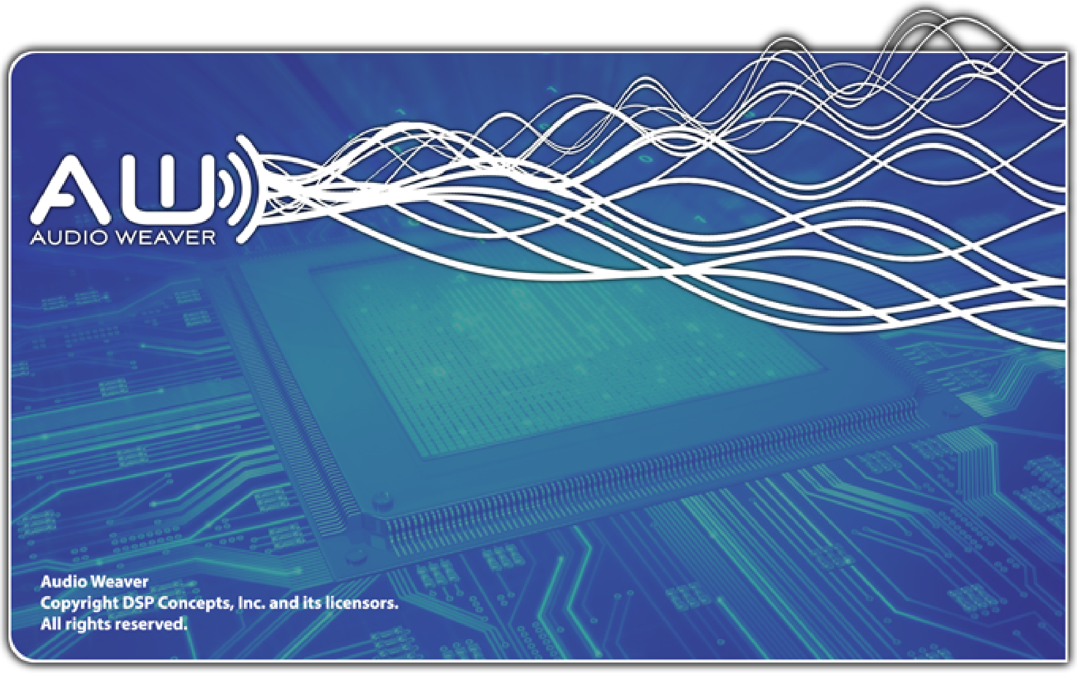
Audio Weaver

Tuning Command Syntax



Version 19 March 2021

Copyright Information

Printed in the USA.

Disclaimer

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

Revision History

Version	Date	Description	Author
01	11/6/17	Changes in the file_logging, create_wire and file_exists commands	PP
02	11/9/17	Changes in the pump and kill_pump	PP
03	11/10/17	Adding server reply info to pump_module and pump_layout	PP
04	11/10/17	Adding sample rate to audio_pump reply. Delete state from set_module_state reply. Changes in command dialog_state.	PP
05	11/13/17	Adding query_wire description. Changes in the foreground command.	PP
06	11/14/17	Adding file_to_pin description. Changes in pin_to_file and rename_pin	PP
07	11/17/17	Command “foreground” supports only server window	PP
08	11/20/17	Added overlooked command descriptions.	DN
09	4/27/18	Added new command “set_core_description'	DN
10	6/19/18	Added new command "clear_credentials"	DN
11	6/21/18	Added document-version to each page (footer) and updated version numbers	MW
12	8/23/18	Added new command 'clear_symbols'.	DN
13	9/7/18	Added new command 'trace'.	DN
14	10/29/18	Fixed copy-paste error.	DWN
15	1/14/19	Added get_rate command.	DWN
16	11/11/19	Added missing PFID_SetValueSetCall.	DWN
16	01/29/20	Updated to support AWECore 8 commands	CHP
17	1/30/20	Minor cleanup, added missing text commands.	DWN
18	5/18/20	Formatting fixes	AN
19	3/10/21	Removed PFID documentation and added link to website	AN
20	9/8/21	Fixed port number typo	DAB

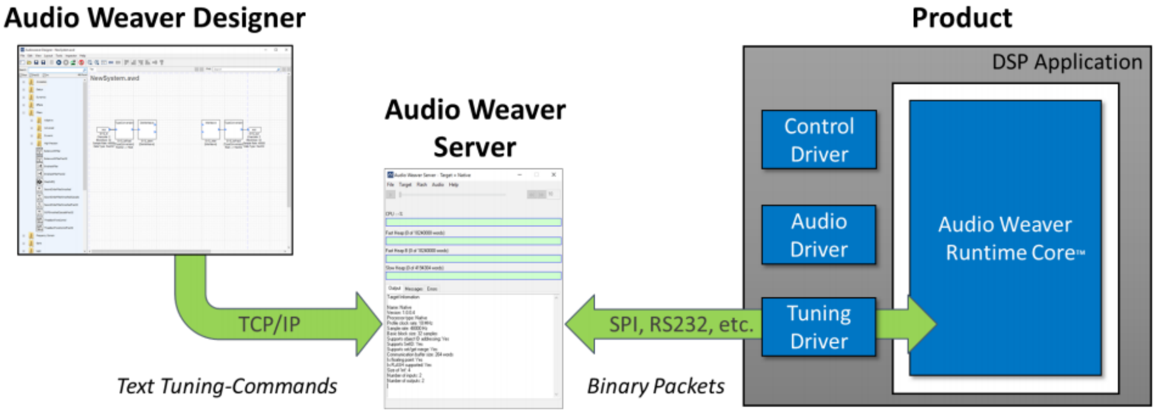
Contents

Definitions, Acronyms, and Abbreviations

API	Application Programming Interface
AWB	Audio Weaver Binary File
AWD	Audio Weaver Design File
AWE	Audio Weaver
BSP	Board Support Package
DLL	Dynamic Link Library
DSP	Digital Signal Processor
GUI	Graphical User Interface
IPC	Inter Process Communication
SMP	Symmetric Multi-processing

1. Overview

As shown below, there are 3 main components of an Audio Weaver system that require communication in order to tune an audio system. The Designer GUI will generate text based tuning commands (AWS) based on the layout and any user tuning commands. These commands are sent to AWE Server, which in turn compiles the AWS commands into binary commands (AWB) that can be sent to the target running the AWE Core. The target processes the AWB commands and then generates a reply message that is sent back to the AWE Server.



