Texas Instruments Incorporated

PASDK - Reference Audio I/O Interfaces

Guidelines to port on custom hardware

DRAFT Version

9/15/2017

This document provides a high level overview of the Reference K2G Audio Platform (HW + SW). This is not intended to be a self-contained document, but as a supplement device TRMs, H/W schematics & other software user-guides & documentation available with PASDK.

Table of Figures

Table of Contents

Introduction

The K2G EVM along with the K2G Audio Daughter Card serves as the basic hardware platform upon which the PASDK reference software validation & testing activities are performed.

At the time of writing this guide, the reference schematics are available here:

- Mistral K2G Rev. D EVM: http://www.ti.com/lit/df/sprr302/sprr302.pdf
- Mistral K2G EVM Audio Daughter Card: http://www.ti.com/lit/df/sprr310/sprr310.pdf

In addition to the above-mentioned, PASDK also employs HDMI as the primary multichannel, digital input interface to handle many audio formats of interest (i.e, IC Certification needs). The HDMI input is enabled by an "add-on" HDMI repeater module from MDS (Momentum Data Systems) that's interfaced with the I2S Header, exposed & available on the Audio Daughter Card.

Note: The HDMI card itself is not supported by TI. It's only used within PASDK setup for extracting HDMI Audio & redirecting it to K2G platform, as appropriate for McASP reception in I2S mode.

While the McASP peripheral is capable of supporting a wide variety of serial data formats, PASDK is mostly concerned about the I2S (or 2-slot TDM) mode of operation.

At the time of writing this guide, the framework has been validated for correctness of operations only with synchronous "I-topology" (i.e, single input, single output) – **I13** in PASDK.

It is expected that this document serves as a high level guide that explains the programming choices to enable the K2G Audio platform (i.e, K2G EVM + Audio DC + HDMI-card) and in turn serves to guide others working with custom target hardware, with different components/interfaces than used on the said reference platform.

Audio I/O on the K2G Hardware

The below topology diagram – from the Audio Daughter Card reference - provides an overview of the various McASP instances that have been wired up as necessary with the various Audio interfaces, for the software validation of the distributed, multichannel audio framework available within PASDK for K2G.

