SIGMASTUDIO FOR SHARC FRAMEWORK

ANALOG DEVICES, INC.

www.analog.com

KT-2252 (REV 2.0, SEPTEMBER-2014)

Table of Contents

List of Figures

List of Tables

Copyright, Disclaimer & Trademark Statements

Copyright Information

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

Analog Devices, the Analog Devices logo, SigmaStudio, Blackfin, SHARC, TigerSHARC, CrossCore, VisualDSP, VisualDSP++, EZ-KIT Lite, EZ-Extender and Collaborative are trademarks and/or registered trademarks "®" of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

Analog Devices' Trademarks and Service Marks may not be used without the express written consent of Analog Devices, such consent only to be provided in a separate written agreement signed by Analog Devices. Subject to the foregoing, such Trademarks and Service Marks must be reproduced according to ADI's Trademark Usage guidelines. Any licensee wishing to reproduce ADI's Trademarks and Service Marks must obtain and follow these guidelines for the specific marks at issue.

1 Introduction

SigmaStudio™ is a development environment from Analog Devices for graphically programming ADI's DSPs. SigmaStudio for SHARC includes an extensive set of algorithms to perform audio processing tasks such as filtering and mixing, as well as basic low-level DSP functions, optimized to run on the SHARC family of processors. SigmaStudio for SHARC also provides support for Analog Devices Software Modules such as the Dolby® Digital AC3 Decoder. SHARC Software Modules can be obtained separately along with their respective SigmaStudio Plug-Ins.

The environment also extends parameter export and filter coefficient generation support for a host microcontroller. Automation API support is provided to connect with many other tools, such as Python, .NET application, Matlab®, and LabVIEW. An easy-to-use graphical interface allows users to create custom filters, compressors and other audio-shaping algorithms to improve or change the characteristics of the audio. SigmaStudio for SHARC Algorithm Designer is provided to convert existing Software Modules or other SHARC libraries into SigmaStudio Plug-Ins. The environment is integrated with CrossCore® Embedded Studio.

1.1 Scope

This document is intended to assist Application or framework design engineers to configure the Default Application to meet custom SHARC Target needs. The document gives implementation and configuration details of different components of the SigmaStudio for SHARC Application.

1.2 Organisation of this Guide

Section 1 : This section contains the introduction.

Section 2 : This section gives an overview of the Application.

Section 3 : This section describes the framework parameters.

Section 4 : This section provides information related to audio CODEC.

Section 5 : This section provides information related to PCMx data format.

Section 6 : This section explains the System Register save/restore ASM macros used in Application.

Section 7 : This section gives the details of Symbol XML file.

Section 8 : This section contains details of the Application macros.

Section 9 : This section contains instructions for building the Application in non-VISA mode.

Section 10 : This section provides information on the interfaces which are used in the Target Framework.

1.3 Acronyms

ADC	Analog to Digital Converter
ADI	Analog Devices Inc.
API	Application Program Interface
DAC	Digital to Analog Converter
DAI	Digital Audio Interface
DMS	Document Management System
I2S	Integrated Interchip Sound
NaN	Not-a-Number
PCG	Precision Clock Generator
PCM	Pulse Code Modulation
S/PDIF	Sony/Philips Digital Interconnect Format
SPI	Serial Peripheral Interface
SPORT	Serial Port
SRC	Sample Rate Converter
SRU	Signal Routing Unit
TDM	Time Division Multiplexing

1.4 References

[1] SigmaStudio_for_SHARC_Users_Guide.pdf
Analog Devices Inc
[2] SigmaStudio_for_SHARC_AlgorithmDesigner.pdf
Analog Devices Inc
[3] SigmaStudio_for_SHARC_ReferenceGuide.pdf
Analog Devices, Inc.
[4] SHARC Audio EZ-Extender Manual,
Analog Devices Inc, Revision 1.1, August 2012
[5] ADSP-214xx SHARC Processor Hardware Reference
Analog Devices Inc, Revision 1.1, April 2013

1.5 Additional Information

For more information on the latest ADI processors, silicon errata, code examples, development tools, system services and devices drivers, technical support and any other additional information, please visit our website at www.analog.com/processors.

2 Overview

The Application described in this document is developed using the supplied SigmaStudio For SHARC Target APIs [3] and the supplied IO drivers. This flexibility allows the user to integrate the SigmaStudio Tuning feature to existing Applications. The sections below explain the configuration and other details about the Application. A quick-start guide to using the Default Application on the ADI EZ-KITs is described in [1]. The modified/created Application Loader File is used for booting the SHARC Target.

The Application Loader File ss_app_shxxx.ldr is loaded from the SigmaStudio Host machine through SPI. The Application Loader File has a SPI connectivity driver, which can receive the protocol packet from the SigmaStudio Host. The Application Loader File contains the code for reading input and playing output audio samples. Additionally, the Application also has code for setting the EZ-KIT LEDs for status indication.

The Application also contains the code to instantiate an empty placeholder for SSn code, data, parameters and tables that will be sent by the SigmaStudio Host. To reserve all the above resources for an SSn instance, the respective APIs are called. Advanced users can write their own Application to use the SPI connectivity and the SigmaStudio Tuning facility. Refer to Section 7 of [3] for more details.

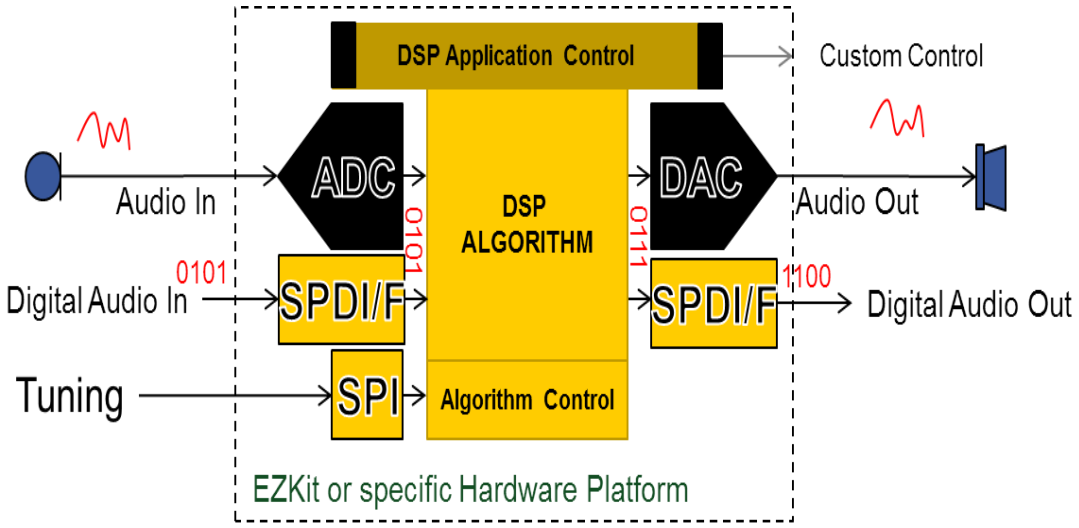


Figure 1: SigmaStudio for SHARC framework - System Overview

The Application framework supports the following features:

- Real-Time Audio Signal Processing and Tuning/Control

- Supports Analog and Digital Inputs/Outputs
- Default Control through SPI.
- Other controls such as UART are supported through the Application.
- Custom controls are supported through the Application

2.1 Audio IO

The Application framework can support the following audio input/outputs

- Analog audio in
- Digital audio in
- Analog audio out
- Digital audio out

2.2 Connectivity

Only SPI Connectivity is supported currently.

2.3 Processing and Control

The Application framework performs/controls/facilitates the following tasks.

- Booting
- Code downloading
- Signal Processing Algorithms
- Algorithm control
- Data Rate control
- Clocking
- Peripheral configurations

3 Framework Parameters

The Application can be rebuilt for different Application Sampling Rates, Application Block Sizes and SPORT buffer sizes. Refer to Annexure A for the impact of framework parameter changes on the SSn state and parameter memories.

3.1 Application Sampling Rate

The Application supports 48 kHz, 96 kHz and 192 kHz Application Sampling Rates in TDM mode. Only 48 kHz Application Sampling Rate is supported in I2S mode. Note that, TDM mode is enabled in the Default Application for ADSP-214xx SHARC Target and I2S mode is enabled for ADSP-213xx SHARC Target. To change the mode to I2S in the ADSP-214xx Application, refer to section 8 .

The default Application Sampling Rate is 48 kHz. To change the Application Sampling Rate to a different value, other than the default value in TDM mode, follow the instructions below:

1) Using the CrossCore Embedded Studio Examples Browser, filter examples by the SigmaStudio For SHARC product and select othe SigmaStudio for SHARC Demo Apps for one of the ADSP-214xx SHARC Targets, and open the project.
2) Open the project properties window. Select CrossCore SHARC C/C++ Compiler➔Preprocessor. Refer to Figure 2 which shows a snapshot of this for ADSP-21469 processor.
3) Edit the macro APP_SAMPLING_RATE=SAMPLING_RATE_48K in Preprocessor definitions to the desired Application Sampling Rate of 48 kHz, 96 kHz or 192 kHz using SAMPLING_RATE_48K, SAMPLING_RATE_96K or SAMPLING_RATE_192K macros respectively.

   For example, to change the Application Sampling Rate to 96 kHz edit the macro as shown below.

   APP_SAMPLING_RATE=SAMPLING_RATE_96K
4) Select Apply and then OK. Clean and rebuild the Application.

Note: The input and output Application Sampling Rates shall always be the same.

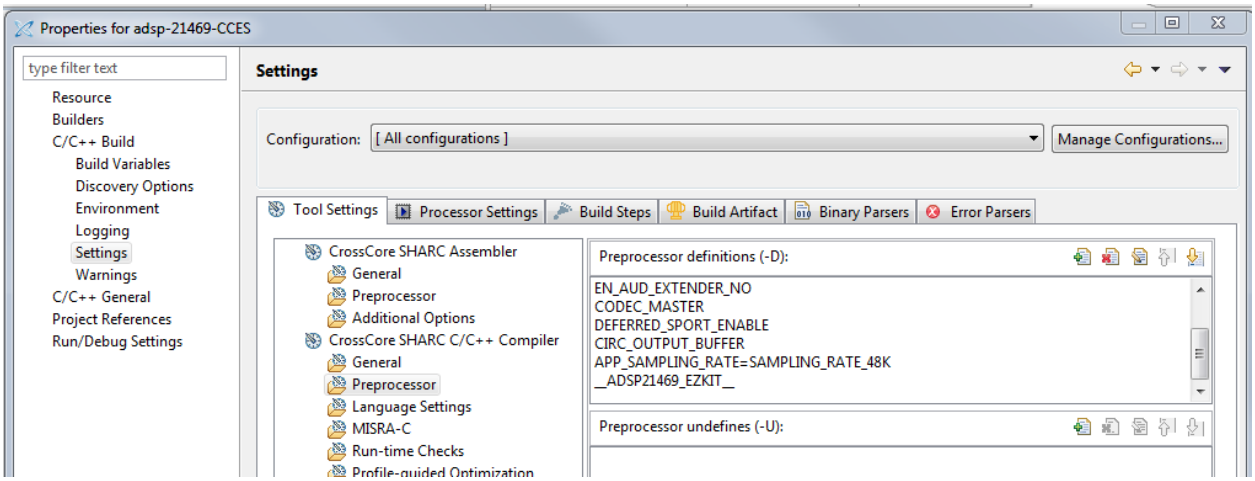


Figure 2: CCES Project Properties – Compiler Preprocessor

3.2 Application Block Size

The Application Block Size can be changed within the Application by changing the macro *PROCESSING_BLK_SIZE* in the file *app.h*. Note that the macro *PROCESSING_BLK_SIZE* is analogous to the macro *NUM_SAMPLES*, used in pre 2.1.0 releases. The Schematic Block Size must always be lesser than or equal to the *PROCESSING_BLK_SIZE*.

3.3 Number of processing buffers

The number of the processing buffers in the Application is denoted by the macro *NUMBER_PROCESSING_BUFFERS*. Refer to Annexure A for more details.

