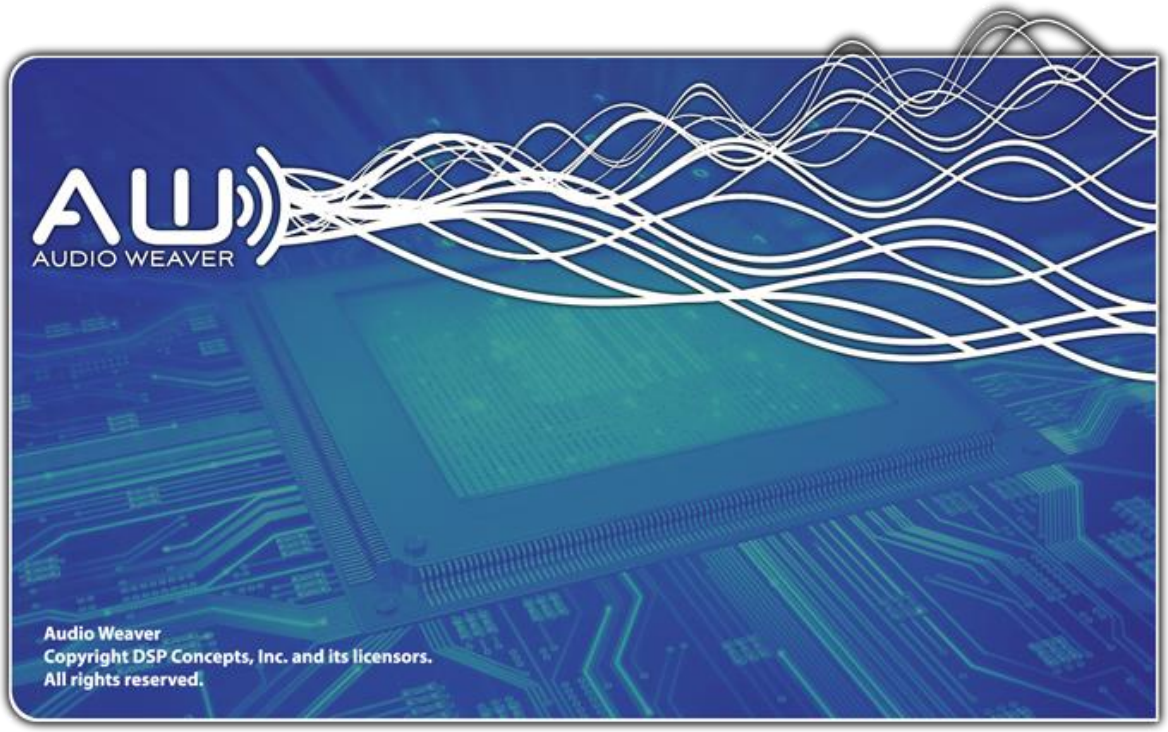
Audio Weaver

Tuning Command Syntax

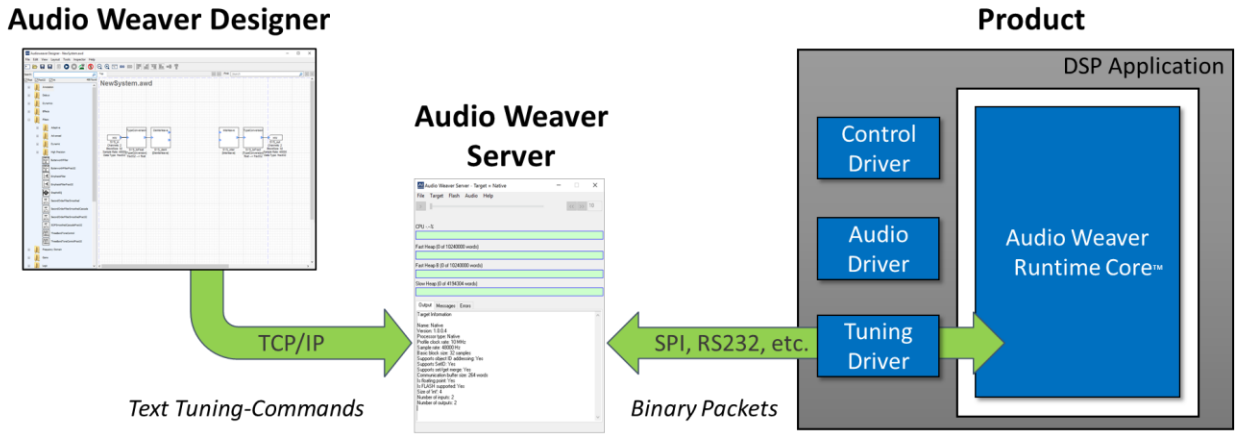


Novermber 2016

Table of Contents

1. Introduction

When using AWE Designer to create and tune a signal processing Layout, tuning commands (and the resulting replies) are exchanged between the PC and the AWE Core. The full path that these tuning commands travel is shown below.



These text-based commands may also be generated external tools and scripts. An example script, written in python to read and write a module in realtime to an active layout is shown below.

```
import socket
import time

TCP_IP = 'localhost'
TCP_PORT = 15001
BUFFER_SIZE = 1024

# Open a TCP socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((TCP_IP, TCP_PORT))

# Read the value of the scaler
s.send('get_value,ScalerDB1.gainDB\n')
data = s.recv(BUFFER_SIZE)
print data

# Attenuate channel 1
s.send('set_value,ScalerDB1.gainDB,-40\n')
data = s.recv(BUFFER_SIZE)
print data
```

This document describes the full set of tuning commands and arguments.

2. INI file settings

Colors may be customized in the INI file as follows:

[InspectorColors]

TileEdge – default 180,180,180, the color the edges of meter and slider controls are drawn.

DrawSides – default 1, when set boxes are drawn around meter and slider controls using TileEdge color

InspectorFace – default 230,230,230, the color dialog faces (other than the server dialog) are drawn

DropList – default 240,240,255, the color drop list backgrounds are drawn

TextColor – default 0,0,0, the color static text controls are drawn

By default none are specified in the INI file, so the given defaults are used.

3. Commands

All commands are sent by TCP/IP. The all commands are of the form:

```
command_key_word [, argN]
```

in other words, a CSV string. At need arguments may be arrays where a quoted CSV string is one of the arguments. Currently, no commands use this syntax. White space is not significant in commands unless within a string value.

The reply from all commands is either:

success [, args …]

or

failed, *reason*

The command keyword is not case sensitive. The commands are summarized as follows:

add_module	Adds one or more modules to a given layout.
add_symbol	Defines a new entry in the symbol table based on an in-memory address of an object.
add_symbol_id	Defines a new entry in the symbol table based on the unique ID of the object.
add_symbol_index	Defines a new entry in the symbol table based on its location (index) within the linked list of objects.
audio_pump	run the layout using either WMA/WAV/MP3 files, or audio line input as the source
audio_stop	stop the audio pump if running
bind_wire	binds a named wore to an input or output pin
compile	Compiles an AWS scrip file to AWB binary form
connect	initiates a connection from a client
create_active	Creates a control to display 4 radio buttons of module state
create_bitmap	creates an image control on a dialog
create_button	creates a button on a dialog
create_checkbox	creates a checkbox on a dialog
create_dialog	creates a named dialog
create_droplist	creates a drop list control on a dialog
create_awslist	creates a drop list control specifying AWS script files allowing the user to select and run presets from the list
create_edit	Creates an edit box control on a dialog

create_filelist	Creates a file list control on a dialog. Used for streaming files to the target.
create_graph	Creates a graph control that represents array elements as bar graphs
create_grid	Creates a grid control operating as as small spreadsheet for manipulating one or two dimensional arrays
create_layout	creates a layout with the specified properties
create_led	creates an LED style control on a dialog
create_lookup	creates an O(1) ID lookup table
create_meter	creates a meter control on a dialog
create_module	creates a module with the specified properties
create_pintype	creates a named pin with specified properties
create_slider	creates a slider or knob control on a dialog
create_spline	Creates an X-Y spline control on a dialog.
create_text	creates a static text control on a dialog
create_wire	creates a wire with specified properties
delete_file	Deletes a FLASH file
destroy	unconditionally destroys all created objects
destroy_dialog	destroys a dialog
dialog_state	Toggles between normal and expanded views
end_binary	Stops logging of binary commands sent to the target
erase_all	Erases all files in the target FLASH file system
exists_dialog	Checks if a dialog with a specified name exists
exit	causes the server to exit
fast_read	Reads arrays of binary data
fast_write	Writes arrays of binary data
file_logging	specifies whether to log commands and replies to a file
foreground	Brings all Audio Weaver windows to the foreground thus making them visible.
getini	Returns an entry from the AWE_Server.ini file
get_call	calls the get_call function of a module

get_executable_dir	Returns the directory containing the AWE_Server.exe executable
get_filesystem_info	Returns information about the target FLASH file system.
get_first_file	Returns the properties of the first FLASH file
get_first_io	returns the properties of the first I/O object
get_first_object	returns the properties of the first object instance
get_heap_count	returns the number of framework heaps
get_heap_size	returns the free space in the given heap
get_moduleclass_count	returns the number of known module classes
get_moduleclass_info	returns information about the specified class
get_module_state	returns the muted etc. state of the given module
get_next_file	Returns the properties of the next FLASH file, or fails if no more
get_next_io	returns the properties of the next I/O object, or fails of no more
get_next_object	returns the properties of the next object, or fails if no more objects
get_object_byaddress	returns the properties of the object at the given absolute address
get_object_byname	returns the properties of the named object
get_schema	returns the schema for a class
get_value	returns the value of a symbolically specified location
get_version	Returns the current version number of Audio Weaver
gui_logging	controls whether commands and replies are logged on the server control panel
make_binary	Starts logging of binary target commands to a named file
open_web_page	Launches a browser and displayed a specified page
pump	pump the entire framework
pump_layout	pumps the given layout once
pump_module	executes the pump function of the given module
query_pin	queries a named pin for its properties
read_file	Reads from a FLASH file
read_float_array	reads values from an array as floats
read_int_array	reads values from an array as integers

read_schema	reads a schema file adding its classes to those already loaded
read_wire	returns the contents of the specified wire buffer
script	executes a script file containing commands from this table
setini	writes a specified INI file entry with a value
set_call	calls the set_call function of a module
set_module_state	sets a given module to muted, activated, bypass, or disabled
set_pointer	assigns a symbolically specified location a pointer value
set_value	assigns a symbolically specified location a value
show	allows the server dialog to be hidden while child dialogs are up
update_lookup	Updates the O(1) ID lookup table after IDs have been changed by assignment
write_file	Write to a FLASH file
write_float_array	writes values to a float array
write_int_array	writes values to an integer array
write_wire	assigns values to elements of the specified wire buffer

3.1. add_module

Syntax:

add_module,*layout_instance_name*,*offset*,*module1*, … ,*moduleN*

where:

layout_instance_name identifier for a previously allocated layout,
offset must be an integer >= 0,
each *moduleI* must be the name of a module created by **create_module**

This call adds the specified modules to the layout. The layout and modules must already have been allocated by previous server calls. The layout internally contains an array of module pointers. This function sets the module pointers starting at the zero-based offset within the array. Call this function multiple times to populate all modules within the layout.

On success the reply is:

success

3.2. add_symbol

Syntax:

```
add_symbol,name,className,address,[check]
```

Adds a entry to the symbol table based on the in-memory address of the object. Arguments:

name - name of the object. Must be unique.
className - class name of the object. (Module class, wire class, pin class, etc.)
address – physical address where the object is stored.
[check] – optional Boolean which specifies whether additional checks should be performed to validate the symbol table entry.

If successful, an object of the specified className will be added to the symbol table. This command is typically used to attach to a running executable.

When check equals 1, the function does additional checks to verify that the symbol entry is valid:

1. Checks if address is within one of the heaps.
2. Checks if the class object pointed to by the object is within one of the heaps.
3. Verifies that the class of the object in memory matches the className.

