PA SDK version 100713

ASP Development Kit

DA8xx ASP Development Kit Overview

Introduction

Audio Stream Processing Development Kit (ASPDK) is a guide for Customers and Third parties to do porting/optimization/integration of Audio Stream Processing Algorithms to Performance Audio Framework (PA/F) operating on DA8xx family of devices. This development kit intends to help in adding ASP algorithms to PA/F and it consolidates all necessary information like example code, utilities and documentation in one place. This package is intended to be used together with PA/F SDK for DA8xx device. All subsequent documentation here assumes that user has already downloaded the PA/F SDK for DA8xx device. Please refer to Getting Started Guide for getting enviroment set up with DA8x hardware and CCSv4. ASPDK talks about only software updates and requirements for custom ASP development to work within PA/F.

Overview of the content of package and chapters in this document

Example ASPs: ASPs demonstating usage of common PA modules and settings

Equalizer example: Demonstrates the usage of Filter Library (FIL).

Virtualizer example: Demonstrates the usage of FFT.

Surrounds eample: Deonstrates the usage of some Common PA library functions (CPL).

FIL example ASPs

pag.exe: This is a utility to generate PA/F layer and ASP wrapper code(main.c)
example_asp.bsp: This is a text based configuration file to be used with pag.exe
Given below is brief description of some important documentation provided in this package.

ASP Prgramming Interface

The PA/F ASP Algorithm components are based on the TI XDAIS Algorithm Standard. The PA/F Algorithms extend the standard XDAIS Algorithm interface, in a standard, fully–compliant way, to provide the required functionality for digital audio processing. Description of ASP programming interface is available in section → ASP Programming Interface .

PA/M Interface

The Performance Audio Messaging Interface serves as the channel for communication with the ASP Algorithm. The PA system utilizes the PA/M protocol for communication between different parts of the system. The communication is achieved by means of word-based units, where each word is a 16-bit unsigned integer of the form 0xhhhh, and is called an *alpha* code word. Description of *alpha* codes and how to write them is avaialble in section → Alpha Codes.

ASP Porting Guide

After the readers acquire an idea of PA/F, ASP interfaces, it's expected that user can proceed to port or integrate the custom algorithm into the PA/F on DA8xx by following a step-by-step → ASP Porting Guide.

PA/F Integration tips

This chapter gives some important tips related to integration of custom ASP algorithm in PA/F. Please refer → this chapter along with following the → ASP porting Guide

PA FIL Library API

PA FIL library aims at providing a library of functions for programmers to implement mips efficient floating point audio filters for TMS320DAxx family of devices. This document describes the FIL library API's. This is avaiable as pdf document `pa/doc/pa-asp-fil.pdf`

Creating New ASP Algorithms using FIL

This appliation report shows how to create ASP algorithms using PA FIL library. The FIL audio examples given in this document were created by copying/modifying FIL example ASPs available in this package. The documents is available as pdf document `pa/doc/pa-fil-fe.pdf`

PA HD Channels Extension

The PA Framework has been originally built to accommodate a maximum of 16 audio channels with 8 channels (7.1) being typically used. This document decsribes PA/F updates to allow processing of the new channel types (possibly upto 32 channels) defined by various new audio formats. The document is available as pdf document `pa/doc/paf-hd.pdf`

DA8xx ASP Programming Interface

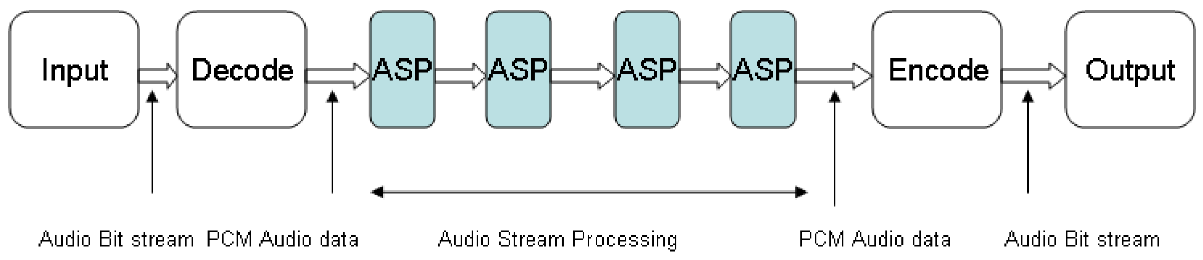
Introduction

This page descibes the PA/F Programming Interface for ASP algorithms.

Performance Audio Framework (PA/F) Overview

The Performance Audio Framework contains all the components needed to form a complete digital audio processing solution. The major components of an example audio stream, as implemented using the PA/F, are shown in Figure 1-1. PA/F is described in detail in PA User's Guide.This section briefly describes the ASP component in PA/F.

Figure 1-1: PA/F Audio Stream Components



Decode Component

This component is responsible for decoding the incoming audio stream which could be encoded as PCM, Dolby Digital, DTS etc.

Encode Component

The PA/F requires Decode and Encode Components, even if there is no “decoding” or “encoding” taking place. In other words, if the incoming bit stream is not encoded, there is no need for a decoding function, but there must still be a Decode Component in the system, a “PCM Decoder”. The same is true of the Encode Component. Usually, it will be a “PCM Encoder”, which really isn’t encoding to a compressed bit-stream format but must still be present.

ASP Component

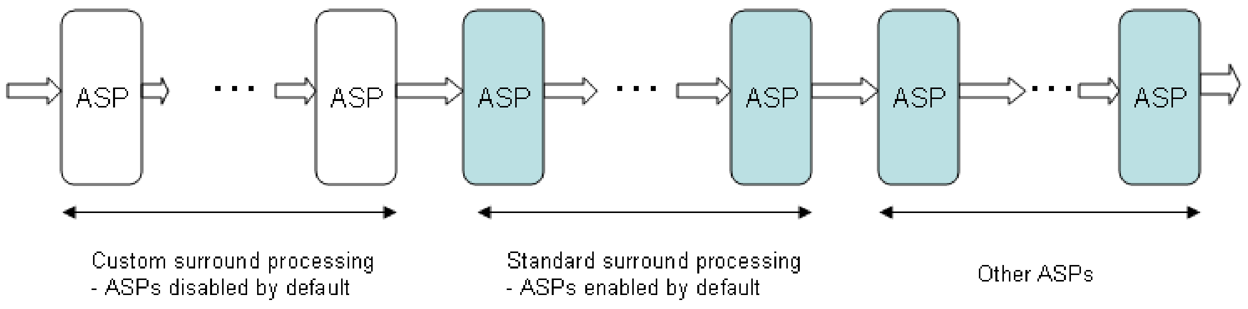
This component corresponds roughly to what might be called "post processing" in other systems that are designed for audio decoding only. In PA/F, there can be more than one ASP component in a processing chain. The most common manner in which customer-specific code is inserted into the PA/F is by creating one or more of these ASP Components.

ASP algorithms may be classified depending on the kind of processing they perform. For example, the custom surround (SUR) algorithm example in this package is a surround processing ASP algorithm. It is important to remember that

- All surround processing ASPs are grouped together in the ASP chain.
- Custom surround processing ASPs, such as SUR are first.
- Standard surround processing ASPs, such as PL2x, come next.

This grouping of ASP algorithms is shown in Figure 1-2 (Arrangement of ASP Algorithms in the ASP Chain). When an ASP algorithm is able to convert an input audio stream into the desired output audio stream, it should prevent the operation of other ASP Algorithms as appropriate. For example, within the group of surround processing ASPs, only one of them may process the audio data. To preclude the operation of other surround processing ASPs further down the chain, the recommended method is to set the channel configuration accordingly to prevent surround processing by ASPs further down the chain. Even though one may be tempted to obtain a similar result by actually disabling some of the other ASP Algorithms using alpha codes (described later), this method is discouraged.

