NXP Semiconductors
User's Guide

Document Number: DSPUG
Rev. 4, 02/2021

DSP User's Guide

1. Introduction

This document provides an overall introduction to the DSP including system architecture, file organization, DSP-related toolchain, and so on. This document helps with the overall understanding of the DSP-related code. Currently, the DSP is used to decode and encode audio streams on the i.MX8 QXP, QM, and MP platforms. The current DSP framework can support up to 15 clients. They support these codecs:

Decoder:

- AAC-LC
- AAC plus(HE-AAC/HE-AACv2)
- BSAC
- DAB+
- MP2
- MP3
- DRM
- SBC
- OGG
- AMR-NB
- AMR-WB
- WMA
- WAV

Encoder:

- SBC

Contents

2. System architecture

Figure 1 provides the overall system architecture of the DSP-related code.

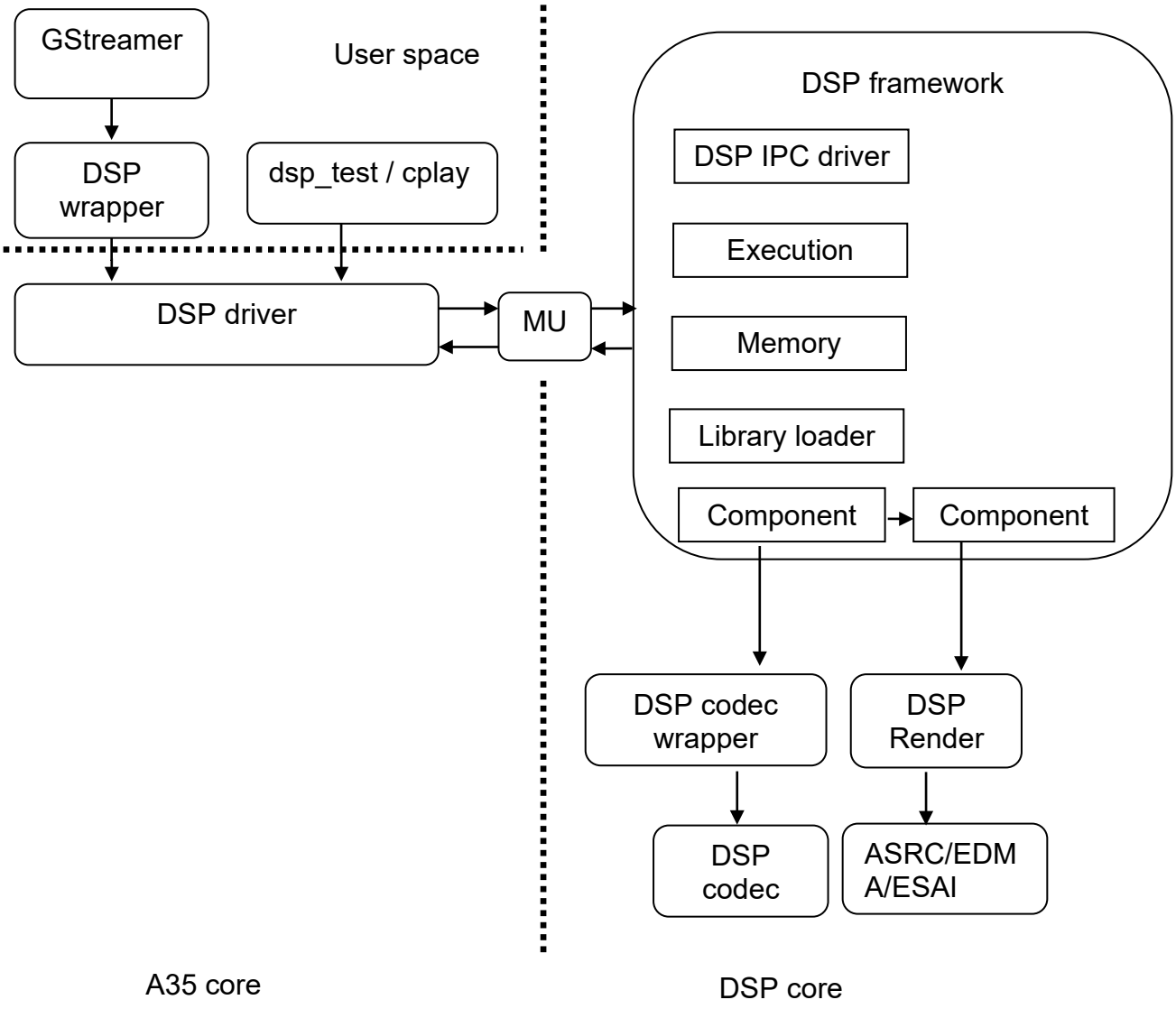

Figure 1. System architecture

The DSP-related code includes the DSP framework, DSP driver, DSP wrapper, unit test, DSP codec wrapper, and DSP codec. The DSP framework is a firmware code which runs on the DSP core. The DSP driver is used to load the DSP firmware into the memory and transfer messages between the user space and the DSP framework. A shared ring buffer is used to transfer messages between the A35 and the DSP core. The MU (Message Unit) is used to trigger interrupts between A35 and DSP core when messages are placed into the ring buffer. The DSP wrapper and the unit test are the application code in the user space, which uses the *ioctl()* interface to transfer messages between the DSP driver and the user space. In addition, the DSP wrapper is used to provide unified interfaces for the GStreamer. The DSP codec provides the actual decoding and encoding functions. The DSP codec wrapper is a wrapping code for the

DSP codec and provides unified interfaces for the DSP framework. The DSP driver also offers a compressed sound card interface to the user space applications. The DSP framework supports direct rendering of the decoded PCM data to an external audio device.

3. File organization

The DSP framework, DSP wrapper, and unit test code are in the *imx-audio-framework* package. The DSP driver code belongs to the Linux OS kernel. The DSP codec wrapper and DSP codec are license-restricted code; a license authorization is required to get them.

3.1. DSP driver

The DSP driver code is in the Linux OS kernel. It includes the following files:

- *linux-kernel/sound/soc/fsl/fsl_dsp.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp.h*
- *linux-kernel/sound/soc/fsl/fsl_dsp_proxy.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp_proxy.h*
- *linux-kernel/include/uapi/linux/mxc_dsp.h*
- *linux-kernel/sound/soc/fsl/fsl_dsp.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp_cpu.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp_cpu.h*
- *linux-kernel/sound/soc/fsl/fsl_dsp_pool.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp_pool.h*
- *linux-kernel/sound/soc/fsl/fsl_dsp_library_load.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp_library_load.h*
- *linux-kernel/sound/soc/fsl/fsl_dsp_xaf_api.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp_xaf_api.h*
- *linux-kernel/sound/soc/fsl/fsl_dsp_platform_compress.c*
- *linux-kernel/sound/soc/fsl/fsl_dsp_platform.h*
- *linux-kernel/sound/soc/fsl/imx-dsp.c*

3.2. DSP framework

The DSP framework code is in this folder:

- *imx-audio-framework/dsp_framework*

3.3. DSP wrapper and unit test

The DSP wrapper and unit test are in these folders:

- *imx-audio-framework/dsp_wrapper*
- *imx-audio-framework/unit_test*

3.4. Interface header files

The DSP-related code includes these four interface header files:

- *imx-audio-framework/include/mxc_dsp.h*
- *imx-audio-framework/dsp_framework/plugins/audio_codec/dsp_codec_interface.h*
- *imx-audio-framework/dsp_wrapper/include/uni_audio/fsl_unia.h*
- *imx-audio-framework/dsp_wrapper/include/uni_audio/fsl_types.h*

The *mxc_dsp.h* file is the same as the header file in the Linux OS kernel. This file includes the interfaces and command definitions that are used by the DSP wrapper and unit test. The *dsp_codec_interface.h* file wraps the DSP codec's header files. It includes unified interfaces and command definitions which can be used by the DSP framework. The *fsl_unia.h* and *fsl_types.h* header files include the interfaces and command definitions which can be used by GStreamer.