3.4 SPORT Buffer Size

The SPORT buffer size is the size in samples/channel, at which the SPORT DMA transfers data packets to the SHARC internal/external memory from the external world or vice versa.

The SPORT buffer size can be changed within the Application by changing the macro *SPORT_BUFFER_SIZE* in the file *app.h*.

This feature gives users the flexibility to control audio I/O interrupt durations in a more granular manner.

3.5 Input To Output Rate and Output To Input Rate ratios

Some processing Modules may support different rates at the input and output. Such Modules require different amounts of buffering at input and output for optimal utilization of system memory. The following two macros allow different amounts of buffering at the input and output.

1. INPUT_TO_OUTPUT_RATE – This macro can be used to specify the input to output rate ratio for the Application. This macro is used to scale up the memory allocation for input buffering of the data acquired through SPORTS, to account for higher input rate compared to output. The default value of this macro is 1.
2. OUTPUT_TO_INPUT_RATE - This macro can be used to specify the output to input rate ratio for the Application. This macro is used to scale up the memory allocation for output buffering of the data rendered through SPORTS, to account for higher output rate compared to input. The default value of this macro is 1.

Note that these macros must take integer values and at least one of these macros must be set to 1. The maximum value for these macros is limited by the available system memory.

3.6 SPORT Buffer Count (Number of SPORT Buffers)

To meet real-time performance, the Application must have more than one SPORT buffer so that while one or more buffers are being filled, processing can happen on one or more different sets of buffers.

The number of input SPORT buffers is denoted by macro *NUM_INPUT_SPORT_BUFFERS* and the number of output SPORT buffers is denoted by macro *NUM_OUTPUT_SPORT_BUFFERS* in *app.h*. These macros are calculated automatically in the Application as in the expressions below.

NUM_INPUT_SPORT_BUFFERS =
*(((NUMBER_PROCESSING_BUFFERS*PROCESSING_BLK_SIZE)/SPORT_BUFFER_SIZE)*
** INPUT_TO_OUTPUT_RATE)*

NUM_OUTPUT_SPORT_BUFFERS =
*(((NUMBER_PROCESSING_BUFFERS*PROCESSING_BLK_SIZE)/SPORT_BUFFER_SIZE)*
** OUTPUT_TO_INPUT_RATE)*

3.7 Configuring Buffer Sizes

There are some restrictions on choosing values for SPORT_BUFFER_SIZE and PROCESSING_BLK_SIZE in the Application as listed below.

1. SPORT_BUFFER_SIZE must be less than or equal to the PROCESSING_BLK_SIZE.
2. The product of SPORT_BUFFER_SIZE and NUM_INPUT_SPORT_BUFFERS must be a multiple of PROCESSING_BLK_SIZE. Similarly the product of SPORT_BUFFER_SIZE and NUM_OUTPUT_SPORT_BUFFERS must be a multiple of PROCESSING_BLK_SIZE. This multiple is defined by the macro "NUMBER_PROCESSING_BUFFERS", in the file *app.h*.

Figure 3 below, illustrates the SPORT_BUFFER_SIZE, NUM_INPUT_SPORT_BUFFERS, NUM_OUTPUT_SPORT_BUFFERS, PROCESSING_BLK_SIZE and NUMBER_PROCESSING_BUFFERS for the case when INPUT_TO_OUTPUT_RATE = 1 and OUTPUT_TO_INPUT_RATE = 1.

Note that when INPUT_TO_OUTPUT_RATE = 1 and OUTPUT_TO_INPUT_RATE = 1, NUM_INPUT_SPORT_BUFFERS = NUM_OUTPUT_SPORT_BUFFERS.

Let us denote,

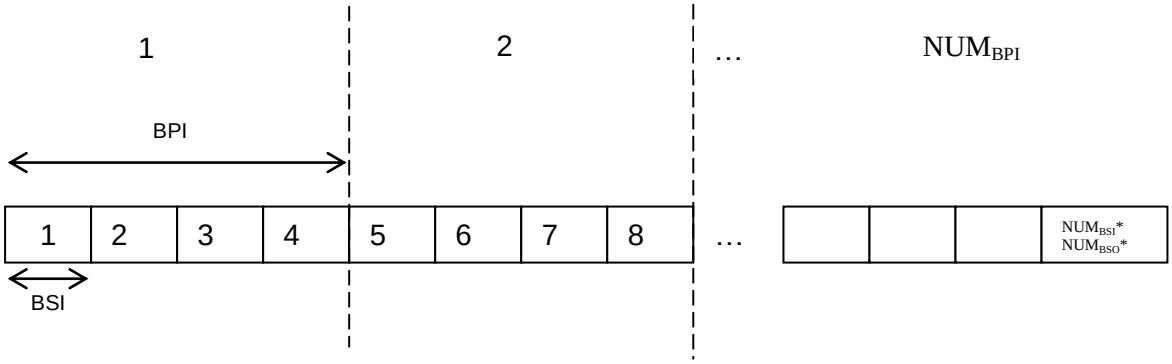
SPORT_BUFFER_SIZE as BSI

PROCESSING_BLK_SIZE as BPI

NUMBER_PROCESSING_BUFFERS as NUM_{BPI}

NUM_INPUT_SPORT_BUFFERS as NUM_{BSI}

NUM_OUTPUT_SPORT_BUFFERS as NUM_{BSO}



*Assumed that INPUT_TO_OUTPUT_RATE = OUTPUT_TO_INPUT_RATE = 1

Figure 3: Illustration of Application buffering terms

Based on the average and peak load of the Schematic, the "PROCESSING_BLK_SIZE", "SPORT_BUFFER_SIZE" and "NUMBER_PROCESSING_BUFFERS" must be chosen accordingly. The values for macros INPUT_TO_OUTPUT_RATE and OUTPUT_TO_INPUT_RATE must be chosen based on the input and output sampling rates.

Increasing the total input buffering (equal to "NUMBER_PROCESSING_BUFFERS * PROCESSING_BLK_SIZE * INPUT_TO_OUTPUT_RATE") or total output buffering (equal to "NUMBER_PROCESSING_BUFFERS * PROCESSING_BLK_SIZE * OUTPUT_TO_INPUT_RATE") ensures that the input buffer does not overflow or output buffer does not underflow while processing blocks of data that consume high MIPS. The table below lists some combinations for these macros. The table assumes INPUT_TO_OUTPUT_RATE = 1 and OUTPUT_TO_INPUT_RATE = 1.

PROCESSING_ BLK_SIZE	SPORT_BUFFER _SIZE	NUMBER_ PROCESSING_ BUFFERS	NUM_INPUT_ SPORT_BUFFERS (calculated automatically in the Application)	NUM_OUTPUT_ SPORT_BUFFERS (calculated automatically in the Application)
64	16	3	12	12
256	16	3	48	48
128	32	4	16	16

Table 1: Example Application macro combinations for buffer management

3.8 Relationship between Buffer Size and peak MIPS of a Module

Let:

BPI = PROCESSING_BLK_SIZE

BSI = SPORT_BUFFER_SIZE (SPORT buffer size)

NUM_{BPI} = NUMBER_PROCESSING_BUFFERS

NUM_{BSI} = NUM_INPUT_SPORT_BUFFERS

NUM_{BSO} = NUM_OUTPUT_SPORT_BUFFERS

RAT_{IO} = INPUT_TO_OUTPUT_RATE

RAT_{OI} = OUTPUT_TO_INPUT_RATE.

BMI = Module input buffer size

$MIPS_{MAX}$= Maximum available MIPS for the chosen SHARC Target.

$MIPS_{PM}$ = Peak MIPS of the Module or Schematic.

BUF_{Min} = Minimum buffering required in the Application before data processing.

BUF_{Tot} = Total input buffer required.

Then:

$MIPS_{BUFMin}$ = Max load the system can handle by buffering BUF_{Min} amount of data =

$$(BUF_{Min}/BMI)* MIPS_{MAX}$$

Peak MIPS of Module needs to be less than the max load that can be handled for given BUF_{Min}, hence

$$MIPS_{PM} < MIPS_{BUFMin}$$

$$MIPS_{PM} < (BUF_{Min}/BMI)* MIPS_{MAX}$$

Hence, minimum buffering required before the data can be processed is,

$$BUF_{Min} > (MIPS_{PM}* BMI) / MIPS_{MAX}$$

Assuming triple buffering, the total input buffer requirement is

$$BUF_{Tot} = BUF_{Min}*3$$

Based on the value of $BUF_{Tot,}$ fix the values of BPI and BSI. The NUM_{BPI} is calculated as

$$NUM_{BPI} = BUF_{Tot} /BPI$$

NUM_{BSI} gets calculated automatically as

$$NUM_{BSI} = (NUM_{BPI}*BPI)/BSI)* RAT_{IO}$$

NUM_{BSO} gets calculated automatically as

$$NUM_{BSO} = (NUM_{BPI}*BPI)/BSI)* RAT_{OI}$$

Example:

Consider an example SHARC ADSP-21469 Module having a peak MIPS of 50 and Module buffer size of 1536 samples. $MIPS_{MAX}$ for ADSP-21469 SHARC Target is 400 MIPS.

To handle a peak load of 50 MIPS, the minimum buffering required before calling process is

$BUF_{Min} >= (50*1536)/400$

$BUF_{Min} >= 192$

Let $BUF_{Min} = 192$

For triple buffering of input,

$BUF_{Tot} = BUF_{Min}*3 = 192*3 = 576$

Let's fix the value of the BPI to be 64 and BSI to be 16.

$NUM_{BPI} = BUF_{Tot}/BPI = 576/64 = 9$

Let the input rate be 4 times the rate of the output.

i.e. $RAT_{IO} = 4$ and $RAT_{OI} = 1$

$NUM_{BSI} = ((NUM_{BPI}*BPI)/BSI)*RAT_{IO} = ((9*64)/16)*4 = 144$ (gets calculated automatically in the Application)

$NUM_{BSO} = ((NUM_{BPI}*BPI)/BSI)*RAT_{OI} = ((9*64)/16)*1 = 36$ (gets calculated automatically in the Application)

Hence the Module requires 144 input SPORT buffers of size 16 samples each and 36 output SPORT buffers of size 16 samples each. The Application Block Size BPI is fixed to 64 samples in this case.

4 Audio data input and output modes

4.1 Application Input and Output buffer pointers

The Application must supply data input and output buffer pointers to the SigmaStudio for SHARC Target Library through the *adi_ss_schematic_process()* API. Refer to section 7.3.6 of [3] for more information on this API. This section explains how the input and output buffer pointers must be initialized from the Application.

The input buffer pointers for the audio modes namely – Analog-In, Digital-In, Digital-Out alone and Analog/Digital Co-Existence are shown in Figure 4.

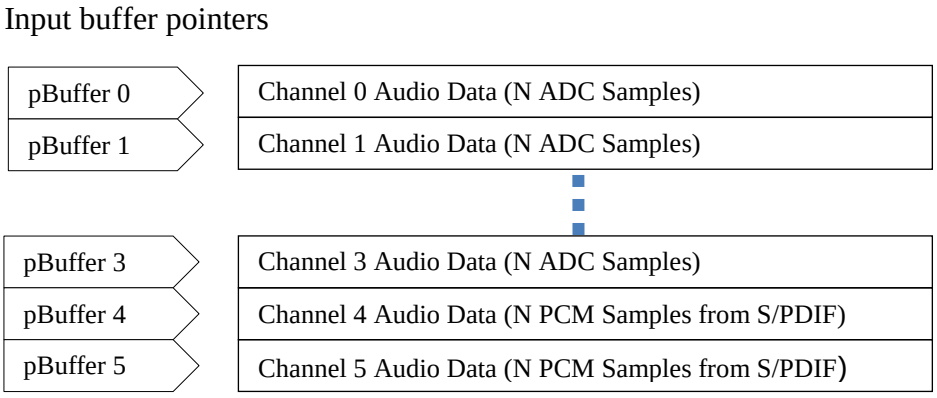


Figure 4: Input buffer pointers

The input buffer pointer for the Analog/Digital Co-existence (Digital Clock) mode is shown in Figure 5.