The reply is one of:

```
success,name=0x%08x
failed, argument count
failed, address invalid
failed, address 0x%08x does not point to an instance of class className
failed, no such class as className
failed, instance name already defined
```

3.3. add_symbol_index

Syntax:

```
add_symbol_index,name,className,index
```

Adds a entry to the symbol table based on its location (index) within the linked list of objects.
Arguments:

name - name of the object. Must be unique.
className - class name of the object. (Module class, wire class, pin class, etc.)
index – location of the object within the linked list.

If successful, an object of the specified className will be added to the symbol table. This command is typically used to attach to a running executable.

The reply is one of:

```
success,name=0x%08x
failed, argument count
failed, index invalid
failed, no such class as className
failed, instance name already defined
```

3.4. add_symbol_id

Syntax:

```
add_symbol_id,name,className,ID
```

Adds a entry to the symbol table based on its ID within the linked list of objects. Arguments:

name - name of the object. Must be unique.
className - class name of the object. (Module class, wire class, pin class, etc.)
ID – unique ID assigned to the object at instantiation time.

If successful, an object of the specified className will be added to the symbol table. This command is typically used to attach to a running executable.

The reply is one of:

```
success,name=0x%08x
failed, argument count
failed, ID invalid
failed, no such class as className
failed, instance name already defined
```

3.5. audio_pump

Syntax:

audio_pump [, *file_name*]

If *file_name* is given, creates a WMA/WAV/MP3 stereo reader at 44.1KHz, otherwise creates a sound card Line In stereo reader at 44.1KHz. It then creates a 44.1KHz 8 channel player, and calls the framework pump at a rate suitable to pump samples.

If there are no wires bound to input or output pins, the code directly connects the input to the first 2 channels of the output, making a simple player. This capability is for testing.

The reply is one of:

success
failed, open sound card for input returned an error
failed, player create returned 0x%08x
failed, renderer create returned 0x%08x

where the value is the error code from DirectSound.

If the server is connected to a target, the *file_name* argument is not permitted.

3.6. audio_stop

Syntax

audio_stop

Unconditionally terminates the audio pump if running. The reply is always:

success

If the server is connected to a target, the target DMA and rendering is halted.

3.7. bind_wire

Syntax

bind_wire,wire_name,I/Opin

Causes ***wire_name*** to be bound to the named I/O pin. It is an error for an I/O pin to be bound more than once. All wire binding is released by ***destroy***.

The reply is:

success,*heap1,heap2,heap3*

3.8. cmd

Syntax:

cmd,<opcode>,<result_count>,<arg1>,....,<argN>

This is a backdoor command, which allows an arbitrary command packet to be sent to the target processor. Where

opcode – 16-bit command opcode (see ProxyIDs.h)
result_count – number of arguments that the command is expected to return. For most commands, this will be zero since the only return is the function return.
arg1, …, arg2 – packet payload. No CRC; this is automatic.

Some commands do not take any arguments. For example, a call to destroy the target would look like

cmd,12,0

Another example is a call to Create Module. It calls ClassModule_Constructor(), and its arguments are:

cmd,15,1,<ClassID>,<nIO>,<K>,<wire1>,...,<wireJ>,<module1>,...,<moduleK>

where the number of wires J is encoded in the nIO bitfield. The command has one result - the module address.

The return is either:

success [<ret1>,...,<retN>]

or a normal failure code.

3.9. compile

Syntax:

compile,*flags*,*source_file*,*destination_file*

Instructs the server to compile ***source_file*** which must be a file containing valid commands from this document into ***destination_file*** in AWB binary format.

If ***flags*** is non-zero, the resulting file will be in relative form (and for V4 targets will use objected+offset addressing), otherwise it will be absolute. Note that when compiling for V4 targets, this should always be non-zero.

The command fails if ***source_file*** contains a **make_binary** command.

The command silently strips any command that tries to read a value in any way, or to operate in any way on a GUI object (inspector dialogs) from the output bit stream.

On success, the reply is:

success

otherwise

failed,*reason*

There are many possible reasons for failure.

3.10. connect

Syntax:

connect,*client_name*,*port*

Instructs the server to reply to *client_name* on the given *port*. *client_name* must be the name of the PC running the client. If both client and server are on the same PC, the name ***localhost*** should be used. The default *port* is 12001. Any port may be used provided it is larger than 1024. The reply is:

success,*client_name*,*port*

On receipt of this reply, the client knows it is connected.

3.11. create_active

Syntax:

create_active,dialog,left,top,moduleName [, bgnd_color [, text_color]]

where:

dialog must be a dialog created by ***create_dialog***,
left,top, describes a position on the dialog surface
moduleName is a dot-expression that evaluates to the name of a module, optionally multiple expressions separated by semicolons may be used
bgnd_color will be the dialog background color, default from [InspectorColors] InspectorFace=230,230,230
text_color will be the text color of text controls, default [InspectorColors] TextColor=0,0,0

Creates a small control comprising 4 radio buttons in the order Active, Muted, Bypassed, Inactive. The control initializes its state from the specified module (or first module if there are several semicolon separated names). At 5Hz, it reads the module state (or first module if there are several semicolon separated names), causing the display to update if the module state changes.

On choosing a radio button, all modules (if there are several semicolon separated names) will be set to the new state.

3.12. create_bitmap

Syntax:

```
create_bitmap,dialog,left,top,width,height,fileName
```

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes a rectangle on the dialog surface
fileName is the name of an image file (BMP only) to display

Causes the specified image to be rendered on the dialog in the specified rectangle. Images are rendered beneath any controls the dialog may have, and, if more than one is specified, are drawn in order – that is the most recently specified bitmap appears above all earlier ones.

If the height or width are negative (usefully –1), then only the [top,left] position is used – the size of the rectangle is obtained from the image; otherwise the image is stretched or shrunk as needed in both axes to fit the rectangle specified.

3.13. create_button

Syntax:

```
create_button,dialog,left,top,width,height,caption,script_file
```

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes a rectangle on the dialog surface
caption is the text to appear on the button face
script_file names a file of commands to be executed when the button is clicked

Creates a button control on the dialog of the specified size. On clicking the button ,the commands in the scrip file are executed.

ScriptFile may be commands instead of a filename, those commands are:

RemoveControls – deletes all the controls on a dialog
RemoveBitmaps – deletes all the images created by create_bitmap.

3.14. create_checkbox

Syntax:

```
create_checkbox,dialog,left,top,width,height,legend,attributes,dot-expression
```

where:
dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes a rectangle on the dialog surface
legend is the text to appear to the right of the checkbox
attributes is a string of attribute controlling the appearance of the check box control
dot-expression is an expression to assign the value of the checkbox (0=not checked, 1=checked) each time the state of the checkbox changes

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

readonly=val – 0 or 1, default 0; when set prevents the user changing the selection

Creates a checkbox control on the dialog of the specified size. On clicking the checkbox (causing its state to toggle) the new check state is assigned to the dot-expression. As with all assignments, the Set() function of the appropriate module is called after the assignment. At a rate of 5Hz, the expression is examined: if it changes the check mark is updated.

3.15. create_dialog

Syntax:

```
create_dialog,dialogName,left,top,width,height,width2,height2,caption[ ,bgnd_color [, combo_color
    [, text_color]]]
```

where:
dialogName must be a an identifier not in use by any object
left,top,width,height describes the size and position of the dialog surface
width2,height2 describes the alternate width and height of the dialog – zero values mean there is no alternate size.
caption will be the dialog caption
bgnd_color will be the dialog background color, default from [InspectorColors] InspectorFace=230,230,230
combo_color will be the color of drop list backgrounds, default from [InspectorColors] DropList=240,240,255

text_color will be the text color of text controls, default [InspectorColors] TextColor=0,0,0

Creates a new dialog with the given name and caption. Dialogs and all their child controls are destroyed either by ***destroy*** or specifically by ***destroy_dialog***.

3.16. create_droplist

Syntax:

```
create_droplist, dialog,left,top,width,height,nameValueList,caption,attributes,dot-expression
```

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes the position and width of the drop list control
nameValueList of the form "string=value …." used to populate the list and specify the value associated with each item
caption specifies the caption to appear above the drop list control
attributes is a string of attribute controlling the appearance of the combo box control
dot-expression is an expression to assign the value of the selection each time the selection changes

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

readonly=val – 0 or 1, default 0; when set prevents the user changing the selection

Creates a droplist control on the dialog of the specified size. On selecting an item in the droplist associated value is assigned to the dot-expression. As with all assignments, the Set() function of the appropriate module is called after the assignment. At a rate of 5Hz, the variable is examined: if it has changed, the selection is updated.

3.17. create_awslist

Syntax:

```
create_awslist, dialog,left,top,width,height,nameValueList,caption,attributes
```

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes the position and width of the drop list control
nameValueList of the form "string=filename …." used to populate the list and specify the file name associated with each item
caption specifies the caption to appear above the drop list control
attributes is a string of attribute controlling the appearance of the combo box control

dot-expression is an expression to assign the value of the selection each time the selection changes

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

readonly=val – 0 or 1, default 0; when set prevents the user changing the selection

Creates a droplist control on the dialog of the specified size. On selecting an item in the droplist the associated file is executed as an AWS script file.

3.18. create_edit

Syntax:

create_edit,dialog,caption,left,top,attributes,caption,dot-expression [,in-expression]

where:
dialog must be a dialog created by ***create_dialog***,
left,top describes the position and width of the drop list control
attributes is a string of attribute controlling the appearance of the edit control
caption specifies the caption to appear above the edit control
dot-expression is an expression to assign the value of the edit box
in-expression if present is checked at 5Hz, and updates the edit control when it changes

Creates an edit control with a caption above in a box 69 wide by 42 high.

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

format=format_specifier – a printf style format to use when formatting values, default %.2f
stepsize=step – default 0, the amount by which displayed values will be quantized
min=val – default -100, the minimum displayable value on the meter
max=val – default 0, the maximum displayable value on the meter
readonly – 0 or 1, default 0; when set prevents the user editing the value

3.19. create_filelist

Syntax:

create_filelist,dialog,name,left,top,height,buffer_expression,buffer_size_expression,async_expression, type_expression[,filepath[,rate]]

where:
dialog must be a dialog created by ***create_dialog***
name specifies the caption to appear above the control

left,top,height describes the position and height of the control
buffer_expression expression specifying the start address of the buffer used to transfer data to the target
buffer_size_expression expression specifying the size of the transfer buffer.
async_expression expression specifying where the PC should write asynchronous notifications.
type_expression expression specifying where the PC should a 32-bit integer containing the first 4 characters of the file extension.
filepath – optional list of files to populate dialog with at startup
rate – rate in Hz at which to poll and fill the transfer buffer.

The file list control is used to stream data from a file to the target. The transfer buffer holds a total of buffer_size + 1 32-bit words. The final word in the transfer buffer, buffer[buffer_size] is the handshaking word. At a 10 Hz rate, the control checks whether

buffer[buffer_size] == 0

If non-zero, nothing happens. If equal to zero, the control opens the current file, seeks to the current seek position, reads buffer_size*4 bytes from it (if possible), fills buffer with the actual bytes read, and closes the file. The low 24 bits of the handshaking word at buffer[buffer_size] is set to the number of bytes reads. The high 8 bits are set to one of the following notifications:

FIOS_NewStream – Indicates that we are at the start of a new file
FIOS_NextBlock – Set for the second block onward until the next to last block
FIOS_LastBlock – Indicates that this is the last block of data in a file.

(These are defined in Framework.h).

Typically, a single write to the target of length buffer_size+1 words occurs. Only at the end of the file are two separate write performed; the data followed by the handshaking word.

If the end of file is reached and there are no more files to play, the writing of data stops. Otherwise, the next file is opened and playback continues.

The asynchronous handshaking word notifies of other conditions.

