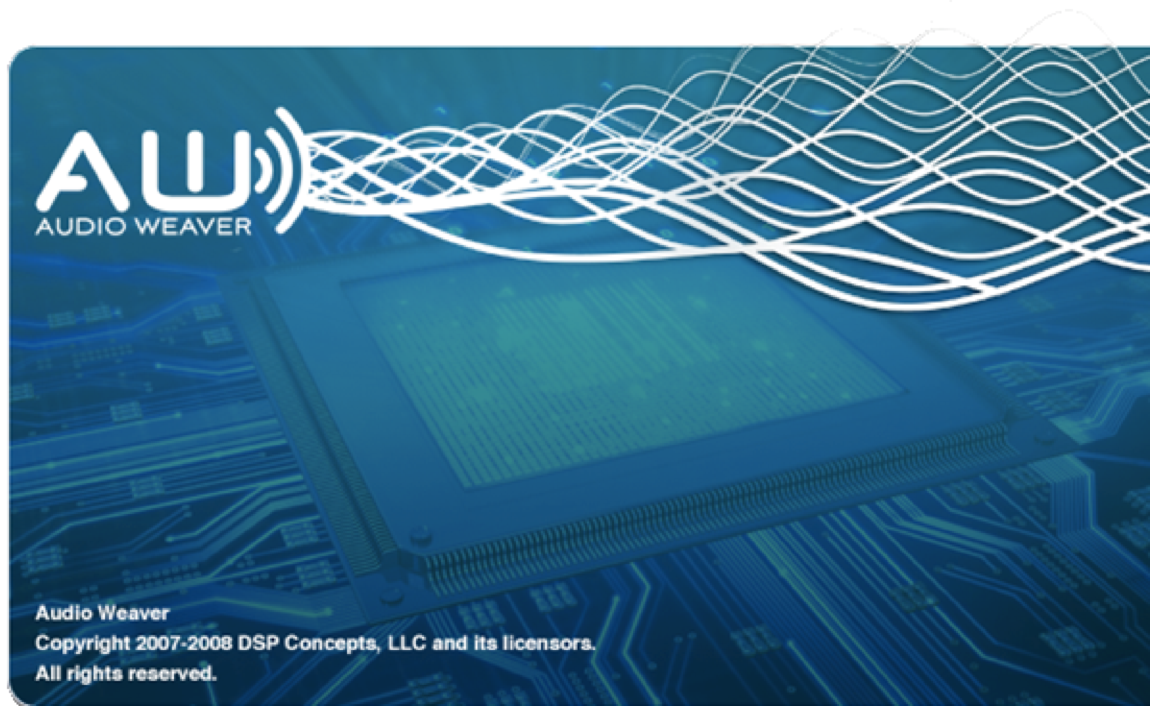
DSP Concepts, LLC.

Audio Weaver 2.0

Module 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. Overview ... 6
1.1. Requirements ... 6
1.2. Overall Development Environment ... 7
2. Scaler Smoothed Module Example ... 9
2.1. Module M-files ... 9
2.2. Audio Module Instance Structure ... 10
2.3. Audio Module Source Code (.c and .h files) ... 12
2.4. Audio Module Scheme File ... 13
2.5. Generating Module Code ... 14
2.6. Template Substitution ... 16
2.7. Building the Audio Module DLL ... 18
2.8. Summary of Steps Involved in Writing an Audio Module ... 18
2.9. Scaler Smoothed Example Source Files ... 19
3. MATLAB Function Reference ... 30
3.1. @awe_variable ... 30
3.2. @awe_module ... 35
3.3. @awe_subsystem ... 43
3.4. new_pin_type.m ... 46
3.5. add_pin.m ... 49
3.6. add_variable.m ... 50
3.7. add_array.m ... 51
3.8. add_module.m ... 52
3.9. connect.m ... 53

3.10. get_variable.m and set_variable.m 56
4. C Run-Time Environment 58
4.1. Data types and Structure Definitions 58
4.2. Class Object and Constructor Functions 63
4.3. Memory Allocation using awe_fwMalloc() 68
4.4. Module Function Details 69
5. Generating Module Libraries 76
5.1. awe_generate_library.m 76
5.2. Code Markers and Template Substitution 83
5.3. Fine Tuning Code Generation 89
5.4. unique_classes.m 90
5.5. awe_generate_module.m 92
6. Generating Documentation 93
6.1. awe_help 93
6.2. awe_generate_doc.m 94
6.3. Selecting the Documentation Format 95
6.4. Guidelines for Documenting Modules and Subsystems 95
6.5. Hilbert Module Example 96
6.6. Adding Equations to Documentation 99
7. The Examples Module Library 101
7.1. Scaler Module 101
7.2. Peak Hold Module 104
7.3. Fader module 105
7.4. Downsampler Module 105

7.5. Look Ahead Limiter Module 105

8. Building Audio Module DLLs 105

8.1. Building ExamplesDLL.dll 105

8.2. Cloning the Examples Library 105

9. VectorLib and ModuleHelperLib 105

1. Overview

This Developers Guide describes creating custom audio modules for Audio Weaver. It is a companion to the *Audio Weaver User's Guide* which focuses on developing audio systems using MATLAB scripts. This document is intended for algorithm developers creating custom audio modules.

This guide assumes that you've read through the *Audio Weaver User's Guide* and can develop audio systems using MATLAB scripts. This is particularly important for module developers since many of the key concepts – pins, variables, modules and system objects - will be built upon here. Modules are developed using a combination C or Assembly code and MATLAB scripts, and understanding the MATLAB scripts described in the Users Guide is critical. This Developers Guide is suitable for Audio Weaver Designer and Audio Weaver Developer. With Audio Weaver Designer, you will be limited to developing custom audio modules on the PC. Audio Weaver Developer gives you the added flexibility of creating custom audio modules for the SHARC or Blackfin processor.

If you are creating a custom hardware target for use with Audio Weaver, you should refer to the *Audio Weaver Platform Developers Guide.*

1.1. Requirements

Developing custom audio modules for the native PC target requires Audio Weaver Designer together with

- MATLAB version 7.3 (2006b) or later
- VisualStudio version 7 (2003)

The audio module code is compiled and packaged into an audio module Dynamic Link Library (DLL). The DLL is read by the Server when it launches and makes the audio modules in the DLL available to the Server.

To develop custom audio modules for the SHARC or Blackfin you'll need Audio Weaver Developer together with

- MATLAB version 7.3 (2006b) or later
- VisualStudio version 7 (2003)
- VisualDSP++ 4.5 2007 release for SHARC or VisualDSP++ 5.0 SP3 for Blackfin.

You first create an audio module DLL using VisualStudio; this describes the audio module to the Server. You then rebuild the target executable and link in your custom modules using VisualDSP++.

1.2. Overall Development Environment

The Audio Weaver environment utilizes MATLAB, a PC-based Server, a hardware target, and development tools for the hardware target, as shown in Figure 1. Much of the design is orchestrated by MATLAB including describing modules, generating code, and generating documentation.

Steps 1 through 9 shown above the dark line in Figure 1 are for creating custom audio module DLLs on the PC. The main output is an audio module DLL shown as item 9. The DLL encapsulates all of the information required by the Server to natively execute an audio module. This includes the module C functions – Constructor, Set, Get, Process, and Bypass – and schema information describing the module instance data structure to the Server. A DLL may contain one or more audio modules.

The custom module DLL is copied into the same directory as the AWE_Server executable. When the Server is launched, it loads all of the audio module DLLs contained in the executable directory and the new modules are then available for use.

Items 10 through 15 shown below the dark line in Figure 1 are associated with creating custom audio modules on an embedded target. You first create the custom audio module DLL on the PC and then compile the generated module source files (item 6) using VisualDSP++. Several libraries and source files are pulled together to create the new executable for the target (item 15).

Also shown in Figure 1 is a C code initializer file (item 20). This file is used when the target lacks a tuning interface, and initializes the audio processing to a known state.

This document focuses exclusively on items 1 through 9 – those associated with creating custom audio module DLLs on the PC. The *Audio Weaver Platform Developers Guide* takes you through the remaining steps needed to update the audio module library on an embedded target and to create C code initializers.

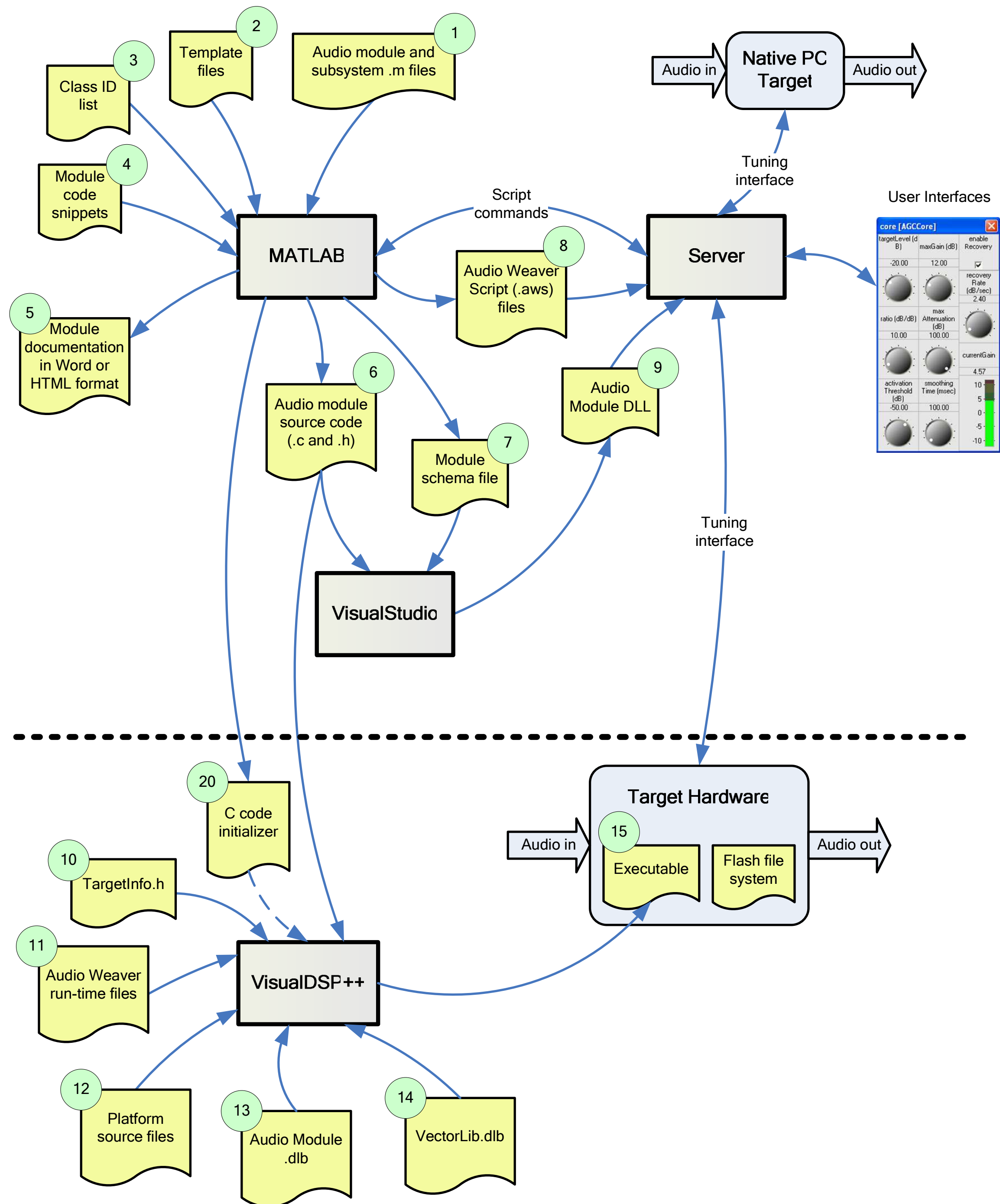


Figure 1. Overall Audio Weaver module design environment. The steps above the dark line are required to develop custom audio modules on the PC. The steps below the dark line are for developing custom audio modules for an embedded processor.

2. Scaler Smoothed Module Example

This section walks through a complete example of an audio module, starting from the high-level MATLAB model and continuing through the generated code. This example is identical to the scaler_smoothed_module.m contained in the BasicAudioFloat32 module library. We go through this module in a systematic fashion and reference the source files listed in Section 2.9. Section 7 contains additional module examples which highlight advanced features.

This module example is part of a larger audio module DLL containing several example modules. The base directory for the audio module examples is

```
<AWE>\ModuleLibs\Examples\
```

where <AWE> refers to the root directory of the Audio Weaver installation. The directory contains several subdirectories and this directory structure must be followed when creating other custom audio module libraries.

2.1. Module M-files

Each audio module has an associated module M-file, item 1 shown in Figure 1. For this example, the file is found in

```
<AWE>\ModuleLibs\Examples\matlab\scaler_smoothed_example_module.m
```

and a listing is shown in Section 2.9.1. When you first initialize Audio Weaver using awe_init.m, this file will be placed on your MATLAB path. The module m-file completely describes the audio module to MATLAB. It contains:

Input and output pins descriptions - Data types, number of channels, block sizes, sample rates, etc.

Instance structure variables – Date types, variable names, array sizes, memory allocation rules.

Documentation – Descriptions of variables, pins, and text describing the overall audio model.

MATLAB implementation of the module – Processing function (optional), and if needed, set function, bypass function, and get function. There may also be a prebuild function which is called when a system is built.

User interface – Individual variables are exposed and tied to controls. Controls are positioned and configured.

All of these items above are documented in the *Audio Weaver User's Guide*. The specific items

in the module m-file pertaining to module generation are variables, code markers, and wiring allocation guidelines described in Sections 5.1.11 and 5.2.

Subsystems are very similar to modules, but also contain a list of internal modules and a list of connections between modules.

2.2. Audio Module Instance Structure

Every audio module has an associated instance data structure that holds the variables – state and parameters – needed by the module. In this example, there are 4 variables described in the module m-file:

```
add_variable(M, 'gain', 'float', 0, 'parameter', 'Target gain');
M.gain.range=[-10 10];
M.gain.units='linear';

add_variable(M, 'smoothingTime', 'float', 10, 'parameter', 'Time
constant of the smoothing process');
M.smoothingTime.range=[0 1000];
M.smoothingTime.units='msec';

add_variable(M, 'currentGain', 'float', M.gain, 'state', 'Instantaneous
gain applied by the module.  This is also the starting gain of the
module.', 1);
M.currentGain.range=M.gain.range;
M.currentGain.units='linear';

add_variable(M, 'smoothingCoeff', 'float', NaN, 'derived', 'Smoothing
coefficient', 1);
```

The type definition for the instance data structure is contained in the file ModScalerSmoothedExample.h and is automatically generated by Audio Weaver:

```
typedef struct _awe_modScalerSmoothedExampleInstance
{
  ModuleInstanceDescriptor instance;
  float           gain;                // Target gain
  float           smoothingTime;       // Time constant of the ...
  float           currentGain;         // Instantaneous gain applied ...
  float           smoothingCoeff;      // Smoothing coefficient
} awe_modScalerSmoothedExampleInstance;
```

The instance structure begins with a common substructure of type ModuleInstanceDescriptor. This substructure points to the module's input and output wires, points to the real-time function currently used by the module (Processing, Bypassed, Muted, etc.), and points to the class instance structure.

The module header is followed by the instance variables described in the MATLAB file. There is a one-to-one correspondence between the variables shown in the instance structure and those added by MATLAB. The description of each variable is used as a comment.

The header file also contains a few other items. First, there is a bit mask for each variable in the data structure. The mask corresponds to the position of the variable within the instance structure. Note that the module header is 8 words long and thus the first bit starts in the 9th bit position. These bit masks can be used by the _Set() and _Get() functions to determine which variable within the instance structure has been modified.

```
#define MASK_ScalerSmoothedExample_gain 0x00000100
#define MASK_ScalerSmoothedExample_smoothingTime 0x00000400
#define MASK_ScalerSmoothedExample_currentGain 0x00000200
#define MASK_ScalerSmoothedExample_smoothingCoeff 0x00000800
```

The header file also defines an offset for each instance variable. Again, this is the offset from the start of the data structure, in 32-bit words. These offsets are provided to help implement system control using a host interface.

```
#define OFFSET_ScalerSmoothedExample_gain 0x00000008
#define OFFSET_ScalerSmoothedExample_smoothingTime 0x0000000A
#define OFFSET_ScalerSmoothedExample_currentGain 0x00000009
#define OFFSET_ScalerSmoothedExample_smoothingCoeff 0x0000000B
```

There is a unique class ID.

```
#define CLASSID_SCALERSMOOTHEDEXAMPLE (CLASS_ID_MODBASE + 32768)
```

This integer uniquely defines the module class to the Server. Next is the definition of the module's constructor function. In this case, since the module does not have any arrays, the generic module constructor ClassModule_Constructor[1] is used.

```
#ifdef AWE_INCLUDE_CONSTRUCTOR_FUNCTION
// This points the constructor for this class to the base constructor
#define awe_modScalerSmoothedExampleConstructor(ARG1, ARG2, ARG3, ARG4,
ARG5) ClassModule_Constructor(CLASSID_SCALERSMOOTHEDEXAMPLE, ARG1, ARG2,
ARG3, ARG4, ARG5)
#endif
```

And finally, there are definitions for the processing and set functions:

```
void awe_modScalerSmoothedExampleProcess(void *pInstance);

UINT awe_modScalerSmoothedExampleSet(void *pInstance, UINT mask);
```

Note that many of the items are surrounded by #ifdef / #endif pairs. This allows various features

[1] The generic module constructor applies to modules that do not have indirect arrays in the instance data structure. It is used by a large number of modules. The constructor function awe_modScalerSmoothedExample_Constructor() is provided as a macro so that the module can be constructed within subsystems. We'll see shortly that the constructor function is defined as NULL within the module's class structure.

of the modules to be enabled or disabled at compile time.

2.3. Audio Module Source Code (.c and .h files)

Each audio module has a set of associated functions. These functions, and their definitions, are placed in .c and .h files, respectively. At a minimum, every audio module requires a processing function. In addition, a module may specify several optional functions:

Constructor() – Performs additional memory allocation or initialization when a module is instantiated. If a module has an indirect array, such as an FIR filter, then a constructor function is required. In many cases, MATLAB can generate the constructor function automatically.

Set() – Implements a module's control functions which translate high-level interface variables to low-level variables. The Set() function is automatically called whenever an internal variable of a module is written by the Server during tuning.

Get() – Implements a module's control functions which translate low-level variables to high-level interface variables. The Get() function is automatically called by the Server whenever an internal variable of a module is read during tuning. Very few modules actually implement this function.

Bypass() – Implements custom bypass functionality. Required when one of the standard bypass functions is not suitable.

The module .c source file also contains a single *class structure* that describes the module to the Audio Weaver run-time and dynamic memory allocation functions. For example, this module contains the class structure:

```
AWE_MOD_SLOW_ANY_CONST
const ModClassModule awe_modScalerSmoothedExampleClass =
    {
#ifdef AWE_INCLUDE_CONSTRUCTOR_FUNCTION
      { CLASSID_SCALERSMOOTHEDEXAMPLE, NULL },
#else
      { CLASSID_SCALERSMOOTHEDEXAMPLE, NULL },
#endif
      ClassModule_PackArgCounts(4, 0),//(Public words, private words)
      awe_modScalerSmoothedExampleProcess,      // Processing function
      IOMatchUpModule_Bypass,                   // Bypass function
      awe_modScalerSmoothedExampleSet,          // Set function
      0,                                        // Get function
    };
```

The class structure contains pointers to the 5 functions listed above. The constructor function is NULL indicating that the module requires no memory outside of its instance structure and that the generic constructor should be used instead.

The class ID, CLASSID_SCALERSMOOTHEDEXAMPLE which is defined above, is a unique integer identifying the module to the Server. When instantiating a module of this class, the

Server is told to instantiate a module of class "ModuleScalerSmoothedExample". This string identifier is then translated to the underlying class ID through the *module DLL* shown as item 9 in Figure 1. Ensuring that each audio module has a unique classID can be tedious. To facilitate this process, each module library has a text file named classids.csv, shown as item 3 in Figure 1, containing a comma separated list of audio modules and their class IDs. A portion of the file is shown below:

```
IDOFFSET=32768

ScalerSmoothedExample,0
ScalerExample,1
FaderExample,2
FaderExampleFract32,3
PeakHoldExample,4
DownsamplerExample,5
```

The classID for the ScalerSmoothedExample module equals the IDOFFSET (32768) plus the value listed next to its class name below (0). Class IDs starting with 32768 are reserved for custom audio modules.

2.4. Audio Module Scheme File

The schema file provide a "blueprint" of all of the available objects on the target processor to the Server. This enables the Audio Weaver Server to instantiate and manipulate objects by name rather than hexadecimal IDs or offsets. The schema file for each module pack is embedded in the module DLL and is shown as item 9 in Figure 1. It contains definitions for all of the available audio modules in the module pack.

The entire schema file for the example module library is shown in Section 2.9.7. The portion corresponding to the ScalerSmoothedExample module is shown below.

```
ModuleScalerSmoothedExample 0x0xBEEF8800, BaseModule

{

      gain                    float      // Target gain
      smoothingTime           float      // Time constant ...
      currentGain             float      // Instantaneous gain ...
      smoothingCoeff          float      // Smoothing coefficient
}
```

Note the one-to-one correspondence between the schema description and the C type definition shown above. Furthermore, the C substructure ModuleInstanceDescriptor maps to the BaseModule descriptor in the schema file.

To create an instance of a smoothly varying scaler on the target, the Server is told "create an instance of class ModuleScalerSmoothedExample and call it Scale1". The Server converts this to the command "create an instance of object 0xBEEF8800" via the schema file. The

hexadecimal value 0xBEEF0C16 corresponds to the unique class ID

```
CLASS_ID_MODBASE + 32768
```

contained in the class structure[2]. On the target, the array of module class objects is traversed looking for class ID 0xBEEF8800. Once the class object is found, the corresponding constructor function, stored as a pointer in the class structure, is used to allocate one instance of the object. The address of the newly instantiated object is returned to the Server, and the Server associates the object's address with the name "Scale1". The Server maintains this mapping between symbol names and actual memory addresses.

The schema file also lists the module's instance variables and is used to compute offsets from the start of an object. When the value "Scale1.gain" is set during tuning, the Server translates this to a specific address on the DSP (PC) using the "Scale1" object address as a base, and then adding an appropriate offset. In this case, the offset equals 8, the size of BaseModule.

2.5. Generating Module Code

The MATLAB file scaler_smoothed_example_module.m has a complete description of the audio module with sufficient information to generate all of the source files and documentation. You can generate the source files for a smoothly varying scaler using the commands:

```
M=scaler_smoothed_example_module('temp');
awe_generate_module(M);
```

The files

```
Include/ModScalerSmoothedExample.h
Source/ModScalerSmoothedExample.c
```

will be created in MATLAB's current working directory. You can specify a particular directory for code generation by using an optional second argument:

```
awe_generate_module(M, DIR);
```

You can also specify that the C code should be formatted using the ident.exe utility (supplied) via a third argument:

```
awe_generate_module(M, DIR, 1);
```

Modules are usually not created in isolation, but are part of a particular module pack library. Modules need to be created as a unified library so that the combined schema file

[2] As a side note, CLASS_ID_MODBASE = 0x BEEF0800, which is defined in Framework.h.

ExamplesSchema.sch can be written. The MATLAB script make_examples.m generates code for the entire Examples module library. The script first creates a cell array of audio modules:

```
MM=cell(0,0);
MM{end+1}=downsampler_example_module('temp');
MM{end+1}=fader_example_fract32_module('temp');
MM{end+1}=fader_example_module('temp');
MM{end+1}=lah_limiter_example_module('temp');
MM{end+1}=peak_hold_example_fract32_module('temp');
MM{end+1}=peak_hold_example_module('temp');
MM{end+1}=scaler_example_module('temp');
MM{end+1}=scaler_smoothed_example_module('temp');
```

Some of other example audio modules are subsystems that utilize modules found in other libraries. We indicate this dependency relationship using the lines

```
[DependMM, DependName]=make_core32(0);
USESLIB{1}.MM=DependMM;
USESLIB{1}.str=DependName;
[DependMM, DependName]=make_basicaudiofloat32(0);
USESLIB{2}.MM=DependMM;
USESLIB{2}.str=DependName;
[DependMM, DependName]=make_basicaudiofract32(0);
USESLIB{3}.MM=DependMM;
USESLIB{3}.str=DependName;
```

Then call a function to generate the source code for the overall library:

```
awe_generate_library(MM, DIR, 'Examples', USESLIB, GENDOC);
```

Code for each module in a module pack is separately generated followed by the combined schema file. The third argument, 'Examples', specifies the module pack name and the forth argument specifies the dependencies on other module packs. The fifth argument, GENDOC, controls whether documentation in HTML format should be generated.

When you run the MATLAB script make_examples you'll see something similar to the following written to your MATLAB output window

```
Generating code for module: DownsamplerExample

Unchanged output file:

<DIR >\Source\ModuleLibs\Examples\include\ModDownsamplerExample.h

Unchanged output file:

 <DIR >\Source\ModuleLibs\Examples\Source\ModDownsamplerExample.c

Generating code for module: FaderExampleFract32

Unchanged output file:

<DIR>\Source\ModuleLibs\Examples\include\ModFaderExampleFract32.h
```

```
Unchanged output file:

 <DIR >\Source\ModuleLibs\Examples\Source\ModFaderExampleFract32.c

...

Generated a total of 8 modules
```

You'll note that all source files are marked as "Unchanged". Audio Weaver only overwrites generated files if there has been an actual change. This minimizes file changes and makes it easier to work with source code control systems.

2.6. Template Substitution

Audio Weaver uses template substitution to generate audio module source and header files. The template files are shown as item 2 in Figure 1, and two files are used

```
<AWE>\matlab\module_generation\templates\awe_module_template.c
<AWE>\matlab\module_generation\templates\awe_module_template.h
```

The template files contain boiler plate text together with string substitution variables and special preprocessor directives. Consider a portion of the file awe_module_template.h:

```
/*
 * @file $baseHFileName$
 */

#ifndef _MOD_$className$_H
#define _MOD_$className$_H

#include "ModCommon.h"
#include "MathHelper.h"
#include "FilterDesign.h"
$hFileInclude$

$hFileDefine$
```

Each substitution point is shown as \$NAME\$ in the template file. These substitution points are replaced with the code markers added via awe_addcodemarker.m commands in the module file. For example, when generating the scaler_smoothed_example_module, the string variable \$baseHFileName\$ is replaced by "ModScalerSmoothedExample.h" and \$className\$ is replaced by "ScalerSmoothedExample" to yield:

```
/*
 * @file ModScalerSmoothedExample.h
 */


#ifndef _MOD_ScalerSmoothedExample_H
#define _MOD_ScalerSmoothedExample_H

#include "ModCommon.h"
```

```
#include "MathHelper.h"
#include "FilterDesign.h"
```

By making changes to the template files, you can make wholesale changes to the entire module library!

