for 'testing' and one for 'unknown', both with a duration of 01m:48s and 02m:50s respectively, and a status of 'conference participant'.

Calling Number	Called Number	User Name	Access Code	Call Begin	Duration	Status	Role	Mute	Hold	Q&A
✘	(866) 508-9020	testing		12:21:48	01m:48s	conference	participant	<input type="checkbox"/>	<input type="checkbox"/>	
✘	(866) 508-9020	unknown		12:20:46	02m:50s	conference	participant	<input type="checkbox"/>	<input type="checkbox"/>	

**Figure 4: Calls Screen for the Conference – Sample 1 Call Flow (without Authorization)**

[Click here to download the Sample 1.](#) The archive files from SAMPLE1 folder should be extracted into SAMPLE folder of your call flow.

### ***Sample 2 – Call Flow with Authorization***

Let's review the following call flow scenario:

- when the call is made we need to playback “welcome” prompt (message);
- after that we need to ask to enter access code;
- we need to make authorization request based on called number and access code entered;
- if authorization is successful we need to playback “access code accepted” message and connect the call to the conference;
- if authorization is unsuccessful we need to set disconnect reason equal “Incorrect access code”, playback “incorrect access code” message and disconnect the call.

We assume that the conference accounts (conference users) should be created for the conference and conference number and access codes should be defined prior the call. This can be made using the following commands:

```
wyde confuser-add subscriber admin did 8665089020 conference 267996 role Host accesscode 1111
```

```
wyde confuser-add subscriber admin did 8665089020 conference 267996 role Participant accesscode 2222
```

In our sample the subscriber PIN is *admin*, DNIS (DID) number is *8665089020*, the conference number is *267996*, and conference accounts with two roles are created: the host with access code *1111* and the participant with access code *2222*.



This also can be made using Web Administration Interface. See “Web Administration Interface – User Guide”, Chapter 2: Web Administration Interface, Section: Create a Conference Account, if you need assistance in creation of new conference accounts. Created conference accounts can be seen on Figure 5.

24/7 Professional Customer Service  
TOLL FREE 866.508.9020

Subscribers | Conferences | Calls | Reports | DNIS | Preferences | Logout | CONFERENCE APPLIANCE MANAGER

Personal Information

\*PIN:

\*Password:

\*Confirm Password:

\*First Name:

\*Last Name:

Email:

Telephone No:

Grant operator permission

\*The field is mandatory

Conference Accounts

List records: 1/6 << Previous | 1 | Next >>

Conference Number	DNIS	Access Code	Role
<input type="checkbox"/> 267996	(866) 508-9020 - SAMPLE	2222	participant
<input type="checkbox"/> 267996	(866) 508-9020 - SAMPLE	1111	host

Figure 5: Two Created Conference Accounts with Host and Participants Roles

To implement the requested scenario the contents of *callflow.spec* and *script.ael* files should be the following:

#### ***callflow.spec***

```
[handlers]
entry_handler = sample_entry_handler

[attributes]
dnis_authorizemethod = local
```

***script.ael***

```

context sample_entry_handler {
    s => {
        WYDE_Playback(welcome);
        WYDE_Input(enter_accesscode|accesscode|#|12);

        WYDE_AGIRequest(conf_authorize);
        if ("${agi_result}" != "1") {
            Set(DISCONNECT_REASON=Incorrect access code);
            WYDE_Playback(incorrect_accesscode);
            Hangup();
        }

        WYDE_Playback(accesscode_accepted);
        Return();
    }
}

```

Let's consider the implemented logic in details. We should add the following two lines into *callflow.spec* file:

```

[attributes]
dnis_authorizemethod = local

```

i.e. into this file we should add new section named *attributes*. In this section we can define default values of the attributes for this call flow. There are different authorization methods such as authorization via local database, authorization via RADIUS and others. The authorization method can be defined independently for each DNIS via the attribute *dnis\_authorizemethod*. In our example we use authorization via the local database.

The *script.ael* file contains new contents (context) of *sample\_entry\_handler* according to the requested call flow logic.

```

WYDE_Playback(welcome); // Play the prompt from a welcome.ul
file.

```

```

WYDE_Input(enter_accesscode|accesscode|#|12); // Enter
access code. Plays the prompt to enter access code and waits
for the entering of DTMF, the entering terminates when the
key '#' was pressed, the maximum length of entered string is
12 chars. Entered chars without '#' will be put into the
variable named accescode.

```

```
WYDE_AGIRequest(conf_authorize); // Make authorization. A
command WYDE_AGIRequest(conf_authorize) does authorization
request based on called_number and accesscode. If
authorization is successful then the variable agi_result will
be equal 1 and the appropriate values will be assigned to the
variables conf_number, role and others. Otherwise, if
authorization is unsuccessful, the agi_result will be equal
0.
```

```
if ("${agi_result}" != "1") // Check was authorization
successful or not.
```

```
Set(DISCONNECT_REASON=Incorrect access code); // If
authorization was unsuccessful set disconnect reason equal to
"Incorrect access code".
```

```
WYDE_Playback(incorrect_accesscode); // Play the prompt
from a incorrect_accesscode.ul file ("incorrect access code"
message).
```

```
Hangup(); // If authorization was unsuccessful interrupt the
call.
```

```
WYDE_Playback(accesscode_accepted); // Play the prompt from
a accesscode_accepted.ul file ("access code accepted"
message).
```

```
Return(); // Return processing to call flows execution
environment (the call flow engine context).
```

Note. If it was not previously made, do not forget to place *enter\_accesscode.ul*, *incorrect\_accesscode.ul*, *accesscode\_accepted.ul* audio files into *sounds* subfolder of *SAMPLE* folder of the call flow; *welcome.ul* file should be already in this folder after the previous sample.

Because *SAMPLE* call flow has already been created in *Sample 1 – Simple Call Flow without Authorization* now we should only update its attributes in a database; to do that the following command should be executed:

```
wyde callflow-attr-update-db callflow SAMPLE
```

To send the signal on the WYDE bridge to the call flow engine to reload the scripts the following command should be executed:

```
wyde callflow-reload
```

Now if you call to the number 8665089020 you will hear “*welcome*” message and after that you will here “*enter access code*” message. After the access code is entered the authorization will be made – if the access code is valid the call will be connected to the conference either with host role or with participant role depending on access code; if the access code is incorrect the call will be disconnected. For instance if two callers call to 8665089020 number, the first caller has entered host access code 1111 and the second caller has entered participant access code 2222, you will see the calls conference screen

similar to shown on Figure 6. Calls report screen if three calls were made (the first with wrong access code, the second with the host access code and the third with the participant access code) is shown on Figure 7.

