Studio Components Reference Guide

Revision S
Version 4
January 2024
Document DOC-TEL-ASC-RM-S-4

Advanced Simulation Technology inc.
500A Huntmar Park Drive ▪ Herndon, Virginia 20170 USA
(703) 471-2104 ▪ asti-usa.com

Studio Components Reference Guide

ASTi
500A Huntmar Park Drive
Herndon, Virginia 20170 USA

Red Hat Enterprise Linux (RHEL) Subscriptions

ASTi is an official Red Hat Embedded Partner. ASTi-provided products based on RHEL include Red Hat software integrated with ASTi's installation. ASTi includes a Red Hat subscription with every purchase of our Software and Information Assurance (SW/IA) maintenance products. Systems with active maintenance receive Red Hat software updates and support directly from ASTi.

Export Restriction

Countries other than the United States may restrict the import, use, or export of software that contains encryption technology. By installing this software, you agree that you shall be solely responsible for compliance with any such import, use, or export restrictions. For full details on Red Hat export restrictions, go to the following:

www.redhat.com/en/about/export-control-product-matrix

Revision history

Date	Revision	Version	Comments
2/26/2009	Preliminary	0	(4.14) New components include **ByteToBit**, **ByteMerger**, **CellIn**, **CellOut**, **VibrationCapture**, and **IntercomTransceiver**. Updated components include **CommPanel** components for optimization. Fixed components include **PFilter** and **AutoDRED** for gain defaults.
3/13/2009	Preliminary	1	(4.15) New components include **Demux**, **Envelope**, **PEnvelope**, **Incrementer**, FDACP (for a specific program).
4/27/2009	Preliminary	2	(4.16) New components include **Compressor**, internal testing components, and program-specific components.
6/30/2009	A	0	(4.18) New components include **Audio**/**Delay**, **MessageList** updated and moved to **Audio** group, **ACU2channel**, **RadiusChannel**, **Satellite**, and program-specific components.
8/04/2009	B	0	(4.19) New components include **NoiseSource**, **TextToSpeech**, and program-specific components. Updated/fixed components included **AutoDRED**, **MessageList**, **RecordReplay**, **Satellite**, **Transceiver**, and **Relay**.
9/19/2009	B	1	(4.20) New components include **5BandFilter**, **MultiFilter**, **ACE_RIU_SerialByteOut**, **ACU2_SerialByteOut**, and **VoisusChannel**. Updated/fixed components include **Compressor**.
10/29/2009	C	0	(4.21) New components include **Delay**. Updated/fixed components include **Compressor**, **AutoDRED**, **Mixer**, **ACU2**, and **Satellite**.
11/24/2009	C	1	(4.22) Updated/fixed components include **Wave**, **MorseKeyer**, **Delay**, **Compressor**, and **Transceiver**.
1/27/2010	D	0	(4.23) New components include **SimpleMixer**, **CommPanel8Stereo**, **AGC**, **CompressorLimiter**, **Expander**, **Gate**, and **ColocatedBeacon**. Updated components include **Playsound**, **MorseKeyer**, and **MarkerTone**,
3/22/2010	D	1	(4.24) Updated components include **Transceiver**, **VORTAC_Controller**, and **RCUbasic**.

Date	Revision	Version	Comments
5/24/2010	E	0	(4.26) New components include **PulseStream** and **PassThrough**. Updated components include **AmpOut** and **Incrementer**.
9/24/2010	F	0	(4.28) Updated components include **Relay**.
11/22/2010	F	1	(4.29) Updated components include **5BandFilter**.
2/23/2011	G	0	(4.30) New components include **ByteSplitter** and **PulseStep**.
4/29/2011	H	0	(4.31) New components include **ComPanel8HRTF4**, **HRTFOut4**, and **URC-200**.
7/2011	H	1	(4.32) Updated components include **Playsound**.
9/2011	H	2	(4.33) Updated components include **Demux**.
11/29/2011	I	0	(4.34) New components include **PulseSequence**.
12/2011	I	1	Updated components include **Playsound**, **Counter**, **RCUbasic**, and **Transceiver**.
3/19/2012	J	0	N/A
2/28/2013	K	0	N/A
6/14/2013	L	0	N/A
1/29/2014	M	0	N/A
N/A	N	0	N/A
2/9/2016	O	0	Fixed broken link and makes small updates.
9/26/2017	P	0	(6.4.0) Added **SpeakerEQ** component description as well as "Set up and run SpeakerEQ" and "Tune SpeakerEQ."Added "StereoWavRecord".
12/1/2017	P	1	Added caution note to "StereoWavRecord".
1/18/2018	P	2	Removed incorrect hyperlink in "Transceiver" section.
4/30/2018	Q	0	Changed *EngineLit* variable's type to Boolean in "EngineLevelD."
8/24/2018	Q	1	Added caution note to "Set up and run SpeakerEQ" and made minor style changes to "Set up and run SpeakerEQ" and "Tune SpeakerEQ."
2/21/2020	R	0	(7.5.0) Documented *CryptoSys* input control in **RCUbasic**. Added **FilterBank** and **IcomBalancer8** components. Edited content for grammar and accuracy.

Date	Revision	Version	Comments
3/29/2021	S	0	Documented *TxCryptoSoundIdx* and *RxCryptoSoundIdx* in **Transceiver** and *SoundIndex*, *Trigger*, and *LibraryId* in **Playsound**. Documented **RTPStream** and added "Add an RTP Stream Map" and "Assign the RTP Stream Map to a Telestra server." Made other minor grammar and style updates.
9/27/2022	S	1	Updated the *PTTselect* description in "ACUchannel." Removed "license" descriptions in the Red Hat Enterprise Linux export statement in the front matter.
2/13/2022	S	2	Documented the **tsr2wav.py** script in "RecordReplay."
3/8/2023	S	3	Updated the Red Hat Enterprise Linux subscription and export statement to the front matter.
1/19/2024	S	4	Updated deprecated "Target," "ACE," and "Remote Management System" terminology; fixed position behavior error in "StereoWavRecord."

Contents

1.0 Introduction

Studio is a powerful suite of software tools providing a software development toolkit for building sound and communications models. In Studio, models are developed using an array of complex components.