4. Building DSP framework on Linux OS

Before you compile the DSP-related code, set up the DSP-related toolchains. The DSP framework, DSP codec wrapper, and DSP codec use Xtensa® development toolchain.

4.1. Installing Xtensa development toolchain

The Xtensa development toolchain consists of two components which are installed separately in the Linux OS, including:

- Configuration-independent Xtensa Tool
- Configuration-specific core files and Xtensa Tool

The configuration-independent Xtensa Tool is released by Cadence®. For the current code, the version of the tool is *XtensaTools_RI_2020_4_linux.tgz*, which is updated from *XtensaTools_RF_2016_4_linux.tgz*. The two versions are compatible. The *XtensaTools_RF_2016_4_linux.tgz* version is used as an example. You may use two ways to get this package. The first is to download it from the Xtensa Xplorer™ and the second is to get it from other people who installed the Xtensa Xplorer.

The configuration-specific core files and the Xtensa Tool are released by NXP. There are two specific Xtensa tools. The *hifi4_nxp_v4_3_1_3_dev_linux.tgz* tool is updated from the *hifi4_nxp_v3_3_1_2_dev_linux.tgz* tool. The *hifi4_nxp_v3_3_1_2_dev_linux.tgz* tool is used as an example for the i.MX8QXP and i.MX8QM boards. The second tool is *hifi4_mscale_v1_0_2_prod_linux.tgz* that is updated from the *hifi4_mscale_v0_0_2_prod_linux.tgz* tool. The *hifi4_mscale_v0_0_2_prod_linux.tgz* tool is used as an example for the i.MX8MP board.

When you have these two components, you can set up the toolchain as follows:

- Open the *imx-audio-framework* folder and execute these commands:

```
mkdir -p ./imx-audio-toolchain/Xtensa_Tool/tools
mkdir -p ./imx-audio-toolchain/Xtensa_Tool/builds
```

- Set up the configuration-independent Xtensa Tool:

```
cd imx-audio-toolchain/Xtensa_Tool
tar zxvf XtensaTools_RF_2016_4_linux.tgz -C ./tools
```

- Set up the configuration-specific core files and the Xtensa Tool:

```
cd imx-audio-toolchain/Xtensa_Tool
tar zxvf hifi4_nxp_v3_3_1_2_dev_linux.tgz -C ./builds
```

- Copy the configurable memory map files which are supported by NXP to the following path:

```
cd imx-audio-toolchain/Xtensa_Tool
```

  - For the i.MX8QXP/QM board:

```
cp -r memmap/mainsim ./builds/ RF-2016.4-linux/hifi4_nxp_v3_3_1_2_dev/xtensa-
elf/lib
```

  - For the i.MX8MP board:

```
cp -r memmap/mainsim/imx8m  ./builds/RF-2016.4-linux/hifi4_mscale
_v3_3_1_2_dev/xtensa-elf/lib
```

- Install the Xtensa development toolchain:

```
cd imx-audio-toolchain/Xtensa_Tool
./ builds/RF-2016.4-linux/hifi4_nxp_v3_3_1_2_dev/install --xtensa-tools ./tools/RF-
2016.4-linux/XtensaTools --registry ./tools/RF-2016.4-linux/XtensaTools/config
```

- Set the PATH environment variable:

```
export PATH= ./imx-audio-toolchain/Xtensa_Tool/tools/RF-2016.4-
linux/XtensaTools/bin:$PATH
```

- Set the LM_LICENSE_FILE environment variable.

The Xtensa development tools use FLEXlm for license management. The FLEXIm licensing is required for tools such as the Xtensa Xplorer, TIE Compiler, and Xtensa C and C++ compiler. If you want to use a floating license, install the FLEXIm license manager and set the LM_LICENSE_FILE environment variable. In case of any problems, you can find useful information in the *Xtensa Development Tools Installation Guide User's Guide.doc* document provided by Cadence.

After the above steps, the Xtensa development toolchain is set up successfully. In addition, the Xtensa Tools and additional tools are provided as 32-bit (x86) binaries. They are supported on 32-bit (x86) systems, and also on recent 64-bit (x86-64) systems that have appropriate 32-bit compatibility packages installed. If you use a 64-bit system (for example; Ubuntu 16.04), install the 32-bit compatibility packages first. Use these commands:

```
sudo apt-get install lib32ncurses5 lib32z1
sudo dpkg --add-architecture i386
sudo apt-get install libc6:i386 libstdc++6:i386
```

4.2. Building DSP framework

After installing the DSP-related toolchains on your Linux OS server, you can compile the DSP framework. Execute the "make" command in the *imx-audio-framework* folder to compile the DSP framework. This way also builds the DSP wrapper and unit test. If you want to compile the DSP framework separately, see the README file in the *imx-audio-framework* folder. After the compiling process, you can find the binary files in the *imx-audio-framework/release* folder.

For the DSP framework, different commands generate different frameworks for different platforms:

- *imx-audio-framework/release/hifi4_imx8qmqxp.bin*
- *imx-audio-framework/release/hifi4_imx8mp.bin*
- *imx-audio-framework/release/hifi4_imx8mp_lpa.bin*

By default, the command generates the *hifi4_imx8qmqxp.bin* file. With the “PLATF=imx8m” attribute, it generates the *hifi4_imx8mp.bin* file. With the “PLATF=imx8mp_lpa” attribute, it generates the *hifi4_imx8mp_lpa.bin* file, which is supported by LPA. With the “DEBUG=1” attribute, it generates the firmware with the debug info. You can see the debug info using UART. For the iMX.QXP/QM board, modify the “LPUART_BASE” from 0x5a090000 to 0x5a080000, disable “lpuart” in “lpuart_probe”, which is in the *fsl_lpuart.c* file of the kernel, and the debug info can print in the UART console.

4.3. DSP DEBUG

Building the DSP framework with the extra “DEBUG=1” attribute, the DSP can print the debug info using the UART console. To enable this feature, do some changes in the kernel and in the DSP side. For a different platform prepare a different board and different changes. The following sections describe what to change for different platforms.

4.3.1. Enabling DSP debug on iMX.MP

Enable the UART for DSP print debug info in the iMX.MP board and add the UART clock in the DTS file and the UART module driver in the DSP.

