VOICE Seeker & VoiceSpot application note for i.MX

GOAL:

- User Guide which explains how to use Voice Seeker + VoiceSpot on the 8MP
- Evaluate public library (beamforming)
- Integrate, and use the library with AEC
- Running the low power voice demo

Feedback

To be more generic as possible

8mp would be an example, use what is already included on BSP

Split or show the customer all options

Contents

Contents

Introduction

NXP have created a Voice Processing environment which is in a continuously improvement. This environment allow customers to use it as a for their application with low latency, low false-positives, low energy consumption and with the minimum effort.

VoiceSeeker and VoiceSpot are some of the tools NXP has to offer for Voice Processing. These can be used on most of the i.MX8M family, with low CPU usage.

Purpose

This document introduces VoiceSeeker, a library providing high resolution beamforming and multi-channel Acoustic Echo Cancellation (AEC), without requiring a predetermined fixed microphone geometry which translates to more flexibility on board design.

Alongside VoiceSeeker, the document presents VoiceSpot, a speech detection engine which doesn't require to be recompiled in order to work with different models. Besides that, it fits perfectly with VoiceSeeker since both of them were designed to work together. VoiceSeeker is in charge of removing the noise from the audio so that VoiceSpot only has to focus on detecting the wake-word. If a wake-word is detected VoiceSpot will tell VoiceSeeker where to focus it's attention.

The main objectives of this document are:

- Full understanding on what VoiceSeeker, VoiceSpot and AFE are.
- Configure VoiceSeeker and VoiceSpot as well as their dependencies.
- Distinguish the difference between default and external configuration.
- Understanding the calibration process.

Overview

VoiceSeeker is a library that can does some feature with the raw audio. On of the feature is removing unwanted noise from the audio that is being captured by the microphones. The library allows the user to clean the microphone's input, leaving the signal with just the artifacts of human voice. Then having a cleaned input ease the application to achieve a higher accuracy doing the speech processing or any other processing that the user requires. NXP provides a wrapper around the library to make the experience with the library more friendly and easy to use. This wrapper is compiled and built as a shared library.

The shared object comes pre-installed on most of the NXP's Linux Distribution so the interested can have a quick taste of some capabilities of the library. However, sometimes the application being built requires some different configurations than the defaults, requiring re-build the library, so it can fit with the user requirements. Some of those times, changing VoiceSeeker might require changing the audio-front-end (AFE) code too.

AFE is another wrapper NXP has that allows to select the desire engine for voice processing at a run time. For example, selecting VoiceSeeker or Conversa. An overview of AFE will be given in the following chapters and will be focused on the use case where it loads the VoiceSeeker library, and its wrapper. For further uses case of the AFE please visit the TODO.md file on GitHub.

Running in another process, VoiceSpot can be found, waiting for the output of VoiceSeeker to detect the spoken word. The spoken word can be either the wake-word or the voice-command itself. If a wake-word is found on the stream VoiceSpot can notify other applications (such as VIT) so they can be aware if a voice command is being spoken. Or, if it is a command, it can also decode it and process it, with the correct configuration. This document only covers how to setup VoiceSeeker and explain how VoiceSpot integrates with VoiceSeeker. For more details information about VoiceSpot APIs can be found in its documentation.

VIT (Voice Intelligent Technology) is another NXP product to handle voice-command which is not part of the scope of the document, but it can be integrated with VoiceSeeker and VoiceSpot (see section VoiceSpot on Arch). If you are interested in learning more about VIT, please visit its website or download its Application Note.

Materials

These are the hardware and software requirements that are needed to properly follow up this document.

Software Requirements

- Linux i.MX Linux BSP version 6.6.3 or above (the recommended version or any other can be downloaded from the official web site at nxp.com).
- Audacity or any similar software.

For more information about how to flash the board and get it ready to run the BSP, please follow i.MX Linux User's Guide.

Hardware Requirements

- i.MX8MP EVK Board.
- 8MIC-RPI-MX8 Board.
- Speakers with aux connection.
- USB-A to USB-C Cable.
- USB-A to USB-B Micro cable.
- 3.5mm jack audio cable.

If there are any doubts about the hardware setup there is the i.MX 8M Plus EVK Quick Start Guide that explain how to download and boot the board.

Voice-UI framework

Integration to NXP Linux BSP

Just as VoiceSeeker Wrapper is pre-installed, the following files and binaries are on each BSP release. Keep in mind that those versions are a demo of the final product. If you are interested in buying the full version of the binaries, please contact our team at **Voice@nxp.com.**

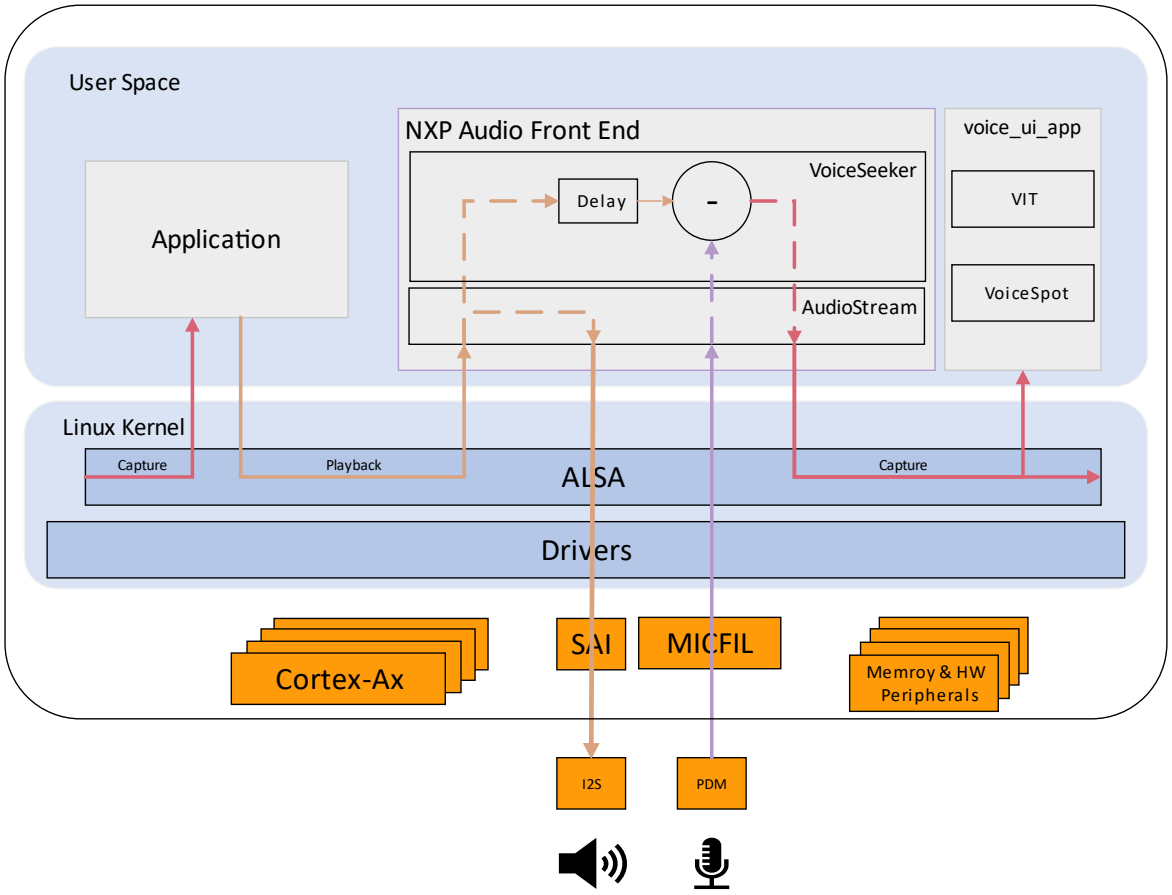
Architecture

VoiceSeeker and VoiceSpot were designed to work independently from one another. However, it is recommended to used together to achieve the best behavior of each library. If VoiceSeeker is working alone, you might want to implement the mechanism to notify VoiceSeeker when a wake word is detected, to reach the best performance on the beamforming. The reason to do so is that each library is running on separate process and VoiceSeeker needs some feedback to be able to adjust to the incoming data. However, if they are running together this mechanism is already implemented on the wrappers on which they are running.