By modifying the components, you can construct anything from a small simulation element to a complete sound and communications audio modeling system for your application. In other words, the components are flexible enough to let you construct basic intercom systems and models that closely match the functionality of a commercial or military platform communication system.

The purpose of this manual is to provide extensive information on the Studio component structure and the operation of each component. The components are organized in the following order.

- **Audio**
- **AudioIO**
- **CommPanel**
- **Control**
- **Dynamics**
- **Environmental Cue**
- **Highway 3D Service**
- **Intercom**
- **IOInterfaces**
- **Platform**
- **Host Control**
- **Radio Components**
- **Speech**

Note: *Not all of the features and menu items that appear on your system are described in this manual.*

While the components allow you to construct much of the audio simulation and infrastructure for a given application, you must still develop a good portion of additional simulation code to drive the constructed model in sufficient fashion, fully achieving the sound and communications operations of your application.

Each component is listed with a general summary and description, and the remainder of the section is divided into table formats for the inputs, outputs, and internal parameters. For the remainder of this document, each component section is organized into the following table sections:

- *Inputs*
  - Audio Inputs
  - Control Inputs
- *Outputs*
  - Audio Outputs
  - Control Outputs
- *Internal parameters*: any values that must be set as part of a component configuration that DO NOT have an external connection port.

***Note**: Not every component has all the tables listed above; tables may vary depending on the complexity of the component.*

Within each table, the parameters are organized alphabetically for search ability.

1.1 Component viewer

The component viewer provides specific component information and the component values. Each component has two views:

- *Filtered:* displays the most important parameters that must be set.
- *Unfiltered:* displays every parameter available for that component.

Tab Name	Description
Data	**Data Viewer** lists the primitives in a tree view. To expand the view, select the arrows to show the variables within a primitive. In some cases, a variable within a primitive contains its own set of variables and can also be expanded. The dotted lines in **From** and **To** represent links to the component.
Links	The link inspector tab displays input and output links and details, including **Out Source Variable**, **Destination**, **Destination Variable**, and **In Source**, **Source Variable**, and **Destination Variable**.
Info	This setting displays information about the component.
View/Edit Description	This setting allows you to add a description of the component.

Table 1: Component viewer tabs

2.0 Audio components

Mix, filter, or add audio components into highway channels through a feeder connection. The following section details audio components and their internal parameters, including the following:

- **AmpMod**
- **AudioFeed**
- **AutoDRED**
- **ComplexPlaysound**
- **Compressor**
- **Delay**
- **Demux**
- **Envelope**
- **Filter**
- **LevelDCapture**
- **Lockout**
- **MessageList**
- **Mixer**
- **NoiseSource**
- **PEnvelope**
- **PFilter**
- **Playsound**
- **Pulse**
- **PulseSequence**
- **PulseStep**
- **PulseStream**
- **RecordReplay**
- **SimpleMixer**
- **Sequencer**
- **VolumeControl**
- **Vox**
- **Wave**

2.1 AmpMod

Summary: **Amplitude Modulator** (i.e., **AmpMod**) generates a carrier signal with an amplitude controlled by a modulating signal. This is useful for general warning tones (e.g., Radar Warning Receivers) that require dynamic control. Complex warning tones can be generated when the amplitude modulator is used with **Pulse**.

Description: **AmpMod** provides a signal multiplication capability between two signals, a carrier waveform, and a modulating envelope.

The carrier waveform is the external signal source connected to the carrier input parameter. *CarrierOffset* is an offset value added to the amplitude of the carrier waveform before modulation. If no signal source is connected to *Carrier*, and *CarrierOffset* is 0, **AmpMod** does not generate an output signal. If no signal source is connected to *Carrier*, and *CarrierOffset* is not 0, **AmpMod** uses the offset value as a DC carrier (i.e., offset).

Gain controls the **AmpMod** output's amplitude. When the gain is less than or equal to 0, **AmpMod** does not output a signal.

The modulating signal is the external signal source connected to *ModulationSignal*. Typically, the modulating signal source is a square wave or pulse; however, you may use any signal type. *ModulationOffset* is the offset value added to the amplitude of the modulating signal before the lag filter is applied. As a result, the modulation signal can be offset from zero to allow for control of the modulation depth. The modulation offset should be 1.0 to provide a full depth of modulation from a square or sinusoidal source. This assumes the gain of the originating signal is set to 1.0, in which case it swings between -1.0 and 1.0, hence the need for a 1.0 offset. The **AmpMod** input variable controls whether the modulating signal is applied to the carrier. If not, the signal source is connected to *ModulationSignal*, and the **AmpMod** does not modulate the carrier signal.

FilterFrequency determines the filter constant for the lag filter, which filters the modulating signal. This lag filter softens the edges, which occur when a square wave modulates a sine wave. The filter constant determines the effective slew rate of the modulating signal.

***Note**: The lag filter is an a-rate function, not a k-rate function.*

Table 2, "AmpMod audio inputs" below displays **AmpMod** audio input variables:

Name	Type	Default Value	Modifier	Modifier_ default	Range
CarrierSignal	**audio**	N/A	N/A	N/A	N/A
CarrierOffset	**float32**	**1.0**	Multiply (*)	**0.0**	**0.0–Inf**
Gain	**float32**	**1.0**	Multiply (*)	**1.0**	**0.0–Inf**
ModulationSignal	**audio**	**N/A**	N/A	N/A	N/A
ModulationOffset	**float32**	**1.0**	Multiply (*)	**0.0**	**0.0–1.0**
Modulate	**Boolean**	**FALSE**	**XOR**	**TRUE**	N/A

Table 2: AmpMod audio inputs

Table 3, "AmpMod audio output" below displays the **AmpMod** audio output variable:

Name	Type	Default Value	Modifier	Modifier_ default	Range
OutSignal	**audio**	N/A	N/A	N/A	N/A

Table 3: AmpMod audio output

Table 4, "AmpMod internal parameter" below displays the **AmpMod** internal parameter variable:

Name	Type	Default Value	Modifier	Modifier_ default	Range
FilterFrequency	**float32**	**25.0**	N/A	N/A	N/A

Table 4: AmpMod internal parameter

2.2 AudioFeed

This component is described in Section 9.0, "Highway 3D Service" on page 159.