The text-based commands may also be generated by external tools and scripts rather than by Designer. An example script, written in python, that connects to the AWE Server at port 15001 and reads and writes a module’s variable in an active layout in real-time is shown below.

```python
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
```

The remainder of 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. All commands are of the form:

```
[coreID,] command_key_word [, argN]
```

in other words, a CSV string. White space is not significant in commands unless within a string value. Any value containing commas (generally only file names) must be quoted with double quotes. Arguments may be redundantly double-quoted.

If present, the numeric coreID specifies the core for which the command is intended. Multicore systems have cores with IDs starting at zero. If no ID is specified, zero is assumed. Single core systems have only one core with ID zero. In the following sections, core IDs are not shown in command descriptions.

Multicore systems have 2 or more cores. On SMP systems (Windows and Linux) you can specify how many cores you want to use. Embedded systems have a fixed number of cores. In both cases, each core is an independent AWE instance sharing system I/O with all other cores. Up to 16 cores may be bound to an input (source) pin, and IPC pins are provided to pass data between cores.

Some commands such as destroy, audio_pump, audio_stop and so on are commands to the system (or BSP), not to any specific core/ Such commands may have a core ID prefix, but it is ignored. These commands are considered to be BSP commands, and are noted as such in the detailed description.

Commands are received on one of 6 sockets:

- 15001 - for Designer use
- 15003 - for Designer GUI use
- 15005 - for special use (advanced)
- 15007 - general purpose
- 15009 - general purpose
- 15011 - general purpose

When using command line server in a user system avoid using the first three sockets so Designer has sockets available for tuning and control. The general purpose sockets are for user use.

Commands marked below as GUI only act only in Windows AWE Server. Those commands report success and are ignored on command line versions.

The reply from all commands is either:

```
success [, args …]
```

or

failed, *reason*

The command keyword is not case sensitive.

All commands may fail for reasons not to do with the command. If the target crashes, or the link goes down, you may get a reply failed,*reason* explaining what failed. Most commonly, these will be:

- failed,heap allocation request too large (ran out of heap in any command)
- failed,bad packet received (target probably crashed)
- failed,message too long (target BSP configuration error likely)
- failed,bad heap pointer (target probably crashed)
- failed,CRC error (link corruption or BSP memory corruption)

- failed,communications error (link corruption or BSP memory corruption)
- failed,message timed out (link failed or target crashed)
- failed,linked list corrupted (target memory corrupted)

Details of all possible target errors are in Errors.h. Most are specifically due to errors in the command. However, obscure cases can arise with any command that can cause any message in Errors.h. All command replies should be tested for failed and appropriate action taken. Never ignore command replies.

The commands are summarized as follows:

add_module	Adds one or more modules to a given layout.
add_symbol_id	Defines a new entry in the symbol table based on the unique ID of the object.
audio_pump	Run the layout using either WAV/MP3 files, or audio line input as the source (BSP)
audio_stop	Stop the audio pump if running (BSP)
bind_wire	Binds a named wire to an input or output pin
close_input	Close an audio device used for combined input, giving zero samples in its place
close_output	Close an audio device used for combined output
cmd	Generic command handler (advanced).
compile	Compiles an AWS script file to AWB binary form (BSP)
connect	Initiates a connection from a client (backward compatibility, ignored, BSP)
clear_credentials	Removes credentials from the INI file
clear_symbols	Empty the server symbol table. (Windows, Linux)
create_active	Creates a control to display 4 radio buttons of module state (BSP, GUI only)
create_bitmap	Creates an image control on a dialog (BSP, GUI only)
create_button	Creates a button on a dialog (BSP, GUI only)
create_checkbox	Creates a checkbox on a dialog (BSP, GUI only)
create_dialog	Creates a named dialog (BSP, GUI only)

create_droplist	Creates a drop list control on a dialog (BSP, GUI only)
create_awslist	Creates a drop list control specifying AWS script files allowing the user to select and run presets from the list (BSP, GUI only)
create_edit	Creates an edit box control on a dialog (BSP, GUI only)
create_filelist	Creates a file list control on a dialog. Used for streaming files to the target (BSP, GUI only)
create_graph	Creates a graph control that represents array elements as bar graphs (BSP, GUI only)
create_grid	Creates a grid control operating as as small spreadsheet for manipulating one or two dimensional arrays (BSP, GUI only)
create_layout	Creates a layout with the specified properties
create_led	Creates an LED style control on a dialog (BSP, GUI only)
create_lookup	Creates an O(1) ID lookup table
create_meter	Creates a meter control on a dialog (BSP, GUI only)
create_module	Creates a module with the specified properties
create_slider	Creates a slider or knob control on a dialog (BSP, GUI only)
create_spline	Creates an X-Y spline control on a dialog (BSP, GUI only)
create_text	Creates a static text control on a dialog (BSP, GUI only)
create_wire	Creates a wire with specified properties
deferred_process	Cause deferred commands to be executed.
delete_file	Deletes a file (BSP)
destroy	Unconditionally destroys all created objects (BSP)
destroy_dialog	Destroys a dialog (BSP, GUI only)
dialog_state	Toggles between normal and expanded views (BSP, GUI only)
end_binary	Stops logging of binary commands sent to the target (BSP)
erase_all	Erases all files in the target file system (BSP)
exists_dialog	Checks if a dialog with a specified name exists (BSP, GUI only)
exit	Causes the server to exit (BSP)
fast_audio_pump	Run a layout using only files at the greatest possible speed. Intended for test