The template files also contain special forms of preprocessor directives – independent of the standard C preprocessor directives. Further down in awe_module_template.h, you'll find the function declarations:

```
void $processFunctionName$(void *pInstance);

##if $useCustomSetFunction$
UINT $setFunctionName$(void *pInstance, UINT mask);
##endif

##if $useCustomGetFunction$
UINT $getFunctionName$(void *pInstance);
##endif

##if $useCustomBypassFunction$
void $bypassFunctionName$(void *pInstance);
##endif
```

The symbol ## identifies a preprocessor directive reserved for the code generator. The directives are evaluated during code generation based on the values of the variables, for example the value of $useCustomSetFunction$ determines if code will be eliminated or exist in the generated file. The scaler smoothed example module has

```
$useCustomSetFunction$ defined as 1
$useCustomGetFunction$ defined as 0
$useCustomBypassFunction$ defined as 0.
```

This yields the generated code:

```
void ModuleScalerSmoothedExample_Process(void *pInstance);

UINT ModuleScalerSmoothedExample_Set(void *pInstance, UINT mask);
```

The variables used during template substitution are referred to as "code markers". Many code markers are generated automatically by the awe_generate_module.m function. Some code markers are explicitly defined in scaler_smoothed_example_module.m.

```
awe_addcodemarker(M, 'processFunction',
'Insert:code\InnerScalerSmoothed_Process.c');
awe_addcodemarker(M, 'setFunction',
'Insert:code\InnerScalerSmoothed_Set.c');
awe_addcodemarker(M, 'srcFileInclude', '#include "FilterDesign.h"');
```

Code markers beginning with the string "Insert:" cause input to be read from a specified file. For example, the processing function template in awe_module_template.c is:

```
AWE_FAST_CODE
  void $processFunctionName$(void *pInstance)
  {
##if $usePreProcessFunction$
    {
    $preProcessFunction$
    }
##endif
    $processFunction$
##if $usePostProcessFunction$
      {
    $postProcessFunction$
      }
##endif
      }
```

The string $processFunction$ is taken from the file

```
code\InnerScalerSmoothedExample_Process.c
```

shown in Section 2.9.2. You'll note that this is bare code missing even the function definition (which is in the template file). The other code markers – "preProcessFunction" and "postProcessFunction" are not defined. After template substitution, we end up with the final processing function ModScalerSmoothedExample.c shown in Section 2.9.5.

2.7. Building the Audio Module DLL

At this point, the source code for all of the modules in the Examples library has been generated. Open up and rebuild the VisualStudio solution file

```
<AWE>\ModuleLibs\Examples\Examples.sln
```

The solution builds the DLL and then copies it into the

```
<AWE>\Bin
```

directory so that it can be referenced by the Server. Further details on this step can be found in Section 8.1.

2.8. Summary of Steps Involved in Writing an Audio Module

To summarize, the steps required in writing a new audio module are:

1. Create the MATLAB module m-file described in Section 2.1. It defines

    a. Input and output pins

    b. Instance structure variables

   c. Links to inner C code.

   d. Documentation

   e. User interface

2. Write the inner C code for the processing function.

3. Write the inner C code for the other module functions, Constructor(), Set(), Get(), and Bypass(), if needed.

4. Pick a unique integer ID (any number in the range 32768 to 63487) for the new module class ID. Add this information to the file classids.csv associated with the module library.

5. Add the module function to the specific module pack library generation script, for example make_examples.m. Run the script file to generate the source code and create the schema file.

6. Add the generated .c and .h files to the project for building the audio module pack library (e.g., BasicAudioFloat32Lib.vcproj). And build the library.

7. Build the audio module DLL so that the new modules are visible by the Server.

2.9. Scaler Smoothed Example Source Files

2.9.1. scaler_smoothed_example_module.m

```
function M=scaler_smoothed_example_module(NAME)
% M=scaler_smoothed_example_module(NAME)
% Creates a smoothly varying scaler module with a single input
% and single output pin.  This module operates on floating-point
% signals.  Arguments:
%    NAME - name of the module.

% Copyright 2007.  DSP Concepts, LLC.  All Rights Reserved.
% Author:  Paul Beckmann
%
% $Revision: 1.1 $

% AudioWeaverModule [This tag makes it appear under awe_help]

% ----------------------------------------------------------------------
% Create the high-level object with interface variables only.
% ----------------------------------------------------------------------

M=awe_module('ScalerSmoothedExample', 'Linear multichannel smoothly varying
scaler');
M.name=NAME;
M.preBuildFunc=@scaler_smoothed_example_prebuild;
M.processFunc=@scaler_smoothed_example_process;
M.setFunc=@scaler_smoothed_example_set;
```

```
PT=new_pin_type;

add_pin(M, 'input', 'in', 'audio input', PT);
add_pin(M, 'output', 'out', 'audio output', PT);

add_variable(M, 'gain', 'float', 0, 'parameter', 'Target gain');
M.gain.range=[-10 10];
M.gain.units='linear';

add_variable(M, 'smoothingTime', 'float', 10, 'parameter', 'Time constant of
the smoothing process');
M.smoothingTime.range=[0 1000];
M.smoothingTime.units='msec';

add_variable(M, 'currentGain', 'float', M.gain, 'state', 'Instantaneous gain
applied by the module.  This is also the starting gain of the module.', 1);
M.currentGain.range=M.gain.range;
M.currentGain.units='linear';

add_variable(M, 'smoothingCoeff', 'float', NaN, 'derived', 'Smoothing
coefficient', 1);

awe_addcodemarker(M, 'processFunction',
'Insert:code\InnerScalerSmoothedExample_Process.c');
awe_addcodemarker(M, 'setFunction',
'Insert:code\InnerScalerSmoothedExample_Set.c');
awe_addcodemarker(M, 'srcFileInclude', '#include "FilterDesign.h"');
M.wireAllocation='across';

% ----------------------------------------------------------------------
% Documentation
% ----------------------------------------------------------------------

M.docInfo.discussion={'Scales all input channels by a single gain value. ',
...
    'Changes to the gain parameter are exponentially smoothed (first order
IIR) at the sample rate, with the time constant determined by the
smoothingTime parameter. ', ...
    'This module is controlled by varying the gain variable.  Internally,
currentGain represents the instantaneous smoothed gain that is applied. ', ...
    'currentGain exponentially approaches gain with a time constant equal to
smoothingTime. ', ...
    '', ...
    'The module''s prebuild function initializes the currentGain equal to the
gain.  Thus, the module begins in a converged state.'};

% ----------------------------------------------------------------------
% Add the inspector information
% ----------------------------------------------------------------------

M.guiInfo.isExpanded=0;

M.gain.guiInfo.controlType='slider';
add_control(M, '.gain');

add_control(M, '.moduleStatus', 'right', 1);
add_control(M, '.smoothingTime', 'below', 1);
```

```
% ----------------------------------------------------------------------
% Call the prebuild function and force the low-level variables to be
% added.
% ----------------------------------------------------------------------

return;

% ----------------------------------------------------------------------
% Prebuild function.  Behavior is based on the data type of the
% input pin
% ----------------------------------------------------------------------

function M=scaler_smoothed_example_prebuild(M, FORCE)

M.currentGain=M.gain;
M.currentGain.range=M.gain.range;

% Propagate the type of the input pin to the output
M.outputPin{1}.type=M.inputPin{1}.type;

return;

% ----------------------------------------------------------------------
% Set function.  Computes the smoothing coefficient
% ----------------------------------------------------------------------

function M=scaler_smoothed_example_set(M)

% Compute the smoothing coefficient based on the smoothing time
SR=M.inputPin{1}.type.sampleRate;
M.smoothingCoeff = design_smoother(M.smoothingTime, SR, 1);

return;
```

2.9.2. InnerScalerSmoothedExample_Process.c

```
awe_modScalerSmoothedExampleInstance *S =
(awe_modScalerSmoothedExampleInstance *)pInstance;
WireInstance **pWires = ClassModule_GetWires(S);
float targetGain = S->gain;
float smoothingCoeff = S->smoothingCoeff;
float *src = (float *)pWires[0]->buffer;
float *dst = (float *)pWires[1]->buffer;
UINT channels = ClassWire_GetChannelCount(pWires[0]);
UINT blockSize = ClassWire_GetBlockSize(pWires[0]);
float currentGain;
UINT i;

for (i = 0; i < channels; i++)
{
  /* The same currentGain is used for each channel. Then we store the result
     from the final channel back into the state. */
  currentGain = S->currentGain;
  awe_vecScaleSmooth(src + i, channels, dst + i, channels, &currentGain,
targetGain,
              smoothingCoeff, blockSize);
}
S->currentGain = currentGain;
```

2.9.3. InnerScalerSmoothedExample_Set.c

```
awe_modScalerSmoothedExampleInstance *S =
(awe_modScalerSmoothedExampleInstance *) pInstance;
WireInstance **pWires = ClassModule_GetWires(S);
float SR;

if (mask & MASK_ScalerSmoothedExample_smoothingTime)
{
  SR = (float) ClassWire_GetSampleRate(pWires[0]);

  S->smoothingCoeff = design_smoother(S->smoothingTime, SR, 1);
}

return(0);
```

2.9.4. ModScalerSmoothedExample.h

```
/**
 *    @addtogroup Modules
 *    @{
 */

/****************************************************************************
*
*           Audio Framework
*           ---------------
*
****************************************************************************
*     ModScalerSmoothedExample.h
****************************************************************************
*
*     Description:      Linear multichannel smoothly varying scaler
*
*         Copyright:    DSP Concepts, LLC, 2007
*
***************************************************************************/

/*
 * @file ModScalerSmoothedExample.h
 */


#ifndef _MOD_SCALERSMOOTHEDEXAMPLE_H
#define _MOD_SCALERSMOOTHEDEXAMPLE_H

#include "ModCommon.h"
#include "MathHelper.h"

#define MASK_ScalerSmoothedExample_gain 0x00000100
#define MASK_ScalerSmoothedExample_smoothingTime 0x00000200
#define MASK_ScalerSmoothedExample_currentGain 0x00000400
#define MASK_ScalerSmoothedExample_smoothingCoeff 0x00000800
#define OFFSET_ScalerSmoothedExample_gain 0x00000008
#define OFFSET_ScalerSmoothedExample_smoothingTime 0x00000009
#define OFFSET_ScalerSmoothedExample_currentGain 0x0000000A
#define OFFSET_ScalerSmoothedExample_smoothingCoeff 0x0000000B
```

```
#define CLASSID_SCALERSMOOTHEDEXAMPLE (CLASS_ID_MODBASE + 32768)

#ifdef __cplusplus
extern "C" {
#endif

// ----------------------------------------------------------------------
// Overall instance class
// ----------------------------------------------------------------------

typedef struct _awe_modScalerSmoothedExampleInstance
{
  ModuleInstanceDescriptor instance;
      float       gain;               // Target gain
      float       smoothingTime;      // Time constant of the smoothing
process
      float       currentGain;        // Instantaneous gain applied by the
module.  This is also the starting gain of the module.
      float       smoothingCoeff;     // Smoothing coefficient

} awe_modScalerSmoothedExampleInstance;

#ifdef AWE_INCLUDE_CLASS_OBJECT
extern const ModClassModule awe_modScalerSmoothedExampleClass;
#endif

#ifdef AWE_INCLUDE_CONSTRUCTOR_FUNCTION
// This points the constructor for this class to the base constructor
#define awe_modScalerSmoothedExampleConstructor(ARG1, ARG2, ARG3, ARG4, ARG5)
ClassModule_Constructor(CLASSID_SCALERSMOOTHEDEXAMPLE, ARG1, ARG2, ARG3, ARG4,
ARG5)
#endif


void awe_modScalerSmoothedExampleProcess(void *pInstance);

UINT awe_modScalerSmoothedExampleSet(void *pInstance, UINT mask);



#ifdef __cplusplus
}
#endif

#endif // _MOD_SCALERSMOOTHEDEXAMPLE_H

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

2.9.5. ModScalerSmoothedExample.c

```
/**
 *    @addtogroup Modules
 *    @{
 */
```

```
/****************************************************************************
 *
 *          Audio Framework
 *          ---------------
 *
 ****************************************************************************
 *
 ****************************************************************************
 *
 *      Description:    Linear multichannel smoothly varying scaler
 *
 *        Copyright:    DSP Concepts, LLC, 2007
 *
 ***************************************************************************/

/*
 * @file
 */


#include "Framework.h"
#include "Errors.h"
#include "VectorLib.h"
#include "ModScalerSmoothedExample.h"

#include "FilterDesign.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

/* -----------------------------------------------------------------------
** Audio module class object.  This describes the audio module to the
** framework.  It contains pointers to functions and number of
** variables.
** -------------------------------------------------------------------- */

AWE_MOD_SLOW_ANY_CONST
  const ModClassModule awe_modScalerSmoothedExampleClass =
    {
#ifdef AWE_INCLUDE_CONSTRUCTOR_FUNCTION
      { CLASSID_SCALERSMOOTHEDEXAMPLE, NULL },
#else
      { CLASSID_SCALERSMOOTHEDEXAMPLE, NULL },
#endif
      ClassModule_PackArgCounts(4, 0),    // (Public words, private words)
      awe_modScalerSmoothedExampleProcess,                        //
```

```
Processing function
      IOMatchUpModule_Bypass,                         // Bypass function
      awe_modScalerSmoothedExampleSet,                            // Set function
      0,                                    // Get function
    };

#ifdef AWE_INCLUDE_CONSTRUCTOR_FUNCTION
/* ----------------------------------------------------------------------
** Memory allocation function.  This is required because the module
** requires additional memory outside of its instance structure.
** ------------------------------------------------------------------- */

AWE_MOD_SLOW_CODE
#endif

/* ----------------------------------------------------------------------
** Real-time Processing function.
** ------------------------------------------------------------------- */

AWE_MOD_FAST_CODE
  void awe_modScalerSmoothedExampleProcess(void *pInstance)
  {
    awe_modScalerSmoothedExampleInstance *S =
(awe_modScalerSmoothedExampleInstance *)pInstance;
    WireInstance **pWires = ClassModule_GetWires(S);
    float targetGain = S->gain;
    float smoothingCoeff = S->smoothingCoeff;
    float *src = (float *)pWires[0]->buffer;
    float *dst = (float *)pWires[1]->buffer;
    UINT channels = ClassWire_GetChannelCount(pWires[0]);
    UINT blockSize = ClassWire_GetBlockSize(pWires[0]);
    float currentGain;
    UINT i;

    for (i = 0; i < channels; i++)
    {
      /* The same currentGain is used for each channel. Then we store the
result
         from the final channel back into the state. */
      currentGain = S->currentGain;
      awe_vecScaleSmooth(src + i, channels, dst + i, channels, &currentGain,
targetGain,
              smoothingCoeff, blockSize);
    }
    S->currentGain = currentGain;
      }

/* ----------------------------------------------------------------------
** Set function which updates derived parameters based on the
** module's interface variables.
** ------------------------------------------------------------------- */

AWE_MOD_SLOW_CODE
  UINT awe_modScalerSmoothedExampleSet(void *pInstance, UINT mask)
  {
    awe_modScalerSmoothedExampleInstance *S =
(awe_modScalerSmoothedExampleInstance *) pInstance;
    WireInstance **pWires = ClassModule_GetWires(S);
    float SR;
```

```
    if (mask & MASK_ScalerSmoothedExample_smoothingTime)
    {
      SR = (float) ClassWire_GetSampleRate(pWires[0]);

      S->smoothingCoeff = design_smoother(S->smoothingTime, SR, 1);
    }

    return(0);
      }

#ifdef __cplusplus
}
#endif

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

2.9.6. classids.csv

```
% Class ID list for the examples that are included in the Audio Weaver
% documentation.

IDOFFSET=32768

ScalerSmoothedExample,0
ScalerExample,1
FaderExample,2
FaderExampleFract32,3
PeakHoldExample,4
DownsamplerExample,5
LAHLimiterExample,6
PeakHoldExampleFract32,7
```

2.9.7. ExamplesSchema.sch

```
ModuleDownsamplerExample 0x0xBEEF8805, BaseModule

{

      D                        int        // Decimation factor.  1 out of every
D samples is output
}

ModuleFaderExampleFract32 0x0xBEEF8803, BaseModule

{

      scaleFval                float      // Scaler Front
      scaleBval                float      // Scaler Back
      smoothingTimeF           float      // Time constant of the smoothing
process
      smoothingTimeB           float      // Time constant of the smoothing
```

```
process
      scalerF                  *ModuleScalerSmoothedFract32 // Linear
multichannel smoothly varying scaler
      scalerB                  *ModuleScalerSmoothedFract32 // Linear
multichannel smoothly varying scaler
      inter                    *ModuleInterleave // Interleaves multiple audio
signals
}

ModuleFaderExample 0x0xBEEF8802, BaseModule

{

      fade                     float      // Front/back Balance.  +1 = front
only.  -1 = rear only.
      smoothingTime            float      // Time constant of the smoothing
process
      scalerF                  *ModuleScalerSmoothed // Linear multichannel
smoothly varying scaler
      scalerB                  *ModuleScalerSmoothed // Linear multichannel
smoothly varying scaler
      inter                    *ModuleInterleave // Interleaves multiple audio
signals
}

ModuleLAHLimiterExample 0x0xBEEF8806, BaseModule

{

      maxDelayTime             float      // Maximum delay time
      max_abs                  *ModuleMaxAbs // Computes the maximum absolute
value of all input channels on a sample-by-sample basis
      core                     *ModuleAGCLimiterCore // Soft knee gain computer
for use in peak limiters
      delay                    *ModuleDelayMsec // Time delay in which the
delay is specified in milliseconds
      mult                     *ModuleAGCMultiplier // Mono x N-channel
multiplier
}

ModulePeakHoldExampleFract32 0x0xBEEF8807, BaseModule

{

      Reset                    int        // reset the current peak values
      attackTime               float      // Envelope detector attack time
constant
      decayTime                float      // Envelope detector decay time
constant
      decayCoef                fract32    // Computed coefficient used for
decay
      attackCoef               fract32    // Computed coefficient used for
attack
      peakHold                 *fract32   // Array of peak values
      peakDecay                *fract32   // Array of decaying peak values
}

ModulePeakHoldExample 0x0xBEEF8804, BaseModule
```

```
{

      Reset                     int        // reset the current peak values
      attackTime                float      // Envelope detector attack time
constant
      decayTime                 float      // Envelope detector decay time
constant
      decayCoef                 float      // Computed coefficient used for
decay
      attackCoef                float      // Computed coefficient used for
attack
      peakHold                  *float     // Array of peak values
      peakDecay                 *float     // Array of decaying peak values
}

ModuleScalerExample 0x0xBEEF8801, BaseModule

{

      gain                      float      // Linear gain
}

ModuleScalerSmoothedExample 0x0xBEEF8800, BaseModule

{

      gain                      float      // Target gain
      smoothingTime             float      // Time constant of the smoothing
process
      currentGain               float      // Instantaneous gain applied by the
module.  This is also the starting gain of the module.
      smoothingCoeff            float      // Smoothing coefficient
}
```

2.9.8. ExamplesSchema.cpp

Not shown. This is a C file with initialized binary arrays.

2.9.9. Examples.h

```
extern const ModClassModule awe_modDownsamplerExampleClass;
extern const ModClassModule awe_modFaderExampleFract32Class;
extern const ModClassModule awe_modFaderExampleClass;
extern const ModClassModule awe_modLAHLimiterExampleClass;
extern const ModClassModule awe_modPeakHoldExampleFract32Class;
extern const ModClassModule awe_modPeakHoldExampleClass;
extern const ModClassModule awe_modScalerExampleClass;
extern const ModClassModule awe_modScalerSmoothedExampleClass;


#define LISTOFCLASSOBJECTS \
&awe_modDownsamplerExampleClass, \
&awe_modFaderExampleFract32Class, \
&awe_modFaderExampleClass, \
&awe_modLAHLimiterExampleClass, \
&awe_modPeakHoldExampleFract32Class, \
&awe_modPeakHoldExampleClass, \
```

```
&awe_modScalerExampleClass, \
&awe_modScalerSmoothedExampleClass

#define USESDLLS "BasicAudioFloat32DLL.dll,BasicAudioFract32DLL.dll"
```

3. MATLAB Function Reference

This section describes Audio Weaver's MATLAB functions related to module generation. Audio Weaver makes heavy use of MATLAB's object oriented features. An object in MATLAB is a data structure with associated functions or methods. Each type of object is referred to as a *class,* and Audio Weaver uses separate classes to represent variable, modules, and subsystems. The class functions are stored in directories that start with the "@" symbol. Under <AWE>\matlab\ are found 3 class directories:

```
@awe_variable\
@awe_module\
@awe_subsystem\
```

It is important to understand how to use and manipulate these classes in order to properly use all of the features of Audio Weaver. We begin by describing each class and then document additional MATLAB commands used for constructing modules.

3.1. @awe_variable

A variable in Audio Weaver represents a single scalar[3] or array variable on the target processor. Even if you are not developing modules, it is good to understand the @awe_variable object so that you can fully utilize variables.

A variable in Audio Weaver represents a single scalar or array variable on the target processor. A new scalar variable is created by the call:

```
awe_variable(NAME, TYPE, VALUE, USAGE, DESCRIPTION, ISHIDDEN, ISCOMPLEX)
```

Variables in Audio Weaver have a close correspondence to variables in the C language. The arguments are:

`NAME` – name of the variable as a string.

`TYPE` – C type of the variable as a string. For example, 'int' or 'float'.

`VALUE` – initial value of the variable.

`USAGE` – a string specifying how the variable is used in Audio Weaver. Possible values are:

'const' – the variable is initialized when the module is allocated and does not

[3] Scalar is a mathematical term and refers to a variable containing only a single value. Do not confuse this with a scaler which is a specific type of audio module.

change thereafter.

'parameter' – the variable can be changed at run-time and is exposed as a control.

'derived' – similar to a parameter, but the value of the variable is computed based on other parameters in the module. Derived variables are not exposed as controls.

'state' – the variable is set at run-time by the processing function.

`DESCRIPTION` – a string describing the function of the variable.

`ISHIDDEN` – an optional Boolean indicating whether the variable is visible (ISHIDDEN=0) or hidden (ISHIDDEN=1). By default, ISHIDDEN=0.

`ISCOMPLEX` – an optional Boolean indicating whether the variable is real valued (ISCOMPLEX=0) or complex (ISCOMPLEX=1). By default, ISCOMPLEX=0. Note, Audio Weaver only supports complex valued arrays, not scalars.

You typically do not use the awe_variable.m function directly. Rather, the function is automatically called when you add variables to modules or subsystems using the add_variable.m or add_array.m functions documented in Sections 3.6 and 3.7.

Only 32-bit data types are supported. Allowable types are: 'float', 'int', 'uint', 'and 'fract32'.

We now look carefully at one of the variables in the agc_example.m system. At the MATLAB prompt, type:

```
SYS=agc_example;
struct(get_variable(SYS.agc.core, 'targetLevel'))
```

The get_variable.m command extracts a single variable from a module and returns the @awe_module object. (If you instead try to access SYS.agc.core.targetLevel you'll only get the *value* of the variable, not the actual structure.) The MATLAB struct.m command turns an object into a data structure revealing each of its internal fields. You'll see:

```
                 name: 'targetLevel'
        hierarchyName: '.agc.core.targetLevel'
                value: -20
                 size: [1 1]
                 type: 'float'
            isComplex: 0
                range: [-50 50]
                usage: 'parameter'
          description: 'Target audio level'
            arrayHeap: ''
        memorySegment: 'AWE_FAST_ANY_DATA'
 arraySizeConstructor: ''
      constructorCode: ''
```

```
        guiInfo: [1x1 struct]
         format: '%g'
          units: 'dB'
         isLive: 1
     isVolatile: 1
       isHidden: 0
       isPreset: 1
        isArray: 0
     targetInfo: [1x1 struct]
     fieldNames: {24x1 cell}
       isLocked: 1
          class: 'awe_variable'
```

Many of the fields are set when the variable was added to the module or at build time. Some of them can be set after a module has been built.

name – string indicating the name of the variable. This was set when the variable was added to the module. Not user editable.

hierarchyName – hierarchical location of the variable in the overall system. Initially empty and then set by build.m. Not user editable.

value – current value of the variable. Although it can be read and written by the user, it is easier to access the value simply by referencing

```
SYS.agc.core.targetLevel
```

size – 1x2 element vector used to represent the size of matrices, [rows columns]. For scalars, this is always [1 1]. Set when the variable was added to the module or by the prebuild function. Not user editable.

type – string specifying the underlying data type of the variable. Allowable values are 'float', 'fract32', 'int', and 'uint'. Set when the variable was added to the module. Not user editable.

isComplex – Boolean indicating whether the variable contains complex data. Set when the variable was added to the module. Only arrays can be complex, not individual scalar variables. Not user editable.

range – a vector or matrix specifying the allowable range of the variable. This is used to validate variable updates and also to draw knobs and sliders. This vector uses the same format as the pin type described in Section 3.4. User editable.

usage – a string specifying how the variable is used in Audio Weaver. Set when the variable was added to the module. Not user editable. Possible values are:

‘const’ – the variable is initialized when the module is allocated and does not change thereafter.

'parameter' – the variable can be changed at run-time and is exposed as a control.

'derived' – similar to a parameter, but the value of the variable is computed based on other parameters in the module. Derived variables are not exposed as controls.

'state' – the variable is set at run-time by the processing function.

description - a string describing the purpose or usage of the variable. User editable.

arrayHeap – used to specify array allocation information to the code generator. User editable.

memorySegment - used to specify array allocation information to the code generator. User editable.

arraySizeConstructor - used to specify array allocation information to the code generator. User editable.

constructorCode - specifies variable initialization when a module exists within a subsystem and is used by the code generator. User editable.

guiInfo – structure used to hold GUI related information. Some fields are user settable. Refer to the chapter in the *Audio Weaver User's Guide* that discusses creating custom inspector interfaces.

format - C formatting string used by when displaying values in the MATLAB output window. Follows the formatting convention of the printf function. User editable.

units - a string containing the underlying units of the variable. For example, 'dB' or 'Hz'. This is used by documentation and on user interface panels. User editable.

isLive – Boolean variable indicating whether the variable is residing on the target (isLive = 1), or if it has not yet been built (isLive=0). This starts out equaling 0 when the module is instantiated and the set to 1 by build.m. Not user editable.

isVolatile – Boolean indicating whether the variable is changed outside of MATLAB and needs to be read from the target each time. Reading of variables from the target only occurs when isLive=1. By default, only 'const' variables have isVolatile set to 0; all others are set to 1. User editable.

isHidden – Boolean indicating whether a variable is hidden. Hidden variables are not shown when a subsystem is displayed in the MATLAB output window. However, hidden variables may still be referenced. User editable.

isPreset – Boolean indicating whether the variable is included in presets. Used by the

create_preset.m function. User editable.

isArray – Boolean indicating whether the variable is a scalar or an array. Scalars values occur in the instance data structure. Arrays have pointers in the instance data structure. This field is set when the variable is first instantiated and should not be changed thereafter.

targetInfo – internal data structure used when tuning the variable at run-time. Not user editable.

fieldNames – internal cell array of field names (actually the names of the fields in this data structure). It is used to accelerate references. Not user editable.

isLocked – internal field used to accelerate references. Not user editable.

class – string specifying the underlying object class. This is always 'awe_variable'. Not user editable.