2.3 AutoDRED

Summary: **AutoDRED** conducts a series of five tests to verify the system's speaker setup.

Description: **AutoDRED** starts the first test by sending pink noise to all speakers in the system setup. Test 2 sends a narrowband noise centered at 100 Hz out from each individual speaker in the setup. Test 3 sends a narrowband noise centered at 1,000 Hz from each individual speaker in the setup. Test 4 sends a narrowband noise centered at 10,000 Hz from each individual speaker in the setup. Test 5 plays pink noise from each individual speaker.

If a test fails, *ErrorCode* displays the channel number that failed. If all tests pass, it confirms that all speakers are working properly in the system's setup.

For the initial automated setup procedure, set *TestEnable* and *SetupEnable* to **TRUE**. This action runs the setup tests for every channel. Setup takes approximately two minutes per channel. Setting up to 16 channels takes approximately 32 minutes.

***Important**: If possible, clear the room or simulator with the speakers during testing. If you must be in the same room during testing, remain still with no movement.*

Table 5, "AutoDRED audio input" below lists and describes the **AutoDRED** audio input variable:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	Connects to the AutoDRED's microphone's audio input signal.

Table 5: AutoDRED audio input

Table 6, "AutoDRED control inputs" below lists and describes **AutoDRED** control input variables:

Name	Type	Default Value	Description
CaseDuration	**float32**	**0.0**	The amount of time sounds are played for each test case.
NumChannels	**uint8**	**0**	Set this value to the number of speakers in the system setup. Limit is 16 channels.
SetupEnable	**Boolean**	**FALSE**	If **TRUE**, **AutoDRED** initiates the automated setup procedure. This process may take a few minutes, depending on the number of channels (about two minutes per channel).
Test2Freq	**float32**	**100.00**	Sets the center frequency for Test 2.
Test3Freq	**float32**	**1000.00**	Sets the center frequency for Test 3.
Test4Freq	**float32**	**7000.00**	Sets the center frequency for Test 4.
TestCaseSelect	**uint8**	**0**	Use this with *TestEnable* to select the test. • ***1****:* sends pink noise to all speakers. • ***2****:* sends a narrowband noise centered at 100 Hz to each individual speaker. • ***3****:* sends a narrowband noise centered at 1,000 Hz to each individual speaker. • ***4****:* sends a narrowband noise centered at 10,000 Hz to each individual speaker. • ***5****:* sends pink noise to each individual speaker.
TestEnable	**Boolean**	**FALSE**	Set to **TRUE** to begin Tests 1–5.

Table 6: AutoDRED control inputs

Table 7, "AutoDRED control outputs" below lists and describes **AutoDRED** control output variables:

Name	Type	Default Value	Description
Channel	**uint8**	**0**	Displays the channel number that you are testing.
ErrorCode	**uint32**	**0**	Displays the number of the speaker that fails. This number appears until *TestEnable* is reset.
LevelDifference	**float32**	**0**	Decibel difference between the last channel tested and the reference value that was created during the initial setup procedure.
TestCase	**uint8**	**0**	Displays the test case that is currently running.
TestRunning	**Boolean**	**FALSE**	**TRUE** indicates that speaker setup or testing is currently in progress.
Time	**float32**	**0.0**	Amount of time the test has been running.

Table 7: AutoDRED control outputs

Table 8, "AutoDRED internal parameters" on the next page lists and describes **AutoDRED** internal parameter variables:

Name	Type	Default Value	Description
BackgroundLevel	**float32**	**0.0**	The measured Root Mean Squared level for the background noise test.
FilteredNoiseGain	**float32**	**1.0**	Filters the gain for pink noise tests.
GainTable	**function**	N/A	Choose a **TableXY** from the **Math Plan**, which adjusts the audio gain for each individual test. For example, you might boost the low frequency noise volume for a subwoofer test.
LevelTable	**function**	N/A	Choose a **TableXY** from the **Math Plan**. This table should contain the measured signal levels for each test. Copy these values from the **/var/tmp/autodred.dat** file following a setup run.
Mode	**dred_mode**	**STOP**	Reports the current state of the component. Options include: • **Stop** • **Test** • **Manual Test** • **Setup**
NoiseGain	**int32**	**0.10**	Controls the gain for pink noise tests.

Name	Type	Default Value	Description
Results1–Results4	**float32**	**0**	The outputs of *Results1–Results4* is the audio level for all speakers at once.
ThresholdTable	**function**	N/A	Choose a **TableXY** from the **Math Plan**. This table contains the failure thresholds for each individual test. Values are in dB.

Table 8: AutoDRED internal parameters

2.3.1 Set up AutoDRED

This procedure assumes a prebuilt **AutoDRED** mode exists. To set up **AutoDRED**, follow these steps:

1. Install the layout, and open the **AutoDRED** model in the load viewer.
2. Adjust **AutoDRED** gains (i.e., *NoiseGain* and *FilteredNoiseGain*) down to be careful at first. *NoiseGain* controls the level of the flat pink noise test. *FilteredNoiseGain* controls the levels of the three other noise tests, which involve bandpass filtered pink noise.
3. Set *SetupEnable* and *TestEnable* to **TRUE** to do a setup run. You should hear it cycle through four noises on each speaker. Wait for it to finish and set *SetupEnable* and *TestEnable* to **FALSE**.
4. SSH into the Telestra server, and go to **/var/tmp/autodred.dat**, which has a nice summary of setup levels and distances above the noise floor for each test. The goal is for most tests to be >10 dB above the noise floor. This level is not possible on some tests (e.g., high-frequency noise through the subwoofers).
5. Adjust *NoiseGain* and *FilteredNoiseGain*, rerun the setup, and reopen **/var/tmp/autodred.dat**. Wait until most tests reach the 10 dB mark. If one particular test (e.g., Speaker 3, Test 4) should be louder, use *GainTable* to boost or cut the specific volume. Otherwise, do not use *GainTable*.
6. When you are satisfied that most tests are 10 dB above the noise floor, create two *TableXYs* in the **MathPlan**. Give the tests unique names (e.g., **DRED_Ref_Levels** and **DRED_Thresholds**).
7. Enter the levels from **/var/tmp/autodred.dat** into the **DRED_Ref_Levels** table. Go to the example tables for how to structure the table. The test number goes along the top, and the speaker number goes down the side.
8. Fill in the **DRED_Thresholds** table to have the value of 3 dB in all entries that correspond to tests that were 10 dB or more above the noise floor. For test cases that were less than 10 dB above the noise floor, enter a high threshold (e.g., 100) to make the case a “don't care."

