NXP Semiconductors
User's Guide

Document Number: DSPUG
Rev. 0, 05/2018

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 platform. The current DSP framework can support up to 64 clients. They support these codecs:

Decoder:

- AAC-LC
- AAC plus(HE-AAC/HE-AACv2)
- BSAC
- DAB+
- MP2
- MP3
- DRM
- SBC

Encoder:

- SBC

Contents

2. System architecture

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

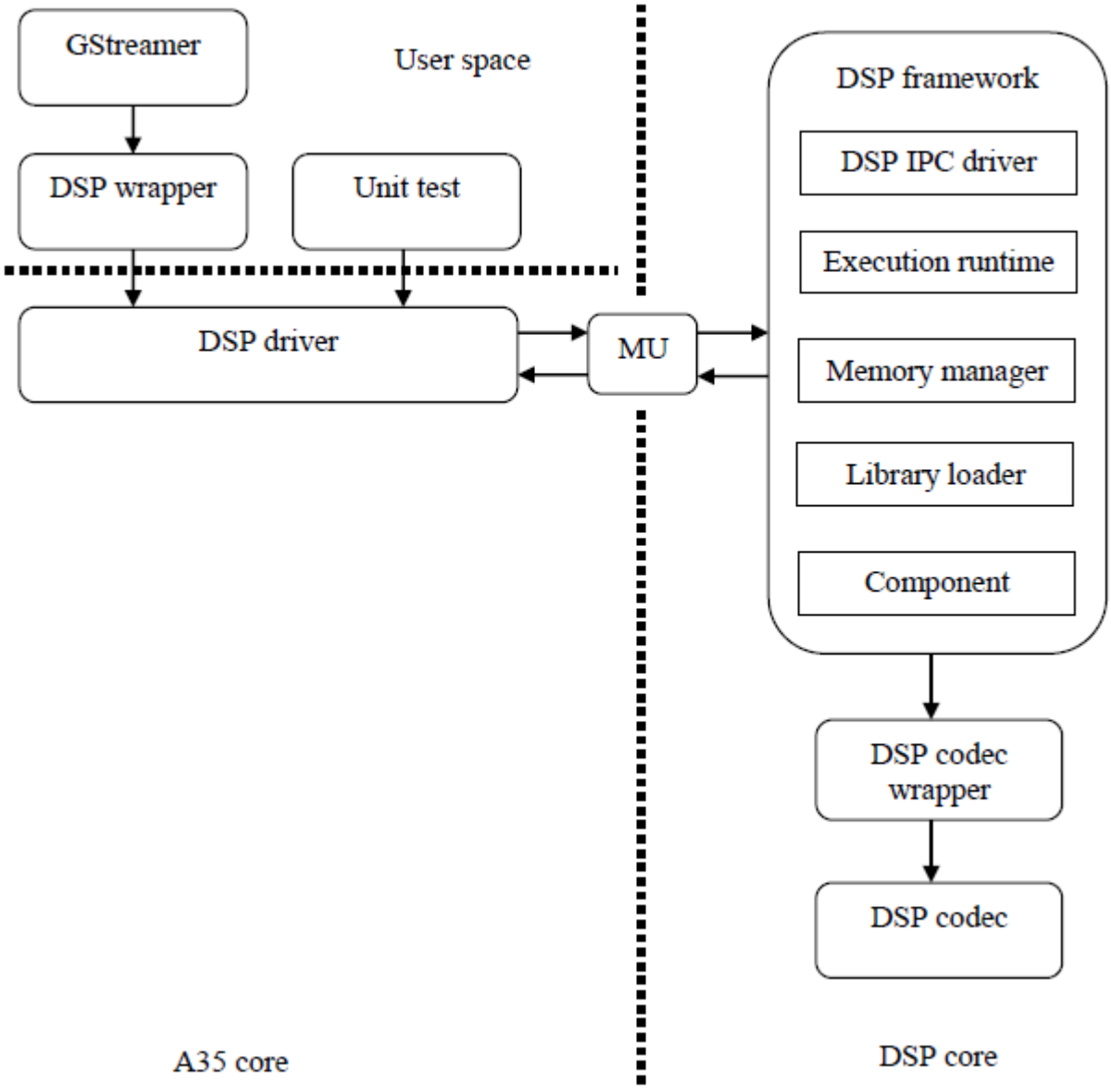

Figure 1. System architecture

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 interrupt 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.