1. Add the UART clock and pinctrl in the DTS.

   Add the UART clock and pinctrl in the DSP node as follows:

```
    &dsp {
          compatible = "fsl,imx8mp-dsp-v1";
          memory-region = <&dsp_reserved>;
          reg = <0x0 0x3B6E8000 0x0 0x88000>;
          pinctrl-0 = <&pinctrl_uart4>;
          clocks = <&audiomix_clk IMX8MP_CLK_AUDIOMIX_OCRAMA_IPG>,
                   …
…
            <&audiomix_clk IMX8MP_CLK_AUDIOMIX_ASRC_IPG>,
            <&clk IMX8MP_CLK_UART4_ROOT>,
            <&clk IMX8MP_CLK_UART4_ROOT>;
   clock-names =

"ocram", "audio_root", "audio_axi", "core", "debug", "mu2", "sdma_root", "sai_ipg", "sa
i_mclk","asrc_ipg", "uart_ipg", "uart_per";
     …
          fsl,dsp-firmware = "imx/dsp/hifi4.bin";
          status = "okay";
};
```

   Then generate the DTB file, replacing the old one. The UART is already added in the *imx8mp-evk-dsp.dts* file. You can boot the system with this DTS file directly and you do not have to change the DTS file.

2. Add the UART driver in the DSP.

   By default, the DSP side already supports enabling the UART. The code is located in the *dsp_framework/arch/peripheral.c* file. Build the DSP firmware with the “DEBUG=1” attribute to generate the *hifi4_imx8mp.bin* file, rename it to *hifi4.bin*, and copy it to the board.

3. Run the DSP and print the debug info.

Run one instance and the following debug info is printed on the fourth serial COM port:

```
DSP Start.....
core initialized
Response queue: write = 0x0 / read = 0x0
Command queue: write = 0x10001 / read = 0x0
ext_msg: [client: 0]:(80008004,4,1000)
Response queue: write = 0x0 / read = 0x0
Command queue: write = 0x10001 / read = 0x10001
alloc size out: 943feff8 4104 avail mem: 16773104
Response queue: write = 0x0 / read = 0x0
Response[client: 0]:(80048000,4,1000)
Command queue: write = 0x10001 / read = 0x10001
Response queue: write = 0x10001 / read = 0x10001
Command queue: write = 0x20002 / read = 0x10001
ext_msg: [client: 0]:(80008004,80000001,15)
Response queue: write = 0x10001 / read = 0x10001
```

4.3.2. Enabling DSP DEBUG on iMX.QXP/QM

To enable the DSP DEBUG on the iMX.QXP\QM platform, you need only one base board, as shown in Figure 2.

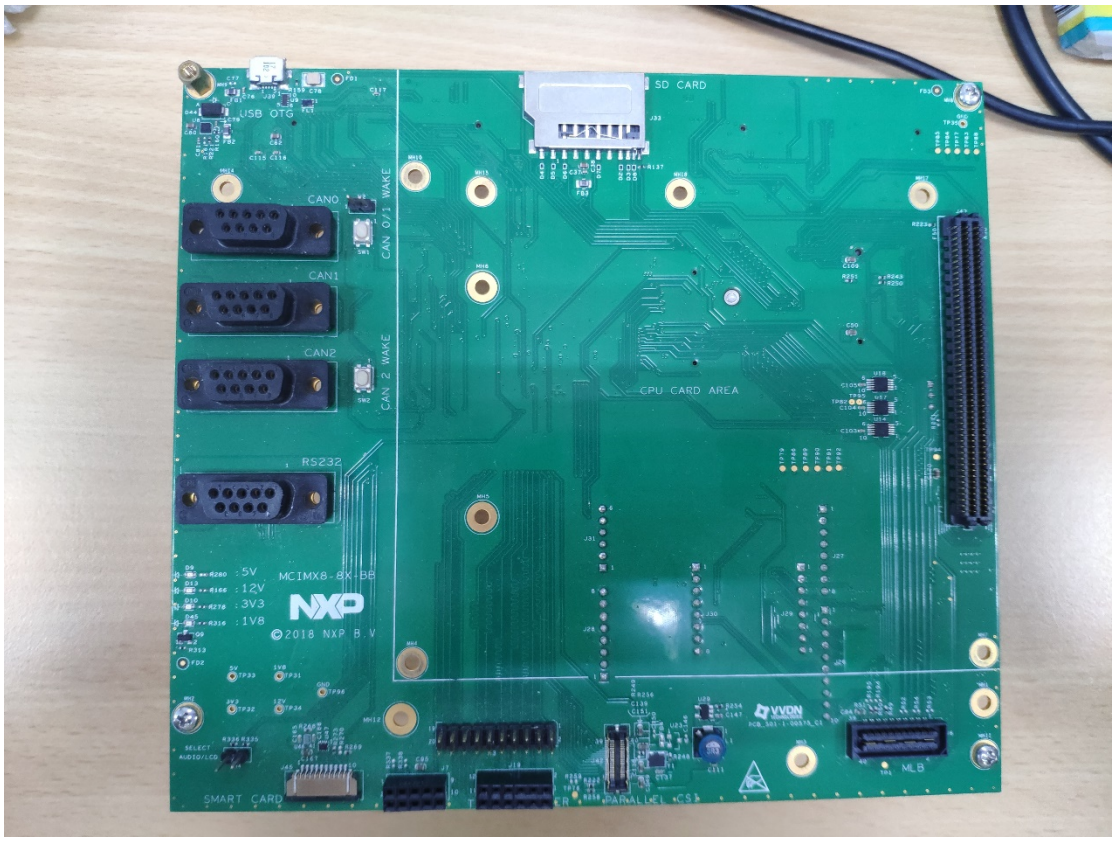

Figure 2. i.MX8-8X-BB

Connect the RS232 to a PC and connect the base board to the iMX.QXP/QM board.

1. Add the UART clock and pinctrl in the DTS.
   Add the UART clock and pinctrl in the DSP node as follows:

```
dsp: dsp@596e8000 {
        compatible = "fsl,imx8qxp-dsp";
        reg = <0x596e8000 0x88000>;
        clocks = ...
                <&uart2_lpcg 1>, <&uart2_lpcg 0>;
        clock-names = ... "uart_ipg", "uart_per";
        assigned-clocks = <&clk IMX_SC_R_UART_2 IMX_SC_PM_CLK_PER>;
        assigned-clock-rates = <80000000>;
        ...
        status = "disabled";
};
```

   Then build the image instead of the old one.

2. Modify the DSP side.

   The DSP supports the LPUART driver in the *dsp_framework/arch/peripheral.c* file. Change the LPUART_BASE from 0x5a090000 to 0x5a080000:

```
--- a/dsp_framework/arch/peripheral.h
+++ b/dsp_framework/arch/peripheral.h
@@ -93,7 +93,7 @@ struct nxp_lpuart {

#define UART_CLK_ROOT (80000000)
#define BAUDRATE (115200)
-#define LPUART_BASE  (0x5a090000)
+#define LPUART_BASE  (0x5a080000)

void dsp_putc(const char c);
void dsp_puts(const char *s);
```

   Then build the DSP with “DEBUG=1” and copy it to the board.

3. Run the DSP and print the debug info.

   This part is the same as on the iMX.MP board. Select the proper serial COM port and you will see the debug info. Note that the debug info cannot print in the QM board, because the UART is taken. See the *imx8mp-evk-dsp.dtb* and *imx8qxp-mek-dsp.dtb* files on how to add the UART CLK.

5. Building DSP framework on Windows OS

The DSP framework can be built also on Windows OS. The Xplorer software can be used to build the DSP framework on Windows OS. This chapter explains how to use Xplorer to build the DSP framework. Firstly, install the Xtensa Xplorer IDE. You can download the Xplorer IDE and Xploere licence form Cadence.

