



**DSP Concepts, LLC.**

# Audio Weaver 2.0

## Platform Developers Guide



Copyright Information

© 2008 DSP Concepts, LLC., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from DSP Concepts, LLC.

Printed in the USA.

Disclaimer

DSP Concepts, LLC reserves the right to change this product without prior notice. Information furnished by DSP Concepts is believed to be accurate and reliable. However, no responsibility is assumed by DSP Concepts 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 DSP Concepts, Inc.

TABLE OF CONTENTS

1. Introduction.....	4
2. Threads and Interrupts .....	5
3. Global Variables and Data Structures.....	6
3.1. Communication Buffer .....	6
3.2. Memory Heaps.....	6
3.3. Module Table .....	6
3.4. Target Info .....	7
4. Tuning / Host Communication API .....	9
4.1. Message Structure.....	9
4.2. Byte Communication Handler Example .....	10
4.3. Framework Messages.....	13
5. Framework Audio I/O API .....	17
6. Flash File System API .....	20
6.1. High Level APIs .....	20
6.2. Low-Level APIs.....	22
7. Deployment.....	23
7.1. Deployment Issues .....	23
7.2. API Reference.....	26
7.3. Examples.....	30
7.4. Comparison of Approaches.....	52
8. Building the Target Executable .....	53
8.1. Memory Sections .....	53
8.2. Macro to Define when Building Module Libraries .....	54

## 1. Introduction

This document describes how to provide Audio Weaver functionality to new target hardware Platforms. To keep porting efforts to a minimum and to maximize software robustness, the target code is divided into two categories. The *Framework* is a set of common Audio Weaver functions. The *Platform* is hardware dependent and is updated for each new target. Some examples of Framework code are the command processing loop and the code to determine details of the audio processing, such as the number of inputs and outputs. Platform specific code includes all hardware configuration (mapping processor SRU pins, initializing external memory, interfacing to codecs), managing interrupts, and interfacing to flash memory.

The Framework and Platform are partitioned such that Audio Weaver can be supported in a variety of software environments. This includes simple interrupt driven schemes or sophisticated designs using commercially available real-time operating systems.

There are several similar hardware targets in Audio Weaver allowing some of the Platform code to be reused. For example, there is a byte oriented tuning interface that constructs messages out of complete 32-bit words. Another example is a flash file system for SPI devices. In these cases, the common Platform code is contained within reusable *Component* libraries.

The Audio Weaver Framework code is provided as libraries. The Platform and Component code for existing hardware targets is provided as source code. When creating a new hardware target, we recommend that you begin with a project for an existing hardware target and rewrite the Platform specific code and tweak the Component code. The main development effort will be in developing the hardware specific Platform code.

The Platform specific code is divided into 3 areas: real-time audio and interrupts, tuning / host communication, and optional flash memory support. This document uses the Audio Weaver Demo board as an example. The Demo Board has a SHARC 21371 processor and interfaces to the PC via USB. The 3 software areas are implemented as follows:

*Real-time audio and interrupts* – Uses DMA driven audio I/O with double buffering and a simple interrupt driven scheduler.

*Host / Tuning Interface* – The board uses an FTDI USB interface chip. The physical communication link is USB. However, the device appears to the PC as a UART. Messages are transferred a byte at a time and assembled into 32-bit words.

*Flash memory* – External SPI flash device.

Audio Weaver supports a general communication interface that can be used for Tuning or Host Communication. On development platforms, such as the Audio Weaver Demo Board, the interface is used for tuning and connects directly to the PC. If the hardware target has a host processor, then the audio processor would interface directly to the host (typically using SPI), and the tuning interface is between the host and the PC.

## **2. Threads and Interrupts**

An Audio Weaver Platform requires a minimum of 3 threads. From highest priority to lowest priority, the threads are arranged as:

- Audio I/O interrupts at a fixed block size (typically 32 samples)
- User interrupt for processing audio at a larger block size (a multiple of 32 samples)
- Main or foreground loop for message handling (non-interrupt based)

In addition, there may be another interrupt for the tuning interface, but this depends upon the target. In the Audio Weaver Demo Board, the tuning interface is handled via polling in the main loop.

Note that separate threads are used for audio I/O and audio processing. This design allows the audio processing to operate at a larger block size compared to the audio I/O. The block size of the audio processing is specified in the MATLAB system description and can be changed without recompiling the target.

The audio processing currently occurs within a single interrupt. The Framework APIs are structured in such a way as to permit processing in multiple interrupts in the future. This design will allow, for example, base audio processing at a block size of 64 samples and frequency domain processing using a block size of 256 in a separate lower priority audio processing thread.

### 3. Global Variables and Data Structures

The target defines a number of global variables and data structures. In the Audio Weaver Demo Board, these items are defined within `Danville_Demo_System.c`. In the example code pieces below, the constant values shown in CAPS are defined in the file `TargetInfo.h`. The names of these global variables must be maintained since they are accessed directly by the Framework code.

#### 3.1. Communication Buffer

The communication buffer is an array of 32-bit unsigned integers. The Platform essentially owns the communication buffer. The Platform allocates the buffer, receives messages over the tuning interface, calls the Framework communication handler to process the message, and then returns the resulting message back to the PC or host.

```
AWE_FW_SLOW_ANY_DATA DWORD s_PacketBuffer[MAX_COMMAND_BUFFER_LEN];
```

#### 3.2. Memory Heaps

The memory heaps are used by the Framework to dynamically allocate the audio processing. There are 3 different memory heaps corresponding to the memory architecture of the SHARC processor: internal DM memory, internal PM memory, and external memory. A module's constructor function specifies which heap to allocate memory from. Modules typically specify a priority of memory heaps. For example, allocate from the internal DM heap. If it is full, allocate from the internal PM heap. If it is full, allocate from external memory. This overflow behavior lends itself to processors that only have two different memory types (e.g., internal and external) or to cache based architectures where no distinction is made between memory types.

```
AWE_FW_SLOW_ANY_CONST UINT g_master_heap_size = MASTER_HEAP_SIZE;
AWE_FW_SLOW_ANY_CONST UINT g_slow_heap_size = SLOW_HEAP_SIZE;
AWE_FW_SLOW_ANY_CONST UINT g_fastb_heap_size = FASTB_HEAP_SIZE;

section("awe_heap_fast")
UINT g_master_heap[MASTER_HEAP_SIZE];

#pragma section("awe_heap_slow",NO_INIT)
UINT g_slow_heap[SLOW_HEAP_SIZE];

section("awe_heap_fastb")
UINT g_fastb_heap[FASTB_HEAP_SIZE];
```

#### 3.3. Module Table

The module table specifies which audio modules are compiled into the executable and available for dynamic allocation. The module table is an array of pointers to module class objects.

```
AWE_FW_SLOW_ANY_DATA
const ModClassModule *g_module_descriptor_table[] =
```

```
{
LISTOFCLASSOBJECTS
};

AWE_FW_SLOW_ANY_DATA
UINT g_module_descriptor_table_size = sizeof(g_module_descriptor_table)
/ sizeof(g_module_descriptor_table[0]);
```

The macro LISTOFCLASSOBJECTS is defined in TargetInfo.h. The macro for the Audio Weaver Demo board is

```
#define LISTOFCLASSOBJECTS \
&awe_modAGCAutoAttackReleaseClass, \
&awe_modAGCCoreClass, \
&awe_modAGCCoreARClass, \
&awe_modAGCGainComputerClass, \
&awe_modAGCLimiterCoreClass, \
...
```

Edit this macro to change the audio modules available on the target.

### 3.4. Target Info

This data structure describes the capabilities of the hardware target to the Audio Weaver Server. When the Server connects, it reads this data structure and displays the information in the output window.

```
AWE_FW_SLOW_ANY_DATA TargetInfo g_target_info =
{
    48000.0f,                // float m_sampleRate;
    263.375e6f,            // float m_profileClockSpeed;
    32,                    // UINT m_base_block_size;

    MAKE_TARGET_INFO_PACKED(1, 2, 4, TRUE, TRUE,
PROCESSOR_TYPE_SHARC),
                                // UINT m_packedData;
    MAKE_TARGET_VERSION(1, 0, 0, 1),
                                // UINT m_version;

    MAKE_PACKED_STRING('3', '7', '1', 'D'),
    MAKE_PACKED_STRING('E', 'M', 'O', '\0'),
                                // UINT m_packedName[2];
    MAX_COMMAND_BUFFER_LEN,
};
```

The fields in the target info data structure are either 32-bit integers or floats. Macros are provided to pack several pieces of information into 32-bit values. These macros are defined in Framework.h

```
MAKE_TARGET_INFO_PACKED(
    size of integer,
    number of audio inputs, number of audio outputs,
    is floating-point, is flash memory supported,
```

processor type enumeration)

MAKE\_TARGET\_VERSION(  
    Major version number, ..., Minor version number)

MAKE\_PACKED\_STRING(  
    4 characters )

## 4. Tuning / Host Communication API

The API is defined in PlatformAPI.h and there are only two functions to call.

```
extern void awe_fwPacketInit(DWORD *packet_buf, int buffer_length);  
  
void awe_fwPacketProcess(void);
```

The Platform owns the message buffer and calls awe\_fwPacketInit() once at initialization time. A pointer to the message buffer and its length are provided as arguments. In the Audio Weaver Demo Board project, this function is called during Platform initialization:

```
awe_fwPacketInit(s_PacketBuffer, MAX_COMMAND_BUFFER_LEN);
```

The message buffer and length are stored by the Framework and used for subsequent message processing.

The message processing function awe\_fwPacketProcess() is called directly from the communication handler found in FT245R\_Comm.c. This function is called from the main processing loop as part of awe\_pltTick(). *In all cases, awe\_fwPacketProcess() must be called from the main loop.* If your communication handler is interrupt based, then you need to set a flag when a complete message is received and have the message processed in the main loop.

All tuning/host messages are represented as arrays of 32-bit values. Incoming messages are stored in s\_PacketBuffer (the first argument to awe\_fwPacketInit) by the Platform specific communication handler. The length of the tuning buffer is part of the TargetInfo data structure and is communicated to the Server when the Server initially connects. When constructing messages, the Server ensures that no messages sent to the target will exceed the message buffer length and that no response from the target will exceed the message buffer length. (The only messages with large payloads are the array read/write commands and the Server is careful to restrict their sizes.)

Only 1 message is processed at a time over the tuning interface. After each message has been processed by the target, a corresponding return message is transmitted to the host. The buffer s\_PacketBuffer is used to hold both the received and the transmitted messages. The transmitted message is constructed by awe\_fwPacketProcess() and it is the responsibility of the Platform to communicate the message back to the PC or host.

### 4.1. Message Structure

Messages use a 1 word header and 1 word CRC as shown below:

```
[16 bit message length | 16 bit command ID]  
Payload[0]  
Payload[1]  
...  
Payload[N-1]
```

CRC Word

The length of the message includes the header and CRC word. Thus, the shortest possible message – one without a payload – is two words in length.

The message header packs a 16-bit message length and 16-bit command ID into a 32-bit word. Message IDs 0x0000 through 0x7FFF are reserved for the Framework. Message IDs outside of this range are reserved for user extensions to the protocol.

The CRC word is a 32-bit value and computed so that when all words of the message, including the CRC, are XOR'ed together, the result is 0. The following code computes the CRC of a packet prior to transmission:

```
nLen = g_PacketBuffer[0] >> 16;

DWORD crc=0;
for(i=0; i < (nLen-1); i++)
{
    crc^=g_PacketBuffer[i];
}

g_PacketBuffer[nLen-1] = crc;
```

On the target side, the CRC calculations are handled by `awe_fwPacketProcess()`. It checks both the CRC of the received message and computes the checksum of the reply.

## 4.2. Byte Communication Handler Example

The Audio Weaver Demo Board has a USB interface chip from FTDI. When connected to the PC, the board appears as a UART although the physical link is USB. Messages are exchanged over the communication link byte-by-byte. This section describes how this communication link works and how bytes are sent and grouped together into 32-bit words. Refer to the file `FT245R_Comm.c` for the low-level hardware link and the file `ByteCommHandler.c` for the functions which combine individual bytes into complete messages. This section serves as an example implementation of an Audio Weaver tuning interface.

The communication link is configured in `awe_pltInit()` via the call

```
FT245R_Init(s_PacketBuffer);
```

The communication link is then periodically polled within the Platform's tick function:

```
AWE_FW_SLOW_CODE void awe_pltTick(void)
{
    FT245R_Poll();
}
```

The polling function is shown below.

```
void FT245R_Poll()
{
    unsigned int bBitSet;
    unsigned char ch;
    unsigned int nStatus;

    nStatus = *pUSB_STATUS;

    // Shift bits of interest into place
    nStatus = nStatus >> 13;

    /* Check if USB is plugged in */
    if ( (nStatus & USB_VALID_BITS_MASK) == 7)
    {
        return;
    }

    bBitSet = nStatus & USB_RXF_BIT;
    if (!bBitSet)
    {
        unsigned int nData = *pUSB_RX_DATA;

        /* Read and dispatch the receive byte. */
        ch = (unsigned char)(nData & 0xFF);
        if (ByteOnRxReady(ch))
        {
            // Process the message
            awe_fwPacketProcess();

            // Send the return message
            ByteCommSend();
        }

        return;
    }

    bBitSet = nStatus & USB_TXE_BIT;
    if (!bBitSet)
    {
        /* Poll for the next transmit byte. */

        DISABLE_INTERRUPTS();
        if (ByteOnTxReady(&ch) != 0)
        {
            *pUSB_TX_DATA = ch;
        }

        ENABLE_INTERRUPTS();
        return;
    }
}
```

Return if USB is not plugged in.