The binaries that comes on the Linux release of each quartet are to demonstrate a use case where a simple Voice Assistances is present, or when is not, where user might want some commands to be processed on the unit rather than going online for the representation of the given

command or when there is no internet and user still wants some commands to always work regardless of the internet connection. The offline processing of the commands is done thanks to VIT, which is running alongside VoiceSpot on voice_ui_app.

The architecture of the full use case described on this document is shown on the following diagram. The use case shown in the document is the base for more complex applications as the once mentioned earlier.



As you can see, the responsible of starting the playback stream is another independent application which needs to use the default ALSA device so AFE can manage the incoming, out coming data, and to store the data being played through the speakers. Assuming the asound.conf has a compatible configuration for AFE. Thus**, AFE must be started before the beginning of the stream, if AFE is not running then the audio won't be heard through the speakers and no error will be promoted on the terminal.** In the other hand, if the asound.conf has not a compatible configuration (AFE won't be able to open its devices), AFE would throw and error and stop its execution. The compatible asound.conf file will be covered later on in the document.

When VoiceSeeker (which is running on AFE) finishes its process VoiceSpot will be notified to look for a wake word in another different thread. Finally, if a wake word is found VoiceSpot can notify a third application for the processing of the voice command and will give some feedback to VoiceSeeker to adjust the beamforming. Since VoiceSpot is the last process on this flow it should

be the first to initialize before even AFE. The initialization process of this uses case or any other similar uses case would be as follow:

1. Start VoiceSpot Library.
2. Start AFE with Voice Seeker for AEC and beamforming.
3. Run the upper application.

External References Signal

This architecture is useful when there is no extra post-processing beyond the AFE (the data handled by AFE is the same data played on the speakers), or it is not a real-time system where meeting a deadline is critical. On those scenarios the architecture of system might require some changes. Fortunately AFE, can handle those scenarios too.

The initialization process would be similar to the above uses case unless you are using a RTOS for real time process. On this scenario you might want to be sure that the RTOS is up and running and on Linux side run VoiceSpot, and AFE as mentioned above.

Both scenarios will be covered on this documents (a loopback and a external configuration). The external configuration means that the audio being played is not being generated on the user space, instead, the audio source is coming from a soundcard and not from an application of user space.

NXP AFE

AFE was created as a solution for Voice Assistant applications, where you need to be able to control the audio inputs and outputs of the system. The output signal must be stored as a reference to filter out noise. The incoming signal needs to be cleaned up before going to a deeper process. Then both signals need to get into a first stage of signal processing where it will filter out noise, and other disturbances generating a third signal. This process is commonly known as Audio Front End.

What AFE does is extract the stream control of the buffers out of the process. This means you can have any library with any process for cleaning up the signal before the actual voice processing. AFE will be calling the process when both buffers are ready to be processed, so the library does not need to control the stream, removing complexity to it.

It is important to fully understand what AFE does. AFE only controls the streams, it does not even fix the delay between the streams (acoustic delay). Keep this in mind because you might require having a mechanism to solve such a delay.

Therefore, AFE on its own does not do any processing to the audio. It needs an additional library for cleaning up the signals. It can load any library that implements the SignalProcessorImplementation class, and it is located at ***/usr/lib/nxp-afe*** . If the library follows these two rules, then AFE would be able to load it and work with it. The way of telling AFE which engine to use is by giving its name as an argument (just the name, no need of giving the full path, nor the extension). For more information on how to use AFE with different libraries please read the TODO.md file.

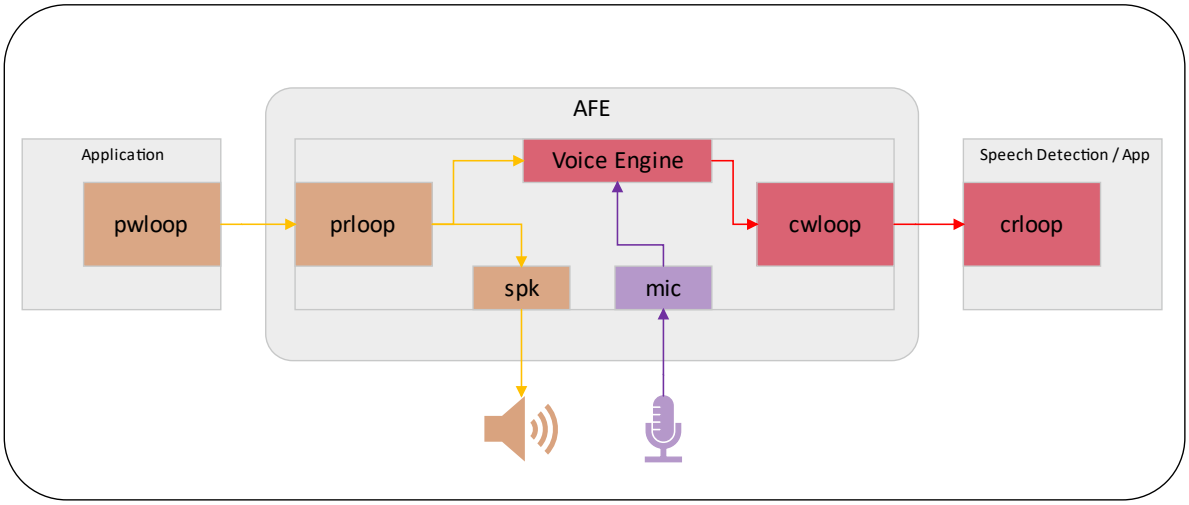
asound.conf

There is a file that AFE relies on, which is the asound.conf. This file is where ALSA (Advanced Linux Sound Architecture) devices are defined and can be opened through the ALSA APIs

(Application Programming Interfaces). AFE uses those APIs to control the data flow ensuring there is no data lost.

On the file there are six main devices. Four of those devices are controlled by AFE and the last two of them are the mainstream devices. One device is used for playback, the application being on development will use this device to play audio though the speakers. The other device is the AFE output, used for capture. Device which should be used to get the noise-free signal. These devices are virtual sinks meaning that, in order to work their counterpart must be opened too. For example, in the playback path, the application will try to play audio through the speakers but what is really happening is that it is writing to a loopback device and AFE controls its counterpart. The counterpart will be opened as 'capture' device. Then AFE will rea the data, save a copy, and finally write to the speaker soundcard. This behavior is similar in the capture path. The application will try to read from microphones, but it is also from a loopback. AFE is writing its output to a playback device and the application is reading the data through the counterpart of the playback device.