	(BSP)
fast_read	Reads float arrays of binary data (Matlab)
fast_read_int	Reads integer arrays of binary data (Matlab)
fast_write	Writes arrays of binary float data (Matlab)
fast_write_int	Writes arrays of binary integer data (Matlab)
fast_write_partial	Writes arrays of binary float data without performing a set call (Matlab)
fast_write_int_partial	Writes arrays of binary integer data without performing a set call (Matlab)
file_exists	Report if a specified file exists (BSP)
file_logging	Specifies whether to log commands and replies to a file (BSP)
file_to_pin	Bind a WAV file to an input pin as a file player (BSP) THE COMMAND IS NOT IMPLEMENTED.
foreground	Brings all Audio Weaver windows to the foreground thus making them visible (BSP, GUI only)
framework	Change the connected target (BSP)
getini	Returns an entry from the AWE_Server.ini file (BSP)
get_call	Calls the get_call function of a module
get_cores	Returns the number of cores or instances.
get_core_list	Returns the number of cores or instances, and their IDs.
get_executable_dir	Returns the directory containing the AWE_Server.exe executable (BSP)
get_extended_info	Returns the target's extended info.
get_filesystem_info	Returns information about the target file system (BSP)
get_first_core	Return info on the first (and possibly only) core (BSP)
get_first_file	Returns the properties of the first file (BSP)
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 and sizes of the heaps
get_instance_table	Returns the number of cores or instances and their IDs.
get_moduleclass_count	Returns the number of module classes

get_moduleclass_info	Returns information about the specified module class
get_module_state	Returns the muted etc. state of the given module
get_next_core	Return info on cores after the first (BSP)
get_next_file	Returns the properties of the next file, or fails if no more (BSP)
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_byname	Returns the properties of the named object
get_rate	Returns the properties of an audio file
get_schema	Returns the schema for a class (BSP)
get_target_info	Returns the detailed info for an AWE instance.
get_type	Report the type of an expression (int, float, ...) (BSP)
get_value	Returns the value of a symbolically specified location
get_version	Returns the current version number of Audio Weaver (BSP)
gui_logging	Controls whether commands and replies are logged on the server control panel (BSP)
kill_pump	Cause the audio pump to die
make_binary	Starts logging of binary target commands to a named file (BSP)
open_web_page	Launches a browser and displayed a specified page (BSP, GUI only)
pin_to_file	Bind a file writer to an output pin THE COMMAND IS NOT IMPLEMENTED.
pump	Pump the entire framework once
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
query_pump	Test if the audio pump is running
query_wire	Queries a named wire for its properties
read_file	Reads from a file (BSP)
read_float_array	Reads values from an array as floats
read_fract_array	Reads values from an array as fracts reported is floating values

read_int_array	Reads values from an array as integers
reboot_target	Cause an embedded target to reboot. Ignored by Windows and Linux
rename_pin	Change the name of a pin THE COMMAND IS NOT IMPLEMENTED.
reopen_input	Reopen an input device previously closed with close_input
reopen_output	Reopen an output device previously closed with close_output
script	Executes a script file containing commands (BSP)
select_core	Choose which core to report on (BSP, GUI only)
setini	Writes a specified INI file entry with a value (BSP)
set_call	Calls the set_call function of a module
set_core_description	Sets a description file used for target emulation
set_cores	For multicore SMP systems you can specify how many cores to use (BSP)
set_instance_id	Change the ID of some module to the specified ID
set_module_state	Sets a given module to muted, activated, bypass, or disabled
set_path	Set the path to search for audio files used by audio_pump
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 (BSP, GUI only)
target_execute	Cause an embedded target to run an AWB file from its local file system
trace	Cause a target to echo a message to stdout.
update_lookup	Updates the O(1) ID lookup table after IDs have been changed by assignment (legacy support only, ignored)
write_file	Write to a target file system file (BSP)
write_float_array	Writes values to a float array
write_float_array_partial	As write_float_array, but no set call.
write_fract_array	Writes values to a fract array
write_fract_array_partial	As write_fract_array, but no set call.
write_int_array	Writes values to an integer array
write_int_array_partial	As write_int_array, but no set call.

write_pump_read	Write input wire, pump, layout, read output wire. Intended for test

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_id**

Syntax:

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

Adds an 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, 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 layout.

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
```

3.3. **audio_pump (BSP)**

Syntax:

```
audio_pump [, file_name [,record=record_file] ]
```

If *file_name* is given, creates a WAV/MP3 file reader at the rate of the file, otherwise creates a sound card reader. It then creates an output device, 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 output, making a simple player. This capability is for testing.

If the second argument starts with 'record=' then a file is specified to capture output. This file will be a WAV file with as many channels as the output and at the same rate. Warning: this file will grow without bound, and should be used only for limited test purposes.

If the layout specified rate does not match the rates supported in hardware, the nearest supported rate is used.

The reply is one of:

```
Success, sample rate
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 the sound subsystem.

If the server is connected to a target, the *file_name* argument is not permitted. If the file is not found as specified, and does not have an absolute path, it is searched for in the audio path. See **set_path**.

3.4. **audio_stop (BSP)**

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.

3.5. **bind_wire**

Syntax

```
bind_wire,wireName,pinName
```

Causes ***wireName*** to be bound to the *pinName*. It is an error for an output I/O pin to be bound more than once. Input or Output are the default I/O pins. All wire binding is released by ***destroy***.

The reply is:

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

3.6. **clear_credentials (Windows)**

Syntax:

```
clear_credentials
```

Unconditionally removes the user credentials and if it exists the off-line license from the INI file. The command has effect on Windows only and does not involve the target in any way.

The reply is:

```
success
```

3.7. **clear_symbols (Windows, Linux)**

Syntax:

```
clear_symbols
```

Empties the server symbol table. This does not involve the target in any way.

The reply is:

```
success
```

3.8. **close_input (BSP)**

Syntax:

```
close_input,devIndex
```

When several device are merged for audio input (numbered 0 - N-1) close one of these devices by its index, causing zero samples to be generated in its place. This command is a specialized command for users that need to disconnect from a device that is about to be reconfigured. See **reopen_input**.