FIOS_Stopped - generated by Stop only
FIOS_Paused - generated by Pause only
FIOS_Error - generated by a file I/O error when reading the current file, no data is sent

The *type_expression* indicates the extension of the file being played to the target processor. *type_expression* is updated whenever the first block of a new file is played. The file extension is converted to upper case, zero-padded or truncated, and packed into a 32-bit

integer. The value written is in little-endian format and the least significant byte of the word holds the first character. For example,

mp3	0x00	0x33	0x50	0x4D
		'3'	'P'	'M'

3.20. create_graph

Syntax:

create_graph,*dialog,left,top,width,height,attributes,dot-expression,count*

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes the position and size of the graph
attributes is a string of attribute controlling the appearance of the meter
dot-expression is describes an element taken to be the first in an array
count is the number of elements to use

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

format=format_specifier – a printf style format to use when formatting values, default %.2f
mapping=[db20|undb20|lin[ear]] – default db20. The value is displayed according to the mapping.
stepsize=step – default 0, the amount by which displayed values will be quantized
meteroffset=offs – default 0, an amount to be added to values before use
min=val – default -100, the minimum displayable value on the meter
max=val – default 0, the maximum displayable value on the meter
numbers – default 0, when non-zero specifies that numbers should be drawn above each element

This command creates a graph object of the specified size. The width of the object is divided by count to give the width of each stripe. 10 times a second, the target array is queried for ***count*** values, and those values used to display the graph stripes. If numbers is set, then the top 16 pixels of the graph is used to display the numeric value of each element according to the format specified. The width of each strip needs to be 25 or more when displaying numbers to avoid truncation of the text.

3.21. create_grid

Syntax:

create_grid,*dialog,left,top,width,height,attributes,dot-expression,count1[,count2]*

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes the position and size of the grid control
attributes is a string of attribute controlling the appearance of the grid control
dot-expression describes an element taken to be the first in an array
count1 is the size of the first dimension
count2 if present is the size of the second dimension

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

format=format_specifier – a printf style format to use when formatting values, default %g
min=val – default –1e10, the minimum displayable value on the grid
max=val – default 1e10, the maximum displayable value on the grid
colwidth – default 50, value must be >= 50, width of column in pixels
sidewidth – default 30, value must be >= 30, width of first column in pixels

The command creates a grid control of the specified size. If count2 is given, the control as count2+1 columns, the first being the index, otherwise the control has 2 columns, the first being the index. The control operates as a very simple spreadsheet. On changing the value of any cell, the underlying array element is assigned, and the corresponding module's set member is called. At 5Hz intervals, the grid will repaint itself if any element has changed value.

3.22. create_layout

Syntax:

create_layout,*layout_instance_name*,*divider*,*nModules*

where:

layout_instance_name must be an identifier not currently defined,
divider must be an integer >= 1,
nModules must be an integer >= 1

This creates a layout object named *layout_instance_name* that can hold a total of *nModules* with the given *divider*. A layout is a collection of modules that are all pumped together at the given division rate. Only memory for the layout is allocated and a few internal fields of the layout structure set; no modules have been added. Modules must be subsequently added by calls to add_module.

On success the reply is:

success, heap1,heap2,heap3,*layout_instance_name=absolute_address*

3.23. create_led

Syntax:

```
create_led, dialog,left,top,width,height legend,dot-expression
```

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes the top-left corner of the LED control
legend is the text to appear to the right of the LED image
dot-expression is an expression to evaluate at 5Hz – if non zero the LED is shown lit

Creates an LED control. If the value described by dot-expression is non-zero, the LED is shown bright green, otherwise dark green. The expression is evaluated every 200mSec.

3.24. create_lookup

Syntax:

```
create_lookup, maxId
```

where:

maxID must be a non-zero integer

Creates a lookup table that handles Ids in the range 1..maxID by providing a fast O(1) lookup table when using relative addressing.

3.25. update_lookup

Syntax:

```
update_lookup
```

where:

maxID must be a non-zero integer

Updates the fast O(1) ID lookup table for relative to match changed IDs. Must be used after any IDs are changed by assignment.

3.26. create_meter

Syntax:

```
create_meter, dialog,left,top,attributes,dot-expression
```

where:

dialog must be a dialog created by ***create_dialog***,
left,top describes the top-left corner of the LED control
attributes is a string of attribute controlling the appearance of the meter
dot-expression is an expression to evaluate at 5Hz – if non zero the LED is shown lit

Creates a meter control. The value described by dot-expression is evaluated every 200mSec, and used to update the appearance of the meter.

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

format=format_specifier – a printf style format to use when formatting values, default %.2f
units=units_name – no default, used to name the units, for example dB
mapping=[db20|undb20|lin[ear]] – default db20. The value is displayed according to the mapping.
ticks=nTicks – default is 2, range is 2-32, this is the number of tick marks to display
useticks=[0|1] – default is 0, when 1 tickmarks are drawn
tickmarks="v1, … , vN" – a list of labels to apply to tickmarks up to a maximum of 8 values, no default
stepsize=step – default 0, the amount by which displayed values will be quantized
meteroffset=offs – default 0, an amount to be added to values before use
min=val – default -100, the minimum displayable value on the meter
max=val – default 0, the maximum displayable value on the meter
height=val – default is natural control height, values larger than default stretch the control vertically downwards

3.27. create_module

Syntax:

create_module,module_instance_name,className,nInputs,nOutputs,nScratch,*[wires]*,args…

where:
module_instance_name must be an identifier not currently defined,
className must be the name of a Module Class,
nInputs is the nuber of module inputs required,
nOutputs is the number of modules required,
nScratch is the number of scratch wires required,
[wires] is a list of wire names obtained from **create_wire**, of which there are exactly *nInputs+nOutputs+nScratch* names,
args… is a set of arguments to initialize the module – the number of arguments is that required by the module

This create a module object named *module_instance_name* with the given properties. Modules are only useful when part of a layout constructed using **create_layout**.

On success the reply is:

success, heap1,heap2,heap3,*module_instance_name=absolute_address*

3.28. create_pintype

Syntax:

create_pintype,*instanceName*,*nSamples*,*nChannels*,*sampleSize*,*sampleRate*,*isComplex*

where:

instanceName must be an identifier not currently defined,
nSamples is the size of the wire buffer in samples,
nChannels is the number of interleaved channels,
sampleSize is the sample size in bytes which must be 4 currently,
sampleRate is the sample rate in suitable units
isComplex Boolean that indicates whether the wire holds complex data

This creates a pin descriptor object named *instanceName* with the given properties. The created object may only be used as input to **create_wire**.

On success, the reply is:

success, heap1,heap2,heap3,*instanceName=absolute_address*

3.29. create_slider

Syntax:

create_slider, dialog,left,top,attributes,dot-expression[,read-expression]

where:

dialog must be a dialog created by ***create_dialog***,
left,*top* describes the top-left corner of the LED control
attributes is a string of attribute controlling the appearance of the meter
dot-expression is an expression to assign the position of the slider to when its position changes. Multiple assignments may be specified by separating expressions with semicolon.
read-expression if present is a location to watch at 5Hz – if it changes, the slider position is changed to match.

Creates a slider or knob control. The value described by dot-expression is assigned the slider value when it changes.

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

min=val – default 0, the minimum value of the slider
max=val – default 1, the maximum value of the slider
value=val – default 0, the initial position of the slider
format=format_specifier – a printf style format to use when formatting values, default %.2f
units=units_name – no defult, used to name the units, for example dB
mapping=[log|lin[ear]|db20|undb20] – default linear. The value is displayed according to the mapping. Log is not possible unless min > 0.
ticks=nTicks – default is 2, range is 2-32, this is the number of tick marks to display
useticks=[0|1] – default is 0, when 1 tickmarks are drawn
fixedticks=nFixedTicks – default is 2, range is 2-32, this is the number of fixed ticks to display
tickmarks=v1, … , vN – a list of labels to apply to tickmarks up to a maximum of 8 values, no default
stepsize=step – default 0, the amount by which displayed values will be quantized
control=[knob|slider] – default slider. If knob, a rotary knob control is shown instead of a slider.
height=val – default is natural control height, values larger than default stretch the control vertically downwards. If control=knob, this value is ignored.
continuous=[0|1] – default 1, when 1 all changes are assigned as they happen, otherwise changes are sent only when the user releases the mouse
muteonmin=[0|1] – default 0, when 1 the underlying variable is set to 0 when the knob is turned to its minimum value. This is useful for dB controls which should mute when turned all the way down.

3.30. create_spline

Syntax:

```
create_spline, dialog,left,top,width,height,attributes,instanceName
```

where:

dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes the control position and size
attributes is a string of attribute controlling the appearance of the control
*instanceName*is a base dot expression within which members of fixed names will be accessed

The attributes string must be a space separated string consisting of one or more of the following. If items are repeated, the right-most one is the one that takes effect.

minx=val – default 0, the minimum X value
maxx=val – default 9, the maximum X value
miny=val – default 0, the minimum Y value
maxy=val – default 3, the maximum Y value
order=val – default 2, for testing only
mapping=[log|lin[ear]] – default linear. The value is displayed according to the mapping. Log is not possible unless miny > 0.
ticks=nTicks – default is 2, range is 2-32, this is the number of tick marks to display
useticks=[0|1] – default is 1, when 1 tickmarks are drawn
fixedticks=nFixedTicks – default is 2, range is 2-32, this is the number of fixed ticks to display
tickmarks=v1, … , vN – a list of labels to apply to tickmarks up to a maximum of 8 values, no default
stepsize=step – default 0, the amount by which displayed values will be quantized
control=[knob|slider] – default slider. If knob, a rotary knob control is shown instead of a slider.
maxpoints=val – default 10, for testing only
points=val – default 10, for testing only

Creates a spline control. This control displays points XY points on a graph. If order==2, the points are joined by straight lines. If order==4, the points are connected by a natural spline. The curve is drawn in green. The points are drawn as small blue boxes. You can drag the boxes around, causing the curve to be redrawn, and the dsp to be updated. On first creation, the control is populated from the DSP.

If the instanceName is empty, the control is stand-alone with 10 points y=sqrt(x), x=0..9 and not connected to the DSP. In this mode, the operation of the spline control may be tested.

3.31. create_text

Syntax:

```
create_text, dialog,left,top,widh,height,legend
```

where:
dialog must be a dialog created by ***create_dialog***,
left,top,width,height describes size of the control
legend is the text to appear

Creates a static text control of the specified size, and sets its text to legend. Any occurrence of '\n' in the legend will cause the legend to wrap.

3.32. create_wire

Syntax:

```
create_wire,instanceName,pinTypeName
```

where:
instanceName must be an identifier not currently defined,
pinTypeName must be the name of a pin instance created by **create_pintype**

This creates a wire object named *instanceName* with the properties specified by the *pinTypeName*.

On success, the reply is:

```
success, heap1,heap2,heap3,wire_instance_name=absolute_address
```

3.33. delete_file

Syntax:

```
delete_file,filename
```

This command deletes the specified file from the FLASH file system if it exists, in which case it reports success. There are many possible failures, including file not found, and file system not implemented.

Note that deleting a file only marks its directory entry deleted, it does not release the storage used by the file or its directory entry. Repeated creating and deleting files will consume all storage eventually. You can return the file system to its initial state with **erase_all**.

3.34. destroy

Syntax:

```
destroy
```

This command unconditionally destroys all objects. On success, the reply is:

```
success,heap1,heap2,heap3
```

3.35. destroy_dialog

Syntax:

```
destroy_dialog,dialog
```

where:
dialog must be a dialog created by ***create_dialog***,

This command destroys the named dialog.

3.36. dialog_state

Syntax:

```
dialog_state,dialog,state
```

where:
dialog must be a dialog created by ***create_dialog***,
state must be 0 or 1

This command sets the given dialog to its initial size if zero, otherwise to its alternate size. If the alternate size dimensions given to *create_dialog* were zero or the same as the initial size, the command has no effect.

3.37. erase_all

Syntax:

```
erase_all
```

This command erases all files on the target FLASH file system, restoring it to the initial empty state. It fails if the target does not have a file system.

Erasing a large FLASH chip can take some time.

3.38. end_binary

Syntax:

```
end_binary
```

Terminates logging of binary commands and writes the file. This command works in conjunction with make_binary.