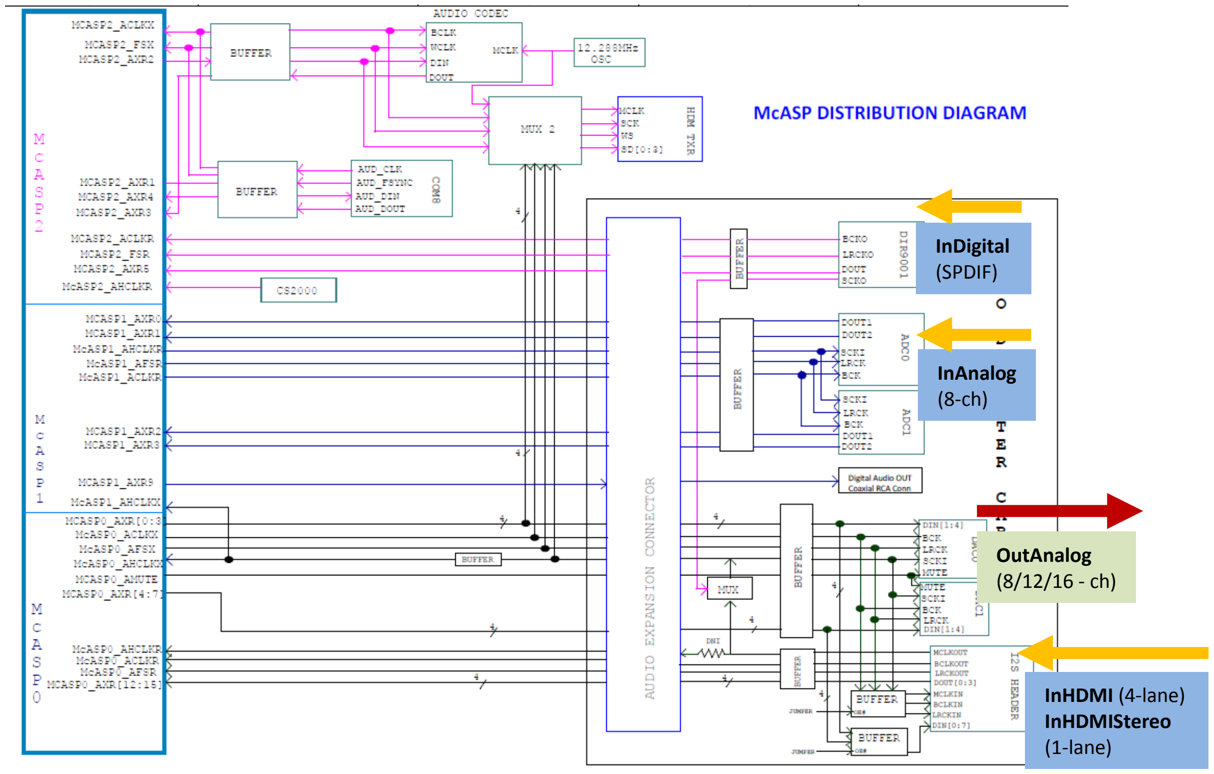


Figure 1: **K2G McASP Topology**

A variety of Audio I/O interfaces are available on these 2 boards, although only a subset of them is validated with the multichannel audio framework.

- SPDIF Input via DIR9001
- MultiChannel (up to 8-ch) Analog Input via 2xPCM1865
- MultiChannel (up to 16-Ch) Analog Output via 2xPCM1690

It's significant to mention here that the AIC3x combo-codec, on-board the K2G "main" EVM, does not figure into PASDK's list of "supported interfaces".

Audio I/O Clocking

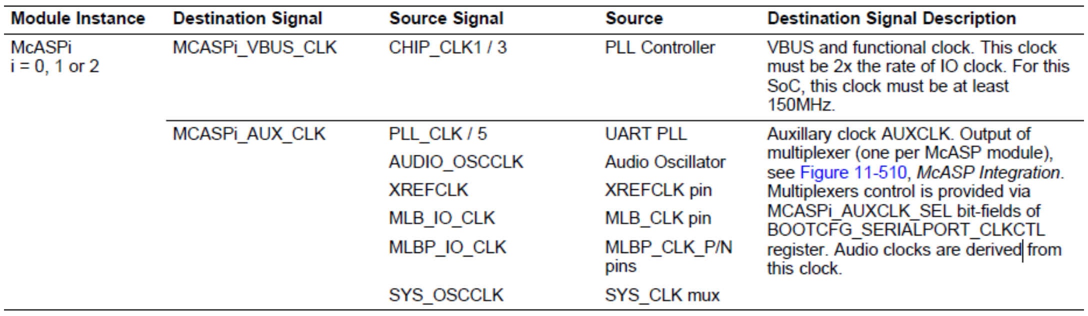
At the time of writing this guide, PASDK supports only *synchronous* audio operations. This means that the input & the output sections are clocked by (or derived from) a single, common source.

Note: Even if identical in frequencies, independent/discrete clock sources are still *asynchronous*. (They're *syntonous,* though.)

This implies that the OutAnalog device (DAC) requires to be *synchronized* with the chosen input device, always. It naturally follows & is also noteworthy that the DAC does not have any dedicated Xtal, but relies upon McASP0 AHCLKX to supply the clock, for its operations.

Owing to the nature of the component's clock routing & mux-options, the following notes on the input interfaces on the K2G EVM + Audio DC hardware platform are deemed to be useful, to be mindful about.

- Analog
    - K2G device is equipped with an on-board Audio-OSC, fed by a 22.5792 MHz Xtal, on the "main EVM". This 'AUDIO_OSCCLK' is available as an internal aux-source for all the McASP instances. (Refer: K2G TRM, ***11.9.3 McASP Integration***)