Check if there is a received byte. If there is, grab the byte and pass it to the ByteOnRxReady() communication handler. This function returns 1 when a complete message has arrived and 0 otherwise.

Process the message and then send the response back to the PC/host.

Check if there is a byte to transmit. If there is, grab the byte and send it to the PC.

The functions ByteOnRxReady() and ByteOnTxReady() are defined in ByteCommHandler.c and implement the lower level byte-by-byte protocol. Each message (array of words) is broken up into individual bytes. Additional framing bytes surrounds the start and end of the message in order to provide additional robustness. Messages are formatted as:

Start of message character (0x02)  
Message sequence character ('0', '1', ... or '9')  
Data payload. Each 32-bit word is broken into 5 bytes  
End of message character (0x03)

The message sequence character is set by the PC and is used to differentiate messages and check for errors in replies. The character starts at 0x30 ('0'), advances by 1 each time, and then wraps around after 0x39 ('9'). If the Platform sees two sequential messages with the same sequence number, it knows that the PC is resending the message and that, most likely, the returned message was corrupted. The duplicate message is not processed a second time. Rather the previous reply is resent.

The data payload is constructed by breaking each 32-bit word into 5 bytes. The data is packed as: 7 bits, 7 bits, 7 bits, 7 bits, and 5 bits. Furthermore, the high bit of each byte is set. Thus, the start message character, sequence character, payload bytes, and end message character are all distinct bytes. Whenever the start message character is received, the communication handler begins a new message – even if it was in the middle of receiving a different message.

An enumerated type at the start of ByteCommHandler.c indicates the possible states of the byte protocol:

```
typedef enum
{
    /* --- States for receiving a command --- */

    tMS_Rx_CmdSTX,          /**< Receive Command STX */
    tMS_Rx_CmdSeq,         /**< Receive Command Sequence ID */
    tMS_Rx_CmdData,        /**< Receive Command Data */
    tMS_Rx_CmdETX,         /**< Receive Command ETX */

    /* --- State while processing command --- */

    tMS_Tx_WaitingForRsp,  /**< Processing Command. */

    /* --- States for sending a response --- */

    tMS_Tx_RspSTX,        /**< Send Response STX */
    tMS_Tx_RspSeq,        /**< Send Response Sequence ID */
    tMS_Tx_RspData,       /**< Send Response Data */
    tMS_Tx_RspETX,        /**< Send Response ETX */
} tMsgState;
```

Under normal error-free operation, the protocol proceeds as:

```
tMS_Rx_CmdSTX

tMS_Rx_CmdSeq
    Look for duplicate sequence characters and retransmit, if
    necessary

tMS_Rx_CmdData
    The first 32-bit data word is received (by packing 5 bytes) and
    the message length determined.
```

Continue receiving bytes until the message is completely formed.

Check for the message end character 0x03

tMS\_Tx\_WaitingForRsp\_

At this point, a complete message has been received and we indicate to the calling function that the message can be processed. The calling function calls the Framework function `awe_fwPacketProcess()`. The message is handled and the reply constructed within the message buffer. The function below is called to kick off transmission of the reply message:

```
void ByteCommSend(void)
{
    /* Save the packet with xor. */
    s_msg_size = (s_msg_data[0] >> 16);

    // Change the state so that the packet starts transmitting back
    s_state     = tMS_Tx_RspSTX;
}
```

You'll note that we simply specify the length of the return message and move the protocol into the next state, `tMS_Tx_RspSTX`. We then advance through the transmit states:

```
tMS_Tx_RspSTX
    Send the start of message character 0x02.

tMS_Tx_RspSeq
    Send the message sequence character

tMS_Tx_RspData
    Send the entire message payload by breaking up each 32-bit word
    into 5 bytes.

tMS_Rx_CmdSTX
    Return to the first receive state. At this point, the message is
    done transmitting and we wait for the PC/Host to start the next
    message.
```

### 4.3. Framework Messages

The function `awe_fwPacketProcess()` in `PacketAPI.c` handles all of the Framework messages. A few of the messages are documented here in order to give insight into message handling and constructing messages.

The global variable `g_PacketBuffer` points to the message buffer and is initialized by a call to `awe_fwPacketInit()`. `g_PacketBuffer` holds both the received message and the Framework generated reply. As discussed in Section 4.1, each message has a 16-bit ID and the file `ProxyIDs.h` holds their definitions.

**4.3.1. PFID\_awe\_fwSetCall**

Call a module's set function.

Received message format:

Message Length = 4	ID = PFID_awe_fwSetCall
Pointer to module instance uint mask uint CRC	

Reply message format:

Message Length = 3	ID = 0
int Error status uint CRC	

**4.3.2. PFID\_awe\_fwGetCall**

Call a module's get function.

Received message format:

Message Length = 4	ID = PFID_awe_fwGetCall
Pointer to module instance uint mask uint CRC	

Reply message format:

Message Length = 3	ID = 0
int Error status uint CRC	

**4.3.3. PFID\_ClassPin\_Constructor**

Controls a pin type object on the target by calling the Framework function ClassPin\_Constructor().

Received message format:

Message Length = 7	ID = PFID_ClassPin_Constructor
int blockSize int numChannels int size of sample, in bytes int sampleRate	

int isComplex CRC
----------------------

Reply message format:

Message Length = 4	ID = 0
Pointer to pin object int Error status uint CRC	

**4.3.4. PFID\_awe\_fwSetValue**

Sets an integer scalar variable on the target

Received message format:

Message Length = 4	ID = PFID_awe_fwSetValue
Address Value CRC	

Reply message format:

Message Length = 3	ID = 0
Error status CRC	

**4.3.5. Message Summary**

The table below summarizes all of the messages supported by the Framework. The structure of individual messages and the corresponding reply can be deduced by examining PacketAPC.c.

Message ID	Description
PFID_awe_fwSetCall	Call the Set function associated with an audio module.
PFID_awe_fwGetCall	Calls the Get function associated with an audio module.
PFID_ClassPin_Constructor	Constructs an instance of a pin type object.
PFID_awe_fwGetClassType	Returns the class type of an object
PFID_awe_fwGetPinType	Queries a pin to determine its properties.
PFID_ClassWire_Constructor	Constructs a single instance of a wire
PFID_BindIOToWire	Binds a wire to a Platform input or output wire.
PFID_awe_fwFetchValue	Reads a single integer value from the Target.
PFID_awe_fwFetchFloatValue	Reads a single floating-point value from the Target
PFID_awe_fwSetValue	Writes a single integer value on the Target
PFID_awe_fwSetFloatValue	Writes a single floating-point value on the Target
PFID_awe_fwGetHeapCount	Returns the number of heaps on the Target.
PFID_awe_fwGetHeapSize	Returns the number of words remaining in a specified heap.

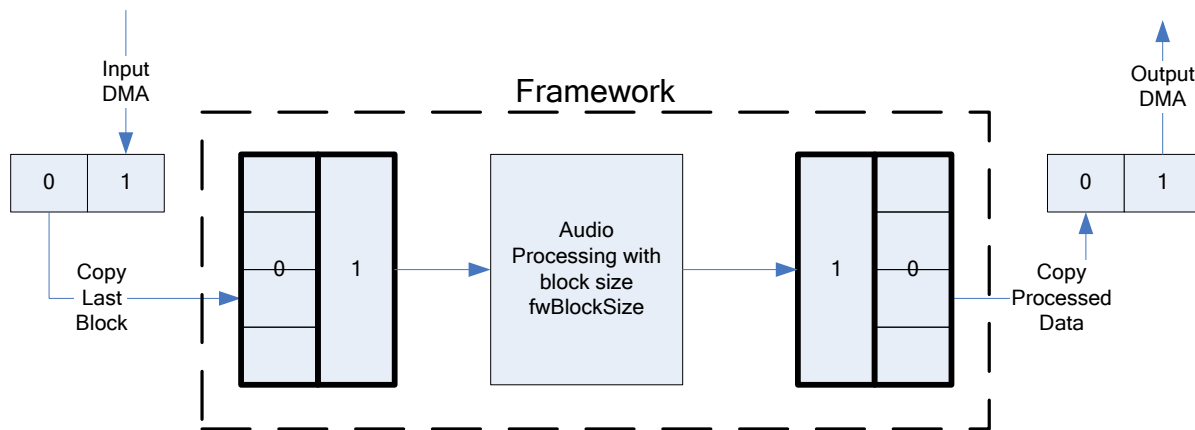
PFID_awe_fwDestroy	Destroys all allocated audio processing on the Target and reinitializes all heaps.
PFID_awe_fwGetCIModuleCount	Returns the number of module classes defined on the Target.
PFID_awe_fwGetCIModuleInfo	Returns information about a specific audio module class.
PFID_ClassModule_Constructor	Instantiates an audio module
PFID_ClassLayout_Constructor	Instantiates the overall processing layout (effectively a list of modules)
PFID_awe_fwSetWire	Sets the data portion of a wire. Used for regression testing.
PFID_awe_fwGetWire	Reads the data portion of a wire. Used for regression testing.
PFID_awe_fwSetModuleState	Sets the run-time status of a module
PFID_awe_fwGetModuleState	Returns the run-time status of a module
PFID_awe_fwPumpModule	Calls the processing function of a single module
PFID_ClassLayout_Process	Processes the currently defined layout
PFID_awe_fwGetFirstObject	Gets information about the first object defined in the memory heaps
PFID_awe_fwGetNextObject	Returns information about the next object defined in the memory heaps
PFID_awe_fwGetIndexObject	Returns information about the Nth instantiated object defined in the memory heaps
PFID_awe_fwGetFirstIO	Returns information about the first Platform I/O pin.
PFID_awe_fwGetNextIO	Returns information about the next Platform I/O pin.
PFID_StartAudio	Starts real-time audio processing
PFID_StopAudio	Stops real-time audio processing
PFID_awe_fwFetchValues	Reads an array of values
PFID_awe_fwSetValues	Writes an array of values
PFID_awe_fwGetSizeofInt	Returns the sizeof(int) evaluated on the Target
PFID_awe_GetTargetInfo	Returns information about the Target (number of inputs, outputs, fundamental block size, etc.)
PFID_awe_fwGetProfileValues	Returns profiling information related to the overall layout
PFID_awe_GetFirstFile	Returns information about the first file in the flash file system
PFID_awe_GetNextFile	Returns information about the next file in the flash file system
PFID_awe_OpenFile	Opens a file for reading or writing
PFID_awe_ReadFile	Reads data from the currently open file
PFID_awe_WriteFile	Writes data to the current open file
PFID_awe_CloseFile	Closes the currently open file
PFID_awe_DeleteFile	Deletes a single file
PFID_awe_ExecuteFile	Executes a compiled script file
PFID_awe_EraseFlash	Erases all of flash memory
PFID_awe_GetFileSystemInfo	Gets information about the flash file system (size, fragmentation, etc.)
PFID_awe_FileSystemReset	Resets the flash file system. It closes any open files.

### 5. Framework Audio I/O API

The Platform is responsible for all setup and buffering. This includes configuring the audio codec, DMA, and managing interrupts. This section describes how to pass audio data between the Platform and Framework within the audio receive interrupt. See the function `AudioReceive_ISR` within the file `AK4683_Audio.c` for an example.

The `TargetInfo` data structure describes the target's audio capabilities back to the Server. This includes the number of input and output channels, and the underlying real-time I/O block size. We'll refer to the block size of the Platform as `pltBlockSize`. The Platform manages all I/O buffering and this typically includes double buffering of the inputs and outputs.

The audio system instantiated by the Framework (either through messages sent by the PC/Host or from messages stored in flash memory) has multiple input and output channels and its own block size. Let `fwBlockSize` refer to the block size of the audio processing running in the Framework. `fwBlockSize` is always a multiple of `pltBlockSize`. The Framework handles double buffering of the input and output channels invisibly to the Platform. All that the Platform has to do is copy audio data of size `pltBlockSize` words to and from the Framework. This is illustrated in Figure 1 below.



**Figure 1. Figure showing the relationship between the Platform block size `pltBlockSize` and the Framework block size `fwBlockSize`. In this example, `fwBlockSize = 4 x pltBlockSize`. The input and output DMA processes operate at the smaller block and the Framework managing double buffering to the size `fwBlockSize` internally.**

When the Framework has received a total of `fwBlockSize` samples, it notifies the Platform that it needs to process the audio. The Platform then triggers a lower priority user interrupt which calls the Framework audio processing function.