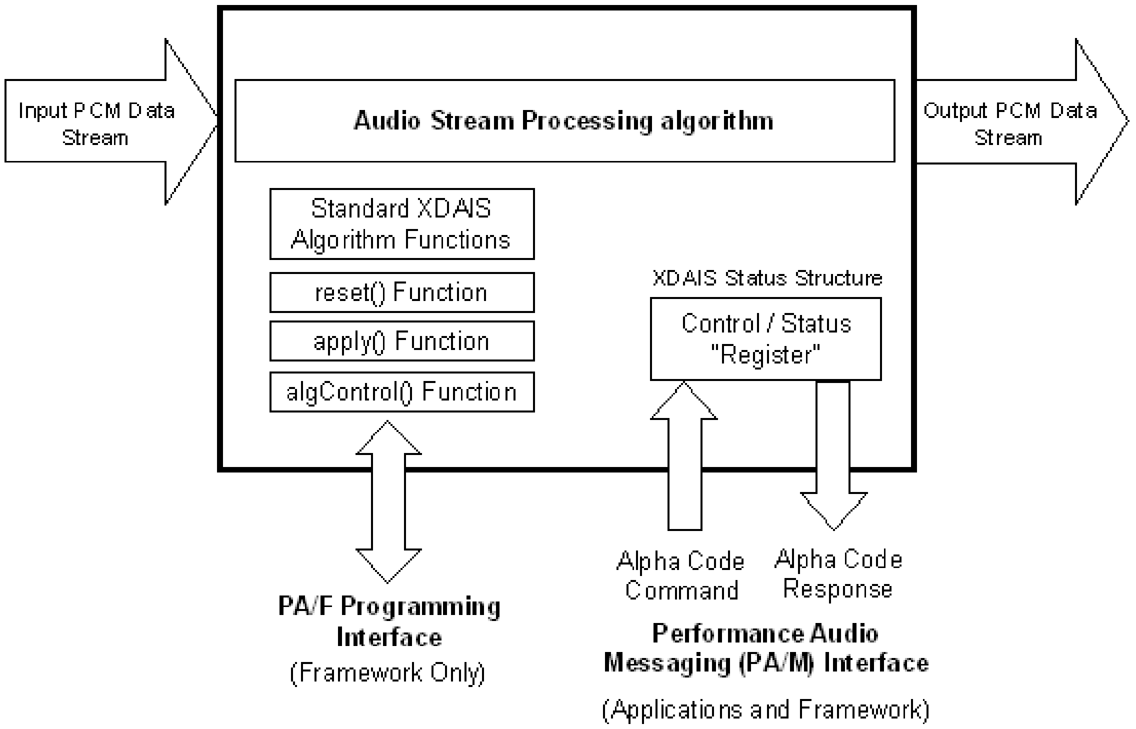
Figure 1-2: Arrangement of ASP algorithms in the ASP chain


ASP
ASP
ASP
ASP
ASP
ASP
Custom surround processing
- ASPs disabled by default
Standard surround processing
- ASPs enabled by default
Other ASPs

Programming Interface

The PA/F Algorithms extend the standard XDAIS Algorithm interface, in a standard, fully–compliant way, to provide the required functionality for digital audio processing as shown in Figure 1-3.

Figure 1-3 XDAIS-compliant Audio Stream Processing (ASP) Algorithm


Input PCM Data Stream
Audio Stream Processing algorithm
Output PCM Data Stream
Standard XDAIS Algorithm Functions
reset() Function
apply() Function
algControl() Function
XDAIS Status Structure
Control / Status "Register"
PA/F Programming Interface
(Framework Only)
Alpha Code Command
Alpha Code Response
Performance Audio Messaging (PA/M) Interface
(Applications and Framework)

The ASP Programming Interface is established with implementation of certain functions that operate on the control and audio data and also allow for incorporation of the ASP algorithm into the Framework. These functions are:

- **ALG_VEN_apply( )**

Processes a frame of audio data. This function also updates the control data needed to manage the processing chain. This should be implemented in `ALG_VEN_iALG.c`.

- **ALG_VEN_reset( )**

Initialises the control data associated with the ASP algorithm. This should be implemented in `ALG_VEN_iALG.c`.

- **ALG_VEN_control( )**

Enables the Framework to “connect” the alpha code messaging mechanism to the ASP Algorithm. This function is typically implemented in the file ALG_VEN_ialg.c.

Description of ALG_VEN_apply()

What parameters are passed to it?

The *_apply()* function for all ASP Algorithms must have the below interface:

```
Int ALG_VENDOR_apply(IALG_Handle handle, PAF_AudioFrame *pAudioFrame)
```

Where:

`IALG_Handle handle` -> ASP Algorithm handle

`PAF_AudioFrame *pAudioFrame` -> Pointer to the *AudioFrame* data structure

The ASP algorithm handle contains the pointer to the algorithm status structure. The status structure of the ASP algorithm provides the interface to the user (e.g., a microcontroller). The status structure works like a memory-mapped register bank. The user can read/write to this structure to pass control information to the ASP Algorithm or to read status information back from the ASP algorithm.

The *AudioFrame* data structure contains the audio data that needs to be processed and control and status information that the ASP algorithms should refer to understand how the processing needs to be applied. If the ASP algorithm modifies the audio samples within the *AudioFrame* data structure, it may be required to modify some of the control information in the *AudioFrame* data structure. This enables other downstream ASP algorithms to understand the contents of the *AudioFrame* data structure. For example, a surround processing ASP algorithm, if it generates more audio channels than existed in the audio stream, may need to modify the `ChannelConfigurationStream` quantity within the *AudioFrame* data structure.

The *AudioFrame* data structure is defined in the file `T:\pa\f\include\paftyp.h` as:

```
typedef struct PAF_AudioFrame {
   PAF_AudioFunctions *fxns;
   XDAS_Int8 mode;
   XDAS_Int8 sampleDecode;
   XDAS_Int8 sampleRate;
   XDAS_Int8 unused[3];
   XDAS_Int16 sampleCount;  /* valid N */
   PAF_AudioFrameData data; /* data[M][N] */
   PAF_ChannelConfiguration channelConfigurationRequest;
   PAF_ChannelConfiguration channelConfigurationStream;
   /* valid M*/
   PAF_ChannelConfigurationMaskTable
     *pChannelConfigurationMaskTable;
   PAF_SampleProcess sampleProcess[PAF_SAMPLEPROCESS_N];
   struct PAF_AudioFrame *root;
} PAF_AudioFrame;
```

The audio data is stored in *AudioFrameData* (indicated in italics in above structure) is defined below.

```
// PAF_AudioFrameData is a fixed structure which defines the
// possible data-carrying capacity of the audio frame.
typedef struct PAF_AudioFrameData {
   XDAS_Int16 nChannels;  /* max M */
   XDAS_Int16 nSamples;  /* max N */
   PAF_AudioData **sample;  /* sample[M][N] */
   PAF_AudioSize *samsiz;  /* samsiz[M] */
} PAF_AudioFrameData;
```

The audio data is stored in the *AudioFrameData* as non-interleaved PCM. That is, the data for the M channels are stored sequentially, one channel after the other. The data for any individual channel is referenced via a pointer such that `sample[M]` is a pointer to a vector of PCM data for channel M. **paf-hd.doc** shows in detail the correspondence between the channels and the elements in this array of pointers to the sample data. A custom ASP algorithm may process the available data present on any of these channels. The information extracted from *AudioFrame* by the *_apply()* function includes:

- The number of samples in each channel
- Available channels in the stream

- The audio data from available chaanels
- Audio sample-size (described in Audio sample-size)

This is achieved by appropriately de-referencing the pointer **pAudioFrame*:

```
sampleCount = pAudioFrame->sampleCount;
     // Number of samples in the audio frame (in each channel)
left = pAudioFrame->data.sample[PAF_LEFT];
    // pointer to left channel audio sample buffer
rght = pAudioFrame->data.sample[PAF_RGHT];
    // pointer to right channel audio sample buffer
cntr = pAudioFrame->data.sample[PAF_CNTR];
   // pointer to center channel audio sample buffer
.
.
.
samsiz = pAudioFrame->data.samsiz; // Audio size
```

Finally after processing the audio data, the audio sample-size, channel configuration and other fields are updated accordingly. Please see *What steps are required to properly implement an Apply Function?*

When is it called?

The PA Framework calls the *_apply()* function for each ASP algorithm in the ASP chain. The ASP algorithms are called after the decoding operation. The generated audio data after all the ASP *_apply()* functions have completed the processing is passed to the encoder.

What is it supposed to do?

The *ALG_VEN_apply()* function is invoked by the PA/F to pass status, control, and audio data to the ASP algorithm. The algorithm uses the status information to understand the processing that needs to be applied. Most of the ASP algorithms use the status information to understand how they should modify the audio data. But ASP algorithms can be developed that may use the status information to perform certain other actions and not necessarily modify the audio data, but such occurrences are rare and not discussed here.

One of the first checks that an ASP algorithm does is to decide whether or not to process the incoming signal. This test is performed in the *ALG_VEN_apply()* function and consists of checking one or more of the following:

- mode control register: whether the mode is enabled or not!
- channel configuration: whether it's allowed/required to operate or not!
- other register(s) as appropriate

After processing, the algorithm must ensure that both the channel configuration and the audio size are updated accordingly.

What steps are required to properly implement an Apply Function?

The following are the important steps that need to be followed by all ASP Algorithms.

- **Proper usage of the mode control register.**

The first element of the algorithm's status structure must be `mode`. The `mode` variable is used to enable or disable the processing of the algorithm. The algorithm must check the `mode` variable as one of the first steps in the *_apply()* function. If the `mode` is disabled, the algorithm should not modify the audio samples or any quantities in the *AudioFrame* structure.