NOTE

Log into the XPG Cadence website with the NXP common XPG login credentials to download installers for the Xplorer IDE, Xtensa tools, and so on. For NXP internal use, contact the DSP owner to get the NXP common XPG login credentials. The Xplorer 7.0.8 version is used as an example and its default installation folder is *C:\usr\xtensa*.

5.1. Adding new configuration packages

Currently, the *hifi4_nxp_v4_3_1_3_dev_win32.tgz* configuration package, which is updated from the *hifi4_nxp_v3_3_1_2_dev_win32.tgz* configuration package, is used to build the DSP framework on Linux OS. The *hifi4_nxp_v3_3_1_2_dev_win32.tgz* configuration package is used as an example. Add this configuration package into Xplorer before building the code. You can get this configuration package and the corresponding memory map linker files from NXP. The required files are as following:

- *hifi4_nxp_v3_3_1_2_dev_win32.tgz*
- *memmap/mainsim folder*

When you have the DSP configuration package and memory map linker files, you can add a new configuration package into Xplorer as follows:

- Download and install Xtensa Tools for Xplorer.

  If you do not have the Xtensa Tools, you shall download and install it using Xplorer. Currently, the Xtensa Tool that we use is *XtensaTools_RF_2016_4_win32.tgz*. You can first open the Xplorer software and click the "RF-2016.4" option in the "XPG View" panel and select the "tools->Xtensa Tools->Xtensa Tools 11.04 for Windows" option. After you select it, you can click the download button to start the downloading process.

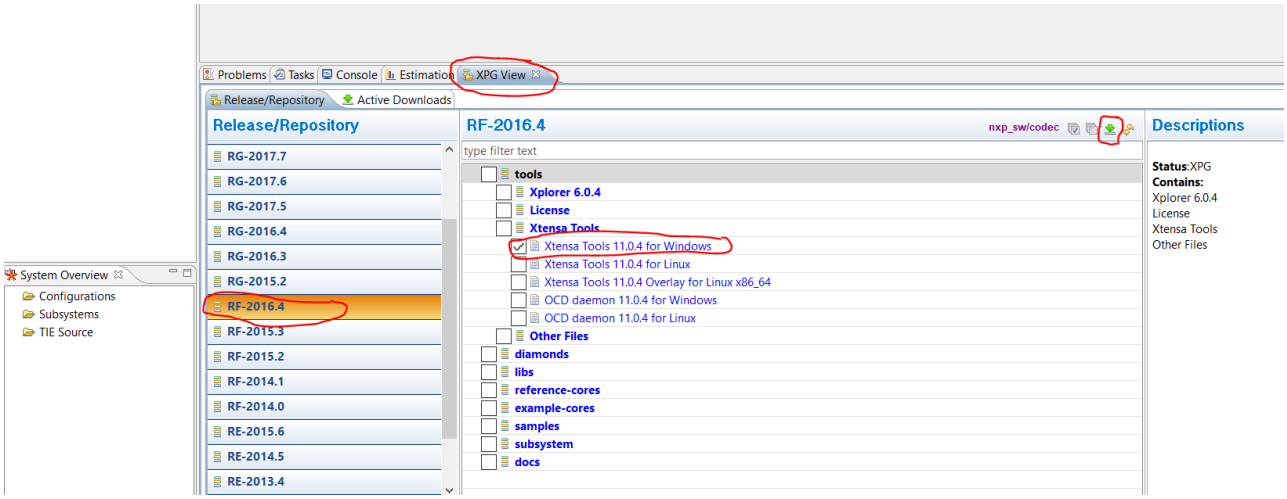

Figure 3. "XPG View" panel

  After the download finishes, right click the "Xtensa Tools 11.0.4 for Windows" option and select the "Install Xtensa Tools…" option in the new dialog. The installing process takes some time. The Xtensa Tool is installed successfully after this step. You can see this folder in the Xplorer's installing folder if everything is ok:

  C:\usr\xtensa\XtDevTools\install\tools\RF-2016.4-win32

- Add the configuration package into Xplorer.

  When you have the *hifi4_nxp_v3_3_1_2_dev_win32.tgz* package from NXP, you can add it into Xplorer. The first thing to do is to create a new folder called *build* in Xplorer's installing path if the *build* folder is not created already. The total path after this operation is as follows:

  C:\usr\xtensa\XtDevTools\downloads\RF-2016.4\build

- Place the *hifi4_nxp_v3_3_1_2_dev_win32.tgz* package into the new *build* folder.
  C:\usr\xtensa\XtDevTools\downloads\RF-2016.4\build\ hifi4_nxp_v3_3_1_2_dev_win32.tgz
- After you have performed the above steps, you can click the refresh button in the “XPG View” panel and find the “build” option in this panel.

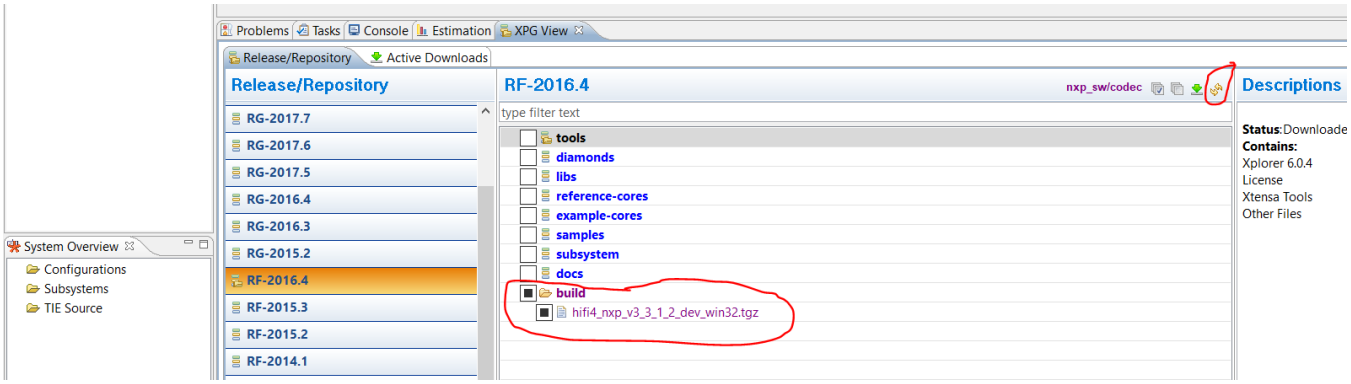

Figure 4. The “build” option

- Right click the *build->hifi4_nxp_v3_3_1_2_dev_win32.tgz* package and click the “Install Build…” option in the new dialog to start the installing process. This takes some time. You can see the following folder in the Xplorer’s installing folder if everything is OK.
  C:\usr\xtensa\XtDevTools\install\builds\RF-2016.4-win32\hifi4_nxp_v3_3_1_2_dev
- Add the new memmap linker files into Xplorer.
  After you add the *hifi4_nxp_v3_3_1_2_dev_win32.tgz* configuration package into Xplorer, you can add the new memmap linker files. You can copy the *mainsim* folder into Xplorer’s install folder to finish this process. The complete folder after this process is as follows:
  C:\usr\xtensa\XtDevTools\install\builds\RF-2016.4-win32\hifi4_nxp_v3_3_1_2_dev\xtensa-elf\lib\mainsim

After you complete the above three steps, the new configuration package and the corresponding memory map linker files are successfully added into Xplorer.