Audio samples are represented in the Audio Weaver as signed 32-bit integers<sup>1</sup>. Audio data

<sup>1</sup> Even on systems with native floating-point processing like the SHARC, audio data arrives as 32-bit integers. The

passed to the Framework should be left justified within each 32-bit word.

The following functions are used to exchange audio data between the Platform and the Framework. First, to determine the number of input and output channels that the current system has, call the function

```
void awe_fwGetChannelCount(int *inCount, int *outCount);
```

If `inCount=0` and `outCount=0`, this indicates that the audio system is not instantiated or that the system does not have any input or output pins. The Platform handles this special case by copying audio data directly from the input DMA buffer to the output buffer. This serves as an aid when debugging the Platform.

To determine where a particular input channel, `chan`, should be copied, call the function:

```
int *awe_fwGetInputChannelPtr(int chan, int *stride);
```

`chan` is an index in the range 0 to `inCount-1`. The function returns a pointer to the Framework's internal double buffer; the spacing between copied audio samples is `*stride`. The Platform then copies the data from its input DMA buffer to the Framework buffer.

Similarly on the output, to determine where a particular channel of output data should be taken from the Framework, call

```
int *awe_fwGetOutputChannelPtr(int chan, int *stride);
```

Again, specify the channel number `chan` in the range 0 to `outCount-1`. The function returns the spacing between samples in `*stride`. The Platform then copies the data out of the Framework and input the output DMA buffer.

After all input and output channels have been copied, call the function

```
unsigned int awe_fwAudioDMAComplete(int sampsPerTick);
```

The argument `sampsPerTick` should be set to `pltBlockSize`. This tells the Framework how many input and output samples have been added / consumed. The function manages the internal Framework double buffers and returns a bit mask to the Platform indicating whether the audio processing should fire. A non-zero bit indicates that a particular user interrupt should be triggered for audio processing. Currently, Audio Weaver supports only a single audio processing interrupt and only bit 0 needs to be examined. In the future, this may be expanded to processing audio in multiple block sizes and additional bits will need to be checked.

A common implementation is as follows:

---

conversion from integer to floating-point is performed within the audio processing layout itself.

```
layout_mask = awe_fwAudioDMAComplete(SAMPS_PER_TICK);  
  
if (layoutMask)  
    raise(SIG_USR0);
```

When Audio Weaver supports multiple simultaneous block sizes, the implementation changes to:

```
layoutMask = awe_fwAudioDMAComplete(SAMPS_PER_TICK);  
  
if (layoutMask & 0x01)  
    raise(SIG_USR0);  
  
if (layoutMask & 0x02)  
    raise(SIG_USR1);
```

## **6. Flash File System API**

Audio Weaver uses a file system to store start up command scripts. The command scripts are generated by the PC and programmed using the Server or MATLAB scripts. The command files use the same binary message format used by the Tuning/Host interface. In fact, the messages in the startup script files are processed by calls to `awe_fwPacketProcess()`. There is one minor difference, though. In order to conserve memory space, the startup scripts do not contain the final CRC word. The CRC word is manually generated prior to executing each message.

The file system API is general enough to support traditional file systems (such as a hard disk drive) or flash memory. The file system API is divided into Framework (FlashFileAPI.c) and Platform (AT26F004.c) portions. The Framework portion provides high-level APIs for opening, reading, and writing files, while the Platform portion interfaces to the physical memory. On the Audio Weaver Demo Board there is another file, FlashFileSystem.c, which encapsulates common features common to flash memory devices.

For simplicity, the design allows only a single file to be open at a time. Trying to open a second file will generate an error.

### **6.1. High Level APIs**

The high-level flash API is defined in `FileSystem.h`. The functions return the error conditions defined in the file `Errors.h`. `E_SUCCESS` indicates that a function completed successfully.

To get information about the first file in the file system call:

```
DWORD awe_fwGetFirstFile(PDIRECTORY_ENTRY * pDirEntry);
```

The directory entry is a structure

```
typedef struct _DIRECTORY_ENTRY
{
    DWORD nFileInfo;
    DWORD nDataDWordCnt;
    DWORD nFilename[ALLOCATION_BLOCKSIZE_DWORDS - 2];
} DIRECTORY_ENTRY, *PDIRECTORY_ENTRY;
```

To get information on the next file in the file system, call

```
DWORD awe_fwGetNextFile(PDIRECTORY_ENTRY * pDirEntry);
```

This function will fail once the last file in the file system is reached.

To open a file for reading or writing use:

```
DWORD awe_fwOpenFile(DWORD nFileAttribute, PDWORD pFileNameInDWords,
```

```
PDWORD nFileLenInDWords);
```

nFileAttribute=0 specifies that the file should be opened for reading; nFileAttribute != 0 specifies that the file should be opened for writing. You specify which file to open by passing a pointer to a file name (4 characters packed per 32-bit word). The function returns the length of the file in nFileLengthInDWords.

To read data sequentially from the currently open file use

```
DWORD awe_fwReadFile(DWORD nWordsToRead, PDWORD pBuffer, PDWORD  
pDWordsRead);
```

To write data sequentially to the currently open file use:

```
DWORD awe_fwWriteFile(DWORD nWordsToWrite, PDWORD pBuffer, PDWORD  
pDWordsWritten);
```

To close the currently open file, call

```
DWORD awe_fwCloseFile(void);
```

To permanently delete a file, use

```
DWORD awe_fwDeleteFile(PDWORD pFileNameInDWords);
```

Deleting a file marks its directory entry as deleted but does not free up any space in the file system. To reformat the file system and recapture and fragmented memory, call

```
DWORD awe_fwEraseFlash(void);
```

To reset the flash file system (in case a file is left open), call

```
DWORD awe_fwResetFileSystem(void);
```

To find a file by name call

```
PDIRECTORY_ENTRY awe_fwFindFile(PDWORD pFileNameInDWords);
```

Finally, to determine the file attributes call

```
DWORD awe_fwGetFileAttribute(PDIRECTORY_ENTRY pDirectoryEntry);
```

This function returns a bit field indicating the properties of the file. Several #define's in FileSystem.h interpret the returned bit mask:

```
#define LOAD_IMAGE 0x01  
#define STARTUP_FILE 0x02
```

```
#define DATA_FILE 0x04
#define COMPILED_SCRIPT 0x08
#define COMMAND_SCRIPT 0x10
#define PRESET_SCRIPT 0x20
#define FILE_DELETED 0x80
```

A startup compiled binary command script is marked:

```
STARTUP_FILE | COMPILED_SCRIPT | COMMAND_SCRIPT = 0x1A = 26
```

A startup compiled binary command script is marked:

```
STARTUP_FILE | COMPILED_SCRIPT | PRESET_SCRIPT = 0x2A = 32
```

## 6.2. Low-Level APIs

The low-level Platform flash API is defined in PlatformAPI.h and implemented in AT26F004.c and FlashFileSystem.c

```
BOOL awe_pltInitFlashFileSystem(void);
BOOL awe_pltReadFlashMemory(DWORD nAddress, PDWORD pBuffer, DWORD
    nWordsToRead);
BOOL awe_pltWriteFlashMemory(DWORD nAddress, PDWORD pBuffer, DWORD
    nWordsToWrite);
BOOL awe_pltEraseFlashMemory4KSector(DWORD nStartingAddress, DWORD
    nNumberOfSectors);
BOOL awe_pltAllocateBlock(DWORD nAddress);
```

## 7. Deployment

This section describes additional issues that need to be resolved when deploying a product created using Audio Weaver. There are several items to consider. First, where will the information needed to instantiate the audio system be stored? Second, how does the Server determine symbol addresses to enable tuning? And finally, how can the audio processing be controlled in the final product, either by a host controller or additional control software?

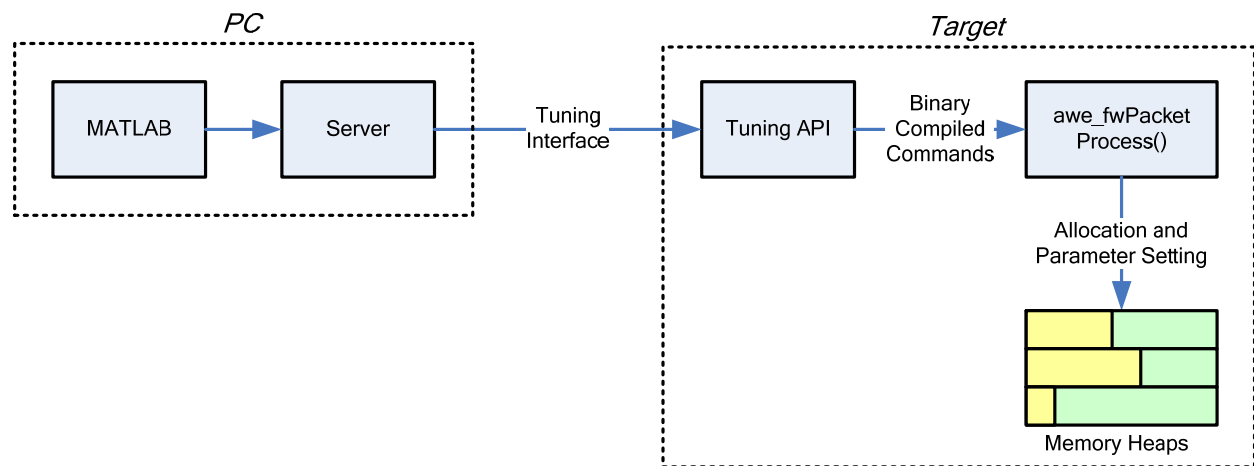
### 7.1. Deployment Issues

This section discusses the issues involved in deploying a system in more detail.

#### 7.1.1. Memory Models

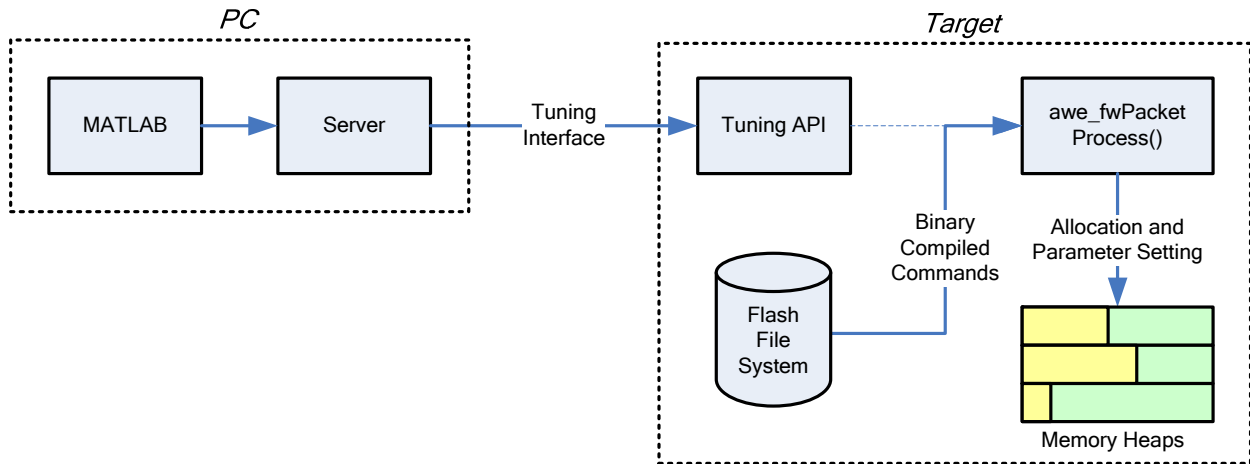
All of the deployment models utilize dynamic memory allocation. That is, all of the data structures associated with the audio design are allocated at run-time. There are 4 types of structures to allocate: pin types, wires, audio modules, and layout structures. In addition, input and output wires are bound to the I/O pins and variable values are set. All of the data structures are allocated from the 3 memory heaps.

When building a system directly from MATLAB, the memory allocation occurs in response to messages sent from the Server to the Target. The messages are sent as arrays of 32-bit binary words and interpreted by the `awe_fwPacketProcess()` function found in `PacketAPI.c`. This function has a large switch statement with separate code for evaluating each command. All of the other operations – binding wires and setting parameter values – are also supported by the binary commands. The overall process is shown in Figure 2.