- **Proper testing of the sample rate.**

If the processing within the ASP algorithm is dependent on the sample rate of the audio stream, the algorithm must read and react to the current sample rate from the `AudioFrame`. If the current sample rate is different than the previous sample rate, the algorithm may have to make a configuration change or reset certain processing states. The algorithm may choose to have a separate function to perform this test and sample-rate dependent operation. Such a function can be called by the *_apply()* function as well as the *_reset()* function, if necessary.

- **Check for and react to the Audio Stream's Channel Configuration information.**

The ASP algorithms must check the `channelConfigurationStream` register within the *AudioFrame* to understand the audio channels that are presented to it.

- **Proper setting of the Audio sample-size register (`samsiz`).**

The Audio sample-size is an indication of the magnitude of the audio sample data on each channel. The `sasiz` variable is used by the ASP algorithm to indicate the magnitude of the audio data on each channel after the processing is completed. Details on `samsiz` usage is available in Audio sample-size

- **Updation of sample-process register (`PAFProcess`).**

The sample-process register contains a multi-byte bit mask of values of the form `(1<<PAF_PROCESS_X)` that indicates the algorithms that have operated on the *AudioFrame* data. It is a read/write quantity, and it is the responsibility of an algorithm to update this register if appropriate. Please refer to `T:/pa/f/include/pafsp.h` to understand how the various bits of this register are defined. Custom ASP algorithms may also use this register to inform whether the ASP operated or not.

- **Proper setting of sample-rate.**

The ASP algorithms that change the sample rate must update the `sampleRate` information in the *AudioFrame*. If the algorithm performs downsampling or upsampling, then the `sampleCount` variable will also need to be modified.

- **Proper setting of Channel Configuration register (`channelConfigurationStream`).**

If the ASP algorithm processing changes the channel configuration, the information must be updated in the `channelConfigurationStream` register within the *AudioFrame* structure. For example, Surround Processing ASP algorithms must check the input channel configuration by checking `channelConfigurationStream` towards the begining of the processing. It also must check the requested channel configuration by checking the `channelConfigurationRequest` register in the *AudioFrame* structure to determine if surround processing is required. If the processing is required and performed, the algorithm must modify the `channelConfigurationStream` variable to indicate the resulting channel configuration of the stream.

Description of ALG_VEN_reset( )

What parameters are passed to it?

The *_reset()* function for all ASP algorithms must have the below interface:

```
Int ALG_VENDOR_reset(IALG_Handle handle, PAF_AudioFrame *pAudioFrame)
```

Where:
`IALG_Handle handle` -> ASP Algorithm handle
`PAF_AudioFrame *pAudioFrame` -> Pointer to the *AudioFrame* structure.

When is it called?

The *ALG_VEN_reset()* function is called before the decode processing is started. If the decode processing is stopped and restarted (e.g., when there is change in the input bitstream or the alpha code `writeDECCommandRestart` is issued) the *ALG_VEN_reset()* function of all the ASP algorithms in the ASP chain is called before the decode processing restarts.

What is it supposed to do?

The *ALG_VEN_reset()* function is invoked by the PA/F to pass status and control data to the ASP chain. The algorithms should use the control data to understand the initialization required to be done before the *_apply()* function is called. For example, based on the sample-rate information in the *AudioFrame* structure, different coefficients may need to be loaded for the filters. Also, clearing of state variables used in the ASP algorithm processing may need to be performed.

The complexity of having a separate function that processes control data without sample data might seem high, but has certain advantages. The main benefit of having this separate function is realized by being able to perform MIPS-intensive processing outside of the normal timing constraints that are imposed when sample data is being processed. For example, large delay buffers can be cleared or certain other computations may be performed as part of the *ALG_VEN_reset()* function. If performed in the *ALG_VEN_apply()* function, such operations would negatively impact the performance of the real-time processing on the audio data.

The ASP algorithms are passed control and status information as part of *ALG_VEN_reset()* function invocation. The ASP algorithms may modify some of the control and status information. This provides a method by which ASP can pass back information to the framework.

Description of ALG_VEN_control( )

The ALG_VEN_control function allows the framework to connect the Alpha Code messaging mechanism to the algorithm. The control function must always provide an algorithm-specific ALG_GETSTATUSADDRESS1 command that returns the address of the status register inside the Algorithm. This function has common functionality across all ASPs and is implemented in COM_TII_control function in com_asp.lib library available in SDK. Individual ASPs can use this function by defining:

```
#define ALG_VEN_control COM_TII_control
```

DA8xx Alpha Codes

Introduction

The PA system utilizes the PA messaging (PA/M) protocol for communication between different parts of the system. The communication is achieved by means of word-based units, where each word is a 16-bit unsigned integer of the form *0xhhhh*, and is called an *alpha* code word. A single operation, such as writing a value to a specific register, usually involves sequence of two or more of these words. An *alpha* code word sequence, also referred to simply as *alpha* code, is represented symbolically by assigning a name that clearly describes the operation performed by the sequence.

This chapter introduces the basics of selecting *alpha* code symbol names and creating the corresponding *alpha* code words for an ASP. It is only meant to serve as an introduction and utilizes the Equalizer ASP (EQU) as an example.

Status structure

Each ASP is required to have a status structure that contains the elements that can be read or modified while it is running. The general form of this status structure is:

```
typedef volatile struct ALG_Status {
   Int size;             /* This value must always be here, and must be
 set to the
                            total size of this structure in 8-bit
bytes, as the
                            sizeof() operator would do. */
    /* Implementation-specific structure elements.  */
} ALG_Status;
```

where ALG is a 2- or 3-character name that you give to your algorithm. Such a status structure conceptually serves as a memory-mapped register bank. The operation of the ASP depends on the contents of each of these registers. While the ASP is running, the PA/M protocol accesses these registers using *alpha* code sequences.

Example: Status structure for Equalizer example ASP

The source files of Equalizer (EQU) example are provided in the folder `t:\pa\asp\aspdk\equ`. The status structure is given in the file `iequ.h` and is reproduced below:

```
/*
*  ======== IEQU_Status ========
*  This Status structure defines the parameters that can be changed or
read
*  during real-time operation of the algorithm.  This structure is
actually
*  instantiated and initialized in iequ.c.
*/
typedef struct IEQU_Status {
   Int size;             /* This value must always be here, and must be
 set to the
                            total size of this structure in 8-bit
bytes, as the
                            sizeof() operator would do. */
   XDAS_Int8 mode;
   XDAS_Int8 reset;
   XDAS_Int16 unused;
   XDAS_Int8 bandGain[10];
} IEQU_Status;
```

Communication with the EQU algorithm involves reading from or writing to the `mode`, `reset` and `bandGain` registers. Each of these are **control registers** as their values can be altered with write operations. The read and write operations are accomplished by means of *alpha* codes.

Alpha Code Symbol Names and Alpha Code Sequences

The status structure described in the previous section contains the elements that will be accessed by PA/M. You need to specify the *alpha* code symbol names and the corresponding *alpha* code sequences that will be used by this communication mechanism. The entire process consists of five steps:

1. Select a 2 or 3 character name for your ASP algorithm.
2. Define a status structure for your ASP.
3. Select *alpha* code symbol names.
4. Determine the *alpha* code sequence corresponding to each of the *alpha* code symbol names.
5. Create a header file containing definitions that map each of the *alpha* code symbol names.

Step1 :Select ASP name

Select a 2 or 3 character name for your ASP algorithm. This name must be unique to your ASP. To determine the names that already exist as part of the SDK that you are using, see the master *alpha* code symbol file, for example: `P:\alpha\pa_i14_evmda830_io_a.hdM`. This header file contains a number of `#include` statements, each corresponding to a unique header file for an ASP that already exists as part of the SDK. The names of the files being included will provide a good indication of the algorithm names already in use.

Step2 :Define a Status Structure for the ASP

The status structure contains the elements to be accessed while communicating with the ASP. In addition to defining the status structure, make a note of the size(in bytes) of each element in the status structure and from this information, calculate the corresponding **offset** (in bytes). These values are needed when selecting the *alpha* code type (explained later in this chapter) and determining the code words assigned to each *alpha* code symbol name.

Example: Size and Offset of elements in Status Structure

For the EQU, the status structure is `IEQU_Status` and the specific elements to be accessed are `mode`, `reset` and `bandgain`. The *size* and *offsets* of these elements are shown in *Table1 (Size and Offset of Elements in the IEQU_Status Structure)*. The *offset* of any element is given by the cumulative sum of the size of all the elements preceding it.