3.39. exists_dialog

Syntax:

```
exists_dialog,name
```

Checks if a dialog with the specified name already exists. The function returns either:

success,0	(Dialog does not exist)
success,1	(Dialog does exist)
failed, argument count	(Incorrect number of arguments to the function)

3.40. fast_read

Syntax:

```
fast_read,addr_expr,count
```

Reads the specified number of elements in an array of data and returns the result as binary data rather than as text. This command is only supported through the MATLAB AWEClient.dll and is not for general use. The format of the binary reply packet is described in section 6.

3.41. fast_write

Syntax:

```
fast_write,addr_expr
```

Writes the arraypassed by Matlab to the target layout starting at the address given. This command is only supported through the MATLAB AWEClient.dll and is not for general use. It causes the AWE server to receive a binary packet containing the Matlab array values as documented in section 6.

3.42. exit

Syntax:

```
exit
```

This command destroys the server.

3.43. file_logging

Syntax:

```
file_logging,full, filename
file_logging,half, filename
file_logging,end
```

The first form starts logging all commands and replies by appending them to the given *filename*. The second form start logging all replies to the given *filename*. The last form turns of logging to file.

The form of received message log items is:

YYYY/MM/DD HH:MM:SS.mmm: << *message*

The form of sent message log items is:

YYYY/MM/DD HH:MM:SS.mmm: >> *message*

In each case, *mmm* is the milliseconds past the second.

On success, the reply is:

success

3.44. foreground

Syntax:

foreground

Brings all Audio Weaver windows to the foreground (top most in Z-order). They are thus made visible if they were behind other windows.

3.45. getini

Syntax:

getini,*section,key*

Replies:

success,section=*section_name*,key=*key_name*,value=*key_value*

3.46. get_call

Syntax:

Get_call,*module_name,mask*

Replies:

success,*module_name=address*

3.47. get_filesystem_info

Syntax:

get_filesystem_info

This command queries the target FLASH file system properties. On success it returns:

success,*type*,*size*,*available*,*overhead*,*deleted*,*inuse*,*free*,*sizes*

where:

type is 1 for Native, or 2 for FLASH.
size is the target device size in words – note that the implementation may only use a portion of the total FLASH storage for the file system.
available is the number of available storage words.
overhead is number of words used for internal data structures
deleted is the number of words used by deleted files
inuse is the number of words used for all purposes
sizes is (block size in words << 16) | max filename length

Note that the file system does not release storage from deleted files – that storage is lost. Repeatedly creating and deleting files will consume all storage. The file system can be restored to its initial empty state with **erase_all**.

3.48. get_first_file

Syntax:

get_first_file

This command gets information about the first file in the FLASH file system. On success the reply is either:

success,1,*length*,*filename*

if there are any files, or

success,0,,

if the file system is empty.

Several failures are possible, including failures due to the target not having a file system.

3.49. get_next_file

Syntax:

get_next_file

This command may only be used after first having used get_first_file. It resturn information about successive files. On success the reply is either:

success,1,*length*,*filename*

if there are any files, or

success,0,,

if there are no more files. Call this as many times as needed to enumerate all files on the target.

Several failures are possible, including failures due to the target not having a file system.

3.50. read_file

Syntax:

read_file,*filename*

This command reads the specified file from the target FLASH file system, and writes the file as AWE_directory/*filename* to your hard drive, in which case it reports success. There are many possible falures including file not found and the target not having a file system.

3.51. write_file

Syntax:

write_file,*filename,attribute*

This command writes the specified local hard disk file to the target FLASH file system with the file ***attribute*** specified. If a file of that name exists on the target, it is first deleted (see **delete_file**). There are many possible failures including file not found on your hard disk, not enough space on the target, and the target not having a file system.

The ***attribute*** value may be any 7 bit value constructed by orring the following together expressed as decimal:

```
#define LOAD_IMAGE              0x01
#define STARTUP_FILE            0x02
#define DATA_FILE               0x04  // Any file of type "Other"
```

```
#define COMPILED_SCRIPT          0x08
#define COMMAND_SCRIPT           0x10
#define PRESET_SCRIPT            0x20
#define COMPILED_PRESET_SCRIPT   0x28
#define LOADER_FILE              0x40
```

Common useful values are 0x18 (= decimal 24) for a compiled AWB file, and 0x1a (= decimal 26) for a bootable compiled AWB file. Other possible attribute combinations are generally not useful.

Targets that have a FLASH file system will locate the first file with the 0x1a attribute and execute as an AWB compiled script during boot. There should be only one file with this attribute in the file system – it is indeterminate which will be executed if there is more than one.

A useful set of commands to compile a script file, and load it into a FLASH file system is:

erase_all
compile,1,*source_file*.aws, *destination_file*.awb
write_file, *destination_file*.awb,26

When you next reset the target, the layout should be running. Note that the AWS and AWB extensions are convention only, you can use anything you like.

3.52. get_first_io

Syntax:

get_first_io

This command returns the first I/O object, as in this example:

success,InputLeft=5651528,Class=InputType,InstanceID=int:2129506305,NextInstance=int:5651504,Class Descriptor=*ClassDescr:5152136,pinSize=int:67174432,zeroFill=int:0,pWire=*Wire:0

The format is

success,*instance_name=address*,Class=*className*,InstanceID=int:*id*,NextInstance=int:*address*,ClassDescriptor=*ClassDescr:*address*,
packedPinSize=int:*value*,sampleRate=int:*value*,pWire=*Wire:*address*

The *packedPinSize* value is a packed binary representation of:

| 8 bits: sampleSize | 8 bits: nChannels | 16 bits: nSamples |

which for the above example, is 0x4010020, which represents:

| 4 | 1 | 32 |

The objects are enumerated in the order **Input**, **Output.**

See **get_next_io**.

3.53. get_first_object

Syntax:

get_first_object

This command returns the first created object. The form of the reply is:

success,*instanceName=address*,Class=*className*,*members*, …

where each member is formatted as:

member_name=member_type:*value*

The layout of all classes is given in the schema file, where each member is named and its type given: *className* will be found in the schema file. The value is displayed appropriately for the type: **float** values are displayed using %g, all other values are displayed as decimal unsigned integers.

If the member is an array of fixed bounds in the schema, then each element of the array is displayed in the form:

member_name[*subscript*]=*type*:*value*

where the subscript ranges from 0 to N-1.

Where members are inherited from a base class, each inherited member is listed.

3.54. get_heap_count

Syntax:

get_heap_count

This command returns:

success, number_of_heaps

3.55. get_heap_size

Syntax:

```
get_heap_size
```

On success the reply is:

```
success, free1,free2,free3,addr1,addr2,addr3,size1,size2,size3
```

where:

freeN - is the number words available in heap N.
addrN – is the address of the next free word in heapN.
sizeN – is the total size of heapN

All sizes are in 32-bit words.

3.56. get_executable_dir

Syntax:

```
get_executable_dir
```

Returns the directory containing the currently connect AWE_Server.exe executable. Reply:

```
success,c:\Program Files\DSP Concepts\Audio Weaver Designer\Bin
```

3.57. get_module_state

Syntax:

```
get_module_state,module_instance_name
```

where:

module_instance_name is the name of a module created by **create_module**, or a dot-expression describing a member of some object that is a module

On success, the reply is:

```
success, module_instance_name=address,state
```

where:

module_instance_name is the argument of the command,
address is the address of the module,
state is a decimal value, and one of

0: active
1: bypass
2: mute

3: inactive

When first created, modules are *active*. See **set_module_state**.

3.58. get_moduleclass_count

Syntax:

```
get_moduleclass_count
```

This command returns:

```
success,module_class_count
```

where:
module_class_count is the number of module classes in the framework.

3.59. get_moduleclass_info

Syntax:

```
get_moduleclass_info,module_class_index
```

where:
module_class_index must be in the range 0 to one less than the value returned by **get_moduleclass_count.**

On success, the return value is:

```
success,className=address,classID,nParams
```

where:
className is the name of the class as it appears in the schema file,
address is the absolute address of the class object,
classID is the numeric value of the class id,
nParams is the number of public and private parameters an instance of the module may take. The values are packed as separate 16 bit numbers into a 32 bit value. The high 16 bits represent the number of private words; the lower 16 bits represent the number of public words.

3.60. get_next_io

Syntax:

```
get_next_io
```

Returns the next I/O object in the form described in **get_first_io**, or:

failed, no more I/O pins

if there are no more I/O objects to enumerate.

3.61. get_next_object

Syntax:

get_next_object

Returns the next object in the form described in **get_first_object**, or:

failed, no more objects

if there are no more objects to enumerate.

3.62. get_object_byaddress

Syntax:

get_object_byaddress,*address*

where:
address is the address of some object

The command looks up *address* in the object symbol table. If found, the reply value is as described in **get_first_object**, otherwise it is:

failed, not address of object

3.63. get_object_byname

Syntax:

get_object_byname,*instanceName*

where:
instanceName is some identifier

The command looks up *instanceName* in the object symbol table. If found, the reply value is as described in **get_first_object**, otherwise it is:

failed, '*instanceName*' is undefined

3.64. get_schema

Syntax:

get_schema,*className*

where:
className is some identifier

The command looks up *className* in the schema symbol table. If found, the reply value is:

success,Class=*className*,ClassID=*id*,*member*,…

where:
className is the argument to the command,
id is the numeric id of the class,
each member is formatted as:
member_name=type

otherwise, the reply is:

failed, class '*className*' is undefined

Note that unlike **get_first_object**/**get_next_object**, the inherited members from base classes are not displayed.

3.65. get_value

Syntax:

get_value,*expression*

where *expression* is formed as follows:

instanceName [. *memberName*]

InstanceName must be the name of some object. The first *memberName* must name a member of the class of which *instanceName* is an instance. Subsequent terms depend on the type of the member as follows:

Member Type	Followed by
int	nothing, reply is success,*address*,int,*intvalue*
float	nothing, reply is success,*address*,float,*floatvalue*
[N]int	[*0 : N-1*], reply is success,*address*,int,*intvalue*
[N]float	[*0 : N-1*], reply is success,*address*,float,*floatvalue*
*int	[*subscript*], reply is success,*address*,int,*intvalue*

*float	[*subscript*], reply is success,*address*,float,*floatvalue*
**className*	**.member** belonging to *className* (follows pointer)
**className	**[***subscript***].member** belonging to *className* (follows subscripted pointer)
className	**.member** belonging to *className* (accesses member)

Note that the final three type name members: if the types of those members are not one of the first 6 scalar forms, then more members must be named to complete the expression. This continues iteratively until the expression reaches one of the first 6 scalar forms.

If the expression is not legal according to these rules, one of the following may be returned:

failed, '*string*' is not an identifier
failed, '*name*' requires dot expression
failed, no such member of '*class*' as '*string*'

3.66. get_version

Syntax:

get_version

Returns version information about the currently connected server. The reply is of the form:

success,2.0,Oct 24 2008 14:10:34

where "2.0" is the first and the rest of the string is the build date and time.

3.67. gui_logging

Syntax:

gui_logging,0
gui_logging,1
gui_logging,off
gui_logging,on

The second and fourth forms cause sent and received messages to be displayed in the server control panel, the remaining forms turn this display off.

The reply is:

success,bool_value

where the value is 1 if display is enabled, otherwise 0.

3.68. make_binary

Syntax:

```
make_binary,filename
```

Begins logging of binary commands sent from the Server to the target. The commands are buffered in internal memory on the PC. When complete, call end_binary to write the commands to the specified file filename.

make_binary is used to create compiled scripts on the target. Only a subset of commands are stored – only those needed to actually instantiate the system and begin processing. The commands logged are:

```
bind_wire
audio_pump
create_layout
create_module
create_pintype
create_wire
destroy
set_module_state
set_value
write_float_array
write_fract_array
write_int_array
```

3.69. open_web_page

Syntax:

```
open_web_page,URL
```

Displays a web page in a browser. URL is a string specifying the address of the page to display. If URL starts with "http://", "file://", or "www.", the URL is used as-is. Otherwise, the program determines if a script is currently running, and URL is a relative path, in which case the file to open is taken relative to the script path, otherwise if no script is running and the URL is a relative path, the file is taken relative to the executable (AWE_Server.exe) path. Only file names ending in ".htm" or ".html" are considered candidates for relative pathing, otherwise the URL is used as-is.