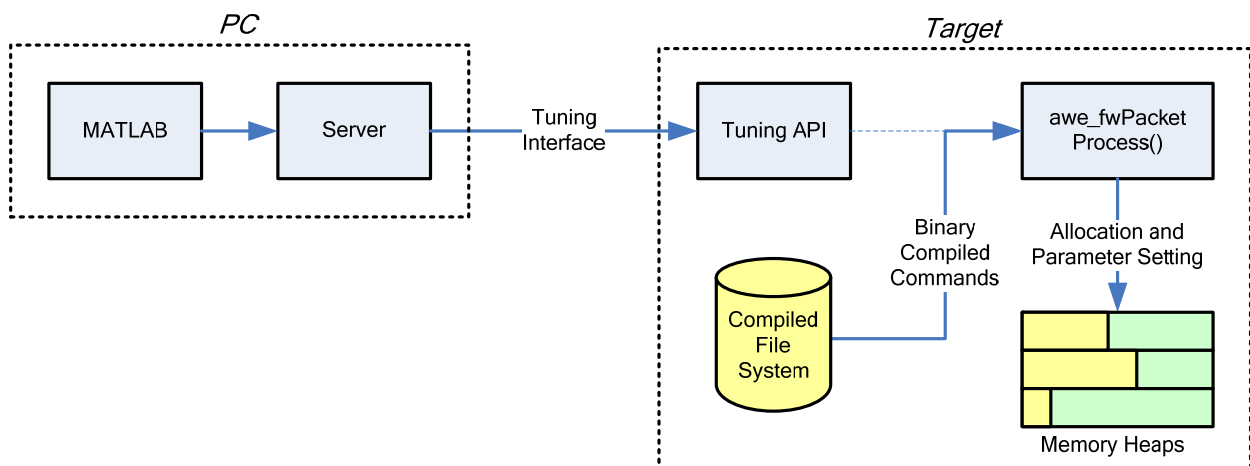
**Figure 2. Figure illustrating the standard build process. Commands are sent from MATLAB to the target to instantiate modules and set parameters.**

Now consider a target with both a tuning interface and a flash file system as illustrated in Figure 3. This is the most flexible and easy to program target. Instead of taking commands from MATLAB, the binary commands are taken from the flash file system. Each command is evaluated by awe\_fwPacketProcess(). Of course, to create the commands in the flash file system requires MATLAB, but there is no need to rebuild the target executable when there is a change to the audio processing.



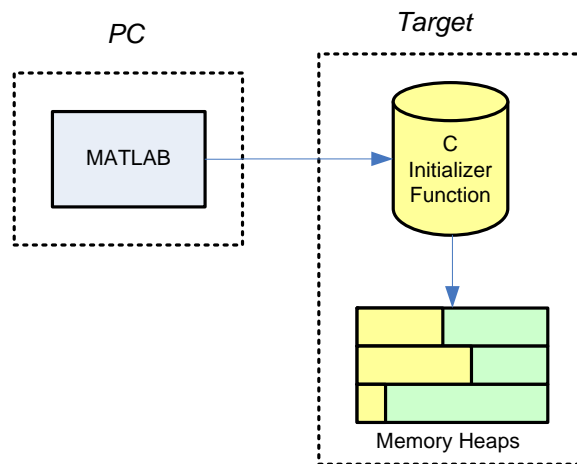
**Figure 3. Standalone system in which the commands to instantiate the audio system are taken from the flash file system. The PC can optionally connect for tuning.**

Now consider a target that only has a tuning interface but no flash file system. This is illustrated in Figure 4. In this case, the binary commands are compiled into the executable and then interpreted at boot time. This is referred to as a *compiled file system*, and the executable must be rebuilt whenever there is a change to the audio processing. Audio Weaver has functions which automate the process of created compiled file systems.



**Figure 4. Standalone system with a compiled file system. The file system is stored in memory and compiled with the executable.**

Finally, consider the case of a target without a tuning interface as shown in Figure 5. This target has limited functionality and is unable to be tuned or interact with the PC. Still, there are times when it is useful to have an Audio Weaver system run in a limited environment. In this case, Audio Weaver generates *C initializers* consisting of a sequence of C statements. Instead of having the binary commands be interpreted by `awe_fwPacketProcess()` as in the previous examples, C code is generated that accomplishes the same thing. The target executable must be rebuilt whenever there is a change to the audio processing.



**Figure 5. System without a tuning interface. MATLAB generates C initializer files which are compiled with the target executable.**

Why can't binary commands be used to initialize a system without a tuning interface? It turns out that the binary commands sent to the target contain physical addresses as arguments. These physical addresses are managed internally by the Server and are only known when connected to an actual target. Without a tuning interface, the symbol addresses are not known and thus binary initialization commands cannot be constructed. Note also that the binary commands are system specific and need to be recomputed if there are changes to the physical memory map of the target (e.g., the location of the memory heaps changes).

### 7.1.2. Attaching to a Target

When development starts, the audio system is prototyped from MATLAB using the standard Audio Weaver build process. The standard build procedure destroys the audio system (by clearing all of the heaps) and the re-instantiates the system. Later on, once the system has been deployed, it is running standalone on the target independently of MATLAB. For debugging purposes it is often useful to be able to reconnect to the running target without disturbing the audio processing. Of course, the standard build procedure could be used, but this wipes out the current state of the system. Audio Weaver proves the ability to *attach* to the running instance of the system.

To attach to a running system from MATLAB you need the original system object together with a *symbol table* describing where objects are located in the memory heaps. Symbol tables are stored in text files containing a sequence of Audio Weaver script commands. We generally use the extension ".tsf" for tuning symbol files, but this is not a requirement. A few lines from a typical tuning symbol file are shown below:

```
add_symbol, scale, ModuleScalerDB, 786893, 1
add_symbol, inputMeter, ModuleMeter, 786905, 1
add_symbol, agc_core, ModuleAGCCore, 786923, 1
add_symbol, agc_mult, ModuleAGCMultiplier, 786950, 1
add_symbol, outputMeter, ModuleMeter, 786961, 1
add_symbol, theLayout, Layout, 786989, 1
```

Each line contains the `add_symbol` command followed by the symbol name, class name and in memory address. The final argument is a Boolean which causes the Server to enforce some basic consistency checks on the symbol. First, the address of the symbol must be within one of the memory heaps, and second, the class ID of the object in memory must match the class ID of the supplied class name.

Tuning symbol files are created by Audio Weaver as a byproduct of compiling an Audio Weaver script file. They are also created by parsing the linker output file in the case of C initializers. The process of creating tuning symbol files is discussed later in this chapter.

### **7.1.3. Target Symbol Table**

Frequently there is a need to control the audio processing in a deployed product. It may be controlled by an external microcontroller by sending commands over the tuning interface; or it may be controlled by additional code running on the target processor itself. In both cases, the control code needs to know the in-memory addresses of the audio module structures.

Unfortunately, when an audio system is instantiated using a set of binary commands, the addresses of the data structures are no available within the C environment. Each binary command is executed in turn without recording any of the resulting addresses. To work around this, Audio Weaver provides a mechanism for generating a symbol table that may be used in C code. This symbol table is generated whenever an Audio Weaver script file is compiled (this is in addition to the .tsf file mentioned above). For flexibility, two separate files are generated. A header file containing `#define`'s for the various symbols, and a C file with a variable definition for each symbol.

## **7.2. API Reference**

This section describes the MATLAB commands that automate the process of deploying systems and attaching to them for tuning.

### **7.2.1. awe\_compile.m**

```
awe_compile(AWSFILE)
```

Compiles an Audio Weaver script file in the context of the currently connected target. The script commands are sent to the target and the system instantiated. The user interface may also be drawn. The argument `AWSFILE` is the name of the file containing the script commands. Assume that it is "file.aws". The function generates a number of output files:

`file.awb` – Binary commands that were sent from the Server to the Target. The Server filters out any commands – such as those for drawing inspectors – that are not needed by the Target. These binary commands are only relevant to the currently connected target.

`file.tsf` – Tuning symbol file which contains Audio Weaver "add\_symbol" script commands.

`file.h` – Header file providing tuning symbol definitions.

`file.c` – C file providing tuning symbol declarations.

The output files are generated in the same directory as the input file `AWSFILE`. The function also takes additional arguments allowing you to override the default output file naming:

```
awe_compile(AWSFILE, AWBFILE, TSFFILE, HFILE, CFILE)
```

`AWBFILE` – overrides naming of the generated `.awb` file.

`TSFFILE` – overrides naming of the generated `.tsf` file.

`HFILE` – overrides naming of the generated `.h` file.

`CFILE` – overrides naming of the generated `.c` file.

When overriding file names, if only the name of the file is specified, (e.g. `myFile.awb`) then the file is written in the same directory as the input file `AWSFILE`. Alternatively, a complete path for each output file can be specified.

The variable names in the `.tsf` file always match the MATLAB module names. By default, the variables names in the `.c` and `.h` file also match the module names. In some cases, you may have name collision with other C variables. To work around, this Audio Weaver provides the ability to specify a prefix for each generated name using the global variable

```
AWE_INFO.buildControl.staticVariablePrefix
```

For example, the original `.c` file contains a line of the form:

```
awe_modScalerDBInstance.*scale =  
(awe_modScalerDBInstance *) AWE_SYM_ADDR_scale;
```

This defines an instance of the scaler module and names it "scale". If you set

```
AWE_INFO.buildControl.staticVariablePrefix='SYS_';
```

and then recompile the script, you'll end up with:

```
awe_modScalerDBInstance *SYS_scale =  
    (awe_modScalerDBInstance*)AWE_SYM_ADDR_scale;
```

## 7.2.2. awe\_cinit.m

Generates a C code initializer which instantiates audio processing. We recommend using this only if the target does not have a tuning interface.

```
awe_cinit(SYS, CFILE)
```

Generates the C code initializer for the system SYS using the target information in structure T and writes the result into the file CFILE. This function works in conjunction with target\_get\_info.m. You should first manually override the target information by calling

```
target_get_info(T)
```

and passing it a data structure containing the target information. Refer to the on-line help for target\_get\_info.m to see the fields in T. Then call awe\_cinit.m. During the build process, MATLAB never communicates with the Server or the Target.

The initializer file contains all of the code needed to instantiate and configures the processing. Note that the commands to destroy the system and start audio processing are not included; these must be added manually to the other Platform code. The function writes two output files, .c and .h. The name of the .h file is generated based on the name of the .c file. You can override the default .h file naming by using a third argument:

```
awe_cinit(SYS, CFILE, HFILE)
```

By default, the variables names in the .c and .h file match the names of the MATLAB modules. In some cases, you may have name collision with other C variables. To work around this, Audio Weaver provides the ability to specify a prefix for each generated name using the global variable

```
AWE_INFO.buildControl.staticVariablePrefix
```

## 7.2.3. awe\_attach.m

Attaches to a running instance of audio processing. The general form of the command is:

```
awe_attach(SYS, FILE)
```

where

SYS is the @awe\_subsystem object describing the system.

FILE – is the file containing the tuning symbols. This can be either a ".tsf" file generated by awe\_compile.m, or in the case of system using C initializers, a VisualDSP++ linker generated .map.xml file.

The tuning symbol file must match the system instantiated on the target. If not, you will access invalid memory addresses.

Internally, the function goes through the first few steps of the build process including matching the processing to the target and flattening the system. If FILE is a tuning symbol file (.tsf), then the set of commands in FILE is sent to the Server to create the symbols. The Server performs some rudimentary error checking on the symbols. First, it checks that the address of the symbol lies within one of the 3 memory heaps. Second, it checks that the class ID of the object on the target (an integer value specifying its class), matches the object class specified in the add\_symbol call.

If FILE is a linker output file (.map.xml) generated by VisualDSP++, then additional error checking is performed. The script checks that all of the symbol addresses required by SYS are contained within the executable. If not, the script fails. After parsing, the symbols read from the .map.xml file are written to a .tsf file for future use.

Once you've connected to the target, the system SYS is now live. Changes made to SYS will be reflected in real-time on the target. To disconnect SYS from the target, use:

```
awe_detach(SYS);
```

### **7.2.4. awe\_filesystem\_initializer.m**

```
awe_filesystem_initializer(OUTPUTNAME, FILE1, ATTRIB1, FILE2, ATTRIB2, ...)
```

Creates a .c file containing a statically generated file system that may be compiled into an executable. The file system appears to the Audio Weaver target as a flash file system but is actually statically initialized data that is part of the executable. Arguments:

OUTPUTNAME - name of the output C file to generate.

FILE1 - first file to include in the flash file system.

ATTRIB1 - attribute byte for the first file.

FILE2, ATTRIB2 - file and attribute information repeated for the second file. The function allows you to specify an arbitrary number of files as pairs of inputs.

The file names may be absolute paths, or paths relative to the current MATLAB working directory.

An example of a file system initializer is shown in Section 7.3.4.5.

### **7.2.5. awe\_detach.m**

```
awe_detach(SYS);
```

Takes a "live" system SYS and detaches it from the running target. After a system is detached, it is "dead" and no longer interacts with the running system.

Before a system is built it is dead. Then upon building, it becomes live.

### **7.3. Examples**

This section contains examples showing how to use the deployment functions in a wide variety of situations. The examples are based on the files in the <AWE>\Examples\deployment folder. The file agc\_example\_sys.m creates – but does not build – the automatic gain control system. The function agc\_example\_inspect.m draws the figure diagram and the inspectors.

The examples require an external DSP target; they won't work with the native PC target. Each of the following cases are considered:

1. Standalone target with a tuning interface and a flash file system
2. Standalone target with a tuning interface but without a flash file system
3. Standalone target without a tuning interface and without a flash file system

The generated source files are listed below in Section 7.3.4.