The .isHidden field can be used to hide variable that the user typically does not need to know about. For example, allocate a 2nd order Biquad filter:

```
>> M=biquad_module('filter')
filter = Biquad // 2nd order IIR filter

  b0: 1          // First numerator coefficient
  b1: 0          // Second numerator coefficient
  b2: 0          // Third numerator coefficient
  a1: 0          // Second denominator coefficient
  a2: 0          // Third denominator coefficient
```

All 5 of the tunable filter coefficients are shown. After the filter is built, state variables are added. You can see them by typing:

```
>> M.state

ans =

     0
     0
```

Of course, this assumes that you know that this module has a variable named .state. To show all hidden variables in the MATLAB output window, set

```
AWE_INFO.displayControl.shownHidden=1;
```

Then, looking at the Biquad filter, the hidden .state variable will be shown as well:

```
filter = Biquad // 2nd order IIR filter
```

```
        b0: 1          // First numerator coefficient
        b1: 0          // Second numerator coefficient
        b2: 0          // Third numerator coefficient
        a1: 0          // Second denominator coefficient
        a2: 0          // Third denominator coefficient
     state: 0
              0]
```

3.2. @awe_module

This class represents a single primitive audio processing function on the target. A module consists of the following components: a set of names, input and output pins, variables, and functions. All of these items exist in MATLAB and many of them have duals on the target itself.

3.2.1. Class Object

To create an audio module object, call the function

```
M=awe_module(CLASSNAME, DESCRIPTION)
```

The first argument, CLASSNAME, is string specifying the class of the module. Each module must have a unique class name, and modules on the Server are instantiated by referencing their class name[4]. The second argument, DESCRIPTION, is a short description of the function of the module. The DESCRIPTION string is used when displaying the module or requesting help.

After the module class is created, set the particular *name* of the module:

```
M.name='moduleName';
```

Note that there is a distinction between the CLASSNAME and the .name of a module. The CLASSNAME is the unique identifier for the *type* of the module. For example, there are different class names for scalers and biquad filters. The .name identifies the module within a system and must be unique within the current level of hierarchy in the system. At this point, we have a bare module without inputs, outputs, variables, or associated functions. These must each be added.

We'll now look more closely at the fields within the @awe_module object. Instead of looking at a bare module as returned by awe_module.m, we'll look at a module that is part of a system and has already been built. We'll choose the core module within the agc_example:

```
SYS=agc_example;
struct(SYS.agc.core)
```

[4] The function classid_lookup.m can be used to determine if a class name is already in use. Refer to *Audio Weaver Module Developers Guide* for more information.

MATLAB displays

```
               name: 'core'
          className: 'AGCCore'
        description: 'Automatic Gain Control gain calculator module'
            classID: []
          mfilePath: [1x85 char]
     mfileDirectory:
'<DIR> \Source\ModuleLibs\BasicAudioFloat32\matlab'
          mfileName: 'agc_core_module.m'
               mode: 'Active'
       clockDivider: 1
           inputPin: {[1x1 struct]}
          outputPin: {[1x1 struct]}
         scratchPin: {}
           variable: {1x17 cell}
       variableName: {1x17 cell}
            control: {1x10 cell}
     wireAllocation: 'distinct'
          resetFunc: []
            getFunc: []
            setFunc: []
        processFunc: []
         bypassFunc: @generic_bypass
           muteFunc: @generic_mute
       preBuildFunc: @agc_core_prebuild_func
           isHidden: 0
           isPreset: 1
             isLive: 1
      hierarchyName: '.agc.core'
           hasFired: 1
         targetInfo: [1x1 struct]
         codeMarker: {[1x1 struct]  [1x1 struct]  [1x1 struct]  [1x1
struct]}
         isTopLevel: 0
            guiInfo: [1x1 struct]
           drawInfo: [1x1 struct]
            docInfo: [1x1 struct]
           isLocked: 1
              class: 'awe_module'
         fieldNames: {36x1 cell}
```

where

name – name of the module within the subsystem. This is specified when the module is instantiated and cannot be changed thereafter. Not user editable.

className – name of the underlying module class. This is specified when the module is instantiated and cannot be changed thereafter. Not user editable.

description – short comment indicating what the module does. User editable.

classID – unique integer which identifies the module class on the target. This value is set by the code generator and is normally blank. Not user editable.

mfilePath – full path to the m-file which instantiated the module (the one containing the call to @awe_module). Used by the code generator. Not user editable.

mfileDirectory – the directory portion of mfilePath. Not user editable.

mfileName – the file name portion of mfilePath. Not user editable.

mode – string indicating the current module status. This can be either 'Active', 'Bypassed', 'Muted', or 'Inactive'. Used internally and is not user editable. Use the functions awe_setstatus.m and awe_getstatus.m to manipulate the module status.

clockDivider – integer indicating how often the module will be run within the layout. This is currently always set to 1 (meaning run every time). This is reserved for new Audio Weaver functionality planned in the future. Not user editable.

inputPin – cell array of input pin information. This information is set by the add_pin.m function and should not be changed. You'll frequently access this to determine the properties of the input pins. Each cell value contains a data structure such as

```
SYS.agc.core.inputPin{1}

ans =

              type: [1x1 struct]
             usage: 'input'
              name: 'in'
       description: 'Audio input'
    referenceCount: 0
        isFeedback: 0
          drawInfo: [1x1 struct]
         wireIndex: 1
```

outputPin – similar to inputPin. It is a cell array describing the output pins.

scratchPin - similar to inputPin. It is a cell array describing the scratch pins.

variable – a cell array of @awe_variable objects corresponding to the variables in this module. This array is updated by the add_variable.m function. Not user editable.

variableName – a cell array of strings, one for each variable in the module. This cell array is used to speed up variable accesses. This array is updated by the add_variable.m command. Not user editable.

control – holds internal information related to the inspector. Not user editable.

wireAllocation – a string specifying how wires should be allocated for the module. Possibilites are 'across' and 'distinct'.

getFunc – optional MATLAB function pointer specifying the module's get function. This

function is called whenever a variable in the module is queried. Used very rarely. Normally this is set to the empty matrix.

setFunc – optional MATLAB function pointer specifying the module's set (or control) function.

processFunc – optional MATLAB function pointer specifying the module's processing function. The processing function is a MATLAB implementation of the audio processing performed by the module.

bypassFunc – optional MATLAB function pointer specifying the module's bypass behavior.

preBuildFunc – optional MATLAB function pointer specifying the module's prebuild functionality. The prebuild function is called prior to building the module or generating code and resolves pin types and array sizes.

isHidden – Boolean specifying whether the module should be shown when part of a subsystem. Similar to the .isHidden field of @awe_variable objects. User editable.

isPreset – Boolean that indicates whether the module will be included in generated presets. By default, this is set to 1. User editable.

isLive –Boolean indicating whether the module has been built and is running on the target. When a module is first instantiated, this is set to 0 and then changed to 1 by the build process. Not user editable.

hierarchyName – hiearchical name of the module within the subsystem. This is filled in during the build process. Not user editable.

hasFired – Boolean field that is used internally by the routing algorithm. Not user editable.

targetInfo – data structure holding target specific information. This is filled in by the build process and is used internally for tuning. Not user editable.

codeMarker – a cell array holding all of the code markers. Code markers are used for module generation and described in Section 5.2. Not user editable.

isTopLevel – Boolean set by the buid process. A value of 1 indicates that a module is the highest level item in a system. All modules have .isTopLevel=0; only the top-level subsystem has .isTopLevel=1. Not user editable.

guiInfo – data structure used for drawing inspectors. Some fields are user editable. See the Section in the *Audio Weaver User's Guide* that discusses creating custom

inspector interfaces.

drawInfo – data structure used internally when creating drawings of subsystems and modules. Not user editable.

docInfo – data structure containing documentation information. This is set in the module's m-file and is used by the automated documentation generator. Some user editable fields. Refer to the Section 6.

isLocked – internal field used to accelerate references. Not user editable.

class – string specifying the underlying object class. This is always 'awe_module'. Not user editable.

fieldNames – internal cell array of field names (actually the names of the fields in this data structure). It is used to accelerate references. Not user editable.

3.2.2. MATLAB Functions for Constructing Modules

The following functions are commonly used for constructing audio modules. They are briefly mentioned here and are documented later on in this guide.

add_variable.m – adds a scalar variable to an audio module object. See Section 3.6

add_array.m – adds an indirect array to an audio module object. See Section 3.7

add_pin.m – adds an input, output, or scratch pin to an audio module object. See Section 3.5.

add_codemarker.m – adds information related to code generation. See Section 5.2.

add_control.m – exposes a single control on the inspector. See the *Audio Weaver User's Guide*.

set_variable.m – replaces an existing module variable. See Section 3.10

get_variable.m – extracts a variable from a module and returns an @awe_variable object. See Section 3.10

3.2.3. Internal Module Functions

Every module in Audio Weaver has an associated MATLAB function file that constructs an @awe_module object describing the module. Audio Weaver uses the convention that these module constructor functions end in "_module.m". In addition to configuring the module's input and output pins, and defining instance variables, the constructor function also assigns a number of method functions, and these functions are typically contained as sub-functions in the

constructor file.

There is a close coupling between the module related C functions the module subfunctions on the target processor. The table below gives a brief description of the functions available.

MATLAB Function Name	Target Function Name	Purpose
`*_module.m`	`*_Constructor()`	Constructs an instance of a module class. The function allocates memory for the base instance structure. The MATLAB function sets default values while on the target default values are passed into the constructor. Required.
`.processFunc`	`*_Process()`	Real-time processing function. Required for C code. Optional for MATLAB.
`.setFunc`	`*_Set()`	Implements the module's control functionality. The function converts high-level interface parameters to lower-level derived values. This function should be called whenever a variable in a module is modified. Optional.
`.getFunc`	`*_Get()`	The counterpart to the _Set() function but is used when module variables are read. *Most modules do not use this function.* When a module instance variable is read, this function is first called and then the instance variable is returned. It could be used, for example, to convert a measurement from energy to dB.
`.bypassFunc`	`*_Bypass()`	Implements the module's bypass functionality. This is an optional function; when it is empty, a generic bypass function is used instead.
`.preBuildFunc`	`N/A`	This function exists only in MATLAB and it is called as part of the build procedure. The prebuild function propagates pin information and may update array sizes. The prebuild function is useful for setting variable values that depend upon the pin type. The prebuild function is optional; if it doesn't exist, the pin type information from the module's first input pin is propagated to the output.

3.2.3.1. MATLAB Module M-File

The MATLAB m-file associated with an audio module creates and returns an instance of the module. The typical function signature for a module m-file is

```
function M=new_module(NAME, ARG1, ARG2,…)
```

where NAME is a string module name and ARG1, ARG2, are optional arguments. Good practice is to set default values for all arguments except NAME. For example:

```
if (nargin < 2)
      ARG1=1;
end
if (nargin < 3)
      ARG2=3;
end
```

The main difference between the MATLAB instantiation function and the C constructor is that the C constructor is responsible for all memory allocation. MATLAB, on the other hand, has the option of doing memory allocation in the prebuild function.

3.2.3.2. .processFunc

This function provides a MATLAB implementation of the module's processing function. The function signature is:

```
function [M, WIRE_OUT]=new_process(M, WIRE_IN)
```

where

M – is the @awe_module object.

WIRE_IN – a cell array of corresponding to the input data to be processed by the module.

Note that M serves as both an input argument and output result of the function. The function modifies the state variables within M and then returns the updated object. WIRE_OUT is a cell array containing the output data.

Each element within the WIRE_IN cell array is an MxN matrix representing the input data at a particular pin. Each column of the input data represents a different channel. The audio data is processed and the result written to WIRE_OUT. For example, the processing function for the scaler_module.m computes

```
function [M, WIRE_OUT]=scaler_process(M, WIRE_IN)

WIRE_OUT=cell(size(WIRE_IN));

for i=1:length(WIRE_OUT)
    WIRE_OUT{i}=WIRE_IN{i}*M.gain;
```

```
end

return;
```

One difference between MATLAB and C is that the wires in MATLAB serve only to pass data in and out of the processing function. Pin information is determined in MATLAB from fields in the audio module structure. For example, to determine the sample rate of the first input, use

```
M.inputPin{1}.type.sampleRate
```

whereas in C, you would use

```
SR = (float) ClassWire_GetSampleRate(pWires[0]);
```

3.2.3.3. .setFunc

Implements the main control functional for a module. This function is called whenever a variable within the audio module is updated. The function signature is:

```
function M=new_set(M)
```

where

M – is the @awe_module object.

Note that the MATLAB set function does not have a MASK argument indicating which variable changed.

3.2.3.4. .getFunc

Implements secondary control functionality. This function is called when a variable in the module is read. The function signature is identical to the .setFunc

```
function M=new_get(M)
```

where

M – is the @awe_module object.

3.2.3.5. .bypassFunc

Provides a custom bypass function for a module. The function signature is identical to the .processFunc

```
function [M, WIRE_OUT]=new_bypass(M, WIRE_IN)
```

where

M – is the @awe_module object.

WIRE_IN – a cell array of corresponding to the input data to be processed by the module.

3.2.3.6. .preBuildFunc

This function is called when a system is built for execution on a target. The function updates any internal variables which depend upon the sample rate, block size, or number of channels. The function also implements non-standard pin propagation functionality. The function signature is:

```
function M=new_prebuild(M)
```

where

M – is the @awe_module object.

The function updates and returns the module object M.

3.3. @awe_subsystem

This class represents both systems and subsystems; they are equivalent and no distinction is made in Audio Weaver. A subsystem has all of the characteristics of an audio module: a class name, input and output pins, variables, and associated sub-functions. In addition, a subsystem contains two other items:

1. Internal modules.
2. Connections between the modules and the subsystem input and output pins.

Key subsystem functions are described next.

3.3.1. ClassObject

The call to create a new empty subsystem is similar to the call to create a new module described in Section 3.2.1

```
SYS=awe_subsystem(CLASSNAME, DESCRIPTION);
```

You have to provide a unique class name and a short description of the function of the subsystem. (To be precise, the CLASSNAME only has to be unique if you are generating C code using the new subsystem class. Then, the CLASSNAME has to be unique among all of the subsystems and audio modules.)

After the subsystem is created, set the particular name of the subsystem:

```
SYS.name='subsystemName';
```

At this point, we have an empty subsystem; no modules, pins, variables, or connections.

As before, we'll look at the internal fields of a subsystem and we'll use a running subsystem as an example. The @awe_subsystem object is derived from an @awe_module object. In fact, the first 36 fields of a subsystem are identical to a module object. The only new fields are:

```
           isPrebuilt: 1
       preProcessFunc: []
      postProcessFunc: []
            buildFunc: @pcserver_build
             drawFunc: []
               module: {1x6 cell}
           moduleName: {1x6 cell}
           connection: {1x7 cell}
       flattenOnBuild: 1
   targetSpecificInfo: [1x1 struct]
```

isPrebuilt – a Boolean specifying where the system has already been through the prebuild process. This is used internally and is not user editable.

preProcessFunc – pointer to a MATLAB function which is called prior to the subsystem being executed. It is the MATLAB dual to the 'preProcessFunc' code marker and is used for simulating subsystems in MATLAB.

postProcessFunc – pointer to a MATLAB function which is called after the subsystem has executed. It is the MATLAB dual to the 'postProcessFunc' code marker and is used for simulating subsystems in MATLAB.

buildFunc – pointer to a MATLAB function which builds the subsystem on the target. This is empty except for top level systems.

drawFunc – pointer to a MATLAB function which overrides drawing of the module in figure windows. This is not yet used and is provided for future enhancements.

module – cell array of the internal modules used by the subsystem. Not user editable.

moduleName – cell array of internal module names. This is used to accelerate references. Not user editator.

connection – cell array describing all of the connections in the subsystem. This array is populated by calls to connect.m. Not user editable.

flattenOnBuild – a Boolean describing how this subsystem is treated during code generation and building. When creating new module classes out of subsystems, set .flattenOnBuild = 1. See Section 7.3 for an example.

targetSpecificInfo – a data structure populated by the build process. It is used to enable

real-time tuning on a target. Not user editable.

3.3.2. MATLAB Functions for Constructing Subsystems

All of the functions listed in Section 3.2.2 which are used to construct modules also apply to subsystems. Additional commands for constructing subsystems are:

connect.m – creates a wiring connection between two pins in a subsystem.

get_module.m – extracts a module from a subsystem and returns the @awe_module object.

These commands are documented in the *Audio Weaver User's Guide*.

3.3.3. Internal Subsystem Functions

Each subsystem has a set of subfunctions that mirror the module functions described in 3.2.3. Fortunately, for most subsystems, generic or generated versions of the functions can be used. The table below describes the operation of each of the functions in the context of generated code.

MATLAB Function Name	Target Function Name	Purpose
`*_module.m`	`*_Constructor()`	This MATLAB code must be manually written as described above. Modules are manually added, configured, and connected. The C code version of this function is automatically generated. The function calls the base module constructor function and then calls the _Constructor() function for all internal modules. The internal modules parameters are set according to their values at build time.
`.processFunc`	`*_Process()`	Audio Weaver automatically generates the MATLAB and C versions of these functions.
`.setFunc`	`*_Set()`	In all cases, this must be manually written.
`.getFunc`	`*_Get()`	In all cases, this must be manually written.
`.bypassFunc`	`*_Bypass()`	The generic function utilized by modules can also be used here.
`.preBuildFunc`	`N/A`	Not needed. The existing pin propagation function can propagate the information through subsystems.

3.3.4. Top-Level Systems

Thus far, we have been using the terms *system* and *subsystem* interchangeably. There is a slight difference, though, that you need to be aware of. The highest level system object that is passed to the build command must be a *top-level system* created by the function:

```
SYS=target_system(CLASSNAME, DESCRIPTION, RT)
```

The top-level system is still an @awe_subsystem object but it is treated differently by the build process. The main difference is how the output pin properties are handled. In a top-level system, the output pins properties are explicitly specified and not derived from pin propagation. As an added check, the pin propagation algorithm verifies that the wires attached to a top-level system's output pins match the properties of each output pin. In contrast, internal subsystems are created by calls to

```
SYS=awe_subsystem(CLASSNAME, DESCRIPTION)
```

and the type information for output pins is determined by pin propagation.

3.4. new_pin_type.m

This function returns a data structure representing a pin. The internal structure of a pin can be seen by examining the pin data structure. At the MATLAB command prompt type:

```
new_pin_type
```

Be sure to leave off the trailing semicolon – this causes MATLAB to display the result of the function call. We see:

```
ans =

          numChannels: 1
            blockSize: 32
           sampleRate: 48000
             dataType: 'float'
            isComplex: 0
     numChannelsRange: []
       blockSizeRange: []
      sampleRateRange: []
        dataTypeRange: {'float'}
       isComplexRange: 0
```

The first 5 fields of the data structure specify the *current* settings of the pin; the last 5 fields represent the *allowable ranges* of the settings. The range information is encoded using the following convention:

[] – the empty matrix indicates that there are no constraints placed on the range.

[M] – a single value indicates that the variable can only take on one value.

[M N] – a 1x2 row vector indicates that the variable has to be in the range M <=x <= N.

[M N step] – a 1x3 row vector indicates that the variable has to be in range M<=x<=N and that it also has to increment by step. In MATLAB notation, the set of allowable values is [M:step:N].

By default, the new_pin_type.m function returns a pin with no constraints on the sampleRate, blockSize, and numChannels. The dataType is floating-point and the data real.

Additional flexibility is built into the range information. Instead of just a row vector, the range can have a matrix of values. Each row is interpreted as a separate allowable range. For example, suppose that a module can only operate at the sample rates 32 kHz, 44.1 kHz, and 48 kHz. To enforce this, set the sampleRateRange to [32000; 44100; 48000]. Note the semicolons which place each sample rate constraint on a new row.

Audio Weaver also interprets NaN's in the matrix as if they were blank. For example, suppose a module can operate at exactly 32 kHz or in the range 44.1 to 48 kHz. To encode this, set sampleRateRange=[32000 NaN; 44100 48000].

The new_pin_type.m function accepts a number of optional arguments:

```
new_pin_type(NUMCHANNELS, BLOCKSIZE, SAMPLERATE, DATATYPE, ISCOMPLEX);
```

These optional arguments allow you to properties of the pin. For example, the call

```
new_pin_type(2, 32, 48000)
```

returns the pin

```
ans =

          numChannels: 2
            blockSize: 32
           sampleRate: 48000
             dataType: 'float'
            isComplex: 0
     numChannelsRange: 2
       blockSizeRange: 32
      sampleRateRange: 48000
        dataTypeRange: {'float'}
       isComplexRange: 0
```

This corresponds to exactly 2 channels with 32 samples each at 48 kHz. Note, that the pin type is represented using a standard MATLAB structure. You can always change the type information after new_pin_type.m is called. For example,

```
PT=new_pin_type;
PT.sampleRateRange=[32000; 44100; 48000];
```

The current values and ranges of values are both provided in Audio Weaver for a number of

reasons. First, the range information allows an algorithm to represent and validate the information in a consistent manner. Second, the pin information is available to the module at design time, allocation time, and at run-time. For example, the sample rate can be used to compute filter coefficients given a cutoff frequency in Hz. Third, most modules in Audio Weaver are designed to operate on an arbitrary number of channels. The module's run-time function interrogates its pins to determine the number of channels and block size, and processes the appropriate number of samples.

Consider the look ahead limiter subsystem introduced in Section 5.3.2. It can be connected to mono, stereo, or 5.1 channel signals and all of the wiring and buffer allocation will be automatically handled. *This generality allows you to design algorithms which operate on an arbitrary number of channels with little added complexity.*

Usage of new_pin_type.m is slightly more complicated than described above. In fact, the 5 arguments passed to the function actually specify the *ranges* of the pin properties and the current values are taken as the first item in each range. For example, consider a module that can operate on even block sizes in the range from 32 to 64. This is specified as:

```
new_pin_type([], [32 64 2])

ans =

         numChannels: 1
           blockSize: 32
          sampleRate: 48000
            dataType: 'float'
           isComplex: 0
    numChannelsRange: []
      blockSizeRange: [32 64 2]
     sampleRateRange: []
       dataTypeRange: {'float'}
      isComplexRange: 0
```

Note that the first argument, the number of channels, is empty. An empty matrix places no constraints on the item. Notice also that the current blockSize equals the first value, 32, in the range of allowable block sizes. Additional examples highlight other ways to use this function.

You can also specify that a pin can hold either floating-point or fract32 data. Pass in a cell array of strings as the 4th argument to new_pin_type:

```
>> P=new_pin_type([], [], [], {'float', 'fract32'})

P =

         numChannels: 1
           blockSize: 32
          sampleRate: 48000
            dataType: 'float'
           isComplex: 0
    numChannelsRange: []
      blockSizeRange: []
```

```
       sampleRateRange: []
         dataTypeRange: {'float'  'fract32'}
        isComplexRange: 0
```

Some modules do nothing more than move data around. Examples include the interleave_module.m and the delay_module.m. These modules do not care about the data type, only that it is 32-bits wide. The new_pin_type.m function uses the shorthand '*32' to represent all 32-bit data type. This currently includes 'float', 'fract32', and 'int':

```
>> PT=new_pin_type([], [], [], '*32')

PT =

           numChannels: 1
             blockSize: 32
            sampleRate: 48000
              dataType: 'float'
             isComplex: 0
      numChannelsRange: []
        blockSizeRange: []
       sampleRateRange: []
         dataTypeRange: {'float'  'int'  'fract32'}
        isComplexRange: 0
```

3.5. add_pin.m

Pins are added to a module by the add_pin.m function. Each pin has an associated Pin Type as described in Section 3.4. After creating the Pin Type, call the function

```
add_pin(M, USAGE, NAME, DESCRIPTION, TYPE)
```

for each input, output, or scratch pin you want to add. The arguments are as follows:

`M` - @awe_module object.

`USAGE` – string specifying whether the pin is an ‘input’, ‘output’, or 'scratch'.

`NAME` – short name which is used as a label for the pin.

`DESCRIPTION` – description of the purpose or function of the pin.

`TYPE` – Pin Type structure.

M.inputPin, M.outputPin, and M.scratchPin are cell arrays that describe the pins. Each call to add_pin.m adds an entry to one of these arrays, depending upon whether it is an input, output, or scratch pin. For example, to determine the number of input pins that a module has, use

```
length(M.inputPin)
```

or to print out all of the names of the output pins

```
for i=1:length(M.outputPin)
      fprintf(1,'%s\n', M.outputPin{i}.name);
end
```

Some processing functions require temporary memory storage. This memory is only needed while processing is active and does not need to be persisted between calls. (On the other hand, memory that needs to be persisted by a module between calls to the processing function appears in the instance structure and has usage "state".) Allocating temporary memory and sharing it between modules is accomplished by *scratch pins*. Scratch pins are added to a module via add_pin.m with the USAGE argument set to 'scratch'. *Scratch pins are typically not used by audio modules, but are often required by subsystems.* For subsystems, the routing algorithm automatically determines scratch pins at build time.

In some cases, you want to add a pin to a subsystem that is of the same type as an internal module. For example, the Hilbert subsystem uses the same pin type as the Biquad filter. This can be achieved programmatically:

```
add_module(SYS, biquad_module('bq11'));
pinType=SYS.bq11.inputPin{1}.type;

add_pin(SYS, 'input', 'in', 'Audio Input', pinType);
add_pin(SYS, 'output', 'out', 'Audio output', pinType);
```

3.6. add_variable.m

```
M=add_variable(M, VAR)
```

Adds a single scalar variable to an @awe_module or @awe_subsystem object and returns the updated object. Arguments:

M - @awe_module or @awe_subsystem object to which to add the variable

VAR - @awe_variable object returned by awe_variable.m

Alternatively, you can construct the variable on the fly as:

```
M=add_variable(M, ARGS...)
```

where ARGS are arguments which get passed to @awe_variable. In this case, the arguments are ordered as:

```
M=add_variable(M, NAME, TYPE, DATA, USAGE, DESCRIPTION, ISHIDDEN, ...
    ISCOMPLEX)
```

When called with no output arguments, as in

```
add_variable(M, VAR);
```

the input module M is updated in the calling environment.

After adding a variable to an audio module, it is a good idea to also specify its range and units. The range field is used when drawing inspectors (sliders and knobs, in particular) and also for validating variable assignments. The units string reminds the user of what units the variable represents. You set these as:

```
M.variableName.range=[min max];
M.variableName.units='msec';
```

Note that after a variable is added to a module, it appears as a *field* within the module's structure and the name of the field equals the name of the variable. Attributes of an individual variable are referenced using the "." structure notation. Refer to Section 2.2 to see the correspondence between variables in the MATLAB object and variables in the C type definition.

3.7. add_array.m

```
M=add_array(M, VAR)
```

Adds an array to an @awe_module or @awe_subsystem object and returns the updated object. Arguments:

M - @awe_module or @awe_subsystem object to which to add the array

VAR - @awe_variable object.