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 these five 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*

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/component/audio/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. DSP toolchain setup

Before you compile the DSP-related code, set up the DSP-related toolchains. The DSP framework, DSP codec wrapper, and DSP codec use the Xtensa® development toolchain. The DSP wrapper and the unit test use the Linaro® compiler toolchain for the Yocto Project™ platform.

4.1. 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 this tool is *XtensaTools_RF_2016_4_linux.tgz*. 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. The current version of this tool is *hifi4_nxp_v3_3_1_2_dev_linux.tgz*. It is recommended to get the configurable memory map linker files from NXP. These files are in the *memmap/mainsim* folder.

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 to this path:

```
cd imx-audio-toolchain/Xtensa_Tool
cp -r memmap/mainsim ./builds/ RF-2016.4-linux/hifi4_nxp_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.

4.2. Linaro compiler toolchain

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 Project platform. Place this toolchain into the *usr* folder of your Linux OS server. To build the code successfully, find more information in the *Makefile* file of the DSP wrapper and unit test.

5. Usage of DSP binary files

5.1. Getting DSP binary files

Get the DSP binary files directly from NXP or compile the source code to produce them yourself.

When the DSP-related toolchains are successfully installed on your server, compile the DSP-related code. You may execute the make command in the *imx-audio-framework* folder to compile the DSP framework, DSP wrapper, and unit test. To compile them separately, see the *README* file in the *imx-audio-framework* folder. After the compiling process, the binary files are in the *imx-audio-framework/release* folder.

- The DSP framework is here:
  imx-audio-framework/release/hifi4.bin
- The DSP wrapper is here:
  imx-audio-framework/release/wrapper/lib_dsp_wrap_arm_elinux.so
- The unit test is here:
  imx-audio-framework/release/exe/dsp_test

You must have authorization for the DSP codec wrapper and the DSP codec binary files.

5.2. 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 should keep it in the specified place. The location of the DSP wrapper is determined by the GStreamer and you should keep it in the specified place. You can change the location of the unit test. The binary files are in these folders:

- 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 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

5.3. Unit test and playing

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
```

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
```

6. 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 1.

Table 1. Memory allocation

Cortex-A35/Cortex-M4	DSP	Content
—	0x80000000 ~ 0x806FFFFF	Reserved (can't be used)
0x59700000 ~ 0x5971FFFF	0x80700000 ~ 0x8071FFFF	DSP OCRAM-system RAM
0x59720000 ~ 0x5973FFFF	0x80720000 ~ 0x8073FFFF	DSP OCRAM-system ROM
—	0x80740000 ~ 0x80FFFFFF	Reserved (can't 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 0x8e000000 ~ 0x8fffffff (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@0x8e000000 {
no-map;
reg = <0 0x8e000000 0 0x1ffffff>;
};
…….....
}
```

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 2.

Table 2. Two memory parts

0x8e000000 ~ 0x8eFFFFFF	DSP firmware (16 MB)
0x8f000000 ~ 0x8FFFFFFF	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.

7. Making loadable library for 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.

7.1. Finding custom LSPs

The loadable libraries must be linked with 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.

7.2. 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 take 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 -fpic -O3 -o library.o -c library.c
```

7.3. Linking the library code

In this step, you will link the library code into a loadable library using the appropriate LSP. For position-independent library, you can use the following 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 code and data loadable separately. If you want to get a contiguous position-independent library, you can use the following 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 DSP framework. In addition, for current DSP framework, it can only support loading code and data section separately.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: DSPUG
Rev. 0
05/2018