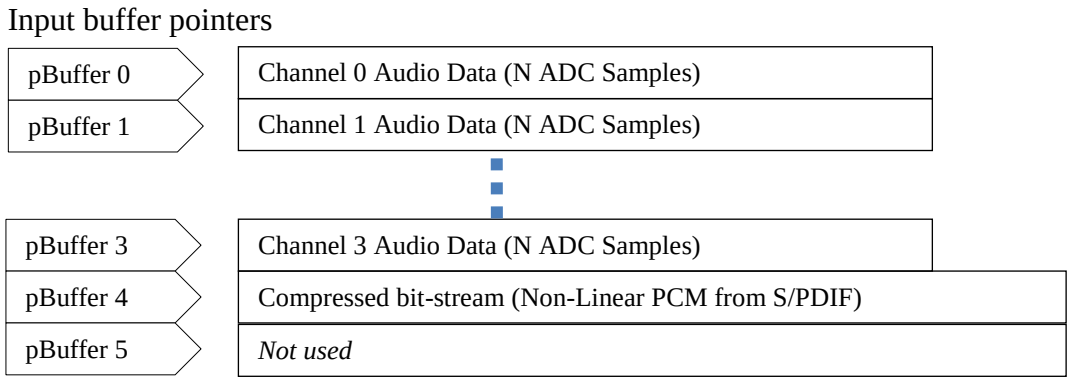


Figure 5: Input buffer pointer for Analog\Digital Co-existence (Digital Clock) mode

The output buffer pointers are shown in Figure 6.

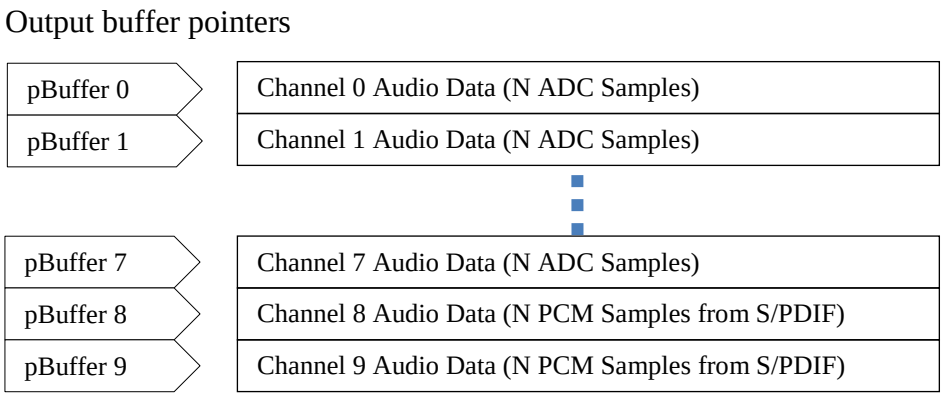


Figure 6: Output buffer pointers

4.2 I2S mode of operation

In I2S mode, a SPORT channel can carry only a single stereo audio channel. The clocking requirement is for the transmission of a stereo channel only. For transmitting/receiving multiple pairs of channels, multiple SPORTs are required to be used. Thus in the I2S mode, several SPORTs are required for data transmission and reception for various channel pairs but at lower clock requirements.

The block diagram in Figure 7 illustrates the I2S mode of operation with the *Input-Output Mode* as "Analog/Digital Co-existence".

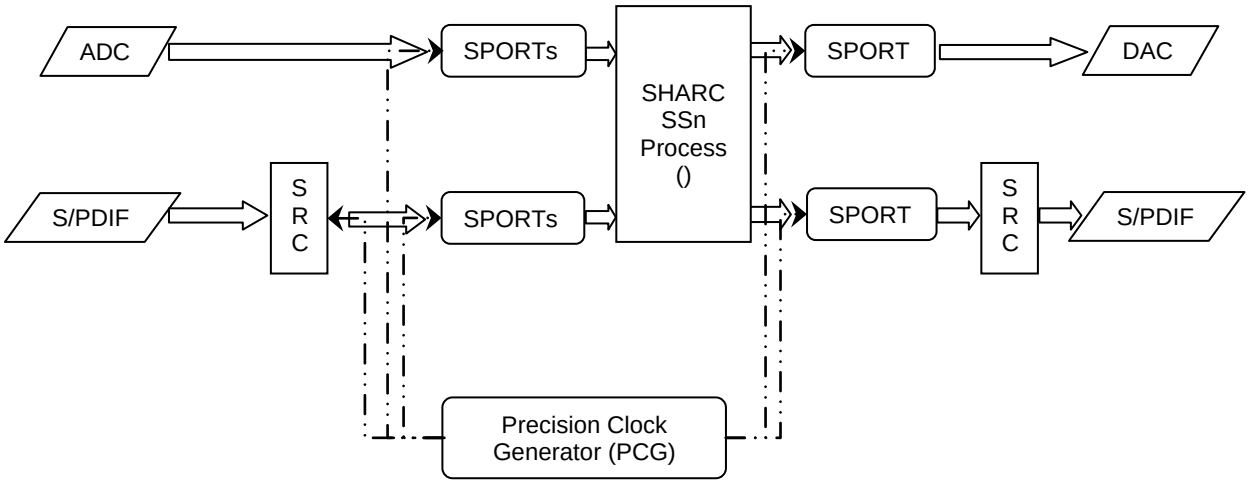


Figure 7: I2S mode of operation for Analog/Digital Co-existence

The block diagram in Figure 8 illustrates the I2S mode of operation with the *Input-Output Mode* as "Analog/Digital Co-existence (Digital Clock)".

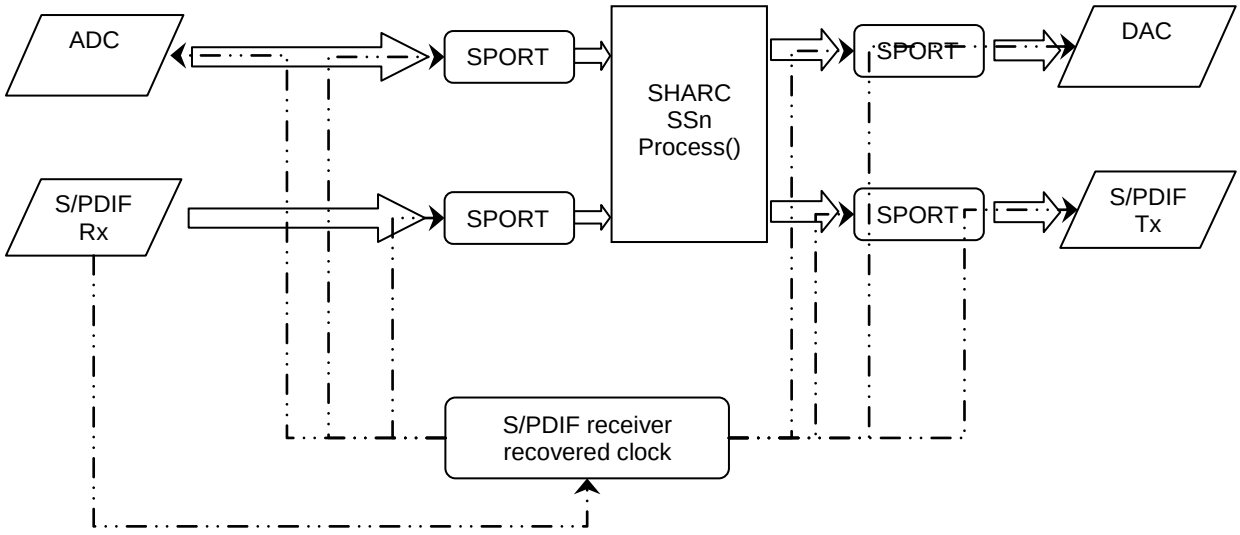


Figure 8: I2S mode of operation for Analog/Digital Co-existence (Digital Clock)

4.3 TDM mode of operation for ADSP-214xx SHARC Targets

The ADSP-214xx SHARC Targets support TDM mode of SPORT operation. In TDM mode, a SPORT channel can carry up to eight stereo channels. Note that the SPORTS have to be clocked at a higher rate in TDM mode of operation.

By configuring the SPORTs in TDM mode, the number of SPORTs required for data input and output can be reduced which in turn reduces the DAI pin count required for the data I/O. Thus, configuring the SPORTs in TDM mode allows for more input and output channels to be supported from the Application.

The bit clock and the frame sync clock required for the SPORTs may be derived either from the audio CODEC on the EZ-KIT or may be internally generated within the SHARC using the PCG. For TDM mode of operation the CODEC is also required to be configured for TDM mode.

If the CODEC clock and frame syncs are used for configuring the SPORTs, then the CODEC is in the master mode. That is, the CODEC is the master and the SHARC is the slave. If the clocks for the SPORTs and CODEC are internally generated within the SHARC using the PCG, then the SHARC is the master and the CODEC is the slave. The CODEC in master mode is described below.

4.3.1 CODEC in master mode

With the CODEC as master, the bit clock and frame sync are generated by the audio CODEC on-board the EZ-KIT. The SRU is configured within the Application such that the bit clock and frame syncs from the CODEC are routed to the SPORTs.

PCG A is used to divide down the TDM clock from the CODEC. This clock from PCG A is used for rate conversion of S/PDIF input data (from S/PDIF receiver) from S/PDIF rate to the CODEC rate using SRC 0. SRC 1 is used for rate conversion of S/PDIF output data (to S/PDIF transmitter) from CODEC rate back to S/PDIF rate. Note that if S/PDIF input is not plugged-in, the clock from PCG A is directly used to clock S/PDIF transmitter (Alt PCG CLK and Alt PCG Fs in Figure 9).

Figure 9 below illustrates the block diagram for the TDM mode of operation with the CODEC as the master.

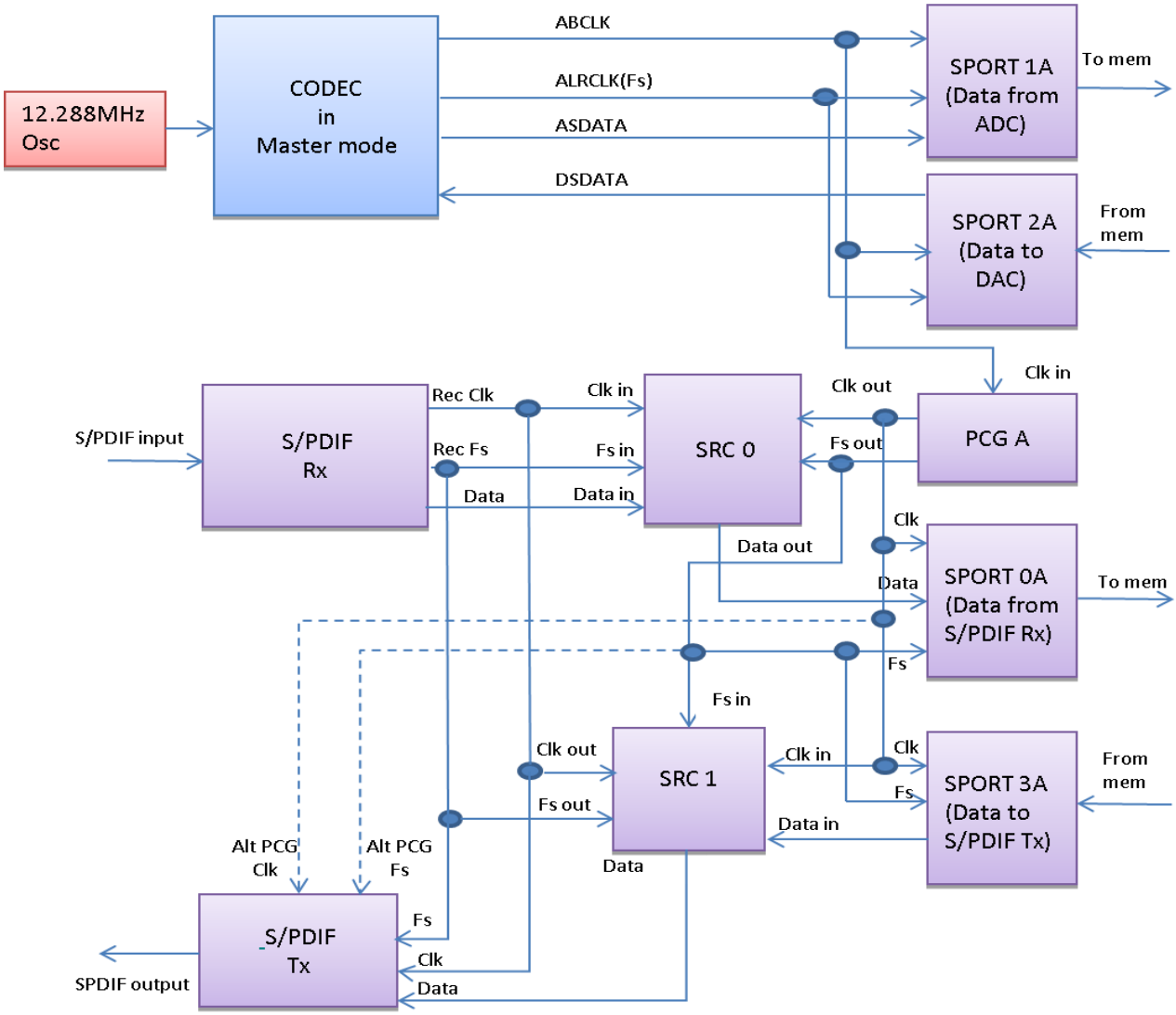


Figure 9: Block diagram depicting the TDM master mode of the CODEC

The DAI pins used on the SHARC are described in the table below.