Alternatively, you can construct the array on the fly as:

```
M=add_array(M, ARGS...)
```

where ARGS are arguments which get passed to @awe_variable. In this case, the arguments are ordered as:

```
M=add_array(M, NAME, TYPE, DATA, USAGE, DESCRIPTION, ISHIDDEN, ...
     ISCOMPLEX)
```

The size of the array is specified when it is first added. You can also subsequently change the array size in the module's prebuild function. When doing so, you need to change the .size field of the variable and then the data itself. For example, the state variables used by the Biquad module are stored in a matrix of size 2 x numChannels. However, numChannels is not known until the module is built and therefore code within the Biquad prebuild function is used to update the array size:

```
M.state.size=[2 M.inputPin{1}.type.numChannels];
M.state=zeros(2, M.inputPin{1}.type.numChannels);
```

Audio Weaver supports 1 and 2 dimensional arrays. The 2 dimensional array representation is maintained in MATLAB. On the target, however, a 2 dimensional array is flattened into a 1 dimensional array on a column by column basis. Both scalar and array variables are represented as @awe_variable objects.

Array variables appear as pointers in the module instance structure. For example, an FIR filter has separate arrays for coefficients and state variables:

```
add_variable(M, 'numTaps', 'int', L, 'const', 'Length of the filter');
M.numTaps.range=[1 5000 1];
M.numTaps.units='samples';

add_array(M, 'coeffs', 'float', [1; zeros(L-1,1)], 'parameter', 'Coefficient
       array');
M.coeffs.arrayHeap='AE_HEAP_FAST2SLOW';
M.coeffs.arraySizeConstructor='S->numTaps * sizeof(float)';

add_variable(M, 'stateIndex', 'int', 0, 'state', 'Index of the oldest state
       variable in the array of state variables');
M.stateIndex.isHidden=1;

% Set to default value here.  This is later updated by the pin function
add_array(M, 'state', 'float', zeros(L, 1), 'state', 'State variable array');
M.state.arrayHeap='AE_HEAP_FASTB2SLOW';
M.state.arraySizeConstructor='ClassWire_GetChannelCount(pWires[0]) * S->numTaps
       * sizeof(float)';
M.state.isHidden=1;
```

The C type definition for this FIR filter is found in the file ModFIR.h:

```
typedef struct _awe_modFIRInstance
{
  ModuleInstanceDescriptor instance;
       int          numTaps;            // Length of the filter
       int          stateIndex;         // Index of the oldest state variable
       float*       coeffs;             // Coefficient array
       float*       state;              // State variable array
} awe_modFIRInstance;
```

Using indirect arrays enables arrays to have variable length and also to be placed in distinct memory banks. This is last time is useful, for example, on the SHARC processor to enable simultaneous memory reads.

3.8. add_module.m

Modules are added to subsystems one at a time using the function

```
add_module(SYS, MOD)
```

The first argument is the subsystem while the second argument is the module (or subsystem) to add. Once a module is added to a subsystem, it appears as a field within the object SYS. The name of the field is determined by the module's name. Modules can be added to subsystems in

any order. The run-time execution order is determined by the routing algorithm.

As modules are added to the subsystem, they are appended to the .module field. You can use standard MATLAB programming syntax to access this information. For example, to determine the number of modules in a subsystem:

```
count=length(SYS.module);
```

Or, to print out the names of all of the modules in a subsystem:

```
for i=1:length(SYS.module)
      fprintf(1, '%s\n', SYS.module{i}.name);
end
```

3.9. connect.m

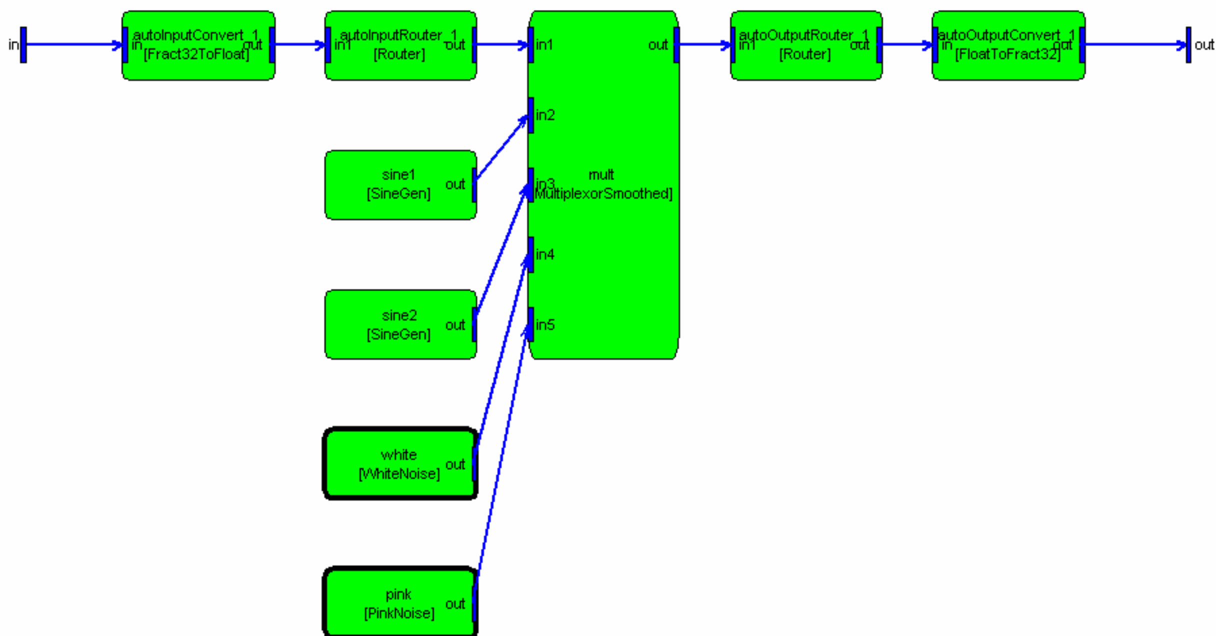
Creates a connection between two pins in a subsystem. The general form is:

```
connect(SYS, SRCPIN, DSTPIN);
```

where SRCPIN and DSTPIN specify the source and destination pins, respectively. Pins are specified as strings to the connect.m command using the syntax:

```
moduleName.pinName
```

Consider the system shown below and contained within multiplexor_example.m.



To connect the output of the sine generator1 to the second input of the multiplexor, use the

command:

```
connect(SYS, 'sine1.out', 'mult.in2');
```

The module names "sine1" and "mult" are obvious because they were specified when the modules were created. The pins names may not be obvious since they appear within the module's constructor function. To determine the names of a module's pins, you can either utilize the detailed help function awe_help (recommended)

```
awe_help multiplexor_smoothed_module
```

Pins

Input Pins

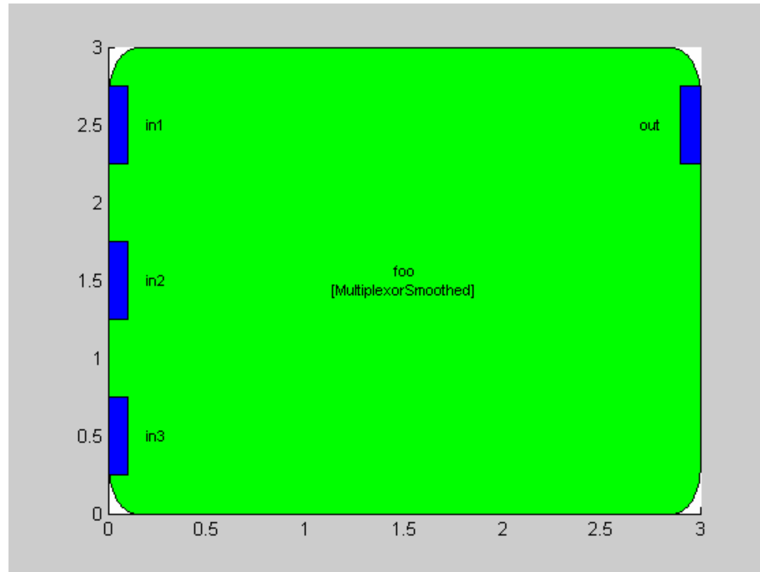
```
              Name: in1
       Description: Input signal
         Data type: float
     Channel range: Unrestricted
  Block size range: Unrestricted
 Sample rate range: Unrestricted
   Complex support: Real

              Name: in2
       Description: Input signal
         Data type: float
     Channel range: Unrestricted
  Block size range: Unrestricted
 Sample rate range: Unrestricted
   Complex support: Real

              Name: in3
       Description: Input signal
         Data type: float
     Channel range: Unrestricted
  Block size range: Unrestricted
 Sample rate range: Unrestricted
   Complex support: Real
```

Output Pins

```
              Name: out
       Description: Output signal
         Data type: float
     Channel range: Unrestricted
  Block size range: Unrestricted
 Sample rate range: Unrestricted
   Complex support: Real
```

or have MATLAB draw the module

```
M=multiplexor_smoothed_module('foo');
draw(M)
```



Several short cuts exists to simplify use of the connect.m command.

If the module has only a single input or output pin, then you need only specify the module name; the pin name is assumed. Since the sine wave generator module in the multiplexor example has only a single output pin, the example above reduces to:

```
connect(SYS, 'sine1', 'mult.in2');
```

Inputs and outputs to the subsystem are specified by the empty string. Thus,

```
connect(SYS, '', 'mult.in1');
```

connects the input of the system to the first input of the multiplexor. Similarly,

```
connect(SYS, 'mult', '');
```

connects the output of the multiplexor to the output of the system.

By default, the connect command performs rudimentary error checking. The function verifies that the named modules and pins exist within the subsystem. At build time, however, exhaustive checking of all connections is done. Audio Weaver verifies that all connections are made to compatible input and output pins (using the range information within the pin type). You can enable this exhaustive checking when a connection is made by supplying a 4th argument:

```
connect(SYS, 'mult', '', 1)
```

This is useful when debugging wiring failures that are revealed at build time.

Output pins are permitted to fan out to an arbitrary number of input pins. Input pins, however, can only have a single connection.

Audio Weaver is set up to handle several special cases of connections at build time. First, if an input pin has no connections, Audio Weaver inserts a source module and connects it to the unconnected input. Either a source_module.m (float) or source_fract32_module.m is added based on the first input to the system[5]. If a module has an output pin without a connection, Audio Weaver inserts a sink module based on the data type of the unconnected pin[6]

```
sink_module.m             Floating-point data
sink_fract32_module.m     Fixed-point data
sink_int_module.m         Integer data
```

If the subsystem has a direct connection from an input pin to an output pin, then a copier_module is inserted. If a module output fans out to N outputs of the subsystem, then N-1 copier modules are inserted[7].

3.10. get_variable.m and set_variable.m

Extract and assign individual @awe_variable objects within an audio module. This is needed because of the object oriented nature of the Audio Weaver interface. When accessing a variable "var" within a module "M", as in:

```
M.var
```

Audio Weaver returns the *value* of the variable, not the variable object itself. If you wish to gain access to the actual variable object, use the call:

```
V=get_variable(M, NAME)
```

where

M – the @awe_module object.

NAME – a string specifying the name of the variable.

The function returns an @awe_variable object. For example, the following code gets the underlying @awe_variable object for the "gain" variable of a smoothed scaler module:

---

[5] Audio Weaver is guessing at the type of pin that needs to be connected.

[6] This is always correct since the pin type is inherited from the module's output pin.

[7] This is required since each output of the subsystem is stored in a separate buffer.

```
M=scaler_smoothed_module('scaler');
V=get_variable(M, 'gain');
```

Similary, the set_variable.m command can be used to replace an existing variable with a given @awe_variable object. "Replace" is used because the variable must already exist within the audio module. The syntax for this command is:

```
M=set_variable(M, NAME, VAR)
```

where

M – the @awe_module object.

NAME – a string specifying the name of the variable.

VAR - @awe_variable object.

The get_variable.m and set_variable.m functions are rarely used in practice. You can always access the internal fields of a @awe_variable object even when it is part of an audio module. For example, to access the "range" field of the variable "gain", use:

```
range=M.gain.range;
```

get_variable.m and set_variable.m also apply to @awe_subsystem objects.

4. C Run-Time Environment

This section describes the portion of the C run-time environment used by audio modules. This includes the module processing and constructor functions, memory allocation, and how the modules interface to the Server running on the target.

Each audio module has an associated header file Mod*ClassName*.h containing a class definition, and a source file Mod*ClassName*.c containing the associated class structure and functions. The header and source files are generated by MATLAB, but it is important to understand their contents and how all of the pieces fit together.

4.1. Data types and Structure Definitions

Let's begin by looking at the class definition for the smoothly varying scaler found in ModScalerSmoothedExample.h:

```
typedef struct _ModuleScalerSmoothedExample
{
      ModuleInstanceDescriptor instance;  // Common module header
      float       gain;            // Target gain
      float       smoothingTime;   // Smoothing time constant
      float       currentGain;     // Instantaneous gain applied
      float       smoothingCoeff;  // Smoothing coefficient
} ModuleScalerSmoothedExampleClass;
```

The first field "instance" is the common header used by all audio modules and the type ModuleInstanceDescriptor is defined in

```
<AWE>\Include\Framework.h.
```

Framework.h contains the primary data types and definitions used by Audio Weaver. Most of the definitions are used internally by the Server, but a few apply to audio modules.

4.1.1. Module Instance Structure

Following the instance header are all of the module specific variables. Only 8 types of module variables are currently supported:

int – 32-bit signed integer

uint – 32-bit unsigned integer

float – 32-bit floating-point value

fract32 – 32-bit fractional in 1.31 format

int * - pointer to an array of signed integers

uint * - pointer to an array of unsigned integers

float * - pointer to an array of floating-point values

fract32 * - pointer to an array of fractional values

You'll note that only 32-bit data types are currently supported. This is for simplicity, but may be expanded in the future.

All scalar module variables must precede arrays in the instance structure. This is due to the manner in which audio modules are allocated and initialized by the Server. The MATLAB module generation scripts automatically reorder variables (using the function awe_reorder_variables.m). Detailed description of awe_reorder_variables.m is found in Section 5.1.9.

The module instance descriptor is also defined within Framework.h:

```
typedef struct _ModuleInstanceDescriptor
{
      ModInstanceDescriptor instanceDescriptor;
      struct _LayoutInstance *pOwner;
      UINT packedFlags;
      float profile_time;
      WireInstance **pWires;
      void (*pProcessFunc)(void *pInstance);
}
ModuleInstanceDescriptor;
```

The module instance header is 8 words in length and contains the fields:

instanceDescriptor – 3 word data structure that allows the Server to keep track of the module once it has been allocated on the target.

pOwner – points to the overall layout, or collection of audio modules.

packedFlag – a packed integer containing the number of input, output, and scratch pins, as well as the current modules status.

profile_time – CPU profiling information. This is measured on a block-by-block basis and smoothed by a first order filter. Time is measured in units of clock ticks, where the resolution is platform dependent. Some platforms, such as the SHARC and Blackfin, are cycle accurate.

pWires – pointer to an array of wires used by the module. Wires are ordered as inputs, outputs, and scratch wires.

pProcessFunc – pointer to the module's current run-time function. This points to the processing function if the module is ACTIVE.

Access to the module instance fields is via macros defined in Framework.h. You should use these macros since it enables us to change the underlying instance structure without breaking existing modules. Let S be a pointer to an audio module. The following macros are provided:

ClassModule_GetWires(S) – returns a pointer to the array of wires associated with the module. The wires are ordered as input wires, output wires, and then scratch wires.

ClassModule_GetNInWires(S) – returns the number of input wires.

ClassModule_GetNOutWires(S) – returns the number of output wires

ClassModule_GetNScratchWires(S) – returns the number of scratch wires

ClassModule_GetModuleState(S) – returns the module's run-time state. ACTIVE=0, BYPASSED=1, MUTED=2, and INACTIVE=3.

4.1.2. Pins and Wires

A wire in Audio Weaver contains more than just audio samples. A wire is defined as:

```
typedef struct _WireInstance
{
      /** The basic instance data. */
      InstanceDescriptor instanceDescriptor;

      /** The pin descriptor describing this wire. */
      PinInstanceDescriptor *pPinDescriptor;

      /** The wire buffer. */
      Sample *buffer;
}
WireInstance;
```

The data type WireInstance describes a wire. To get a pointer to the data portion of the first wire, use the macro:

```
Sample *buffer = (pWires[0]->buffer);
```

*buffer points to the buffer of audio samples. The data type Sample is a union of integer and floating point values:

```
typedef union _Sample
{
      int iVal;
      unsigned int uiVal;
      float fVal;
}
Sample;
```

Since Sample is a union, you must cast it to one of the possible data types:

```
(float *) W->buffer;
(int *) W->buffer;
(fract32 *) W->buffer;
(unsigned int *) W->buffer;
```

The fract32 data type is actually defined as int32. Providing the fract32 data type is for convenience allowing the programmer to distinguish between integer and fractional values.

A wire can have an arbitrary number of interleaved channels and an arbitrary block size (number of samples per channel). A wire with N channels and a block size B has a total of NB audio samples. The samples are ordered as:

```
Channel 0, Sample 0
Channel 1, Sample 0
Channel 2, Sample 0
...
Channel N-1, Sample 0
Channel 0, Sample 1
Channel 1, Sample 1
Channel 2, Sample 1,
...
...
Channel N-1, Sample B-1
```

The items in the wire's pin descriptor are accessed via macros:

ClassWire_GetChannelCount(W) – returns the number of channels in the wire.

ClassWire_GetBlockSize(W) – returns the block size of the wire.

ClassWire_GetNumSamples(W) – returns the total number of samples in the wire. This equals the number of channels times the block size.

ClassWire_GetSampleRate(W) – returns the sample rate of the data in the wire. This is used by design functions, such as filter coefficient generation, that depend upon the sample rate of the data. The sample rate is in Hz.

A pin descriptor structure can be shared by multiple wires as long as the number of channels, block size, and sample rate are the same. Allocating and assigning pin descriptors is handled by MATLAB when a system is being built.

You'll note that the pin descriptor does not specify the type of data (float, int, or fract32) contained in the wire. This allows a wire containing floating-point data to be reused for integer data as long as the number of channels, block size, and sample rate are identical.

4.1.3. Examples

The absolute value module has a single input and output pin, and can handle an arbitrary number of channels, block size, and sample rate. S is a pointer to the module instance. The processing

code is:

```
void ModuleAbs_Process(void *pInstance)
{
  ModuleAbsClass *S = (ModuleAbsClass *)pInstance;
  WireInstance **pWires = ClassModule_GetWires(S);

  Vec_Abs((float *)pWires[0]->buffer, 1,
        (float *)pWires[1]->buffer, 1,
        ClassWire_GetNumSamples(pWires[0]));
}
```

where

(float *) pWires[0]->buffer points to the buffer of input samples.

(float *) pWires[1]->buffer points to the buffer of output samples

ClassWire_GetNumSampels(pWires[0]) is the total number of samples in the input wire.

Vec_Abs() is an optimized vector function for computing the absolute value.

A second example is the processing function found ModScalerN.c. This module has a single input pin with multiple channels. Each channel is scaled by a separate gain. The processing function is:

```
void ModuleScalerN_Process(void *pInstance)
{
  ModuleScalerNClass *S = (ModuleScalerNClass *)pInstance;
  WireInstance **pWires = ClassModule_GetWires(S);

  UINT blockSize = ClassWire_GetBlockSize(pWires[0]);
  UINT numChannels = ClassWire_GetChannelCount(pWires[0]);
  UINT channel;
  float *dst;
  float *src;

  src = (float *) pWires[0]->buffer;
  dst = (float *) pWires[1]->buffer;

  for(channel=0;channel<numChannels;channel++)
    {
      Vec_Scale(src+channel, numChannels,
                dst+channel, numChannels,
                S->gain[channel], blockSize);
    }
}
```

This module has a multichannel input and output pin. At the start of the processing function, the number of channels and block size are determined as well as pointers to the input and output buffers. The module then loops over each channel scaling it by the appropriate scale factor from the gain[] array. The arguments to the Vec_Scale function are:

src+channel – pointer to the buffer of input data.

numChannels – stride for the input data.

dst+channel – pointer to the buffer of the output data

numChannels – stride for the output data

S->gain[channel] – scale factor for a particular channel.

blockSize – number of samples to scale.

You can clearly see that the data is interleaved based on the offset to the start of the src and dst data as well as the strides.

Both of the module examples use a set of "vector" functions to perform the actual computation. It is possible to perform the computation within the module's processing function – many modules do this. For some modules, however, we have chosen to use vector functions since smaller functions may be more easily optimized in assembly and a single function may be reused by multiple modules. For example, Vec_Scale() is used by the scaler_module.m, scalern_module.m, and scaler_db_module.m. Thus, by optimizing a single function vector for a target processor we obtain 3 optimized modules.

4.2. Class Object and Constructor Functions

The module constructor function is called in response to a create_module command sent from the Server to the target. A module's constructor function is called once per module instance and is responsible for allocating memory for the module and copying arguments from the create_module command into the instance structure.

The operation of the constructor function hinges on information within the module's class structure. All audio modules of the same class share a single module class structure. Thus, if a target contains 100 different classes of audio modules, there will be 100 class structures defined. The class structure is defined in Framework.h as:

```
/** Class descriptor for audio modules. */
typedef struct _ModClassModule
{
	/** The basic class data. */
	ModClassDescriptor modClassDescriptor;

	/** The number of construction module parameters. */
	UINT nPackedParameters;

	/** Pump function for the module. */
	void (*pProcessFunc)(void *pInstance);

	/** Bypass function for the module. */
	void (*pBypassFunc)(void *pInstance);
```

```
        /** Set function. */
        UINT (*pSet)(void *pInstance, UINT mask);

        /** Get function. */
        UINT (*pGet)(void *pInstance, UINT mask);
}
ModClassModule;
```

The first item of type ModClassDescriptor is defined as:

```
typedef struct _ModClassDescriptor
{
        /** Unique ID of the class - set at compile time. */
        UINT classID;

        struct _ModInstanceDescriptor *(*Constructor)(
                        int * FW_RESTRICT retVal,
                        UINT nIO,
                        WireInstance ** FW_RESTRICT pWires,
                        size_t argCount,
                        const Sample * FW_RESTRICT args);
}
ModClassDescriptor;
```

Let's look at each of these fields more closely.

classID – is a unique integer that identifies the module on the target.

When the target is asked to create a scaler_module.m, it is asked to create a module with a specific integer class ID. In order to ensure that all classID's are unique, they are stored as offsets in the file classids.csv. When asked to construct a module, the target searches the table of module class pointers looking for the specified classID. If there is a match, then the Constructor function, also contained within ModClassDescriptor is called.

Constructor() - pointer to the module's constructor function.

The constructor function takes a specific set of arguments:

```
Constructor(int * FW_RESTRICT retVal,
          UINT nIO,
          WireInstance ** FW_RESTRICT pWires,
          size_t argCount,
          const Sample * FW_RESTRICT args);
```

Now returning to the other fields in ModClassModule. The modClassDescriptor is followed by:

```
UINT nPackedParameters;
```

This integer specifies the number of constructor arguments required by the module. Two separate values are packed into this 32-bit integer and these are accessed via separate macros:

ClassMod_GetNPublicArgs(packed) – returns the number of public arguments that the module requires. The number of public arguments equals the number of scalar variables that must be sent from the Server to the Constructor() function.

ClassMod_GetNPrivateArgs(packed) – returns the number of private arguments required by the module. The private arguments are not sent by the Server but must be separately managed by the Constructor function. Private arguments typically hold pointers to array variables.

The total size of a module's instance structure equals sizeof(ModClassModule) + # of public words + # of private words.

The remaining items in the module class structure are pointers to module functions.

pProcessFunc – pointer to the module's processing function. Called when the module is in ACTIVE mode.

pBypassFunc – pointer to the module's bypass function. Called when the module is in BYPASSED mode.

pSet – pointer to the module's set function. This function is called after the Server updates one of the module's variables. This function is used to implement custom design functions on the target.

pGet – pointer to the module's get function. This function is called *prior* to reading one of the module's instance variables by the Server.

A module must provide processing and bypass functions; all other functions are optional.

Let's examine the class structure for the biquad_module.m. This module implements a 5 multiply second order IIR filter. The module's class structure defined in ModBiquad.c is:

```
const ModClassModule ClassModule_Biquad =
    {
#ifdef AWE_INCLUDE_CONSTRUCTOR_FUNCTION
      { CLASSID_ModuleBiquad, ModuleBiquad_Constructor },
#else
      { CLASSID_ModuleBiquad, NULL },
#endif
      ClassModule_PackArgCounts(5, 1),    // (Public words, private
words)
      ModuleBiquad_Process,                     // Processing function
      IOMatchUpModule_Bypass,                   // Bypass function
      0,                                  // Set function
      0,                                  // Get function
    };
```

The CLASSID_ModuleBiquad equals a base value plus an offset:

```
CLASS_ID_MODBASE + 13
```

The offset is defined in the file classids.csv:

```
Biquad,13
```

The module has a custom constructor function, ModuleBiquadConstructor(), and a total of 5 public instance variables and 1 private instance variable. Looking at the module's instance structure, we see:

```
typedef struct _ModuleBiquad
{
  ModuleInstanceDescriptor instance;
  float           b0;
  float           b1;
  float           b2;
  float           a1;
  float           a2;
  float*          state;
} ModuleBiquadClass;
```

The 5 public variables are the filter coefficients: b0, b1, b2, a1, and a2. The private variable is the pointer to the array of state variables. The Biquad module operates on multichannel signals and allocates two state variables per channel. The Biquad module utilizes the processing function ModuleBiquad_Process(), the bypass function IOMatchUpModule_Bypass(), and does not define a Get or Set function.

The module constructor function first allocates memory for the instance structure, then sets public variables using supplied values, and finally, allocates memory for any arrays. Constructor functions all have a similar structure exemplified by the constructor for the Biquad module. Starting with the function arguments:

```
ModInstanceDescriptor *ModuleBiquad_Constructor(int * FW_RESTRICT
retVal,
                                    UINT nIO,
                                    WireInstance ** FW_RESTRICT pWires,
                                    size_t argCount,
                                    const Sample * FW_RESTRICT args)
{
  ModuleBiquadClass *S = (ModuleBiquadClass *)
    BaseClassModule_Constructor(&ClassModule_Biquad, retVal, nIO,
                                pWires, argCount, args);

  if (S == NULL)
    {
      return 0;
    }

  if ((S->state = (float *)
awe_fwMalloc(ClassWire_GetChannelCount(pWires[0]) * 2 * sizeof(float),
                  AE_HEAP_FAST2SLOW, retVal)) == 0)
    {
```

```
            // Error code is in *retVal
            return 0;
          }

        *retVal = E_SUCCESS;
        return ((ModInstanceDescriptor *) S);
      }
```

The function arguments are:

retVal – pointer to the function's error return value. If successful, the function sets *retVal = E_SUCCESS.

nIO – a packed integer specifying the number of input, output, and scratch wires used by the module.

pWires – pointer to the array of wires used by the module.

argCount – total number of arguments supplied to the constructor. Typically, the constructor arguments are computed by MATLAB and supplied to the create_module Server function.

args – pointer to the list of function arguments. The function arguments can be either integer or floating-point values and are specified as a union. The constructor function must know whether each argument is an integer or floating-point value. arg[0].iVal is the first argument as an integer; arg[0].fVal is the first argument as a floating-point value; arg[0].uiVal is the first argument as an unsigned integer.