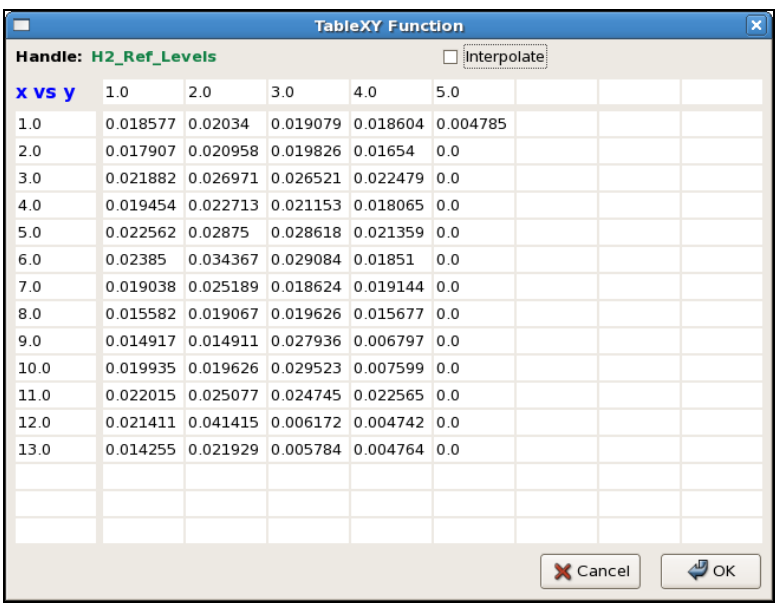
9. Select **Notify Telestra Server** in the **Mathplan** window to push the tables to the Telestra server.

10. Select the two tables in the **AutoDRED** component, filling in *LevelTable* and *ThresholdTable*.

11. Run the test a few times, and observe the level differences on each test. Generally tests should be within 2 dB of the setup run.

Tips

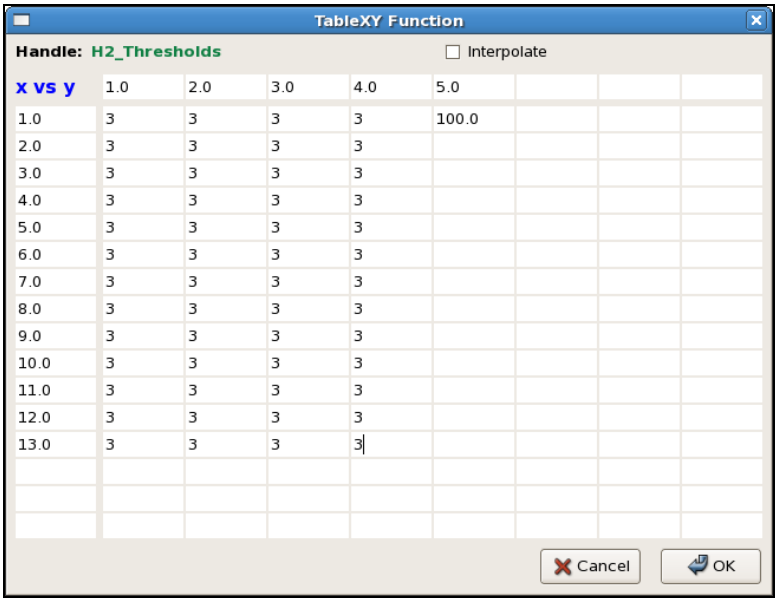
- The 3 dB tolerances are adjustable to be tighter or looser depending on the requirements and what needs to be proved.
- The **/var/tmp/autodred.dat** file provides easier viewing of all the data. The same signal levels are shown in the component data viewer. Distance above noise floor is only shown in the file.
- The background noise test (Test 5) can be useful, but it should only fail if there is a drastic change in background noise level. In the **MathPlan** tables for levels and thresholds, the background noise test is the first entry in the **5.0** column. To start, set a larger threshold (e.g., 6 dB).



x vs y	1.0	2.0	3.0	4.0	5.0
1.0	0.018577	0.02034	0.019079	0.018604	0.004785
2.0	0.017907	0.020958	0.019826	0.01654	0.0
3.0	0.021882	0.026971	0.026521	0.022479	0.0
4.0	0.019454	0.022713	0.021153	0.018065	0.0
5.0	0.022562	0.02875	0.028618	0.021359	0.0
6.0	0.02385	0.034367	0.029084	0.01851	0.0
7.0	0.019038	0.025189	0.018624	0.019144	0.0
8.0	0.015582	0.019067	0.019626	0.015677	0.0
9.0	0.014917	0.014911	0.027936	0.006797	0.0
10.0	0.019935	0.019626	0.029523	0.007599	0.0
11.0	0.022015	0.025077	0.024745	0.022565	0.0
12.0	0.021411	0.041415	0.006172	0.004742	0.0
13.0	0.014255	0.021929	0.005784	0.004764	0.0

Figure 1: AutoDRED RefLevel table

Populate this **TableXY Function** window with the measured Root Mean Squared signal level for each speaker and test. Copy these levels from the **/var/tmp/autodred.dat** file after a setup run.



x vs y	1.0	2.0	3.0	4.0	5.0
1.0	3	3	3	3	100.0
2.0	3	3	3	3	
3.0	3	3	3	3	
4.0	3	3	3	3	
5.0	3	3	3	3	
6.0	3	3	3	3	
7.0	3	3	3	3	
8.0	3	3	3	3	
9.0	3	3	3	3	
10.0	3	3	3	3	
11.0	3	3	3	3	
12.0	3	3	3	3	
13.0	3	3	3	3	

Figure 2: AutoDRED threshold table

Enter failure thresholds in dB into this **TableXY Function** for each speaker and test. The **AutoDRED** test fails if one of the measured signal levels differs from the reference levels by more than the specified failure threshold.