DAI pin number	Pin direction	Functionality
DAI pin 7	Input	Bit clock (ABCLK)
DAI pin 8	Input	Frame sync (ALRCLK)
DAI pin 5	Input	Data from ADC to SPORT 1 channel A (ASDATA)
DAI pin 12	Output	Data to DAC from SPORT 2 channel A (DSDATA)
DAI pin 10	Output	Data to DAC from SPORT 2 channel B (for 192kHz Application Sampling Rate only)
DAI pin 18	Input	S/PDIF digital input data
DAI pin 1	Output	S/PDIF digital output data

Table 2: SHARC DAI pin usage for CODEC in master mode

5 PCMx data format

PCMx is a generic data format which contains a header block along with the data block. The header provides information about the data block. The header information includes the data start offset, the data size, Application Sampling Rate, PCMx type etc. Thus this format can be used to carry any kind of data and it is left to the user for data interpretation.

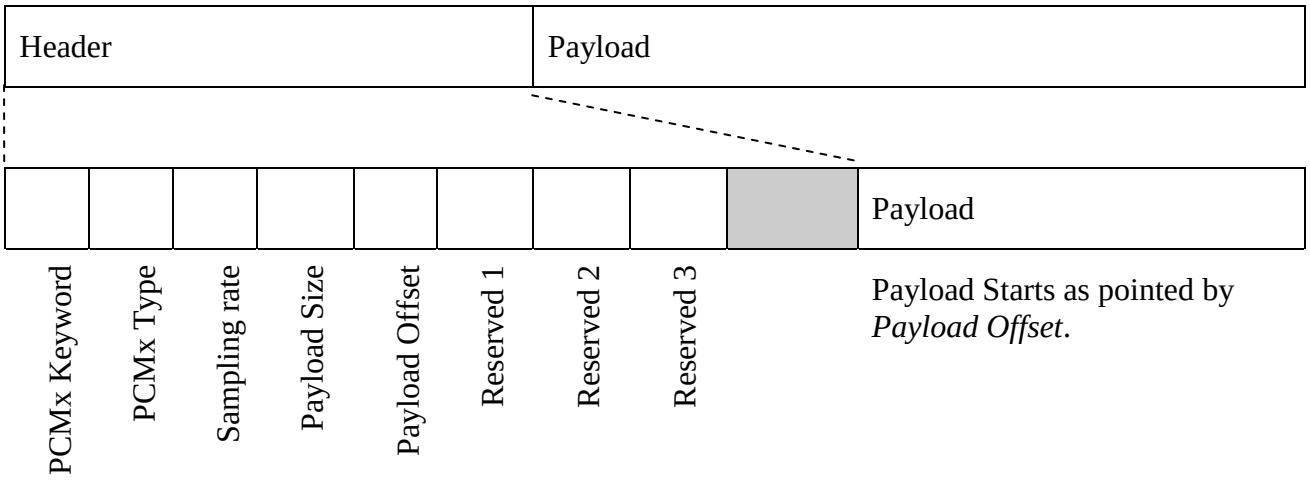


Figure 10: PCMx Data format

The PCMx header is listed below:

```
typedef struct _PCMxHeaderInfo
{
    int     nDataType;       /* Compulsory PCMx id SS_STREAM_DATATYPE_PCMx */
    int     nPCMxType;       /* PCMx type                                  */
    int     nSamplingRate;   /* Sampling rate                              */
    int     nPayloadSize;    /* Payload size                               */
    int     nPayloadOffset;  /* Offset of the payload from the start */
    tPCMxReserved oPCMxReserved;
} PCMxHeaderInfo;
```

The fields in the PCMxHeaderInfo structure are explained below:

nDataType: This field holds a compulsory value indicating that the data is of PCMx type. This value is defined by a macro SS_STREAM_DATATYPE_PCMx.

nPCMxType: This field indicates the type of PCMx data being carried. The user can group different kinds of PCMx data and assign a type to it. Users can then interpret the different kinds of PCMx data based on this field.

nSamplingRate: This field indicates the Application Sampling Rate of the data being carried in PCMx format.

nPayloadSize: This field indicates the size of the data being carried.

nPayloadOffset: This field indicates the start of the payload data within the PCMx buffer. This field also includes the size of the PCMx header.

oPCMxReserved: This is an instance of a union of structures which has 3 integers as in the listing below.

```
typedef struct _tPCMxDefault
{
   int32_t     nReserved1;             /* Reserved field 1 */
   int32_t     nReserved2;             /* Reserved field 2 */
   int32_t     nReserved3;             /* Reserved field 3 */
}tPCMxDefault;

typedef union tPCMxReserved
{
    tPCMxDefault oPCMxDefault;
}tPCMxReserved;
```

5.1 Application changes for PCMx data

The Application has to be modified to carry PCMx data. The provided Default Application supports PCMx input and output. Two Application macros control PCMx input and PCMx output data support. These macros are explained below:

1. ENABLE_PCMX_IN: This macro enables the Application support for the PCMx input. The memory sizes of the input buffers must be modified to hold the input PCMx content.

```
#pragma section("ss_fw_block2_data")
volatile adi_ss_sample_t
Block_In[NUM_INPUT_CHANNELS*PROCESSING_BLK_SIZE_PCMX_IN];

#pragma section("ss_fw_block2_data")
volatile adi_ss_bitstream_t
Block_In_Spdif[NUM_INPUT_SPDIF_CHANNELS*PROCESSING_BLK_SIZE_PCMX_IN];
```

The Application must also append the PCMx header to the data fetched from the SPORT into the input buffers before calling the *adi_ss_schematic_process()* API as shown in the code snippet below.

```
pPCMxHeaderInfo = (PCMxHeaderInfo*)&pInBuff[0];
pPCMxHeaderInfo->nDataType      = SS_STREAM_DATATYPE_PCMx;
pPCMxHeaderInfo->nPCMxType      = SS_STREAM_DATATYPE_PCMx_MULTI_RATE;
pPCMxHeaderInfo->nSamplingRate  = pAppInfo->nInputSamplingFreq;
pPCMxHeaderInfo->nPayloadSize    = PROCESSING_BLK_SIZE;
pPCMxHeaderInfo->nPayloadOffset = PROCESSING_BLK_SIZE;
```

A sample Schematic with PCMx input connections is shown in Figure 11.

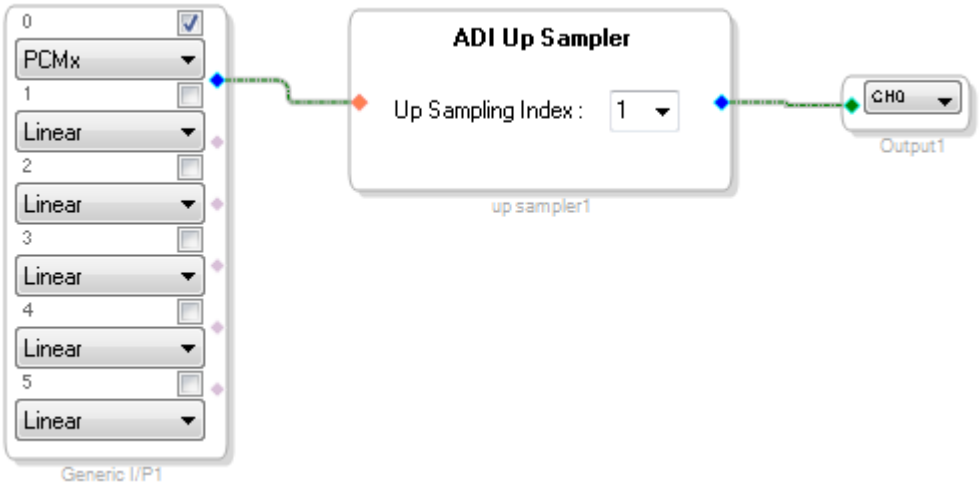


Figure 11: Sample Schematic with PCMx input

2. ENABLE_PCMX_OUT: This macro enables the Application support for the PCMx output. The memory sizes of output buffers must be modified to hold the output PCMx content.

```
#pragma section("ss_fw_block2_data")
volatile adi_ss_sample_t
Block_Out[NUM_OUTPUT_CHANNELS*PROCESSING_BLK_SIZE_PCMX_OUT];
```

The Application must ensure that the output data from the output buffers is copied into the SPORT buffers from the correct offset indicated by the PCMx header. The number of samples to be copied into the SPORT buffers must also be retrieved from the PCMx header as shown in the code snippet below.

```
pPCMxHeaderInfo =
(PCMxHeaderInfo*)&Block_Out[nSlot*nOutChPerSlot*PROCESSING_BLK_SIZE_PCMX_OUT+(nC
h*PROCESSING_BLK_SIZE_PCMX_OUT)];

nPayLoadSize = pPCMxHeaderInfo->nPayloadSize;

/* This will depend on the type of application. */
nPayLoadOffset = pPCMxHeaderInfo->nPayloadOffset;
```

A sample Schematic with PCMx output connections is shown in Figure 12.

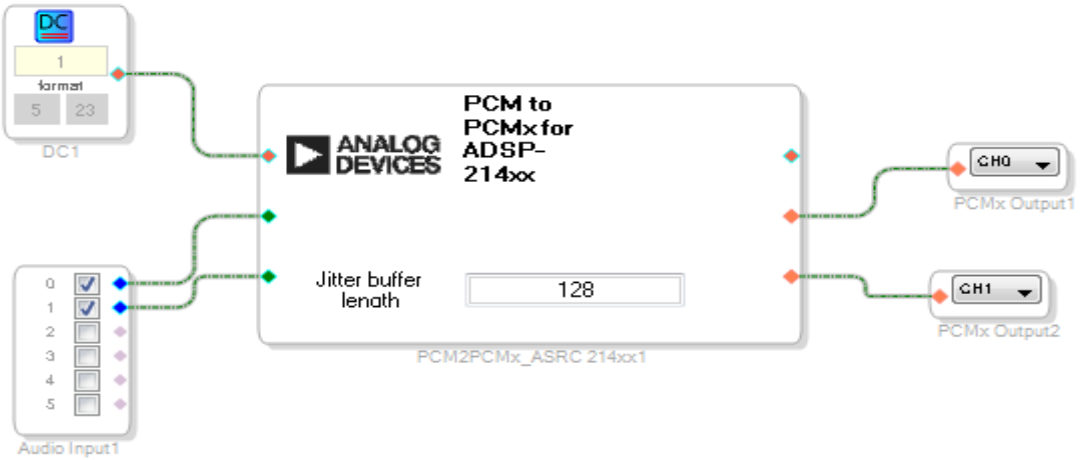


Figure 12: Sample Schematic with PCMx output

5.1.1 Schematic with PCMx type when Application Block Size (PROCESSING_BLK_SIZE) and Schematic Block Size are different

Figure 13 shows PCMx data propagation when Application Block Size and Schematic Block Size are different using an example case. It assumes the Application Block Size is 64 and the Schematic Block Size is 16. The Block of samples propagates as follows:

A: The Application buffer with PCMx header as defined in 5 . This has to be implemented by the framework programmer.
B: SigmaStudio framework feeds samples through the PCMx Input Cell. This is internal to SigmaStudio and performed by SigmaStudio.
C: PCMx Samples for each call are as follows:

Call 1: PCMx Packet as in A is passed on.
Call 2-4: PCMx Packet as in A is passed on, however the module must not interpret it.