Table 1 Size and Offset of elements in the IEQU_Status structure

Element	Register size(bits)	Register size(bytes)	Offset(bytes)
Size	32	4	0
Mode	8	1	4
Reset	8	1	5
Unused	16	2	6
bandGain[10]	80	10	8

Step3: Select Alpha Code Symbol Names

A standard nomenclature has been developed for selecting the names of the *alpha* code symbols. All *alpha* code symbol names must adhere to one of the following seven forms:

1. read*ALGRegisterName*
2. write*ALGRegisterName*
3. write*ALGRegisterNameValue*
4. write*ALGRegisterName(N)*
5. wrote*ALGRegisterName*
6. wrote*ALGRegisterNameValue*

where the constructs like *ALG*, *RegisterName*, and *Value* should be replaced with the algorithm name (e.g. EQU), the name of an element of the *ALG_Status* structure and an appropriate value respectively. Alpha code symbol names that begin with the word *wrote* serve as responses to *read* operations. For these *read* operations, *write* symbol names do not exist. The next step describes how to determine the bit patterns that correspond to the *alpha* code word sequences to be assigned to the *alpha* code symbol names selected in this step.

Example: Alpha Code Symbol Names

For the EQU ASP, the *ALG* name is EQU, and the *RegisterName* is `mode`, `reset` and `bandGain`. Accordingly, the following *alpha* code symbol names have been chosen:

1. `readEQUMode` - to read the value in the mode register
2. `writeEQUModeDisable` - to write the value 0 in the mode register
3. `writeEQUModeEnable` - to write the value 1 in the mode register
4. `readEQUSpare` - to read the value in the spare (unused) register.
5. `writeEQUSpare(N)`- to write the value N in the spare register
6. `writeEQUBandGain(N0,N1,N2,N3,N4,N5,N6,N7,N8,N9)` - to write the values No, N1 etc in each `bandGain` register respectively.

Step 4: Assign Alpha commnads to the Alpha Code Symbol Names.

Communication using *alpha* codes is achieved by means of word-based units where each word is a 16-bit unsigned integer of the form 0xhhhh. The *alpha* codes corresponding to a multi-word sequence are transmitted (or received) with the least-significant word first. This is an important point to keep in mind when assigning the 16-bit code words to the *alpha* code symbol names.

Example: Alpha Code Word Sequences for EQU Example

The *alpha* code word sequences for the EQU example are defined in the file: `t:\pa\asp\aspdk\equ\equ_a.h` as

```
#define readEQUMode            0xf200+CUS_BETA_EQU,0x0400
#define writeEQUModeDisable    0xfa00+CUS_BETA_EQU,0x0400
```

```
#define writeEQUModeEnable      0xfa00+CUS_BETA_EQU,0x0401
#define readEQUReset            0xf200+CUS_BETA_EQU,0x0500
#define writeEQUResetDisable    0xfa00+CUS_BETA_EQU,0x0500
#define writeEQUResetEnable     0xfa00+CUS_BETA_EQU,0x0501
#define readEQUSpare            0xf300+CUS_BETA_EQU,0x0600
#define writeEQUSpare(N)
0xfb00+CUS_BETA_EQU,0x0600+((N)&0xff)
#define readEQUStatus           0xf508,0x0000+CUS_BETA_EQU
#define readEQUControl          readEQUStatus
#define readEQUBandGain         0xf600+CUS_BETA_EQU,0x080a
#define writeEQUBandGain(N0,N1,N2,N3,N4,N5,N6,N7,N8,N9) \

0xfe00+CUS_BETA_EQU,0x080a,TWOUP(N0,N1),TWOUP(N2,N3),TWOUP(N4,N5),TWOUP(N6,N7),TWOUP(N8,N
```

Consider the definition for `readEQUMode`: `#define readEQUMode 0xf200+CUS_BETA_EQU, 0x0400`. The sequence corresponding to the symbol name `readEQUMode` consists of two words; 0xf200 and 0x0400. The first word transmitted (received) is the least-significant word, 0xf200+CUS_BETA_EQU. This is then followed by 0x0400.

An *alpha* code sequence may consist of two or more words. The operation to be performed by the sequence is determined by bit patterns for each of the words that make up the sequence. The description of the bit fields of the least-significant word is common to all sequences. These bit fields are shown in *Figure 1 (Bit Fields in the Least-Significant Word of an Alpha Code Word Sequence)* and described in *Table 2 (Description of Bit Fields in the Least-Significant Word of an Alpha Code Word Sequence).*

Figure 1: Bit Fields in the Least-Significant Word of an Alpha Code Word Sequence

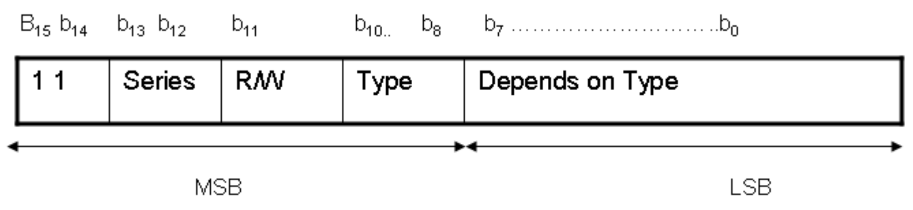
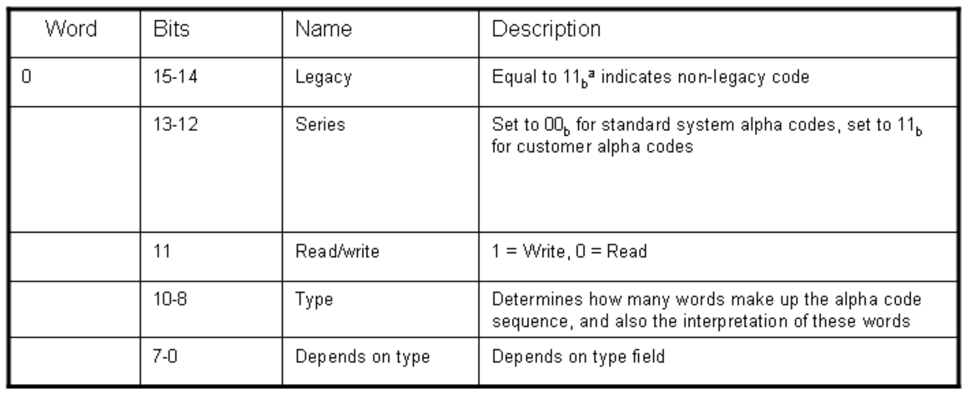


Table 2: Description of Bit Fields in the Least-Significant Word of an Alpha Code Word Sequenc



Word	Bits	Name	Description
0	15-14	Legacy	Equal to 11_b[a] indicates non-legacy code
	13-12	Series	Set to 00_b for standard system alpha codes, set to 11_b for customer alpha codes
	11	Read/write	1 = Write, 0 = Read
	10-8	Type	Determines how many words make up the alpha code sequence, and also the interpretation of these words
	7-0	Depends on type	Depends on type field

a. The subscript b is used to denote binary, or base 2, numbers.

- The *Legacy* field, bits 14 and 15, should be set to 11_b.
- The *Series* field, bits 12 and 13, should be set to 11_b.

- The *R/W (Read / Write)* field, bit 11, is quite straight forward. Set this field to 0 for a read operation, and 1 for a write operation.
- The *Type* field, bits 8 to 10, determines the *alpha* code type. In this section we will focus on four types which are types 2, 3, 5 and 6.

Type 2 (b10:b8 = 010) Alpha Code

The *Table 4(Description of Type 2 Alpha Codes)* shows the number of words required in the *alpha* code sequence and also the description for each of these words for Type 2 *alpha* codes. A Type 2 *alpha* code has bits 10:8 of word 0 set to 010b which indicates the *length (=2)* of this *alpha* code.

Table 4 Description of Type 2 Alpha Codes

Type	Word	Byte	Description
2	0	MSB	As shown in Figure 1 and Table 2
-	-	LSB	Beta Unit Number as defined in cusbeta.h
-	1	MSB	Offset of register to be read or written
-	-	LSB	Data for write operation,unused for read operation

For all types of *alpha* codes, the 8 bits in the MSB of word 0 always correspond to the *Legacy*, *Series*, *Read/Write* and *Type* fields as shown in *Figure 1* and *Table 2*. For a Type 2 *alpha* code, the LSB of word 0 corresponds to the Beta Unit number. The Beta Unit number is a unique identifier for an ASP. For example, the Beta unit number for the EQU ASP is 0x00, and is defined in the file `cusbeta.h` as follows:

```
#define CUS_BETA_EQU  0x00
```

The file *cusbeta.h* should be modified accordingly to specify the Beta unit number that you want to assign to your ASP. Type 2 *alpha* codes are used to read from or write to 8-bit registers. The offset of the register to be accessed corresponds to the MSB of word 1. For a write operation, the 8-bit data value is given in the the LSB of word 1.