#### **7.3.1. Standalone target with a tuning interface and a flash file system**

First connect to the hardware target. Then, build the system and create an .aws file using the diary functionality:

```
SYS=agc_example_sys;  
awe_diary('on', 'agc_example.aws');  
SYS=build(SYS);  
test_start_audio;  
awe_diary('off');
```

This builds the system on the target and starts real-time audio processing. Then draw the inspectors

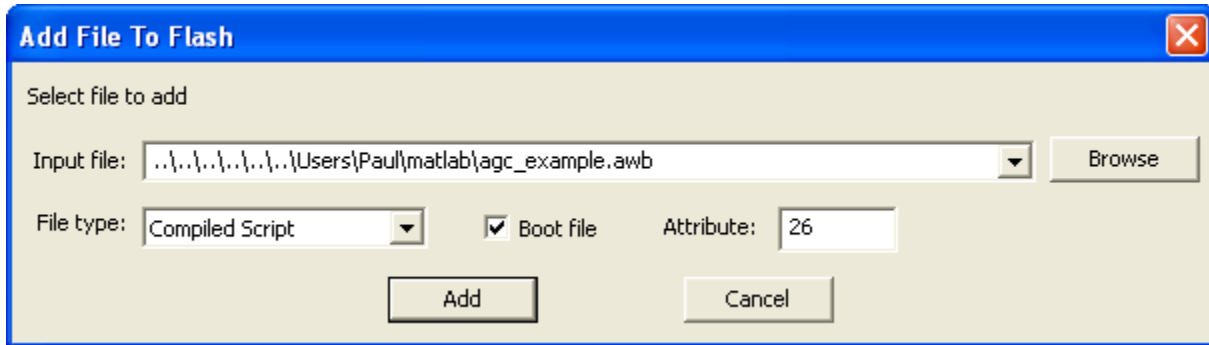
```
agc_example_inspect(SYS);
```

So far, this is the normal build process contained within the file agc\_example.m. Next, we'll create the compiled script file and associated tuning files:

```
awe_compile('agc_example.aws');
```

The command executes the script and rebuilds the system on the target. Compiling also generates the files `agc_example.awb`, `agc_example.tsf`, `agc_example.c`, and `agc_example.h`. These files are shown in Section 7.3.4.

Next, use the flash manager and store the `agc_example.awb` into the flash file system on the target. Configure the file attributes so that the target boots up and executes the script.

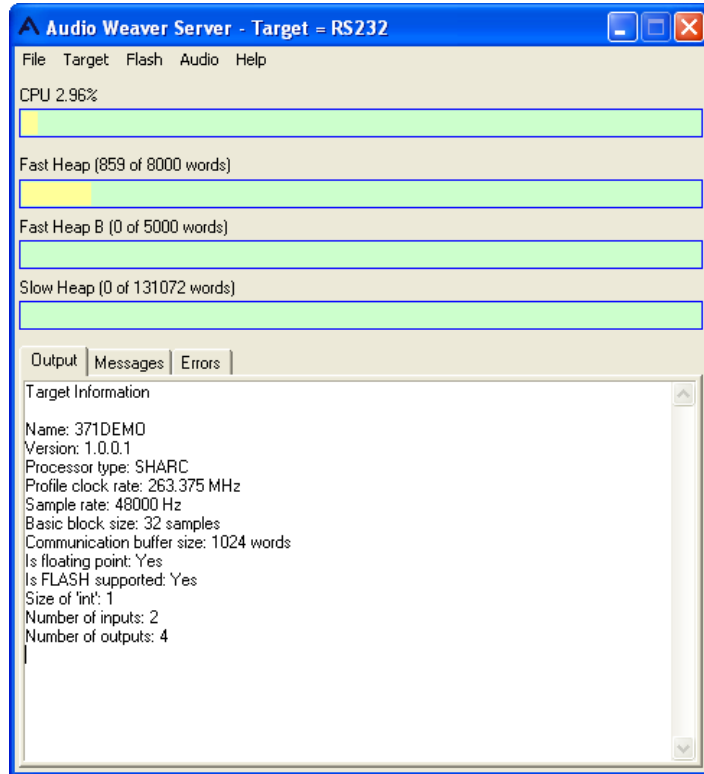


Shut down the Server and reset the DSP target. The board will boot up and instantiate the audio processing. Since the command `test_start_audio` was captured by the diary, the Target will automatically begin processing audio.

Now, let's attach to the running audio processing on the target. Start the Server again from MATLAB

```
awe_init;
```

When the Server connects to the target, you'll see the target information in the Output window and also notice that audio processing is active. The CPU meter indicates activity and the memory heaps have been partially allocated.



Reinstantiate the AGC system from MATLAB

```
SYS=agc_example_sys;
```

Then attach to the running instance.

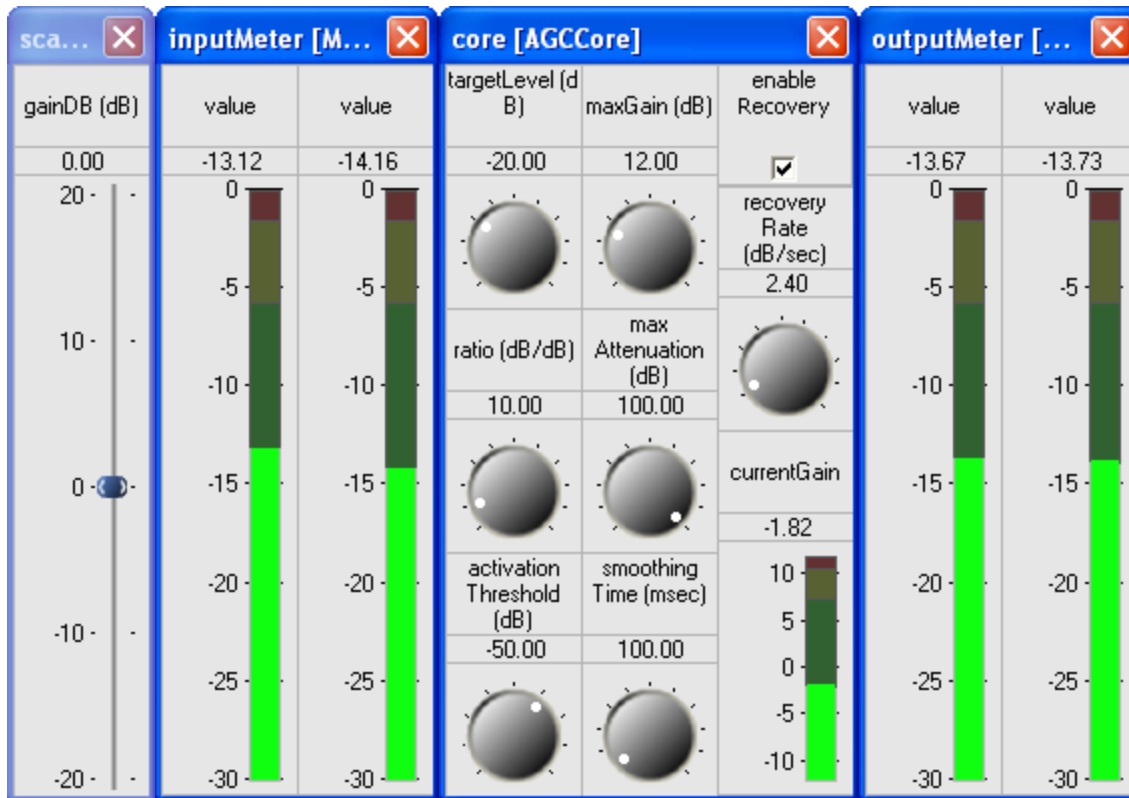
```
SYS=awe_attach(SYS, 'agc_example.tsf');
```

The tuning symbol file simply contains the commands. While attaching, the "add\_symbol" commands contained within the .tsf file are executed. These commands tell the Server where the objects are located in memory. When attaching, the target continues to run without interruption. The returned system SYS is live. You can see the audio modules updating in real-time:

```
>> SYS.inputMeter.value  
  
ans =  
  
    0.0501  
    0.0547  
  
>> SYS.inputMeter.value  
  
ans =  
  
    0.0496  
    0.0487
```

You can also create the inspectors for the running system:

```
agc_example_inspect(SYS);
```



One final step would be to write control code on the target for manipulating the audio processing. This can be done using the `agc_example.c` and `agc_example.h` files that were generated by the `awe_compile.m` commands. These files are shown in Section 7.3.4.

`agc_example.c` contains the variable declarations. A variable is initialized for each pin type, wire, module, and layout in the system. The `agc_example.h` file contains the associated variable addresses and extern definitions. For example, using this file you can adjust the level of the AGC Core module using the C commands:

```
PREFIX_agc_core->targetLevel = -10.0;
ClassModule_Set(agc_core, MASK_AGCCore_targetLevel);
```

### 7.3.2. Standalone target with a tuning interface but without a flash file system

On this target, you can dynamically instantiate audio systems over the tuning interface. Once you have a design that works, it needs to be stored on the target and instantiated at boot time. There are two possible approaches: a compiled file system, or C initializer code. We'll use a compiled file system in this case and save the C initializer code for Section 7.3.3.

As before, connect to the hardware target. Then build the system and create an `.aws` file using

the diary functionality:

```
SYS=agc_example_sys;  
awe_diary('on', 'agc_example.aws');  
SYS=build(SYS);  
test_start_audio;  
awe_diary('off');
```

Then compile the script file that was just captured

```
awe_compile('agc_example.aws');
```

The command executes the script and rebuilds the system on the target. As before, several files are generated: `agc_example.awb`, `agc_example.tsf`, `agc_example.c`, and `agc_example.h`.

Instead of storing the `agc_example.awb` file into the flash file system, we'll create C data structures which mimic the flash file storage. Use the command:

```
awe_filesystem_initializer('agc_example_fs.c', 'agc_example.awb', 26);
```

Next add the `agc_example_fs.c` file to the project which builds your target executable. If you look at the start of the file, shown in Section 7.3.4.5, you'll see that you need to set the `#define USE_COMPILED_FILE_SYSTEM` to enable the compiled file system. Note that your project should define *either* `USE_COMPILED_FILE_SYSTEM` or `USE_FLASH_FILE_SYSTEM`, but *never both*. Rebuild the target executable and run. The system instantiate the AGC system upon startup.

For standalone tuning, the files `agc_example.c` and `agc_example.h`, created by `awe_compile.m` provide the needed symbols and variable declarations.

As before, MATLAB can attach to the live system. Reinstantiate the AGC system from MATLAB

```
SYS=agc_example_sys;
```

Then attach to the running instance.

```
SYS=awe_attach(SYS, 'agc_example.tsf');
```

This last step relied upon the fact that the tuning symbols in `agc_example.tsf`, based up the dynamic allocation performed by MATLAB, are still valid after the Target has been recompiled. *For this to work, the memory heaps should be placed in their own memory sections in the linker definition file (LDF) so that they will not move upon recompiling.*

### **7.3.3. Standalone target without a tuning interface and without a flash file system**

This last example applies to all situations where there is no tuning interface between the Server

and the Target processor. The audio system cannot be instantiated in response to binary commands sent from the Server (standard build), binary commands stored in the flash file system (Section 7.3.1), or binary commands stored in a compiled file system (Section 7.3.2). In fact, many of the MATLAB commands, such as `target_get_info`, will no longer work in the traditional sense since there is no Target to communicate with.

Instead, we use the C initializers to instantiate the system. First, create a data structure that describes the Target hardware. In this example, a structure simulating the capabilities of the Audio Weaver Demo Board will be created:

```
T.name='BoardWithoutTuningInterface';
T.version='0.0.0.0';
T.processorType='SHARC';
T.isFloatingPoint=1;
T.isFlashSupported=0;
T.numIn=2;
T.numOut=4;
T.inputPinType.numChannels=2;
T.inputPinType.blockSize=32;
T.inputPinType.sampleRate=48000;
T.inputPinType.dataType='float';
T.inputPinType.isComplex=0;
T.inputPinType.numChannelsRange=2;
T.inputPinType.blockSizeRange=32;
T.inputPinType.sampleRateRange=48000;
T.inputPinType.dataTypeRange={'float'};
T.inputPinType.isComplexRange=0;
T.outputPinType.numChannels=4;
T.outputPinType.blockSize=32;
T.outputPinType.sampleRate=48000;
T.outputPinType.dataType='float';
T.outputPinType.isComplex=0;
T.outputPinType.numChannelsRange=4;
T.outputPinType.blockSizeRange=32;
T.outputPinType.sampleRateRange=48000;
T.outputPinType.dataTypeRange={'float'};
T.outputPinType.isComplexRange=0;
T.fundamentalBlockSize=32;
T.sampleRate=48000;
T.nsizeofInt=1;
T.profileClock=263e6;
```

Then pass this data structure to the `target_get_info.m` function.

```
target_get_info(T);
```

From now on, whenever the `target_get_info.m` function is called, it will return the data structure `T` rather than request the information from the target.