The following diagram illustrate the data flow through the AFE:



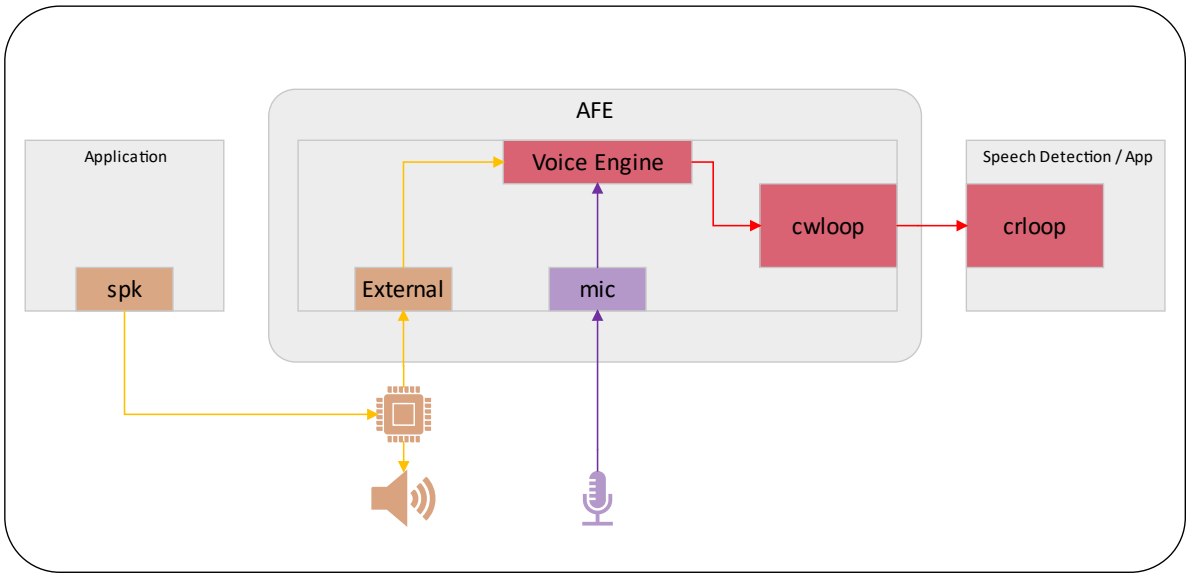
On the following table you can see the properties of the six devices.

Device Name	Owner	Access	Type	Pipeline
pwloop	Application	Write	Loopback	Playback
prloop	AFE	Read	Loopback	Playback
spk	AFE	Write	Hardware	Playback

mic	AFE	Read	Hardware	Capture
cwloop	AFE	Write	Loopback	Capture
crloop	Application	Read	Loopback	Capture

External References Signal

On the External References configuration AFE disables the 'spk' device since it is not responsible to playback the audio and the 'prloop' is not opened. Instead, it will open another device which must have the same data that is being played on the speakers. The capture pipeline will remain the same.



VoiceSeeker

VoiceSeeker is our own process NXP has to offer for cleaning up the incoming data. The full name of our solution is VoiceSeeker Light (VSL). As mentioned earlier VoiceSeeker can:

- Detect from which microphone the spoken word comes from and focus on that microphone (Beamforming).
- Remove the ambient background noise and give a clean version of the word being spelled (AEC).
- Adapt to any microphone array regardless of its physical location between them.
- Fix the acoustic delay.

VoiceSeeker is an example of an implementation of the SignalProcessorImplementation class, which has all its methods, and it is built as a share object allowing AFE to load it if required. AFE can load VSL with the following command: ***afe libvoiceseekerlight***. When it finished loading it VoiceSeeker will tell AFE how to set up the ALSA devices, format, channels, size of the buffers, etc.

Once all the devices are opened and configured, AFE will then warm up the devices and then start the streaming. AFE will control the streams, and when there is enough data to process, VSL process will be called. Inside VSL process the delay will be fixed before processing the data, and then it will clean up the audio. When VSL finish processing the data, it will then write the data to the output buffer which AFE will be in charge to write it to the loopback interface where VoiceSpot will be listening.

This is a brief explanation of what VSL does inside AFE. However, there are important parts of the process the user should be aware of and be able to configure. Starting from the beginning, on VSL initialization, there are a few things going on. First, the initialization of some variables which depends on the number of channels to be processed, if you want to use more than the defaults channels (four microphones and two references signals) you might want to change the ***heap_size*** and ***scratch_size***. Another variable that depends on the number of references channels is the ***sizeBufDelay*** which also depends on the amount of the acoustic delay there is on the system. So, if you have more reference signals, or you have microphones too far from the speakers you might want to change this variable.

Continuing with the initialization, VSL can be configure using the variable vsl_config which is a structure that contains following elements:

Element	Type	Description
num_mics	uint32_t	Number of microphones
num_spks	uint32_t	Number of speakers (references)
framesize_out	uint32_t	Output buffer size
buffer_length_sec	rdsp_float	Time of the expected wake word
aec_filter_length_ms	uint32_t	Windows size
create_aec	int32_t	Enagle AEC
create_doa	int32_t	Enables DOA (Directional of Arrival)
mic_xyz_mm	Pointer to rdsp_xyz_t structure.	Distance between microphones
device_id	RDSP_DeviceId_en	Platform Id

The first two variables are self-explanatory, they are just telling the amount of reference and microphones. The third one is a little bit tricky. Voice Seeker processes chunks of 32 frames per iteration and it will return a pointer to the filtered buffer when the buffer length has the number of samples requested. This number of samples are set with the ***framesize_out*** file, which in this default and recommended value is 200 samples due to VoiceSpot limitation. The fourth element is related to the wake word size. For example, if you are using a long word for wake word like “Hay NXP” you might want to increase this value up to 3 seconds but if it is shorter like “Alexa” the value of this element can be set to 1.5 seconds (which is the default value).

aec_filter_length_ms is the windows size with a higher value it would take longer on each iteration and most of the time is not worth it. Keeping it between 150ms and 300ms is a viable choice.
The next variable enables the AEC algorithm. The free version does not allow the enable of AEC. To get the full version please contact our team at **Voice@nxp.com**. This AppNote covers the free and non-free version of the library. The other thing you can enable is the DOA (Direction of Arrival) but this feature is disable by default on the pre-installed version and will not be covered on this document.

The ***mic_xyz_mm*** is a matrix with size of number of channel times 3 dimensions (width, depth, and height) the distances is between an origin and the microphones measured in millimeter. One thing to mention is that the origin can be anywhere, not necessarily in one of the microphones.

Finally, VSL needs to know where it is running since it has some optimization depending on which platform it is running on and that is what it is the last variable for.

The last thing worth mentioning about the constructor function is that the mechanism used to fix the acoustic delay is created and initialized. This mechanism needs calibration before it is used to achieve a reliable performance. The calibration method would be covered in section ADD_A_FANCY_SECTION.

Config.ini

VoiceSeeker reads a file to be properly configured. This file is called Config.ini, and it is located on ***/usr/unit-tests/nxp-afe.*** During the process of initialization VoiceSeeker, as well as VoiceSpot, reads and parses the file to configure itself according to the user's requirements.

The parameters of the file and what they do are described in the following table.

Field	Type	Description	Options
WWDectionDisable	Int	Tell the library if there is a Speech Engine for wake-word detection.	0 or 1
WakeWordEngine	String	The engine to use for wake-word.	VoiceSpot VIT
DebugEnable	Int	Enable the dump of the buffer into a file. Required for the library calibration	0 or 1
RefSignalDelay	Int	Value of the existing acoustic delay in samples.	[0, sizeBuffDelay]
mic0	Float	Measure between the origin at the 1st mic in milliseconds.	Float
mic1	Float	Measure between the origin at the 2nd mic in milliseconds.	Float
mic2	Float	Measure between the origin at the 3rd mic in milliseconds.	Float
mic3	Float	Measure between the origin at the 4th mic in milliseconds.	Float
VoiceSpotModel	File	File of the model to use for VoiceSpot. This is a VoiceSpot	String

		field. VoiseSpot will search for the file at the same path where Config.ini is.	
VoiceSpotParams	File	File of the model parameters. This is a VoiceSpot field. VoiceSpot will search for the file at the same path where Config.ini is.	String
VITLanguage	String	Language of the VIT model	Please visit VIT for the available languages.

Although VoiceSeeker uses this file to configure itself. If the file is not present VoiceSeeker still will work by using the default values. The default values are the same as listed on the file. Similarly, if any field is not present on the configuration file VoiceSeeker will setup with the default one, so please make sure there is no typo on the fields.

VoiceSpot

As well as VoiceSeeker, VoiceSpot is a library that is wrapped on a class (called ***SignalProcessor_VoiceSpot***) for easy use of the voice engine. VoiceSpot is an independent program that can communicate to VoiceSeeker for retrieving feedback of where and when a voice signal was detected. This is the reason VoiceSeeker and VoiceSpot go perfectly well together. VoiceSeeker notifies when there is a chunk of data available and VoiceSpot will tell VoiceSeeker if there is speech on that incoming data so VoiceSeeker can focus its attention on that channel and re-adjust its filter on the run.