D: Samples as that of C.
E: 16 PCM samples for every call to the SSn framework. The PCMx2PCM module must buffer the data and output 16 PCM samples at a time for every call.
F: 64 PCM samples from SSn framework.
G: PCMx Samples for each call are as follows:

Call 1: PCMx Packet as in D is passed from SSn framework.
Call 2-4: PCMx Packet as in D is passed on, however the PCMx output module does not interpret it.

H: 64 PCM samples to the Application.
I: PCMx samples to the Application with PCMx header.

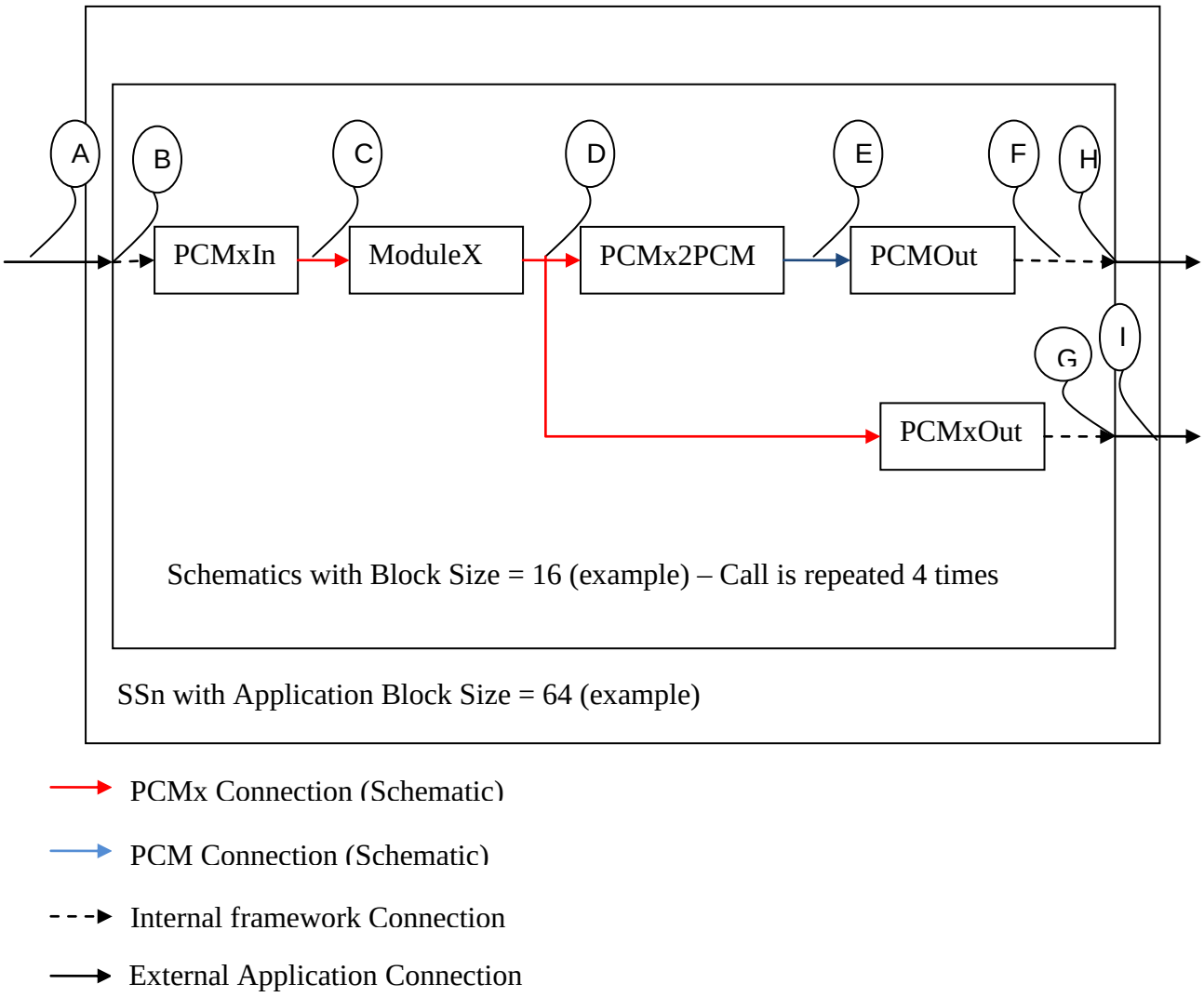


Figure 13: PCMx data propagation

Note that the modules such as "ModuleX" which accept PCMx input must be aware of the Application Block Size so that they do not process the incoming data from the second call onwards to the SSn framework. The Application Block Size can be communicated to the module using the "Reserved" fields of the PCMx header.

6 ASM Macros for Saving and Restoring System Registers

Sample code to save and restore the registers is given below. These macros can be used for cases when the entire context is not saved by the routine or thread calling the library function. The same sample code is available in the file *"<path>\Target\Demo\src\SPORTisr.c"*.

```
int aRegStorage[5][16];
void ISR_ProcessCall(void)
{

asm("%0 = m0;":  "=d"(aRegStorage[1][0]) : :);
.
asm("%0 = m15;": "=d"(aRegStorage[1][15]) : :);

asm("%0 = b0;":  "=d"(aRegStorage[2][0]) : :);
.
/* No need to save b6 and b7*/
.
asm("%0 = b15;": "=d"(aRegStorage[2][15]) : :);

asm("%0 = l0;":  "=d"(aRegStorage[3][0]) : :);
.
/* No need to save l6 and l7*/
.
asm("%0 = l15;": "=d"(aRegStorage[3][15]) : :);

/* Clear L register, critical section */
asm("l0 = 0;");
asm("l15 = 0;");

asm("%0 = i0;":  "=d"(aRegStorage[0][0]) : :);
.
/* No need to save i6 and i7 */
.
asm("%0 = i15;": "=d"(aRegStorage[0][15]) : :);

/* Function Call; eSSnReturnB  = adi_ss_schematic_process(hSSn, nBlkSize, Block_In, Block_Out,
oProps); */
/* Restore all saved registers*/
asm("m0 = %0;":   :"d"(aRegStorage[1][0]):);
.
.
asm("m15 = %0;":   :"d"(aRegStorage[1][15]):);


asm("b0 = %0;":   :"d"(aRegStorage[2][0]):);
.
.
asm("b15 = %0;":   :"d"(aRegStorage[2][15]):);

asm("l0 = %0;":   :"d"(aRegStorage[3][0]):);
.
.
asm("l15 = %0;":   :"d"(aRegStorage[3][15]):);

asm("i0 = %0;":   :"d"(aRegStorage[0][0]):);
.
.
asm("i15 = %0;":   :"d"(aRegStorage[0][15]):);
}
```

7 Symbol XML File

The Symbol XML file is used to load the address of buffers and function pointers used in the SHARC Target Application to SigmaStudio. These buffer and address pointers can be used by the Plug-Ins to access the functions and buffers that are part of the Application. Refer to section 8 of [2] for more details on accessing external symbols from the Plug-In. The Symbol XML file can also be used to select the Loader File to be used for booting the SHARC Target. The following elements are included in the XML.

- Name and path of Loader File and boot argument.
- Target DSP for the Application
- Symbols and respective address in the SHARC Target Application.

```
<?xml version="1.0" standalone="yes"?>
<SS4SH name="ss4sh_ic_xml" description="Analog Devices SigmaStudio for SHARC IC Configuration"
version="2.0.0.0">
	<ic name="ADSP-21489">
		<ldr name="ss_app_sh489.ldr" path=".\" arg="2"/>
		<symbol type="extSymbol" extSymbolName="Scale" module="myscale" address="0xbb1a4"
id="none" />
	</ic>
</SS4SH>
```

7.1 XML Elements

7.1.1 <SS4SH>

The `SS4SH` is the outermost element of the XML file. This can appear only once in the XML file and should be exactly as shown in the example above.

The attributes of `SS4SH` element are:

- `name`
  String indicating the name of the XML file. This should always be "ss4sh_ic_xml". If the name is different, SigmaStudio treats this as an invalid XML file.
- `description`
  Description of the xml of type string.
- `version`
  Version of the SigmaStudio for SHARC package in a.b.c.d format.

The contents of the `SS4SH` element are:

- `ic`
  Refer to section 7.1.2 for details. One `SS4SH` element can have only one `ic` element.

7.1.2 <ic>

The `ic` element defines a SHARC Target. There can be only one `ic` element inside an XML.

The attributes of `ic` element are:

- `name`
  String indicating the SHARC Target processor.

The contents of the `ic` element are:

- `ldr`
  Refer to section 7.1.3 for details. One `ic` element can have only one `ldr` elements. It is not mandatory to have the `ldr` element.

- `symbol`
  Refer to section 7.1.4 for details. One `module` element can have any number of `symbol` elements.

7.1.3 <ldr>

The `ldr` element defines the Loader File to be used for booting the SHARC Target. One `ic` element can have only one `ldr` element.

The attributes of `ldr` element are:

- `name`
  String indicating the name of the Loader File.

- `path`
  String indicating the folder path of the Loader File. The path can either be an absolute location or a location relative to the path location of the XML file. Use back-slash '\' to separate folders in the path.

- `arg`
  Boot option. Following are the options.
  - "2": Analog-In
  - "3": Digital-In
  - "4": Digital-Out alone
  - "5": Analog\Digital Co-existence
  - 10: Analog\Digital Co-existence (Digital Clock)

The `ldr` element cannot have any contents.

7.1.4 <symbol>

The `symbol` element defines the symbol in the SHARC Target Application or the Loader File. There can be any number of `symbol` elements inside an `ic` element.

The attributes of `symbol` element are:

- `type`
  Indicates the type of the symbol. Should always be "extSymbol".
- `extSymbolName`
  String indicating the name of the symbol.
- `module`
  Indicates the module where the symbol is used. Currently this field is not used.
- `address`
  SHARC Target address of the symbol in hexadecimal.
- `id`
  This field is currently not used.

The `symbol` element cannot have any contents.

8 Application preprocessor macros

This section describes the macros available in the Application. The following macros are available, that can be configured in the Application.

- DO_CYCLE_COUNTS: This macro enables MIPS calculation. This macro is enabled in the Default Application. To disable this macro, remove it from the compiler preprocessor options and rebuild the Application.
- I2S_MODE: This macro enables I2S mode of SPORT and CODEC operation. ADSP-214xx SHARC Targets support TDM mode of operation in which multiple input and output channels are obtained through single input and output pins of the SHARC Target. By default, I2S mode is enabled for ADSP-213xx SHARC Targets and TDM mode is enabled for ADSP-214xx SHARC Targets. TDM mode is not supported for ADSP-213xx SHARC Targets. To enable I2S mode for ADSP-214xx SHARC Targets, add the macro "I2S_MODE" in the compiler preprocessor options and rebuild the Application.
- CODEC_MASTER: Defining this macro configures the Application such that the CODEC is the master and SHARC Target is the slave for the ADSP-214xx SHARC Targets. This macro must always be defined in both I2S and TDM modes of operation of the CODEC and SPORT. Define this macro in compiler preprocessor options to enable this feature.
- ENABLE_PCMX_IN: Macro for enabling the PCMx input functionality. By default, this macro is disabled. Define this macro in compiler preprocessor options to enable this feature.
- ENABLE_PCMX_OUT: Macro for enabling the PCMx output functionality. By default, this macro is disabled. Define this macro in compiler preprocessor options to enable this feature.
- CIRC_OUTPUT_BUFFER: This macro enables the circular buffering of the SPORT output buffers. This macro is enabled by default for ADSP-214xx SHARC Target and can be used only in TDM mode. Define this macro in compiler preprocessor options to enable this feature.
- DEFERRED_SPORT_ENABLE: This macro enables the input and output SPORTs at different instances in time for the output data pre-rolling. By default, this macro is enabled for ADSP-214xx SHARC Targets. Define this macro in compiler preprocessor options to enable this feature.
- APP_SAMPLING_RATE: This macro is used to set the desired Application Sampling Rate of 48 kHz, 96 kHz or 192 kHz using SAMPLING_RATE_48K, SAMPLING_RATE_96K or SAMPLING_RATE_192K macros respectively. Refer to section 3.1 to change the Application Sampling Rate to a different value other than the default value in TDM mode for ADSP-214xx Application.