5.2. Creating the DSP framework Xplorer project

The DSP framework project must be created before using Xplorer to build it. The DSP framework code is in the *imx-audio-framework* package:

- *imx-audio-framework\dsp_framework*

You can create the DSP framework as follows:

- Open Xplorer and click the “File->New->Xtensa C/C++ project” option in the menu bar. You will see this dialog:

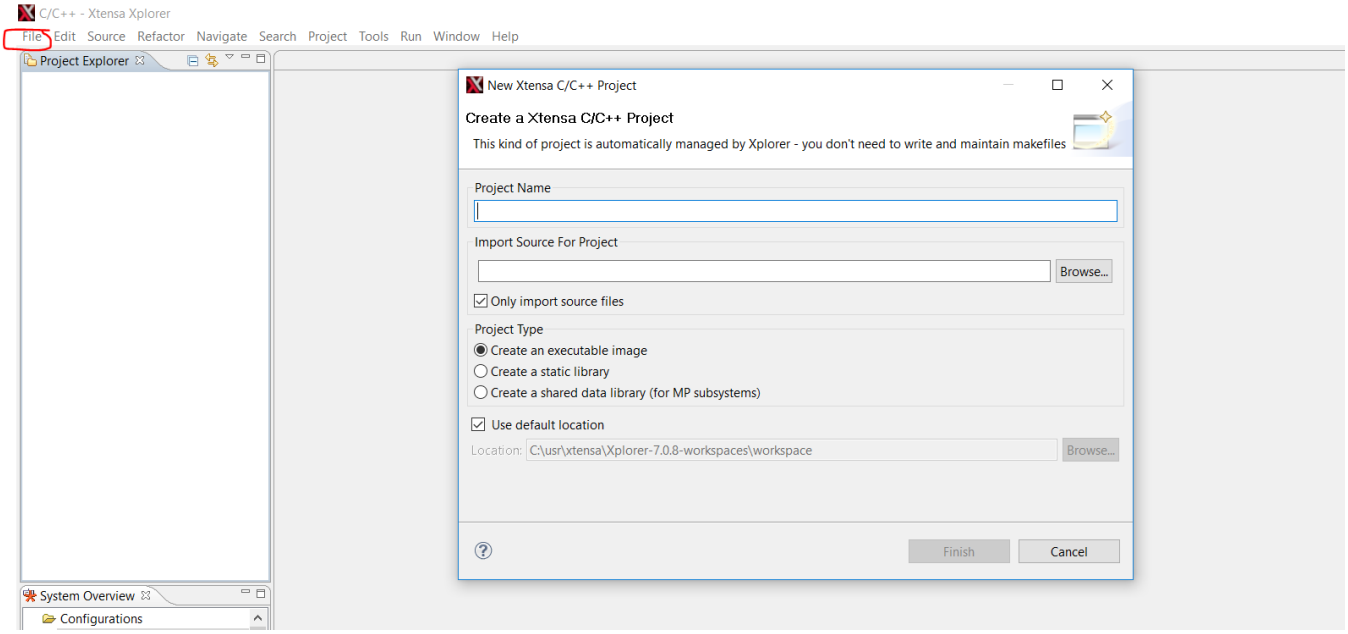

Figure 5. New Xtensa project

- Enter the project name and import DSP framework source code into the “New Xtensa C/C++ Project” dialog. Then click the “Finish” button.

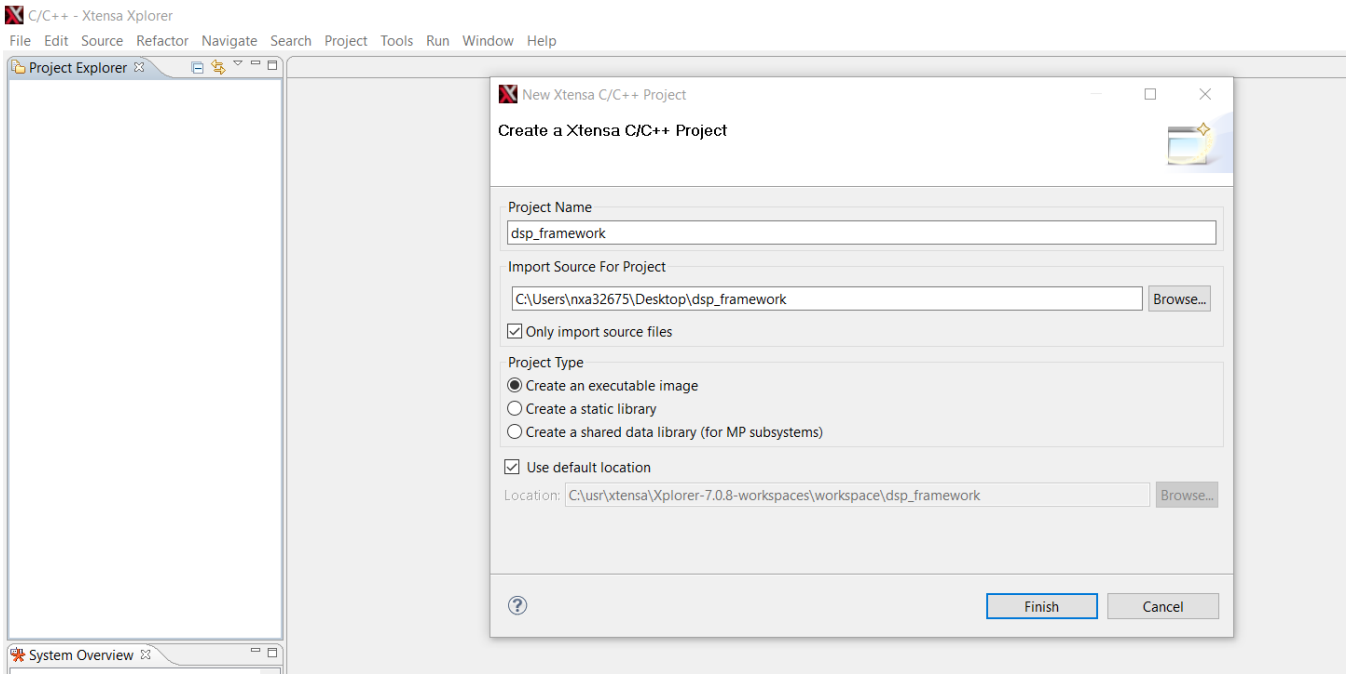

Figure 6. The “Finish” button

After the above two steps, the DSP framework project is successfully created. You can see the project in Figure 6.