3.70. pump

Syntax:

```
pump
```

This command causes all current layouts to be pumped. Layouts that have dividers of 1 are pumped on every call, layouts with larger values are pumped on every Nth call.

If there are no layouts to pump, the replay is:

failed, no layouts to pump

otherwise it is:

success

This call is intended to be used with **write_wire** and **read_wire** for testing. See also **fast_write** and **fast_read**.

3.71. pump_layout

Syntax:

pump_layout,*layout_instance_name*

where *layout_instance_name* must be an object created by **create_layout**.

This command pumps a single layout as though by **pump** above. It is intended to be used with **write_wire** and **read_wire** for testing. See also **fast_write** and **fast_read**.

3.72. pump_module

Syntax:

pump_module,*module_instance_name*

where *module_instance_name* must be an object created by **create_module**.

This command pumps a single module as though by **pump_layout** above. It is intended to be used with **write_wire** and **read_wire** for testing. See also **fast_write** and **fast_read**.

3.73. query_pin

Syntax:

query_pin,*pintype_name*

where *pintype_name* is any of the reserved names **Input**, **Output**, or is the name of a pin type created using **create_pintype**.

On success, the reply is:

success,*pintype_name=address*,*nChannels*,*nSamples*,*sampleSize*,*sampleRate*

where:

pintype_name is the argument of the command,
address is the address of the object,
nChannels is the number of channels
nSamples is the number of samples
sampleSize is the sample size in bytes
sampleRate is the sample rate in suitable units

3.74. read_float_array

Syntax:

read_float_array,*address*,*count*

where:
address is the absolute address to read from, or a dot-expression that evaluates to an address
count is the number of values to read

The reply is

success,*val[0]*, …, *val[count-1]*

where each value is formatted using %g.

The server will crash if the reply string exceeds 64K characters.

3.75. read_int_array

Syntax:

read_int_array,*address*,*count*

where:
address is the absolute address to read from, or a dot-expression that evaluates to an address
count is the number of values to read

The reply is

success,*val[0]*, …, *val[count-1]*

where each value is formatted using %d.

The server will crash if the reply string exceeds 64K characters.

3.76. read_schema

Syntax:

read_schema,*filename*

This command reads *filename* as a schema file, and adds its definitions to those already in the schema symbol table. It is an error to attempt to define an already defined symbol. The schema file being read may refer freely to any symbol already defined.

3.77. read_wire

Syntax:

read_wire,wire_instance_name

This command reads the buffer of *wire_instance_name* and returns:

success,*wire_instance_name=addres*,*val[0]*, …, *val[N-1]*

where:
wire_instance_name is the argument of the command,
address is the address of the wire object
val[i] are the wire buffer samples formatted using %g

3.78. script

Syntax:

script,fileName

This command executes the commands stored in *fileName*.

3.79. setini

Syntax:

setini,section,key,value

Assigns to or creates in the INI file an item of the form:

[section]
key=value

3.80. set_call

Syntax:

set_call,*module_name*,*mask*

This command calls the set_call function of a module. On success the reply is:

success,*module_name=address*

3.81. set_module_state

Syntax:

set_module_state,*module_instance_name*,*state*

where:

module_instance_name is the name of a module created with **create_module**, or is a dot-expression naming a member of an object that is a module

state is a decimal value, and one of

- 0: active
- 1: bypass
- 2: mute
- 3: inactive

This command sets the state of the module. On success the reply is:

success, module_instance_name=address,state

See also **get_module_state**.

3.82. set_pointer

Syntax:

set_pointer,*destination_expression*,*pointer_expression*

This command assigns the address of the pointer_expression to the location destination_expression. On success the reply is:

success

3.83. set_timeout

Syntax:

set_timeout,*N*

where:
N is the time out in milliseconds

Sets the communication time out between the Server and the target processor. By default, the value is 10000, or 10 seconds. (This command is useful to prevent issues when you issue a command, such as erase FLASH memory, which take a long time to execute.)

On success the reply is:

success

3.84. set_value

Syntax:

set_value,*expression*,*value [, expression,value]**

where:
expression is as described in **get_value**,
value is a number to be assigned to the location described by *expression*

There may be any number of ***[expression,value]*** pairs given to the command. On completion of the last assignment, the Set() command of each unique module instance (if any) referenced by any of the *expression*s are called.

Any value may be an expression of the form '*&dot-expression*'. This has the value of the address of the specified member. When used, the corresponding expression must have a type of ***int***.

On success the reply is:

*success,address,type,value [,address,type,value]**

3.85. show

Syntax:

show,[0|1]

If the server has any dialogs created by ***create_dialog***, then *show,0* causes the server dialog to be hidden. The dialog is un-hidden by ***destroy***, using ***destroy_dialog*** to destroy the last child dialog, or by ***show,1***. The *show,0* command does nothing if there are no child dialogs.

3.86. write_float_array

Syntax:

write_float_array,*address*,*val0*,….,*valN-1*

where:

address is the absolute address to read from, or a dot-expression that evaluates to an address

This command writes the values to each successive float location starting at ***address***. The reply is

success

Misuse of the command can corrupt storage.

3.87. write_int_array

Syntax:

write_int_array,*address*,*val0*,…,*valN-1*

where:

address is the absolute address to read from, or a dot-expression that evaluates to an address

This command writes the values to each successive int location starting at ***address***. The reply is

success

Misuse of the command can corrupt storage.

3.88. write_wire

Syntax:

write_wire,wire_instance_name,values…

where:

wire_instance_name is a wire created by **create_wire**,
values are an unbounded list of numeric values

This command writes the values to the buffer of ***wire_instance_name***. If too few values are supplied to fill the buffer, the buffer is zero filled. If too many values are supplied, the extra are ignored.

On success, the reply is:

success, *wire_instance_name=address*

4. Error Messages

Commands can produce error messages from the following table:

Text	Description
failed, heap type index range	A heap index was not in the range of heaps
failed, ae_malloc no more storage	The given heap does not have enough storage to satisfy the requested size
failed, ae_scratch_alloc no more storage	The scratch heap does not have enough storage to satisfy the requested size
failed, constructor argument count	A create_xx call has an incorrect number of arguments
failed, class index out of range	The given class index is not in the range of classes
failed, class not found	The named class was not found in the symbol table
failed, module already owned	An attempt was made to give a module to a layout when it is already in another layout
failed, address outside heap	An attempt was made to assign to a location not in any heap
failed, not a wire	A wire argument to create_module is not actually a wire
failed, number of inputs and outputs must match	Some modules require that the number of inputs and outputs are the same
failed, input pin types must be the same	Some modules require that the types of input and output pins be the same
failed, module needs at least one input	Many modules require at least one input
failed, module needs at least one output	Many modules require at least one output
failed, inputs must match corresponding outputs	Some module require that each ***i***th input have the same type as each ***i***th output
failed, not a module	An attempt was made to give an object not a module instance to create_layout
failed, I/O count error	The input/output count is not acceptable
failed, parameter error	A parameter given to create_module is wrong for the specified module class
failed, no more objects	There are no more objects for get_next_object to display
failed, not object pointer	The address of some object was expected, but the argument is not the address of any object

failed, not input pin	create_input_wire requires that the argument be an input pin
failed, I/O pin in use	An attempt was made to bind an I/O pin that was already bound with create_input_wire or create_output_wire
failed, pin types not compatible	An attempt was made using create_input_wire or create_output_wire to bind a wire incompatible with the I/O pin
failed, pin sizes not compatible	An attempt was made using create_input_wire or create_output_wire to bind a wire not an integer multiple of the I/O pin size
failed, not output pin	create_output_wire requires that the argument be an output pin
failed, no more I/O pins	There are no more pin objects for get_next_io to display
failed, no layouts to pump	'pump' was called when no layouts exist
failed, module must have only one output	Many modules require only one output
failed, output wire must have only one sample	Some modules require that an output have only a single sample
failed, incompatible block sizes	All contained modules must have the same block size
failed, wire index out of range	A container wire vector indexed a wire out of range
failed, unknown error %d	An unknown error occurred
failed, argument count	A command had an invalid number of arguments
failed, instance name '%s' not identifier	The argument must be an identifier
failed, instance name '%s' is already used	The instance name has already been defined
failed, class name '%s' is not defined	An attempt was made to use an undefined class name
failed, class name '%s' has different classID than created instance	An object was created, but then found to have a different class than it should have
failed, instance name '%s' is not a pin type	The argument must be a pin type
failed, name '%s' undefined	A name was seen that has not been defined
failed, name '%s' is not an InputType	The name is not that of an InputType class instance (input I/O pin)
failed, name '%s' is not an OutputType	The name is not that of an OutputType class instance (output I/O pin)
failed, expression error	The dot-expression given to get_value or set_value had an error
failed, wire name '%s' undefined	A supposed wire name given to create_module is not defined
failed, wire name '%s' is not a Wire	A supposed wire name given to create_module is not of type Wire

failed, module name '%s' undefined	A supposed module name given to create_layout is undefined
failed, '%s' is not a module	A supposed module name given to create_layout is not a module
failed, unknown argument	A command that takes a symbolic argument had an unknown string argument
failed, open sound card for input returned an error	'audio_pump' could not open the sound card for input
failed, player create returned 0x%08x	'audio_pump' could not open the sound card for output
failed, renderer create returned 0x%08x	'audio_pump' could not create DirectSound object
failed, empty filename	A required filename was empty
failed, unknown command '%s'	The command keyword is unknown
failed, empty command	The command was empty
failed, can't find instance class	An attempt to lookup the class of an instance failed
failed, can't find instance	An attempt to lookup an instance address failed
failed, '%s' requires subscript	A dot-expression requires a subscript
failed, '%s' syntax error: missing ']'	Malformed subscript
failed, '%s' subscript %d out of range	A subscript is outside the array bounds
failed, '%s' requires dot expression	A dot expression stopped early
failed, no such member of '%s' as '%s'	The member name given is not a member
failed, %s(%d): empty class name	Empty class name in schema file
failed, %s(%d): syntax error in alias of class '%s'	Error while aliasing one class to another in schema file
failed, %s(%d): unknown base class'%s' in alias of class '%s'	Reference to unknown base class while aliasing one class to another in schema file
failed, %s(%d): comma expected in class '%s'	Missing comma in schema file
failed, %s(%d): syntax error in derivation of class '%s'	Syntax error parsing derived class in schema file
failed, %s(%d): unknown base class '%s' in alias of class '%s'	Attempt to derive a class from an unknown base in schema file
failed, %s(%d): unexpected '{' in body of class %s	There can only be one level of bracing in schema files
failed, %s(%d): empty member name in class %s	Member name is empty in schema file

failed, %s(%d): non-numeric dimension in class %s	Array dimension has non-numeric subscript in schema file
failed, %s(%d): expected ']' to close dimension in class %s	Missing close bracket in array dimension in schema file
failed, %s(%d): empty type name in class %s	A type name is empty in schema file
failed, %s(%d): unknown type name '%s' in class %s	A type name is undefined in schema file.
failed, %s(%d): unexpected end of file in class %s	Unexpected end of file in schema file

5. Schema Files

Schema files provide a means for describing the layout of DSP storage that is compact and has a simple grammar, and does not need the complexity of the C/C++ type system.

The server has a file **Schemas.sch** that defines all the classes in the DSP. Each schema corresponds to a structure in the code. Class names and member names must be identifiers in the C/C++ sense. Schema files support C++ comments only.

The form of a schema is:

```
ClassName
{
    member1             type
    ….
    member N            type
}
```

This is the simplest form, and directly maps to a C **struct**.