3.9. **close_output (BSP)**

Syntax:

```
close_output,devIndex
```

When several device are merged for audio output (numbered 0 - N-1) close one of these devices by its index. Samples being written to that device are discarded. This command is a specialized command for users that need to disconnect from a device that is about to be reconfigured. See **reopen_output**,

3.10. **cmd**

Syntax:

```
cmd,opcode [,arg1, ... ,argN]
```

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

opcode – 8-bit command opcode (see ProxyIDs.h)
arg1, … , argN – 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
```

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.

3.11. **compile (BSP)**

Syntax:

```
compile,flags,source_file,destination_file [, target_buffer_size ]
```

Instructs the server to compile ***source_file*** which must be a file containing valid commands from this document (generally an AWS file created by Designer) into ***destination_file*** in AWB binary format.

If ***target_buffer_size*** is not specified, then 264 is assumed. Commands will be compiled so as to fit in the specified command buffer. It is important when compiling for a target with a non-default buffer size to specify the buffer size to use. Note that current AWE builds have a buffer size of 4105. The command will fail if the buffer size specified is outside the range 16-4105.

The ***flags*** argument is no longer used. It is retained for backwarsd compatability, supply 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.

3.12. connect (BSP)

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. The reply is:

```
success,client_name,port
```

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

This command is for legacy purposes only, and does nothing.

3.13. create_active (BSP, GUI only)

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.

3.14. create_bitmap (BSP, GUI only)

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.

3.15. create_button (BSP, GUI only)

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.

3.16. create_checkbox (BSP, GUI only)

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.

3.17. create_dialog (BSP, GUI only)

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***.

3.18. create_droplist (BSP, GUI only)

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.

3.19. create_awslist (BSP, GUI only)

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.

3.20. create_edit (BSP, GUI only)

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

3.21. create_filelist (BSP, GUI only)

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'

3.22. create_graph (BSP, GUI only)

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.

3.23. create_grid (BSP, GUI only)

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.

3.24. 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=instanceID
```

3.25. create_led (BSP, GUI only)

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.

3.26. 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. If no table is specified, lookups are O(N/2).

3.27. update_lookup

Syntax:

```
update_lookup
```

This command is for legacy purposes, and does nothing.

3.28. create_meter (BSP, GUI only)

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

3.29. **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 creates 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=instanceID
```

3.30. create_slider (BSP, GUI only)

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.

3.31. **create_spline (BSP, GUI only)**

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.

3.32. create_text (BSP, GUI only)

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.

3.33. create_wire

Syntax:

```
create_wire,wireName,sampleRate,nChannels,blockSize,complex,maxBlockSize
```

where:

wireName must be an identifier not currently defined,
sampleRate is the effective sample rate of the wire
nChannels is the number of wire channels
blockSize is the number of samples in the wire
complex is non-zero if the wire samples are complex
maxBlockSize must be the same as blockSize

This creates a wire object named *wireName* with the properties specified.

On success, the reply is:

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

3.34. deferred_process

Syntax:

```
deferred_process
```

During operation, a number of audio modules may set up actions to be performed later. These are referred to as delayed actions. This command causes one pass through the object list executing these actions. It is only needed for test, since normal audio pumping calls this on every pump cycle.

3.35. delete_file (BSP)

Syntax:

```
delete_file,filename
```

This command deletes the specified file from the 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 FLASH 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 FLASH files will consume all storage eventually. You can return the file system to its initial state with **erase_all**.

3.36. destroy (BSP)

Syntax:

```
destroy
```

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

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

3.37. destroy_dialog (BSP, GUI only)

Syntax:

```
destroy_dialog,dialog
```

where:

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

This command destroys the named dialog.

3.38. dialog_state (BSP, GUI only)

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.

On success, the reply is: `success`

3.39. erase_all (BSP)

Syntax:

```
erase_all
```

This command erases all files on the target 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.

3.40. end_binary (BSP)

Syntax:

```
end_binary
```

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

3.41. exists_dialog (BSP, GUI only)

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)
```

3.42. exit (BSP)

Syntax:

```
exit
```

This command destroys the server.

3.43. fast_audio_pump (BSP)

Syntax:

```
fast_audio_pump,input_file [ ,record_file ]
```

Plays the input file through the layout as fast as possible, optionally recording the layout output in a WAV file. The output file will be at whatever rate the layout specifies, and have as many channels as the output wire.

The command does not complete until all input samples have been processed. With long files, this may take a while.

3.44. fast_read (Matlab)

Syntax:

```
fast_read,expression,count
```

Reads the specified number of float 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.

3.45. fast_read_int (Matlab)

Syntax:

```
fast_read_int,expression,count
```

Reads the specified number of integer 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.

3.46. fast_write (Matlab)

Syntax:

```
fast_write,expression
```

Writes the float array passed 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.

3.47. fast_write_int (Matlab)

Syntax:

```
fast_write_int,expression
```

Writes the integer array passed 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.

3.48. fast_write_partial (Matlab)

Syntax:

```
fast_write_partial,expression
```

Writes the float array passed 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.

The final set call performed by **fast_write** is suppressed.

3.49. fast_write_int_partial (Matlab)

Syntax:

```
fast_write_int_partial,expression
```

Writes the integer array passed 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.

The final set call performed by **fast_write_int** is suppressed.

3.50. file_exists (BSP)

Syntax:

```
file_exists,file
```

Tests to see if the specified file or directory can be found.

This command either succeeds or fails. The possible failures are:

```
* failed,argument count    // command takes only one argument
* failed,no target    // command requires a connected target
```

If it succeeds the possible replies:

```
* success,0,    // name not found in path
* success,1,full_path    // found this file in the path
* success,2,full_path    // found this directory in the path
```

If found, the full_path will be in the convention of the OS on which the server is running. The only supported OSs are Windows and Linux.

If a server is connected to a target such as a SHARC, the search takes place in the server file system.

The path to search is specified in the server INI file. The implied '.' (server working directory - always the directory containing the server executable binary) is always searched first. Due to this if files are placed there, the search path need not be specified.