Since VoiceSpot is an object there must be a main process that controls the workflow of the data and creates an instance of the library. This program is called ***voice_ui_app***, previously called ***voicespot***. The program is responsible for opening the capture ALSA device. When the data is ready it will either call VoiceSpot or VIT for the voice decoding. Finally, it will notify (if configured to do so) a third application so it can start an action based on the command that just arrived or on the wake-word. Such an application can be any Voice Assistances or similar applications.

Voice_ui_app works with VIT for the wake-word detection and command detection or just the command detection. On the first scenario VoiceSpot will be completely disabled and VIT will be the responsible on notifying VoiceSeeker when a wake word or a command is detected. When VoiceSpot is enabled, voice_ui_app will first call VoiceSpot to seek for the wake word. If VoiceSpot is able to detect the wake word, then it will pass the buffer to VIT to process the incoming voice command. To set up VIT for the wake word engine make sure you edit the configuration file. Set the *WakeWordEngine* variable to VIT instead of VoiceSpot.

The notification mechanism can be enabled by passing the ***-notify*** flag when running ***voice_ui_app***. If the flag is passed both engines will notify the third application. This means VoiceSpot will notify when there is a wake word on the buffer and then VIT will notify the ID of the voice command detected. When VoiceSpot is disabled, VIT will still notify when a wake word is detected.

Model

VoiceSpot uses a machine learning (ML) model to detect the wake-word, which had been trained with the best qualities and environment in order to achieve great precision even in a nosy environment. Therefore, if there is any interest in using VoiceSpot but with a different model please contact Voice@nxp.com to get a better idea of the costs and the time it will take to train the model for your custom wake word.

Once you have your model and the parameters of the model be sure that those files live in ***/usr/unit-tests/nxp-afe*** so VoiceSpot will be able to find them.

Installation guide

This section describes the installation.

Hardware checklist

This section describes the hardware checklist.

For i.MX 8M Plus

VoiceSeeker and VoiceSpot are currently supported by NXP i.MX ArmV8-A processors like the i.MX 8M Plus, i.MX 8M MINI and support the ArmV8.2 architecture like i.MX 93. The i.MX 8M Plus EVK packages contains (as well as the other packages) the following items for evaluating VoiceSeeker and VoiceSpot:

Quantity	Name	Description
1	i.MX 8M Plus EVK board	NXP evaluation board for the i.MX 8M Plus
1	Power supply	USB Type-C 45-W power delivery supply, 5 V/3 A; 9 V/3 A; 15 V/3 A; 20 V/2.25 A supported
1	Type-C male to type-A male cable	Assembly, USB 3.0 compliant
1	Type-A male to micro-B male cable	Assembly, USB 2.0 for UART debug

Additional hardware is also required:

Quantity	Name	Description
1	8MIC-RPI-MX8	NXP 8 microphone board. Available at https://www.nxp.com/part/8MIC-RPI-MX8#/.
1	Speakers	A speaker or two speakers compatible with a 3.5-mm audio jack connection.
1	3.5-mm cable	A 3.5mm audio connection cable which connects the EVK and the speakers.

Setup View

This section describes a common setup for testing the VoiceSeeker and VoiceSpot libraries. Please be aware that each setup can vary but the connection must be the same.

[A nice photo of my setup].

1. For the 8MIC-RPI-IMX8 board:
    a. Mount the board onto the EXP_CN of the EVK. Make sure that its pin numbering matches the one on the EVK.
    b. Unmute the board by pulling down its mute switch. The switch will indicate muting when the board is powered on if this is not set.
    c. Pull up the MIC_SEL so the ON position is active.
2. For the 8M Plus:
    a. Connect the audio jack table to the EVK and to the speakers.
    b. Connect the power supply.
    c. Connect the debug table to a PC.

Software installation

This section describes all the software, programs and files that require VoiceSeeker and VoiceSpot.

Software Checklist

The recommended BSP version is MM_04.08.02_2312_lf6.1.55 or above. Latest image can be installed from: https://www.nxp.com/design/design-center/software/embedded-software/i-mx-software/embedded-linux-for-i-mx-applications-processors:IMXLINUX.

Once the image is fully downloaded you can install it following the steps on the I.MX Linux User's Guide.

The image should contain the following files and drivers.

File	Location	Description
libvoiceseekerlight.so	/usr/lib/nxp-afe/	VoiceSeeker wrapper. Loaded by AFE at runtime
afe	/unit_tests/nxp-afe/	AFE binary
voice_ui_app	/unit_tests/nxp-afe/	Main app
asound.conf_imx8mp	/unit_tests/nxp-afe/	ALSA configuration file
Confing.ini	/unit_tests/nxp-afe/	VoiceSeeker, VoiceSpot configuration file
HeyNXP_1_params.bin	/unit_tests/nxp-afe/	NXP parameters for wakeword
HeyNXP_en-US_1.bin	/unit_tests/nxp-afe/	NXP wake-word model
snd-aloop	/lib/modules/<bs_version> /kernel/sound/drivers	Driver

Software setup

Selecting the device tree for the 8MIC-RPI-MX8 board

After flashing the Linux BSP to the desired storage medium of the EVK, boot the board and stop on the U-Boot terminal. This can be done by pressing any key after the U-Boot prompt during boot. When the U-Boot console is ready, perform the following commands:

```
u-boot=> setenv fdtfile
edit: imx8mp-evk-<revision>-8mic-revE.dtb
```

```
u-boot=> saveenv

u-boot=> boot
```

The given device trees for the revisions are:

Device Tree	Supported i.MX 8MP EVK revision
Imx8mp-evk-revb4-8mic-revE.dtb	B3, B4
Imx8mp-evk-revA3-8mic-revE.dtb	A3, B, B1
Imx8mp-evk.dtb	A2 or older

Install Missing Drivers

By default, the image does not load the drivers that allow ALSA to do a loopback stream. Therefore, it is required to manually installed it with the following command:

```
$ modprobe snd-aloop
```

Editing the ALSA asound configuration for VoiceSeeker

The following commands overwrite the ALSA virtual devices to properly work with VoiceSeeker while saving a backup of the original configuration.

```
$ cd /unit_tests/nxp-afe/

$ mv /etc/asound.conf /etc/asound.conf.org

$ cp asound.conf_imx8mp /etc/asound.conf
```

Or copy the asound.conf_imx8mp_revb4 if the EVK revision is B3 or B4.

Audio lab guides

This section describes how to put the library in shape as well as ways of testing the library.

Delay Measurement

As mentioned in the Introduction, the AFE performs AEC (by loading VoiceSeeker library) by subtracting the playback signal from the capture signal (which is a mix of the playback and voice signals). But the playback signal that comes mixed in the capture stream has some delay with respect to the original playback because of the time that takes the sound to travel from the speakers to the microphones and be recorded, so it's important to have a precise measurement of this delay to achieve a good performance.

Before measuring the delay it is necessary to mention a few things:

1. This is a one-time calibration as long as the setup doesn't change. If the distance between microphones and speakers does change it will require recalibration.
2. Delay can vary between measurements, but the range between measurements should be around one or two samples.
3. There are other ways to measure the delay. For example, another good approach to measure the delay is to use a cross-correlation between L/R (Left and Right) speakers, and the microphones

channels. For it you can use MATLAB, Python or any other language that has a library which does that.

4. The method used on this document is a more visual method with an audio-tunning application, which counts the samples between two points in the analyzed audio stream.

To measure the delay, it's necessary to make a sample recording using the AFE, in this sample recording the AFE will generate three audio files: the playback + capture stream, the playback stream alone and the capture stream alone (the result of the AEC, which for the calibration purpose is not needed). These files will help us to measure the delay.