WYDE VOICE 24/7 Professional Customer Service TOLL FREE 866.508.9020

Subscribers | Conferences | **Calls** | Reports | DNIS | Preferences | Logout | CONFERENCE APPLIANCE MANAGER

Conference # 267996 open secure: OFF hold: OFF ASN: OFF recording: OFF Dialout

Keyword: First Name Search Number of calls: 2 Set refresh: No Refresh

List records: 1/2 << Previous | 1 | Next >>

Calling Number	Called Number	User Name	Access Code	Call Begin	Duration	Status	Role	Mute	Hold	Q&A
✘	(866) 508-9020	testing	1111	12:56:36	03m:19s	conference	host	<input type="checkbox"/>	<input type="checkbox"/>	
✘	(866) 508-9020	unknown	2222	12:57:05	02m:50s	conference	participant	<input type="checkbox"/>	<input type="checkbox"/>	

Figure 6: Calls Screen for the Conference – Sample 2 Call Flow (with Authorization)

WYDE VOICE 24/7 Professional Customer Service TOLL FREE 866.508.9020

Subscribers | Conferences | Calls | **Reports** | DNIS | Preferences | Logout | CONFERENCE APPLIANCE MANAGER

From: 11/16/2009 To: 11/17/2009 Calls report Create

Number of calls: 11 Total minutes: 93m:10s

List records: 1/11 << Previous | 1 | Next >>

Call Created	Conference Number	Conference Id	Access code	Subscriber Name	Custom Name	Called number	Calling number	Call duration	In conference	Time	Reason	Initiator
< 11/17/2009 12:57:05	267996	27	2222		unknown	(866) 508-9020		42m:07s	42m:07s	11/17/2009 13:39:12	Normal	USER
< 11/17/2009 12:58:38	267996	27	1111		testing	(866) 508-9020		42m:26s	42m:15s	11/17/2009 13:39:02	Normal	USER
< 11/17/2009 12:58:13						(866) 508-9020		00m:12s	00m:00s	11/17/2009 12:58:25	Incorrect access code	BRIDGE

Figure 7: Calls Report – Sample 2 Call Flow (with Authorization)

[Click here to download the Sample 2.](#) The archive files from SAMPLE2 folder should be extracted into SAMPLE folder of your call flow.

### Sample 3 – Call Flow with DTMF Processing

Let's assume that we should add to the previous functionality (Sample 2 – Call Flow with Authorization) DTMP keys processing. For processing of DTMP keys we can either use predefined functions or write own functions. Let's review the call flow scenario where we should use three predefined functions:

- `conference_mute` on press \*1, accessible only for host;
- `conference_lock` on press \*2, accessible only for host;
- `call_mute` on press \*3, accessible for both host and participant.

To implement the requested scenario the contents of `callflow.spec` and `script.ael` files should be the following:

***callflow.spec***

```
[handlers]
entry_handler = sample_entry_handler

[attributes]
dnis_authorizemethod = local
conference_mute_dtmf = h
conference_mute_dtmf_binding = *1
conference_lock_dtmf = h
conference_lock_dtmf_binding = *2
call_mute_dtmf = hp
call_mute_dtmf_binding = *3

[dtmf_commands]
conference_mute_dtmf = conference_mute_switch
conference_lock_dtmf = conference_lock_switch
call_mute_dtmf = call_mute_switch
```

***script.ael***

```
context sample_entry_handler {
    s => {
        WYDE_Playback(welcome);
        WYDE_Input(enter_accesscode|accesscode|#|12);

        WYDE_AGIRequest(conf_authorize);
        if ("${agi_result}" != "1") {
            Set(DISCONNECT_REASON=Incorrect access code);
            WYDE_Playback(incorrect_accesscode);
            Hangup();
        }

        WYDE_Playback(accesscode_accepted);
        Return();
    }
}
```

Let's consider the implemented logic in details. Note that to implement this scenario *script.ael* file was not changed, so we will not describe it here. But we should make the following changes in the *callflow.spec* file:

- We should add into attributes section of the *callflow.spec* file the following attribute definitions:

```
conference_mute_dtmf = h
conference_mute_dtmf_binding = *1
conference_lock_dtmf = h
conference_lock_dtmf_binding = *2
call_mute_dtmf = hp
```

```
call_mute_dtmf_binding = *3
```

As you can see each DTMF handler is being defined by pair of call flow attributes – the policy attribute and the DTMF binding attribute. The name of the policy attribute is `<function_name>_dtmf`; this attribute defines accessibility of function for the role; there could be one of the 3 possible policies: “hpl” (the option is available for hosts (h), participants (p), and listeners (l) of the conference), “hp” (the option is available for hosts (h) and participants (p) of the conference), and “h” (the option is available for conference hosts (h) only). In our sample the policy attributes are `conference_mute_dtmf` (available for hosts), `conference_lock_dtmf` (available for hosts), and `call_mute_dtmf` (available for hosts and participants). The name of the binding attribute is `<function_name>_dtmf_binding`; this attribute defines the sequence of DTMF keys that should be pressed to invoke the function. In our sample the binding attributes are `conference_mute_dtmf_binding` (\*1 should be pressed), `conference_lock_dtmf_binding` (\*2 should be pressed), and `call_mute_dtmf_binding` (\*3 should be pressed).

You can see “Web Administration Interface – User Guide”, Chapter 3: Call Flows, for additional information about call flow attributes.

- We should add new `dtmf_commands` section into `callflow.spec` file with the following contents:

```
[dtmf_commands]
conference_mute_dtmf = conference_mute_switch
conference_lock_dtmf = conference_lock_switch
call_mute_dtmf = call_mute_switch
```

Here we describe the binding of DTMF keys commands and the functions that process these DTMF keys.

These functions `conference_mute_switch`, `call_mute_switch`, `conference_lock_switch` are predefined functions and they are defined in the `/usr/local/DNCA/lib/CallFlows/functions.ael` file. The contents of these functions are shown in Appendix A: Call Flow Library. You can also use this appendix as the set of samples to build your own functions.

Some sound files are required to correct work of these functions; these files should be placed in `sounds` subfolder of the `SAMPLE` call flow folder. That files contain announcements about the changes in the state of the call or the conference.