Then instantiate the system

```
SYS=agc_example_sys;
```

You'll see the message

```
Overriding target information
```

shown in the MATLAB output window. This is a warning reminding you that the target information has been overridden. Next, we'll specify a prefix for the generated variable names

```
global AWE_INFO  
AWE_INFO.buildControl.staticVariablePrefix='PREFIX_';
```

Finally, generate the C initializer function

```
awe_cinit(SYS, 'agc_example_cinit.c');
```

This generates a pair of files, `agc_example_static.c` and `agc_example_static.h`, both shown later on in Section 7.3.4.6 and 7.3.4.7. The file `agc_example_cinit.c` contains a single function `test_Init()`. The function name is based on the class name of the system, "test" with "\_Init()" appended to it. Add the file `agc_example_static.c` to the target project and call the initialization function from the main function. Also manually start the audio processing:

```
/* Initialize the audio processing */  
test_Init();  
  
/* Start real-time audio */  
awe_pltAudioStart();
```

Note that in this case, the audio processing must be manually started. This is in contrast to compiled script files (`.awb`) which may contain commands to start audio processing.

`agc_example_cinit.h` and `agc_example_cinit.c` contain calls to the Audio Weaver constructor functions. Arguments are passed in integer arrays with floating-point values represented as 32-bit unsigned integers. Comments are supplies to make it easy to interpret the arguments.

The initialization function returns an integer value indicating success (`E_SUCCESS`). Other error codes are defined in `Errors.h`.

### **7.3.4. Generated Files**

This section shows all of the generated files used in the examples above. We set

```
AWE_INFO.buildControl.staticVariablePrefix='PREFIX_';
```

to highlight the use the prefix variable.

#### **7.3.4.1. awe\_example.tsf**

You'll see the MATLAB module names directly without the prefixes. (The prefixes are only

used when generating C variable names.)

```
add_symbol, audio2x32x48000_real, PinType, 786432, 1
add_symbol, audio4x32x48000_real, PinType, 786437, 1
add_symbol, audio1x32x48000_real, PinType, 786442, 1
add_symbol, wire1, Wire, 786447, 1
add_symbol, wire2, Wire, 786516, 1
add_symbol, wire3, Wire, 786649, 1
add_symbol, wire4, Wire, 786718, 1
add_symbol, wire5, Wire, 786755, 1
add_symbol, wire6, Wire, 786824, 1
add_symbol, autoInputConvert_1, ModuleFract32ToFloat, 787149, 1
add_symbol, scale, ModuleScalerDB, 787159, 1
add_symbol, inputMeter, ModuleMeter, 787171, 1
add_symbol, agc_core, ModuleAGCCore, 787189, 1
add_symbol, agc_mult, ModuleAGCMultiplier, 787216, 1
add_symbol, outputMeter, ModuleMeter, 787227, 1
add_symbol, autoOutputRouter_1, ModuleRouter, 787245, 1
add_symbol, autoOutputConvert_1, ModuleFloatToFract32, 787260, 1
add_symbol, theLayout, Layout, 787270, 1
```

**7.3.4.2.      awe\_example.awb**

This file is binary and cannot be shown.

**7.3.4.3.      awe\_example.h**

This file defines the tuning symbol addresses. There are two entries per object. First, there is the object ID which reflects the order in which the modules were allocated. The object ID starts at 1 and increments per object. The Audio Weaver run-time stores the objects in a linked list and the object ID corresponds to the location of the object within this list. The object ID is currently not used but is provided for future functionality. The second item is the physical address of the object, in hexadecimal.

The file also has an extern declaration for each object variable.

```
/*
 *
 *            Target Tuning Symbol File
 *            -----
 *
 *            Generated on:    20-Aug-2008 16:24:17
 *    Based on script file:    C:\Users\Paul\matlab\agc_example.awb
 *
 */

#ifndef AGC_EXAMPLE_H
#define AGC_EXAMPLE_H

#include "Framework.h"

#include "ModAGCCore.h"
#include "ModAGCMultiplier.h"
```

```
#include "ModFloatToFract32.h"
#include "ModFract32ToFloat.h"
#include "ModMeter.h"
#include "ModRouter.h"
#include "ModScalerDB.h"

#define PREFIX_audio2x32x48000_real_ID 1
#define PREFIX_audio2x32x48000_real_ADDR 0x000c0000
#define PREFIX_audio4x32x48000_real_ID 2
#define PREFIX_audio4x32x48000_real_ADDR 0x000c0005
#define PREFIX_audiolx32x48000_real_ID 3
#define PREFIX_audiolx32x48000_real_ADDR 0x000c000a
#define PREFIX_wire1_ID 4
#define PREFIX_wire1_ADDR 0x000c000f
#define PREFIX_wire2_ID 5
#define PREFIX_wire2_ADDR 0x000c0054
#define PREFIX_wire3_ID 6
#define PREFIX_wire3_ADDR 0x000c00d9
#define PREFIX_wire4_ID 7
#define PREFIX_wire4_ADDR 0x000c011e
#define PREFIX_wire5_ID 8
#define PREFIX_wire5_ADDR 0x000c0143
#define PREFIX_wire6_ID 9
#define PREFIX_wire6_ADDR 0x000c0188
#define PREFIX_autoInputConvert_1_ID 10
#define PREFIX_autoInputConvert_1_ADDR 0x000c02cd
#define PREFIX_scale_ID 11
#define PREFIX_scale_ADDR 0x000c02d7
#define PREFIX_inputMeter_ID 12
#define PREFIX_inputMeter_ADDR 0x000c02e3
#define PREFIX_agc_core_ID 13
#define PREFIX_agc_core_ADDR 0x000c02f5
#define PREFIX_agc_mult_ID 14
#define PREFIX_agc_mult_ADDR 0x000c0310
#define PREFIX_outputMeter_ID 15
#define PREFIX_outputMeter_ADDR 0x000c031b
#define PREFIX_autoOutputRouter_1_ID 16
#define PREFIX_autoOutputRouter_1_ADDR 0x000c032d
#define PREFIX_autoOutputConvert_1_ID 17
#define PREFIX_autoOutputConvert_1_ADDR 0x000c033c
#define PREFIX_theLayout_ID 18
#define PREFIX_theLayout_ADDR 0x000c0346

/* -----
** Object variables declarations.
** ----- */

extern PinInstanceDescriptor *PREFIX_audio2x32x48000_real;
extern PinInstanceDescriptor *PREFIX_audio4x32x48000_real;
extern PinInstanceDescriptor *PREFIX_audiolx32x48000_real;
extern WireInstance *PREFIX_wire1;
extern WireInstance *PREFIX_wire2;
extern WireInstance *PREFIX_wire3;
extern WireInstance *PREFIX_wire4;
extern WireInstance *PREFIX_wire5;
extern WireInstance *PREFIX_wire6;
extern awe_modFract32ToFloatInstance *PREFIX_autoInputConvert_1;
extern awe_modScalerDBInstance *PREFIX_scale;
extern awe_modMeterInstance *PREFIX_inputMeter;
```

```
extern awe_modAGCCoreInstance *PREFIX_agc_core;
extern awe_modAGCMultiplierInstance *PREFIX_agc_mult;
extern awe_modMeterInstance *PREFIX_outputMeter;
extern awe_modRouterInstance *PREFIX_autoOutputRouter_1;
extern awe_modFloatToFract32Instance *PREFIX_autoOutputConvert_1;
extern LayoutInstance *PREFIX_theLayout;

#endif // AGC_EXAMPLE_H

/**
 * @}
 *
 * End of file.
 */
```

**7.3.4.4.      awe\_example.c**

This file declares a pointer to each of the allocated objects. The pointer addresses are taken from agc\_example.h.

```
/*
 *
 * Target Tuning Symbol File
 *
 * -----
 *
 * Generated on: 20-Aug-2008 16:24:17
 * Based on script file: C:\Users\Paul\matlab\agc_example.awc
 *
 */

#include "agc_example.h"

#ifdef __cplusplus
extern "C" {
#endif

/* -----
** Object variables declarations.
** ----- */

AWE_MOD_SLOW_ANY_DATA PinInstanceDescriptor *PREFIX_audio2x32x48000_real =
(PinInstanceDescriptor *) PREFIX_audio2x32x48000_real_ADDR;
AWE_MOD_SLOW_ANY_DATA PinInstanceDescriptor *PREFIX_audio4x32x48000_real =
(PinInstanceDescriptor *) PREFIX_audio4x32x48000_real_ADDR;
AWE_MOD_SLOW_ANY_DATA PinInstanceDescriptor *PREFIX_audio1x32x48000_real =
(PinInstanceDescriptor *) PREFIX_audio1x32x48000_real_ADDR;
AWE_MOD_SLOW_ANY_DATA WireInstance *PREFIX_wire1 = (WireInstance *)
PREFIX_wire1_ADDR;
AWE_MOD_SLOW_ANY_DATA WireInstance *PREFIX_wire2 = (WireInstance *)
PREFIX_wire2_ADDR;
AWE_MOD_SLOW_ANY_DATA WireInstance *PREFIX_wire3 = (WireInstance *)
PREFIX_wire3_ADDR;
AWE_MOD_SLOW_ANY_DATA WireInstance *PREFIX_wire4 = (WireInstance *)
PREFIX_wire4_ADDR;
AWE_MOD_SLOW_ANY_DATA WireInstance *PREFIX_wire5 = (WireInstance *)
PREFIX_wire5_ADDR;
AWE_MOD_SLOW_ANY_DATA WireInstance *PREFIX_wire6 = (WireInstance *)
```

```
PREFIX_wire6_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modFract32ToFloatInstance *PREFIX_autoInputConvert_1
= (awe_modFract32ToFloatInstance *) PREFIX_autoInputConvert_1_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modScalerDBInstance *PREFIX_scale =
(awe_modScalerDBInstance *) PREFIX_scale_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modMeterInstance *PREFIX_inputMeter =
(awe_modMeterInstance *) PREFIX_inputMeter_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modAGCCoreInstance *PREFIX_agc_core =
(awe_modAGCCoreInstance *) PREFIX_agc_core_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modAGCMultiplierInstance *PREFIX_agc_mult =
(awe_modAGCMultiplierInstance *) PREFIX_agc_mult_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modMeterInstance *PREFIX_outputMeter =
(awe_modMeterInstance *) PREFIX_outputMeter_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modRouterInstance *PREFIX_autoOutputRouter_1 =
(awe_modRouterInstance *) PREFIX_autoOutputRouter_1_ADDR;
AWE_MOD_SLOW_ANY_DATA awe_modFloatToFract32Instance
*PREFIX_autoOutputConvert_1 = (awe_modFloatToFract32Instance *)
PREFIX_autoOutputConvert_1_ADDR;
AWE_MOD_SLOW_ANY_DATA LayoutInstance *PREFIX_theLayout = (LayoutInstance *)
PREFIX_theLayout_ADDR;

#ifdef __cplusplus
}
#endif

/**
 * @}
 *
 * End of file.
 */
```

## 7.3.4.5.      agc\_example\_fs.c

This file contains the compiled file system; essentially binary data corresponding to the compiled commands. There is also an overall directory structure.

```
/*
 *
 *      Autogenerated file system
 *
 */

#ifdef USE_COMPILED_FILE_SYSTEM

#include "FileInfo.h"
#include "Framework.h"

#ifdef __cplusplus
extern "C" {
#endif

// Define block size for the file system as 16 words/block (64 bytes/block).
#define FLASHBLOCKSIZE            16
AWE_FW_SLOW_ANY_DATA DWORD g_nFlashBlockSize = FLASHBLOCKSIZE;

// Count of number of files
```

```
AWE_FW_SLOW_ANY_DATA int g_nFileCnt = 1;

AWE_FW_SLOW_ANY_DATA DIRECTORY_ENTRY FileDir[] =
{
    // attr=26, name='agc_example.awb', length(words)=138, start block=0
    { 0x1aff0000, 0x0000008a, 0x5f636761, 0x6d617865, 0x2e656c70,
0x00627761, 0x00000000, 0x00000000 },
};

AWE_FW_SLOW_ANY_DATA DWORD FileData[][FLASHBLOCKSIZE] =
{
    //File 'agc_example.awb', block 0
    {
        0x0002000e, 0x00070003, 0x00000020, 0x00000002, 0x00000004,
0x0000bb80, 0x00000000, 0x00070003,
        0x00000020, 0x00000004, 0x00000004, 0x0000bb80, 0x00000000,
0x00070003, 0x00000020, 0x00000001,
    },
    //File 'agc_example.awb', block 1
    {
        0x00000004, 0x0000bb80, 0x00000000, 0x00030006, 0x000c0000,
0x00030006, 0x000c0005, 0x00030006,
        0x000c0000, 0x00030006, 0x000c000a, 0x00030006, 0x000c0000,
0x00030006, 0x000c0005, 0x00040007,
    },
    //File 'agc_example.awb', block 2
    {
        0x000c000f, 0x0021800d, 0x00040007, 0x000c0054, 0x00218004,
0x00070011, 0xbeef0810, 0x00000101,
        0x00000000, 0x000c000f, 0x000c00d9, 0x00090011, 0xbeef0c11,
0x00000101, 0x00000002, 0x000c00d9,
    },
    //File 'agc_example.awb', block 3
    {
        0x000c00d9, 0x00000000, 0x3f800000, 0x000c0011, 0xbeef0c00,
0x00010001, 0x00000005, 0x000c00d9,
        0x000c011e, 0x00000000, 0x40a00000, 0x4487e000, 0x3c1c8b37,
0x38393888, 0x00180011, 0xbeef0bec,
    },
    //File 'agc_example.awb', block 4
    {
        0x00000101, 0x00000011, 0x000c00d9, 0x000c011e, 0xc1a00000,
0x42c80000, 0x41400000, 0x41200000,
        0xc2480000, 0x42c80000, 0x00000001, 0x4019999a, 0x00000000,
0x3f666666, 0x395a6e3a, 0x3bd9ba0e,
    },
    //File 'agc_example.awb', block 5
    {
        0x3f800609, 0x3f7ff3ee, 0x00000000, 0x00000000, 0x3c800000,
0x00080011, 0xbeef0bf0, 0x00000102,
        0x00000000, 0x000c011e, 0x000c00d9, 0x000c0143, 0x000c0011,
0xbeef0c00, 0x00010001, 0x00000005,
    },
},
```

```
        //File 'agc_example.awb', block 6
        {
            0x000c0143, 0x000c011e, 0x00000000, 0x40a00000, 0x4487e000,
0x3c1c8b37, 0x38393888, 0x00070011,
            0xbeef080f, 0x00000101, 0x00000000, 0x000c0143, 0x000c0188,
0x00090020, 0x000c0338, 0x00000000,

        },
        //File 'agc_example.awb', block 7
        {
            0x00000004, 0x00000000, 0x00000001, 0x00000002, 0x00000003,
0x00040001, 0x000c032d, 0x00000100,
            0x00070011, 0xbeef0811, 0x00000101, 0x00000000, 0x000c0188,
0x000c0054, 0x000c0012, 0x00000009,

        },
        //File 'agc_example.awb', block 8
        {
            0x00000001, 0x000c02cd, 0x000c02d7, 0x000c02e3, 0x000c02f5,
0x000c0310, 0x000c031b, 0x000c032d,
            0x000c033c, 0x0002001d,

        },
};

PDWORD g_pFileData = (PDWORD)FileData;

#ifdef    __cplusplus
}
#endif

#endif // USE_COMPILED_FILE_SYSTEM

/* End of file. */
```