The first responsibility of the Constructor function is to call BaseClassModule_Constructor(). This function does several things:

Allocates memory for the module's instance structure.

Verifies that argCount equals the number of public instance variables specified in the module's class structure.

Copies the initial values of the public variables from args[] to the variables at the start of the instance structure. (This method of initializing module variables requires that the public module variables be located contiguously at the start of the instance structure.)

Sets the pWires and packedFlags fields of the ModuleInstanceDescriptor.

Configures the module for ACTIVE mode.

The Constructor function next allocates memory for the state array;

```
if ((S->state = (float *)
```

```
awe_fwMalloc(ClassWire_GetChannelCount(pWires[0]) * 2 * sizeof(float),
AE_HEAP_FAST2SLOW, retVal)) == 0)
     {
       // Error code is in *retVal
       return 0;
     }
```

Note that the size of the array is computed based on the number of channels in the first input wire. The function awe_fwMalloc() allocates memory from one of Audio Weaver's memory heaps. The heap is specified by the second argument: AE_HEAP_FAST2SLOW and user can find all the macros defined in Framework.h.

Lastly, the Constructor function sets the return value and returns a pointer to the allocated and initialized object:

```
*retVal = E_SUCCESS;
return ((ModInstanceDescriptor *) S);
```

The instance pointer is cast to a common ModInstanceDescriptor return type.

The Biquad module was used as an example to illustrate the constructor function because it requires an array to be allocated. If the Constructor function pointer in the module's class structure is set to NULL, then the generic module constructor is called. The generic constructor simply calls BaseClassModule_Constructor() and is sufficient for modules that do not require additional memory allocation outside of the instance structure or other custom initialization. The generic constructor is sufficient for a large number of modules and reduces the code size of the target executable.

4.3. Memory Allocation using awe_fwMalloc()

Dynamic memory allocation is accomplished in Audio Weaver by the function awe_fwMalloc(). To avoid memory fragmentation, memory can only be allocated but never individually freed. You can only free all memory completely using the destroy command and then start again.

The memory allocation function is declared as:

```
void *awe_fwMalloc(size_t size, UINT heapIndex, int *retVal);
```

All memory allocation in Audio Weaver occurs in units of 32-bit words. The first argument to awe_fwMalloc() specifies the number of 32-bit words to allocate.

The second argument, heapIndex, specifies which memory heap should be used for allocation. Audio Weaver provides 3 heaps and the heaps match the memory architecture of the SHARC processor. There are two separate internal heaps – traditionally referred to as "DM" and "PM" on the SHARC – which are placed in different memory blocks; and a large heap in external memory. Having separate internal memory heaps is required to obtain optimal performance from the SHARC. For example, an FIR filter implemented on the SHARC processor requires that the

state and coefficients be in separate memory blocks. On the Blackfin with cache enabled, the heaps specifiers are irrelevant.

Audio Weaver refers to the heaps as "FAST", "FASTB", and "SLOW". The heapIndex argument is set using #define's in Framework.h. In the simplest case, you specify one of the following:

`AWE_HEAP_FAST` – fast internal memory. Defined as 1.

`AWE_HEAP_FASTB` – a second internal memory heap located in a different block of memory than AWE_HEAP_FAST. Defined as 2.

`AWE_HEAP_SLOW` – slower memory; usually external. Defined as 3.

The heapIndex argument can encode several different heaps one nibble (4-bits) at a time. Set the low 4 bits of heapIndex to the heap you want to allocate memory from first; then set bits 4 to 7 to the next heap; and so forth. For example, if you want to allocate memory from AWE_HEAP_FAST and then overflow into AWE_HEAP_SLOW, set the heapIndex argument to:

```
AWE_HEAP_FAST | (AWE_HEAP_SLOW << 4)
```

Several variations are already defined for you.

```
#define AWE_HEAP_FAST2SLOW (AWE_HEAP_FAST | (AWE_HEAP_FASTB << 4) |
(AWE_HEAP_SLOW << 8))
```

Allocate in the FAST heap first. If this fails, then allocate in FASTB. Finally, if this fails, then allocate SLOW external heap.

```
#define AWE_HEAP_FASTB2SLOW (AWE_HEAP_FASTB | (AWE_HEAP_SLOW << 4))
```

Allocate in FASTB. If this fails, then allocate from the SLOW heap.

Most modules supplied with Audio Weaver take advantage of this overflow behavior to smoothly transition from internal to external memory once the heaps have filled up.

4.4. Module Function Details

Each audio module has a set of 5 functions associated with it. This section describes the arguments to each of the functions.

4.4.1. Constructor Function

This function is usually generated by MATLAB but at times it needs to be hand written – especially if the module does more than just allocate memory. Audio Weaver names the constructor function using the convention

```
awe_modCLASSNAMEConstructor()
```

where CLASSNAME is the module class. The function allocates memory for the module from the heaps and then returns a pointer to the newly created object. The constructor function is declared as:

```
ModInstanceDescriptor *awe_modCLASSNAMEConstructor(
      int * FW_RESTRICT retVal,
      UINT nIO,
      WireInstance ** FW_RESTRICT pWires,
      size_t argCount,
      const Sample * FW_RESTRICT args)
```

The return type ModInstanceDescriptor is a common data structure which starts off all audio modules. The function arguments are:

**retVal* – integer return value. The constructor should set this to E_SUCCESS (=0) upon successful completion. The file Errors.h has a complete list of error codes to choose from.

nIO – packed integer specifying the number of input, output, scratch, and feedback wires. Each number is packed into 8-bits as follows:

```
INPUT + (OUTPUT >> 8) + (SCRATCH >> 16) + (FEEDBACK >> 24)
```

Several macros are provided in Framework.h to extract the wire counts:

```
#define GetInputPinCount(x)   (x & 0xff)
#define GetOutputPinCount(x)  ((x >> 8) & 0xff)
#define GetScratchPinCount(x) ((x >> 16) & 0xff)
#define GetFeedbackPinCount(x)((x >> 24) & 0xff)

#define GetWireCount(x)       ( ((x >> 24) & 0xff) + ((x >> 16) & 0xff)
       + ((x >> 8) & 0xff) + (x & 0xff))
```

pWires – a pointer to an array of wire pointers. The wires are ordered as input, output, and scratch.

argCount – number of optional arguments supplied in the Sample array.

args – array of optional arguments. This array is of type Sample which is union:

```
typedef union _Sample
{
      int iVal;
      unsigned int uiVal;
      float fVal;
}
Sample;
```

Each argument can thus be a floating-point, or signed or unsigned integer value.

The module constructor function should first call

```
ModInstanceDescriptor *BaseClassModule_Constructor(
            const ModClassModule * FW_RESTRICT pClass,
            int * FW_RESTRICT retVal,
            UINT nIO,
            WireInstance ** FW_RESTRICT pWires,
            size_t argCount,
            const Sample * FW_RESTRICT args);
```

The first argument, pClass, is a pointer to the class object for the module under construction. The remaining arguments are identical to the Constructor arguments. For example, in ModPeakHold.c, we have

```
awe_modPeakHoldInstance *S = (awe_modPeakHoldInstance *)
      BaseClassModule_Constructor(&awe_modPeakHoldClass, retVal,
                                  nIO, pWires, argCount, args);
```

The BaseClassModule_Constructor allocates memory for the module instance structure and wire pointer array used by the module. The function also initializes the module instance variables by copying argCount words from the args array starting to the first instance variable in the structure. The class structure awe_modPeakHoldClass contains the entry

```
ClassModule_PackArgCounts(5, 2),    // (Public words, private words)
```

Thus an instance of a peak hold module has a total of 7 variables with the first 5 being set at construction time and the last 2 used internally (or arrays) argCount must then equal 5 and BaseClassModule_Constructor checks for this. The remaining two variables in the instance structures are pointers to arrays which are separately initialized by the constructor function.

The final step performed by BaseClassModule_Constructor is to add the newly created module instance to the linked list of objects on the target and then return a pointer to the module instance.

After calling BaseClassModule_Constructor, the module constructor performs any other initialization that is needed. This may include initializing indirect array variables, like in the ModPeakHold.c example. When done, the constructor function sets the return error value in *retVal and returns a pointer to the created module instance.

The module constructor function is typically called by the framework in response to a message received from the PC over the tuning interface or from a message stored in a file system. After a module constructor function is called, the framework checks if the module has an associated Set function. If so, the Set function is called with a mask of 0xFFFFFFFF. *Thus there is no need to manually call the Set function within the Constructor. The framework takes care of this.*

4.4.2. Processing Function

The processing function has the signature

```
void awe_modPeakHoldProcess(void *pInstance)
```

The function only gets a single argument which is a pointer to the module instance. This is first cast to the appropriate module instance data type. For example

```
awe_modPeakHoldInstance *S = (awe_modPeakHoldInstance *)pInstance;
```

Using macros defined in Framework.h, the processing function can obtain a pointer to the list of wires used by the module

```
WireInstance **pWires = ClassModule_GetWires(S);
```

The number of input wires

```
int numInWires = ClassModule_GetNInWires(S);
```

The number of output wires

```
int numOutWires = ClassModule_GetNOutWires(S);
```

The number of scratch wires

```
int numScratchWires = ClassModule_GetNScratchWires(S);
```

And the current run-time state of the module

```
int state = ClassModule_GetModuleState(S)
```

where

```
MODULE_ACTIVE = 0
MODULE_BYPASS = 1
MODULE_MUTE = 2
MODULE_INACTIVE = 3
```

Note, the processing function is only called when the state is active so checking the state is meaningless in the processing function.

The wire buffers are ordered as input, output, and scratch wires. Each wire has certain properties that you can query using macros:

```
ClassWire_GetBlockSize(W)
ClassWire_GetChannelCount(W)
```

```
ClassWire_GetSampleRate(W)
ClassWire_GetNumSamples(W)
```

where the last macro returns the total number of samples in the wire and equals blockSize x numChannels.

Each wire contains a pointer to its data samples in the →buffer field. buffer is defined as the union Sample and must be cast to a pointer of the appropriate type. For example,

```
(float *) (pWires[0]->buffer)
```

The processing function then performs the audio processing and writes the result into the output wire buffers. Local state variables in the instance structure are updated, as needed.

Note that the processing function should never overwrite the data contained in the input wires; it may be used by other modules.

The processing function does not return an error code. If you need to track error conditions, do this by adding an error code to the instance structure and then tracking it in the inspector.

4.4.3. Set Function

This function implements a module's control functionality and typically converts high-level variables to low-level variables. The PeakHold module has the function signature

```
UINT awe_modPeakHoldSet(void *pInstance, UINT mask)
```

The first argument is a pointer to the module instance structure. This needs to be cast to a pointer of the appropriate type:

```
awe_modPeakHoldInstance *S = (awe_modPeakHoldInstance *)pInstance;
```

Once you have a pointer to the module instance, you have full access to the module wires using the macros defined in Framework.h. This is useful, for example, when determining the sample rate or the block size of the module.

The second argument to the Set function is a bit mask that specifies which module variable(s) have changed. The bit mask is set by the Server when module variables are updated while tuning. The bit mask may also be set by control code running on the target. Use of the bit mask is optional and is only really necessary when the module writer needs to distinguish between heavy duty changes requiring significant amounts of computation and lightweight changes.

In the bit mask, bit N is set when the Nth module variable is set. The bit counting includes the 8 word module instance header and thus the first tunable module variable is numbered 8. The module header file contains a set of #define's indicating the bit value for each variable. For example, ModPeakHold.h contains

```
#define MASK_PeakHold_Reset 0x00000100
#define MASK_PeakHold_attackTime 0x00000200
#define MASK_PeakHold_decayTime 0x00000400
#define MASK_PeakHold_decayCoef 0x00000800
#define MASK_PeakHold_attackCoef 0x00001000
#define MASK_PeakHold_peakHold 0x00002000
#define MASK_PeakHold_peakDecay 0x00004000
```

A few more things to keep in mind regarding masks. When an array variable is set, the bit mask corresponding to the array is set as expected. However, the Set function is not told *which* portion of the array has been changed. Finally, since the mask is only 32 bits in length, it is possible to run out of unique bits if the instance structure is very long. The high bit of the mask, bit 31, has special meaning. When this bit is set, it indicates that at least one of the high order instance variables has changed. However, you don't know precisely which one.

4.4.4. Get Function

This function is rarely used in practice and translates low-level variables to higher-level variables. This function has the same signature as the Set function

```
UINT awe_modClassNameGet(void *pInstance, UINT mask)
```

When called, the function should update instance variables and can use the mask argument to determine which variables should be updated.

4.4.5. BypassFunction

This function implements a module's bypass functionality. The function signature is identical to the processing function

```
void awe_modClassNameBypass(void *pInstance)
```

There are several standard bypass functions defined in ModCommon.h and may be used by modules.

IOMatchModule_Bypass() – Works with modules that have the same number of input as output wires. It copies the data from the Nth input wire to the Nth output wire. It assumes (and does not check) that all input and output wires have the exact same number of samples.

IOAcrossModule_Bypass() – Works with modules that have a single input wire and single output wire. The wires may have a different number of channels, though. Let INCOUNT and OUTCOUNT represent the number of channels in the input and output wires, respectively. The function copies the first min(INCOUNT, OUTCOUNT) channels across one at a time from input to output. If OUTCOUNT > INCOUNT, then the remaining output channels are set to 0.

IOAllOnesModule_Bypass() – Sets all samples in the output wire to floating point 1.0. This is used by some dynamics processors which compute time-varying gains.

IOMatchUpModule_Bypass() – Suitable for modules that have an arbitrary number of input and output wires. The function starts with the first output wire and looks for the first input wire that matches it in terms of block size and number of channels. If a match is found, the data from that input wire is copied to the output wire, and this input wire is no longer used for subsequent matches. If not match is found for a particular output wire, then all samples are set to 0.0. This process repeats for all output wires.

If a module's m-file does not specify a bypass function, then the code generator assigns IOMatchUpModule_Bypass() by default.

5. Generating Module Libraries

This section provides further details on module generation. We begin by discussing template substitution, the underlying process by which the generated .c and .h files are created. The section also documents MATLAB commands that are related to module generation. Many of the commands shown were used in previous examples and are fully documented here.

The main phases in creating an audio module library are:

1. For each audio module in the library
    a. Add code markers
    b. Generate the module's .c and .h files using template substitution
2. Generate the combined schema file describing all of the modules in the library
3. Translate the schema information into a C data structure that can be compiled and included with the DLL.
4. Compile the generated files and create an audio module dynamic link library (DLL).

Steps 1 and 3 are automated using the awe_generate_library.m command. Many other MATLAB functions are internally called. Only a few other functions are documented here to help you better understand the code generation process. Step 4 is done separately in VisualStudio and described in Section 6.

5.1. awe_generate_library.m

```
awe_generate_library(MM, DIR, LIBNAME, USESDLLS, GENDOC)
```

This is the primary function for generating an audio module library. It generates sources files for all of the modules in the library. Arguments:

MM – cell array of audio module structures.

DIR – base directory in which the generated files should be placed. Files are written here and in several other subdirectories.

LIBNAME – string specifying the name of the library. For example, 'Core32'.

USESDLLS – structure specifying other module libraries that the newly built library depends upon. A library has a dependencies when it internally utilizes a module contained in another library. By default, USESDLLS=[] and the library is independent of others.

GENDOC – integer indicating whether module documentation should be generated. By default, GENDOC=0 and no documentation is generated. If GENDOC=1, then a single document is created for the entire module library. If GENDOC=2, then a separate document is created for each audio module.

5.1.1. Creating the Cell Array of Modules

The argument, MM, is a cell array populated with modules to be placed in the final library. For example, it might contain:

```
MM=cell(0,0);
MM{end+1}=downsampler_example_module('temp');
MM{end+1}=fader_example_fract32_module('temp');
MM{end+1}=fader_example_module('temp');
MM{end+1}=lah_limiter_example_module('temp');
MM{end+1}=peak_hold_example_fract32_module('temp');
MM{end+1}=peak_hold_example_module('temp');
MM{end+1}=scaler_example_module('temp');
MM{end+1}=scaler_smoothed_example_module('temp');
```

5.1.2. Specifying Dependencies

USESDLLS is a cell array of structures that specifies dependencies. Each structure contains the fields:

.MM – cell array of audio modules in the library.

.str – library name.

To achieve automation, each audio module library supplied with Audio Weaver has a master function which generates the library. For example, the Core32 module library uses the script make_core32.m. The script is designed so that if you request output arguments, as in

```
[MM, NAME]=make_core32;
```

the function will not build the library but will instead return a cell array of modules in MM and the name of the module library in NAME. You should follow this convention when creating your own custom module libraries. This allows you to specify dependencies in a straightforward manner. For example, the make_examples.m script is dependent upon 3 other libraries:

```
[DependMM, DependName]=make_core32(0);
USESLIB{1}.MM=DependMM;
USESLIB{1}.str=DependName;
[DependMM, DependName]=make_basicaudiofloat32(0);
USESLIB{2}.MM=DependMM;
USESLIB{2}.str=DependName;
[DependMM, DependName]=make_basicaudiofract32(0);
USESLIB{3}.MM=DependMM;
USESLIB{3}.str=DependName;
```

If you try to build a library and do not properly account for all dependencies, then awe_make_library.m will report an error. For example, suppose that you try and build the examples library but eliminate the dependency on the BasicAudioFloat32 library, you'll see:

```
Module library has unsatisfiable or circular dependencies.
Modules with dependencies that could not be resolved:
FaderExample
      Missing ScalerSmoothed
LAHLimiter
      Missing AGCLimiterCore
      Missing AGCMultiplier
      Missing MaxAbs
```

The error message indicates that the FaderExample module has a dependency on ScalerSmoothed and that ScalerSmoothed is not in the dependency list. Similarly, LAHLimiter has 3 unsatisfied dependencies.

5.1.3. Specifying the Output Directory

The argument DIR to awe_generate_library.m specifies the directory in which to generate the modules. There are different ways to specify the directory including hard coding it.

Often it is useful to specify the directory relative to the location of the master library make script. The following MATLAB code, contained in make_examples.m, shows how to determine the location of make_examples.m and then pull off the lowest level directory. make_examples.m is located at:

```
<AWE>\ModulesLib\Examples\matlab\
```

The output directory is set to

```
<AWE>\ModulesLib\Examples\
```

by the code below.

```
MFILE=mfilename('fullpath');
[pathstr, name]=fileparts(MFILE);

% Remove the last directory level
ind=find(pathstr == filesep);
ind=max(ind);

DIR=pathstr(1:ind-1);
```

5.1.4. Generated Files

Let DIR be the base directory for the audio module library and let 'Examples' be the name of the library. awe_generate_library.m creates the following files:

<DIR>\Examples.h – Header file containing extern definitions for each module class object. It also has macros defining the entire list of modules and all dependencies.

<DIR>\Examples.sch – Overall schema file for the library.

<DIR>\ExamplesSchema.cpp – Compiled schema file.

<DIR>\Doc – Location of generated documentation

<DIR>\Include – Location of generated module include files

<DIR>\matlab – Location of the module m-files. This also contains the master library make script.

<DIR>\matlab\code – Suggested location for the inner code pieces.

<DIR>\matlab\process – Suggested location of the MATLAB versions of the processing functions.

<DIR>\matlab\test – Suggested location of MATLAB test scripts which validate the operation of the modules.

<DIR>\Source – Location of the generated module source files.

5.1.5. SchemaBuilder.exe

You'll notice this executable within the Audio Weaver Bin directory. This executable compiles the schema information. That is, it takes a .sch file and compiles into a .cpp file. The .cpp file is then included in the project for building the module DLL. SchemaBuilder.exe is automatically called by awe_generate_library.m and there is no need to call it separately.

5.1.6. AWE_INFO.buildControl

The global variable AWE_INFO contain the structure .buildControl which controls the library generation process. AWE_INFO.buildControl contains many fields which are used internally. We point out the user settable values.

```
AWE_INFO.buildControl.combineSourceFiles
```

This Boolean specifies whether each audio module should be placed into a separate source file. By default, combineSourceFiles=0 and each module is written to a separate file.

```
AWE_INFO.buildControl.indentSourceFiles
```

This Boolean specifies whether the source code should be formatted after generation using the executable indent.exe. Formatting the code regularizes identing, line breaks, etc. By default,

indentSourceFiles=0. Formatting the source code significantly slows down the module generation process.

5.1.7. Setting the Audio Module Search Path

When you are developing a custom audio module, you must add the base module directory to the Audio Weaver module search path. *It is not enough to merely modify your MATLAB path.* To add a directory to the Audio Weaver module search path, use the command

```
add_module_path(DIR)
```

The command adds the directory

```
<DIR>\matlab
```

to the MATLAB search path; this directory has to exist. It then optionally adds

```
<DIR>\matlab\test
<DIR>\matlab\process
```

if these directories exist. In addition, the function modifies the global variable

```
AWE_INFO.buildControl.modulePath
```

to include the new directory. This global variable is used when searching for class IDs.

The newly added module directory is placed at the end of the module search path. You can specify that the directory be added to either the beginning or end of the module path using the optional second argument

```
add_module_path(PATH, '-begin');
add_module_path(PATH, '-end');
```

5.1.8. Specifying Class IDs

Every audio module in the library must have a unique class ID. This 32-bit integer is specified in the file

```
<DIR>\classids.csv
```

This comma separated text file contains entries of the form

```
ClassName,ID
```

The file can also contain a single entry at the start of the form:

```
IDOFFSET=32768
```

The IDOFFSET is added to each ID listed in the file. Using IDOFFSET allows you to quickly make wholesale changes to all of the modules in the library.

DSP Concepts reserves class IDs in the range 0 to 32767 for module libraries shipped with Audio Weaver. You are free to use any IDs in the range 32768 to 63487 for your custom modules.

Audio Weaver provides a function for looking up classIDs across a range of audio module libraries.

```
ID=classid_lookup(CLASSNAME)
```

You pass the function a class name string and the classID is returned. The function works by using the audio module directory search path contained in

```
AWE_INFO.buildControl.modulePath
```

For each directory, the file classids.csv is opened and examined. The function classid_lookup.m uses internal caching to speed up the search process.

5.1.9. Reordering of Render Variables

As part of the build process, Audio Weaver reorders the variables in a module or subsystem to match the manner in which modules are instantiated by the Server. The reordering is performed by the function

```
M=awe_reorder_variables(M)
```

where M is an @awe_module or @awe_subsystem object. Variables are ordered according to the rules

1. All non-hidden scalar variables
2. All hidden scalar variables
3. All array variables (pointers)
4. Pointers to internal modules (subsystems only)

The ordering enforces that all scalar variables, which are initialized by the base module constructor function, are located at the start of the instance structure. Refer to Section 4.4.1 for additional details.

5.1.10. Overwriting Existing Source Files

Audio Weaver is frequently used in conjunction with a source code control system. The awe_generate_library.m function overwrites generated files only if there is an actual change to the file. If nothing has changed, the file is untouched.

Each generated file is first written to a temporary file. Then the temporary file is compared with the existing version. If the files are identical, then the existing file is untouched and the temporary file is deleted. If the temporary file differs from the existing file, then the existing file is overwritten.

5.1.11. Specifying Wiring Constraints

The .wireAllocation field of an audio module allows you to specify wiring constraints for the routing algorithm. There are two possible values

'distinct' – None of the input wire buffers will be used as output wire buffers. That is, the outputs will be "distinct" from the inputs.

'across' – The routing algorithm will try and reuse wire buffers across the module, whenever possible. That is, the wire buffer assigned to input pin N will be reused for output pin N. The wires are the same "across" the module.

awe_module.m sets .wireAllocation to 'distinct' by default. This is the safest approach but consumes more memory. 'across' conserves memory and is often easier to debug. When 'distinct' is specified, the output wires will always be different compared to the inputs. When 'across' is specified, the routing algorithm will attempt to reuse the buffers, but this may not always be the case.

The choice between 'distinct' and 'across' depends upon the internal details of your processing algorithm. Buffer pointer usage within the algorithm has to be studied to determine if 'across' wiring is possible. Consider the scaler module processing function shown below. This module has N inputs and N outputs with the Nth input being scaled and written to the Nth output.

```
void awe_modScalerProcess(void *pInstance)
{
  awe_modScalerInstance *S = (awe_modScalerInstance *)pInstance;
  WireInstance **pWires = ClassModule_GetWires(S);
  int numPins = ClassModule_GetNInWires(S);
  int pin;

  for(pin=0;pin<numPins;pin++)
    {
      awe_vecScale((float *)(pWires[pin]->buffer), 1,
                   (float *)(pWires[numPins+pin]->buffer), 1,
                   S->gain, ClassWire_GetNumSamples(pWires[pin]));
    }
}
```

It is safe to use 'across' wiring in this case. Now consider the Mixer module. This module has 1 input pin with M channels and 1 output pin with N channels. Each output channel is formed as a weighted sum of input channels with each output channel written in order. In this case, you can no longer use 'across' wiring because each output channel depends upon each input channel. If 'across' wiring were used, then writing the first output channel would overwrite the first input channel.

Note that 'across' is a suggestion to the routing algorithm. In certain cases, 'across' wiring cannot be used and distinct buffers will be provided. You should always write your audio module processing function anticipating that the buffer pointers may be distinct. For example, the inner loop of the awe_modScalerExampleProcess function should not be written as:

```
for(pin=0;pin<numPins;pin++)
  {
    awe_vecScale((float *)(pWires[pin]->buffer), 1,
             (float *)(pWires[pin]->buffer), 1,
             S->gain, ClassWire_GetNumSamples(pWires[pin]));
  }
```

The routing algorithm may decide to assign a distinct output buffer; always pass in the given output buffer pointer.

The most common situation when 'across' wiring cannot be applied is when a module sits between the input and output of a system. For routing reasons, the wires attached to the input and output of a subsystem have to be distinct. Placing a module with .wireAllocation='across' between them will still lead to distinct buffer allocation.

5.1.12. AudioWeaverModule Tag

The function awe_help.m displays a list of all Audio Weaver modules on the current module search path. In order for a module m-file to be picked up, you need to add the tag AudioWeaverModule somewhere within the m-file. This tag allows you to differentiate between actual module m-files and other MATLAB files contained in the same directories. The tag can be placed anywhere within the file, usually in a comment as shown below:

```
% AudioWeaverModule [This tag makes it appear under awe_help]
```

5.2. Code Markers and Template Substitution

Template substitution is an automated method of replacing strings within a file to generate an output file. The process starts with a template file and Audio Weaver provides separate templates for the generated .c and .h files:

```
<AWE>\matlab\module_generation\templates\awe_module_template.h
<AWE>\matlab\module_generation\templates\awe_module_template.c
```

A template file contains normal text interspersed with entries of the form

```
$IDENTIFIER$
```

where IDENTIFIER is a string. The template substitution function is given a list of identifiers together with their replacements. The substitutions are performed and a new output file is created. If an identifier within a file is not defined, then it is deleted and does not appear in the output.