The search order is that specified by the search path always looking in '.' first. If there are multiple hits, the first match is reported.

3.51. file_logging (BSP)

Syntax:

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

The first form starts logging all commands and replies by appending them to the given *filename*. The second form starts logging all replies to the given *filename*. The last form turns of logging to file. The path to filename is specified in the server INI 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
```

3.52. file_to_pin (BSP) NOT IMPLEMENTED

Syntax:

```
File_to_pin,pin_name,file_path
```

Binds a file to an input pin to act as a fake audio device. Only possible with non-DMA pins not otherwise in use.

On success, the reply is:

```
success
```

Following are the possible failure replies:

```
* failed, argument count        // needs 2 arguments
* failed, no target             // must be connected
* failed, duplicate pin name    // can't bind the same pin more than once
* failed, name undefined        // pin name not defined
* failed, not a pin             // pin named is not a pin
* failed, can't be Input        // can't bind to pin Input
* failed, not an input pin      // pin must be an input
* failed,' not in core %d       // pin name not in the core
* failed,' not a public pin     // only public pins can be bound
```

3.53. foreground (BSP, GUI only)

Syntax:

```
foreground
```

Brings Audio Weaver Server window to the foreground (top most in Z-order). Make the window visible if it were behind other windows.

On success, the reply is:

```
success
```

3.54. framework (BSP)

Syntax:

```
framework,proxyIndex
```

Changes the proxy the server uses to that specified by the index. If no target can be found on that proxy, switches back to prior proxy. The target previously connected remains in the state it was in when the connection changes. Various INI file settings for proxies are used - there are no arguments to specify for example the IP address of an Ethernet target.

The available proxies are:

0 - Native
1 - RS232
2 - USB
3 - Ethernet
4 - SPI
8 - FTDI
9 - TotalPhaseSPI

Only those proxies enabled in the INI file can be specified. Linux supports only the Ethernet proxy. The default proxy is Native, and is always available.

Replies:

```
failed,framework %d is current framework
failed,no such framework as %d
failed,can't create framework %d: old connection %d restored
success
```

3.55. getini

Syntax:

getini,*section,key*

Reads the AWE server INI file.

Replies:

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

A return value of '0,no such key' indicates the INI file does not contain the specified key.

3.56. get_call

Syntax:

get_call,*module_name,mask*

Performs a get call on the specified module with the given mask. See module documentation for specifics. If the module has no get function, does nothing.

Replies:

success,*module_name=instanceID*

3.57. get_filesystem_info (BSP)

Syntax:

get_filesystem_info

This command queries the target 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**.

3.58. get_first_core (BSP)

Syntax:

```
get_first_core
```

Returns the target info for the first core in the system which should have ID zero. See **get_target_info**.

3.59. get_next_core (BSP)

Syntax:

```
get_next_core,prevCoreID
```

Returns the target info for the core following prevCoreID in the system. If there is no successor core, it fails. See **get_target_info**.

3.60. get_cores (BSP)

Syntax:

```
get_cores
```

Returns the number of cores (or instances) defined by the target.

```
success,numberOfCores
```

3.61. get_core_list (BSP)

Syntax:

```
get_core_list
```

Returns the number of cores (or instances) defined by the target, and all their IDs. See also get_instance_table below which is an alias.

```
success,number of cores, <list of core IDs>
```

3.62. get_instance_table (BSP)

Syntax:

```
get_instance_table
```

Returns the number of cores (or instances) defined by the target, and all their IDs. This is an alias for get_core_list above.

```
success,number of cores, <list of core IDs>
```

3.63. get_first_file (BSP)

Syntax:

```
get_first_file
```

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

```
success,1,attributes,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.

3.64. get_next_file (BSP)

Syntax:

```
get_next_file
```

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

```
success,1,attributes,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.

3.65. **read_file (BSP)**

Syntax:

```
read_file,filename
```

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

3.66. **reopen_input (BSP)**

Syntax:

```
reopen_input,devIndex
```

Reopens a device previously closed by **close_input**. The same parameters are used as when the device was originally opened when audio started. It is an error to reopen an already open device, or use an index that does not exist.

This command is intended for rare cases where a device has to be reconfigured, but can't be while AWE has an open handle on it.

3.67. **reopen_output (BSP)**

Syntax:

```
reopen_output,devIndex
```

Reopens a device previously closed by **close_output**. The same parameters are used as when the device was originally opened when audio started. It is an error to reopen an already open device, or use an index that does not exist.

This command is intended for rare cases where a device has to be reconfigured, but can't be while AWE has an open handle on it.

3.68. **write_file (BSP)**

Syntax:

write_file,*filename*,*attribute*

This command writes the specified local hard disk file to the target 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 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 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.

3.69. **get_extended_info**

Syntax:

```
get_exterded_info
```

This command returns user version followed by 12 undefined words

```
success,user_version,<12 undefined words>
```

3.70. get_first_io

Syntax:

```
get_first_io
```

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

```
success,Input,1,1,48000,1074003968,256,4194304,
```

The format is

success,*instance_name,instanceID,boundID,sampleRate,info1,info2,info3,*

boundID is zero if the pin is not bound, othereise the ID of the bound wire.
info1 is a packed bitfield of 10 bit channels | 17 bits blockSize | 4 bit sample size bytes | 1 bit complex
info2 is a packed bitfield of 17 bits blockSize | 6 bits data type
info3 is a packed bitfield of 10 bits rows | 10 bits cols | 1 bit isIPC | 1 bit isPrivate | 1 bit clock master | 1 bit special

For more information on these fields see Framework.h.

See **get_next_io**.

3.71. get_first_object

Syntax:

```
get_first_object
get_first_object,1
```

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

success,*instanceName*=*instanceID*,Class=*className*

The second form reply is:

success,*instanceName*=*instanceID*,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.

3.72. **get_heap_count**

Syntax:

```
get_heap_count
```

This command returns:

```
success, number_of_heaps
```

Currently this value is always 3.

3.73. **get_heap_size**

Syntax:

```
get_heap_size
```

On success the reply is:

```
success, free1,free2,free3,size1,size2,size3
```

where:

freeN - is the number words available in heap N.
sizeN – is the total size of heap N

All sizes are in 32-bit words.

3.74. **get_executable_dir (BSP)**

Syntax:

```
get_executable_dir
```

Returns the directory containing the current AWE_Server.exe executable.

Reply:

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

3.75. 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=instanceID,state
```

where:

module_instance_name is the argument of the command,
instanceID is the ID 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**.