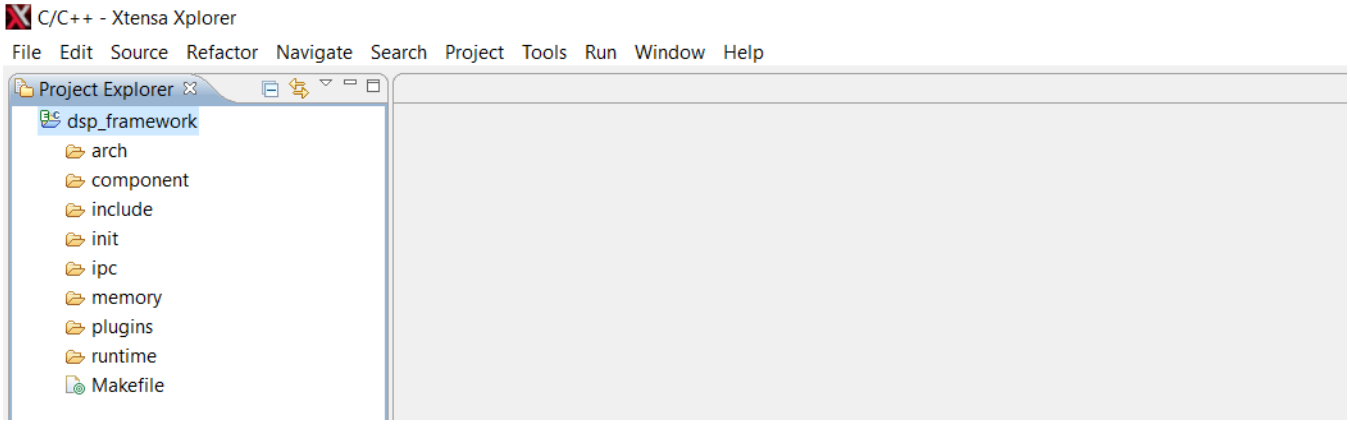

Figure 7. DSP framework project

5.3. Building DSP framework

When you created the DSP framework project, you can build its code. Choose the *memmap* linker files before the building process.

- Right-click the name of the DSP framework project in the “Project Explorer” panel and choose the “Build Properties…” option. You will see the “Build Properties for dsp_framework” dialog. The dialog is shown in Figure 7.

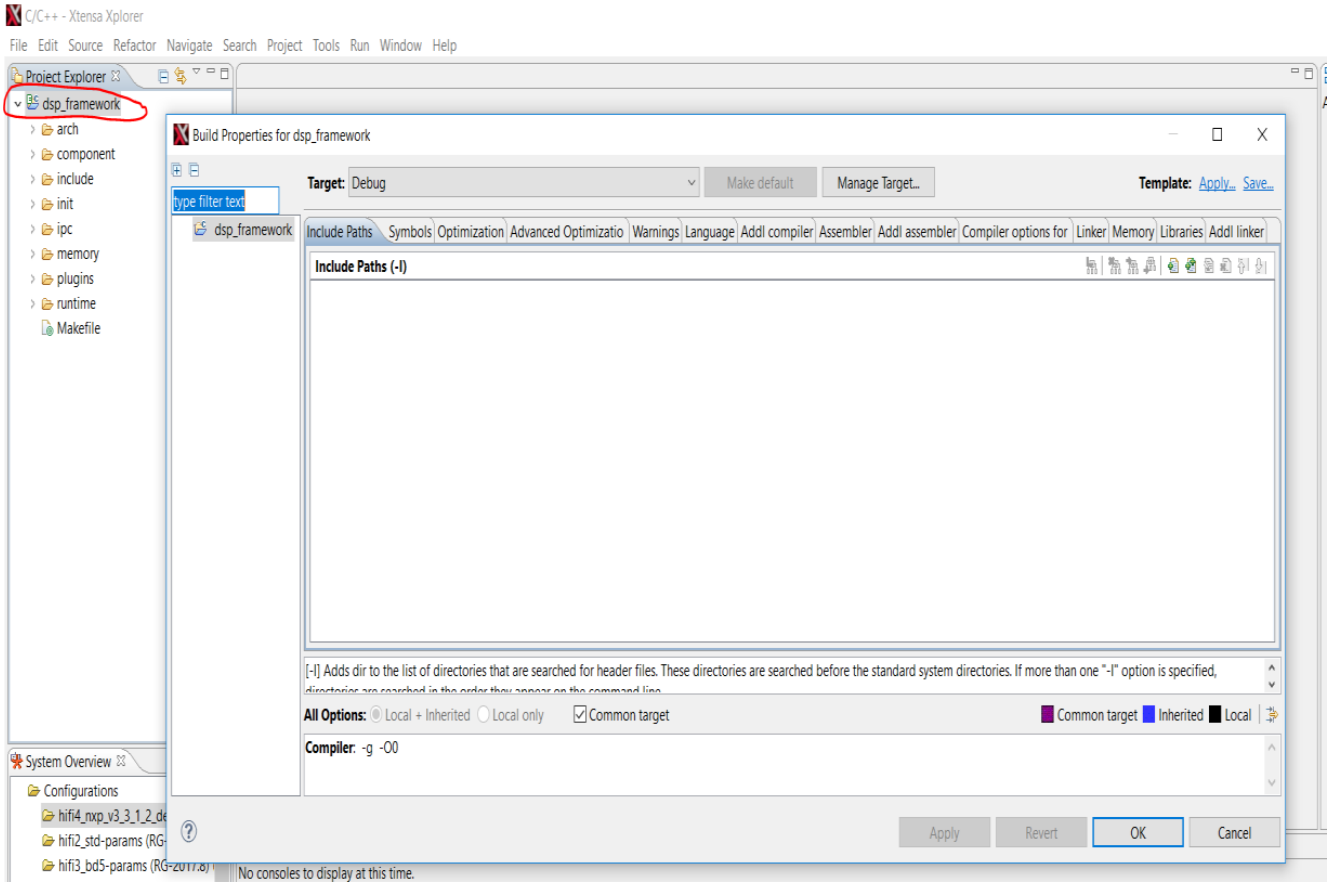

Figure 8. "Build Properties" dialog

- Click the "Addl" compiler to add options. Add the "-DPLATF_8m" attribute to build the firmware for i.MX8mp. Add the "-DPLATF_8MP_LPA" attribute to build the firmware for LPA. Add the "-DDEBUG" attribute to build the firmware with the print debug information.
- Click the "Linker" option and configure the custom LSP path as shown in Figure 8. Click the "OK" button to finish this process.