Before running the programs as the TODO.md file suggests, the VoiceSeeker configuration must be changed. First, the debug feature should be enabled and set up the delay property to 0. The way of doing so is by changing the Config.ini file as follow:

```
$ vi Config.ini
```

Set the following values for DebugEnable and RefSignalDelay properties. The DebugEnable variable can be turned off once the calibration is done, but for the sake of this document it will be kept on till the end.

```
DebugEnable = 1

RefSignalDelay = 0
```

Safe the file and quit Vi.

Launch voice_ui_app as well as afe with VoiceSeeker library, as it is described on the TODO.md file. It is important to have both afe and voice_ui_app running in the background. If one of them is not it might cause unexpected behaviors.

```
$ ./voice_ui_app &
```

A way to verify that voice_ui_app initialized property is by looking to the log. If voice_ui_app prints the VIT available commands it means that voice_ui_app could initialize VoiceSpot and VIT properly.

```
$ ./afe libvoiceseekerlight &
```

To produce a known signal, let use GST for convenience. Creating a minimal pipeline with gst-launch-1.0 will do the work. The command will produce a periodic Tick which will make it easier for us to measure the delay. Play the audio for a few Ticks.

```
$ gst-launch-1.0 audiotestsrc wave=8 ! alsasink
```

The Ticks should be hearable through out the speakers.

For further information about any GST plugin you can always run the gst-inspect-1.0 command.

```
$ gst-inspect-1.0 audiotestsrc #As an example
```

After played the audio for a few Ticks, stop the pipeline with Ctrl + C and kill *afe* and *voice_ui_app*.

```
$ pkill afe

$ pkill voice_ui_app
```

The three mentioned audios should be generated and located in */tmp/* directory

```
$ ls /tmp/ -l

-rw-r--r-- 1 root root 4812844 Mar  6 11:34 mic_in_delay_S0_E1.wav
-rw-r--r-- 1 root root 1203244 Mar  6 11:34 mic_out_S0_E1.wav
-rw-r--r-- 1 root root 2406444 Mar  6 11:34 ref_in_delay_S0_E1.wav
```

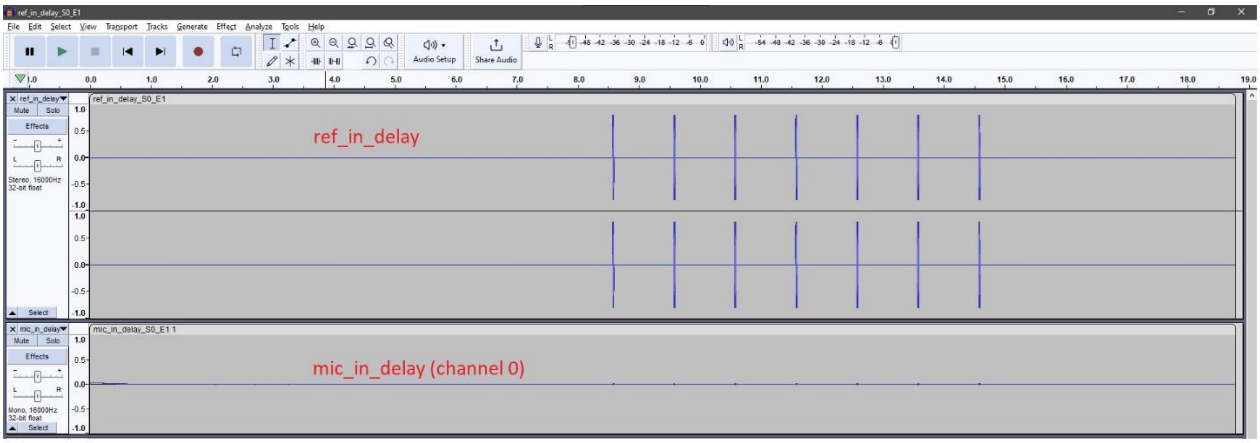
- The **mic_in_delay_S0_E1.wav** file contains the mixed signal (playback + voice) played from the speaker and recorded from the 8MIC board (delayed signal).
- The **mic_out_S0_E1.wav** file contains the clean voice (ideally null in this case since no voice is recorded) obtained from AEC algorithm.
- The **ref_in_delay_S0_E1.wav** file contains the original playback stream (tick tone) without being recorded (signal with no delay).

Where:

- _S0_ (Start minute): stand for the minutes it started to record.
- _E1_ (End minute): stand for the minute it stops recording.

By default, the library record pair minutes (0-1, 2-3, 4,5, etc…). In order to change the default behavior it is necessary to change a configuration macro called *MINUTE_INTERVAL_WAV_FILE* (located on *<root_dir>/voiceseeker/src/SignalProcessor_VoiceSeekerLight.h*) from 3 to 1.

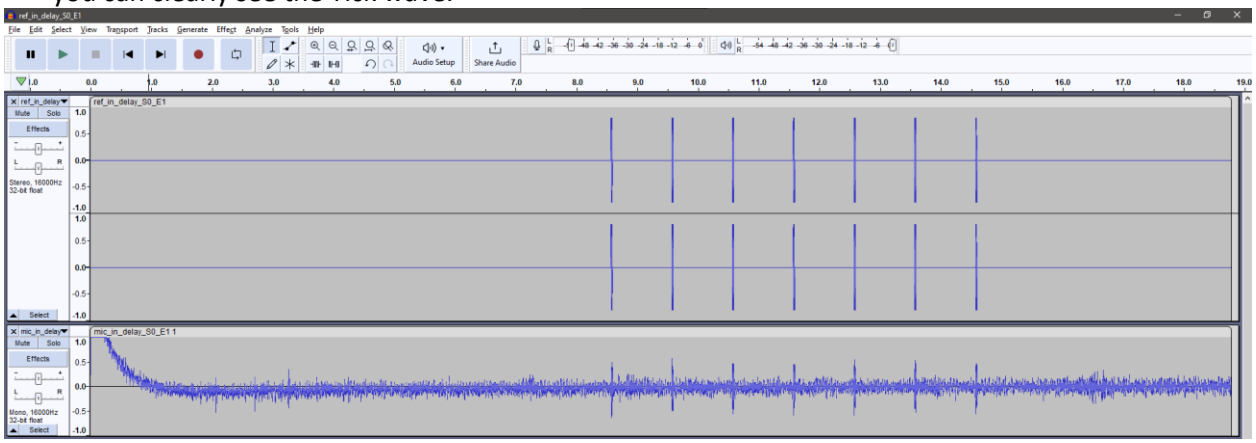
To measure the delay only the mic_in_delay_S0_E1.wav and ref_in_delay_S0_E1.wav files are needed. Copy those files to the host machine (with *scp* or any other method), open one of them with Audacity and then, the other wave file drag and drop it in to the same windows of Audacity.