3.76. 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.

3.77. 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,classID,nParams,DLLName
```

where:

className is the name of the class as it appears in the schema file,
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.
DLLname is the library the module is found in.

3.78. 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.

3.79. get_next_object

Syntax:

```
get_next_object
get_next_object,1
```

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

```
failed, no more objects
```

if there are no more objects to enumerate.

if there are no more objects to enumerate.

3.80. **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
```

3.81. **get_rate**

Syntax:

```
get_rate,filename
```

This command opens an audio file and returns its rate, channels, and duration. On Windows it can open any file type for which a codec is installed in the OS (as a minimum always supports MP3). On Linux it supports only WAV files. The file is closed after reading its properties.

The file is searched for in the audio path unless an absolute path is specified.

On success the reply is:

```
success,rate,channels,duration
```

The command fails if the file can’t be found or if the format is not a supported audio file. The reported duration is in seconds as floating point.

3.82. get_schema (BSP)

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.

3.83. get_target_info

Syntax:

```
get_target_info
```

Reply:

```
success,sampleRate,profileFreq,packetBufLen,nCores,nThreads,nInputs,nOutputs,baseBlockSize,packedInfo,version,CpuType,targetName,proxyName,CpuFreq,instanceID,isSMP,nInputPins,nOutputPins,featureBits
```

Where:

sampleRate - target sample rate in Hz
profileFreq - target profiling clock frequency in Hz
packetBufferLen - the size in words of the target's packet buffer
nCores - the number of cores the target currently has. For SMP targets the default 2. nCores can be changed using the set_cores command.
isSMP – flag indicating if core supports SMP
nInputPins – number of input pins
nOutputPins – number of output pins

baseBlockSize - the base block size for audio. Layouts must use an integer multiple of this
nThreads - the number of concurrent threads per core. Typically 2 for embedded targets and 4 for SMP targets.
nInputs - the number of input channels
nOutputs - the number of output channels
packedInfo – 7 bits CPU type | 8 bits output channels | 8 bits input channels | 1 bit floating point | 1 bit file system | 4 bits sizeof(int)
version - 32 bit version number usually expressed as w.x.y.z where each field is a byte of the word most significant first
targetName - an up to 8 character name for the target
CpuFreq - the frequency of the target clock in Hz
instanceID - the ID of this core in the range ((0…15) * 16) i.e. (0, 16, 32, …, 240)
isSMP - 1 if the core is SMP (Windows or Linux)
featureBits - usually 0, reserved for DSP Concepts internal use

3.84. **get_type (BSP)**

Syntax:

get_type,*expression*

Evaluates the type of *expression* and returns the type as an integer if the expression is legal. The possible reported values are:

0 - integer
1 - unsigned integer
2 - float
3 - fract
4 - object

5 - pointer to integer
6 - pointer to unsigned integer
7 - pointer to float
8 - pointer to fract
9 - pointer to object

3.85. **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*
**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 4 scalar forms, then more members must be named to complete the expression. This continues iteratively until the expression reaches one of the first 4 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'
```

3.86. get_version (BSP)

Syntax:

```
get_version
```

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

```
success,,Jul 12 2017 14:10:34
```

where the first value is empty, and the rest of the string is the build date and time.

3.87. gui_logging (BSP)

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.

3.88. **kill_pump (BSP)**

Syntax:

```
kill_pump
```

This command is only supported on Windows and Linux server in Native. It causes the audio pump thread to terminate. The error message “Server response: "failed,not playing" will appear if audio pump thread doesn’t exist. Otherwise it will report success. It is intended only for test.

3.89. **make_binary (BSP)**

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_wire
destroy
set_module_state
set_value
write_float_array
write_fract_array
write_int_array
```

3.90. **open_web_page (BSP, GUI only)**

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.

3.91. **pin_to_file (NOT IMPLEMENTED)**

Syntax:

```
pin_to_file,pin_name,file_name
```

Binds a file to an output pin to act as a fake audio device. Only possible with non-DMA pins not otherwise in use. Arranges that samples written to pin_name will be written to the file. The file will have the rate and number of channels of the pin - which really means of the wire connected to the pin. The file will be created each time audio starts.

The command will fail if the pin is not an output pin, if the special pin Output is used, or if a file is already bound to the pin.

```
* failed, argument count        // needs 2 arguments
* failed, no target             // must be connected
* failed, duplicate pin name    // can't bind the same pin more than once
* failed, name undefined        // pin name not defined
* failed, not a pin             // pin named is not a pin
* failed, can't be Output       // can't bind to pin Output
* failed, not an output pin     // pin must be an output
* failed, not in core %d        // pin name not in the core
* failed, not a public pin      // only public pins can be bound
```

3.92. **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 error message is:

```
Server response:  "failed,no layout(s) to pump"
```

If server is not connected to the target, the error message is:

```
Server response:  "failed,not connected to the target"
```

otherwise the replay is:

```
success, cycles, cycles interval
```

This call is intended to be used with the command **write_pump_read** for testing. See also **fast_write** and **fast_read**.

3.93. pump_layout

Syntax:

pump_layout,*layout_instance_name [,pump cycles]*

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 writing an input wire and reading an output wire for testing. See also **fast_write** and **fast_read**.

On success the reply is

```
success, instance_name=instanceID [, pump cycles]
```

Possible replies from server in case of failure:

```
-   failed,argument count                  // must be one [or two] arguments only
-   failed,no target                       // must be connected to target
-   failed,instance name not identifier    // must be identifier
-   failed,name undefined                  // layout name must be defined
```

3.94. 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 writing an input wire and reading an output wire for testing. See also **fast_write** and **fast_read**.