Example: EQU example - `readEQUMode`

The *alpha* code symbol name `readEQUMode` is used to read the contents of the 8-bit `mode` register. It falls into the category of a Type 2 *alpha* code and requires a total of two words. Thus word 0 of the *alpha* code sequence for *readEQUMode* is equal to 0xf200 and is shown in *Figure 1 (Word 0 of the Alpha Code Word Sequence for readEQUMode).*

The MSB corresponds to the Legacy, Series, Read/Write and Type fields, whereas the LSB(=00h) is the Beta unit number as defined in the file `cusbeta.h`. The word 1 of the *alpha* code is equal to 0x0400 and is shown in *Figure 2(Word 1 of the Alpha Code Word sequence for readEQUMode)*. The MSB(=04) is the offset of the `mode` register as obtained from *Table 1*. Since the LSB is not used for a read operation, it is assigned the value 0x00.

Figure 2: Word 0 of the Alpha Code Word Sequence for readEQUMode

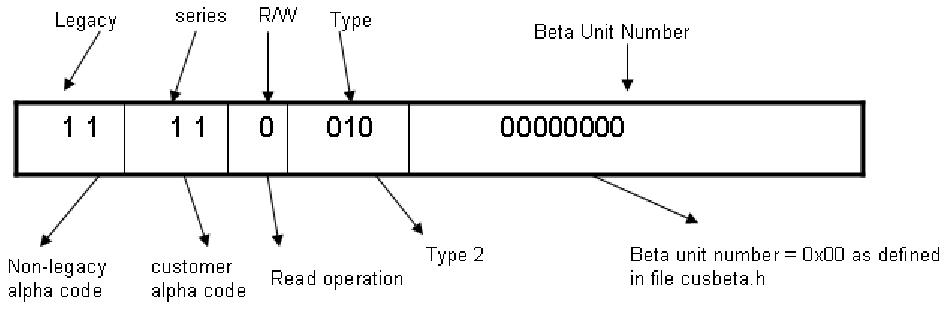
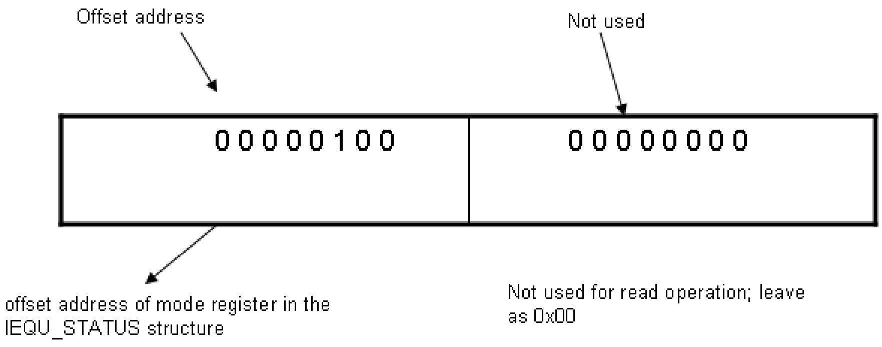


Figure 3: Word 1 of the Alpha Code Word Sequence for readEQUMode


Offset address
Not used
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
offset address of mode register in the IEQU_STATUS structure
Not used for read operation; leave as 0x00

Type 3 (b10:b8 = 011) Alpha Code

The *Table 5 (Description of Type 3 Alpha Codes)* shows the number of words required in the *alpha* code sequence and also the description for each of these words for Type 3 *alpha* codes. A Type 3 *alpha* code has bits 10:8 of word 0 set to 011b which indicates the *length (=3)* of this *alpha* code.

Table 5 Description of Type 3 Alpha Codes

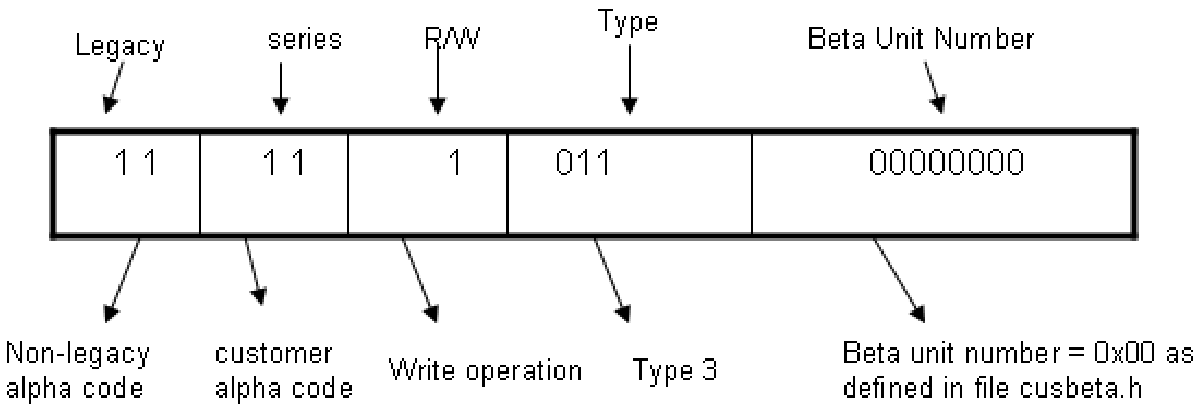
Type	Word	Byte	Description
3	0	MSB	As shown in Figure 1 and Table 2
-	-	LSB	Beta Unit Number as defined in cusbeta.h
-	1	All	Offset of register to be read or written
-	2	All	Data for write operation,unused for read operation

Similar to Type 2 *alpha* codes, the LSB of word 0 for Type 3 *alpha* codes also corresponds to the Beta unit number. The *offset* of the register to be accessed corresponds to word 1. Type 3 *alpha* codes are used to read from or write to, 16-bit registers. The 16-bit data is specified as word 2 for a write operation. The word 2 is not used for a read operation.

Example: EQU Example- `writeEQUSpare`

The *alpha* code `writeEQUSpare` is used to write to the 16-bit `unused` register. It falls into the category of a Type 3 *alpha* code and requires a total of 3 words. As specified in *Table 2 (Description of Bit Fields in the Least-Significant Word of an Alpha Code Word Sequence)*, word 0 of the *alpha* code word sequence for `writeEQUSpare` is equal to 0xfb00 and is shown in *Figure 4*. The MSB corresponds to the Legacy, Series, Read/Write, and Type fields, whereas the LSB is the Beta unit number.

Figure 4: Word 0 of the Alpha Code Word Sequence for writeEQUSpare



The word 1 of the *alpha* code contains the *offset* address, and is equal to 0x0006. The offset address was calculated in *Table 1*. The word 2 of the *alpha* code is the 16-bit data to be written. Thus word 2 is equal to 0xhhhh where 0xhhhh corresponds to the hexadecimal representation of the data.

Type 5 (b10:b8 = 101) Alpha Code

The *Table 6(Description of Type 5 Alpha Codes)* shows the number of words required in the *alpha* code sequence and also the description for each of these words for Type 5 *alpha* codes. A Type 5 *alpha* code has bits 10:8 of word 0 set to 101b. The length of a Type 5 *alpha* code could be anywhere from 2 to N words.

Table 6 Description of Type 5 Alpha Codes

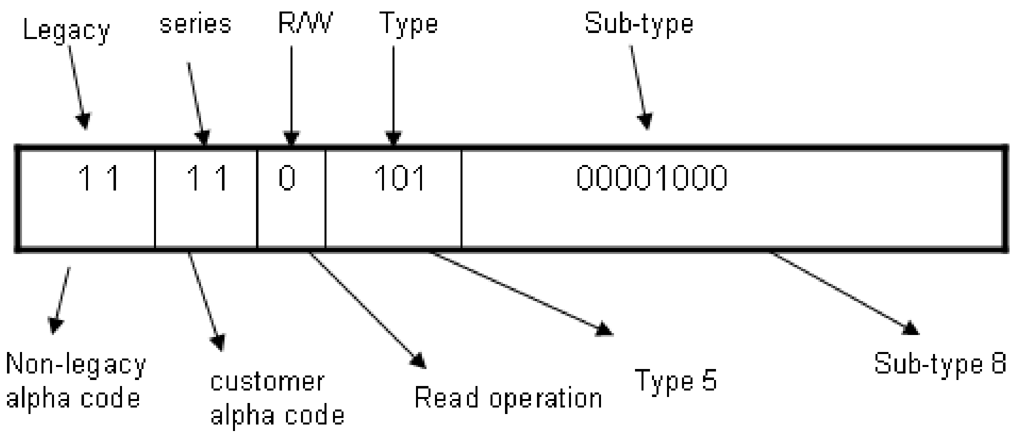
Type	Word	Byte	Description
5	0	MSB	As shown in Figure 1 and Table 2
-	-	LSB	Extended alpha code filed
-	1-N	All	Various

For a Type 5 *alpha* code, the LSB of word 0 *does not* contain the Beta unit number. Instead, it indicates a sub-type. A listing of all possible extended *alpha* code sub-types is given in *Table 12 of the Performance Audio Messaging Application Protocol Application Report*. In this section, the discussion is limited to sub-type 8.