2.4 ComplexPlaysound

Summary: **ComplexPlaysound** plays digitally encoded sound files with dynamically varying elements. The elements occur in three sequences a ramp-up sound, a loop file, and a run-down sound. This component is similar to the MBV sound library complex loop, except a separate component provides this functionality.

***Note**: Set **Playsounds** in **ComplexPlaysound** to buffer for proper operation. In the **Sound Library**, set the sound buffer to **TRUE**.*

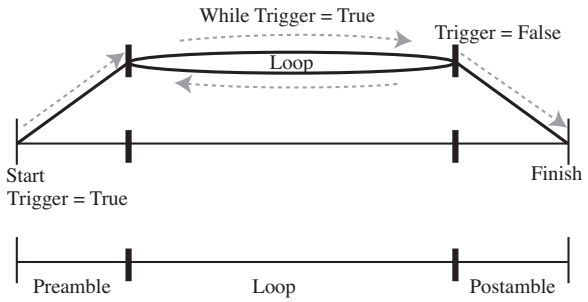
Description: **ComplexPlaysound** starts playing the preamble sound file when the trigger is **TRUE**. After playing the preamble in its entirety, the loop sound file plays. The loop sound file continues to play in a loop until the trigger becomes **FALSE**. At this point, the postamble sound file is played.

When **TRUE**, *ProportionPostamblePlay* allows the component to interrupt the preamble and proportionally play the postamble. The postamble sound's starting point depends on how much of the preamble sound has been played. When **FALSE**, it plays the entire postamble sound regardless of the preamble sound's interruption point.

Trigger and *Pause* control playback. In**ComplexPlaysound**, set *LibraryID*. You can set *GroupID* in **ComplexPlaysound** or modify it using an external control. The indices (e.g., *Preamble*, *Postamble*, *Loopsound*) determine which sound files are used within the specified library and group. A group value of 0 indicates that the sound file is not in a group but directly under the library.

The output signal can connect to any component that accepts audio as an input (e.g., **Mixer)**:

Complex Playsound



For example, when *Trigger* is **TRUE**, *Preamble* plays to the 60 percent position. When *Trigger* is **FALSE**, *Postamble* plays to the 40 percent position.

Proportional Postamble Play

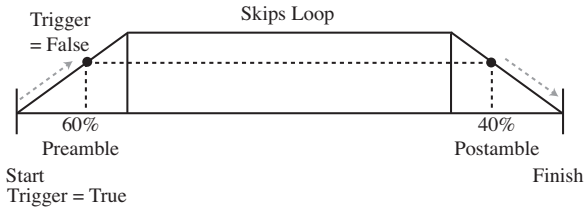


Table 9, "ComplexPlaysound audio inputs" on the next page lists and describes **ComplexPlaysound** audio input variables:

Name	Type	Default Value	Description
LoopSoundIdx	**playsound_sound**	**0**	The value of the loop file index. This index selects a file from within a group. If it doesn't find a match, then it doesn't play a file. If no external variable is connected to the index, the offset is used.
PostambleSoundIdx	**playsound_sound**	**0**	The value of the postamble file index. This index selects a file from within a group. If it doesn't find a match, then it doesn't play a file. If no external variable is connected to Index, the offset is used.

Name	Type	Default Value	Description
PreambleSoundIdx	**playsound_ sound**	**0**	The value of the preamble file index to be played. This index is used to select a file from within a group. If no matches are found then no file is played. If no external variable is connected to the index, the offset is used.
Trigger	**Boolean**	**FALSE**	The trigger state; **TRUE** starts playing the sound file.

Table 9: ComplexPlaysound audio inputs

Table 10, "ComplexPlaysound audio output" below lists and describes the **ComplexPlaysound** audio output variable:

Name	Type	Default Value	Description
Out	**audio**	N/A	The audio output signal from **ComplexPlaysound**. This signal can connect to the audio input of another component.

Table 10: ComplexPlaysound audio output

Table 11, "ComplexPlaysound control inputs" on the facing page lists and describes **ComplexPlaysound** control input variables:

Name	Type	Default Value	Description
GroupID	**playsound_ group**	**0**	The value of *GroupID*. *GroupID* chooses a group from within a sound library. • *Modifier*: add (**+**) • *Modifier_default*: **0** • *Range*: **0**–**255**
OutGain	**float32**	**1.0**	Applies amplitude gain control to the output signal. If no external control is connected to *OutGain*, the scale factor is the *OutGain* value.

Name	Type	Default Value	Description
Pause	**Boolean**	**FALSE**	The pause state; **TRUE** freezes the sound file playing, and **FALSE** allows the sound file to continue from the current file position. If *Pause* isn't connected to an external variable, the modifier is the local value. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
ProportionPostamblePlay	**Boolean**	**FALSE**	Offsets the starting position of the postamble sound file relative to the current position of the preamble sound file. Go to the diagram for an example.

Table 11: ComplexPlaysound control inputs

Table 12, "ComplexPlaysound internal parameter" below lists and describes the **ComplexPlaysound** internal parameter variable:

Name	Type	Default Value	Description
LibraryID	**playsound_library**	**<Select>**	The *LibraryID* selects the library, which includes sound files you may choose from.

Table 12: ComplexPlaysound internal parameter

2.5 Compressor

Summary: **Compressor** modifies the dynamic range of an audio signal to provide a form of automatic volume control. This component includes the **Gate**, **Expander**, **Compressor**, and **Limiter** stages.

Description:

Compressor consists of four stages (i.e., subcomponents):

1. ***Gate**:* ensures that no audio signal below a certain threshold level is passed to the next stage. This stage establishes a minimum level for a signal to be processed.

2. ***Expander**:* reduces or increases the level of an audio signal if it is below a certain threshold. This stage acts as an automatic level control for a microphone.

3. ***Compressor**:* reduces the level of and audio signal if it is above a certain threshold. This stage prevents an input stage from over-driving or softens loud sections of audio.

4. ***Limiter**:* a **Compressor** with an infinite ratio and instant attack so that the output signal is a scaled version of the input signal with the highest peak level reaching the threshold. This stage typically prevents over-driving amplifiers and speakers.