Function	File Name	Prompt	Message Content
<code>conference_mute_switch</code>	<code>line_mute.ul</code>	<code>line_mute</code>	Muted
<code>conference_mute_switch</code>	<code>line_unmute.ul</code>	<code>line_unmute</code>	Unmuted
<code>conference_mute_switch</code>	<code>gl_mute_open.ul</code>	<code>gl_mute_open</code>	All callers are unmuted
<code>conference_mute_switch</code>	<code>gl_mute_question.ul</code>	<code>gl_mute_question</code>	All callers are muted and they can unmute themselves.
<code>conference_mute_switch</code>	<code>gl_mute_close.ul</code>	<code>gl_mute_close</code>	All callers are muted
<code>conference_lock_switch</code>	<code>conf_secure.ul</code>	<code>conf_secure</code>	The conference has been locked.
<code>conference_lock_switch</code>	<code>conf_open.ul</code>	<code>conf_open</code>	The conference has been unlocked.
<code>call_mute_switch</code>	<code>line_mute.ul</code>	<code>line_mute</code>	Muted
<code>call_mute_switch</code>	<code>line_unmute.ul</code>	<code>line_unmute</code>	Unmuted

Because *callflow.spec* file has been changed now we should update the call flow attributes in a database; to do that the following command should be executed:

```
wyde callflow-attr-update-db callflow SAMPLE
```

If you placed the requested sound files into *sounds* subfolder of the *SAMPLE* call flow folder just now (when you implemented this sample), you should execute the following command:

```
wyde callflow-reload
```

If these files already were in this folder it is not necessary to run this command (because no sounds files were updated and *script.ael* file was not changed as well).

The updated *SAMPLE* call flow with new attributes is shown on Figure 8.

24/7 Professional Customer Service  
TOLL FREE 866.508.9020

Subscribers | Conferences | Calls | Reports | DNIS | Preferences | Logout | CONFERENCE APPLIANCE MANAGER

\*Call Flow Name

\*Directory Path

	Description	Name ▲	Role	Value
<u>call</u>	Mute self DTMF policy	call_mute_dtmf	Conference	<input type="text" value="hp"/>
	Mute self DTMF binding	call_mute_dtmf_binding	DNIS	<input type="text" value="*3"/>
<u>conference</u>	Lock conference DTMF policy	conference_lock_dtmf	Conference	<input type="text" value="h"/>
	Lock conference DTMF binding	conference_lock_dtmf_binding	DNIS	<input type="text" value="*2"/>
	Mute mode DTMF policy	conference_mute_dtmf	Conference	<input type="text" value="h"/>
	Mute mode DTMF binding	conference_mute_dtmf_binding	DNIS	<input type="text" value="*1"/>
<u>dnis</u>	Authorize method	dnis_authorizemethod	DNIS	<input type="text" value="local"/>

**Figure 8: SAMPLE Call Flow with New Lock and Mute Attributes**

Now if you call to the number *8665089020* and connect to the conference

- if you are the host you are able to mute the conference or mute your call or lock the conference;
- if you are the participant you are able to mute/unmute your call only.

For instance if two callers (the host and the participant) call to *8665089020* number and the host has locked the conference, has muted the conference and has mutes himself, you will see the calls conference screen similar to shown on Figure 9.

WYDE VOICE 24/7 Professional Customer Service TOLL FREE 866.508.9020

Subscribers | Conferences | Calls | Reports | DNIS | Preferences | Logout | CONFERENCE APPLIANCE MANAGER

Conference # 267996 relaxed secure: ON hold: OFF ASN: OFF recording: OFF Dialout

Keyword: First Name Search Number of 2 Set refresh: No Refresh

List records: 1/2 << Previous | 1 | Next >>

Calling Number	Called Number	User Name	Access Code	Call Begin	Duration	Status	Role	Mute	Hold	Q&A
	(866) 508-9020	testing	1111	14:11:19	01m:18s	conference	host	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
	(866) 508-9020	unknown	2222	14:11:04	01m:33s	conference	participant	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

Figure 9: Calls Screen for the Conference – Sample 3 Call Flow (with DTMF Processing)

[Click here to download the Sample 3.](#) The archive files from SAMPLE3 folder should be extracted into SAMPLE folder of your call flow.

### Sample 4 – Call Flow with Custom Handlers

Let's assume that we should add to the previous functionality (Sample 3 – Call Flow with DTMF Processing) the following logic:

- before the call joins to the conference the system should tell (pronounce) the number of participants currently connected to this conference;
- on \*4 pressed the system should play the current date and time, the function should be accessible for both host and participant;
- on \*5 pressed the conference recording should be started, if there was the recording in the conference the file with the recording should be copied into `/home/recordings` folder right after the recording is completed.

We can get the number of participants in the conference only after the call has been joined to the conference. At the moment of the `entry_handler` execution the call is not joined to the conference yet and because of that this information is not available in this handler. Therefore it is necessary to define a `welcome_handler` that is being always executed immediately after the call has been joined to the conference.

To play the current date and time we should create the custom call flow attributes pair – the policy attribute and the DTMF binding attribute and write the handler for the DTMF keys pressed. For instance the attributes names are `call_saytime_dtmf` and `call_saytime_dtmf_binding` and the created handler name is `sample_saytime_handler`.

To start and stop the recording we should have call flow attributes pair – the policy attribute and the DTMF binding attribute and use the standard handler for the DTMF keys pressed. In our sample the attributes names are `recording_dtmf` and `recording_dtmf_binding` and the standard handler that should be used is `recording_switch`. To copy the recorded file it is necessary to define a



`recstop_handler` that is being always executed immediately after the recording is completed.