Module Instance	Destination Signal	Source Signal	Source	Destination Signal Description
McASPi i = 0, 1 or 2	MCASPi_VBUS_CLK	CHIP_CLK1 / 3	PLL Controller	VBUS and functional clock. This clock must be 2x the rate of IO clock. For this SoC, this clock must be at least 150MHz.
	MCASPi_AUX_CLK	PLL_CLK / 5	UART PLL	Auxillary clock AUXCLK. Output of multiplexer (one per McASP module), see Figure 11-510, *McASP Integration*. Multiplexers control is provided via MCASPi_AUXCLK_SEL bit-fields of BOOTCFG_SERIALPORT_CLKCTL register. Audio clocks are derived from this clock.
		AUDIO_OSCCLK	Audio Oscillator	
		XREFCLK	XREFCLK pin	
		MLB_IO_CLK	MLB_CLK pin	
		MLBP_IO_CLK	MLBP_CLK_P/N pins	
		SYS_OSCCLK	SYS_CLK mux	

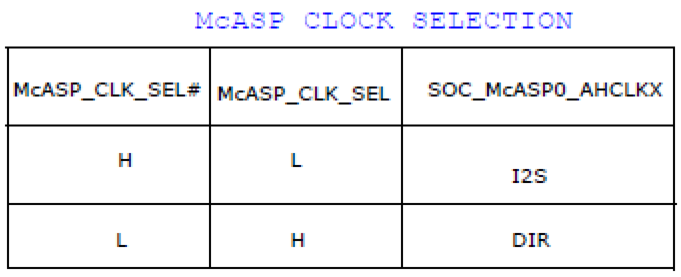
Figure 2: **K2G McASP Aux Clock**

    - The ADCs is clocked with McASP1 AHCLKR (i.e, McASP Master) in lieu of the need for the DACs to stay synchronous. i.e, McASP0 is also programmed with the same AUX clock source.

- Digital (SPDIF)
    - While active, the PLL (on board the DIR9001) recovers the active clock and supplies the same via these output signals: a Master Clock (DIR_SCKO), a bit-clock (DIR_BCKO) & a Frame Clock (DIR_LRCKO).
    - The Bit & the Frame Clocks are sufficient to master the *receive* section of McASP2.
    - The DIR_SCKO signal is used to clock AHCLKX i.e, the *transmit* section of McASP0. (i.e, then, McASP masters the DACs with internally generated Frame+Bit clocks).

- HDMI
    - The “I2S Header” exposed on the Audio DC is used to interface with the HDMI Repeater card, which supplies the necessary clocks.
    - I2SHDR_McASP0_AFSR & I2SHDR_McASP0_ACLKR are used to master the Frame & Bit clocks, respectively, of the *receive* section of McASP0.
    - I2SHDR_MCLKOUT is used as the external source to clock AHCLKX i.e, the *transmit* section of McASP0. (i.e, then, McASP masters the DACs with internally generated Frame+Bit clocks).

- The Audio DC is equipped with a flexible, programmable mux that allows the McASP0’s AHCLKX is to be chosen between the above-mentioned 2 sources.



McASP CLOCK SELECTION

McASP_CLK_SEL#	McASP_CLK_SEL	SOC_McASP0_AHCLKX
H	L	I2S
L	H	DIR

Figure 3: **Audio DC McASP0 Clock Mux**

Note: On the reference K2G Audio Platform, owing to the above-explained clock routing/muxing options available, there will be certain limitations in the reference software’s flexibility to demonstrate seamless IO Switching (across multiple sources or sinks), underscored by the need to maintain synchronous relationships between Input & Output devices.

Audio I/O Software Architecture – An Overview

PASDK employs distributed software architecture for processing audio data on K2G. Within this architecture, software interfaces exist at key locations in the data flow to allow for generality, modularity, and extensibility. This data flow typically begins and ends at a physical interface, such as a serial port. Since PASDK is built atop the SYS/BIOS operating system, one of the choices available for managing physical interfaces is the SIO device driver standard. Specifically, the interface between the core processing loop of the framework and the reception and transmission of audio data consists of SIO data structures and function calls.

The management of these interfaces takes place as part of a larger, more general facility known as Input/Output Switching (IOS). As its name implies, the primary function of IOS is to provide the mechanism through which various sources and sinks are selected under user control for connection to the core processing of the Audio Framework.