**7.3.4.6.      agc\_example\_cinit.h**

This file contains symbol declarations for the statically initialized code. If you compare this with agc\_example.h in Section 7.3.4.3, you'll note that there are no symbol addresses defined here. That is because the objects have not yet been allocated. The allocation occurs within agc\_example\_cinit.c and the symbol addresses are then assigned to the object pointer variables.

```
/*
 *
 *            C initializer function
 *            -----
 *
 *            Class:    test
 *            Description:
 *            Generated on:    20-Aug-2008 20:40:25
 *
 *
 *
 */
```

```
#ifndef _MOD_TEST_H
#define _MOD_TEST_H

#include "Framework.h"

#include "ModAGCCore.h"
#include "ModAGCMultiplier.h"
#include "ModFloatToFract32.h"
#include "ModFract32ToFloat.h"
#include "ModMeter.h"
#include "ModRouter.h"
#include "ModScalerDB.h"

/* -----
** Constructor function
** ----- */

int test_Init(void);

/* -----
** Object variables declarations.
** ----- */

// Pin type structures
extern PinInstanceDescriptor *PREFIX_audio2x32x48000_real;
extern PinInstanceDescriptor *PREFIX_audio4x32x48000_real;
extern PinInstanceDescriptor *PREFIX_audiolx32x48000_real;

// Wire structures
extern WireInstance *PREFIX_wire1;
extern WireInstance *PREFIX_wire2;
extern WireInstance *PREFIX_wire3;
extern WireInstance *PREFIX_wire4;
extern WireInstance *PREFIX_wire5;
extern WireInstance *PREFIX_wire6;

// Module structures
extern awe_modFract32ToFloatInstance *PREFIX_autoInputConvert_1;
extern awe_modScalerDBInstance *PREFIX_scale;
extern awe_modMeterInstance *PREFIX_inputMeter;
extern awe_modAGCCoreInstance *PREFIX_agc_core;
extern awe_modAGCMultiplierInstance *PREFIX_agc_mult;
extern awe_modMeterInstance *PREFIX_outputMeter;
extern awe_modRouterInstance *PREFIX_autoOutputRouter_1;
extern awe_modFloatToFract32Instance *PREFIX_autoOutputConvert_1;
extern LayoutInstance *PREFIX_theLayout;

#endif // _MOD_TEST_H

/**
 * @}
 *
 * End of file.
 */
```

## 7.3.4.7.      agc\_example\_cinit.c

Generated code for the C initialization routine. The top of the file defines all of the object pointers. The object pointers are set by the constructor functions within the `_Init()` function.

The `_Init()` function follows a standard format:

1. Allocate all of the pin type data structures
2. Allocate all of the wires
3. Bind the input and output wires to the input and output pins of the system
4. Allocate all of the audio modules
5. Allocate the overall layout

Most of the steps are straightforward, but we'll provide additional details on the module allocation.

The modules allocated correspond to the modules in the flattened system. For each module, we call into the Framework function `ClassModule_Constructor`. For example, the code to allocate the `ScalerDB` module is:

```
{
// [0] gainDB = 0
// [1] gain = 1
WireInstance *pWires[] = { PREFIX_wire3,PREFIX_wire3 };
uint args[] = { 0x00000000,0x3f800000 };
PREFIX_scale = (awe_modScalerDBInstance *) ClassModule_Constructor (
    (uint) CLASSID_SCALERDB,
    &retVal,
    ClassModule_PackFlags(1, 1, 0),
    pWires,
    2,
    (Sample *) &args[0]);
if (retVal != E_SUCCESS)
    return(retVal);
}
```

The constructor function takes the following arguments:

`CLASSID_SCALERDB` – integer that specifies the object class to the Framework.  
Given the `classID`, the Framework looks up the module in the class library.

`&retVal` – status information returned here. Code checks if `retVal == E_SUCCESS` immediately after the call to the constructor.

`ClassModule_PackFlags(1, 1, 0)` – indicates the number of wires that the module uses ordered as (numInput, numOutput, numScratch). This module has a single input wire and a single output wire; no scratch wires.

pWires – pointer to an array of wires. This array is defined above. So, the input wire is PREFIX\_wire3 and the output wire is PREFIX\_wire3. Thus, the module is doing its processing in place.

2 – number of additional arguments to the constructor function.

&args[0] – array of additional arguments, defined above.

The args[] array contains the default parameter settings for the module. The arguments are listed in precisely the same order as they appear in the module's instance structure. Looking in ModScalerDB.h, we see:

```
typedef struct _awe_modScalerDBInstance
{
    ModuleInstanceDescriptor instance;
    float      gainDB;           // Gain in DB
    float      gain;            // Gain in linear units
} awe_modScalerDBInstance;
```

Thus, the values in args[] will be written to gainDB and gain. args[] is defined as an array of unsigned integers and floating-point values are stored as unsigned integer. The top of the generated C code has comments to help you interpret the values assigned:

```
// [0] gainDB = 0
// [1] gain = 1
```

The array args[] contains one initializer per scaler instance variable. For example, a little further down in agc\_example\_cinit.c you'll see the initializer code for the input meter:

```
// [0] meterType = 0
// [1] attackTime = 5
// [2] releaseTime = 1087
// [3] attackCoeff = 0.0095547
// [4] releaseCoeff = 4.416e-005
WireInstance *pWires[] = { PREFIX_wire3, PREFIX_wire4 };
uint args[] = { 0x00000000, 0x40a00000, 0x4487e000, 0x3c1c8b37, 0x38393888
};
PREFIX_inputMeter = (awe_modMeterInstance *) ClassModule_Constructor (
    (uint) CLASSID_METER,
    &retVal,
    ClassModule_PackFlags(1, 0, 1),
    pWires,
    5,
    (Sample *) &args[0]);
if (retVal != E_SUCCESS)
    return(retVal);
}
```

There are 5 instance variables which are set. If you look at the instance data structure, you'll see that there is another variable, float \*value, containing a pointer to an array of measured values:

```
typedef struct _awe_modMeterInstance
{
    ModuleInstanceDescriptor instance;
    int          meterType;
    float        attackTime;
    float        releaseTime;
    float        attackCoeff;
    float        releaseCoeff;
    float*       value;
} awe_modMeterInstance;
```

The variables in the instance structure are ordered such that all arrays appear at the end. This allows the allocation scheme to work. The meter constructor function internally allocates the memory needed for the value array. By default, `awe_fwMalloc()` sets the contents of allocated memory to zero. Since this is the desired starting value for the array, the module construction is done.

Consider another case, the initializer for the `OutputRouter` module. This module has an array called `channelIndex` which needs to be initialized to `{0, 1, 2, 3}`. The code for this module is:

```
{
    WireInstance *pWires[] = { PREFIX_wire5, PREFIX_wire6 };
    PREFIX_autoOutputRouter_1 = (awe_modRouterInstance *)
ClassModule_Constructor ( (uint) CLASSID_ROUTER, &retVal,
ClassModule_PackFlags(1, 1, 0), pWires, 0, (Sample *) NULL);
    if (retVal != E_SUCCESS)
        return(retVal);
}

{
    int arrayInitializer[] = { 0,1,2,3 };
    awe_vecCopyInt32((int *) &arrayInitializer[0], 1, (int *)
&PREFIX_autoOutputRouter_1->channelIndex[0], 1, 4);
}
```

The call to `ClassModule_Constructor` appears as before. This allocates memory for the module and sets any scalar variables in the structure. In this case, the module has no constructor variables and therefore `args[]` does not appear; `NULL` is passed to the constructor function instead.

Below the constructor you'll see the array initialization code. The values `{0, 1, 2, 3}` are stored in `arrayInitializer[]` and then copied into the structure instance by a call to `awe_vecCopyInt32()`.

Things are even more involved in the case of compiled subsystems; these are subsystems that contain class definitions on the target and have `.flattenOnBuild=0`. For compiled subsystems, the scalar and array variables of the top-level subsystem are set as before, but then additional code is generated to initialize variables of the internal modules. In this example, there are no initializers of this type.

When a module is constructed, its status is set to active by default. If a module is in another state (bypassed, muted, or inactive), you'll see additional code to change the module status.

```
/******  
*  
*            C initializer function  
*            -----  
*  
*            Class:     test  
*            Description:  
*            Generated on:    20-Aug-2008 20:40:25  
*  
*  
*****/  
  
/*  
* @file  
*/  
  
#include "Framework.h"  
#include "Errors.h"  
#include "VectorLib.h"  
  
#include "agc_example_cinit.h"  
  
/* -----  
** Set default memory sections on the SHARC and Blackfin.  These  
** sections are used for any unspecified code and data items.  
** ----- */  
  
#if defined(__ADSP21000__) || defined(__ADSPBLACKFIN__)  
#pragma default_section(CODE, "awe_mod_slowcode")  
#pragma default_section(ALLDATA, "awe_mod_slowanydata")  
#pragma default_section(SWITCH, "awe_mod_slowanydata")  
#endif  
  
#ifdef __cplusplus  
extern "C" {  
#endif  
  
/* -----  
** Object variables declarations.  
** -----  
  
// Pin type structures  
PinInstanceDescriptor *PREFIX_audio2x32x48000_real;  
PinInstanceDescriptor *PREFIX_audio4x32x48000_real;  
PinInstanceDescriptor *PREFIX_audiolx32x48000_real;  
  
// Wire structures  
WireInstance *PREFIX_wire1;  
WireInstance *PREFIX_wire2;  
WireInstance *PREFIX_wire3;  
WireInstance *PREFIX_wire4;  
WireInstance *PREFIX_wire5;  
WireInstance *PREFIX_wire6;  
  
// Module structures  
awe_modFract32ToFloatInstance *PREFIX_autoInputConvert_1;  
awe_modScalerDBInstance *PREFIX_scale;  
awe_modMeterInstance *PREFIX_inputMeter;  
awe_modAGCCoreInstance *PREFIX_agc_core;
```

Variable declarations for pins, wires, modules, and the overall layout.