To implement the requested scenario the contents of *callflow.spec* and *script.ael* files should be the following:

***callflow.spec***

```
[handlers]
entry_handler = sample_entry_handler
welcome_handler = sample_welcome_handler
recstop_handler = sample_recstop_handler

[custom_attributes]
call_saytime_dtmf = conference:role:Say time DTMF policy
call_saytime_dtmf_binding = dnis:string:Say time DTMF
binding

[attributes]
dnis_authorizemethod = local
conference_mute_dtmf = h
conference_mute_dtmf_binding = *1
conference_lock_dtmf = h
conference_lock_dtmf_binding = *2
call_mute_dtmf = hp
call_mute_dtmf_binding = *3
call_saytime_dtmf = hp
call_saytime_dtmf_binding = *4
recording_dtmf = h
recording_dtmf_binding = *5
recording_method = local_trusted

[dtmf_commands]
conference_mute_dtmf = conference_mute_switch
conference_lock_dtmf = conference_lock_switch
call_mute_dtmf = call_mute_switch
call_saytime_dtmf = sample_saytime_handler
recording_dtmf = recording_switch
```

**script.ael**

```

context sample_entry_handler {
    s => {
        WYDE_Playback(welcome);
        WYDE_Input(enter_accesscode|accesscode|#|12);

        WYDE_AGIRequest(conf_authorize);
        if ("${agi_result}" != "1") {
            Set(DISCONNECT_REASON=Incorrect access code);
            WYDE_Playback(incorrect_accesscode);
            Hangup();
        }

        WYDE_Playback(accesscode_accepted);
        Return();
    }
}

context sample_welcome_handler {
    s => {

WYDE_Playback(thereare&n:${WYDE_IVRStat(ses_count)}-
1]&callers|d);

        Return();
    }
}

context sample_saytime_handler {
    s => {
        SayUnixTime();

        Return();
    }
}

context sample_recstop_handler {
    s => {
        System(/bin/cp
${VARLIB_DIR}/recordings/${conf_subdir}/${conf_number}/recor
d/${conf_id}.${PREFERED_CODEC} /home/recordings);

        Return();
    }
}

```

Let's consider the implemented logic in details.

- We should add the following line to the `handlers` section of the `callflow.spec` file:

```
welcome_handler = sample_welcome_handler
```

This line defines that `sample_welcome_handler` handler should be used as `welcome_handler` of the call.

- We should also add the following line to the `handlers` section of the `callflow.spec` file:

```
recstop_handler = sample_recstop_handler
```

This line defines that `sample_recstop_handler` handler should be used as `recstop_handler` of the call.

- In addition to create the custom attributes we should add the `custom_attributes` section into the `callflow.spec` file:

```
[custom_attributes]
```

```
call_saytime_dtmf = conference:role:Say time DTMF policy
```

```
call_saytime_dtmf_binding = dnis:string:Say time DTMF binding
```

- and we should add these attributes into `attributes` section:

```
call_saytime_dtmf = hp
```

```
call_saytime_dtmf_binding = *4
```

- and finally we should define the custom handler in `dtmf_commands` section:

```
call_saytime_dtmf = sample_saytime_handler
```

This line defines that `sample_saytime_handler` handler should be used when `*4` (defined in `call_saytime_dtmf_binding` attribute) is pressed.

- To define the recording attributes we should add these attributes into `attributes` section:

```
recording_dtmf = h
```

```
recording_dtmf_binding = *5
```

```
recording_method = local_trusted
```

- and finally we should define the standard handler in `dtmf_commands` section that should be used to start and stop the conference recording:

```
recording_dtmf = recording_switch
```

This line defines that the standard `recording_switch` handler (see Chapter 3: Function Reference, section: DTMF commands, command: `recording_switch`) should be used when `*5` (defined in `recording_dtmf_binding` attribute) is pressed. The recording file will be saved in `/${VARLIB_DIR}/recordings` folder.

Also to the `script.ael` file we should add

- the contents (context) of `sample_welcome_handler`:

```
context sample_welcome_handler {
    s => {
```

```
WYDE_Playback(thereare&n:${WYDE_IVRStat(ses_count)}-
1)&callers|d);
```

```
Return();
```

```
}
```

```

}
• the contents (context) of sample_saytime_handler:
context sample_saytime_handler {
    s => {
        SayUnixTime();

        Return();
    }
}
• and the contents (context) of sample_recstop_handler:
    s => {
        System(/bin/cp
${VARLIB_DIR}/recordings/${conf_subdir}/${conf_number}/record
/${conf_id}.${PREFERRED_CODECS} /home/recordings);

        Return();
    }
}

```

The `WYDE_IVRStat(ses_count)` function returns the number of the participants of the conference. The `WYDE_Playback` function plays the files that are transferred to this function as parameters separated via delimiter '&'. First of all this function plays the message “there are” from the *thereare.ul* file; after that it plays the number of participants of the current conference; and after that it plays message “participants in the conference” from the *callers.ul* file.

Note. To play the numbers it is necessary to create the *digits* subfolder in the *sounds* folder; this folder should contain the audio files that pronounce numbers: 0.ul 1.ul 2.ul 3.ul 4.ul 5.ul 6.ul 7.ul 8.ul 9.ul 10.ul 11.ul 12.ul 13.ul 14.ul 15.ul 16.ul 17.ul 18.ul 19.ul 20.ul 30.ul 40.ul 50.ul 60.ul 70.ul 80.ul 90.ul hundred.ul thousand.ul (i.e. “zero”, “one”, ..., “ten”, “eleven”, “twelve”, ..., “twenty”, “thirty”, ..., “ninety”, “hundred”, “thousand”). See Table 1 for detail information.

The `SayUnixTime()` function is the standard Asterisk function that says a date and/or time to the caller.

Note. To play date and time this function uses some of the sound files stored in */var/lib/asterisk/sounds* to construct a phrase saying the specified date and/or time in the specified format.

The `System` function runs system copy command (*/bin/cp*) and copies the requested recording file into */home/recordings* folder.

Note. To play recording prompts the *sound* folder should contain the following sound files: *start\_recording.ul* (“this conference is now being recorded” message), *stop\_recording.ul* (“this conference is no longer being recorded” message), *recording\_dtmf.ul* (“to record a conference press” message), *reconfirm.ul* (“to begin recording this conference press 1, to return – press \*” message), *rec\_stop\_confirm.ul* (“press 1 to stop recording this conference, to return – press \*” message).

To update the SAMPLE call flow attributes in a database the following command should be executed:

```
wyde callflow-attr-update-db callflow SAMPLE
```

To send the signal on the WYDE bridge to the call flow engine to reload the scripts the following command should be executed:

```
wyde callflow-reload
```

The updated SAMPLE call flow with new attributes is shown on Figure 10.

The screenshot shows the WYDE VOICE CONFERENCE APPLIANCE MANAGER interface. At the top, there is a navigation bar with links: Subscribers | Conferences | Calls | Reports | **DNIS** | Preferences | Logout | CONFERENCE APPLIANCE MANAGER. The main content area displays the configuration for a call flow named 'SAMPLE'.

At the top of the configuration, there are two input fields:
 

- \*Call Flow Name: SAMPLE
- \*Directory Path: /usr/local/DNCA/callflows/SAMPLE

Below these fields is a table of attributes. The table has four columns: Description, Name, Role, and Value. The attributes are grouped into three sections: call, conference, and dnis.