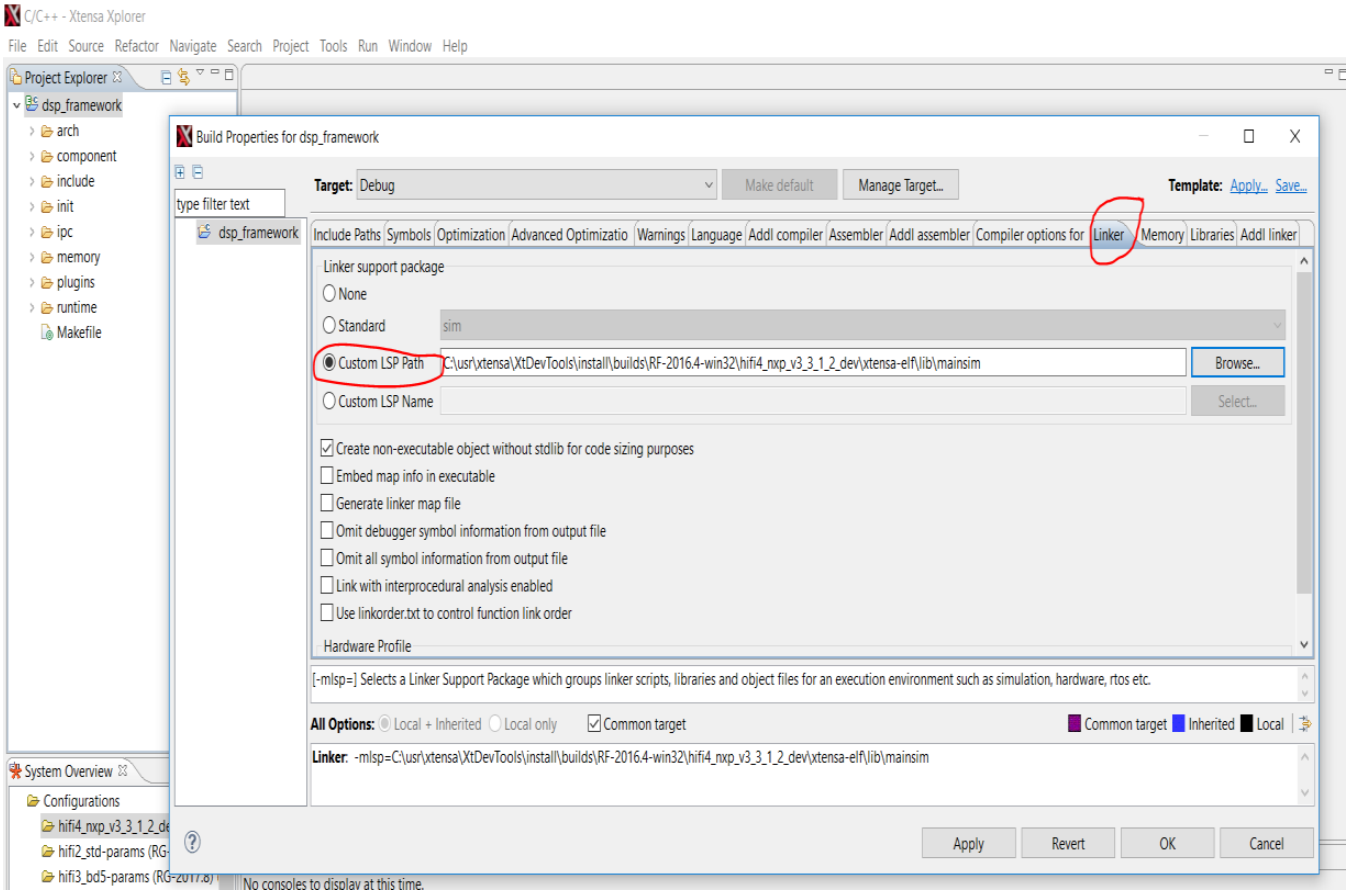

Figure 9. "Linker" option

- When you configured the *memmap* linker files, you can choose the *dsp_framework* project and the required DSP configuration to start the building process. The configuration is as follows:

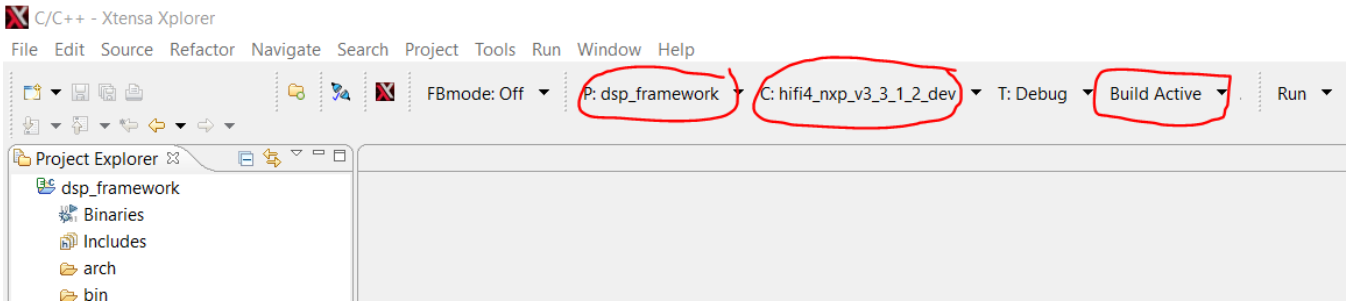

Figure 10. Build process configuration

- Click the "Build Active->Build Active" option to start building the DSP framework. This takes some time.
- After performing the above steps, you get the binary file called *dsp_framework* (which is the firmware of the DSP) in the following folder.

  C:\usr\xtensa\Xplorer-7.0.8-workspaces\workspace\dsp_framework\bin\hifi4_nxp_v3_3_1_2_dev\Debug\dsp_framework

If you want to use this binary file to run on a real board, rename the *dsp_framework* binary file as *hifi4.bin* and place it to a right place of "rootfs".

6. Building DSP wrapper and unit test

Before you compile the DSP wrapper and the unit test, set up the related toolchain. The DSP wrapper and the unit test use the Linaro compiler toolchain for the Yocto platform.

6.1. Installing Linaro compiler toolchain

Currently, the "gcc-linaro-4.9-2015.02-3-x86_64_aarch64-linux-gnu" toolchain is used to compile the DSP wrapper and the unit test's code for the Yocto platform. This toolchain shall be placed into the */usr* folder of your Linux OS server. If you want to successfully build the code, you can get more information from the *Makefile* file of the DSP wrapper and the unit test.

6.2. Building the code

When the Linaro toolchain is successfully installed on your server, you can compile the DSP-related code. You can execute the "make" command in the *imx-audio-framework* folder to compile the DSP wrapper and the unit test. If you want to compile them separately, see the *README* file in the *imx-audio-framework* folder. After the compiling process, you can find the binary files in the *imx-audio-framework/release* folder.

For the DSP wrapper:

- *imx-audio-framework/release/wrapper/lib_dsp_wrap_arm_elinux.so*

For the unit test:

- *imx-audio-framework/release/exe/dsp_test*

7. Usage of DSP binary files

7.1. Getting DSP binary files

You can get the DSP binary files of the DSP framework, DSP wrapper, and unit test directly from NXP or compile the source code to produce them yourself. Authorization is needed for the DSP codec wrapper and DSP codec binary files.

7.2. Binary files in Linux OS rootfs

To run these binary files, place them into the Linux OS rootfs. The location of the DSP framework is determined by the DSP driver, so you shall keep it in the specified place. The location of the DSP wrapper is determined by the GStreamer and you shall keep it in the specified place. You can change the location of the unit test. The binary files are in these folders:

- The unit test is here (default path):
  /unit_tests/DSP/dsp_test.out
- The DSP framework is here:
  /lib/firmware/imx/dsp/hifi4.bin
- The DSP wrapper is here:
  /usr/lib/imx-mm/audio-codec/wrap/lib_dsp_wrap_arm_elinux.so
- You can keep the DSP codec wrapper and the DSP codec in these folders of the Linux OS rootfs:
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_codec_wrap.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_mp3_dec.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_aac_dec.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_bsac_dec.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_dabplus_dec.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_drm_dec.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_mp2_dec.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_sbc_dec.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_dsp_sbc_enc.so
- Add DSP NXP codec wrapper lib:
  /usr/lib/imx-mm/audio-codec/dsp/lib_mp3d_wrap_dsp.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_aacd_wrap_dsp.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_vorbisd_wrap_dsp.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_wma10d_wrap_dsp.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_nbamrd_wrap_dsp.so
  /usr/lib/imx-mm/audio-codec/dsp/lib_wbamrd_wrap_dsp.so
- cplay utility (used to play compressed audio files):
  /usr/bin/cplay