The prefix specified was 'PREFIX\_' and thus this string appears at the start of each variable name.

```
awe_modAGCMultiplierInstance *PREFIX_agc_mult;
awe_modMeterInstance *PREFIX_outputMeter;
awe_modRouterInstance *PREFIX_autoOutputRouter_1;
awe_modFloatToFract32Instance *PREFIX_autoOutputConvert_1;
LayoutInstance *PREFIX_theLayout;

/* -----
** The initializer function. Returns E_SUCCESS if the allocation was
** successful. Otherwise, it returns an error code.
** ----- */

AWE_MOD_SLOW_CODE int test_Init()
{
    int retVal;

    /* -----
    ** Pin type data structures
    ** ----- */

    {
        uint args[] = {32, 2, 4, 48000, 0};
        PREFIX_audio2x32x48000_real = (PinInstanceDescriptor *)
ClassPin_Constructor(&retVal, 5, (Sample *) &args[0]);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    {
        uint args[] = {32, 4, 4, 48000, 0};
        PREFIX_audio4x32x48000_real = (PinInstanceDescriptor *)
ClassPin_Constructor(&retVal, 5, (Sample *) &args[0]);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    {
        uint args[] = {32, 1, 4, 48000, 0};
        PREFIX_audio1x32x48000_real = (PinInstanceDescriptor *)
ClassPin_Constructor(&retVal, 5, (Sample *) &args[0]);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    /* -----
    ** Wire data structures
    ** ----- */

    {
        PREFIX_wire1 = (WireInstance *) ClassWire_Constructor(&retVal, 1, (Sample *)
&PREFIX_audio2x32x48000_real);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    {
        PREFIX_wire2 = (WireInstance *) ClassWire_Constructor(&retVal, 1, (Sample *)
&PREFIX_audio4x32x48000_real);
        if (retVal != E_SUCCESS)
            return(retVal);
    }
}
```

Entry point for the function. There are no arguments and it returns an integer status result defined in Errors.h.

Allocate each pin

Allocate each wire.

```
    }

    {
        PREFIX_wire3 = (WireInstance *) ClassWire_Constructor(&retVal, 1, (Sample *)
&PREFIX_audio2x32x48000_real);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    {
        PREFIX_wire4 = (WireInstance *) ClassWire_Constructor(&retVal, 1, (Sample *)
&PREFIX_audio1x32x48000_real);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    {
        PREFIX_wire5 = (WireInstance *) ClassWire_Constructor(&retVal, 1, (Sample *)
&PREFIX_audio2x32x48000_real);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    {
        PREFIX_wire6 = (WireInstance *) ClassWire_Constructor(&retVal, 1, (Sample *)
&PREFIX_audio4x32x48000_real);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    /* -----
    ** Bind input and output wires to the system I/O pins
    ** ----- */

    {
        uint args[] = {(uint) PREFIX_wire1, (uint) &InterleavedInputPin};
        retVal = BindIOToWire(2, (Sample *) &args[0]);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    {
        uint args[] = {(uint) PREFIX_wire2, (uint) &InterleavedOutputPin};
        retVal = BindIOToWire(2, (Sample *) &args[0]);
        if (retVal != E_SUCCESS)
            return(retVal);
    }

    /* -----
    ** Instantiate all of the modules
    ** ----- */

    {
        WireInstance *pWires[] = { PREFIX_wire1,PREFIX_wire3 };
        PREFIX_autoInputConvert_1 = (awe_modFract32ToFloatInstance *)
ClassModule_Constructor ( (uint) CLASSID_FRACT32TOFLOAT, &retVal,
ClassModule_PackFlags(1, 1, 0), pWires, 0, (Sample *) NULL);
        if (retVal != E_SUCCESS)
            return(retVal);
    }
}
```

Bind wire1 to the system input pin and wire3 to the system output pin.

```

}

{
// [0] gainDB = 0
// [1] gain = 1
WireInstance *pWires[] = { PREFIX_wire3,PREFIX_wire3 };
uint args[] = { 0x00000000,0x3f800000 };
PREFIX_scale = (awe_modScalerDBInstance *) ClassModule_Constructor ( (uint)
CLASSID_SCALERDB, &retVal, ClassModule_PackFlags(1, 1, 0), pWires, 2, (Sample
*) &args[0]);
if (retVal != E_SUCCESS)
return(retVal);
}

{
// [0] meterType = 0
// [1] attackTime = 5
// [2] releaseTime = 1087
// [3] attackCoeff = 0.0095547
// [4] releaseCoeff = 4.416e-005
WireInstance *pWires[] = { PREFIX_wire3,PREFIX_wire4 };
uint args[] = { 0x00000000,0x40a00000,0x4487e000,0x3c1c8b37,0x38393888 };
PREFIX_inputMeter = (awe_modMeterInstance *) ClassModule_Constructor (
(uint) CLASSID_METER, &retVal, ClassModule_PackFlags(1, 0, 1), pWires, 5,
(Sample *) &args[0]);
if (retVal != E_SUCCESS)
return(retVal);
}

{
// [0] targetLevel = -20
// [1] maxAttenuation = 100
// [2] maxGain = 12
// [3] ratio = 10
// [4] activationThreshold = -50
// [5] smoothingTime = 100
// [6] enableRecovery = 1
// [7] recoveryRate = 2.4
// [8] currentGain = 0
// [9] oneOverSlope = 0.9
// [10] smoothingCoeffSample = 0.00020831
// [11] smoothingCoeffBlock = 0.0066445
// [12] recoveryRateUp = 1.0002
// [13] recoveryRateDown = 0.99982
// [14] targetGain = 0
// [15] energy = 0
// [16] oneOverNumSamples = 0.015625
WireInstance *pWires[] = { PREFIX_wire3,PREFIX_wire4 };
uint args[] = {
0xcla00000,0x42c80000,0x41400000,0x41200000,0xc2480000,0x42c80000,0x00000001,0
x4019999a,0x00000000,0x3f666666,0x395a6e3a,0x3bd9ba0e,0x3f800609,0x3f7ff3ee,0x
00000000,0x00000000,0x3c800000 };
PREFIX_agc_core = (awe_modAGCCoreInstance *) ClassModule_Constructor (
(uint) CLASSID_AGCCORE, &retVal, ClassModule_PackFlags(1, 1, 0), pWires, 17,
(Sample *) &args[0]);
if (retVal != E_SUCCESS)
return(retVal);
}

```

Instantiate each audio module. The constructor function takes an array of uint arguments. Comments show the actual value of each argument. The arguments are written to the first set of structure variables.

```
{
    WireInstance *pWires[] = { PREFIX_wire4,PREFIX_wire3,PREFIX_wire5 };
    PREFIX_agc_mult = (awe_modAGCMultiplierInstance *) ClassModule_Constructor (
(uint) CLASSID_AGCMULTIPLIER, &retVal, ClassModule_PackFlags(2, 1, 0), pWires,
0, (Sample *) NULL);
    if (retVal != E_SUCCESS)
        return(retVal);
}

{
    // [0] meterType = 0
    // [1] attackTime = 5
    // [2] releaseTime = 1087
    // [3] attackCoeff = 0.0095547
    // [4] releaseCoeff = 4.416e-005
    WireInstance *pWires[] = { PREFIX_wire5,PREFIX_wire4 };
    uint args[] = { 0x00000000,0x40a00000,0x4487e000,0x3c1c8b37,0x38393888 };
    PREFIX_outputMeter = (awe_modMeterInstance *) ClassModule_Constructor (
(uint) CLASSID_METER, &retVal, ClassModule_PackFlags(1, 0, 1), pWires, 5,
(Sample *) &args[0]);
    if (retVal != E_SUCCESS)
        return(retVal);
}

{
    WireInstance *pWires[] = { PREFIX_wire5,PREFIX_wire6 };
    PREFIX_autoOutputRouter_1 = (awe_modRouterInstance *)
ClassModule_Constructor ( (uint) CLASSID_ROUTER, &retVal,
ClassModule_PackFlags(1, 1, 0), pWires, 0, (Sample *) NULL);
    if (retVal != E_SUCCESS)
        return(retVal);
}

{
    int arrayInitializer[] = { 0,1,2,3 };
    awe_vecCopyInt32((int *) &arrayInitializer[0], 1, (int *)
&PREFIX_autoOutputRouter_1->channelIndex[0], 1, 4);
}

{
    WireInstance *pWires[] = { PREFIX_wire6,PREFIX_wire2 };
    PREFIX_autoOutputConvert_1 = (awe_modFloatToFract32Instance *)
ClassModule_Constructor ( (uint) CLASSID_FLOATTOFRACT32, &retVal,
ClassModule_PackFlags(1, 1, 0), pWires, 0, (Sample *) NULL);
    if (retVal != E_SUCCESS)
        return(retVal);
}

/* -----
** Instantiate the overall layout
** ----- */

{
    uint args[] = {1, (uint) PREFIX_autoInputConvert_1,(uint)
PREFIX_scale,(uint) PREFIX_inputMeter,
    (uint) PREFIX_agc_core,(uint) PREFIX_agc_mult,(uint)
PREFIX_outputMeter,(uint) PREFIX_autoOutputRouter_1,
    (uint) PREFIX_autoOutputConvert_1};
}
```

```
PREFIX_theLayout = (LayoutInstance *) ClassLayout_Constructor( &retVal, 9,
(Sample *) &args[0]);
if (retVal != E_SUCCESS)
    return(retVal);
}

return(E_SUCCESS);
}

#ifdef __cplusplus
}
#endif

/**
 * @}
 *
 * End of file.
 */
```

Create the overall audio processing layout. The modules are executed in the order shown in the args array. This is the same order that they were allocated in.

## 7.4. Comparison of Approaches

This chapter discussed 3 different ways of deploying a product using Audio Weaver. The methods differ primarily in where the description of the audio processing design is stored:

1. In the flash file system.
2. In a compiled file system built with the executable.
3. In C initializer code.

The preferred (and recommended) approach is to use a flash file system. This allows the audio processing to be redesigned without rebuilding the target executable. The commands are sent from the PC and then stored in flash. The binary commands provide a compact representation of the audio processing. In the case of the AGC example used throughout this chapter, the compiled script file `agc_example.awb` is only 552 bytes in length. Finally, by storing the commands in flash, we free up precious SRAM for other purposes.

If flash memory is not available, then use the compiled file system approach. This again leads to a fairly compact representation. Only 552 bytes of RAM are required (plus a little more for the directory structure). The main draw back is that the target executable must be rebuilt whenever there is a change to the audio processing. This approach also requires a set of development tools for the target processor.

Finally, the C initializer code has all of the drawbacks of the compiled file system, plus it is not as compact as the compiled file system. For example the function `test_Init()` in `agc_example_cinit.c` is 3228 bytes in length; much longer than the 552 bytes required to store the binary initialization commands.

## 8. Building the Target Executable

This section discusses practical issues related to building the Target executable using VisualDSP++. Since there are many items that need to be configured, it is recommended that an existing Target project is cloned rather than starting from scratch.

VisualDSP++4.5 June 2007 release should be used for building the SHARC targets or module libraries. VisualDSP++ 5.0 SP4 should be used for the Blackfin.

### 8.1. Memory Sections

The Audio Weaver module libraries and Framework code utilize a large number of memory sections. The memory sections are defined using macros in Framework.h and appear within the Linker Definition File (LDF). The memory sections apply only to embedded processors. On the PC, the macros expand to the empty string. The memory sections are particularly important on the SHARC. On the Blackfin with cache enabled, the sections must still be included in the LDF, but their placement is unimportant.

The sections related to code placement:

Macro	Section Name	Description
AWE_MOD_SLOW_CODE	awe_mod_slowcode	Non time-critical code used by the audio modules.
AWE_MOD_FAST_CODE	awe_mod_fastcode	Time-critical code used by the audio modules.
AWE_FW_SLOW_CODE	awe_fw_slowcode	Non time-critical Framework code.
AWE_FW_FAST_CODE	awe_fw_fastcode	Time-critical Framework code.

There are 24 memory sections related to data placement. The naming convention used by the macros is:

`AWE_{MOD/FW}_{SLOW/FAST}_{PM/DM/ANY}_{DATA/CONST}`

There are  $2 \times 2 \times 3 \times 3 = 24$  different permutations and each subfield has a specific meaning:

MOD/FW – specifies whether the section applies to modules (MOD) or to the Framework (FW).

SLOW/FAST – specifies the time criticality of the memory section. SLOW memory is referenced infrequently or from non-real-time code. It is suitable to be placed in external memory with little loss of performance. FAST memory is time-critical and should be placed internally. If internal placement of a FAST section is not

possible, then there will be a performance penalty.

PM/DM/ANY – specifies internal memory placement on the SHARC. PM and DM refer to two distinct internal memory blocks and enable parallel memory accesses. Traditionally, these are called "PM" and "DM" spaces on the SHARC. More recent processors with more than 2 internal memory blocks require placement in distinct blocks. ANY indicates that placement in PM or DM is unimportant; the section can go anywhere.

DATA/CONST – DATA sections must be placed in RAM and hold variables that are being updated. CONST sections store initialized constant data and are suitable for inclusion into ROM.

The corresponding section names (which must appear within the LDF) are lower case. For example, AWE\_MOD\_SLOW\_DM\_DATA is placed in the section awe\_mod\_slowdmdata.

The Audio Weaver Demo Board defines one additional memory section. This section is used to hold DMA data and must be in internal memory.

<b>Macro</b>	<b>Section Name</b>	<b>Description</b>
AWE_FAST_DMA_DATA	awe_fastdmadata	Holds the Platforms DMA buffers. Must be in internal memory on the SHARC.

### **8.2. Macro to Define when Building Module Libraries**

You must define the following macro when compiling an audio module source file:

```
AWE_INCLUDE_CONSTRUCTOR_FUNCTION
```

The macro enables the inclusion of the module's constructor function. If undefined, then the module cannot be dynamically allocated.

Undefine this macro if you do not need to use the module constructor functions.