On success the reply is

```
success,Name=instanceID
```

Possible replies from server in case of failure:

```
-    failed,argument count;                  // must be only one argument
-    failed,no target;                       // must be connected to target
-    failed,instance name not identifier;    // must be identifier
-    failed,name undefined;                  // module name must be defined
```

3.95. query_pin

Syntax:

query_pin,*pin_name*

where *pin_name* is any of the pins on the target.

On success, the reply is as described in **get_first_io**.

3.96. query_pump (BSP)

Syntax:

```
query_pump
```

This command reports the audio pump status as an integer. The possible values are:

0 - target has no layout, not pumping
1 - target has a layout, not pumping
2 - target has no layout, pumping
3 - target has a layout, pumping

This command is intended for use by an external monitoring process which watches for the pump dying by transitioning from 3 to 1, which usually indicates an I/O error on audio hardware which can only happen on Linux systems.

3.97. query_wire

Syntax:

```
query_wire,wireNname
```

where wireName must be an object created by **create_wire**.

On success the reply is

```
success,wireName=objectID,sampleRate,info1,info2
```

Possible replies from server in case of failure:

```
-    failed,argument count                   // must have 1 argument
-    failed,no target                        // must be connected to a target
-    failed,instance name not identifier     // name must be identifier
-    failed,instance name undefined          // name must be defined
```

- failed,instance name is not a wire // must name a wire

3.98. read_float_array

Syntax:

```
read_float_array,expression,count
```

where:

expression is an expression that evaluates to an array element, so must be subscripted
count is the number of values to read starting at that element

The reply is

```
success,val[0], …, val[count-1]
```

where each value is formatted using %g.

There is no bounds checking - elements past the end of the array will report junk.

3.99. read_fract_array

Syntax:

```
read_fract_array,expression,count
```

where:

expression is an expression that evaluates to an array element, so must be subscripted
count is the number of values to read

The reply is

```
success,val[0], …, val[count-1]
```

where each value is reported as float interpreting each value as fract32, and so constrianed to report a value in the range -1 to 1.

There is no bounds checking - elements past the end of the array will report junk.

3.100. read_int_array

Syntax:

```
read_int_array,expression,count
```

where:

expression is an expression that evaluates to an array element, so must be subscripted
count is the number of values to read

The reply is

success,*val[0]*, …, *val[count-1]*

where each value is formatted using %d.

There is no bounds checking - elements past the end of the array will report junk.

3.101. **reboot_target (BSP)**

Syntax:

reboot_target

Causes an embedded target to reboot as though by reset or power cycle. Not implemented for Windows or Linux server.

3.102. **rename_pin (BSP) (NOT IMPLEMENTED)**

Syntax:

```
rename_pin,oldPinName,newPinName
```

This command renames a pin that only has a default name. You can't rename Input or Output (the default I/O pins), or any pin that was given a specific name by a BSP.

Reply:

```
success,
```

Following are the possible failures:

```
* failed, argument count          // needs 2 arguments
* failed, no target               // must be connected
* failed, too long                // new names must be <= 8 characters long
* failed, name already used       // new name is already defined
* failed, name undefined          // old name must be defined
* failed, not a pin               // old name is not a pin
* failed, not a public pin        // old name is not public
* failed, pin is not IPC          // can't rename DMA pins
```

After renaming, you must refer to the pin by its new name.

3.103. **script (BSP)**

Syntax:

```
script,fileName
```

This command executes the commands stored in *fileName*. Generally, these files have an AWS extension, and are generated by Designer.

3.104. **setini (BSP)**

Syntax:

```
setini,section,key,value
```

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

```
[section]
key=value
```

3.105. **set_call**

Syntax:

```
set_call,module_name,mask
```

This command calls the set function of a module. If the module has no set function, nothing happens, See module documentation for specific details. On success the reply is:

```
success,module_name=address
```

3.106. **select_core (BSP, Windows)**

Syntax:

```
select_core,N
```

For Windows only, cause the server GUI to display the data for the specified core. The allowed range of N is 1 through cores where cores is as reported by get_cores.

3.107. **set_core_description(BSP, Windows, Linux)**

Syntax:

```
set_core_description,file
```

File may be one of:

- SMP
- smp
- none

in which case it forces the PC to be SMP Native. Otherwise, it is a text file containing the description of the target to emulate. If so, the current system is completely destroyed, an emulation is created specified by the file, and the server UI is updated to show the emulation details. The emulation will not support set_cores,

If the description file has errors, the offending error is reported by **failed,parse error**, and the detailed error will be in awelog.txt. It is outside the scope of this document to describe the description file format and the possible parse errors which are legion. On error, the existing system is unchanged.

Any attempt to set the description file currently in use reports **success,no change**. If the command succeeded in loading a new file (or reverting to SMP) the reply is **success**.

3.108. **set_cores (BSP, WIndows, Linux)**

Syntax:

```
set_cores,N
```

For SMP targets only, destroys all existing core objects, and creates N new ones. The allowed range is 1-16. The number of cores on server start is 2. The current number of cores can be found using get_cores. Embedded targets with multiple cores have a fixed number of physical cares and do not implement this command.

Warming: it is possible to create more core instances than the target has. Be aware that cores are simulated in the BSP by threads, and spawning more threads than physical core count - 1 can exceed the CPU bandwidth of the machine. Also be aware that Intel HyperThreading reports twice as many cores as physical cores, but the actual CPU bandwidth is not that high - you can end up with less computation than this number would lead you to expect.

3.109. **set_instance_id**

Syntax:

set_instance_id,*object_name*,*id*

As objects are created, they are assigned IDs starting at 1. If a layout needs to assign a new ID, it must be >= 30000, and not in use by another object. The *object_name* must be that of an existing object.