Description	Name	Role	Value
<b>call</b>			
Mute self DTMF policy	call_mute_dtmf	Conference	hp
Mute self DTMF binding	call_mute_dtmf_binding	DNIS	*3
Say time DTMF policy	call_saytime_dtmf	Conference	hp
Say time DTMF binding	call_saytime_dtmf_binding	DNIS	*4
<b>conference</b>			
Lock conference DTMF policy	conference_lock_dtmf	Conference	h
Lock conference DTMF binding	conference_lock_dtmf_binding	DNIS	*2
Mute mode DTMF policy	conference_mute_dtmf	Conference	h
Mute mode DTMF binding	conference_mute_dtmf_binding	DNIS	*1
<b>dnis</b>			
Authorize method	dnis_authorizemethod	DNIS	local

At the bottom of the table, there are two buttons: Update and Cancel.

**Figure 10: SAMPLE Call Flow with New Custom Attributes**

Now if you call to the number 8665089020 after you entered the access code and connected to the conference you can hear “*there are N participants in the conference*” message (where  $N$  – is the number of participants that were connected to the conference before you). When you joined to the conference if you press \*4 you will here the current date and time prompt.

[Click here to download the Sample 4.](#) The archive files from SAMPLE4 folder should be extracted into SAMPLE folder of your call flow.

## Chapter 3: Function Reference

### *Callback Handlers*

Callbacks are using for the call flow customization. If a callback handler does not defined in the call flow then the call flow engine are using a default handler. The list of all callback handlers sorted in order of execution is the following:

- `entry_handler`
- `fastjoin_handler`
- `waitmoderator_handler`
- `holdline_handler`
- `welcome_handler`
- `gotomp_handler`

The following callback handlers are being asynchronously executed when relevant events occurred:

- `recstop_handler`
- `terminate_handler`
- `hangup_handler`

All these handlers are described in this section of the guide.

#### **entry\_handler**

This handler is being called immediately when the call has been received by WYDE core. At the moment of execution the session ID and the DID attributes are already assigned for the call.

The handler must be defined by user to inform the system to what conference the call should be joined, i.e. to define the conference number for the call.

If this handler is not written the default handler drops the call.

#### **fastjoin\_handler**

This handler is being called instead of `entry_handler` if the access code and/or the role are specified during the call.

The default handler tries to authorize the call using the specified access code and the role. If the access code is wrong the call is being dropped. For call flows without authorization the role parameter determines what role (i.e. Moderator, Participant, Listener) should be assigned to the call (the default role is participant); for call flows with authorization the role parameter is being ignored. If authorization is successful the call is being connected to the conference and the subsequent handlers are being called.

**waitmoderator\_handler**

This handler is being called after the call has been joined to the conference if the attribute `conference_start_how` equals “moderator” and the call has Participant role and there are no any other moderator calls joined to the conference.

The default handler holds the call in state *MusicOnHold* until the moderator connected to the conference.

**holdline\_handler**

This handler is being called after the call has been joined to the conference if condition for `waitmoderator_handler` is not satisfied and the call has “on hold” state.

The default handler holds the call while its status is “on hold”.

**welcome\_handler**

This handler is being always called after the call has been joined to the conference (after `waitmoderator_handler` and `holdline_handler`).

The default handler does nothing.

**gotomp\_handler**

This handler is being called before the call is being connected to the conference, i.e. before the call is forwarded to the Media Processor (MP).

The default handler holds the call in the state *MusicOnHold* while there is only one call in the conference and there is no activated recording or broadcasting. Before forwarding the call to the MP it plays the prompt `announceyourself` or `line_mute` if the call is muted.

**recstop\_handler**

This handler is being called when the conference recording is completed (the recording has been stopped). This handler is being used to get the access to the recorded file right after the recording has been completed. For instance in this handler it is possible to copy the recorded file to the remote server or another folder. The sample of the handler that copies the recorded file into specified folder is shown in Sample 4 – Call Flow with Custom Handlers.

The following variables can be used in this handler:

- `${VARLIB_DIR}/recordings/${conf_subdir}/${conf_number}/record/${conf_id}.${PREFERED_CODEC}` – the full path to the recorded file;
- `${conf_id}.${PREFERED_CODEC}` – the recorded file name, where
  - `${conf_id}` – the current conference identifier;
  - `${PREFERED_CODEC}` – the preferred (i.e. top-priority) codec in the system (usually ul).

Note: if during the conference the recording was started and stopped multiple times each new portion of the recording is being added to the end of the recording file, i.e. there is only one recording file for the specific conference.

This handler is being executed asynchronously in separate thread and does not affect on the conference call and possible subsequent recordings.

The default handler does nothing.

### **terminate\_handler**

This handler is being called when the WYDE core has sent the signal TERMINATE to the call (i.e. when the call is being terminated by the bridge). The variable DISCONNECT\_REASON specifies a signal reason. Possible values:

- `moderator_left` – moderator left the conference the attribute `conference_stop_how` equals “moderator”;
- `moderator_not_up` – the attribute `conference_start_how` equals “moderator” and time of moderator waiting is exceeded;
- `conference_maxcalls` – the maximum number of calls in the conference is exceeded;
- `conference_maxduration` – the maximum duration of conference is exceeded.

This handler is being called only if one of the reasons described above took place. Note this event occurred only when the bridge terminates the call, not when the user terminates the call. If there was abnormal call termination this handler is not being called. This handler always called before `hangup_handler` call.

For instance this handler can be used to play specific message for the callers. In addition this handler can be used to manage should the call be hanged up (the handler should call `Hangup()` function for this purpose) or should not (the handler should call `Return()` function for this purpose).

The default handler drops the call.

### **hangup\_handler**

This handler is being called when the call is hanged up either by the user or by the bridge. If the call is being terminated by the bridge under one of the reasons described above for `terminate_handler`) the `terminate_handler` is being called prior to this handler.

The default handler does nothing.

## ***DTMF commands***

### **call\_participantsnumber**

Plays number of participants in the conference.  
Required prompts:



- `you_are_the_only` – Only one participant;
- `hereare` – There are;
- `thereare2` – ... participants in a conference;

### **call\_exit**

Requests confirmation on a call end. Ends the call if it has been confirmed.

Required prompts:

- `conf_exit_confirm` – You entered the key to exit from the conference if this is your intention please press 1, otherwise please press 2
- `conf_exit` – Thank you for using our service;
- `goodbye` – Goodbye;

### **call\_instructions**

Plays instructions about available DTMF functions.