9 ADSP-214xx demo Applications with non-VISA mode

The Default Application for ADSP-214xx SHARC Targets is built for the short word code (SWC) i.e. VISA mode. The steps below show how to build the Application for the normal word code (non-VISA mode).

1) Open the ADSP-214xx demo Application with CrossCore Embedded Studio.
2) Select the **Manage configurations for the current project -> ReleaseNWC** configuration for the non-VISA release mode or **Manage configurations for the current project -> DebugNWC** configuration for non-VISA debug mode.
3) Rebuild the Application and use the newly generated Loader File.

10 Target Framework Interfaces

This section provides information on the interfaces which are used in the Target Framework.

10.1 Framework Interface functions

10.1.1 Target Framework Initialization

This section describes the interface used to initialize the SigmaStudio for SHARC Target Framework.

10.1.1.1 InitAudioSystem

Prototype

```
SS4SAppStatus InitAudioSystem(tAppInfo *pAppInfo)
```

Description

Initializes peripherals for Audio input and output.

Parameters

Name:	`pAppInfo`
Type:	`tAppInfo *`
Direction:	Input
Description:	Pointer to the Application Information structure.

Return value

An appropriate error code of type `E_ADI_SS_APP_STATUS_SUCCESS` is returned. For the list of supported error codes refer to section 10.2 .

10.1.2 DAI Peripherals

This section describes the interfaces used to setup/initialize the DAI peripherals used in the Target Framework.

10.1.2.1 SPORT

This section describes the interface used to initialize the sport in I2S or TDM mode of operation.

10.1.2.1.1 InitializeSport

Prototype

```
void InitializeSport(tAppInfo *pAppInfo)
```

Description

Initialize the Sports in I2S or TDM mode which is defined by the user in the preprocessor definitions of the Application.

Parameters

Name:	**pAppInfo**
Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**

Return value

None

10.1.2.2 PCG

This section describes the interface used to configure the PCG based on the master/slave mode of operation of the SHARC Target.

10.1.2.2.1 setupPCG

Prototype

```
SS4SAppStatus setupPCG(tAppInfo *pAppInfo,tProcMode eProcMode)
```

Description

Configures the PCG based on the master/slave mode of operation of the SHARC Target.

Parameters

Name:	**pAppInfo**

Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**

Name:	**eProcMode**
Type:	**tProcMode**
Direction:	**Input**
Description:	**Enumeration for processor master/slave mode of operation**

Return value

An appropriate error code of type `E_ADI_SS_APP_STATUS_SUCCESS` is returned. For the list of supported error codes refer to section 10.2 .

10.1.2.3 SRC

This section describes the interface used to initialize the SRC module based on the user settings for the specified SRC pair.

10.1.2.3.1 InitSRC

Prototype

```
SS4SAppRes InitSRC(tAppInfo *pAppInfo, volatile unsigned int *pCtrlRegSRC, int nModeSRC, tSRCPair
eEnableSRCPair, tSRC eEnableSRC)
```

Description

Initializes the SRC module based on user settings for the specified SRC pair.

Parameter

Name:	**pAppInfo**
Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**

Name:	**pCtrlRegSRC**
Type:	**volatile unsigned int ***
Direction:	**Input**
Description:	**Pointer to SRC Control Register**

Name:	**nModeSRC**
Type:	**int**
Direction:	**Input**
Description:	**Settings for SRC mode of operation which is configured by the user**

Name:	**eEnableSRCPair**
Type:	**tSRCPair**
Direction:	**Input**
Description:	**Enumeration for the SRC pair to be configured**

Name:	**eEnableSRC**
Type:	**tSRC**
Direction:	**Input**
Description:	**Enumeration for enabling the SRC unit of a SRC pair**

Return value

An appropriate error code of type `APP_RES_FAILURE` is returned. For the list of supported error codes refer to section 10.2 .

10.1.2.4 S/PDIF

This section describes the APIs used to initialize and update the channel status of the S/PDIF hardware peripheral.

10.1.2.4.1 InitSPDIF

Prototype

```
SS4SAppStatus InitSPDIF(tSPDIFConfig *pSPDIFConfig);
```

Description

This function initializes the S/PDIF transmitter and receiver hardware peripherals. It also sets the S/PDIF channel status bits to the values configured by the user in `pSPDIFConfig` structure. This API may be called to reinitialize the channel status bits to a different value.

Parameter

Name:	**pSPDIFConfig**
Type:	**tSPDIFConfig***
Direction:	**Input**
Description:	**Pointer to the S/PDIF channel status config structure.**

Return value

An error code of type `SS4SAppStatus`.

10.1.2.4.2 CopySPDIFStatusInfoFromRxToTx

Prototype

```
void CopySPDIFStatusInfoFromRxToTx(void);
```

Description

This function copies the channel status information obtained from S/PDIF receiver to S/PDIF transmitter. Note that in Analog/Digital co-existence Digital clock mode, the fields corresponding to word length, sample word length, non-audio and category code are set to the user configured value than whatever is received by S/PDIF receiver. Refer to Annexure F for more details on S/PDIF channel status information.

Parameter

None.

Return value

None.

10.1.3 System Peripherals

10.1.3.1 CODEC

This section describes the interface used to setup the audio codec use in the Target Framework.

10.1.3.1.1 InitAudioCodec

Prototype

```
void InitAudioCodec(tAD1939Info *pAD1939Info, int nConfigSize, int nCodec, int bAnDigCoex)
```

Description

Function to set up the AD1939 registers via SPI.

Parameters

Name:	**pAD1939Info**
Type:	**tAD1939Info ***
Direction:	**Input**
Description:	**Pointer to the AD1939 Information structure.**
Name:	**nConfigSize**
Type:	**Int**
Direction:	**Input**
Description:	**Size of the structure in elements**
Name:	**nCodec**
Type:	**Int**
Direction:	**Input**
Description:	**Device select used to Setup the SPI Flag register**
Name:	**bAnDigCoex**
Type:	**Int**
Direction:	**Input**
Description:	**Flag indicating analog digital co-existence**

Return value

None

10.1.4 Input-Output

This section describes the interfaces used to get input data and write output data to/from the SPORT buffers in I2S or TDM mode of operation of the SPORTS.

10.1.4.1 Audio Input Data

This section describes the interfaces used to get input data from the SPORT buffers in I2S or TDM mode of operation of the SPORTS.

10.1.4.1.1 GetInputDataI2S

Prototype

```
void GetInputDataI2S(tAppInfo *pAppInfo,int nNumInputCh,int nNumInputSPDIFCh)
```

Description

Read input sample from SPORT buffers to input buffers in I2S mode.

Parameters

Name:	**pAppInfo**
Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**
Name:	**nNumInputCh**
Type:	**Int**
Direction:	**Input**
Description:	**Number of input analog channels**
Name:	**nNumInputSPDIFCh**
Type:	**Int**
Direction:	**Input**
Description:	**Number of input SPDIF channels**

Return value

None

10.1.4.1.2 GetInputDataTDM

Prototype

```
void GetInputDataTDM(tAppInfo *pAppInfo,int nNumInputCh,int nNumInputSPDIFCh)
```

Description

Read input sample from SPORT buffers to input buffers in TDM mode.

Parameters

Name:	**pAppInfo**
Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**

Name:	**nNumInputCh**
Type:	**Int**
Direction:	**Input**
Description:	**Number of input analog channels**

Name:	**nNumInputSPDIFCh**
Type:	**Int**
Direction:	**Input**
Description:	**Number of input SPDIF channels**

Return value

None

10.1.4.2 Audio Output Data

This section describes the interfaces used to write output data to the SPORT buffers in I2S or TDM mode of operation of SPORTS.

10.1.4.2.1 WriteOutputDataI2S

Prototype

```
int WriteOutputDataI2S(tAppInfo *pAppInfo,int nNumOutputCh,int nNumOutputSPDIFCh)
```

Description

Write output data to SPORT buffers in I2S mode.

Parameters

Name:	**pAppInfo**
Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**

Name:	**nNumOutputCh**
Type:	**Int**
Direction:	**Input**
Description:	**Number of output analog channels**

Name:	**nNumOutputSPDIFCh**
Type:	**Int**
Direction:	**Input**
Description:	**Number of output SPDIF channels**

Return value

Returns "0" if there is no clipping and "1" if any sample is clipped.

10.1.4.2.2 WriteOutputDataTDM

Prototype

```
int WriteOutputDataTDM(tAppInfo *pAppInfo,int nNumOutputCh,int nNumOutputSPDIFCh)
```

Description

Write output data to SPORT buffers in TDM mode.

Parameters

Name:	**pAppInfo**
Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**

Name:	**nNumOutputCh**
Type:	**Int**
Direction:	**Input**
Description:	**Number of output analog channels**

Name:	**nNumOutputSPDIFCh**
Type:	**Int**
Direction:	**Input**

Description:	Number of output SPDIF channels

Return value

Returns "0" if there is no clipping and "1" if any sample is clipped.

10.1.4.3 SPI Read-Back

This interface is used to send the data back to the host through SPI Read-Back. This interface also checks for custom command received and sets the flags accordingly.

10.1.4.3.1 HandleBackChCustomCmd

Prototype

```
void HandleBackChCustomCmd(tAppInfo *pAppInfo, ADI_SS_SSNPROPERTIES *pGetProperties)
```

Description

This function handles any back channel or custom command received from the SigmaStudio Host.

Parameters

Name:	**pAppInfo**
Type:	**tAppInfo ***
Direction:	**Input**
Description:	**Pointer to the Application Information structure.**
Name:	**pGetProperties**
Type:	**ADI_SS_SSNPROPERTIES**
Direction:	**Input**
Description:	**Pointer to SSn configuration structure**

Return value

None

10.2 Framework Enumerations

10.2.1 SS4SAppRes

```
typedef enum SS4SAppRes
```

```
{
       APP_RES_FAILURE = E_APP_RES_FAILURE,
       APP_RES_SUCCESS = E_APP_RES_SUCCESS,
       APP_RES_MEM_INSUFFICIENT = E_APP_RES_MEM_INSUFFICIENT,
       APP_RES_MEM_ERR_WRONG_INDEX = E_APP_RES_MEM_ERR_WRONG_INDEX
}SS4SAppRes;
```

Description

This enumeration represents the general error codes for the SigmaStudio for SHARC Target Default Application.

10.2.2 SS4SAppStatus

```
typedef enum SS4SAppStatus
{
       ADI_SS_APP_STATUS_SUCCESS                  = E_ADI_SS_APP_STATUS_SUCCESS,
       ADI_SS_APP_STATUS_COMM_CREATE_ERR    = E_ADI_SS_APP_STATUS_COMM_CREATE_ERR,
       ADI_SS_APP_STATUS_COMM_INIT_ERR      = E_ADI_SS_APP_STATUS_COMM_INIT_ERR,
    ADI_SS_APP_STATUS_PCG_ERR                     = E_ADI_SS_APP_STATUS_PCG_ERR,
    ADI_SS_APP_STATUS_SPDIF_ERR                   = E_ADI_SS_APP_STATUS_SPDIF_ERR
}SS4SAppStatus;
```

Description

This enumeration represents the specific error codes for the SigmaStudio for SHARC Target Default Application.