5.2.1. Adding Code Markers

A "code marker" is an identifier together with a replacement string. Each module and subsystem has its own list of code markers. The MATLAB function

```
awe_addcodemarker(M, IDENTIFER, SUBSTITUTION)
```

creates a new code marker. The function arguments are:

M - @awe_module or @awe_subsystem object

IDENTIFIER – string indicating a specific identifier in a template file.

SUBSTITUTION – string containing the replacement value for the identifier.

SUBSTITUTION can be either a single string or a cell array of strings for multi-line replacements. If you call awe_addcodemarker.m and an IDENTIFIER of the specified name already exists, then the new SUBSTITUTION string is appended as another line. For example, the following entry located near the top of awe_module_template.c file is used to specify include files:

```
$srcFileInclude$
```

If you call

```
awe_addcodemarker(M, 'srcFileInclude', '#include "file1.h"');
awe_addcodemarker(M, 'srcFileInclude', '#include "file2.h"');
```

Then the generated file will include both lines:

```
#include "file1.h"
#include "file2.h"
```

An optional 4th argument selects between appending and overwriting duplicate code markers.

```
awe_addcodemarker(M, IDENTIFER, SUBSTITUTION, APPEND)
```

By default, APPEND=1 and duplicate code markers are appended. If you set APPEND=0, then an existing code marker of the same name is overwritten. For example,

```
awe_addcodemarker(M, 'srcFileInclude', '#include "file1.h"');
awe_addcodemarker(M, 'srcFileInclude', '#include "file2.h"', 0);
```

Since $srcFileInclude$ is already defined, the second call to awe_addcodemarker.m will overwrite the code marker and the generated file will only contain

```
#include "file2.h"
```

The call awe_addcodemarker.m adds the code marker information into the field .codeMarker of the module. You could, for example, examine the code markers defined for the scaler_module.m as follows:

```
>> M=scaler_module('fred');
>> M.codeMarker{1}

ans =

    name: 'processFunction'
    text: 'Insert:code\InnerScaler_Process.c'

>> M.codeMarker{2}

ans =

    name: 'discussion'
    text: {1x9 cell}
```

5.2.2. Inserting Files

In many cases, the SUBSTITUTION string contains many lines and it is unwieldy to represent in MATLAB. Instead, it is easier to copy the SUBSTITUTION text from another file. If the SUBSTITUTION string has the form "Insert:filename.txt" then the text will be copied from filename.txt. filename.txt is referenced relative to the location of the module's m-file. You can specify subdirectories as well. For example:

```
awe_addcodemarker(M, 'processFunction', ...
      'Insert:code\InnerPeakHold_Process.c');
```

inserts the file InnerPeakHold_Process.c located in the subdirectory "code".

5.2.3. awe_lookupcodemarker.m

The MATLAB function

```
STR=awe_lookupcodemarker(M, IDENTIFIER)
```

returns the contents of a code marker. If IDENTIFIER is not defined, then the function returns an empty string. The returned value STR is either a string, for simple replacements, or a cell array in the case of multi-line replacements.

5.2.4. awe_deletecodemarker.m

```
M=awe_deletecodemarker(M, IDENTIFIER)
```

Deletes the code marker named IDENTIFIER.

```
M=awe_deletecodemarker(M)
```

Deletes all code markers associated with a module or subsystem.

5.2.5. Template Substitution Preprocessor

The template files also contain their own preprocessor directives. These directives are similar to those used by the C preprocessor but are handled by the template substitution function. The directives are identified by ## and only a few variations are supported.

```
##if 1
… This code will appear in the output file …
##endif

##if 0
… This code will NOT appear in the output file …
##endif
```

When the argument to ##if is 1, then the text between the ##if and ##endif will appear in the generated file. Otherwise, the text will not appear. You can also use an identifier with an ##if statement as in

```
##if $myVariable$
Optional text
##endif
```

If you set

```
awe_addcodemarker(M, 'myVariable', '1')
```

then the optional text will appear in the generated file. Note that $myVariable$ is being set to the string '1' not to the numeric value 1. The template preprocessor directive also supports an else clause

```
##if $combineSourceFiles$
#include "$combinedIncludeName$"
##else
#include "$baseHFileName$"
```

```
##endif
```

5.2.6. Frequently Defined Code Markers

This section lists out code markers that are frequently defined when developing custom audio modules.

5.2.6.1. $bypassFunction$

C code which specifies the inner portion of the bypass function. When this is defined, the bypass function is written into generated C file.

5.2.6.2. $bypassFunctionName$

Allows you to specify a bypass function by name. This is typically used in conjunction with one of the predefined bypass functions listed in Section 4.4.5. For example,

```
awe_addcodemarker(M, 'bypassFunctionName', 'IOAcrossModule_Bypass');
```

The following logic defines the bypass function.

1. If $bypassFunction$ is defined, then the code is placed into the generated .c file and $bypassFunctionName$ is set to awe_modClassNameBypass,
2. If $bypassFunctionName$ is defined, then this function is used for bypass behavior. No additional code is placed into the generated .c file.
3. Otherwise, the default bypass function IOMatchUpModule_Bypass() is used. No additional code is placed into the generated .c file.

5.2.6.3. $constructorFunction$

Specifies the inner portion of a custom constructor function. The function must follow the calling convention outlined in Section 4.4.1.

Simple audio modules without indirect arrays – No need to define the constructor function. The BaseClassModule_Constructor() usually does everything needed.

Audio modules with direct arrays – Set the .arrayHeap and .arraySizeConstructor fields of the array variables in MATLAB. Audio Weaver will then auto generate a suitable constructor function.

Complex modules requiring further initialization – Write a custom $constructorFunction$ or use the $postConstructorFunction$ shown below.

Modules created out of subsystems – Audio Weaver automatically generates code for

these systems. If you need further initialization, use the $postConstructorFunction $ shown below.

5.2.6.4. $postConstructorFunction$

This code marker follows immediately after the $constructorFunction$ code marker in awe_module_template.c. This marker allows you to add your own code in cases where Audio Weaver generates the memory allocation and instantiation code but further initialization is needed. It is particularly useful for subsystems.

5.2.6.5. $discussion$

Defines the discussion section of the help documentation. This has been grandfathered for backwards compatibility. When writing new modules, set the M.docInfo.discussion field of the audio module instead.

5.2.6.6. $getFunction$

Specifies the inner portion of a module's Get function. Follows the calling convention outlined in Section 4.4.4.

5.2.6.7. $setFunction$

Specifies the inner portion of a module's Set function. Follows the calling convention outlined in Section 4.4.3.

5.2.6.8. $hFileDefine$

Located near the top of the template file awe_module_template.h. Allows you to add additional preprocessor directives (or anything else) at the start of the header file. Example,

```
awe_addcodemarker(M, 'hFileDefine', '#define TRUE 1');
```

5.2.6.9. $hFileInclude$

Located near the top of the template file awe_module_template.h. Allows you to include additional header files.

```
awe_addcodemarker(M, 'hFileInclude', '#define <stdlib.h>');
```

5.2.6.10. $processFunction$

Specifies the inner code for the module's processing function. This must always be defined for modules. For subsystems, the auto-generated should be used. Refer to Section 4.4.2 for a discussion of how to define this function.

5.2.6.11. $preProcessFunction$

This code marker is located immediately before the $processFunction$ marker. It is typically used with subsystems when you want to insert your code custom code immediately prior to the auto-generated $processFunction$.

5.2.6.12. $postProcessFunction$

Similar to $preProcessFunction$ but the code here is inserted immediately after the $processFunction$. It is typically used with subsystems when you want to insert your custom code immediately after the auto-generated $processFunction$.

5.2.6.13. $srcFileDefine$

Located near the top of the template file awe_module_template.c. Allows you to add additional preprocessor directives (or anything else) at the start of the source file.

5.2.6.14. $srcFileInclude$

Located near the top of the template file awe_module_template.c. Allows you to specify additional includes files.

5.2.6.15. $hFileTemplate$ and $srcFileTemplate$

Audio Weaver uses the default template files specified at the start of Section 5.2. You can override the templates used by a particular module by setting these code markers. There are separate markers for the header file, $hFileTemplate$, and the source file, $srcFileTemplate$. It is handy to override these values if you want to change copyright information in the generated files.

5.3. Fine Tuning Code Generation

This section contains further instructions for fine tuning code generation. Many of the items apply to modules generated using compiled subsystems.

5.3.1. Automatically Generated Array Initializers

In many cases, an audio module requires a custom constructor function solely for the purpose of allocating indirect arrays. If the size of the allocated memory can be defined using existing module variables, then Audio Weaver can automatically generate a constructor function.

The generated array constructor code is of the form:

```
if ((S->ARRAYVAR = (TYPE *) awe_fwMalloc(SIZE, HEAP, retVal)) == 0)
{
    // Error code is in *retVal
```

```
        return 0;
    }
```

where

ARRAYVAR – is the name of the variable

TYPE – is the type of the module (either float, fract32, or int)

SIZE – is a user specified string

HEAP – is a user specified string.

ARRAYVAR and TYPE are already known to MATLAB. SIZE and HEAP must be separately specified as C code to embed into the generated code. The string SIZE is specified in MATLAB by setting field .arraySizeConstructor field of the array variable. Similarly, HEAP is specified by setting the .arrayHeap field of the array variable. For an example, see the Peak Hold module in Section 7.2

5.3.2. Specifying Module Constructor Arguments in Subsystems

This advanced technique is frequently used when compiling subsystems to generate new module classes. Audio Weaver automatically generates a Constructor function for the compiled subsystem, and this constructor in turn calls the constructor functions of the internal modules. The difficulty arises because the arguments to the internal constructor functions are hard coded in the generated file based on the values at code generation time. This is not always the desired behavior.

To get around this problem, Audio Weaver allows you to specify a piece of C code which overrides the default numeric initializer for an internal module variable. The C code is written to the .constructorCode field of the variable. For an example, refer to the look ahead limiter example of Section 7.5.

5.3.3. Avoiding Wire Allocation Problems with Subsystems

This issue also applies to the case of compiling subsystems to form new module classes. If your subsystem has .flattenOnBuild=1 (which means that it is not compiled), then you can ignore this section. This problem arises because Audio Weaver modules can operate on an arbitrary number of channels. In certain circumstances, the routing algorithm incorrectly allocates wire buffers during code generation such that audio processing crashes at run-time. In these cases, you need to be careful how you specify default channel counts and block sizes within modules. The look ahead limiter example of Section 7.4 demonstrates the problem and presents a solution.

5.4. unique_classes.m

```
C=unique_classes(SYS)
```

where

SYS - @awe_subsystem object.

Returns a list of all the unique module and subsystem classes found within subsystem SYS. The function recursively searches through the system and identifies individual modules and subsystems. The return result C is a cell array of strings. If called without any output arguments, the function displays the class list to the MATLAB output window. The class name of the input SYS is not included in the C. This function is useful for dependency checking.

```
[C, M]=unique_classes(SYS)
```

An optional second output receives a cell array of the actual modules/subsystems listed in C.

SYS can also be a cell array of subsystems. In this case, the function returns the overall set of unique classes needed to create all of the systems in SYS.

```
[C, M]=unique_classes(SYS CLIST)
```

An optional Boolean argument CLIST specifies that the list of unique module classes should be displayed in a manner appropriate for inclusion in the TargetInfo.h file.

For example, to determine all of the unique module classes needed for the agc_example, type:

```
>> SYS=agc_example;
>> unique_classes(SYS)
Unique classes within the subsystem of class: test
AGC
AGCCore
AGCMultiplier
Meter
ScalerDB
```

If you want to build a target with only the modules required for the agc_example, you would set CLIST=1. The following C code is then printed to the output window:

```
extern const ModClassModule awe_modAGCClass;
extern const ModClassModule awe_modAGCCoreClass;
extern const ModClassModule awe_modAGCMultiplierClass;
extern const ModClassModule awe_modMeterClass;
extern const ModClassModule awe_modScalerDBClass;


#define LISTOFCLASSOBJECTS \
&awe_modAGCClass, \
&awe_modAGCCoreClass, \
&awe_modAGCMultiplierClass, \
&awe_modMeterClass, \
&awe_modScalerDBClass
```

This text can be pasted directly into TargetInfo.h.

5.5. awe_generate_module.m

M=awe_generate_module(M, DIRECTORY, WRITEFILES)

Internal function used by awe_generate_library.m. In some cases, such as when generating documentation, you may want to call this function directly. Arguments:

M - @awe_module object.

DIRECTORY – directory in which the generated files should be written to. By default, this is set to MATLAB's current working directory (returned by the pwd command).

WRITEFILES – an optional Boolean argument which specifies whether the generated files should actually be written. If you set WRITEFILES=0, then the module M will be updated with code markers, but no output files will be written. Use WRITEFILES=0 when generating documentation as described in the next Section.

6. Generating Documentation

In addition to generating source and header files, Audio Weaver also generates documentation for modules and subsystems in Microsoft Word format. The generated documentation is shown in Figure 1 as item 5. The information in the documentation is pulled from several different places:

- The help comments at the start of the MATLAB M-file. [manual]
- Inserted code markers. [manual]
- Variable and module information. [automatic]
- Input, output, and scratch pins. [automatic]
- Subsystem drawings. [automatic]

In the list above, "manual" indicates that the content is created by the module developer. "automatic" indicates that the content is generated by Audio Weaver based on information in the module m-file.

6.1. awe_help

For example, to see the generated help for the scaler smoothed module, type

```
awe_help scaler_smoothed_example_module
```

at the MATLAB command line. The documentation shown in Figure 2 appears.

Behind the scenes, MATLAB checks if documentation already exists for the module. Since the scaler_smoothed_example_module.m function exists in the directory

```
<Audio Weaver>\ModuleLibs\Examples
```

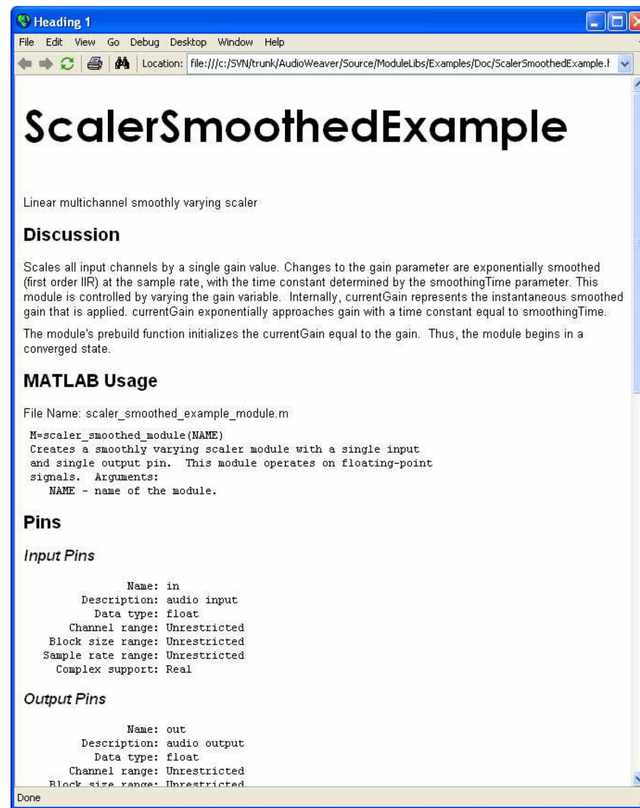
it checks if the file

```
<Audio Weaver>\ModuleLibs\Examples\Doc\ScalerSmoothedExample.html
```

exists. The name "ScalerSmoothedExample.html" is based on the module's class name. If the file exists, it is opened and displayed in the MATLAB HTML browser. If it doesn't exist, then the documentation is generated on the fly using Microsoft Word and then saved in HTML format. The process of generating new documentation is quite involved and can take several seconds to complete.

If the HTML file exists, you can force it to be rebuilt using an optional argument:

```
awe_help hilbert_module -rebuild
```



ScalerSmoothedExample

Linear multichannel smoothly varying scaler

Discussion

Scales all input channels by a single gain value. Changes to the gain parameter are exponentially smoothed (first order IIR) at the sample rate, with the time constant determined by the smoothingTime parameter. This module is controlled by varying the gain variable. Internally, currentGain represents the instantaneous smoothed gain that is applied. currentGain exponentially approaches gain with a time constant equal to smoothingTime.

The module's prebuild function initializes the currentGain equal to the gain. Thus, the module begins in a converged state.

MATLAB Usage

File Name: scaler_smoothed_example_module.m

```
M=scaler_smoothed_module(NAME)
Creates a smoothly varying scaler module with a single input
and single output pin.  This module operates on floating-point
signals.  Arguments:
   NAME - name of the module.
```

Pins

Input Pins

```
             Name: in
      Description: audio input
        Data type: float
    Channel range: Unrestricted
 Block size range: Unrestricted
Sample rate range: Unrestricted
  Complex support: Real
```

Output Pins

```
             Name: out
      Description: audio output
        Data type: float
    Channel range: Unrestricted
 Block size range: Unrestricted
```

Figure 2. Automatically generated documentation for the scaler_smoothed_example_module.m.

6.2. awe_generate_doc.m

awe_help.m uses the function awe_generate_doc.m behind the scenes. Using awe_generate_doc.m directly gives you more control over the final documentation including the ability to create a single document describing an entire module library.

```
awe_generate_doc(M, FILENAME)
```

Generates help documentation for an audio module in a specified format. Arguments:

M - @awe_module object.

FILENAME - file in which to store the resulting Word document in. This needs to have a .doc extension.

The audio module must already have been through the code generation process and have suitable code markers added.

If M is a cell array containing modules, then the function generates documentation for the entire

library. By default, modules appear in the documentation in the same order as in the cell array M. An optional third argument, SORT, allows you to specify that the modules should be sorted by their classNames.

```
awe_generate_doc(M, FILENAME, SORT)
```

6.3. Selecting the Documentation Format

The global variable AWE_INFO.docControl.format determines the format of the generated documentation. By default, this is set to

```
AWE_INFO.docControl.format='html';
```

and documentation is displayed as HTML. Switching to

```
AWE_INFO.docControl.format='doc';
```

causes the documentation to be displayed in Microsoft Word format. In both cases, Microsoft Word is used to generate the underlying document and the .format variable controls which format the document is saved in. Generating documentation requires that you have Microsoft Word installed.

Note that .docControl does not affect the HTML format used by awe_help.m.

6.4. Guidelines for Documenting Modules and Subsystems

The generated documentation relies upon two manually written sections as shown above. We suggest that you use the following guidelines when documenting modules. Within the help comments at the start of the M-file discuss:

- A brief summary of the module.
- The purpose of each argument to the M-file and its default value.

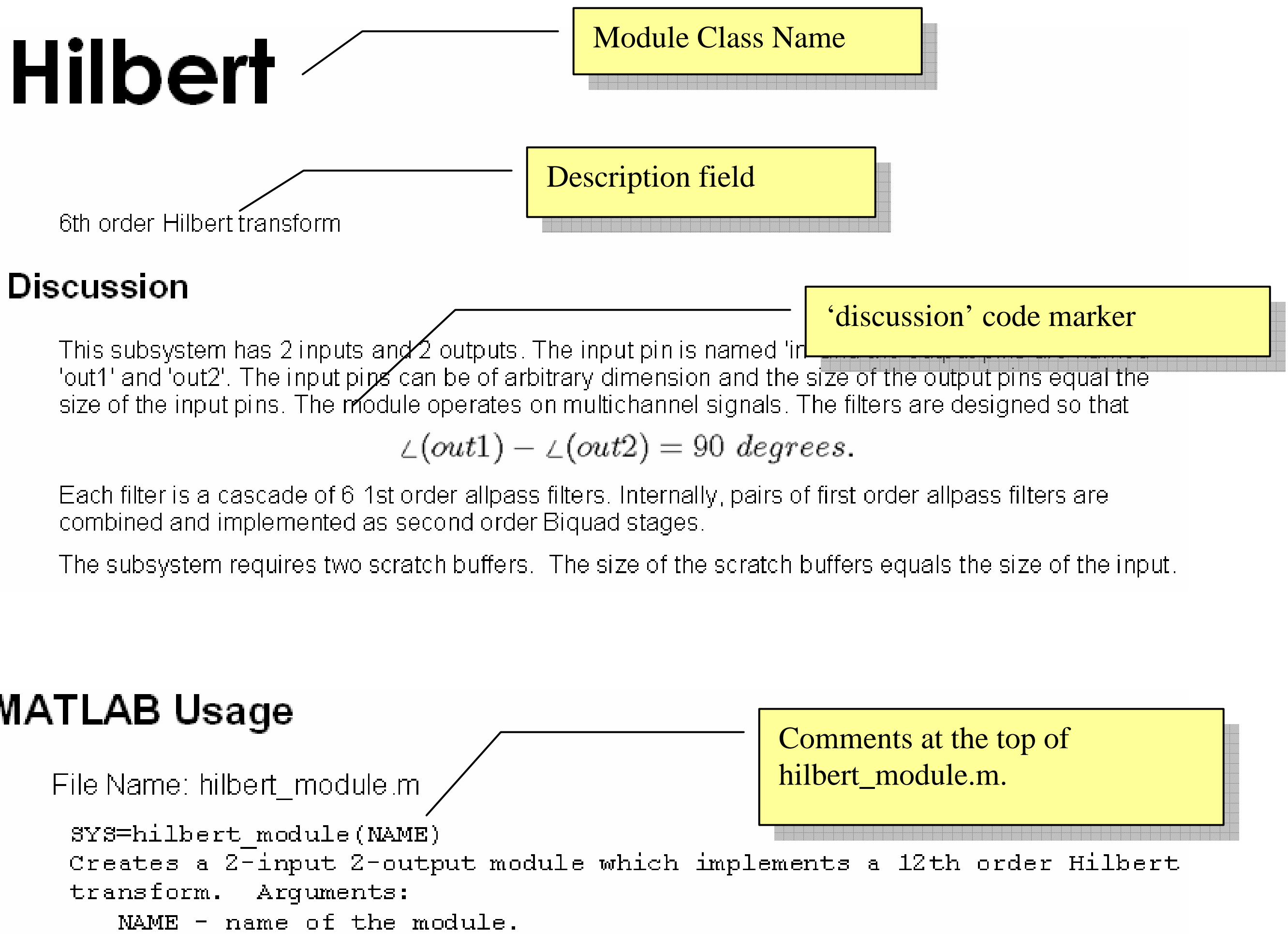
The module field .docInfo.discussion (or a code marker named 'discussion’) contains the text which is inserted under the “Discussion” section of the documentation. The discussion should contain:

- A lengthy description of how the module operates.
- Whether the module has a fixed or variable number of pins.
- Scratch pin requirements and how the sizes relate to the sizes of the input and output pins. (The automatically generated information lists out the scratch pin sizes, but the values shown are only based on the current module settings. The relationship between the

scratch sizes and other pin sizes is not described in the automatically generated text.)

6.5. Hilbert Module Example

We now walk through the automatically generated help for the hilbert_module.m and describe where each of the items shown is taken from.



Pins

Input Pins

Automatically generated based on the input pin information.

```
             Name: inReal
      Description: Audio Input for real valued signals
        Data type: float
    Channel range: Unrestricted
 Block size range: Unrestricted
Sample rate range: Unrestricted
  Complex support: Real

             Name: inImag
      Description: Audio Input for imaginary valued signals
        Data type: float
    Channel range: Unrestricted
 Block size range: Unrestricted
Sample rate range: Unrestricted
  Complex support: Real
```

Output Pins

Automatically generated based on the output pin information.

```
             Name: outReal
      Description: Audio output for real valued signals
        Data type: float
    Channel range: Unrestricted
 Block size range: Unrestricted
Sample rate range: Unrestricted
  Complex support: Real

             Name: outImag
      Description: Audio output for imaginary valued signals
        Data type: float
    Channel range: Unrestricted
 Block size range: Unrestricted
Sample rate range: Unrestricted
  Complex support: Real
```

Scratch Pins

Automatically generated based on the scratch pin information.

```
Channel count: 1
   Block size: 32
  Sample rate: 48000

Channel count: 1
   Block size: 32
  Sample rate: 48000
```

Type Definition

```
typedef struct _ModuleHilbert
{
        ModuleBiquadClass   *bq11;
        ModuleBiquadClass   *bq21;
        ModuleBiquadClass   *bq12;
        ModuleBiquadClass   *bq22;
        ModuleBiquadClass   *bq13;
        ModuleBiquadClass   *bq23;
} ModuleHilbertClass;
```

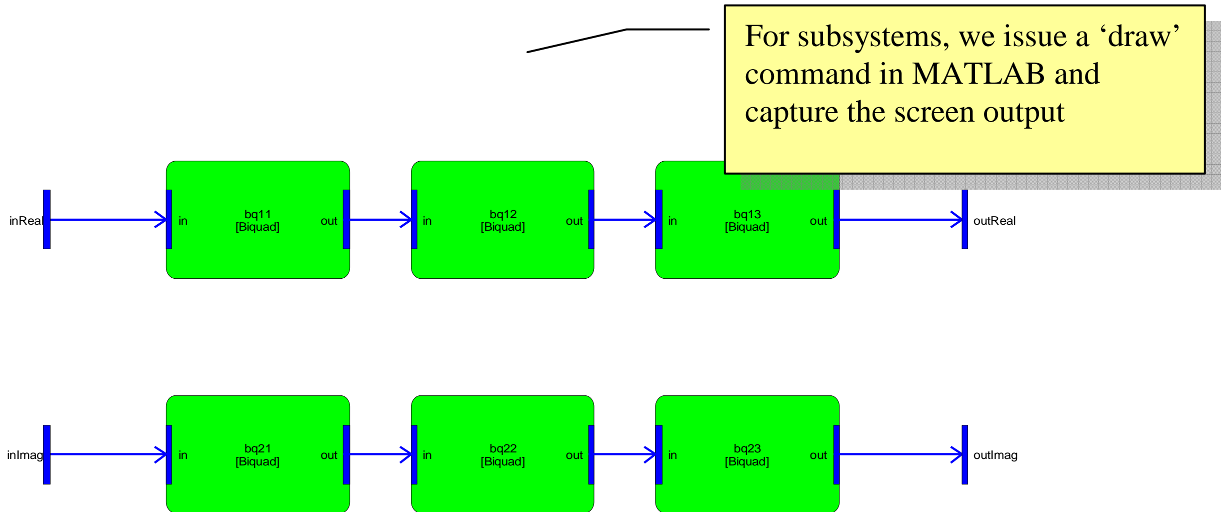
Automatically generated based on the variables and modules that are part of the subsystem.

Subsystem Information

- bq11 - 2nd order IIR filter
- bq21 - 2nd order IIR filter
- bq12 - 2nd order IIR filter
- bq22 - 2nd order IIR filter
- bq13 - 2nd order IIR filter
- bq23 - 2nd order IIR filter

Automatically generated based on the variables and modules that are part of the subsystem.

tempName [Hilbert]



C Usage

C Usage information is automatically generated.