Prompts:

- `instructions_begin` – The following conference commands are available...
- `<function_name>_dtmf` – The description of the function.

### **call\_mute\_switch**

Mutes/un-mutes the call. If a Q&A mode is on then adds or removes the call to/from a queue.

Prompts:

- `line_mute` – muted;
- `line_unmute` – un-muted;
- `can_not_unmute` – The conference host has muted the conference, this line can not be un-muted;
- `qa_req_submit_confirm` – If you'd like to place yourself to the queue to ask a question to the moderator press 1, otherwise press 2;
- `qa_req_remove_confirm` – If you'd like to remove yourself from the queue to ask a question to the moderator press 1, otherwise press 2;
- `qa_req_queued` – Your request has been received;
- `qa_req_removed` – Your request has been removed from queue.

### **conference\_mute\_switch**

The switch of conference mute and Q&A mode (opened/relaxed/strict/question).

Prompts:

- `line_mute` – muted;
- `line_unmute` – un-muted;
- `gl_mute_open` – All callers are un-muted;
- `gl_mute_question` – All callers are muted and they can un-mute themselves by pressing \*6;
- `gl_mute_close` – All callers are muted.

**conference\_lock\_switch**

Blocking the conference from new participants connecting.

Prompts:

- `conf_secure` – Secured conference;
- `conf_open` – Open conference.

**conference\_entryexittones\_switch**

The switch of entry/exit tones.

Prompts:

- `en_off_ex_on` – Entry chimes the off, exit chimes the on;
- `en_on_ex_on` – Entry chimes the on, exit chimes the on;
- `en_off_ex_off` – Entry chimes the off, exit chimes the off;
- `en_on_ex_off` – Entry chimes the on, exit chimes the off.

**conference\_qa\_moderator**

Manages Q&A sessions.

Prompts:

- `qa_main_menu` – To start Q&A session or to switch to the next question press 1 to mute or unmute questioner press 2 to end session press 3.
- `qa_talk_enabled` – Q&A session started.
- `qa_talk_disabled` – Q&A session is over.
- `qa_talk_notfound` – There are no questions pending yet.
- `qa_talk_enabled_announce` – Now you can ask you question.
- `qa_talk_disabled_announce` – This line is now muted.
- `qa_talk_muted` – Questioner line now muted.
- `qa_talk_unmuted` – Questioner line now unmuted.
- `qa_talk_next` – Next questioner.

**recording\_switch**

Starts/stops of the conference recording.

Prompts:

- `reconfirm` – To start conference recording press 1, to return to the conference press \*;
- `rec_stop_confirm` – To stop conference recording press 1, to return to the conference press \*;
- `start_recording` – This conference is being recorded;
- `stop_recording` – Recording has been stopped.

***Dialplan commands and functions***

In callback handlers as dialplan commands and functions you can use either standard Asterisk functions or commands and functions provided by WYDE software.

For example you can make HTTP calls to external servers using `CURL Asterisk` function. Also you can make requests via another protocols (not only HTTP) using different Asterisk functions. In addition you can make your own AGI server or AGI script and make requests via AGI protocol.

You can read Asterisk documentation for the detail list of Asterisk functions.

This section of the guide describes `WYDE` commands and functions that could be used in callback handlers.

## **WYDE\_Playback**

Syntax:

```
WYDE_Playback(parameter[&parameter2...] [|options])
```

Plays the prompt or the list of prompts. If the option “d” is specified then breaks playing on DTMF and stores the DTMF key into `DTMF_INPUT` variable.

Parameters could specify a format qualifier. If the format qualifier is not specified then the parameter will be interpreted as the file name from the sounds subdirectory.

Possible format qualifiers:

- `n`: – play the parameter as a decimal number;
- `d`: – play the parameter as a sequence of decimal digits;
- `date`: – play the parameter as a date.

## **WYDE\_Input**

Syntax:

```
WYDE_Input(prompt|variable [|ends [|maxlen [|timeout [|retries]]]
]);
```

Plays the prompt and waits for DTMF input. Input terminates if pressed one of key specified in the ends. Entered string without the input terminator will be placed into the variable.

Parameters:

- `prompt` – the prompt file name;
- `variable` – the name of the variable where the entered string will be placed;
- `ends` – the list of the input terminators separated by '^' (for example `*^#`);
- `maxlen` – the maximum length of input;
- `timeout` – the maximum time of waiting of input;
- `retries` – the maximum number of retry attempts.

## **WYDE\_Choice**

Syntax:

```
WYDE_Choice(prompt|variable|digits^digits^... [|invalid_input [|retries [|timeout]]])
```

Plays the prompt and waits for the input of the on of the possible answers.

Parameters:

- `prompt` – the prompt file name;
- `variable` – the name of the variable where the entered choice will be placed;

- `digits^digits^...` – the list of possible choices of the answer, choice can have one or more characters;
- `invalid_input` – the prompt which will be played if the impossible choice was entered;
- `retries` – the maximum number of retries of input;
- `timeout` – the maximum time of waiting of input;

### WYDE\_AGIRequest

Syntax:

```
WYDE_AGIRequest(request[,parameters_list])
```

Executes the request to the FastAGI server.

Sets up the `agi_result=1` if the request has been completed successfully, otherwise `agi_result=0`.

### WYDE\_IVRStat

Syntax:

```
WYDE_IVRStat(variable)
```

A dialplan function. Returns the current states of the call or of the conference. This function returns correct values only if the call has already been joined to the conference.

Parameters:

- `inconference` – equals 1 if the call already joined to the conference, else – 0;
- `ses_count` – the number of calls in the conference;
- `regular_ses_count` – the number of the regular calls in the conference (non control);
- `ivr_ses_count` – the number of calls of the conference being at present on the IVR;
- `mp_ses_count` – the number of calls of the conference being at present on the MP;
- `noaudio_count` – the number of control calls in the conference;
- `mp_noaudio_count` – the number of control calls of conference being at present on the MP;
- `host_ses_count` – the number of moderator's calls in the conference;
- `participant_ses_count` – the number of participant's calls in the conference;
- `listener_ses_count` – the number of listener's calls in the conference;
- `moh_required` – equals 1 if need to hold the call on MOH, otherwise – 0;
- `secure` – equals 1 if the conference is blocked for new participants connecting, otherwise – 0;
- `recording` – equals 1 if recording is started, otherwise – 0;
- `broadcast` – equals 1 if broadcast is started, otherwise – 0;
- `lecture_mode` – returns the current value of lecture mode (False/Relaxed/Strict);
- `control` – equals 1 if the call is control call, otherwise – 0;
- `role` – returns a role of the call (Host/Participant/Listener);
- `realtime` – equals 1 if the RT protocol is activated for the call;
- `mute` – equals 1 if the call is muted;
- `hold` – equals 1 if the call is on hold;

- `entry_tones` – equals 1 if the entry tones are on for the conference, otherwise – 0;
- `exit_tones` – equals 1 if the exit tones are on for the conference, otherwise – 0;
- `qa_request` – equals 1 if the call is waits for the Q&A session;
- `qa_mode` – equals 1 if the Q&A session is activated for the call;
- `operator_mode` – equals 1 if the call waits for the operator or is connected to the operator;
- `operator_wait` – equals 1 if the call waits for the operator;
- `operator_talk` – equals 1 if the call is connected to the operator.