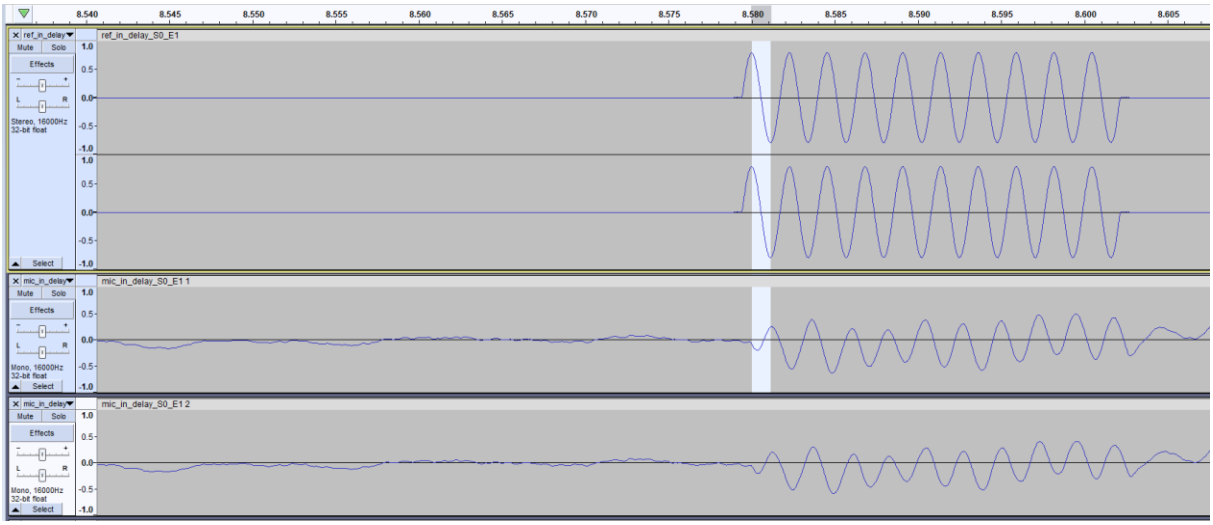
It is recommend to add some gain to the microphone data. To achieve a better view of the data follow the next step to add some gain to the microphone input.

1. Select the four microphone channels.
2. Go to -> Effect -> Volume and Compression -> Amplify.

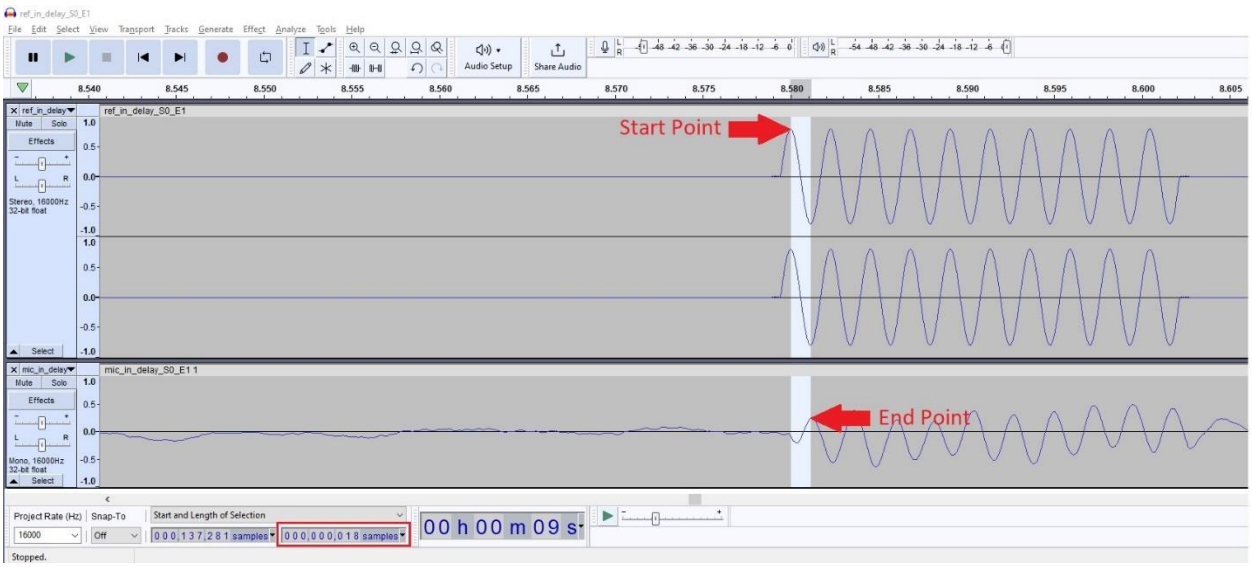
3. A pop-up window will appear. Check the “Allow clipping” box and set the amplification to a value you can clearly see the Tick wave.



To achieve a good accuracy, zoom in to the beginning of the first Tick on the references signal and select all the samples between the reference point and its representation on the microphone channel.



The delay can be measured in time or in sample units. VoiceSeeker works with the delay in samples. Typically, audio editors allow time-frame measurements in sample units. To do so on Audacity, go to the bottom left corner, change on the option bar to “Start and Length of Selection”. Then you will see how much sample the signal is delayed.



Now, the only thing left to do is to set the *RefSignalDelay* with this new value.

Beamforming

This section illustrates the performance of the beamforming capabilities of VoiceSeeker in conjunction with VoiceSpot.

Beamforming is the ability to highlight a section of a signal when it detects certain properties. VoiceSeeker use beamforming when it detects a wake-word.

The clearest way of seeing this feature working is by playing a pure tone and then trigger the wake-word and then saying a voice command from VIT.

Before running the programs again lets clean the temporary files, just to be sure we are working with the latest wave files.

```
$ rm -v /tmp/*.wav
```

Start VoiceSpot and VIT with *voice_ui_app* and VoiceSeeker with *afe.*

```
$ ./voice_ui_app &
$ ./afe libvoiceseekerlight &
```

Play a pure tone with GST.

```
$ gst-launch-1.0 audiotestsrc wave=0 ! alsasink
```

Said the wake-word and a voice command. For example: “Hey NXP! Mute”. Wait a few seconds and then repeat, the wake-word and the voice command could be changed if desired.

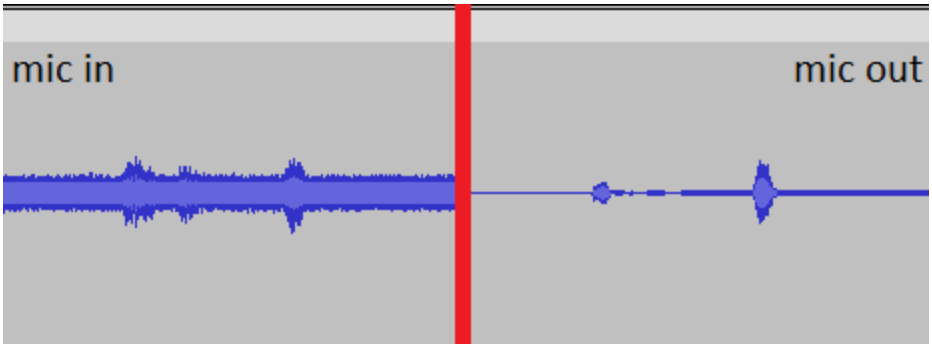
Press Ctrl + C to stop GST and then stop the processes just as before.

```
$ pkill voice_ui_app
$ pkill afe
```

This time the required wave files are the mic_in_delay_S0_E1.wav and mic_out_S0_E1.wav where in the first file the voice sound as it were in the background and the second one as if voice has moved to the front.

Copy those files to the host machine and first open the mic_out_S0_E1.wav file and then drag-and-drop the other one.

Select all the channels and amply some gain to them. Before playing the audios you will see how beamforming is acting on the microphone output signal isolating the voice when it detect it.

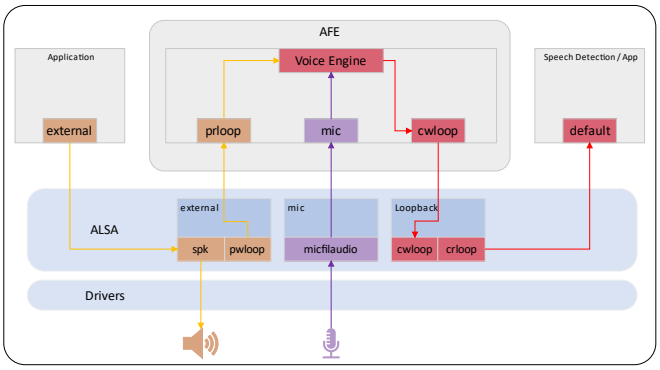


Another way to compare the audio is by playing each one on solo mode by pressing the 'solo' button on the left hand side on each channel. When playing any channel from the microphone the voice will be heard as it where at the background, which it truly is, since the speakers are closer to the microphone. However, when playing the microphone output data the voice seems to be closer to the microphones than the speakers.