As a concrete example, assume that a single stream of input audio data is received from the McASP which is connected to an external DIR. Likewise, a single stream of output data is transmitted via the McASP to external DACs. Since it is a serial port, the McASP shifts in one audio word at a time into its internal shift register. For efficient processing, these words are collected into a block of samples. This buffer is then examined for supported IEC bitstreams and is handed to the appropriate decoder which converts the raw input data into floating point PCM data. This data is then processed (e.g. via custom ASPs) and subsequently the data is encoded into a form suitable for transmission. Finally, the encoded data is shifted out the McASP one word at a time.

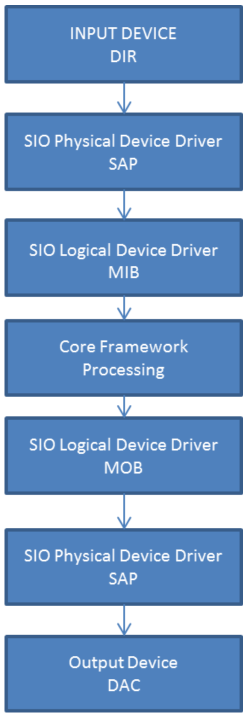

INPUT DEVICE
DIR
SIO Physical Device Driver
SAP
SIO Logical Device Driver
MIB
Core Framework
Processing
SIO Logical Device Driver
MOB
SIO Physical Device Driver
SAP
Output Device
DAC

Figure 4: Example/Reference Data Flow in PASDK

Using the above example, we can highlight two general interfaces of interest.

- First, the code which configures the McASPs and collects the data into blocks is called the SIO Physical Device Driver.
- Second, the code which scans the input for bitstreams is called the SIO Logical Device Driver.

In this example, both layers are provided by PASDK referencecode. The physical device driver for both the input and the output is SAP (Serial Audio Port), the input logical driver is MIB (Multichannel Input Buffer) and the output logical driver is MOB (Multichannel Output Buffer).

Note that the core processing loop of the Audio Framework only interfaces with the logical device layer. In a sense, the physical device layer is hidden or abstracted from the core via the logical device layer. The connectivity of the two layers is accomplished through the SIO technique of driver stacking.

It is important to note that the functionality of each layer can be replaced without modifying other layers. For example, if the McASP is connected to an ADC then one could replace MIB with another device driver which doesn't perform ‘auto-detection’ (i.e, identification of IEC bitstream format), but the physical device layer (SAP) could be re-used without change.

Audio flow begins with the choice of specific input and output devices. What is actually selected, via IOS, are individual data structures representing each device. These data structures are referred to as Device Configuration Parameters; or when the context is clear, just *parameters*. A typical system will have one set of parameters for each possible device selection. In the above example, there would be a parameter structure for the DIR input and one for DAC output.

A PASDK-compliant SIO driver must define a device parameter structure which can be used, via IOS, to configure particular device instances.

By creating one or more such structures and placing their pointers in the *devinp* or *devout* array, as appropriate, PASDK is able to manage the use and configuration of any device. In other words, a general purpose peripheral driver (*e.g.*, SAP) is *parameterized* via the use of a parameter structure to work with a particular device in a particular configuration. In order for PA/F to handle different devices in a uniform manner all such parameter structures must use a common header which is defined as PAF_SIO_Params (paf_sio.h).

A particular driver uses this common structure upon which to build a more extensive parameter structure which includes peripheral or device specifics.

Here we describe the PAF_SIO_Params structure in detail.

```
struct DXX_Params_
{
   const char   *name;         // driver name, e.g. "SAP"
   XDAS_Int32    moduleNum;
   XDAS_Void    *pConfig;
   XDAS_Int16    wordSize;
   XDAS_Int16    precision;

   XDAS_Int32  (*control)( DEV_Handle, const struct PAF_SIO_Params *, XDAS_Int32,
XDAS_Int32 );
};


typedef struct PAF_SIO_Params
{
   Int              size;    // Type-specific size
   struct DXX_Params_ sio;     // Common parameters
} PAF_SIO_Params;
```

PAF SIO Parameter	Description
size	Used as per XDAIS and describes the sizeof the data structure in bytes.
sio.name	String which specifies the physical and logical drivers used to communicate with the device.
sio.moduleNum	Driver specific. In general, it is the peripheral module's ID (0, 1, etc.).
sio.pConfig	Driver specific. In general, it is a pointer to a configuration structure specific to the underyling peripheral.
sio.wordSize	Driver specific. In general, it is the maximum and default word size, in bytes, for this device configuration.
sio.precision	Driver specific. In general, it is the maximum and default precision, in bits, for this device configuration.
sio.control	Pointer to function which processes SIO control codes.