### **WYDE\_IVRConfStat**

Syntax:

`WYDE_IVRConfStat(variable)`

A dialplan function. Returns the current states of the conference referenced by the variable `conf_number`. The call can be either joined or does not joined to this conference.

Parameters:

- `ses_count` – the number of calls in the conference;
- `regular_ses_count` – the number of the regular calls in the conference (non control);
- `ivr_ses_count` – the number of calls of conference being at present on the IVR;
- `mp_ses_count` – the number of calls of conference being at present on the MP;
- `noaudio_count` – the number of control calls in the conference;
- `host_ses_count` – the number of moderator's calls in the conference;
- `participant_ses_count` – the number of participant's calls in the conference;
- `listener_ses_count` – the number of listener's calls in the conference;
- `secure` – equals 1 if the conference is blocked for new participants connecting, otherwise – 0;
- `recording` – equals 1 if the recording is started, otherwise – 0;
- `broadcast` – equals 1 if the broadcasting is started, otherwise – 0.

### **WYDE\_IVRVar**

Syntax:

`WYDE_IVRConfStat(<variable>)`

A dialplan function. Reads/writes the variables of the conference.

### **WYDE\_IVRCheckRole**

Syntax:

`WYDE_IVRCheckRole()`

Checks if the feature is accessible for the role.

## Appendix A: Call Flow Library

### */usr/local/DNCA/lib/CallFlows/functions.ael*

```
//-----
//      Proc Events
//-----

// assign new audiokey for the call
context call_associate {
    s => {
        Set(event_args=${FILTER(0123456789|${ses_event_param})});
        GoSub(call_set_bundle|s|1);
        Return();
    };
};

// assign new audiokey for the call
context call_set_bundle {
    s => {
        WYDE_IVRHelper(conf_command|set_bundle|${event_args});
        Return();
    };
};

// play announce to the call
context call_play_announce {
    s => {
        WYDE_Playback(${event_args}|d);
        Return();
    }
}

// play number of participants in the conference
context call_participantsnumber {
    s => {
        Set(count=${WYDE_IVRStat(regular_ses_count)});
        if ("${count}" = "1") {
            WYDE_Playback(you_are_the_only|d);
        } else {
            WYDE_Playback(thereare&n:${count}&thereare2|d);
        };
        Return();
    };
}

// play number of participants in the conference (more simple variant)
context call_participantsnumber_simple {
    s => {
        WYDE_Playback(thereare&n:${WYDE_IVRStat(regular_ses_count)}&thereare2|d);
        Return();
    }
}

// exit from the conference
context call_exit {
    s => {
        WYDE_IVRHelper(musiconhold_start);
        WYDE_Choice(conf_exit_confirm|choice|1^2|invalid_input|1);
        if( "${choice}" = "1") {
            WYDE_Playback(conf_exit&goodbye|d);
            Hangup();
        };
        Return();
    };
};

// play instructions about accessible DTMF commands
```

```

context call_instructions {
    s => {
        WYDE_IVRHelper(play_instructions);
        Return();
    };
}

// mute/unmute group of participants in the conference
context conference_mute_switch {
    s => {
        if ( "${WYDE_IVRStat(qa_mode)}" != "1" ) {
            WYDE_IVRHelper(conf_command|mute_group|switch|Participant);
            if( "${mutegroup_moderator_enabled}" = "y" ) {
                Set(NO_ANNOUNCE=1);
                WYDE_IVRHelper(conf_command|mute_group|switch|Host);
                Set(NO_ANNOUNCE=);
            }
        } else {
            WYDE_Playback(unaccessible_command|d);
        }
        Return();
    }
}

// mute/unmute call
context call_mute_switch {
    s => {
        if( "${WYDE_IVRStat(role)}" != "Host" ) {
            if( "${WYDE_IVRStat(qa_mode)}" = "1" ) {
                SetIfVar(accept_key=qa_accept_key:1);
                SetIfVar(cancel_key=qa_cancel_key:2);

                if( "${WYDE_IVRStat(qa_request)}" = "0" ) {
                    Set(ARRAY(input_prompt,command)=qa_req_submit_confirm\,start);
                } else {
                    if( "${WYDE_IVRStat(qa_talk)}" = "1" ) {
                        WYDE_IVRHelper(conf_command|mute|switch|self);
                        Return();
                    }
                }

                Set(ARRAY(input_prompt,command)=qa_req_remove_confirm\,stop);
            }

            WYDE_Choice(${input_prompt}|choice|${accept_key}^${cancel_key}|invalid_input|3);
            if( "${choice}" = "${accept_key}" ) {
                WYDE_IVRHelper(conf_command|qa_request|${command});
            }
        } else if( "${WYDE_IVRStat(lecture_mode)}" != "Strict" ) {
            WYDE_IVRHelper(conf_command|mute|switch|self);
        } else {
            WYDE_Playback(can_not_unmute|d);
        }
    } else {
        WYDE_IVRHelper(conf_command|mute|switch|self);
    }
    Return();
};
}

// mute/unmute call (simple variant)
context call_mute {
    s => {
        WYDE_IVRHelper(conf_command|mute|${event_args});
        Return();
    };
}