The supported types are as follows:

Type	**Description**
int	32 bit integer
float	32 bit IEEE float
[N]int	array of integer with N elements
[N]float	array of floats with N elements
*int	pointer to array of integer with unknown number of elements
*float	pointer to array of float with an unknown number of elements
*className	pointer to a class instance
**className	pointer to an array of pointers to class instance with unknown number of elements

className	a nested structure

To support mapping to DSP code, class names may have an associated class ID like this:

```
className value
{
    ….
}
```

If *value* is not present, the value zero (unknown ID) is used. The value may be in hex or decimal.

Classes may derive from other classes like this:

```
A
{
    ….
}

B, A
{
    ….
}
```

The meaning is the same as public derivation in C++. In the example above, **B** inherits all the members of **A**.

The use of a class ID may be combined with inheritance like this:

```
className value, baseClass
{
    ….
}
```

As expected, the new class gets the given class ID, and also inherits all the members of the base class. There is no limit to inheritance depth.

All type names must be declared before use. This means that a circular definition such as:

```
A
{
    m *B
}
B
{
    m *A
}
```

can't be written, since an attempt is made to refer to B before it is declared.

6. Binary packets

There are occasions when text commands are too burdensome on bandwidth. Writing samples to an input wire and reading samples from an output wire are two cases handled specially. These cases permit pumping raw audio samples into a layout for regression test, or for cases where the data is not coming from or going to a real audio device.

If a command starts with the 4-byte sequence \x03 \x00 \xff \x07 (0x07ff0003) a sequence that is not possible for text, it announces that what follows is a binary array of 32 bit values preceded by a header, of which this sequence is the first word.

The packet header looks like this:

```
struct SBinaryPacket
{
        /** Magic packet header value. */
        unsigned int m_magic;

        /** Length of data in bytes. */
        size_t m_len;

        /** Length of data in floats. */
        size_t m_nFloats;

        /** Command opcode. */
        unsigned int m_opcode;
};
```

m_magic – contains 0x07ff0003
m_len – total packet size in bytes
m_nFloats – payload size in words – note that payload data is not constrained to floats
m_opcode – command opcode, always 30 to server, always 29 from server

It is always required that string data is also sent with a command to the server to specify a destination address as an expression. This data follows the last payload word. Let us assume a payload of 32 words, and string value containing 10 characters including the terminating NULL. Then the length values will be:

m_nFloats = 32
m_len = sizeof(SbinaryPacket) + 32 * sizeof(float) + 10

The server handles incoming binary packets specially by:

- verifying the opcode is 30
- decoding the string expression to a target address
- copying the payload data to that address

Any binary message with an opcode other than 30 causes a server assertion failure, since binary messages are intended for internal AWE use only, and would be a serious error with other opcodes. The reply to this message will be **success** or **failed,<reason>**, as with other messages.

This command is sent to the AWE server by the Maltalb DLL when it processes **fast_write**.

The text command documented earlier **fast_read** generates a binary reply with payload of the number of words requested, and with no string part ***or*** a string message **failed,<reason>**. For that message, we have:

m_nFloats = <number_of_payload_words>
m_len = sizeof((SbinaryPacket) +m_nFloats * sizeof(float)
m_opcode = 29

Currently, the only code that expects this reply is the MATLAB plugin DLL.

7. Supported Messages

Let's move this into the Server Command Syntax document. Then this document will be a lot shorter and less imposing.

The function awe_fwPacketProcess() in PacketAPI.c handles all of the Framework messages. The global variable g_PacketBuffer points to the message buffer and is initialized by a call to awe_fwPacketInit(). g_PacketBuffer holds both the received message and the Framework generated reply. As discussed in Section **Error! Reference source not found.**, each message has a 16-bit ID and the file ProxyIDs.h holds their definitions.

The following table gives a summary of the messages available.

Message ID	Description
PFID_SetCall	Calls the Set function associated with an audio module.
PFID_GetCall	Calls the Get function associated with an audio module.
PFID_ClassPin_Constructor	Constructs an instance of a pin type object.
PFID_GetClassType	Returns the class type of an object
PFID_GetPinType	Queries a pin to determine its properties.
PFID_ClassWire_Constructor	Constructs a single instance of a wire
PFID_BindIOToWire	Binds a wire to a Platform input or output wire.
PFID_FetchValue	Reads a single integer value from the Target.
PFID_SetValue	Writes a single integer value on the Target
PFID_GetHeapCount	Returns the number of heaps on the Target.
PFID_GetHeapSize	Returns the number of words remaining in a specified heap.
PFID_Destroy	Destroys all allocated audio processing on the Target and reinitializes all heaps.
PFID_GetCIModuleCount	Returns the number of module classes defined on the Target.
PFID_GetCIModuleInfo	Returns information about a specific audio module class.
PFID_ClassModule_Constructor	Instantiates an audio module
PFID_ClassLayout_Constructor	Instantiates the overall processing layout (effectively a list of modules)
PFID_SetWire	Sets the data portion of a wire. Used for regression testing.
PFID_GetWire	Reads the data portion of a wire. Used for regression testing.
PFID_SetModuleState	Sets the run-time status of a module
PFID_GetModuleState	Returns the run-time status of a module
PFID_PumpModule	Calls the processing function of a single module
PFID_ClassLayout_Process	Processes the currently defined layout
PFID_GetFirstObject	Gets information about the first object defined in the memory heaps
PFID_GetNextObject	Returns information about the next object defined in the memory heaps
PFID_GetFirstIO	Returns information about the first Platform I/O pin.
PFID_GetNextIO	Returns information abou the next Platform I/O pin.
PFID_StartAudio	Starts real-time audio processing
PFID_StopAudio	Stops real-time audio processing
PFID_FetchValues	Reads an array of values
PFID_SetValues	Writes an array of values
PFID_GetSizeofInt	Returns the sizeof(int) evaluated on the Target
PFID_GetTargetInfo	Returns information about the Target (number of inputs, outputs, fundamental block size, etc.)
PFID_GetProfileValues	Returns profiling information related to the overall layout

PFID_GetFirstFile	Returns information about the first file in the flash file system
PFID_GetNextFile	Returns information about the next file in the flash file system
PFID_OpenFile	Opens a file for reading or writing
PFID_ReadFile	Reads data from the currently open file
PFID_WriteFile	Writes data to the current open file
PFID_CloseFile	Closes the currently open file
PFID_DeleteFile	Deletes a single file
PFID_ExecuteFile	Executes a compiled script file
PFID_EraseFlash	Erases all of flash memory
PFID_GetFileSystemInfo	Gets information about the flash file system (size, fragmentation, etc.)
PFID_FileSystemReset	Resets the flash file system. It closes any open files.
PFID_GetObjectByIndex	Returns information about the Nth instantiated object defined in the memory heaps
PFID_GetObjectByID	Returns information about the instantiated object defined in the memory heaps based on the ObjectID.
PFID_AddModuleToLayout	Adds one or more modules to an existing layout.
PFID_SetValueCall	This is the common command to set the value and call the set function of the module.
PFID_EnableAddressTranslation	Enables or disables address translation. When address translation is off, then PacketAPI uses absolute addresses. When address translation is on, relative addresses are used.

7.1. PFID_SetCall (ID = 1)

Calls a module's set function. This message maps directly to the framework function

```
int awe_fwSetCall(ModuleInstanceDescriptor *pModule, UINT mask);
```

Received message format:

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

Reply message format:

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

7.2. PFID_GetCall (ID = 2)

Calls a module's get function. This message maps directly to the framework function

```
int awe_fwGetCall(ModuleInstanceDescriptor *pModule, UINT mask);
```

Received message format:

Message Length = 4	ID = PFID_GetCall
Pointer to instance uint mask uint CRC	

Reply message format:

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

7.3. PFID_ClassPin_Constructor (ID = 3)

Creates a pin type object on the target. This message maps directly to the framework function

```
InstanceDescriptor *ClassPin_Constructor(int *retVal,
                    size_t argCount,
                    const Sample *args);
```

with argCount set to 5. Received message format:

Message Length = 7	ID = PFID_ClassPin_Constructor
int blockSize (up to 16383) int numChannels (up to 255) int size of sample, in bytes (up to 255) int sampleRate int isComplex (Boolean, single bit) CRC	

Reply message format:

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

7.4. PFID_GetClassType (ID = 4)

Returns the type of the class this instance was created from. This message maps directly to the framework function

```
UINT awe_fwGetClassType(const InstanceDescriptor *pClass);
```

Received message format:

Message Length = 3	ID = PFID_GetClassType
Pointer to instance CRC	

Reply message format:

Message Length = 3	ID = 0
int ClassType uint CRC	

The ClassType integer is interpreted as follows:

System Input Pin = 0xBEEF0001
System Output Pin = 0xBEEF0002
Pin type = 0xBEEF0003
Layout = 0xBEEF0004
Wire = 0xBEEF0080
InputWire = 0xBEEF0081
OutputWire = 0xBEEF0082

Module classes start at 0xBEEF0800

7.5. PFID_GetPinType (ID = 5)

Returns detailed information regarding a pin type object in the system. This message maps directly to the framework function

```
int awe_fwGetPinType(InstanceDescriptor *pInstance,
        int *numChannels,
        int *numSamples,
        int *size,
        int *sampleRate);
```

Received message format:

Message Length = 3	ID = PFID_GetPinType
Pointer to pin instance CRC	

Reply message format:

Message Length = 7	ID = 0
int error status int numChannels int numSamples int size int sampleRate uint CRC	

7.6. PFID_ClassWire_Constructor (ID = 6)

Instantiates a wire on the target. This call maps directly to the framework function

```
InstanceDescriptor *GenericWire_Constructor(int *retVal,
        UINT bufferSize,
        PinInstanceDescriptor *pPin,
      const ClassDescriptor *pClass);
```

Received message format:

Message Length = 3	ID = PFID_ClassWire_Constructor
Pointer to pin instance CRC	

The wire created is based on an already instantiated pin.

Reply message format:

Message Length = 4	ID = 0
pointer to created wire instance int error status uint CRC	

7.7. PFID_BindIOToWire (ID = 7)

Attaches a wire to an input or output pin of the system. This message maps to the framework function

```
int BindIOToWire(size_t argCount, const Sample *args);
```

Received message format:

Message Length = 4	ID = PFID_BindIOToWire
Pointer to wire instance	
Pointer to an INPUTPIN or OUTPUTPIN object class CRC	

The call also allocates memory for a second data buffer equal in size to the wire's data buffer. The allocated buffer serves as the second half of the double buffer needed to manage the I/O.

Reply message format:

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

7.8. PFID_FetchValue (ID = 8)

Reads a single 32-bit integer value from a specified memory address. Maps directly to the framework function

```
int awe_fwFetchValue(UINT address);
```

Can be used with all 32-bit data types (float, fract32, and int).

Received message format:

Message Length = 3	ID = PFID_FetchValue
int address CRC	

Reply message format:

Message Length = 3	ID = 0
int value CRC	

7.9. PFID_SetValue (ID = 9)

Sets a 32-bit integer value on the target. Maps directly to the framework function:

```
int awe_fwSetValue(UINT address, int value);
```

Can be used with all 32-bit data types (float, fract32, and int). Note, this function does *not* call the module's control function, it only sets the parameter. To set a value and call the module's control function use the call PFID_SetValueCall described in Section 7.48.