Example 6: EQU Example - `readEQUStatus`

The *alpha* code word corresponding to `readEQUStatus` is of Type 5-8. This command is used to read the contents of the entire `IEQU_Status` structure. The MSB of word 0 of the *alpha* code is as specified in *Table 2(Description of Bit Fields in the Least-Significant Word of an Alpha Code Word Sequence)*. The LSB of word 0 specifies the sub-type, i.e. 8. Thus word 0 is equal to 0xf508 and is shown in *Figure 5 (Word 0 of the Alpha Code Word Sequence for readEQUStatus)*.

Figure 5: Word 0 of the Alpha Code Word Sequence for readEQUStatus



The word 1 of the *alpha* code is the Beta unit number and is equal to 0x0000. This particular *alpha* code word sequence of Type 5-8 requires a total of two words.

Type 6 (b10:b8 = 110) Alpha Code

The *Table 7(Description of Type 6 Alpha Codes)* shows the number of words required in the *alpha* code sequence, and also the description for each of these words, for Type 6 *alpha* codes. A Type 6 *alpha* code has bits 10:8 of word 0 set to 101b. The length of a Type 6 *alpha* code could be anywhere from 2 to N words. **Table 7 Description of Type 6 Alpha Codes**

Type	Word	Byte	Description
6	0	MSB	As shown in Figure 1 and Table 2
-	-	LSB	Beta Unit Number as defined in cusbeta.h
-	1	All	Offset of register to be read or written
-	2	All	Offset for how many variables to be read/written
-	2-N	various	value to be wriiten, unused for read

Data is read from the Beta Unit at the base address indicated by the 8-bit beta field and the 8-bit offset in bytes given by the gamma field. The number of bytes read is given by the 8-bit kappa field. It may also be referred to as a variable-length read. The return value is an *alpha* code type 6 write with the appropriate length and data. The length of an *alpha* code type 6 read is 2 words.

Data is written to the Beta Unit at the base address indicated by the 8-bit beta field and the 8-bit offset in bytes given by the gamma field. The number of bytes written is given by the 8-bit kappa field. It may also be referred to as a variable-length write. The length of an *alpha* code type 6 write is 2+(k+1)/2 words. The return value is null.

Step 5: Create a header file *alg_a.h*

The final step required is to create a header file *alg_a.h* and place it in the folder `T:\pa\asp\alg\alpha\` where *alg* is the name for your algorithm. The header file contains definitions that map each of the *alpha* code symbol names from step-3 with the corresponding *alpha* code sequence from step-4. The header file will contain one or more definitions of the form

```
#define  readALGRegisterName 0xhhhh …
#define writeALGRegisterName 0xhhhh …
#define writeALGRegisterNameValue  0xhhhh …
#define writeALGRegisterName(N)  0xhhhh …
#define wroteALGRegisterName  0xhhhh …
#define wroteALGRegisterNameValue  0xhhhh …
```

where

- 0xhhhh specifies the binary values, represented as a 16-bit word, assigned to the *alpha* code symbol name. The symbol names were determined in step 3, and the corresponding 16-bit values were calculated in step 4.
- the ellipses, …, specify that more than one 16-bit word may be assigned to the *alpha* code symbol name. This is usually the case.

For eg: the *alpha* header file for equ example has these defined as described in Example 5 like:

```
#define readEQUMode 0xf200+CUS_BETA_EQU,0x0400
#define writeEQUModeDisable 0xfa00+CUS_BETA_EQU,0x0400
...
```

DA8xx ASP Porting Guide

Introduction

The term Audio Stream Processing (ASP or also popularly known as post-processing) operates on the audio data following decoding and preceding encoding in Performance Audio Framework. The audio stream processing is implemented via a collection of individual XDAIS algorithms, each with a common, standardized interface. This document provides a guidance to the ASP developer as how custom algorithm can be integrated into PA/F. This chapter also describes common settings and recommendations for using an ASP within PA framework. Some familiarity with the XDAIS standard is assumed. Also, it's assumed that the developer has already gone through the chapter that describes the ASP interface to PA/F.

Integrating ASP into PA/F

It's assumed that the reader already have installed PA/F SDK for DA8xx device and gone through the Getting Started Guide, set-up the build environment and DA8xx EVM.

Step1: Create work area

- Unzip firmware deliverable from PA/F SDK to an empty folder preserving the path information.
- Unzip ASP Development Kit deliverable to the same folder preserving the path information, letting it overwrite existing files, if any.
- Map the folder as T: drive such that the newly created `pa` folder falls in the root of T: drive.

Step2: Generate PA/F layer using pag.exe

- Move to `aspdk` folder

```
$ cd t:/pa/asp/aspdk
```

- Modify `example_asp.bsp` for the ASP name, vendor name and any other relevant information.
- Run `pag` utility

```
$ ./pag.exe -g asp example_asp.bsp
```

This will create a folder `asp` with:

- all necessary PA/F interface files
- example ASP wrapper that can be used with *VC* build and *CCS* unit testing
- Move `asp` folder to folder for your custom asp development and do file rearaangement. Below example shows `sur` as the custom asp name and folder.

```
$ mv asp ../sur
$ cd ../sur
$ mkdir alpha
$ cp sur_a.h alpha/
$ cp main.c ../
```

Step3: Customize PA/F layer code

- Customize `asp_ven_ialg.c` file for any additional memory allocation/intialisation. Note that all memory allocations have to happen in `asp_ven_alloc()` function through `memTabs`. The library should not have any inside malloc/free calls. This is required to make library re-entrant.
- Customize `asp_ven_iasp.c` for the custom asp with calls to other functions implementing the functionality and other necessary settings of the custom algorithm.

Step4: Create and build CCSv4 project

- This can be done in CCSv4 GUI or through command line. Please see CCSv4 help for first option. Command line way to create/build project is explained here.
- Following commands creates project `sur` in directory `t:/pa/asp/sur` in CCSv4 workspace `c:/workspace_asp`. When no configuration is specified; this creates *Debug* and *Release* configurations with *Release* configuration set to optimised options *-o3* and *no-Debug*. Please add any additional options/include files during project creation. The script `createproject.sh` in the package also includes these commands. Running this script will generate and build the CCSv4 project. In order to include any additional options please

update the script.

More information on command line way to create/build project is available in [1]

```
$ cp ../aspdk/createproject.sh .
$ ./createproject.sh
```

or

```
$ CCS4_DIR="C:/Program Files/Texas Instruments/ccsv4"
$ C6000_CG_ROOT="C:/Program Files/Texas
Instruments/ccsv4/tools/compiler/c6000"
$ BIOS_CG_ROOT="C:/Program Files/Texas Instruments/bios_6_21_00_13"
$ XDC_CG_ROOT="C:/Program Files/Texas Instruments/xdctools_3_16_02_32"
$ XDAIS_CG_ROOT="C:/Program Files/Texas Instruments/xdais"
$ PROJ_PATH="t:/pa/asp/aspdk/sur"
$ PROJ_NAME=sur
$ rm -rvf T:/pa/asp/aspdk/sur/{.[cps]*,Release}
c:/workspace_asp/.metadata
$ "${CCS4_DIR}"/eclipse/jre/bin/java -jar
"${CCS4_DIR}"/eclipse/startup.jar -data  c:/workspace_asp \
$ -application com.ti.ccstudio.apps.projectCreate \
$ -ccs.name "${PROJ_NAME}" \
$ -ccs.location "${PROJ_PATH}" \
$ -ccs.device com.ti.ccstudio.deviceModel.C6000.GenericC674xDevice \
$ -ccs.kind com.ti.ccstudio.managedbuild.core.ProjectKind_StaticLibrary
 \
$ -ccs.endianness little \
$ -ccs.artifactName "${PROJ_NAME}" \
$ -ccs.artifactExtension lib \
$ -ccs.cgtVersion 6.1.13 \
$ -ccs.setBuildOption \
$ -ccs.outputFormat coff \
$ -ccs.rts rts6740.lib \
$ -ccs.setCompilerOptions "-mv6740 --symdebug:none -o3" @configurations
 Release \
$ -ccs.setCompilerOptions "-mv6740 -g -o0" @configurations Debug \
$ -ccs.setCompilerOptions "-I t:/pa/f/include -I T:/pa/dec/com -I
t:/pa/f/alpha -I t:/pa/f/s3 -I t:/pa/asp/com -I t:/pa/asp/std -I
t:/pa/sio/acp1 -I ${PROJ_PATH} -I ${PROJ_PATH}/alpha -I
${BIOS_CG_ROOT}/packages/ti/bios/include -I ${BIOS_CG_ROOT}/packages -I
 ${XDC_CG_ROOT}/packages -I ${C6000_CG_ROOT}/include -I
${XDAIS_CG_ROOT}/include -I ${XDAIS_CG_ROOT}/src/api -I
T:/pa/asp/fil/alg -I T:/pa/asp/fil/src"
$ -ccs.overwrite keep
```

- Build the project using below command. This builds the library in *Release* configuration.

```
$ "${CCS4_DIR}"/eclipse/jre/bin/java -jar
"${CCS4_DIR}"/eclipse/startup.jar -data c:/workspace_asp -application
com.ti.ccstudio.apps.projectBuild -ccs.projects "${PROJ_NAME}"
```

```
-ccs.configuration Release
```

Step5: Add Alpha codes

- Customize the alpha header file generated by `pag` utility (e.g: `sur_a.h`) for your applcaition.