11 Default Target Library Configuration Parameters set in the Application

This section explains about the default configuration parameters set in the Application for Target Library API functions.

11.1 adi_ss_comm_init

This section describes the default configuration parameters set in the Application for *adi_ss_comm_init* Target Library API. Refer Table 3 for the default values set in the Application.

Communication configuration parameter	Default values set in the Application
baudRateRx	Currently not used internally
baudRateTx	Currently not used internally
nSelectSPI	SELECT_SPI0
bCRCBypass	0
bFullPacketCRC	1
(*pfCommCallBack)()	app_ss_comm_callback_cmd4
(*pfRegSPIIsrCallBack)(int32_t nPriority,void (*pfSPIISR)(int32_t))	app_ss_comm_regspiisr_callback

Table 3: Default configuration parameters set in the Application for *adi_ss_comm_init* Target Library API

11.2 adi_ss_init

This section describes the default configuration parameters set in the Application for *adi_ss_init* Target Library API. Refer Table 4 for the default values set in the Application.

Communication configuration parameter	Default values set in the Application
nBlockSize	BLOCK_SIZE = 64
nInChannels	NUM_INPUT_CHANNELS = 6
nOutChannels	NUM_OUTPUT_CHANNELS = 10
bSkipProcessOnCRCError	0
bSkipInitialDownload	0
hSSComm	hSSComm = 0

Table 4: Default configuration parameters set in the Application for *adi_ss_init* Target Library API

A. Maintaining sync between memories reserved in the Application for SSn and Schematic IC control form

The code, data, and parameter memories for the SSn are reserved in the Application. The following table lists the Application macros, which define the code, data and parameter memory sizes for the SSn.

Macro	Memory	Memory Size
ADI_SS_SIZE_BLOCK_1	L1 code memory (Code)	48-bits or 16-bits depending on non-VISA or VISA mode
ADI_SS_SIZE_BLOCK_4	L1 state memory (State)	32-bits
ADI_SS_SIZE_BLOCK_5	L1 parameter memory	32-bits
ADI_SS_SIZE_BLOCK_6	L1 extended precision state memory	48-bits
ADI_SS_SIZE_BLOCK_7	L3 code memory (Code B)	48-bits or 16-bits depending on non-VISA or VISA mode
ADI_SS_SIZE_BLOCK_8	L1 state memory (State B) in extended precision memory block	32-bits
ADI_SS_SIZE_BLOCK_9	L3 state memory (State C)	32-bits

Table 5: Application macros that define the SSn memory sizes

The default SSn memory sizes are described in Annexure A of [1]. The 32-bit L1 state and the parameter memory sizes for the SSn are dependent on the memory used by the framework state. If the framework state memory is altered by changing any of the macros mentioned in Table 6, the 32-bit L1 state and parameter memory sizes for the SSn also get changed automatically. This automatic change is controlled by the Application macros “MEMORY_USAGE_FACTOR_BLK1” and “MEMORY_USAGE_FACTOR_BLK2” as shown below.

```
/* State Memory */
#ifdef __ADSP21369__
#define ADI_SS_SIZE_BLOCK_4      (54*1024/4-
(MEMORY_USAGE_FACTOR_BLK1+MEMORY_USAGE_FACTOR_BLK2))
#elif defined(__ADSP21364__)
#define ADI_SS_SIZE_BLOCK_4      (90*1024/4-MEMORY_USAGE_FACTOR_BLK1)
#else   /* 214xx */
#define ADI_SS_SIZE_BLOCK_4      (142*1024/4-MEMORY_USAGE_FACTOR_BLK1)
#endif
```

```
/* Parameter Memory */
#ifdef __ADSP21369__
#define ADI_SS_SIZE_BLOCK_5      (32*1024/4)
#elif defined(__ADSP21364__)
#define ADI_SS_SIZE_BLOCK_5      (60*1024/4-MEMORY_USAGE_FACTOR_BLK2)
#else   /* 214xx */
#define ADI_SS_SIZE_BLOCK_5      (99*1024/4-MEMORY_USAGE_FACTOR_BLK2)
#endif
```

Macro	Definition	Default value (in I2S mode)	Default value (in TDM mode)
NUM_INPUT_CHANNELS	Number of analog and S/PDIF input channels	6 for ADSP-214xx SHARC Targets. 4 for ADSP-213xx SHARC Targets.	6 for ADSP-214xx SHARC Targets. 4 for ADSP-213xx SHARC Targets.
NUM_OUTPUT_CHANNELS	Number of analog and S/PDIF output channels	10	10
SPORT_BUFFER_SIZE	SPORT buffer size	64	64
PROCESSING_BLK_SIZE	Application Block Size	64	64
NUMBER_PROCESSING_BUFFERS	Number of SPORT buffers in terms of PROCESSING_BLK_SIZE	3	3
INPUT_TO_OUTPUT_RATE	Input to output sample rate ratio	1	1

OUTPUT_TO_INPUT_RATE	Output to input sample rate ratio	1	1
NUM_INPUT_SPORT_BUFFERS	Number of input SPORT buffers	3	3
NUM_OUTPUT_SPORT_BUFFERS	Number of output SPORT buffers	3	3
NUM_INPUT_SPDIF_CHANNELS	Number of input S/PDIF channels	2	2
SPORT_TCB_SIZE	TCB size of SPORT	4	4
MAX_OUT_SLOTS	Maximum number of output slots	3	3

Table 6: Application macros which influence the SSn state and parameter memory sizes

A.1 Impact of macro "MEMORY_USAGE_FACTOR_BLK1"

The "MEMORY_USAGE_FACTOR_BLK1" macro is dependent on the size of the "tSportBufferInfo" structure as given below.

```
#define MEMORY_USAGE_FACTOR_BLK1 (sizeof(tSportBuffInfo)-DEFAULT_SIZE_BLK1)
```

The "tSportBufferInfo" structure is given below.

```
struct tSportBufferInfo
{
unsigned int
aSportInBuff[NUM_INPUT_CHANNELS*SPORT_BUFFER_SIZE*NUM_INPUT_SPORT_BUFFERS];

unsigned int
aSportOutBuff[NUM_OUTPUT_CHANNELS*SPORT_BUFFER_SIZE*NUM_OUTPUT_SPORT_BUFFERS];

int
aSportTCBIn[(NUM_INPUT_CHANNELS>>1)*NUM_INPUT_SPORT_BUFFERS*SPORT_TCB_SIZE];

int
aSportTCBOut[(NUM_OUTPUT_CHANNELS>>1)*NUM_OUTPUT_SPORT_BUFFERS*SPORT_TCB_SIZE];

unsigned int    *pInBuf[NUM_INPUT_CHANNELS>>1][NUM_INPUT_SPORT_BUFFERS];
unsigned int    *pOutBuf[NUM_OUTPUT_CHANNELS>>1][NUM_OUTPUT_SPORT_BUFFERS];
int             *pTCBIn[NUM_INPUT_CHANNELS>>1][NUM_INPUT_SPORT_BUFFERS];
int             *pTCBOut[NUM_OUTPUT_CHANNELS>>1][NUM_OUTPUT_SPORT_BUFFERS];
/* Offsets for circular output buffering */
int             aOutSportBuffOffset[MAX_OUT_SLOTS];
};
```

The size of "tSportBufferInfo" structure, based on the default values of the macros it uses as described in Table 6, is defined by the macro "DEFAULT_SIZE_BLK1" and is equal to 3651 words for the ADSP-214xx SHARC Target. Hence, if any of the macros listed in Table 6 changes, the size of "tSportBufferInfo" structure changes, causing the macro "MEMORY_USAGE_FACTOR_BLK1" to change. The value of this macro reduces or increases the size of SSn state memory defined by "ADI_SS_SIZE_BLOCK_4".

MEMORY_USAGE_FACTOR_BLK1 =
NUM_INPUT_CHANNELS*SPORT_BUFFER_SIZE*NUM_INPUT_SPORT_BUFFERS+
NUM_OUTPUT_CHANNELS*SPORT_BUFFER_SIZE*NUM_OUTPUT_SPORT_BUFFERS+
(NUM_INPUT_CHANNELS>>1)*NUM_INPUT_SPORT_BUFFERS*SPORT_TCB_SIZE+
(NUM_OUTPUT_CHANNELS>>1)*NUM_OUTPUT_SPORT_BUFFERS*SPORT_TCB_SIZE+
NUM_INPUT_CHANNELS*NUM_INPUT_SPORT_BUFFERS+
NUM_OUTPUT_CHANNELS*NUM_OUTPUT_SPORT_BUFFERS+
MAX_OUT_SLOTS-
DEFAULT_SIZE_BLK1

Consider an example where the NUMBER_PROCESSING_BUFFERS is changed from its default value to a value equal to 6 for an ADSP-214xx SHARC Target. Let us assume INPUT_TO_OUTPUT_RATE = 1 and OUTPUT_TO_INPUT_RATE = 1. Then,

NUM_INPUT_SPORT_BUFFERS = (6*64/16)*1 = 24

NUM_OUTPUT_SPORT_BUFFERS = (6*64/16)*1 = 24

MEMORY_USAGE_FACTOR_BLK1 = 6*16*24+10*16*24+(6/2)*24*4+(10/2)*24*4+6*24+
10*24+3-3651

= 7299-3651
= 3648

Hence the SSn state memory now shall be:

ADI_SS_SIZE_BLOCK_4 = (142*1024/4-MEMORY_USAGE_FACTOR_BLK1)
= 36352-3648
= 32704 words

This value must be entered in the "32 Bit State" tab of Schematic IC control form, in bytes (32704*4). This value in words can also be obtained from the map file by searching for the variable "_adi_ss_mem4".

A.2 Impact of macro "MEMORY_USAGE_FACTOR_BLK2"

"MEMORY_USAGE_FACTOR_BLK2" macro is dependent on Application Block Size and input and output number of channels as given below.

```
#define MEMORY_USAGE_FACTOR_BLK2
(NUM_OUTPUT_CHANNELS+NUM_INPUT_CHANNELS+NUM_INPUT_SPDIF_CHANNELS)*(PROCESSING_BL
K_SIZE-64)
```

A change in any of these parameters causes macro "MEMORY_USAGE_FACTOR_BLK2" to change, which in turn reduces or increases the SSn parameter memory size defined by the macro "ADI_SS_SIZE_BLOCK_5".

MEMORY_USAGE_FACTOR_BLK2 = NUM_OUTPUT_CHANNELS +
NUM_INPUT_CHANNELS + NUM_INPUT_SPDIF_CHANNELS) *
(PROCESSING_BLK_SIZE-64)

Consider an example where the PROCESSING_BLK_SIZE is changed to 128 from its default value of 64 for an ADSP-214xx SHARC Target, then
MEMORY_USAGE_FACTOR_BLK2 = (10+6+2)*(128-64)
= 1152

Hence the SSn parameter memory now shall be:

ADI_SS_SIZE_BLOCK_5 = (99*1024/4-1152)
= 24192 words

This value in bytes (24192 *4) must be entered in the "Parameter" tab of Schematic IC control form. This value in words can also be obtained from the map file by searching for the variable "_adi_ss_mem5".

Thus, if the Schematic uses Plug-Ins and if there is any change in macros defined in Table 6, the user has to calculate the values of SSn state and parameter memories defined by macros "ADI_SS_SIZE_BLOCK_4" and "ADI_SS_SIZE_BLOCK_5" and populate the sizes in the IC Control Window accordingly.

B. Computation of Average and Peak MIPS in the Application

The average and peak MIPS of a Schematic are computed for every call to the SigmaStudio for SHARC process API (*adi_ss_schematic_process*()). The *adi_ss_schematic_process*() API is described in section 7.3.6 of [3].

The *MIPS measurement period* for average and peak MIPS measurement is set to 5 seconds within the Application. The average and peak MIPS are reset after a duration of every *MIPS measurement period*. To change the *MIPS measurement period* within the Application to a different value, set the value of the macro "SS_APP_MIPS_MEASURE_PERIOD" in file *app.h* to the desired duration.