Received message format:

Message Length = 4	ID = PFID_SetValue
int address int value CRC	

Reply message format:

Message Length = 3	ID = 0
Error status CRC	

7.10. PFID_GetHeapCount (ID = 10)

Returns the number of available memory heaps on the target. The message maps directly to the framework function

```
size_t awe_fwGetHeapCount(void);
```

Received message format:

Message Length = 2	ID = PFID_GetHeapCount
CRC	

Reply message format:

Message Length = 3	ID = 0
int count CRC	

7.11. PFID_GetHeapSize (ID = 11)

Returns the sizes of all of the memory heaps on the target. The size is reported in units of 32-bit words. The message returns both the overall size and the available memory in each heap. This message maps directly to the framework function:

```
int awe_fwGetHeapSize(int *pHeaps);
```

Received message format:

Message Length = 3	ID = PFID_GetHeapSize
int whichHeap (ignored) CRC	

Reply message format:

Message Length = 12	ID = 0
Error status int heap 1 free space int heap 2 free space int heap 3 free space int heap 1 first free address int heap 2 first free address int heap 3 first free address int heap 1 overall size int heap 2 overall size int heap 3 overall size CRC	

7.12. PFID_Destroy (ID = 12)

Resets the framework to its original state. This includes freeing all allocated memory and halting real-time audio float. This message maps directly to the framework function:

```
void awe_fwDestroy(void);
```

Received message format:

Message Length = 2	ID = PFID_Destroy
CRC	

Reply message format:

Message Length = 2	ID = 0
CRC	

7.13. PFID_GetCIModuleCount (ID = 13)

Returns the total number of module classes (distinct type of audio modules) on the target processor. This message maps directly to the framework function:

```
UINT awe_fwGetCIModuleCount(void);
```

Received message format:

Message Length = 2	ID = PFID_GetCIModuleCount
CRC	

Reply message format:

Message Length = 3	ID = 0
int count CRC	

7.14. PFID_GetCIModuleInfo (ID = 14)

Gets information about a particular module class available on the target. This message maps directly to the framework function:

```
int awe_fwGetCIModuleInfo(size_t index,
        const ModClassModule **pDescr,
        UINT *classID,
        size_t *numParameters);
```

Received message format:

Message Length = 3	ID = PFID_GetCIModuleInfo
int index CRC	

Reply message format:

Message Length = 6	ID = 0
int Error Status int address of class structure int class ID int number of allocation parameters. This is the number of arguments required by the module's constructor function. CRC	

7.15. PFID_ClassModule_Constructor (ID = 15)

Instantiates a single instance of a module class. This message maps directly to the framework function:

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

When a module is allocated, you have to specify the module class (classID), the number of input, output, and scratch wires (packed int nIO), an array of wires (pWires; arranged as input, output, and scratch wires), the number of arguments to the constructor function (argCount).

Received message format:

Message Length = 5 + numWires + argCount	ID = PFID_ClassModule_Constructor
int classID unsigned int nIO int argCount int pointer to wire 1 int pointer to wire 2 … int pointer to wire N int arg 1 int arg 2 … int arg M	

CRC

All of the wires used by a module must be constructed prior to constructing the module. The number of wires used by the module is packed into 8-bit fields within the 32-bit integer nIO:

```
(numFeedback << 24) + (numScratch << 16) + (numOutput << 8) + (numInput)
```

Reply message format:

Message Length = 4	ID = 0
int address of module instance structure int Error Status CRC	

7.16. PFID_ClassLayout_Constructor (ID = 16)

This function creates a new layout instance. The message maps directly to the framework function:

```
InstanceDescriptor *ClassLayout_Constructor(INT32 *retVal,
                                            INT32 nModules,
                                            INT32 nDivider);
```

If the return is zero, there is an error, and retVal will be assigned the reason for the error.

The arguments have meaning as follows:

0: nDivider = clock divider, default is 1
1+ array of nModules pointers (names) of module instances

Received message format:

Message Length = 4	ID = PFID_ClassLayout_Constructor
unsigned long numModules unsigned long divider CRC	

Reply message format:

Message Length = 4	ID = 0
int * layout instance pointer int Error Status	

CRC

7.17. PFID_SetWire (ID = 17)

This function is used primarily for regression testing. It fills data into a wire's data buffer. The message maps directly to the framework function:

```
int awe_fwSetWire(InstanceDescriptor *pWireInst,
        size_t offset,
        size_t argCount,
        const Sample *args);
```

Received message format:

Message Length = 5 + numSamples written	ID = PFID_SetWire
int pointer to wire instance int offset int number of words to write int data sample 0 int data sample 1 … int data sample N-1 CRC	

The data is written into the wire's data buffer starting at the specified offset. It is the responsibility of the caller to ensure that the size of the transmitted message does not exceed the size of the message buffer on the target.

Reply message format:

Message Length = 3	ID = 0
int Error Status CRC	

7.18. PFID_GetWire (ID = 18)

This function is used primarily for regression testing. It reads data from a wire's data buffer. The message maps to the framework function:

```
int awe_fwGetWire(InstanceDescriptor *pWireInst,
        size_t offset,
        size_t argSize,
```

const Sample *args);

Received message format:

Message Length = 5	ID = PFID_GetWire
int pointer to wire instance int offset int number of words to read CRC	

The return message is variable length and it is the responsibility of the caller to ensure that the size of the reply message does not exceed the size of the message buffer on the target.

Reply message format:

Message Length = 3 + number of words read	ID = 0
int Error Status int data sample 0 int data sample 1 … int data sample N-1 CRC	

7.19. PFID_SetModuleState (ID = 19)

Changes the run-time state of an audio module. This message maps directly to the framework function

int awe_fwSetModuleState(ModuleInstanceDescriptor *pModule, int state);

The run-time state of the module is specified by an integer

0	Active
1	Bypassed
2	Muted
3	Inactive

Received message format:

Message Length = 4	ID = PFID_SetModuleState
int pointer to module instance structure int state CRC	

Reply message format:

Message Length = 3	ID = 0
int Error Status CRC	

7.20. PFID_GetModuleState (ID = 20)

Returns the run-time state of an audio module. This message maps directly to the framework function:

```
int awe_fwGetModuleState(ModuleInstanceDescriptor *pModule);
```

The module state is returned as an integer. See Section 7.19 for a listing of allowable states.

Received message format:

Message Length = 3	ID = PFID_GetModuleState
int pointer to module instance structure CRC	

Reply message format:

Message Length = 3	ID = 0
int state CRC	

7.21. PFID_PumpModule (ID = 21)

Calls the processing function once for an audio module instance. This function is primarily used for regression testing. You have to first construct the module, fill the input wires, and then call this function. This message maps directly to the framework function:

int awe_fwPumpModule(ModuleInstanceDescriptor *pModule);

Received message format:

Message Length = 3	ID = PFID_PumpModule
int pointer to module instance structure CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.22. PFID_ClassLayout_Process (ID = 22)

Calls the layout processing function. This executes all modules in the system in the order that they were added to the layout. This message maps directly to the framework function:

```
void ClassLayout_Process(LayoutInstance *pInstance);
```

Received message format:

Message Length = 3	ID = PFID_ClassLayout_Process
int pointer to the layout structure CRC	

Reply message format:

Message Length = 2	ID = 0
CRC	

7.23. PFID_GetFirstObject (ID = 23)

Gets information about the first object that was constructed on the target. This message maps directly to the framework function:

```
int awe_fwGetFirstObject(InstanceDescriptor **pObject, UINT *pClassID);
```

Received message format:

Message Length = 2	ID = PFID_GetFirstObject
CRC	

Reply message format:

Message Length = 5	ID = 0
int error status int instance address int class ID	

CRC

7.24. PFID_GetNextObject (ID = 24)

Gets information about the next object that exists on the target. First call PFID_GetFirstObject to get information about the first object. Then make repeated calls to this function to obtain information about subsequent objects. Once you've reached the last object, this message will fail. This message maps directly to the framework function:

```
int awe_fwGetNextObject(InstanceDescriptor *currentObject,
        InstanceDescriptor **pObject,
        UINT *pClassID);
```

Information is returned about the object following currentObject. Received message format:

Message Length = 3	ID = PFID_GetNextObject
pointer to current object CRC	

Reply message format:

Message Length = 5	ID = 0
int error status int instance address int class ID CRC	

7.25. PFID_GetFirstIO (ID = 25)

Returns a pointer to and the class ID for, the first I/O pin in the system. This message maps directly to the framework function:

```
int awe_fwGetFirstIO(InstanceDescriptor **pObject, UINT *pClassID);
```

Received message format:

Message Length = 2	ID = PFID_GetFirstIO
CRC	

Reply message format:

Message Length = 5	ID = 0
int error status int instance address	

int class ID CRC

The class ID for input pins will be CLASS_ID_INPUTPIN and the class ID for output pins will be CLASS_ID_OUTPUTPIN.

7.26. PFID_GetNextIO (ID = 26)

Returns information about the next I/O pin. This message maps directly to the framework function

```
int awe_fwGetNextIO(InstanceDescriptor *currentObject,
    InstanceDescriptor **pObject,
    UINT *pClassID);
```

Information is returned about the I/O pin following currentObject. Received message format:

Message Length = 3	ID = PFID_GetNextIO
pointer to the current object CRC	

Reply message format:

Message Length = 5	ID = 0
int error status int instance address int class ID CRC	

7.27. PFID_StartAudio (ID = 27)

Starts real-time audio processing on the target. This message maps directly to the platform function

```
int awe_pltAudioStart(void);
```

Received message format:

Message Length = 2	ID = PFID_StartAudio
CRC	

Reply message format:

Message Length = 3	ID = 0

int error status
CRC

7.28. PFID_StopAudio (ID = 28)

Halts real-time audio processing. This message maps directly to the platform function

```
int awe_pltAudioStop(void);
```

Received message format:

Message Length = 2	ID = PFID_StopAudio
CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.29. PFID_FetchValues (ID = 29)

Reads a block of values from the target processor's memory. This message maps directly to the framework function:

```
int awe_fwFetchValues(UINT address,
        size_t offset,
        size_t argSize,
        Sample *args);
```

Received message format:

Message Length = 5	ID = PFID_FetchValues
int address int offset int num words to read CRC	

The return message has variable size. It is the responsibility of the caller to ensure that the message buffer on the target is large enough to hold the returned message.

Reply message format:

Message Length = 3 + numWords	ID = 0
int error status	

int value 0 int value 1 … int value N-1 CRC

7.30. PFID_SetValues (ID = 30)

Writes a block of values in the target processor's memory. This message maps directly to the framework function:

```
int awe_fwSetValues(UINT address,
        size_t offset,
        size_t argCount,
        const Sample *args);
```

The transmitted message is variable length and it is the responsibility of the caller to ensure that the length of the message does not exceed the length of the message buffer on the target.

Received message format:

Message Length = 5 + numWords	ID = PFID_SetValues
int address int offset int num words to write int value 0 int value 1 … int value N-1 CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.31. PFID_GetSizeofInt (ID = 31)

Returns the value sizeof(int) evaluated on the target. This is used by Audio Weaver to determine address offsets. This message maps directly to the framework function:

```
int awe_fwGetSizeofInt();
```

Received message format:

Message Length = 2	ID = PFID_GetSizeofInt
CRC	

Reply message format:

Message Length = 3	ID = 0
int size CRC	

7.32. PFID_GetFirstFile (ID = 32)

Read the first file directory entry from the target flash file system. This message maps directly to the framework function:

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

Received message format:

Message Length = 2	ID = PFID_GetFirstFile
CRC	

Reply message format:

Message Length = 11	ID = 0
int error status unsigned long file attribute unsigned long data word count unsigned long file name …….. …….. CRC	

7.33. PFID_GetNextFile (ID = 33)

Read the next file directory entry from the target flash file system. This message maps directly to the framework function:

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

Received message format:

Message Length = 2	ID = PFID_GetNextFile

CRC

Reply message format:

Message Length = 11	ID = 0
int error status unsigned long file attribute unsigned long data word count unsigned long file name …….. …….. CRC	

7.34. PFID_OpenFile (ID = 34)

Opens a file for reading or creates a new file for writing. Attribute byte must be 0 if opening a file for reading. If file opened for writing and it already exists and is not marked as deleted an error is returned. This message maps directly to the framework function:

```
int awe_fwOpenFile(unsigned long nFileAttribute,
                   unsigned long * pFileNameInDWords,
                   unsigned long  * nFileLenInDWords);
```

Received message format:

Message Length = 3 + numWords	ID = PFID_OpenFile
unsigned long file attribute unsigned long file name …… CRC	

Reply message format:

Message Length = 4	ID = 0
int error status unsigned long file length in words CRC	

7.35. PFID_ReadFile (ID = 35)

Read the indicated number of words from an opened file. The number of words returned can be

less than the number asked for if the end of file is reached. This message maps directly to the framework function:

```
int awe_fwReadFile(unsigned long nWordsToRead,
                   unsigned long * pBuffer,
                   unsigned long  * pDWordsRead);
```

Received message format:

Message Length = 3	ID = PFID_ReadFile
unsigned long no of 32-bit words to read CRC	

Reply message format:

Message Length = 4 + numWords	ID = 0
int error status unsigned long number of words read unsigned long data ….. CRC	

7.36. PFID_WriteFile (ID = 36)

Write the indicated number of words to an opened file. This message maps directly to the framework function:

```
int awe_fwWriteFile(unsigned long nFileAttribute,
                    unsigned long * pFileNameInDWords,
                    unsigned long  * nFileLenInDWords);
```

Received message format:

Message Length = 3 + numWords	ID = PFID_WriteFile
unsigned long file attribute unsigned long file name …… CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.37. PFID_CloseFile (ID = 37)

Closes an opened file and writes the directory entry if file was opened for write. This message maps directly to the framework function:

```
int awe_fwCloseFile();
```

Received message format:

Message Length = 2	ID = PFID_CloseFile
CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.38. PFID_DeleteFile (ID = 38)

Mark a file as deleted. This message maps directly to the framework function:

```
int awe_fwDeleteFile( unsigned long * pFileNameInDWords);
```

Received message format:

Message Length = 3 + numWords	ID = PFID_DeleteFile
unsigned long file attribute unsigned long file name …… CRC	

Reply message format:

Message Length = 3	ID = 0
int error status	

CRC

7.39. PFID_ExecuteFile (ID = 39)

This command is not implemented in current framework.