3.110. 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=instanceID
```

See also **get_module_state**.

3.111. set_path (BSP)

Syntax:

```
set_path,path_item1 [, path_item2 ... ,path_itemN ]
```

This command takes any number of arguments, each of which is a file system path. It is used to set the search path for audio files specified to **audio_pump**. The paths are persisted in the server INI file as:

```
{AudioPath]
Path=item1| ... |itemN
```

When **audio_pump** is searching for a file, it works through this list in order, and uses the first match found.

3.112. set_pointer

Syntax:

```
set_pointer,destination_expression,pointer_expression
```

This command assigns the address of the pointer_expression to the location destination_expression which must be a pointer value. On success the reply is:

```
success
```

3.113. set_timeout (Matlab)

Syntax:

```
set_timeout,N
```

where:

N is the time out in milliseconds

Sets the communication time out between Matlab and the server. By default, the value is 4000, or 4 seconds. (This command is useful to prevent issues when you issue a command, such as **erase_all**, which may take a long time to execute.)

On success the reply is:

```
success
```

3.114. 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 function of each unique module instance (if any) referenced by any of the *expression*s are called.

On success the reply is:

```
success,instanceID,type,value [,instanceID,type,value]*
```

3.115. show (BSP, GUI only)

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.

3.116. target_execute (BSP)

Syntax:

```
target_execute,file
```

Causes an embedded target to load the specified file from the local file system which must be in AWB format. An AWB file contains a sequence of binary commands which are usually those for constructing a layout. The command is not implemented on Windows or Linux server - see binary and script commands for server specific file loading from native file systems.

The command will only work on those targets that have a local file system (such as flash) implemented by AWE into which AWB files have been stored. Targets with no local file system do not implement the command. Do not confuse local file systems implemented by AWE with native file systems such as Windows or Linux implemented by an operating system.

Loading may fail for a large number of reasons even if the file exists. Causes may range from the AWB file referring to cores the target does not have, to the file referring to audio modules not present in the target code, to lack of storage for the layout.

3.117. trace (BSP, Linux)

Syntax:

trace,*message*

Causes a Linux target to write to stdout the message. The message must be double quoted if it contains space or comma. The reply is always:

```
success
```

on all targets.

3.118. write_float_array

Syntax:

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

where:

expression is an expression that evaluates to an array element, and so must be subscripted

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

```
success
```

Misuse of the command can corrupt storage, because there is no bounds checking for arrays.

3.119. write_float_aray_partial

Syntax:

write_float_array_partial,*expression*,*val0*,….,*valN-1*

As **write_float_aray** except the final set call is suppressed.

3.120. write_fract_array

Syntax:

write_fract_array,*expression*,*val0*,….,*valN-1*

where:

expression is an expression that evaluates to an array element, and so must be subscripted

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

success

Misuse of the command can corrupt storage, because there is no bounds checking for arrays.

3.121. write_fract_array_partial

Syntax:

write_fract_array_partial,*expression*,*val0*,….,*valN-1*

As **write_fract_aray** except the final set call is suppressed.

3.122. write_int_array

Syntax:

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

where:

expression is an expression that evaluates to an array element, and must be subscripted

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

success

Misuse of the command can corrupt storage, because there is no bounds checking for arrays.

3.123. **write_int_array_partial**

Syntax:

```
write_int_array_partial,expression,val0,…,valN-1
```

As **write_int_aray** except the final set call is suppressed.

3.124. **write_pump_read**

Syntax:

```
write_pump_read,layout,inputWire,outputWire,val1, ... ,valN
```

This command writes the val***i*** to *inputWire*, then pumps the *layout*, then reads *outputWire* and replies:

```
success,cycles,val1, ... ,valN
```

Layout may be either the index of a layout (zero based), or a layout name.

Cycles is the number of profile cycles the pump call took.

The input and output samples are raw values on the wire. The number provided should be the same as channels*blockSize. For example, if the wire is stereo 32 long, then 64 values must be provided in interleaved form. The assumption is input and output wire are the same, and so the number of output samples is the same as the number of input samples.

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, awe_fwMalloc no more storage	The given heap does not have enough storage to satisfy the requested size
failed, awe_fwMallocScratch 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 add_module
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	An expression expected to be of pointer type is not
failed, not input pin	The pin must 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 bind_wire
failed, pin types not compatible	An attempt was made using bind_wire to bind a wire incompatible with the I/O pin
failed, pin sizes not compatible	An attempt was made using bind_wire to bind a wire not an integer multiple of the I/O pin size
failed, not output pin	The pin must 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, not implemented on target	The target does not implement the command
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, expression error	The 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 add_module is undefined
failed, '%s' is not a module	A supposed module name given to add_module 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	'audio_pump' could not open the sound card for input

error	
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 output 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 failed
failed, '%s' requires subscript	An 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 (static arrays only, very rare)
failed, '%s' requires dot expression	An 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
failed, no such core	A core ID was specified that the target does not have.
failed, too many bound wires	An attempt was made to bind more than 17 times to an input pin

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
    ….
    memberN          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. Internal Binary packets

There are occasions when text commands are too burdensome on bandwidth. 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. */
        unsigned int m_len;

        /** Length of data in floats. */
        unsigned int 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 and External Binary Packets

The default target packet buffer is 4105 words - enough for the largest command header plus 4096 argument values. Targets are free to define the packet buffer as small as 16 words with a greatly increased transport overhead. A compromise used on many targets is 264 words - enough for the largest command header plus 256 argument values.

Binary packets have the form:

word 0: 16 bit length | 8 bit core ID | 8 bit opcode
word 1 - N-2 command payload
word N-1: XOR sum of all preceding words

Reply packets have the form:

word 0: 16 bit length in words | 16 bit zero
word 1 - N-2 reply payload
word N-1: XOR sum of all preceding words

Commands may have no payload. Replies always have at least one payload word, and for almost all of them word 1 is the return value - frequently the error code. Many replies only have this one payload word.

For more about binary packets, including a detailed list of available commands, see the ‘Tuning Command Syntax and Protocol’ section of the AWECore Integration Guide at this location: https://w.dspconcepts.com/docs.