Features/Files to be customized:

For the K2G Audio setup, the below platform-level definitions enable the hardware to be serviced by the audio i/o drivers. The EVM or Audio DC (incl components) or the HDMI related configurations provided with PASDK are provided as a validated reference or example.

For other custom platforms, equivalent configurations (in equivalent custom board files) need to be defined, as appropriate, for the drivers to service them.

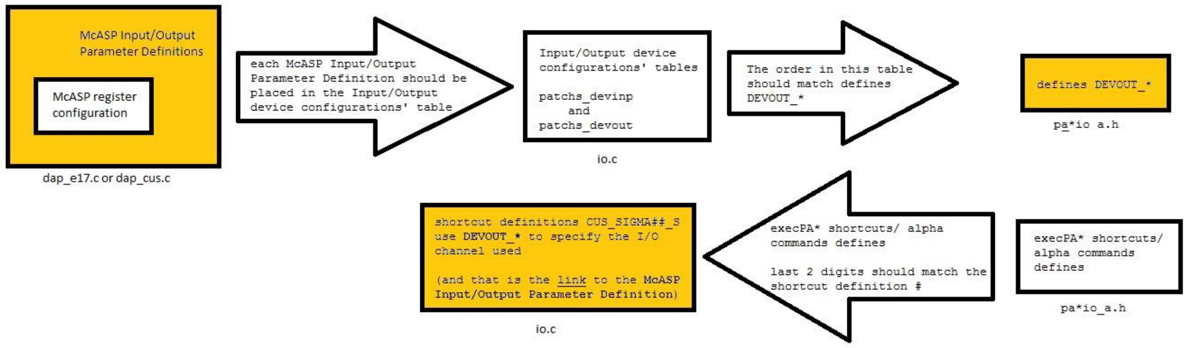

McASP Input/Output Parameter Definitions
McASP register configuration
dap_e17.c or dap_cus.c
each McASP Input/Output Parameter Definition should be placed in the Input/Output device configurations' table
Input/Output device configurations' tables
patchs_devinp and patchs_devout
io.c
The order in this table should match defines DEVOUT_*
defines DEVOUT_*
pa*io a.h
shortcut definitions CUS_SIGMA##_S use DEVOUT_* to specify the I/O channel used
(and that is the link to the McASP Input/Output Parameter Definition)
io.c
execPA* shortcuts/ alpha commands defines
last 2 digits should match the shortcut definition #
execPA* shortcuts/ alpha commands defines
pa*io_a.h

Figure 5: **Overview of the Audio IO Structures**

Note: Only a limited set of functionalities provided by the audio interfaces on the reference platform are exploited (as necessary for PASDK operations) and this is not intended to be an exhaustive reference for the components (ADC/DAC/DIR/HDMI etc) themselves.

Input/Output device configuration tables (io.c & pa_i13_evmk2g_io_a.h)

Each McASP Input/Output Parameter Definition should be placed in the Input/Output device configurations' tables - patchs_devinp and patchs_devout.

```
// Input device configurations & shortcut definitions

patchs_devinp[1] =
{
    DEVINP_N,
        // These values reflect the definitions DEVINP_* in pa*io_a.h:
       NULL,                                               // InNone
      (const PAF_SIO_Params *) &SAP_D10_RX_HDMI_STEREO,     // InHDMIStereo
      (const PAF_SIO_Params *) &SAP_D10_RX_HDMI,           // InHDMI
      (const PAF_SIO_Params *) &SAP_D10_RX_DIR,            // InDigital
      (const PAF_SIO_Params *) &SAP_D10_RX_ADC_44100HZ,    // InAnalog
};
```

The order in this table matches DEVINP_* in **pa_i13_evmk2g_io_a.h**

```
// These values reflect the definition of devinp[]
#define DEVINP_NULL              0
#define DEVINP_HDMI_STEREO       1
#define DEVINP_HDMI              2
```

```
#define DEVINP_DIR              3
#define DEVINP_ADC              4
#define DEVINP_N                5
```

McASP register configurations (sap_d10.c)

These are generic sets of register configurations that can be used for multiple inputs/outputs. These are not dependent on the McASP port number.