Figure 3, "Compressor stages" below shows **Compressor's** process:

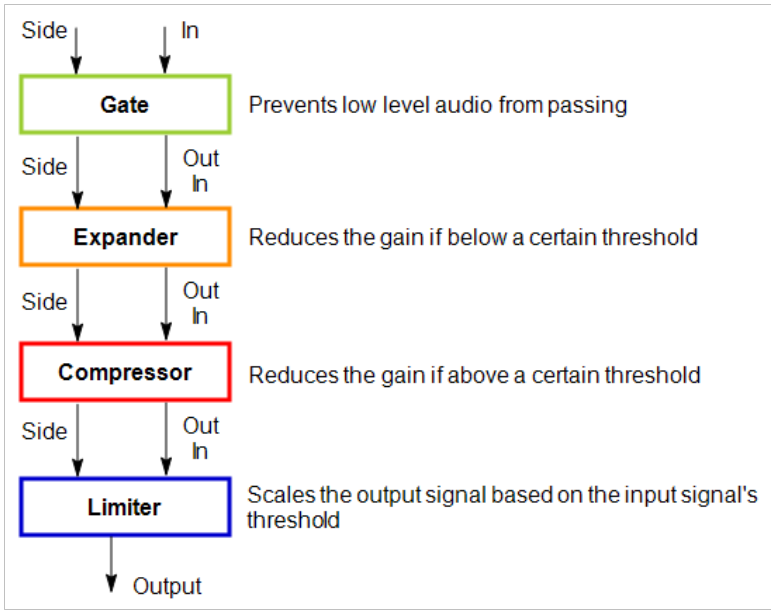


Figure 3: Compressor stages

Figure 4, "Compressor static curve" below shows **Compressor's** static curve:

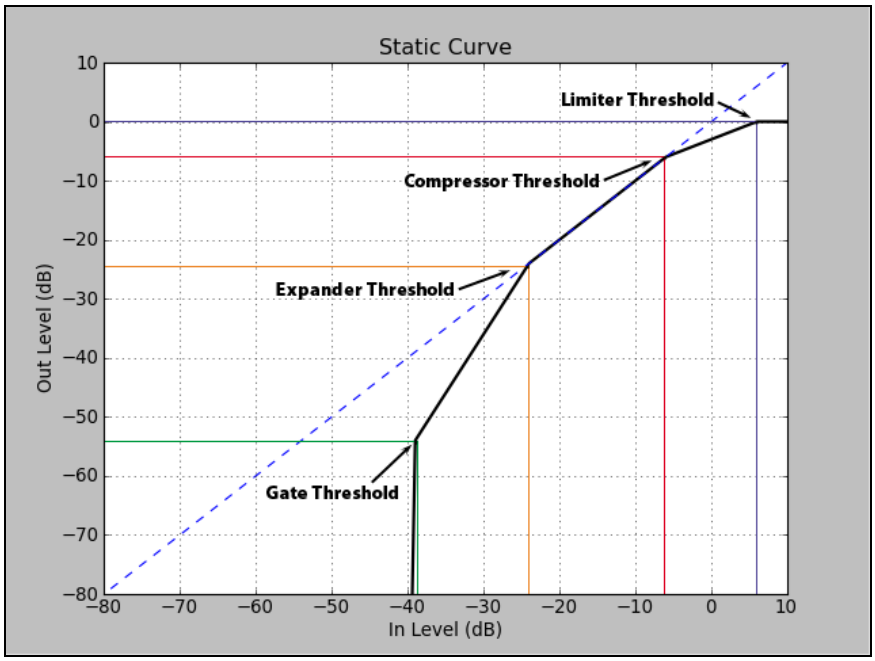


Figure 4: Compressor static curve

Figure 5, "Compressor input" below shows **Compressor's** input threshold:

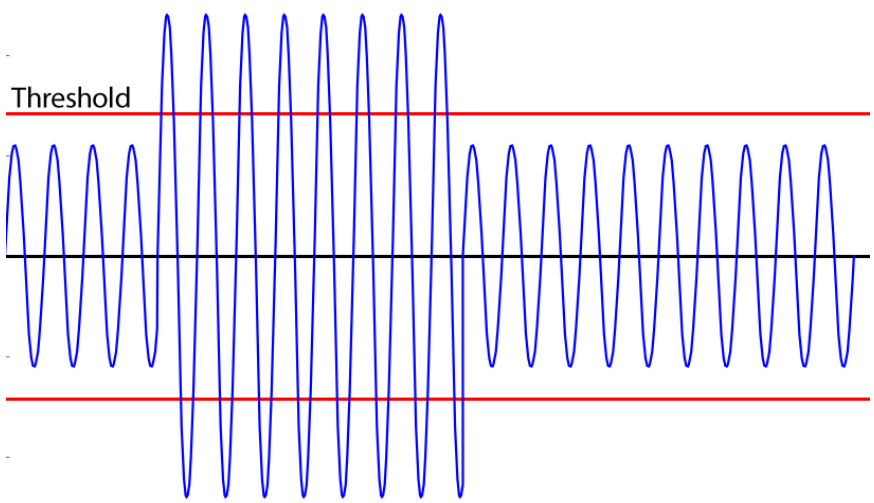


Figure 5: Compressor input

Figure 6, "Compressor output threshold" below shows **Compressor's** output threshold:

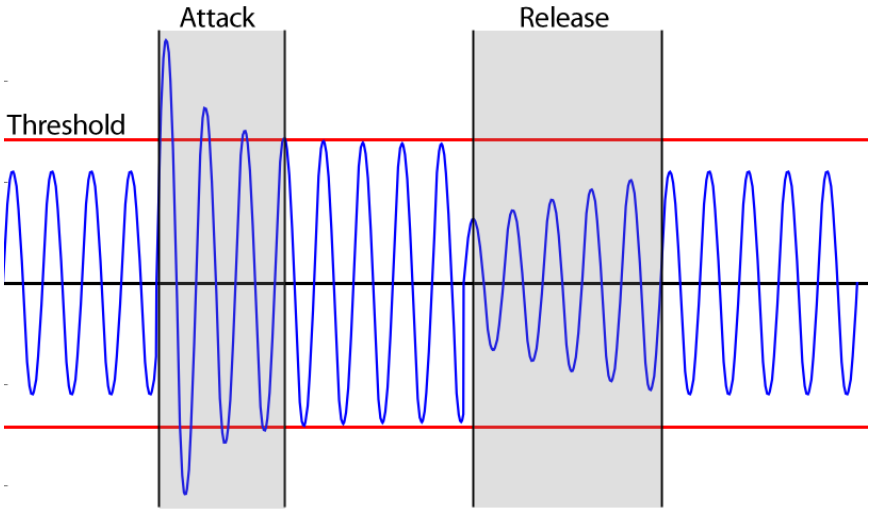


Figure 6: Compressor output threshold

Table 13, "Compressor audio inputs and outputs" below lists and describes **Compressor** audio inputs and outputs:

Name	Type	Default Value	Description
Audio Inputs			
InSignal	**audio**	N/A	Input signal to **Compressor**.
SideIn	**audio**	N/A	Side-chain input to **Compressor**.
Audio Outputs			
OutSignal	**audio**	N/A	The signal after **Compressor** has processed it.

Table 13: Compressor audio inputs and outputs

Table 14, "Gate control input" below lists and describes the **Gate** stage's control input variables:

Name	Type	Default Value	Description
GateEnable	**Boolean**	**FALSE**	If **TRUE**, the **Gate** stage is enabled and processes the audio signal. If **FALSE**, the input audio is passed unaltered to the next stage.
GateSideEnable	**Boolean**	**FALSE**	When **TRUE**, allows the amplitude of an audio signal at *SideIn* to control **Gate**, which acts on the audio signal at *InSignal*.
GateThreshold	**float32**	**1.0**	The threshold level in dB; when the input signal amplitude falls below this threshold, **Gate** prevents the signal from passing. • *Modifier:* **-60.0**
GateRelease	**float32**	**1.0**	Release time in milliseconds of **Gate**. The longer this time is, the longer **Gate** remains open after the input signal falls below the threshold. • *Modifier:* **100.00**
GateHold	**float32**	**1.0**	Once the input signal amplitude falls below the threshold, the input signal amplitude cannot trigger it again until after this time in ms elapses. • *Modifier:* **200.00**
GateOutGain	**float32**	**1.0**	The gain of the output audio from **Gate**.

Table 14: Gate control input

Table 15, "Expander control inputs" below lists and describes the **Expander** stage's control inputs:

Name	Type	Default Value	Description
ExEnable	**Boolean**	**FALSE**	If **TRUE**, the expander stage is enabled and processes the audio signal. If **FALSE**, the input audio passes unaltered to the next stage.
ExSideEnable	**Boolean**	**FALSE**	When **TRUE**, allows the amplitude of an audio signal at *SideIn* to control the **Expander** stage, which acts on the audio signal at *InSignal*.
ExThreshold	**float32**	**1.0**	Expands any signal content below this level. Values are in dB. • *Modifier:* **-50.00**
ExRatio	**float32**	**1.0**	The input-to-output ratio applied once the input signal level falls below the **Expander** threshold. For example, if the input signal is below the threshold by 1 dB, a ratio of 0.5 (1:2) produces an output under the threshold by 2 dB. Setting the ratio above 1 makes the **Expander** an upward **Expander**. Usable ranges for upward **Expanders** are not much greater than 1 since they tend to boost signals dramatically. • *Modifier:* **0.5**
ExRelease	**float32**	**1.0**	**Expander's** release time in milliseconds. Greater values delay **Expander's** response to falling signal levels. • *Modifier:* **100.00**
ExAttack	**float32**	**1.0**	**Expander's** attack time in milliseconds. Greater values delay **Expander's** response to rising signal levels. • *Modifier:* **25.0**
ExOutGain	**float32**	**1.0**	The output audio gain from **Expander**.

Table 15: Expander control inputs

Table 16, "Compressor control inputs" below lists and describes the **Compressor** stage's control inputs:

Name	Type	Default Value	Description
CompEnable	**Boolean**	**TRUE**	If **TRUE**, the **Compressor** stage is enabled and processes the audio signal. If **FALSE**, the input audio passes unaltered to the next stage.
CompAttack	**float32**	**1.0**	The amount of time it takes for gain reduction to take full effect when the input level exceeds the threshold. Values are in ms. • *Modifier:* **50.0**
CompSideEnable	**Boolean**	**FALSE**	When **TRUE**, allows the amplitude of an audio signal at *SideIn* to control the **Compressor** stage, which acts on a signal at *InSignal*.
CompRatio	**float32**	**1.0**	The input-to-output ratio applied once the input signal amplitude rises above the **Compressor** stage's threshold. For example, when the value is 4.0 (i.e., 4:1), an input signal that is 4 dB above the threshold yields an output signal that only 1 dB above the threshold. • *Modifier:* **4.0**
CompRelease	**float32**	**1.0**	The amount of time it takes for gain reduction to dissipate when the input level falls below the threshold. Values are in ms. • *Modifier:* **500.00**
CompThreshold	**float32**	**1.0**	Any signal content above this level is compressed. Values are in dB. • *Modifier:* **-6.0**
CompOutGain	**float32**	**1.0**	Sets the linear gain factor for the output of the **Compressor**. This gain defaults to **1** if the **Compressor** is not enabled. The gain of the output audio from the **Compressor** stage.

Table 16: Compressor control inputs

Table 17, "Limiter control inputs" below lists and describes the **Limiter** stage's control inputs:

Name	Type	Default Value	Description
LimAllowClip	**Boolean**	**FALSE**	Enables a simple **Limiter** algorithm that clips the signal if it breaks the specified threshold.
LimEnable	**Boolean**	**FALSE**	If **TRUE**, the **Limiter** stage is enabled and processes the audio signal. If **FALSE**, the input audio is passed unaltered.
LimThreshold	**float32**	**1.0**	The specified level that the signal is limited to. Values are in dB. • *Modifier:* **0.0**
LimSideEnable	**Boolean**	**FALSE**	When **TRUE**, allows the amplitude of an audio signal at *SideIn* to control the **Limiter**, which acts on the audio signal at *InSignal*.
OutGain	**float32**	**1.0**	Sets the gain factor for the output of the **Limiter** stage. This gain is applied even if *LimEnable* is **FALSE**.
LimRelease	**float32**	**100.0**	The release time (in milliseconds) of the **Limiter** stage. The longer this time, the longer it takes for the effect of **Limiter** to dissipate after the input signal level falls below the threshold.