Source Files

```
ModHilbert.c
ModHilbert.h
```

Class Structure

```
ClassModule_Hilbert
```

Constructor Function

```
ModInstanceDescriptor * ModuleHilbert_Constructor(
     int * FW_RESTRICT retVal,
     UINT nIO, WireInstance ** FW_RESTRICT pWires,
     size_t argCount,
     const Sample * FW_RESTRICT args);
```

6.6. Adding Equations to Documentation

Audio Weaver supports adding equations to the discussion section of a module. Equations are

entered as text commands within the 'discussion' portion in LaTeX format. Equations must be enclosed between dollar signs. For example,

```
$\angle(out1)-\angle(out2) = 90~degrees.$
```

is expanded to

$$\angle(out1) - \angle(out2) = 90~degrees.$$

in the document. If you are new to LaTeX, there are many good references available on-line.

7. The Examples Module Library

This section discusses examples which illustrate advanced module design techniques. The modules are part of the Examples module library introduced in Section 2. In addition to the scaler_smoothed_example_module.m, the library contains

scaler_example_module.m – Simple scaler. This module demonstrates how to support multiple input and output pins each with, possibly, a different number of channels.

peak_hold_example_module.m – Similar to a level meter in that it measures the peak signal level. This module demonstrates how to allocate arrays and update array sizes within the module's prebuild function. A fixed-point version of this example is also provided: peak_hold_example_fract32_module.m.

fader_example_module.m – Front-back fader which pans a signal between two different outputs. This module demonstrates compiled subsystems, that is, how to build a new module class out of a subsystem. A fixed-point version of this example is also provided: fader_example_fract32_module.m.

downsampler_example_module.m – Downsampler which keeps 1 out of every D samples; no filtering is performed. This module demonstrates how to operate on arbitrary 32-bit data types as well as changing the output block size and sample rate.

lah_limiter_example_module.m – Look ahead limiter delivered as a compiled subsystem. This module demonstrates some advanced features of compiled subsystems including initializing internal module variables and clarifying wire buffer usage.

The example library is found in the

```
<AWE>\ModuleLibs\Examples
```

directory. Run the main script make_examples.m to generate code for the entire library.

7.1. Scaler Module

This example is contained in the file scaler_example_module.m. It is equivalent to scaler_module.m provided in the BasicAudioFloat32 module library. This example illustrates how to handle an arbitrary number of input and output pins. This module implements a straightforward linear scaler without any smoothing. The module operates on an arbitrary number of input pins and uses the same scale factor across all pins.

The input arguments to the MATLAB constructor function are:

```
function M=scaler_float_module(NAME, NUMPINS)
```

The first argument is the name of the module (a string) and the second argument specifies the number of input pins. By default, if the second argument is not supplied, the number of pins is set to 1.

```
if (nargin < 2)
    NUMPINS=1;
end
```

We next create and name the @awe_module object

```
M=awe_module('ScalerExample', 'Linear multichannel scaler');
M.name=NAME;
```

and set the MATLAB processing and prebuild functions

```
M.processFunc=@scaler_example_process;
M.preBuildFunc=@scaler_example_prebuild;
```

The .gain variable is added. This gain is applied across all channels:

```
add_variable(M, 'gain', 'float', 1, 'parameter', 'Linear gain');
```

The default range information for the variable is set. This can be changed after the module has been instantiated in MATLAB and is used when drawing the inspector. We also set the units to 'linear'. This is just for display purposes and as a reminder in the documentation.

```
M.gain.range=[-10 10];
M.gain.units='linear';
```

Input and output pins are added to the module. If NUMPINS is 1, then the pins are named 'in' and 'out'. Otherwise, if NUMPINS >1, the input pins are named 'in1', 'in2', etc. and the output pins named 'out1', 'out2', etc. Note that the module always has the same number of output pins as input pins.

```
PT=new_pin_type;
if (NUMPINS == 1)
    add_pin(M, 'input', 'in', 'Input signal', PT);
    add_pin(M, 'output', 'out', 'Output signal', PT);
else
    for i=1:NUMPINS
        add_pin(M, 'input', sprintf('in%d', i), 'Input signal', PT);
        add_pin(M, 'output', sprintf('out%d', i), 'Output signal', PT);
    end
end
```

The C file containing the processing function is specified

```
awe_addcodemarker(M, 'processFunction',
'Insert:code\InnerScalerExample_Process.c');
```

followed by documentation for the module:

```
awe_addcodemarker(M, 'discussion', {'The Scaler module scales
multichannel signals by a single gain value. ', ...
'By default, the module has a 1 input and 1 output pin.', ...
'', ...
'The module can also be configured for an arbitrary number of input and
output pins. ', ...
'In this case, the ith input pin is scaled and copied to the ith output
pin. ', ...
'The same scale factor is used in all cases and the dimensions and
sample rate of each signal can be arbitrary', ...
'', ...
'Note that the module is not smoothly varying.  If you update the gain
value, you may introduce an audible click. ', ...
'For clickless operation, use the ScalerSmoothed module instead.', ...
'', ...
'This module is an example to be used with the documentation.'});
```

We also indicate that the module can reuse the same wires for input and output.

```
M.wireAllocation='across';
```

The module is designed so that each input pin can have a different block size, number of channels, and sample rate. The module's prebuild function passes the pin type information of the ith input pin to the ith output pin. This is accomplished by the code shown below:

```
function M=scaler_example_prebuild(M)
for i=1:length(M.inputPin)
      M.outputPin{i}.type=M.inputPin{i}.type;
end
return;
```

It is also instructive to look at the module's processing function to see how it handles arbitrary size pins. The processing code is contained in the file InnerScalerExample_Process.c.

```
awe_modScalerExampleInstance *S = (awe_modScalerExampleInstance
*)pInstance;
WireInstance **pWires = ClassModule_GetWires(S);
int numPins = ClassModule_GetNInWires(S);
int pin;

for(pin=0;pin<numPins;pin++)
{
  awe_vecScale((float *)(pWires[pin]->buffer), 1,
             (float *)(pWires[numPins+pin]->buffer), 1,
             S->gain, ClassWire_GetNumSamples(pWires[pin]));
}
```

The macro

```
int numPins = ClassModule_GetNInWires(S);
```

gets the number of input pins from the module instance. We then loop over each pin and call the vector scaling function. The input data is located at:

```
pWires[pin]->buffer
```

and the output data is placed into

```
pWires[numPins+pin]->buffer
```

Each input pin can contain multiple channels and since the gain S->gain is applied to all samples, the data is treated as a single contiguous block. We set the input and output stride to 1 in the call to awe_vecScale(). The total number of samples in the pin comes from the macro

```
ClassWire_GetNumSamples(pWires[pin])
```

7.2. Peak Hold Module

The peak hold module demonstrates the use of variable size arrays and shows how the constructor function can be specified completely in MATLAB code. The peak hold module has a single multichannel input pin. The module computes the maximum absolute value for each channel on a block-by-block basis. The module records the peak absolute value seen since the module started running and also computes a peak detector with settable attack and decay rates. In many ways, the peak hold module is similar to a meter module working in block-by-block mode. This module is contained within the file peak_hold_example_module.m. A related fixed-point version is found in peak_hold_example_fract32_module.m

The module begins in the usual way. The MATLAB function takes a single argument which is the name of the module:

```
function M=peak_hold_module(NAME)
```

We then create the underlying module object, set its name and also set two internal functions.

```
M=awe_module('PeakHoldExample', 'Displays peak value');
M.name=NAME;
M.preBuildFunc=@peak_hold_example_prebuild;
M.setFunc=@peak_hold_example_set;
```

Next an input pin is added. This pin accepts floating-point data and places no restrictions on the block size or sample rate.

```
PTIN=new_pin_type;
add_pin(M, 'input', 'in', 'audio input', PTIN);
```

Several top-level parameters are added. These parameters set the attack and decay times (in milliseconds) and also specify a variable which allows the peak values to be reset.

```
add_variable(M, 'Reset', 'int', 0, 'parameter', 'reset the current peak
values');
M.Reset.range=[0 1];

add_variable(M, 'attackTime', 'float', 0.01, 'parameter', 'Envelope
detector attack time constant');
M.attackTime.units='msec';
M.attackTime.range=[0 100];

add_variable(M, 'decayTime', 'float', 0.01, 'parameter', 'Envelope
detector decay time constant');
M.attackTime.units='msec';
M.attackTime.range=[0 100];
```

Add two additional hidden variables which are the attack and decay coefficients. These coefficients are computed based on the sampling rate and block size and the calculation occurs within the set function. Both variables are marked as being hidden.

```
add_variable(M, 'attackCoef', 'float', 0, 'derived', 'Computed
coefficient used for attack');
M.attackCoef.isHidden=1;

add_variable(M, 'decayCoef', 'float', 0, 'derived', 'Computed
coefficient used for decay');
M.attackCoef.isHidden=1;
```

The next part is key to this example. We add two arrays whose size depends upon the number of channels in the input signal. The code is:

```
add_array(M, 'peakHold', 'float', [], 'state', 'Array of peak values');
M.peakHold.arrayHeap='AWE_HEAP_FAST2SLOW';
M.peakHold.arraySizeConstructor='ClassWire_GetChannelCount(pWires[0]) *
sizeof(float)';

add_array(M, 'peakDecay', 'float', [], 'state', 'Array of decaying peak
values');
M.peakDecay.arrayHeap='AWE_HEAP_FAST2SLOW';
M.peakDecay.arraySizeConstructor='ClassWire_GetChannelCount(pWires[0]) *
sizeof(float)';
```

At this point, the array is set to the empty matrix; its size is based on the number of input channels which is currently not known. The array size is set below in the module's prebuild function.

Two additional fields are set for each array variable. These variables provide information enabling the C constructor function to be automatically generated. For each array, a call to the framework memory allocation function is made

```
void *awe_fwMalloc(size_t size, UINT heapIndex, int *retVal);
```

The first argument is the number of words to allocate (the size) and the second argument specifies which memory heap to allocate the memory from. In the MATLAB code, set

```
M.peakHold.arraySizeConstructor='ClassWire_GetChannelCount(pWires[0]) *
sizeof(float)';
```

This string becomes the first argument to awe_fwMalloc(). The variable pWires is an argument to the constructor function and points to an array of wire pointers. The code contained in the string determines the number of channels in the first input wire and multiplies by sizeof(float). The net result is the amount of memory to allocate for the array.

The second argument to the awe_fwMalloc function is specified by the MATLAB code

```
M.peakHold.arrayHeap='AWE_HEAP_FAST2SLOW';
```

This value is #define'd within Framework.h and specifies that memory should first be taken from the fast internal heap, and if this heap is full, is should then be taken from the slow heap.

Together, the .arraySizeConstructor and .arrayHeap settings get turned into the following code by the code generator:

```
if ((S->peakHold = (float *)
awe_fwMalloc(ClassWire_GetChannelCount(pWires[0]) * sizeof(float),
AWE_HEAP_FAST2SLOW, retVal)) == 0)
    {
        // Error code is in *retVal
        return 0;
    }
```

This process repeats for the second array .peakDecay.

Thus far, the code pieces within MATLAB generate the C constructor function. MATLAB also needs to know the sizes of the arrays for tuning and in order to draw the user interface. The arrays sizes are set in the MATLAB prebuild function. This is also located in the file peak_hold_example_module.m:

```
function M=peak_hold_example_prebuild(M)

numChannels=M.inputPin{1}.type.numChannels;

M.peakDecay.size = [numChannels 1];
M.peakDecay = zeros(numChannels, 1);

M.peakHold.size = [numChannels 1];
M.peakHold = zeros(numChannels, 1);

return
```

For each array, we first set the size to [numChannels 1] and then set the default value to an array of all zeros of the same size. Note that the order of operations is important. When doing array assignments, Audio Weaver checks that the assigned data is of the appropriate size. If the code were instead

```
M.peakDecay = zeros(numChannels, 1);
M.peakDecay.size = [numChannels 1];
```

Then the first assignment of zeros() would fail since the size of the array has not yet been set.

The rest of the function is fairly straightforward. We specify the C processing and set functions using code markers

```
awe_addcodemarker(M, 'processFunction',
'Insert:code\InnerPeakHoldExample_Process.c');
awe_addcodemarker(M, 'setFunction',
'Insert:code\InnerPeakHoldExample_Set.c');
```

Documentation is then added (not shown here). A line is added to the generated C file which includes the file FilterDesign.h. This file contains some of the design equations used by the C set function.

```
awe_addcodemarker(M, 'srcFileInclude', '#include "FilterDesign.h"');
```

There are both MATLAB and C versions of the set function. The functions translate the high-level attackTime and decayTime variables to the low-level attackCoeff and decayCoeff variables. The MATLAB code is

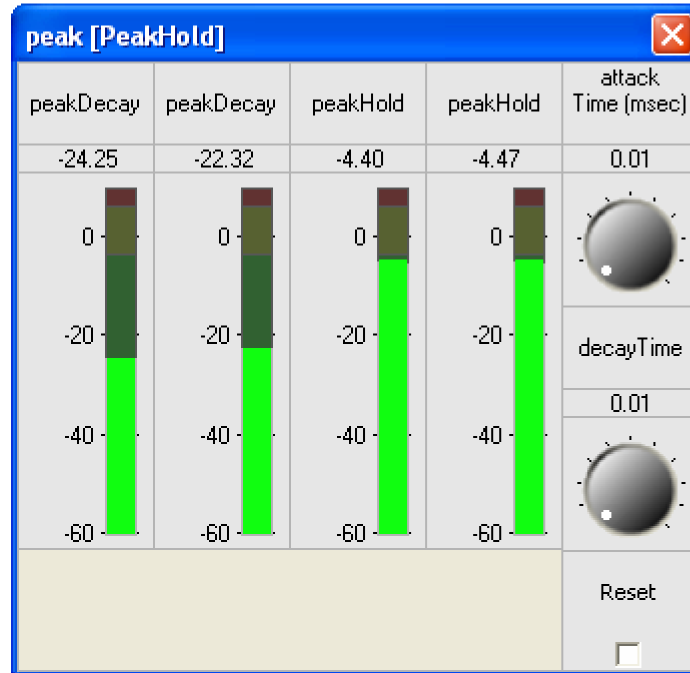
```
function M=peak_hold_example_set(M)

SR=M.inputPin{1}.type.sampleRate;

M.attackCoef=design_smoother(M.attackTime, SR, 1);
M.decayCoef=design_smoother(M.decayTime, SR, 1);


return;
```

The C code contained in InnerPeakHoldExample_Set.c is:

```
awe_modPeakHoldExampleInstance *S;
WireInstance **pWires;
float SR;
S = (awe_modPeakHoldExampleInstance *)pInstance;
pWires = ClassModule_GetWires(S);
SR = (float) ClassWire_GetSampleRate(pWires[0]);

S->attackCoef = design_smoother(S->attackTime, SR, 1);
S->decayCoef= design_smoother(S->decayTime, SR, 1);

return(0);
```

Why are two versions of the set function required? For this module, the MATLAB function is not required. After a module is allocated on the target in C, the C set function is automatically called and the coefficients computed. The MATLAB code is for convenience and can also be used for regression testing.

The last portion of the peak_hold_example_module.m file contains the inspector information. We only show the .peakHold variable since the others are similar or have already been discussed. The inspector drawn is:



This example has a stereo input. The first two meters shown are .peakDecay for both channels. The second two meters show .peakHold. The MATLAB code for drawing the .peakDecay meters is:

```
M.peakDecay.guiInfo.transposeArray=1;
M.peakDecay.guiInfo.range=[-60 10];
M.peakDecay.guiInfo.controlType='meter';
M.peakDecay.guiInfo.attribStr='mapping=db20';
add_control(M, '.peakDecay', 'topRight');
```

The .peakDecay variable is defined in MATLAB as a column vector. In this case, since there are two input channels, the vector is of dimension 2x1. When the meter control is drawn, the default behavior is to draw it as 2 high and 1 wide (this matches the MATLAB variable size)

This is not what we want – we want the controls to be side by side. To achieve this, the array of values has to be *transposed* to form a row vector. The statement

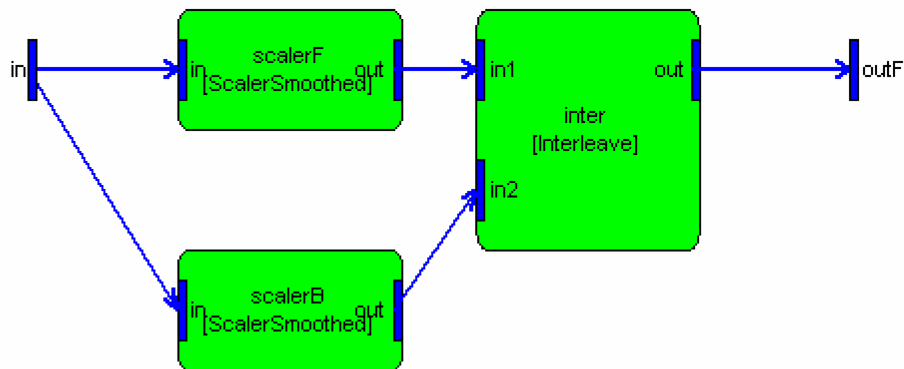
```
M.peakDecay.guiInfo.transposeArray=1;
```

achieves this. Next, the range of the control is set, the variable is tied to a meter control, and finally, the control is instructed to convert from linear measurements to dB for display. This completes the MATLAB code for the .peakDecay inspector control. The .peakHold inspector specification is almost identical.

The C processing function is straightforward and is not shown here. Refer to the file InnerPeakHoldExample_Process.c.

7.3. Fader module

This example teaches how to create a new module class out of a subsystem and shows how to implement a custom bypass function. The fader module has a single mono input pin, a stereo output pin, and 3 internal modules:



The input signal is scaled by scalerF to yield the front channel and by scalerB to yield the back channel. The output pin is stereo with the first channel representing the front and the second channel the back. The scalers are set so that a portion of the input signal is fed to each of the front and back channels. The fader implements constant energy panning using a sine/cosine response.

The fader example is found in fader_example_module.m. As usual, create a subsystem and set its .name based on an argument to the function.

```
SYS=awe_subsystem('FaderExample', 'Splits a mono input signal into front
      and back channels with constant energy panning.');
SYS.name=NAME;
```

Add input and output pins. The input pin is a single channel without any restrictions on the blockSize or sampleRate. The output pin is stereo without any restrictions on the blockSize or sampleRate.

```
PTIN=new_pin_type(1);
PTOUT=new_pin_type(2);

add_pin(SYS, 'input', 'in', 'audio input', PTIN);
add_pin(SYS, 'output', 'out', 'audio output', PTOUT);
```

Then add the modules to create the subsystem

```
add_module(SYS, scaler_smoothed_module('scalerF'));
add_module(SYS, scaler_smoothed_module('scalerB'));
add_module(SYS, interleave_module('inter', 2));
```

Then connect them together:

```
connect(SYS, '', 'scalerF');
connect(SYS, 'scalerF', 'inter.in1');
connect(SYS, '', 'scalerB');
connect(SYS, 'scalerB', 'inter.in2');
```

```
connect(SYS, 'inter', '');
```

None of the bypass functions provided with Audio Weaver is suitable for this module. The desired bypass behavior is to copy the input channel to each of the two output channels and scale by 0.707. To implement this custom bypass functionality, write an inner bypass function as shown below:

```
awe_modFaderExampleInstance *S = (awe_modFaderExampleInstance
*)pInstance;
WireInstance **pWires = ClassModule_GetWires(S);
float *in = (float *) pWires[0]->buffer;
float *outF = (float *) pWires[1]->buffer;
float *outB = (float *) outF + 1;
UINT numSamples = ClassWire_GetNumSamples(pWires[0]);

 awe_vecScale((float *)(pWires[0]->buffer), 1,
               (float *)(pWires[1]->buffer), 2,
                     (float)0.707, numSamples);

 awe_vecScale((float *)(pWires[0]->buffer), 1,
                     (float *)(outB), 2,
                     (float)0.707, numSamples);
```

Then set the 'bypassFunction' code marker to insert this file:

```
awe_addcodemarker(SYS, 'bypassFunction',
'Insert:code\InnerFaderExample_Bypass.c');
```

The key lesson this example is the MATLAB code

```
SYS.flattenOnBuild=0;
```

This identifies the subsystem as one that is not flattened when building. That is, the FaderExample module class exists natively on the target. The Server only sends a single command to the target and the entire subsystem will be instantiated. The automatically generated constructor function is found in ModFaderExample.c and shown below. The constructor function allocates the base module and then allocates each of the 3 internal modules. The module is marked as requiring a total of 4 pins: 1 input, 1 output, and 2 scratch. These pins are allocated outside of the function and then passed in. By looking through the code you can see how the wire information is passed to each of the internal modules' constructor function.

```
AWE_MOD_SLOW_CODE
ModInstanceDescriptor *awe_modFaderExampleConstructor(int * FW_RESTRICT
retVal, UINT nIO, WireInstance ** FW_RESTRICT pWires, size_t argCount,
const Sample * FW_RESTRICT args)
  {
    // Create the module which holds the overall subsystem
    awe_modFaderExampleInstance *S = (awe_modFaderExampleInstance *)
BaseClassModule_Constructor(&awe_modFaderExampleClass, retVal,

ClassModule_PackFlags(1, 1, 2),
                                                               pWires,
argCount, args);
```

```
    {
      WireInstance *scalerFWires[2]={ pWires[0], pWires[2] };
      Sample scalerFArgs[4];
      scalerFArgs[0].fVal = 7.0710677e-001f; // scalerF.gain
      scalerFArgs[1].fVal = 1.0000000e+001f; // scalerF.smoothingTime
      scalerFArgs[2].fVal = 7.0710677e-001f; // scalerF.currentGain
      scalerFArgs[3].fVal = 2.0811646e-003f; // scalerF.smoothingCoeff
      S->scalerF = (awe_modScalerSmoothedInstance *)
ClassModule_Constructor(CLASSID_SCALERSMOOTHED, retVal,
ClassModule_PackFlags(1, 1, 0), scalerFWires, 4, (Sample *)scalerFArgs);
      if (S->scalerF == NULL)
      {
        // Error code is in *retVal
        return 0;
      }
    }
    {
      WireInstance *scalerBWires[2]={ pWires[0], pWires[3] };
      Sample scalerBArgs[4];
      scalerBArgs[0].fVal = 7.0710677e-001f; // scalerB.gain
      scalerBArgs[1].fVal = 1.0000000e+001f; // scalerB.smoothingTime
      scalerBArgs[2].fVal = 7.0710677e-001f; // scalerB.currentGain
      scalerBArgs[3].fVal = 2.0811646e-003f; // scalerB.smoothingCoeff
      S->scalerB = (awe_modScalerSmoothedInstance *)
ClassModule_Constructor(CLASSID_SCALERSMOOTHED, retVal,
ClassModule_PackFlags(1, 1, 0), scalerBWires, 4, (Sample *)scalerBArgs);
      if (S->scalerB == NULL)
      {
        // Error code is in *retVal
        return 0;
      }
    }
    {
      WireInstance *interWires[3]={ pWires[2], pWires[3], pWires[1] };
      S->inter = (awe_modInterleaveInstance *)
ClassModule_Constructor(CLASSID_INTERLEAVE, retVal,
ClassModule_PackFlags(2, 1, 0), interWires, 0, (Sample *)NULL);
      if (S->inter == NULL)
      {
        // Error code is in *retVal
        return 0;
      }
    }

    return ((ModInstanceDescriptor *) S);
  }
```

Similarly, since this module is created out of a subsystem, there is no need to provide a processing function; it will be automatically generated by Audio Weaver. (Yes, a custom processing function could be supplied, but the generated one works fine.) The automatically generated processing function is shown in ModFaderExample.c. It simply calls the 3 processing functions for the 3 internal modules:

```
AWE_MOD_FAST_CODE
void awe_modFaderExampleProcess(void *pInstance)
{
  awe_modFaderExampleInstance *S = (awe_modFaderExampleInstance *)
```

```
pInstance;

    ClassModule_Execute(S->scalerF);
    ClassModule_Execute(S->scalerB);
    ClassModule_Execute(S->inter);
}
```

The set function for the subsystem takes the high-level .smoothingTime variable and passes it down to each of the internal smoothed scaler modules. The function also takes the .fade variable and computes two separate gains using a sin/cos pan.

```
  AWE_MOD_SLOW_CODE
  UINT awe_modFaderSet(void *pInstance, UINT mask)
  {
    awe_modFaderInstance *S;
    float arg;

    S = (awe_modFaderInstance *)pInstance;

    S->scalerF->smoothingTime=S->smoothingTime;
    S->scalerB->smoothingTime=S->smoothingTime;

    // Compute the front and back gains from the .fade setting.
    arg = (float)((1+S->fade)*PIDIV4);
    S->scalerF->gain = sinf(arg);
    S->scalerB->gain = cosf(arg);

    ClassModule_Set(S->scalerF, MASK_ScalerSmoothed_smoothingTime |
MASK_ScalerSmoothed_gain);
    ClassModule_Set(S->scalerB, MASK_ScalerSmoothed_smoothingTime |
MASK_ScalerSmoothed_gain);

    return(0);
  }
```

The function sets the high-level variables of the smoothed scaler modules and then calls the set function for the module. The set function takes a pointer to the smoothed scaler module and also a bit mask which identifies the specific variables that were modified. In this case, we are setting the smoothingTime and gain fields of the modules.

There is also a fract32 version of this module called fader_example_fract32_module.m See this module for examples of fixed-point processing.

7.4. Downsampler Module

This module downsamples a signal by keeping only one out of every D input samples. This example demonstrates two further techniques used when creating audio modules. First, the module essentially copies 32-bit samples from input to output and doesn't care if the data type is float, fract32, or int. All of these types will work with the same processing function. This is obvious in C but this flexibility must be encoded into the MATLAB scripts. The second technique shown is how to change the block size and sample rate as part of the prebuild function.

This example is contained in the file downsampler_example_module.m and is equivalent to downsampler_module.m found in the Core32 module library.

The MATLAB code defining the input and output pins of the module is:

```
PT=new_pin_type([], [], [], '*32');
add_pin(M, 'input', 'in', 'Input signal', PT);
add_pin(M, 'output', 'out', 'Output signal', PT);
```

In this case, the data type is specified as '*32'. This abbreviation is expanded to all possible 32-bit data types in the add_pin function: {'float', 'fract32', 'int'}. It is possible to specify the data type to the new_pin_type call as the cell array

```
{'float', 'fract32', 'int'}
```

Using the '*32' abbreviation gives us flexibility in case the set of data types is ever expanded in the future.