External References

This section describes how the external signal feature could be tested with the current setup.

The idea is creating a virtual device which will be in charge to write to the speakers and feeding AFE with the references signal, as show on the diagram below:



Asound.conf

To test the feature it is required to add some devices to the asound.conf file. The idea is to duplicate the playback stream and let ALSA, GST or any other application to manage the writing to the speaker and sending it back to VoiceSeeker.

Open *asound.conf* with you editor of choice and add the following lines to it.

```
pcm.external {
        type plug
        slave.pcm "loopNspk"
}
pcm.loopNspk {
        type multi
        slaves.a.pcm "spk"
        slaves.a.channels 2
        slaves.b.pcm "pwloop"
        slaves.b.channels 2
```

```
        bindings.0 { slave a; channel 0; }
        bindings.1 { slave a; channel 1; }
        bindings.2 { slave b; channel 0; }
        bindings.3 { slave b; channel 1; }
```

```
}
```

Save the file and quit.

What above lines do is creating two devices. The first one (*external*), is the first element of the chain and the one the application needs to talk too. It's responsible for duplicating the channels into 4 channels. The other one (*loopNspk*) is where the main part is. This device is responsible for writing to two different soundcards, one for the speaker and the other opens the device which writes to the input device for *afe*.

The difference between using these devices and the default one is that now the application will manage the playback thread. This means that running the following command the speakers will start playing some data without the need of *afe* being running.

```
$ gst-launch-1.0 audiotestsrc wave=0 ! alsasink device=external
```

Running the Pipeline

Voice_UI

voice_ui_app runs in the same way as earlier, it does not require any change.

```
$ ./voice_ui_app &
```

AFE

Running *afe* in external configuration is very easy as described early. The must important on this configuration is knowing which device should be opened and *afe* will do the rest. In this case the device that needs to be opened in order to get the reference signal is the default one, *prloop.* However, running the program in the same way as before it might give an error, since the speaker device is already taken by another application. So, *afe* needs to disable as well this device to avoid any error at run time. All of this is done with the following command.

```
$ ./afe libvoiceseekerlight prloop &
```

Now that afe has opened the device, which contains the reference data, the playback application can be executed.

```
$ gst-launch-1.0 audiotestsrc wave=0 ! alsasink device=external
```

Stop the application after a few seconds. The ref_in_delay_S0_E1.wav can be copied to the host machine and then opened with Audacity to verify that the file is not empty or doing an *hexdump* should be enough.

AEC Enablement

This section describes how to build VoiceSeeker with the AEC enabled (a similar process is required to build VoiceSpot without timeout).

VoiceSeeker AEC gives between 25 and 30dB attenuation of the references signal when it is property configured and there is no any type of vibration on the system or extra post processing on the signal that afe is not aware of.

Getting the library

In order to enable the AEC (acoustic echo cancellation) user needs to upgrade VoiceSeeker library by communicating with NXP Voice Team to the following email: voice@nxp.com.

The Voice Team will delivere a folder for the requested platform, similar to the ones which are located on the voiceseeker/*platforms* folder on the imx-voiceui repo. After getting the folder, it needs to be installed on the source code and compile again the binaries.

Installing the library

Clone the repo from GitHub.

```
$ git clone https://github.com/nxp-imx/imx-voiceui.git
```

Move to the latest branch.

```
$ cd imx-voiceui
$ git switch MM_04.08.02_2310_L6.1.y
```

The next step is to replace the folder from *voiceseeker/platforms/iMX8M_CortexA53* with the new one which contains the full version of VoiceSeeker.

```
$ cp ../iMX8M_CortexA53 ./voiceseeker/platforms/iMX8M_CortexA53
```

Compile the library

The instructions to compile the library with AEC enabled are described on the readme. However, setting up the AEC variable from the terminal will be easier and will required less keypress when building the image.

```
$ export AEC=1
$ make
```

With previous step you should see the following message just before the compilation starts.

```
Building with AEC
```

Building the library will generate a *release* folder which contains *voice_ui_app* binary and *libvoiceseekerlight.so.2.0*, a long side with other files which really don't need to be deployed to the board since those have not changed.

Board deployment

What is really necessary is the shared library but it is a good practice to overwrite both binaries.

```
$ scp release/libvoiceseekerlight.so.2.0 root@<BOARD_IP>:/usr/lib/nxp-afe/
$ scp release/voice_ui_app root@<BOARD_IP>:/unit_tests/nxp-afe/
```

Once the binaries have been installed on the target board, run them to verify they are working properly.

When VoiceSeeker is initialized it dump a configuration status telling how it is configured, on that part there is a property called *create_aec* which has a value of *1* if AEC is enabled and a value of 0 when it is not.

Low Power Voice demo

Since this topic is a little bit too long there is already a document which tell how to enable it.

Appendix

Using NXP SWPDM with AFE

This chapter describes how to integrate the NXP SWPDM ALSA Plugin with Voice's pipeline.

NXP SWPDM

NXP SWPDM is a solution for the 8M Mini hardware which does not have a 32 bit width resolution required for Voice Processing tasks. By adding this ALSA plugin to the pipeline increases the accuracy of the wake-words and voice-commands without adding too much load to the CPU.

SWPDM Plugin uses a CIC Filter algorithm to convert the PDM data from the microphones to a PCM format required for any audio post-processing activities.

Adding SWPDM to VoiceSeeker pipeline

This section describes the steps needed to integrate SWPDM to VoiceSeeker pipeline. Perhaps it is not required for the 8M Plus both the steps are more likely the same for the i.MX8MM or i.MX8MN.

DTB

In order to use the SWPDM algorithm it is required to disable the MICFIL peripheral, letting Linux to receives raw data from the microphones. To do so, it is required to reboot the board, stop at U-Boot and change the dtb.

```
u-boot=> edit fdtfile
edit: imx8mp-evk-8mic-swpdm.dtb
```

If desired save the environmental variables and boot the board.

Use of SWPDM

There is a document that explains how to add the SWPDM plugin to the *asound.conf*. Basically it tells the structure which needs to be added to the file and some of the restrictions the plugin has. The required structure is as follow:

```
pcm.cic {
        type cicFilter
        slave "hw:imxswpdmaudio,0"
        delay 100000
        gain 0
        OSR 48
}
```

Capture some data to verify everything is working as it should:

```
$ arecord -D cic -c 4 -f s32_le -r 16000 -d5 swpdm.wav
```

The audio might be with a very low volume which can be fixed playing with the *gain* property of the *cicFilter* plugin type. A good value for this exercise could be 15 but please keep it mind that it needs a proper calibration.

Asound

Integrating the SWPDM ALSA Plugin is very easy, the only thing needed for it is to change the name of the device to the one that AFE opens, which is well known as *mic*. So, open the *asound.conf* file one last time and replace the name from *pcm.cic* to *pcm.mic* and then rename the original *pcm.mic* device so ALSA doesn't give you an error.

The last thing to do before closing the file is removing the delay of the plugin by setting up a value of 0. If not, *afe* will start delaying the pipeline due to this delay. So, the structure of the device will look as follow:

```
pcm.cic {
        type cicFilter
        slave "hw:imxswpdmaudio,0"
        delay 0
        gain 15
        OSR 48
}
```

Finally, run *voice_ui_app* and *afe* as normal.