Sections B.1 and B.2 below describe the computation of average and peak MIPS of a Schematic.

B.1 Average MIPS

The average MIPS is computed based on the *Average cycles per sample* over the *MIPS measurement period.*

The average MIPS is computed as given below.

Average MIPS of the Schematic $MIPS_{Avg}$ *= (Average cycles per sample * Application Sampling Rate) / 1000000*

Where, *Average cycles per sample* is the average cycles consumed to process one sample over the *MIPS measurement period = (Total cycles / Total samples processed per channel).*

The Schematic average MIPS must not exceed the maximum MIPS possible for the chosen SHARC Target.

B.2 Peak MIPS

The peak MIPS is computed based on absolute maximum cycle count consumed for each call to the SigmaStudio process API over the *MIPS measurement period.*

The peak MIPS is computed as given below.

Peak MIPS of the Schematic $MIPS_{Peak}$ =

(Maximum cycles * Application Sampling Rate) / (BPI * 1000000)

Where, BPI = PROCESSING_BLK_SIZE

The value of peak MIPS may be higher than the maximum MIPS supported by the SHARC Target since the process() API of a Plug-In may be called only for some of the frames, depending on the Module input buffering size (BMI).

The peak MIPS is related to Module peak MIPS ($MIPS_{PeakMod}$), BMI and BPI as in the equation given below

$MIPS_{Peak} = MIPS_{PeakMod}*(BMI/BPI)$

For example, consider a Module with the following values

BMI = 512
BPI = 48
$MIPS_{PeakMod} = 76$

Then, $MIPS_{Peak} = 76 * (512/48) = 810$.

The peak MIPS provides useful insight into the buffering requirements of the Application. The number of processing buffers (*NUMBER_PROCESSING_BUFFERS*) is directly proportional to the peak MIPS.

$MIPS_{Peak} \alpha$ NUMBER_PROCESSING_BUFFERS

Moreover, the Default Application provides peak MIPS over a *MIPS measurement period* of 5s, which is a long enough duration to ascertain the true peak MIPS of a Schematic. Hence it is possible to arrive at optimal buffering requirements for input and output by knowing the value of the peak MIPS of a Schematic. Typically, if *NUMBER_PROCESSING_BUFFERS* is 3, then the absolute peak MIPS can go up to 800 MIPS.

C. NaN Handling in the framework

Under unusual circumstances, when the Schematic produces NaN output during processing, the NaN values are to be removed from the state memory. The Application searches for NaN values in the output buffer using the function *IsNanInf().* If found, clear the entire state buffer by calling the *adi_ss_clearState()* API [3]. The details of function *IsNanInf()* is as follows:

Prototype

```
int IsNanInf (float *pInput, int nInBufSize);
```

Description

Check if NaN found in the output samples.

Parameters

Name:	`pInput`
Type:	`float *`
Direction:	Input
Description:	Pointer to the sample array.
Name:	`nInBufSize`
Type:	`Int`
Direction:	Input
Description:	Size of the sample array.

Return value

*IsNanInf()*returns 1 if NaN is detected in the output buffer else it returns 0.

When NaN is detected, the Application clears the state memory by calling *adi_ss_clearState()* API. Once the above API is called, the next call to *adi_ss_schematic_process()*re-initializes all the Plug-Ins.

The Default Application implements the above operation in *CheckNaNInf()*. If there are multiple output buffers, the NaN detection has to be repeated for all the output buffers. Upon detecting NaN in any of the output buffers, the state memory has to be cleared.

D. Porting Application to other SHARC variants or other platforms

The SigmaStudio for SHARC package contains Default Applications for the following variants of the SHARC Target.

- ADSP-21364 SHARC Target using ADSP-21364 EZ-KIT Lite platform.
- ADSP-21369 SHARC Target using ADSP-21369 EZ-KIT Lite platform.
- ADSP-21469 SHARC Target using ADSP-21469 EZ-Board platform.
- ADSP-21479 SHARC Target using ADSP-21479 EZ-Board platform.
- ADSP-21489 SHARC Target using ADSP-21489 EZ-Board platform.

The following points must be noted while porting the Application for a different SHARC variant.

1. Processor specific aspects:
    a. Memory: The L1 memory available may vary across different SHARC variants. The memory for the SSn code, data and parameters must be allocated depending on the memory requirements of the Schematic and the available L1 memory for the chosen variant.
    b. System initialization: The maximum core and peripheral clock speeds supported by different variants are not the same. Also some variants support DDR/SDRAM. System initialization such as PLL and DDR/SDRAM initializations must be performed in accordance with the chosen variant.
2. Platform specific aspects: Platform in this context refers to the entire hardware environment other than the SHARC Target. The Default Application provided with the package is for the EZ-KIT Lite/EZ-Board platforms as mentioned above. The following aspects must be considered while moving to a platform other than the EZ-KIT Lite/EZ-Board.
    a. CODEC: The EZ-KIT Lite/EZ-Board has an onboard CODEC which is programmed from the SigmaStudio for SHARC Application. If a different CODEC is to be used, then, it must be programmed accordingly by modifying the Application.
    b. LED/push button: The EZ-KIT Lite/EZ-Board have LEDs and push buttons which are mapped to SHARC Target DAI/DPI pins through switches/jumpers. The switch/jumper settings for the Default Application are explained in detail in section 5.2 of [1]. These parts of the Application code have to be reviewed when moving to a platform other than EZ-KIT Lite/EZ-Board.
    c. SRU routing: The SRU routing for the SPORTS/SPI and other peripherals have to be reconsidered while moving to a platform other than EZ-KIT Lite/EZ-Board.

D.1 Platform specific build time macros in the Application

The Default Application Loader Files provided within the package are built with a platform specific build time macro. The platform specific build time macros are listed below:

1. __ADSP21364_EZKIT__ - This macro must be defined for building a Loader File for ADSP-21364 SHARC Target on ADSP-21364 EZ-KIT Lite platform.
2. __ADSP21369_EZKIT__ - This macro must be defined for building a Loader File for ADSP-21369 SHARC Target on ADSP-21369 EZ-KIT Lite platform.
3. __ADSP21469_EZKIT__ - This macro must be defined for building a Loader File for ADSP-21469 SHARC Target on ADSP-21469 EZ-Board platform.
4. __ADSP21479_EZKIT__ - This macro must be defined for building a Loader File for ADSP-21479 SHARC Target on ADSP-21479 EZ-Board platform.
5. __ADSP21489_EZKIT__ - This macro must be defined for building a Loader File for ADSP-21489 SHARC Target on ADSP-21489 EZ-Board platform.

D.2 Steps to be followed for porting the Application to a different SHARC variant which uses the EZ-KIT Lite/EZ-Board platform

Follow the steps below for porting the Application to a different SHARC variant which uses the EZ-KIT Lite/EZ-Board platform.

1. Identify the SHARC Target which closely matches the variant chosen, from the list of the SHARC Targets supported by the SigmaStudio for SHARC Application described in section D.
2. Open the Application for the identified SHARC Target in CrossCore Embedded Studio.
3. Change the processor type to match the variant chosen. This can be done by clicking the "Properties➔C/C++ Build➔Settings" and choosing the processor in the "Processor Settings" tab.
4. If the L1 memory available on the chosen variant is identical to that of the identified SHARC Target in step 1, change the "Architecture" to the chosen variant in the LDF file. If available L1 memory is different, or if the variant differs to the identified SHARC Target in step 1 in supporting L3 memory, then generate a new LDF file and map the input sections within the Application to different output sections of the LDF.

5. Rebuild the Application and follow the instructions in the #error statements displayed while building the Application.

D.3 Steps to be followed for porting the Application to a SHARC variant on a custom platform

When porting the Application to a custom platform, please follow the steps mentioned below.

1. If the processor is one among the SHARC Targets supported by the SigmaStudio for SHARC DefaultApplication, as described in section D, then,
    a. Open the Default Application in CrossCore Embedded Studio.
    b. From the "Properties➔C/C++ Build➔Settings➔CrossCore SHARC C/C++ Compiler➔Preprocessor" tab, undefine the platform specific preprocessor macro listed in section D.1.
    c. Rebuild the Application and follow the instructions in the #error statements displayed while building the Application. It is recommended to visit all sections of code, which use the platform specific macros listed in section D.1 and make necessary modifications for the custom platform.
2. If the processor is a variant of the SHARC Targets supported by the SigmaStudio for SHARC Default Application as described in section D, then, follow the instructions outlined in section D.2 first before following the instructions in step 1 above.

E. Framework performance parameters

The framework MIPS and memory requirements are tabulated below.

Processor	Code RAM (bytes)	Data RAM (bytes)	Average MIPS
ADSP-21469	28712	19128	12.46
ADSP-21479	28484	19128	12.14
ADSP-21489	28486	19128	12.48
ADSP-21364	29436	16484	11.39
ADSP-21369	32910	16560	11.51

Table 7: Performance figures for SigmaStudio for SHARC framework

Note:

1. The above measurements are obtained with CrossCore Embedded Studio tool chain v1.1.0 and with the Default Application Loader Files.

2. The Average MIPS of the framework is measured with *Analog\Digital Co-existence* as the Input-Output Mode for all the processors.

F. S/PDIF Channel Status

The channel status fields provide information related to the audio data that is carried over the S/PDIF interface. Only Consumer format (i.e., first 40 bits) of the S/PDIF channel status fields are supported in this release of SigmaStudio for SHARC.

In I2S mode of SPORT's operation, if the Input-Output Mode is set to *Digital-Out alone*, the SHARC S/PDIF Transmitter channel status fields are the user set values which are configured in *SetSPDIFChStatusFields()* function of *SigmaStudio for SHARC* Application. For other Input-Output Modes, the received S/PDIF channel status fields from the SHARC S/PDIF receiver are copied to the S/PDIF channels status fields of SHARC S/PDIF transmitter.

In TDM mode of SPORT's operation, the received S/PDIF channel status fields from the SHARC S/PDIF receiver are copied to the S/PDIF channels status fields of SHARC S/PDIF transmitter, if the S/PDIF signal is plugged-in into the SHARC Target. If the S/PDIF signal is unplugged from SHARC Target then, the SHARC S/PDIF Transmitter channel status fields are the user set values which are configured in *SetSPDIFChStatusFields()* function of *SigmaStudio for SHARC* Application.

Note that, the received S/PDIF channel status fields from the SHARC S/PDIF receiver are copied to the S/PDIF channels status fields of SHARC S/PDIF transmitter if the Input-Output Mode is *Analog/Digital Co-existence (Digital Clock)* irrespective of I2S or TDM mode of SPORT's operation.

F.1 Default S/PDIF channel status fields set in SigmaStudio for SHARC Application

The default S/PDIF channel status fields which are set in *SetSPDIFChStatusFields()* function of *SigmaStudio for SHARC* Application are given in the Table 8.

SPDIF Channel Status fields	Default values
Mode	SPDIF_FORMAT_CONSUMER
Audio Mode	SPDIF_LINEAR_PCM_SAMPLES
Copyright	SPDIF_NO_COPYRIGHT_ASSERTED
Emphasis	SPDIF_2CH_NO_PRE_EMPHASIS
Channel Mode	SPDIF_CHANNEL_STATUS_MODE0
Category Code	SPDIF_CATEGORY_CODE_GENERAL
Source Number	SPDIF_SOURCE_NUMBER_NOT_INDICATED
Channel Number	SPDIF_CHANNEL_NUMBER_NOT_INDICATED

Sample Frequency	SAMPLING_RATE_48K
Clock Accuracy	SPDIF_CLOCK_ACCURACY_LEVEL_2
Word Length	SPDIF_MAX_WORD_LENGTH_24_BITS
Sample Word Length	SPDIF_SAMPLE_WORD_LENGTH_24_BITS
Original Sampling Frequency	SPDIF_ORIGINAL_SAMPLING_FREQ_NOT_INDICATED

Table 8: Default S/PDIF channel status parameters set in the SigmaStudio for SHARC Application