These structures are defined in sap_d10.c file under these sections:

```
// McASP Input Configuration Definitions

// McASP Output Configuration Definitions
```

For example, the below configuration for a digital input device basically configures the McASP peripheral. Please pay attention to the clocks' configuration (INTERNAL/ EXTERNAL) and how they relate to the K2G reference hardware platform.

```
static const MCASP_ConfigRcv rxConfigDIR =
{
    MCASP_RMASK_OF(0xFFFFFFFF),
    MCASP_RFMT_RMK(
        MCASP_RFMT_RDATDLY_1BIT,
        MCASP_RFMT_RRVRS_MSBFIRST,
        MCASP_RFMT_RPAD_RPBIT,
        MCASP_RFMT_RPBIT_OF(0),
        MCASP_RFMT_RSSZ_32BITS,
        MCASP_RFMT_RBUSEL_DAT,
        MCASP_RFMT_RROT_NONE),
    MCASP_AFSRCTL_RMK(
        MCASP_AFSRCTL_RMOD_OF(2),
        MCASP_AFSRCTL_FRWID_WORD,
        MCASP_AFSRCTL_FSRM_EXTERNAL,
        MCASP_AFSRCTL_FSRP_ACTIVELOW),
    MCASP_ACLKRCTL_RMK(
        MCASP_ACLKRCTL_CLKRP_RISING,
        MCASP_ACLKRCTL_CLKRM_EXTERNAL,
        MCASP_ACLKRCTL_CLKRDIV_DEFAULT),
    MCASP_AHCLKRCTL_RMK(
        MCASP_AHCLKRCTL_HCLKRM_EXTERNAL,
        MCASP_AHCLKRCTL_HCLKRP_RISING,
        MCASP_AHCLKRCTL_HCLKRDIV_DEFAULT),
    MCASP_RTDM_OF(3),
    MCASP_RINTCTL_DEFAULT,
    MCASP_RCLKCHK_DEFAULT
};
```

Note: Please refer to K2G TRM & McASP peripheral descriptions to understand, modify the individual register/field values appropriately.

McASP I/O parameter definitions (sap_d10.c)

These are structures that are specific for each input/output. These are dependent on the McASP port number and actual pins used for that I/O (pinmask). The **McASP Input/Output Parameter Definitions** point to a **McASP register configuration**. Multiple **McASP Input/Output Parameter Definitions** can use the same **McASP register configuration** if the parameters are common between I/Os.

Here is an example of McASP Input Parameter Definition for the SPDIF receiver device:

```
const SAP_D10_Rx_Params SAP_D10_RX_DIR =
{
    sizeof (SAP_D10_Rx_Params),                  // size
    "SAP",                                       // name
    MCASP_DEV2,                                  // moduleNum --> mcasp #
    (Void *)&rxConfigDIR,                        // pConfig
    4,                                          // wordSize (unused)
    24,                                          // precision (unused)
    D10_sapControl,                              // control
    0x00000020,                                  // pinMask
    (D10_MCLK_DIR << D10_MCLK_SHIFT),                // mode
    0,0                                          // unused[2]
};
```

Please note that the last fields are custom fields that are currently used for the EVM, one may add or remove fields to this structure (after pinmask) as needed for the custom hardware:

```
(D10_MCLK_DIR << D10_MCLK_SHIFT),              // mode
 0,0                                          // unused[2]
```

The above structures are available under these sections, in the said file.

```
// SAP Input Parameter Definitions
// SAP Output Parameter Definitions
```

The information of McASP port number and pinmask will come from the custom system's block diagram and/or schematic. In the above example, **McASP2** has been used to interface with the SPDIF receiver.