```

```

// hold line
context call_hold {
    s => {
        WYDE_IVRHelper(conf_command|hold|${event_args});
        if( "${WYDE_IVRStat(hold)}" = "1" ) {
            WYDE_IVRHelper(musiconhold_start);
        }
        Return();
    };
}

// Q&A request
context call_qa_request {
    s => {
        WYDE_IVRHelper(conf_command|qa_request|${event_args});
        Return();
    };
}

// wait while moderator come to the conference
context call_wait_moderator {
    s => {
        WYDE_IVRHelper(conf_command|wait_moderator|${event_args});
        Return();
    };
}

// lock/unlock conference
context conference_lock_switch {
    s => {
        WYDE_IVRHelper(conf_command|secure|secure_switch);
        Return();
    };
};

// switch the entry/exit tones status
context conference_entryexittones_switch {
    s => {
        WYDE_IVRHelper(conf_command|entryexit_tones);
        Return();
    };
};

// set jobcode for the conference
context conference_jobcode {
    s => {
        WYDE_Input(enter_jobcode|value|#|16);
        if( "${value}" != "" ) {
            WYDE_Playback(you_entered&d:${value}|d);
            WYDE_IVRHelper(conf_command|conf_set|job_code|${value});
        }
        Return();
    };
}

// start/stop recording
context recording_switch {
    s => {
        // Recording
        WYDE_IVRHelper(musiconhold_start);

        if( "${WYDE_IVRStat(recording)}" != "1" ) {
            Set(prompt=reconfirm|helper=start|recording_originator=1);
        } else {
            Set(prompt=rec_stop_confirm|helper=stop);
        }

        if("${recording_auth}" = "1" ||
            "${recording_method}" = "remote_trusted" || "${recording_method}" =
            "local_trusted") {
            SetIfVar(accept_key=recording_accept_key:1);
        }
    };
}

```



```

        SetIfVar(cancel_key=recording_cancel_key:2);

WYDE_Choice(${prompt}|choice|${accept_key}^${cancel_key}|invalid_input|3);
    if( "${choice}" = "${cancel_key}" || "${choice}" = "" ) {
        Return();
    };
} else {
    Set(INPUT_ENDER=|count=0|recording_passcode=);
    while( ${count} < 3 && "${recording_passcode}" = "" ) {
        WYDE_Input(subscriber_pin_prompt|recording_passcode|#|10);
        NoOp(${recording_passcode});
        count=${count}+1;
    }

    if ("${recording_passcode}" = "" ) {
        Return();
    }

    GoSub(recording_auth|s|1);
    if( "${recording_auth}" != "1" ) {
        Return();
    }
}

WYDE_IVRHelper(conf_command|recording|${helper}|${accesscode}|${recording_passcode})
;
    Set(inprogress=1);
    while( ${inprogress} > 0 ) {
        WYDE_IVRHelper(wait_event);
        GoSub(core_proc_events|s|1);
    }
    Return();
}

auth_error => {
    if( "${recording_auth}" = "-1" ) {
        WYDE_Playback(subscriber_pin_incorrect|d);
        Return();
    } else if ( "${recording_auth}" = "-2" ) {
        WYDE_Playback(service-unavail|d);
        Return();
    }
}

}

// initiate outgoing call
context dialout {
    s => {
        WYDE_AGIRequest(get_dialout_accesscode);
        if ("${dialout_accesscode}" = "" ) {
            WYDE_IVRHelper(conf_command|ses_get|accesscode);
            Set(dialout_accesscode=${VARIABLE_VALUE});
        }

        WYDE_IVRHelper(dialout|${phone_number}|60|${conf_number}|${called_number}|core_conf_
enter_participant^s^1|${dialout_accesscode}|Participant|${dialout_mode}tAC);
        if( "${DIALSTATUS}" != "JOINPEER" && "${DIALSTATUS}" != "DISCONNECT" &&
"${DIALSTATUS}" != "ASYNC" ) {
            WYDE_Playback(party_did_not_answer);
        }
        Return();
    }
}

// set new role for the call
context call_setrole {
    s => {
        Set(accesscode=${event_args});
        WYDE_AGIRequest(conf_authorize);
    }
}

```

```

        if( "${agi_result}" == "1" ) {
            WYDE_IVRHelper(set_role|${role});
        }
        Return();
    };
}

// set custom name
context call_set_customname {
    s => {
        if( ${WYDE_IVRStat(inconference)} = 1 ) {
            WYDE_IVRHelper(conf_command|ses_set|customname|${event_args});
        }
        else {
            WYDE_IVRHelper(ses_command|ses_set|customname|${event_args});
        }
        Return();
    };
}

// set environment variables
context call_setvars {
    s => {
        Set(${event_args});
        Return();
    }
}

// call pause
context call_pause {
    s => {
        Set(inprogress=1);
        while( ${inprogress} > 0 ) {
            WYDE_IVRHelper(wait_event);
            GoSub(core_proc_events|s|1);
        }
        Return();
    }
}

// manage Q&A session
context conference_qa_moderator {
    s => {
        if( "${conference_qa_dtmf}" != "" ) {
            WYDE_Playback(sil&sil&sil&sil|d);
            if( "${DTMF_INPUT}" = "" ) {
                WYDE_Choice(qa_main_menu|choice|1^2^3^4^5^*|invalid_input|3);
            } else {
                Set(choice=${DTMF_INPUT});
            }

            if ( "${choice}" = "1" ) {
                WYDE_IVRHelper(conf_command|qa_mode|start);
            } else if( "${choice}" = "2" ) {
                WYDE_IVRHelper(conf_command|qa_talk|ivr);
            } else if( "${choice}" = "3" ) {
                WYDE_IVRHelper(conf_command|qa_mode|stop);
            } else if( "${choice}" = "4" ) {
                WYDE_IVRHelper(conf_command|qa_mute);
            } else if( "${choice}" = "5" ) {
                WYDE_IVRHelper(conf_command|qa_mode|clear);
            }
        };
        Return();
    }
}
}

```

## **Appendix B: Support Resources**

If you have difficulty with this guide and any of the procedures listed herein, please contact us using the following support resources.

### ***Support Documentation***

In addition to this Guide, you may obtain other WYDE Voice documentation from WYDE Voice or from the support section of <http://www.wydevoice.com/>.

### ***Web Support***

Our support website is available 24 hours a day, 7 days a week, and 365 days a year at <http://www.wydevoice.com>. You may download patches, support documentation and other technical support information.

### ***Telephone Support***

For difficulties with any procedures described in this Guide, please contact us at 866-508-9020 during our normal phone support hours of 7:00 am to 6:00 pm Pacific Standard Time (PST). An engineer will respond to your inquiry within 24 hours.

### ***Email Support***

You may also email us your questions at [support@wydevoice.com](mailto:support@wydevoice.com). We will respond to your question within 24 hours.