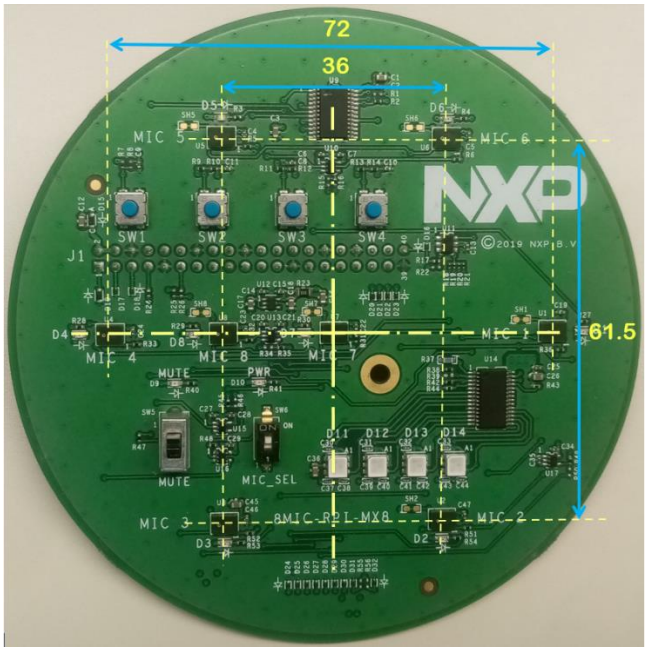
Microphone Arrangement

This chapter describes how to change the geometry of the microphones using the 8MiC-RPI-MX8 board.

Microphone Routing

The 8MiC-RPI-MX8 board has 8 microphones, numbered from 0 to 7 and placed as shown in figure 1.

The microphone relative positions are also needed to enhance the AEC, these positions are shown in figure 2. In this case the distance are take form the center of the microphones and are measured in millimeters.



Every microphone can be routed to any available channel of the recording audio stream by modifying */etc/asound.conf* file using *ttable:*

```
pcm.mic {
      type route
      slave.pcm “hw:micfilauio,0”
      slave.channels 8
      ttable.0.0 1
      ttable.1.1 1
      ttable.2.2 1
      ttable.3.3 1
      ttable.4.4 1
      ttable.5.5 1
      ttable.6.6 1
      ttable.7.7 1
}
```

ttable property can re-route a physical microphone to any audio stream channel as follows:

I lines like: ***ttable.A.B 1***

Where:

- **A** *represents the channel index used on the audio stream.*
- **B** represents the physical mic index on the board.
- **1** represents the number of mics.

So, the line ***ttable.3.6 1*** would re-route the audio signal coming from **mic 6** to **channel 3.**

Microphone Position Configuration

The AFE uses the Config.ini file to setup the microphone relative coordinates.

The Config.ini file must also be edited, the code lines below show a typical mic coordinate setup.

```
mic0 = 0.00, 0.00, 0.00
mic1 = 36.00, 0.00, 0.00
mic2 = -18.00, 30.75, 0.00
mic3 = -18.00, -30.75, 0.00
mic4 = 18.00, -30.75, 0.00
mic5 = -36.00, 0.00, 0.00
mic6 = 18.00, 30.75, 0.00
mic7 = -18.00, 0.00, 0.00
```

The number **x** in **micx** variable refers to the virtual mic channel in the audio stream and not to the physical mic index. So, if you want to set up the coordinates of physical microphone 3, you must first route **microphone 3** to some channel (let's use **channel 5** as an example) on *asound.conf*:

```
      ttable.5.3 1
```

And then set the coordinates to the same channel (**channel 5**) in Config.ini

```
mic5 = -35.0, 18.0, 0.0
```

The coordinates are written in the form:

```
micX = x-coordinate, y-coordinate, z-coordinate
```

and are all in millimeters.

The coordinates are relative, that means there is not an absolute reference for them.

For simplicity a microphone can be chosen as the origin by setting its coordinates to 0, 0, 0 and then writing the coordinates of the other microphones relative to the origin mic.

Microphone Arrangement Example

This example shows the configuration needed to use a custom set of microphones (the ones circled in purple). The intentions of this example is to route microphones 6, 5, 3, 1 to channels, 0, 1, 2, 3 respectively.

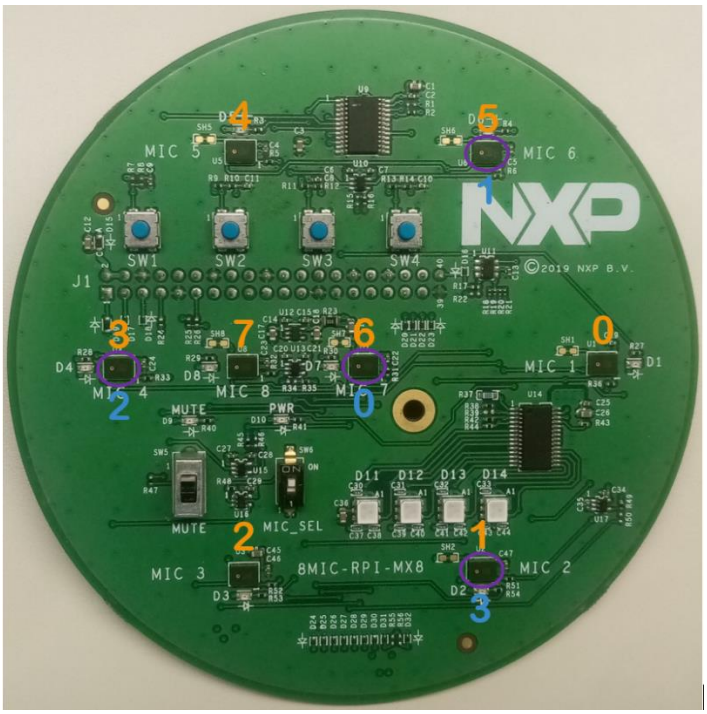


Figure 3 shows the mic numbers in yellow and the desired channels in blue.

To route the microphones to the desired channels we would use the next *ttable* configuration in *asound.conf* file:

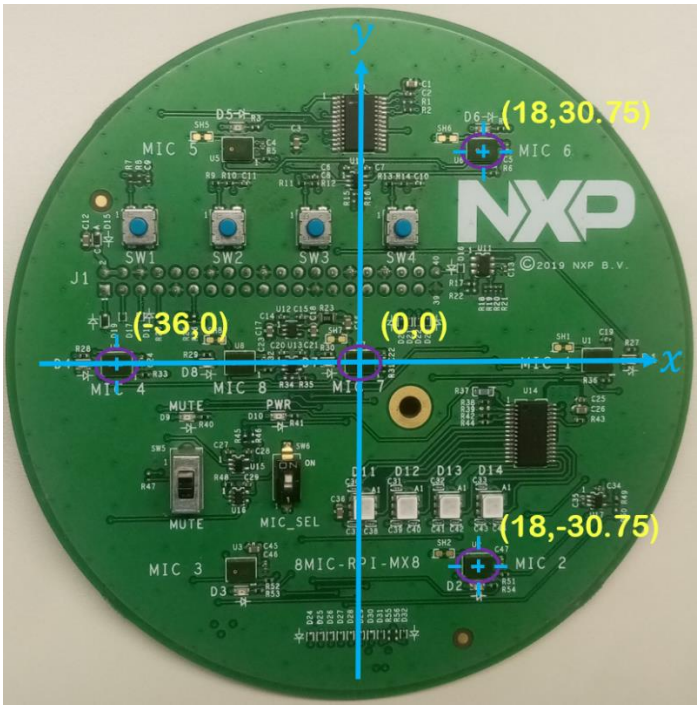
```
pcm.mic {
      type route
      slave.pcm ”hw:micfilaudio,0”
      slave.channels 8
      ttable.0.6 1
      ttable.1.5 1
      ttable.2.3 1
      ttable.3.1 1

}
```

And to set the microphone corresponding coordinates we would use the next configuration in Config.ini file:

```
mic0 = 0.0, 0.0, 0.0
mic1 = 18.0, 30.0, 0.0
mic2 = -35.0, 0.0, 0.0
mic3 = 18.0, -30.0, 0.0
```

In this case the physical mic 6 has been selected as channel 0 as the origin, and the other three channels positions are measured using this microphone as the origin. Figure 4 shows the four chosen microphones and their relative coordinates.



Working other PDM microphones

Benchmarks

Voice Seeker processing latency

Debug Enable latency

VoiceSpot data transmit