7.40. PFID_EraseFlash (ID = 40)

Erase the entire Flash file system. This message maps directly to the framework function:

```
int awe_fwEraseFlash();
```

Received message format:

Message Length = 2	ID = PFID_EraseFlash
CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.41. PFID_GetTargetInfo (ID = 41)

Returns infromation about the currently connected target. This message maps directly to the framework function

UINT GetTargetInfo(TargetInfo *pTarget);

This function maps a copy of the TargetInfo data structure which resides on the target. This structure is defined in TargetInfo.h:

```
typedef struct _TargetInfo
{
        float m_sampleRate;                 // 0
        float m_profileClockSpeed;          // 4
        UINT m_base_block_size;             // 8
        UINT m_packedData;                  // 12
        UINT m_version;                     // 16
        UINT m_packedName[2];               // 20
        UINT m_proxy_buffer_size;           // 28
}
TargetInfo;             // size=32 (7 words)
```

Received message format:

Message Length = 2	ID = PFID_GetTargetInfo
CRC	

Reply message format:

Message Length = 11	ID = 0
int error status float sample rate float processor clock speed in Hz uint fundamental I/O block size uint packedData uint versioin information uint packedName 1 uint packedName 2 uint communication buffer size (in words) CRC	

The fundamental I/O block size refers to the size of the input and output audio DMA. All wires connected to the input or output of the system must have a block size that is a multiple of the fundamental size.

packedData contains several pieces of information packed into 32-bits. From the lsb to the msb, the items are:

Size of integers (sizeo(int))	4 bits
isFlashSupported	1 bit
isFloatingPoint	1 bit
Unused	2 bits
numInputChannels	8 bits
numOutputChannels	8 bits
processorType	8 bits

The following values are defined for processor type in Framework.h

#define PROCESSOR_TYPE_NATIVE	1
#define PROCESSOR_TYPE_SHARC	2
#define PROCESSOR_TYPE_BLACKFIN	3
#define PROCESSOR_TYPE_NXPLPCxxx	4

7.42. PFID_GetFileSystemInfo (ID = 42)

Get the flash file system information from the target. This message maps directly to the framework function:

```
int awe_fwGetFileSystemInfo(FileSystemInfo *pFileSystemInfo);
```

Received message format:

Message Length = 2	ID = PFID_GetFileSystemInfo
CRC	

Reply message format:

Message Length = 11	ID = 0
int error status unsigned long file system type unsigned long flash device size unsigned long no of words available for files unsigned long no of words used by the file system dada structures unsigned long words no of allocated to deleted or corrupted files unsigned long no of words in use by files unsigned long no of words available for new files unsigned long allocation block size and max file length CRC	

7.43. PFID_GetProfileValues (ID = 43)

Returns overall MIPs profiling information for a layout. This message maps directly to the framework function

```
int awe_fwGetProfileValues(int layoutNumber,
    float *pAverageCycles,
    float *pTimePerProcess);
```

*pAverageCycles is the average number of clock cycles required to process the entire layout. This is measured in real-time and averaged over approximately 100 executions of the layout. *pTimePerProcess is the amount of time between calls to the layout processing function. This indicates how much total processing is available in the system.

Received message format:

Message Length = 2	ID = PFID_GetProfileValue
CRC	

Reply message format:

Message Length = 5	ID = 0
int error status float averageCycles float timePerProcess CRC	

7.44. PFID_FileSystemReset (ID = 44)

Force the target to close any open files and reset the flash file system variables to default state. This message maps directly to the framework function:

```
int awe_fwResetFileSystem();
```

Received message format:

Message Length = 2	ID = PFID_FileSystemReset
CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.45. PFID_GetObjectByIndex (ID = 45)

This function is complementary to GetFirstObject and GetNextObject. Instead of returning objects in the order that they were instantiated, this function provides direct access to the Nth object. This message maps directly to the framework function:

```
int awe_fwGetObjectByIndex(UINT index,
        InstanceDescriptor **pObject,
        UINT *pClassID);
```

Received message format:

Message Length = 3	ID = PFID_GetObjectByIndex
int index CRC	

Reply message format:

Message Length = 5	ID = 0
int error status	

int instance address int class ID CRC

7.46. PFID_GetObjectByID (ID = 46)

This function is complementary to GetFirstObject and GetNextObject. Instead of returning objects in the order that they were instantiated, this function provides direct access to the object based on its objectID. This message maps directly to the framework function:

```
int awe_fwGetObjectByID(UINT ID,
        InstanceDescriptor **pObject,
        UINT *pClassID);
```

Received message format:

Message Length = 3	ID = PFID_GetObjectByID
int object ID CRC	

Reply message format:

Message Length = 5	ID = 0
int error status int instance address int class ID CRC	

7.47. PFID_AddModuleToLayout (ID = 47)

This function adds the one or more modules to the current layout instantiated. This message maps directly to the framework function:

```
int awe_fwAddModuleToLayout(size_t argCount,
        const Sample *args);
```

Received message format:

Message Length = 2 + argCount	ID = PFID_AddModuleToLayout
int layout instance address int number of modules array of module pointers	

…..
CRC

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.48. PFID_SetValueCall (ID = 48)

This is the single command for PFID_SetValue and PFID_SetCall commands. This message maps directly to the framework functions:

```
int awe_fwSetValue(UINT address, int value);
int awe_fwSetCall(ModuleInstanceDescriptor *pModule, UINT mask);
```

Received message format:

Message Length = 6	ID = PFID_SetValueCall
unsigned long address float value unsigned long pModule unsigned long mask CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.49. PFID_UpdateFirmware (ID = 49)

It sends a command to the target saying that Firmware updating is going to happen. It sets the `UpdateFirmware` FLAG in target.

Received message format:

Message Length = 2	ID = PFID_ UpdateFirmware
CRC	

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.50. PFID_FlashReadOpen (ID = 50)

Returns the Flash Memory Size in words.

Received message format:

Message Length = 2	ID = PFID_ FlashReadOpen
CRC	

Reply message format:

Message Length = 4	ID = 0
int Error Status Flash Memory Size in Words CRC	

7.51. PFID_FlashRead (ID = 51)

Opens a file for writing the Flash content in binary format.This message maps directly to the framework function:

```
int awe_fwFullFlashRead(unsigned long nWordsToRead, unsigned long*  pBuffer,
unsigned long*  pDWordsRead);
```

Received message format:

Message Length = 3	ID = PFID_ FlashRead
int `WordsToRead` CRC	

Reply message format:

Message Length = 4 + numWords	ID = 0
unsigned long number of words read int error status unsigned long data ….. CRC	

7.52. PFID_SetObjectValueCall (ID = 52)

This command simplifies setting object values referenced using object IDs rather than addresses. The message allows single values or multiple contiguous values to be written.

Received message format:

Message Length = 7+numWords	ID = PFID_SetObjectValueCall
UINT32 objectID UINT32 long fieldOffset INT32 arrayOffset UINT32 numValues UINT32 mask INT32 value[0] INT32 value[1] … INT32 value[numValues-1] UINT32 CRC	

where

objectID = ID of the object to write to. This is used to determine the starting address of the object in memory.

fieldOffset = offset from the start of the object structure specifying where the write should occur.

arrayOffset = An optional value which is used to write to arrays.

numValues = number of 32-bit words to write.

mask = bit field which indicates which fields within the object structure changed. This is passed along to the module's set function.

value[] = array of 32-bit values to write.

The call first translates the objectID into an actual pointer (pseudo code only):

```
awe_fwGetObjectByID(objectID, &pObject, *pClassID);
```

If arrayOffset = -1, then a field within the object structure is written as:

```
ptr = ((INT32 *) pObject) + fieldOffset;
for(i=0;i<numValues;i++)
      *ptr++ = value[i];
```

Otherwise, if arrayOffset != -1, then it is assumed that ptr+fieldOffset contains an array pointer. We dereference and set the array value:

```
ptr = *(ptr + arrayOffset);
for(i=0;i<numValues;i++)
            *ptr++ = value[i];
```

After the value is written, the module's set function is called with the specified mask:

```
awe_fwSetCall(pObject, mask);
```

Reply message format:

Message Length = 3	ID = 0
int error status CRC	

7.53. PFID_FetchObjectValueCall (ID = 53)

This command simplifies fetching object values referenced using object IDs rather than addresses.

Received message format:

Message Length = 7	ID = PFID_FetchObjectValueCall
UINT32 objectID UINT32 long fieldOffset INT32 arrayOffset UINT32 numValues UINT32 mask UINT32 CRC	

where

objectID = ID of the object to read from. This is used to determine the starting address of the object in memory.

fieldOffset = offset from the start of the object structure specifying where the read should occur.

arrayOffset = An optional value which is used to read arrays.

numValues = number of 32-bit words to read.

mask = bit field which indicates which fields within the object structure were read. This value is passed to the module's get function.

The call first translates the objectID into an actual pointer (pseudo code only):

```
awe_fwGetObjectByID(objectID, &pObject, *pClassID);
```

Then, the module's set function is called with the specified mask.

```
awe_fwGetCall(pObject, mask);
```

If arrayOffset = -1, then fields within the object structure is read:

```
ptr = ((INT32 *) pObject) + fieldOffset;
for(i=0;i<numValues;i++)
      value[i] = *ptr++;
```

Otherwise, if arrayOffset != -1, then it is assumed that ptr+fieldOffset contains an array pointer. We dereference and set the array value:

```
ptr = *(ptr + arrayOffset);
for(i=0;i<numValues;i++)
      value[i] = *ptr++;
```

Reply message format:

Message Length = 3 + numWords	ID = 0
int error status int value 0 int value 1 … int value numValues-1 CRC	

7.54. PFID_Tick (ID = 54)

This function is used exclusively on the PC to all the module set functions at a rate of 10 Hz. *Not used on any embedded targets.*

7.55. PFID_EnableAddressTranslation (ID = 55)

Enables or disables addresses translation. Address translation occurs locally within the message processing function awe_fwPacketProcess(). When address translation is disabled (the default), addresses sent from the Server to the target processor are unchanged. When address translation is enabled, addresses are sent from the Server as a head ID and offset. The addresses are then translated to physical addresses on the target processor. See Section TBD for a discussion of address translation.

Received message format:

Message Length = 3	ID = PFID_EnableAddressTranslation
int enable uint CRC	

Reply message format:

Message Length = 2	ID = 0
uint CRC	