7.3. Unit test and playing

7.3.1. dsp_test

After placing the binary files into the correct location of the rootfs, you can decode or encode audio streams directly using the unit test binary file. To decode one **.mp3* file, use this command:

```
./dsp_test -f1 -d16 -itest.mp3 -otest.pcm
```

```
dsp_test.out -f3 -r32 -t49 -d16 -ithetest_48000ps_chbr32.nac -othetest_48000ps_chbr32.pcm
dsp_test.out -f4 -il2-fl11.mp2 -ol2-fl11.pcm
```

For more information about the *dsp_test*, use this command:

```
./dsp_test
```

To play one music file using the GStreamer and DSP wrapper, use this command:

```
gplay-1.0 test.mp3
```

7.3.2. cplay

cplay is a standard Linux utility used to play compressed audio files over compressed audio interfaces. Currently, the DSP framework supports playing **.mp3* and **.wav* files. It also supports the pause and pause release features.

```
ls /snd/comprC1D0
cplay -c 1 -d 0 test.mp3
cplay -c 3 -d 0 -I PCM audio8k16S.wav
```

7.3.3. LPA

The Low-Power Audio (LPA) runs on the DSP in the OCRAM. To play streams, use the following command:

```
gplay-1.0 test.mp3 --audio-sink="tinycompresssink device=hw:3,0 enable-lpa=true provide-
clock=true"
```

8. Making codec wrapper and codec library

The library of the DSP codec wrapper and DSP codec is the loadable library. This chapter describes how to make the loadable library for the DSP.

The DSP loadable library is available as two different types: a fixed-location overlay and a position-independent library. For a fixed-location overlay, you can load the code into a predetermined location in the memory. For a position-independent library, you can load the code at an address determined during run time. You can link the loadable library using a special LSP named “piload” or “pisplitload” (see the *Xtensa Linker Support Packages (LSPs) Reference Manual*). The binary files that are used by the DSP framework belong to the position-independent library, so this chapter briefly discusses how to generate the position-independent library. For more detailed information, see Chapter 4 of the *Xtensa System Software Reference Manual*.

A position-independent library can be loaded and run at any address that supports both code and data, like a normal system RAM. Alternatively, you can use the “pisplitload” LSP to load the code and data into separate memory blocks located in local RAMs. The library location must be decided before the run time.

The Xtensa development toolchain must be installed before making a loadable library. After that, you can follow the steps below.

8.1. Finding custom LSPs

The loadable libraries must be linked to a custom linker support package. For the position-independent libraries, you don't have to generate or edit an LSP. Instead, you must link your position-independent library using the standard "pisplitload" LSP that is provided as a part of your configuration.

8.2. Source code modifying and compiling

The API only allows the main program to directly access a single symbol in the library, the "_start" symbol. The library cannot access any symbols in the main program directly. Any other symbol's address must be passed to or from the library as an argument to the "_start" function. This code is an example:

```
#include <stdio.h>
/* declare a printf function pointer */
int (*printf_ptr)(const char *format, …);
/* replace all calls to printf with calls through the pointer */
#define printf printf_ptr
/* This is the function provided by the library */
char * interface_func(unsigned int input)
{
       printf("executing function interface_func\n"); 13

       return "this is string returned from interface_func";
}
void * _start(int (*printf_func)(const char *format, …))
{
       printf_ptr = printf_func;
       /* The main application wants to call the function interface_func, but can't
       directly reference it. Therefore, this function returns a pointer to it, and
       the main application will be able to call it via this pointer. */
       return interface_func;
}
```

The main application calls the "_start" function, passes a pointer to "printf", and takes a pointer to *interface_func()* in return. If the library and the main program must communicate a value of more than one symbol, then the "_start" function call can return arrays of pointers, rather than just single pointers.

After finishing your source code, you can use "xt-xcc" of the Xtensa development toolchain to compile the code. Because the position-independent libraries can be loaded at any address, make sure that the code in the library is position-independent using the "-fpic" flag along with your normal compile options, as shown here:

```
 xt-xcc -O3 -o library.o -c library.c
```

8.3. Linking the library code

In this step, link the library code into a loadable library using the appropriate LSP. For position-independent library, you can use this command:

```
xt-xcc -mlsp=pisplitload -Wl,--shared-pagesize=128 -Wl,-pie -lgcc -lc -o library.so
library.o
```

After this command, you can get a position-independent library with the code and data loadable separately. If you want to get a contiguous position-independent library, you can use this command.

```
xt-xcc -mlsp=piload -Wl,--shared-pagesize=128 -Wl,-pie -lgcc -lc -o library.so
library.o
```

After the linking stage, you can get a loadable library which can be loaded by the DSP framework. The current DSP framework only supports loading the code and data sections separately.

9. Revision history

Table summarizes the changes done to this document since the initial release.

Table 1. Revision history

Revision number	Date	Substantive changes
0	06/2018	Initial release
1	01/2019	Added details about using the sound card feature that allows users to play mp3 files over ALSA compressed interface.
2	05/2020	Updated sections Introduction, Building DSP framework, and Binary files in Linux OS rootfs.
3	09/2020	Added support for the i.MX8 MP board. Added support for *.wav files playback by the ALSA compressed interface. Added details about DSP framework building.
4	01/2021	Updated the version of the toolchain. Added details about the firmware generation and the LPA.

Appendix A. Memory allocation for DSP

The DSP firmware is loaded into the memory by the DSP driver. The loading address is defined by the memory map linker files of the Xtensa development toolchain. You may change the loading address based on the memory map list of i.MX8 QXP, shown in Table 2. Note that the changed loading address is not used in i.MX8 QXP devices right now and the i.MX8 MP board does not support this change.

Table 2. Memory allocation on i.MX8 QXP

Cortex-A35/Cortex-M4	DSP	Content
—	0x80000000 ~ 0x806FFFFF	Reserved (cannot be used)
0x59700000 ~ 0x5971FFFF	0x80700000 ~ 0x8071FFFF	DSP OCRAM-system RAM
0x59720000 ~ 0x5973FFFF	0x80720000 ~ 0x8073FFFF	DSP OCRAM-system ROM
—	0x80740000 ~ 0x80FFFFFF	Reserved (cannot be used)
0x80700000 ~ 0x8073FFFF	—	Linux OS kernel (not visible from DSP)
0x81000000 ~ 0x9FFFFFFF	0x81000000 ~ 0x9FFFFFFF	SDRAM

Currently, the Linux OS kernel reserves the memory for the DSP in the SDRAM separately. The range of the reserved memory is 0x92400000 ~ 0x943fffff (32 MB). You may set this reserved memory by changing the *fsl-imx8qxp.dtsi* file in the *linux-kernel/arch/arm64/boot/dts/freescale* folder.

```
reserved-memory {
        ……..
        dsp_reserved: dsp@0x92400000 {
                no-map;
                reg = <0 0x92400000 0 0x2000000>;
        };
        ……..…..
```

```
}
```

The DSP driver splits the current reserved memory into two parts. One part is used to store the DSP firmware and the other part is a scratch memory for the DSP framework. The detailed information about these two parts is shown in Table 3.

Table 3. Two memory parts

0x92400000 ~ 0x933FFFFF	DSP firmware (16 MB)
0x93400000 ~ 0x943FFFFF	Scratch memory (16 MB)

NOTE

If you make changes in the memory map linker files of the Xtensa development toolchain, make the related changes for the DSP driver.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: DSPUG
Rev. 4
02/2021