D10 is the codename for the reference audio platform (i.e, K2G EVM + Audio DC + HSR41). The below function is responsible for managing the input-status & clock-divider values.

```
XDAS_Int32 D10_sapControl (DEV2_Handle device, const PAF_SIO_Params *pParams,
XDAS_Int32 code, XDAS_Int32 arg);
```

This function is called by the peripheral driver (SAP) in response to various SIO_ctrl() calls made by the framework. The stacked nature of the driver architecture allows for anyone to implement their own _equivalent_ control function, as appropriate for their custom hardware.

Shortcuts definitions and Alpha commands (pa_i13_evmk2g_io_a.h & io.c)

The last 2 digits of the value defined for execPA* shortcuts/alpha commands should match the shortcut definition # in io.c. For example, for digital (spdif) input in pa_i13_evmk2g_io_a.h is:

#define execPAIInDigital 0xf1**23**

And 0x23 = 35 (in 0xf1**23**). So in io.c the digital (spdif) input will have the shortcut **CUS_SIGMA35_S**:

```
// execPAIInDigital
#define CUS_SIGMA35_S \
  writeDECSourceSelectNone, \
    writePA3Await(rb32DECSourceDecode,ob32DECSourceDecodeNone), \
    writeIBUnknownTimeoutN(2*2048), \
    writeIBScanAtHighSampleRateModeDisable, \
    writePCMChannelConfigurationProgramStereoUnknown, \
    writePCMScaleVolumeN(0), \
    writeDECChannelMapFrom16(0,1,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3), \
    writeIBEmphasisOverrideDisable, \
    writeIBPrecisionDefaultOriginal, \
    writeIBPrecisionOverrideDetect, \
    writeIBSampleRateOverrideStandard, \
    writeIBSioSelectN(DEVINP_DIR), \
    wroteDECSourceProgramUnknown, \
    writeDECSourceSelectAuto, \
    0xcdf0,execPAIInDigital

#pragma DATA_SECTION(cus_sigma35_s0, ".none")
const ACP_Unit cus_sigma35_s0[] = {
    0xc900 + 0 - 1,
    CUS_SIGMA35_S,
};

const ACP_Unit cus_sigma35_s[] = {
    0xc900 + sizeof (cus_sigma35_s0) / 2 - 1,
    CUS_SIGMA35_S,
};
```

Note: The shortcut definitions CUS_SIGMA##_S use DEVINP_* or DEVOUT_* to specify the I/O devices used. Eg: execPAIInDigital employs: **writeIBSioSelectN(DEVINP_DIR)**

Note: The function ACP_main_cus() in **acp_main_cus.c** allows for these CUStom Input & Output shortcut definitions to be extended beyond the validated number by modifying/extending structure definitions of this type: *extern const ACP_Unit cus_sigmaXY*

Note: The functions defined in audio_dc_cfg.c are specific to the components/configurations appropriate for the reference D10 Audio Platform. Equivalent replacement functions can be authored using these as a reference, if necessary.

Channel Maps

PASDK supports channel mapping both at the input (decoder) and also for the output (encoder).There is alpha code support provided to easily handle the various channel mapping options. Channel Map gives you a flexible mapping ability of input/output pin , in/out buffer and channel buffer.

Channel Mapping for Decoder

Here is the example of setting, as employed in the definition of **execPAIInHDMI,** which can receive 8 input audio channels on 4 McASP pins:

writeDECChannelMapFrom16(0,4,1,5,2,6,3,7,-3,-3,-3,-3,-3,-3,-3,-3)

- This function can map each input pin and input buffer.
- The relation between McASP pins and index is as follows:

0, ..., 7 is index of input buffer.

Audio samples are read in the order of index. (0->1->2->...->7)

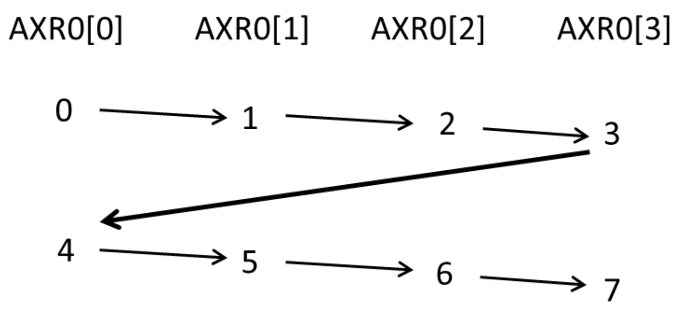


Figure 6: McASP Input Buffer

EDMA reads audio data from the McASP pins. The data is read sample by sample. The data is read starting from lower number McASP pin. First sample is read from AXR0[0]. The next sample is picked from AXR0[1] and so on. Thus, the EDMA picks up one sample from each McASP pin and loops back once it has read one sample from each of the McASP pins.