Details on alpha codes and how to write them is explained in Alpha Codes.

Step6: Adding custom ASP to the ASP chain

- Add paths to the source files for custom ASPs in the final PA project.
  - Example: add `sur.lib` to `pa_i14_evmda830` project
- Include custom ASP header files (ASP.h and ASP_VEN.h) in patchs.c .
  - Example: add `#include sur.h #include sur_tii.h` to `pa/f/s19/i14/patchs.c`
- Add custom ASP in the PA/F ASP chain by including an appropriate PAF_ASP_LINKINIT() macro to PAF_ASP_LinkInit table in patchs.c.
  - Example: add `PAF_ASP_LINKINIT (CUS, SUR, TII),` macro to `aspLinkInitCusI14` table and `aspLinkInitAllI14` table
  - Note:Location of custom ASP algorithm in the Customized Audio Stream processing chain is specified by the location of PAF_ASP_LINKINIT() macro in patchs.c.So add this line depending on the desired location of your ASP in the ASP chain. For example if location of ASPs are specified like below; then SUR ASP comes before EQU ASP in the PA/F ASP chain:

```
const PAF_ASP_LinkInit aspLinkInitCusI14[] =
{
   PAF_ASP_LINKINIT (CUS, SUR, TII),
   PAF_ASP_LINKINIT (CUS, EQU, TII),
   PAF_ASP_LINKNONE,
};
```

- Note that:
  - All surround processing ASPs are grouped together in the ASP chain.
  - Custom surround processing ASPs, such as SUR are first.
  - Standard surround processing ASPs, such as PL2x, come next.
- Add an include statement for the custom alpha header file to `P:\i14_a.h`
- Add apropriate alpha code inverse symbol definitions to `P:\i14_a.hdm`. This enable use of the defined alpha command directly.
  - Example:sur_a.h includes defintions like below:

```
#define  readSURMode            0xf200+CUS_BETA_SUR,0x0400
#define writeSURModeDisable     0xfa00+CUS_BETA_SUR,0x0400
#define writeSURModeEnable      0xfa00+CUS_BETA_SUR,0x0401
 Assuming that CUS_BETA_SUR is defined to 0x00, add below definitions
to i14_a.hdm
#define readSURMode 0xf200,0x0400
#define writeSURModeDisable 0xfa00,0x0400
#define writeSURModeEnable 0xfa00,0x0401
```

Miscellaneous Information

Communicating with other algorithms

Sometimes it may be required to communicate with other algorithms to query status or issue commands. Especially when there are more than one custom algorithms related to each other.

Following method can be used if such a communication is desired:

- Create an ACP instance.
  - Required only once, before any communication is made:

```
#include <acp.h>
#include <acp_mds.h>

static ACP_Handle acp = NULL;

    ACP_MDS_init ();
    acp = ACP_create (&ACP_MDS_IACP, &ACP_PARAMS);
    if (!acp)
        return;
```

- To query status, send alpha code and get response.
  - This example shows type-2 read:

```
#include <pafdec_a.h>

    ACP_Unit from[] = {0xc902, readDECSourceProgram};
    ACP_Unit to[5];
    int program = 0;

    if (acp->fxns->sequence (acp, from, to))
        return;
    program = to[2] & 0xFF;
```

- To issue command, send alpha code
  - This example shows type-2 write:

```
#include <pafenc_a.h>

    ACP_Unit from[] = {0xc902, writeENCCommandMute};
    ACP_Unit to[5];

    if (acp->fxns->sequence (acp, from, to))
        return;
```

References

[1] http://tiexpressdsp.com/index.php/Projects_-_Command_Line_Build/Create

DA8xx PA/F Integration tips

Audio sample-size

The audio sample-size is an indication of the magnitude of the audio sample data on each channel. The `samsiz` variable is used by the ASP Algorithm to indicate the magnitude of the audio data on each channel. For algorithms like Bass Management, the magnitude of the SUB (subwoofer) channel can exceed 0 dB. The ASP Algorithm must provides the audio sample-size information so that the audio data samples can be adjusted during the encode process (conversion from floating-point to fixed-point) to ensure that no clipping of audio data occurs. The unit of the audio sample-size register is in 0.5 dB step. The audio sample-size information should not be written directly, but appended to the already present sample-size for each channel. If the audio level generated by ASP Algorithm is expected to never exceed 0 dB and an external entity (e.g., a microcontroller) adjusts the DSP's output levels to ensure that no clipping occurs, then setting the audio sample-size register in the ASP Algorithm is not necessary.

How to set the audio sample-size register?

The `*samsiz` element of the `AudioFrameData` structure is a pointer to an array of size M, where M is the maximum number of channels.

```
samsiz = pAudioFrame->data.samsiz; // Audio size
```

Each element of the array is an audio sample-size register that is used to represent the theoretical maximum value that an audio channel can reach after the ASP has operated on it. The audio sample-size registers represent values in decibels in units of 0.5 dB. That is, the audio size registers are in Q1 format. For example, the bit pattern 0x0004 represents the value 2 dB.

Example1: In EQU example, max gain applied to all the channels is tracked and samsiz is updated accordingly so that no attenuation happens in analog domain. For all available channels, the samsiz is set as: For channel i; the audio size is set as:

```
samsiz[i] += 2 * 20 *log10 (maxgain);
```

The multiplication by 2 is because the audio size represetation is in Q1 format; ie, in units of 0.5dB.

Example2: Generating Centre channel from Left and Right channels: Denoting the `i`th sample on the left, center, and right channels as `left[i]`, `cntr[i]`,and `rght[i]` respectively, the center channel is calculated as:

```
 cntr[i] = ccScale * (left[i] + rght[i]);
```

where `ccScale` is a scale factor. Assuming that the `i`th sample on the left and right channels, `left[i]` and `rght[i]`, have the same maximum possible value (size), then the corresponding sample for the center channel `cntr[i]`, can be twice as large as `left[i]` (or `rght[i]`). The value of "twice" is equivalent to 6 dB. The multiplication of 6 by the value 2, resulting in:

```
samsiz[PAF_CNTR] = 2*6 + samsiz[PAF_LEFT];
```

Recommendation for Memory usage

Placement of buffers in IRAM/L3RAM/SDRAM

DA8xx device DSP internal memory consists of L1P, L1D and L2 cache memory. The internal memory configuration is:

- L1P memory includes 32KB of RAM. The DSP program memory controller allows to configure part or all of the L1P RAM as normal program RAM or as cache.
- L1D memory includes 32 KB of RAM. The DSP data memory controller (DMC) allows to configure part of the L1D RAM as normal data RAM or as cache.
- L2 memory includes 256 KB of RAM. The DSP unified memory controller (UMC) allows to configure part or all of the L2 RAM as normal RAM or as cache.
- L2 memory also includes 1024 KB of ROM.

This device also offers an on-chip 128-KB shared RAM, apart from the internal memories. This shared RAM is referred to as L3RAM in PA/F context. Access to L3RAM happens via cache. This device has two external memory interfaces that provide multiple external memory options accessible by the CPU and master peripherals:

- EMIFA: 16-bit SDRAM with 128-MB address space
- EMIFB: 32-/16-bit SDRAM with up to 256-MB SDRAM address space

See the RAM usage report in SDK for information on the memory usage in each feature set. This should give a feel of how much of IRAM/L3RAM is free for use by custom ASPs. Since the device has cache based architecture, it is recommended to make use of L3RAM and SDRAM wherever possible except for performance critical applications. It is also recommended to set the `memTab` parameters as configurable via the `IAlg_Config` structure; so that placement of buffers can be changed init-time.