The decimation factor D is the second argument to the MATLAB function and is specified at instantiation time. The variable D appears in the instance data structure and its usage is listed as 'const'. This means that the variable D can no longer be changed.

```
add_variable(M, 'D', 'int', D, 'const', ...
      'Decimation factor.  1 out of every D samples is output');
```

The module's prebuild function computes the type of the output pin. The block size and sample rate are reduced by the decimation factor D; the number of channels and data type are unchanged. We also verify that the block size of the input is divisible by the decimation factor:

```
blockSize=M.inputPin{1}.type.blockSize;

if (rem(blockSize, M.D) ~= 0)
      warning('The downsample factor D does not evenly divide the
blockSize');
      newBlockSize=floor(blockSize/M.D);
else
      newBlockSize=blockSize/M.D;
end

M.outputPin{1}.type.sampleRate=M.inputPin{1}.type.sampleRate/M.D;
M.outputPin{1}.type.numChannels=M.inputPin{1}.type.numChannels;
M.outputPin{1}.type.dataType=M.inputPin{1}.type.dataType;
M.outputPin{1}.type.blockSize=newBlockSize;
```

The module's MATLAB implementation of the processing function is found in downsampler_example_process.m. It simply returns 1 out of every D input samples.

```
function [M, WIRE_OUT]=downsampler_process(M, WIRE_IN)

IN=WIRE_IN{1};
```

```
WIRE_OUT{1}=IN(1:M.D:end, :);

return;
```

The C processing function is similar. Although written for 32-bit floats, the function works equally well with any 32-bit data type.

```
AWE_MOD_FAST_CODE
void awe_modDownsamplerExampleProcess(void *pInstance)
{
  awe_modDownsamplerInstance *S = (awe_modDownsamplerInstance *)
pInstance;
  WireInstance **pWires = ClassModule_GetWires(S);
  UINT blockSize = ClassWire_GetBlockSize(pWires[0]);
  UINT numChannels = ClassWire_GetChannelCount(pWires[0]);
  UINT channel;
  int blockSizeD = (blockSize / S->D);

  // blockSize is the number of samples in each input channel
  // blockSizeD is the number of samples in each output channel

  float *src = (float *)pWires[0]->buffer;
  float *dst = (float *)pWires[1]->buffer;

  for(channel=0;channel<numChannels;channel++)
    {
      awe_vecCopy(src+channel, (numChannels*S->D), dst+channel,
numChannels, blockSizeD);
    }
}
```

7.5. Look Ahead Limiter Module

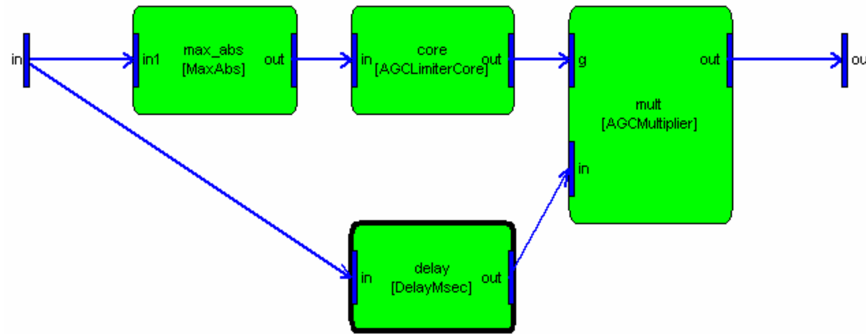
This example demonstrates some advanced code generation techniques. The look ahead limiter is a dynamics processor which applies a time-varying gain to a signal. The module m-file takes two arguments.

```
SYS=lah_limiter_example_module(NAME, MAXDELAY)
```

The first is the NAME of the subsystem and the second, MAXDELAY, is the maximum amount of delay, in milliseconds. MAXDELAY is used internally when allocating the delaymsec_module.m:

```
add_module(SYS, delaymsec_module('delay', MAXDELAY));
SYS.delay.currentDelayTime=MAXDELAY;
```

The compiled subsystem contains 4 internal modules.



7.5.1. Specifying Internal Module Constructor Arguments

If we follow the normal code generation procedure, then the constructor function for the internal delaymsec_module will be:

```
{
  WireInstance *delayWires[2]={ pWires[0], pWires[5] };
  Sample delayArgs[2];
  delayArgs[0].fVal = 5.0000000e+000f; // delay.maxDelayTime
  delayArgs[1].fVal = 5.0000000e+000f; // delay.currentDelayTime
  S->delay = (awe_modDelayMsecInstance *) ClassModule_Constructor(CLASSID_DELAYMSEC,
retVal, ClassModule_PackFlags(1, 1, 0), delayWires, 2, (Sample *)delayArgs);
  if (S->delay == NULL)
        {
          // Error code is in *retVal
          return 0;
        }
}
```

If you look at the highlighted code, you'll see that the maximum delay time of the internal module is hard coded to 5 msec. This is not what we wanted; the delay time should instead be specified at construction time.

To fix this, add a high-level variable to the look ahead limiter system. We'll name it 'maxDelayTime' to be consistent with the delaymsec_module.m.

```
add_variable(SYS, 'maxDelayTime', 'float', MAXDELAY, 'const', ...
       'Maximum delay time');
SYS.maxDelayTime.units='msec';
```

Next, we'll use the maxDelayTime field of the look ahead limiter instance structure as an initializer for the maxDelayTime and currentDelayTime fields of the internal module. This is done by setting the .constructorCode field of the delaymsec_module variables as shown below:

```
SYS.delay.maxDelayTime.constructorCode='S->maxDelayTime';
SYS.delay.currentDelayTime.constructorCode='S->maxDelayTime';
```

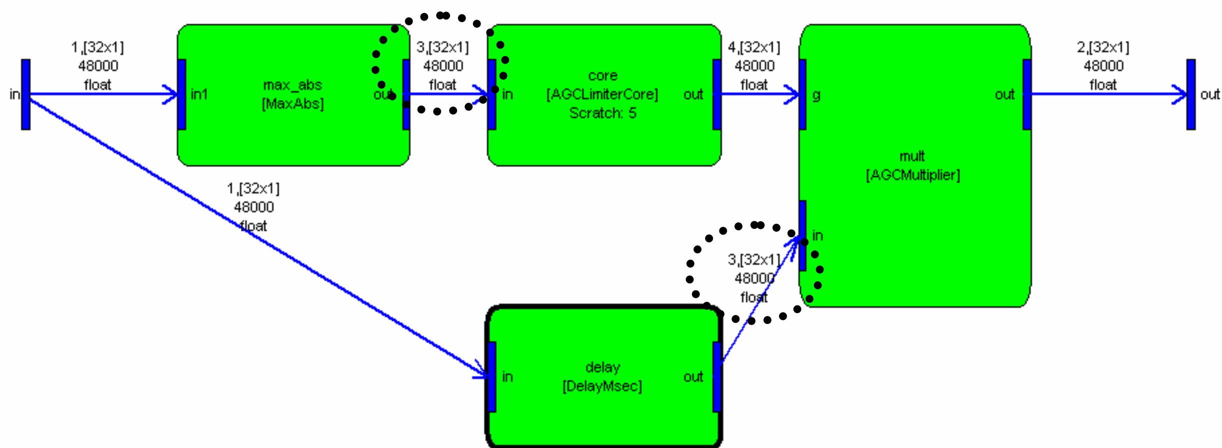
In the automatically generated constructor code, S always refers to the base module structure which has already been allocated and initialized by BaseClassModule_Constructor(). The new constructor code for the delay is:

```
{
  WireInstance *delayWires[2]={ pWires[0], pWires[5] };
  Sample delayArgs[2];
  delayArgs[0].fVal = S->maxDelayTime; // delay.maxDelayTime
  delayArgs[1].fVal = S->maxDelayTime; // delay.currentDelayTime
  S->delay = (awe_modDelayMsecInstance *)
ClassModule_Constructor(CLASSID_DELAYMSEC, retVal,
ClassModule_PackFlags(1, 1, 0), delayWires, 2, (Sample *)delayArgs);
  if (S->delay == NULL)
      {
        // Error code is in *retVal
        return 0;
      }
}
```

The argument S->maxDelayTime initializes the internal module achieving the desired effect.

7.5.2. Disambiguating Wire Allocation

We if look at the default subsystem including wiring information, we see:



By default, the input and output pins are mono and the system ends up using 3 scratch wires:

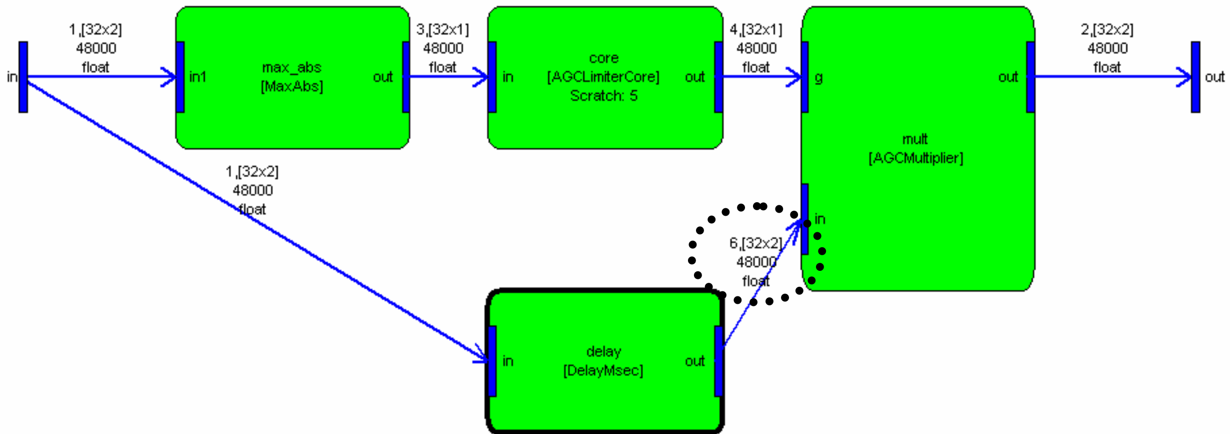
```
>> SYS=lah_limiter_example_module('LAH');
>> SYS=prebuild(SYS);
>> SYS.scratchPin

ans =

    [1x1 struct]    [1x1 struct]    [1x1 struct]
```

This subsystem is designed to operate on an arbitrary number of channels. However, the wire allocation shown above is only applicable when the input is mono. (To see this, consider the MaxAbs module. This module always has a mono output and is assigned to wire #3. The signal path through DelayMsec can be N-channel and is also assigned to wire #3). If this system is run with stereo inputs, then the wire allocation is incorrect and the audio processing will fail.

To fix this problem, force the system input pin to have 2 channels by default.

```
pinType=SYS.max_abs.inputPin{1}.type;
add_pin(SYS, 'input', 'in', 'Audio Input', pinType);
SYS.inputPin{1}.type.numChannels=2;
```

The pin still holds an arbitrary number of channels, but its default value is 2. When the subsystem is now routed, the wires will be allocated as:



The system now requires a total of 4 scratch wires, with a new scratch wire, #6 assigned to the output of DelayMsec. Furthermore, the buffer allocation is now general purpose and will work with any number of input channels.

8. Building Audio Module DLLs

The final phase in developing an audio module library is to compile the source files and create a Dynamic Link Library (DLL) for the Server. Typically, a DLL contains multiple modules organized around a common theme. For example, Audio Weaver provides several module packs:

BasicAudioFloat32.dll – Basic audio functions that operate on 32-bit floating-point values.

MathFloat32.dll – Audio modules based on the C floating-point math library.

BasicAudioFract32.dll – Basic audio functions that operate on 32-bit fractional values.

After a DLL is built, it is copied into the directory containing the Server executable

```
<AWE>\Bin
```

The Server then loads the DLL next time it launches. The audio module DLL contains all of the information needed by the Server to instantiate, execute, and control the module. The DLL also contains the schema information facilitating real-time tuning.

This section describes how to generate an audio module DLL by working through the Examples module library discussed in Section 7. Building an audio module DLL requires VisualStudio version 7 (2003). If you are running custom audio modules on an embedded processor, you'll need to build an audio module DLL and also rebuild the target executable. This section focuses on building the DLL. Refer to the *Audio Weaver Platform Developers Guide* for the additional steps needed on the target.

The steps described below indicate one method of structuring files and scripts to generate the module DLL. You can adapt this to fit your development process.

8.1. Building ExamplesDLL.dll

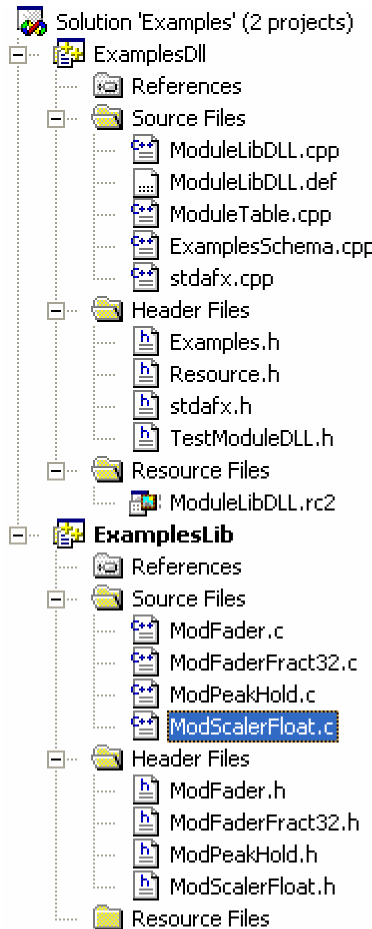
Start by running the master code generation script for this library

```
make_examples
```

Next, open up the VisualStudio solution

```
<AWE>\ModuleLibs\Examples\Examples.sln
```

The solution contains two projects shown below



The ExamplesLib compiles the module source files and builds an overall library. The ExamplesDLL project builds the DLL. ExamplesLib is easily understood by C programmers. Understanding ExamplesDLL requires some Windows expertise. Fortunately, you can use the Examples projects as starting points for your own custom DLL development and do not have to delve into the underlying Windows issues.

To build the audio module DLL, right-click on ExamplesDll and select Rebuild. The project dependencies are set so that first ExamplesLib will be built followed by ExampleDll.

ExamplesDLL also contains a post-build event which automatically copies the DLL into the <AWE>\Bin directory so that it can be referenced by the Server.

Command Line	**copy Debug*.dll ..\..\..\Bin**
Description	
Excluded From Build	No

If the AWE_Server.exe executable is not in the expected place, then change the destination directory in this post-build step.

If the DLL was properly built, you'll be able to open the Server and see the new library of examples listed. Select Target→List Modules and scroll down the list looking for the ExamplesDLL.dll entry:

```
From ExamplesDLL.dll:
32768    ModuleScalerSmoothedExample
32769    ModuleScalerExample
32770    ModuleFader
32771    ModuleFaderFract32
32772    ModulePeakHold
32773    ModuleDownsamplerExample
32774    ModuleLAHLimiter
32775    ModulePeakHoldFract32
```

8.2. Cloning the Examples Library

We recommend that you use the Examples directory as a starting point for your own custom module libraries. We also recommend maintaining the existing directory structure. Begin by copying and renaming the entire Examples directory in <AWE>/<ModuleLibs>. We'll assume that the new module library will be called MyModules and will be stored in the directory <MYMODULES>.

Then, do the following steps:

1. Delete all generated files, they will be regenerated:

```
<MYMODULES>/Examples.h
<MYMODULES>/ExamplesSchema.cpp
<MYMODULES>/ExamplesSchema.sch
<MYMODULES>/Examples.ncb, if exists
<MYMODULES>/Examples.suo, if exists
<MYMODULES>/Source/*
<MYMODULES>/Include/*
<MYMODULES>/Doc/*
```

2. Also delete all files in <MYMODULES>/matlab except make_examples.m

3. Rename the following files from "Example" to "MyModules". You'll end up with:

```
<MYMODULES>/BuildDll/MyModulesDLL.vcproj
<MYMODULES>/BuildLib/MyModulesLib.vcproj
<MYMODULES>/matlab/make_mymodules.m
<MYMODULES>/MyModules.sln
```

4. Clear out any module m-files in

```
<MYMODULES>\matlab
```

   folder and any code pieces in

```
<MYMODULES>\matlab\code
```

```
<MYMODULES>\matlab\process
<MYMODULES>\matlab\test
```

Copy in the files for your custom modules.

5. Edit make_mymodules.m and replace all the occurrences of “Examples” to “MyModules”. Also specify the list of modules you would like included in the library.

6. Edit the class ID file

```
<MYMODULES>\classids.csv
```

and specify unique class IDs for the modules in your library.

7. Edit awe_init.m and update the audio module path.

```
if (exist([pathstr, '\ModuleLibs\MyModules']))
   add_module_path(fullfile(pathstr, 'ModuleLibs', 'MyModules'),
'-end', REMOVE);
end
```

8. Issue the command “awe_init” in the MATLAB command window.

9. Issue the command “make_mymodules” in the MATLAB command window.

10. Edit the *.vcproj files in a text editor and delete the ProjectGUID string. This string is *supposed* to be unique and thus uniquely identify the project within VisualStudio. However, since we just copied and renamed the .vcproj file, the ProjectGUID will no longer be unique. Find the line with the ProjectGUID keyword near the top each .vcproj file:

```
ProjectGUID="{5FEC38A8-50BF-41E6-A94F-E489381E2B85}"
```

The line will be similar to this, but not exact. Change it to:

```
ProjectGUID=""
```

This forces VisualStudio to update the GUID next time the file is opened. Change all occurrences of"Examples" to "MyModules" throughout the file.

11. Edit the file

```
<MYMODULES>\BuildDll\ModuleTable.cpp
```

and instead of including "Examples.h", change it to "MyModules.h".

12. Open up the file

```
<MYMODULES>\BuildDll\ModuleLibDLL.def
```

and change

```
LIBRARY             "ExamplesDLL"
```

to

```
LIBRARY             "MyModulesDLL"
```

13. Open the MyModules.sln in any editor window and look for “Examples” and replace this with “MyModules”. If doesn’t work, delete MyModules.sln and create a new solution. Then add the two projects, “MyModulesDLL.vcproj” and “MyModulesLib.vcproj”, to the solution.

14. Add the generated source files from <MYMODULES>\Source\ to the Source Files of “MyModulesLib.vcproj” and add files from <MYMODULES>\Include to the Include Files of “MyModulesLib.vcproj”.

At this point, the projects are ready to build. Build the solution, and when complete, you'll have MyModulesDLL.dll in the <AWE>\Bin directory.

9. VectorLib and ModuleHelperLib

These libraries contain a collection of useful functions for developing custom audio modules. In fact, much of the Audio Weaver module library is built upon these functions. At a minimum, each function is implemented in C to provide functionality on the PC and baseline performance on embedded processors. In some cases, functions have been optimized in assembly for specific processors.

The functions provided fall into 4 broad categories

Vector library – basic signal processing operations that operate on blocks of data. There are separate floating-point and fractional functions. See VectorLib.h.

Fractional math library – code which emulates fractional arithmetic on the PC. This is used to first develop Blackfin modules on the PC. See FractMath.h.

Filter design library – filter design and coefficient calculation routines for basic filter types. In floating-point or double precision. See ModuleHelperLib.h

Math helper library – floating-point versions of the math library. Used on the PC. See MathHelper.h.

The audio module library can be considered one level higher in abstraction compared to the vector library. There is some loss of efficiency since it introduces another function call. However, there is a significant benefit since a single vector function may be used by multiple modules. This allows the several module processing functions to be written in C and only requires a single vector function to be written in assembly. Furthermore, adding support for other processor families in Audio Weaver is then primarily a matter of optimizing the vector library and not rewriting a large number of individual module functions.

Refer to the individual header files for a description of each function and its arguments.

##if
  preprocessor directive, 17, 86
$IDENTIFIER$, 84
@awe_variable, **30**
across
  wire allocation, 103
  wiring constraint, 82
add_array, **51**
add_code_marker.m, 84
add_module.m, **52**
add_module_path.m, 80
add_pin.m, **49**
add_variable.m, 10, **50**
agc_example.m, 31
array
  overriding sizes, 51
arrayHeap, 33, 90, 106
arraySizeConstructor, 33, 90, 106
audio module
  class name, 35
  class structure, 12
  dependencies, 15
  description, 35
  functions, 12
  internal functions, 39
  internal representation, 36
  search path, 80
  type definition, 10
  variable masks, 11
  variable offsets, 11
Audio module
  instance structure, 10
audio module m-file, 41
AudioWeaverModule
  tag for documentation, 83
awe_addcodemarker.m, 16
awe_deletecodemarker.m, 86
awe_fwMalloc, 68, 105
awe_generate_doc.m, 94
awe_generate_library.m, 15, 76
awe_generate_module.m, 14, 92
AWE_HEAP_FAST, 69
AWE_HEAP_FAST2SLOW, 69
AWE_HEAP_FASTB, 69
AWE_HEAP_SLOW, 69
awe_help.m, 54, **93**
AWE_INFO.buildControl, 79
AWE_INFO.displayControl.showHidden, 34
awe_lookupcodemarker.m, 85
awe_module.m, **35**
awe_module_template.c, 16, 83
awe_module_template.h, 16, 83
awe_reorder_variables.m, 81
awe_subsystem.m, **43**
BaseClassModule_Constructor, 71
buildFunc
  awe_subsystem field, 44
Bypass
  audio module function, 12
bypass function
  example of custom, 111
bypassFunc, 42
  audio module field, 38
bypassFunction
  code marker, 87
BypassFunction
  C function, 74
bypassFunctionName
  code marker, 87
class
  audio module field, 39
class name
  audio module, 35
class structure
  audio module, 12
CLASS_ID_BASE, 14
classID, 11, 13
  audio module field, 36
classid_lookup.m, 81
classids.csv, 26, 80
ClassMod_GetNPrivateArgs, 65
ClassMod_GetNPublicArgs, 65
ClassModule_GetModuleState, 60, 72
ClassModule_GetNInWires, 60, 72
ClassModule_GetNOutWires, 60, 72

ClassModule_GetNScratchWires, 60, 72
ClassModule_GetWires, 60, 72
ClassModule_PackArgCounts, 71
className
  audio module field, 36
ClassWire_GetBlockSize, 61, 72
ClassWire_GetChannelCount, 61, 72
ClassWire_GetNumSamples, 61, 73
ClassWire_GetSampleRate, 61, 73
clockDivider
  audio module field, 37
code markers, 17
Code Markers, 83
codeMarker
  audio module field, 38
connect.m, 45, 53
connection
  awe_subsystem field, 44
const
  usage, 32
Constructor
  audio module function, 12
  C function arguments, 69
constructorCode, 33, 90, 116
constructorFunction
  code marker, 87
data type
  variable field, 32
data types, 31
  in C, 59
dependencies
  in audio module libraries, 15
derived
  usage, 33
description field
  audio module, 35
  audio module field, 36
discussion
  code marker, 88
distinct
  wiring constraint, 82
DLL
  audio module library, 119
docControl, 95
docInfo
  audio module field, 39
Docmentation format
  selecting, 95
Documentation
  generating, 93
downsampler_example_module.m, 101, 114
drawFunc
  awe_subsystem field, 44
drawInfo
  audio module field, 39
Dynamic Link Libraries
  building, 119
equations
  in documentation, 99
Example, 101
Examples.h, 28
Examples.sln, 18, 119
ExamplesDLL.dll, 119
ExamplesSchema.cpp, 28
ExamplesSchema.sch, 26
fader_example_module.m, 101, 110
fieldNames
  audio module field, 39
  variable field, 34
flattenOnBuild, 90
  awe_subsystem field, 44
format
  variable field, 33
FractMath.h, 124
Framework.h, 58
GENDOC, 15, 77
Get
  audio module function, 12
  C function, 74
get_module.m, 45
get_variable.m, 31, 56
GetFeedbackPinCount, 70
getFunc, 42
  audio module field, 37
getFunction
  code marker, 88
GetInputPinCount, 70
GetOutputPinCount, 70

GetScratchPinCount, 70
GetWireCount, 70
guiInfo
audio module field, 38
variable field, 33
hasFired
audio module field, 38
hFileDefine
code marker, 88
hFileInclude
code marker, 88
hFileTemplate
code marker, 89
hierarchyName
audio module field, 38
variable field, 32
HTML
documentation format, 95
IDOFFSET, 13, 81
InnerScalerSmoothedExample_Process.c, 21
InnerScalerSmoothedExample_Set.c, 22
inputPin
audio module field, 37
Insert
code marker, 85
Insert:
code marker, 17
instance structure
audio module, 10
IOAcrossModule_Bypass, 74
IOAllOnesModule_Bypass, 75
IOMatchModule_Bypass, 74
IOMatchUpModule_Bypass, 75
isArray
variable field, 34
isComplex
variable field, 32
ISCOMPLEX, 31
isHidden
audio module field, 38
variable field, 33
ISHIDDEN, 31
isLive
variable field, 33
isLocked
audio module field, 39
variable field, 34
isPrebuilt
awe_subsystem field, 44
isPreset
audio module field, 38
variable field, 33
isTopLevel
audio module field, 38
isVolatile
variable field, 33
lah_limiter_example_module.m, 101, 115
LaTeX
equations, 99
make_examples.m, 15, 101
mask
audio module variables, 11
MASK_
audio module, 74
MathHelper.h., 124
MATLAB
supported versions, 6
memory allocation
from C, 68
memorySegment, 33
mfileDirectory
audio module field, 37
mfileName
audio module field, 37
mfilePath
audio module field, 37
ModClassDescriptor, 64
ModClassModule, 64
mode
audio module field, 37
ModScalerSmoothedExample.c, 23
ModScalerSmoothedExample.h, 22
module
awe_subsystem field, 44
module libraries
generating, 76
MODULE_ACTIVE, 72
MODULE_BYPASS, 72

MODULE_INACTIVE, 72
MODULE_MUTE, 72
ModuleHelperLib, 124
ModuleHelperLib.h, 124
ModuleInstanceDescriptor, 59
moduleName
  awe_subsystem field, 44
name
  audio module field, 36
  variable field, 32
new_pin_type.m, 46
nIO, 70
outputPin
  audio module field, 37
parameter
  usage, 33
peak_hold_example_module.m, 101, 104
pin
  data structure, 46
  documentation, 54
  new_pin_type.m, 46
  range information, 46
postConstructorFunction
  code marker, 88
postProcessFunc
  awe_subsystem field, 44
postProcessFunction
  code marker, 89
preBuildFunc, 43
  audio module field, 38
preProcessFunc
  awe_subsystem field, 44
preProcessFunction
  code marker, 89
preprocessor directives, 17
Process
  audio module function, 72
processFunc, 41
  audio module field, 38
processFunction
  code marker, 88
range
  variable field, 32
Requirements, 6
Sample, 60
  C data type, 60, 70
scaler_example_module.m, 101
scaler_smoothed_example_module.m, 9, 19
Schema file, 13
SchemaBuilder.exe, 79
scratchPin
  audio module field, 37
Set
  audio module function, 12
  C function, 73
set_variable.m, 56
setFunc, 42
  audio module field, 38
setFunction
  code marker, 88
showHidden
  AWE_INFO, 34
size
  variable field, 32
srcFileDefine
  code marker, 89
srcFileInclude
  code marker, 89
srcFileTemplate
  code marker, 89
state
  usage, 33
target_system.m, 46
targetInfo
  audio module field, 38
  variable field, 34
targetSpecificInfo
  awe_subsystem field, 44
template file, 16
template substitution, 16
  preprocessor, 86
Template Substitution, 83
top-level systems, 46
unique_classes.m, 90
units
  variable field, 33
unsatisfiable dependencies, 78
usage

variable field, 32
USESDLLS, 76, 77
variable
internal representation, 32
specifying range information, 46
variable field
audio module field, 37
variableName
audio module field, 37
VectorLib, 124
VectorLib.h, 124
VisualDSP
supported versions, 6
VisualStudio
supported version, 6
wire allocation
clarifying, 117
WIRE_IN, 41
wireAllocation
audio module field, 37
WireInstance, 60
Word
documentation format, 95