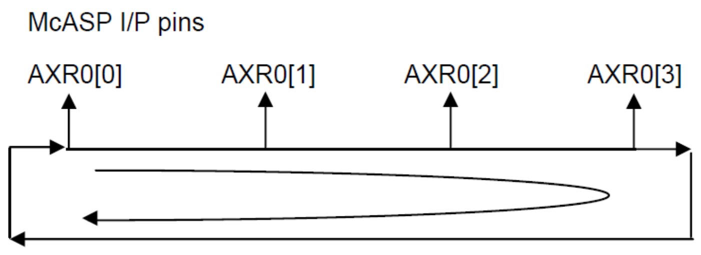


Figure 7: EDMA Reads from McASP Input

The audio channels are present on the input pins like shown below:

AXR0[0] = L/R AXR0[1] = SL/SR AXR0[2] = C/SW AXR0[3] = LB/RB

The samples read from the McASP pins are accumulated in data buffers. Thus input buffer is generated in the following order:

Figure 8: **Driver's view of Input Buffer**

The audio channels from the driver buffers need to be copied to the framework buffers. The DEC and the ASP algorithms read the data from the framework buffers.

The mapping of the channels is fixed in the framework buffers. Thus, the first buffer will always be for Left channel. Similarly second buffer will always be for Right channel.

The decoder needs to read the data **FROM** the input driver buffer and copy it **TO** the framework buffer:

LEFT	---	**READ FROM**	-->	0 (Input Buffer)	---	**WRITE TO**	-->	0 (FWK Buffer)
RIGHT	---	**READ FROM**	-->	4 (Input Buffer)	---	**WRITE TO**	-->	1 (FWK Buffer)
SURL	---	**READ FROM**	-->	1 (Input Buffer)	---	**WRITE TO**	-->	8 (FWK Buffer)
SURR	---	**READ FROM**	-->	5 (Input Buffer)	---	**WRITE TO**	-->	9 (FWK Buffer)
CNTR	---	**READ FROM**	-->	2 (Input Buffer)	---	**WRITE TO**	-->	2 (FWK Buffer)
SUBW	---	**READ FROM**	-->	6 (Input Buffer)	---	**WRITE TO**	-->	12 (FWK Buffer)
LBAK	---	**READ FROM**	-->	3 (Input Buffer)	---	**WRITE TO**	-->	10 (FWK Buffer)
RBAK	---	**READ FROM**	-->	7 (Input Buffer)	---	**WRITE TO**	-->	11 (FWK Buffer)

Figure 9: **DECChannelMap From & To**

The mapping between the input & the framework buffers, to accomplish the above, are specified in **atboot.c**, as thus:

writeDECChannelMapTo16(PAF_LEFT,PAF_RGHT,8,9,2,12,10,11,-3,-3,-3,-3,-3,-3,-3,-3)

Also, instead of reading the data from the input driver buffer, the data can be forced to be read as zero for any channel by specifying the buffer number as **–3** (or any negative integer).

Thus, **writeDECChannelMapFrom8(-1,-1,-1,-1,-1,-1,-1,-1)** - for instance - will cause the data to be read by the decoder as zero for all channels.

Channel Mapping for Encoder

EDMA writes audio data to the McASP pins. The data is written sample by sample. The data is written starting from lower number McASP pin.

Refer the above section to understand the channel mapping described for the Decoder. The mirror-image structure is applicable to the mapping that enables the Encoder to interface with the Output Buffer & the McASP pins.

The equivalent mapping functions employed on the output side are **writeENCChannelMapTo16 & writeENCChannelMapFrom16.**