Scratch Memory Usage

ASPs may use some scratch memory whose contents need not be maintained across the decode processing calls. Hence this memory can be shared among other algorithms active in the system. The PA/F allocates a scratch memory equal to the maximum scratch requirement of all algorithms present in the chain (This is approx 8K in the PA SDK, see memory footprint of each featureset for exact information). If scratch requirement of the custom ASP falls within this maximum; no additional scratch allocation happens for the custom ASP, otherwise the maximum scratch allocated is as per the custom ASP. Setting `memTab` attribute to `IALG_SCRATCH` takes care of this.

Adding more channels (> 8)

How to add more channels in the system?

By default PA SDK has support for 12 channels. See PA Reference Guide for details on the supported channels and paf-hd.pdf for additional channel definitions. In order to add more channels in the system following changes are required:

- Allocate frame buffers for additional channels by changing number of channels in `pa/f/s19/I14/params.c`

```
const SmInt PAF_AST_params_numchan_PAi[1] =
{
  12,
};
```

- Modify `ipce_i14.c` to set the number of output channels

```
#define IPCE_OUTPUT_NUMCHAN       12
```

- Modify `pa/asp/std/ccm.c` to select the channels used
- Add `ccm.c` to final project.
- Update Channel configuration for additional channels as explained in section Channel Configuration Settings.

How to test 10ch output on DA8xx EVM?

Only 8 channels can be output from the EVM simultaneously. But more than 8 channels can be tested on the DA8xx EVM using channel mapping alpha commands. See PA Reference Guide for detailed information on channel mapping alpha commands. EncChannelMapFrom and EncChannelMapTo alpha commands can be used to test all channels on EVM. These alpha commands route the frame buffer output of any channel to the desired MCASP pins. For Eg: In order to check height channel output; you can route it to the back channels on the MCASP pins using below alpha commands:

```
writeENCChannelMapFrom16(PAF_LEFT,PAF_RGHT,PAF_LSUR,PAF_RSUR,PAF_CNTR,PAF_SUBW,PAF_LBAK,P
writeENCChannelMapTo16(0,4,1,5,2,6,-1,-1,3,7,-1,-1,-1,-1,-1,-1)
```

Channel Configuration Settings

A Channel Configuration is the manner in which the channels of the AudioFrame are organized:

- The Request Channel Configuration is the form of output audio that an ASP Algorithm should attempt to produce for system output, typically a read-only but application-variable quantity.
- The Stream Channel Configuration describes the actual form of the AudioFrame data. As such, this is a read/write, application-variable quantity that describes the form of the AudioFrame data presented to an ASP Algorithm, which is to be updated by that ASP Algorithm to describe the AudioFrame data that is output.

Channel configuration registers are 32-bit wide and it consists of four, 8-bit fields or parts:

- The satellite part indicates the channel configuration for satellite, that is, non-bass channels. This includes front channels Left, Right, and Center, as well as rear channels Left and Right Surround and Left and Right Back.
- The subwoofer part indicates the channel configuration for bass channels. This consists of either the LFE channel (before bass management) or subwoofer channels (after bass management).
- The auxiliary part indicates special information about the encoding of the data in the channels:
  - If the satellite part indicates that the channel configuration is two-channel, the auxiliary part gives auxiliary information about how that two-channel data should be interpreted. This includes stereo, stereo-unknown, surround-encoded, mono, and dual-mono. Note that only two-channel data uses the auxiliary part in this manner.
  - If the satellite part indicates that the channel configuration is multi-channel with two channel surround, the auxiliary part gives auxiliary information about how that two channel surround data should be interpreted. This includes stereo-surround, surround-unknown, back-encoded surround, and mono-surround. Note that only multi-channel data with two surround channels uses the auxiliary part in this manner.
- extMask part contain 8-bits each sepcifying one or two extended channels, as shown in Table 1. These indicate the presence of additional channels from the standard 7.2 channels.

Table 1 Extension Mask of Channel Configuration

Bit Position	Indicated Channels
PAF_CC_EXT_LwRw (0)	Lw + Rw
PAF_CC_EXT_LcRc (1)	Lc + Rc
PAF_CC_EXT_LhRh (2)	Lh + Rh
PAF_CC_EXT_Cvh (3)	Cvh
PAF_CC_EXT_Ts (4)	Ts
PAF_CC_EXT_LhsRhs (5)	Lhs + Rhs
PAF_CC_EXT_LhrRhr (6)	Lhr + Rhr
PAF_CC_EXT_Chr (7)	Chr

Channel Assignment of these channels are indicated in paf-hd.pdf. Bit positions in “extMask” are enum-erated in “t:/pa/f/include/pafcc.h”. The extMask for each of the above channel/channel pair is indicated below:

```
    #define PAF_CC_EXTMASK_LwRw     (1u << PAF_CC_EXT_LwRw)
    #define PAF_CC_EXTMASK_LcRc     (1u << PAF_CC_EXT_LcRc)
    #define PAF_CC_EXTMASK_LhRh     (1u << PAF_CC_EXT_LhRh)
    #define PAF_CC_EXTMASK_Cvh      (1u << PAF_CC_EXT_Cvh)
    #define PAF_CC_EXTMASK_Ts       (1u << PAF_CC_EXT_Ts)
    #define PAF_CC_EXTMASK_LhsRhs   (1u << PAF_CC_EXT_LhsRhs)
    #define PAF_CC_EXTMASK_LhrRhr   (1u << PAF_CC_EXT_LhrRhr)
    #define PAF_CC_EXTMASK_Chr      (1u << PAF_CC_EXT_Chr)
```

When an ASP generates any extended channels it is required to set the extMask correspoding to that channel(s)in PAF_ChannelConfiguration.

Channel Mask

The Channel Configuration Mask Table is a data structure to be used for conversion of channel configuration information into Channel Masks. It is a read-only, application-constant quantity. The Channel Configuration Mask Table is typically used via the channelMask member function of the Audio Frame Data Structure. The original Channel Mask is 16-bits wide and does not represnt all extendec channels. A new 32-bit Channel Mask has been added for representing the newly added channels.

```
  typedef XDAS_Int32 PAF_ChannelMask_HD;
```

There is a one-to-one correspondence between bits in the Channel Mask and channels in Audio Frame. The helper function “pAudioFrame->fxns->channelMask()” has been changed to return a 32-bit PAF_ChannelMask_HD, instead of a 16-bit PAF_ChannelMask. This function now takes care of extMask field of PAF_ChannelCOnfiguration. ASPs can check for the presence of additional channels by making use of function pAudioFrame->fxns->channelMask().

Sample Process

The Sample Process is a multi-byte bit mask of values of the form (1<<PAF_PROCESS_X) that indicates the processing algorithms that have operated on the Audio Frame Data. It is a read/write quantity, and it is the responsibility of an ASP Algorithm to realize this update if appropriate. For example, if the PL Algorithm and BM Algorithms have operated on the Audio Frame data, the Sample Process value will be `(1<<PAF_PROCESS_PL)|(1<<PAF_PROCESS_BM)` if the corresponding symbols exist. Sample Process

symbols shall be defined on an as-needed basis, depending upon whether such information is required subsequently in the audio stream processing chain.

How to measure peak mips?

Peak mips refers to the worst mips taken in block processing for a frame. PA/F works on block of samples (256) and there could be situations where there is high variation in mips across various calls. If there is high difference between mips for processing different blocks; there could be peak mips problems. You could see the worst case mips associted with your ASP by measuring mips across each apply call after cache-flush operation. The ASP wrapper code in the package shows how to measure mips under worst case conditions.If there is a very significance difference in mips across different blocks; this could lead to real-time issues when run with PA. So algorithm has to take care of averaging out mips across all the calls for a frame. If this is not possible; this may have to be solved at framwork level by managing output buffers. Please contact PA support if there exists such a situation.

Article Sources and Contributors

DA8xx ASP Development Kit Overview *Source*: http://processors.wiki.ti.com/index.php?oldid=36527 *Contributors*: A0393170, JoeCap, Meenae

DA8xx ASP Programming Interface *Source*: http://processors.wiki.ti.com/index.php?oldid=37063 *Contributors*: A0393170, JoeCap, Meenae

DA8xx Alpha Codes *Source*: http://processors.wiki.ti.com/index.php?oldid=35232 *Contributors*: A0393170, JoeCap, Meenae

DA8xx ASP Porting Guide *Source*: http://processors.wiki.ti.com/index.php?oldid=37062 *Contributors*: A0393170, JoeCap, Meenae, UrmilParikh

DA8xx PA/F Integration tips *Source*: http://processors.wiki.ti.com/index.php?oldid=35931 *Contributors*: A0393170, JoeCap, Meenae

Image Sources, Licenses and Contributors

License