Table 17: Limiter control inputs

Table 18, "Compressor internal parameters" below lists and describes the **Compressor** stage's internal parameters:

Name	Type	Default Value	Description
CompOutSignal	**audio**	N/A	The signal after the **Compressor** stage processes it.
GateOutSignal	**audio**	N/A	The signal after the **Gate** stage processes it.
ExOutSignal	**audio**	N/A	The signal after the **Expander** stage processes it.

Table 18: Compressor internal parameters

2.6 Delay

Summary: Delays the input audio.

Description: **Delay** postpones audio by the specified number of frames (1.33 ms) up to one second.

Table 19, "Delay audio input and output" below lists and describes **Delay** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	The delayed audio signal's input.
Audio Output			
OutSignal	**audio**	N/A	The delayed audio signal's output.

Table 19: Delay audio input and output

Table 20, "Delay control inputs" below lists and describes **Delay** control input variables:

Name	Type	Default Value	Description
DelayFrames	**int32**	**0.0**	The number of frames to delay the input audio. Each frame equals 1.33 ms. ***Important**: Editing DelayFrames while it is processing audio may cause audible popping sounds.*
Enable	**Boolean**	**FALSE**	When **TRUE**, the audio signal is delayed.
OutGain	**float32**	**1.0**	The gain of the output audio.

Table 20: Delay control inputs

Table 21, "Delay internal parameter" below describes the **Delay** internal parameter variable:

Name	Type	Default Value	Description
DelayMS	**float32**	**0.0**	The frame delay expressed in milliseconds. *DelayFrames* × 1.33 ms

Table 21: Delay internal parameter

2.7 Demux

Summary: **Demultiplexer** (i.e., **Demux**) takes an input signal and divides it among 16 outputs, as specified by the control.

Description: **Demux** is made up of one signal input, one control input, and 16 signal outputs. **Demux** is commonly used for environmental cue applications where it is necessary to test different speakers in the simulator.

Table 22, "Demux variables" below describes **Demux's** variables:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	Provides a connection to an audio signal that is to be distributed to the signal outputs.

Table 22: Demux variables

Table 23, "Demux audio outputs" below describes **Demux** audio outputs:

Name	Type	Default Value	Description
SignalOut00–SignalOut15	**audio**	N/A	*SignalOut00–SignalOut15* are the audio output signals from **Demux**. The audio signal can be routed out to one out signal or up to 16 out signals, as designated by the control input.

Table 23: Demux audio outputs

Table 24, "Demux control input" below describes the **Demux** control input:

Name	Type	Default Value	Description
Control	**int32**	**65535**	Specifies which of the 16 *SignalOut* variables receive the audio signal. The default of 65535 sends the audio signal to all 16 outputs. The input value is binary weighted.

Table 24: Demux control input

2.8 Envelope

Summary: **Envelope** applies a filter to an input signal based on three math functions.

Description: **Envelope** applies a type of filtering to the input signal. The type of filtering may be lowpass, bandpass, or highpass. MathFunction defines the input signal's frequency, gain, and QFactor. Figure 7, "Audio Envelope workflow" below shows **Envelope's** workflow:

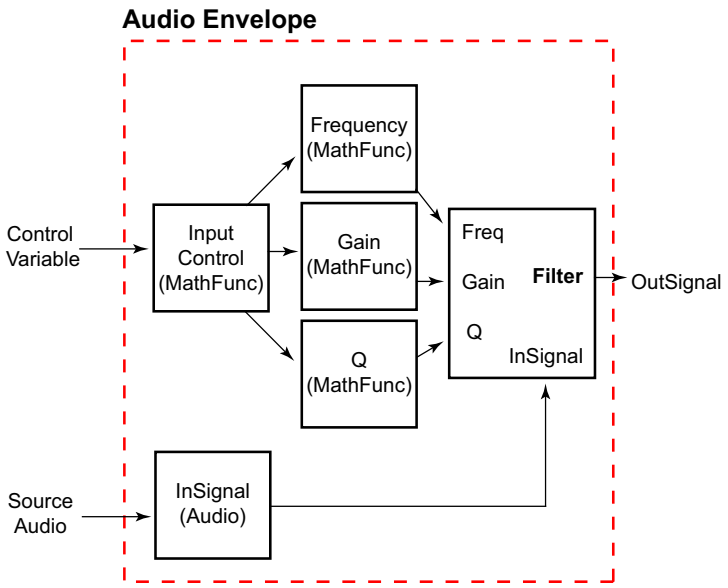


Figure 7: Audio Envelope workflow

Table 25, "Envelope audio input and output" below lists and describes **Envelope** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	Provides a connection to an audio signal for filtering.
Audio Output			
OutSignal	**audio**	N/A	The output audio signal after it is filtered.

Table 25: Envelope audio input and output

Table 26, "Envelope control inputs" below lists and describes **Envelope** control input variables:

Name	Type	Default Value	Description
Control	**float32**	1.0	Determines what the input is going to be based on the **MathFunctions**.
Enable	**Boolean**	**FALSE**	When **TRUE**, the filter is turned on.

Table 26: Envelope control inputs

Table 27, "Envelope internal parameters" below lists and describes **Envelope** internal parameter variables:

Name	Type	Default Value	Description
FilterType	**filter_type2**	**LowPass**	Value used to select the filter type.
FreqFunction	**function**	**<Select>**	The name of the **MathFunction** that sets the filter's center frequency.
GainFunction	**function**	**<Select>**	The name of the **MathFunction** that sets the filter's gain.
QFunction	**function**	**<Select>**	The name of the **MathFunction** that sets the filter's Q.

Table 27: Envelope internal parameters

2.9 Filter

Summary: **Filter** applies a filter to an input signal.

Description: **Filter** applies lowpass, bandpass, or highpass filters to an input signal. Use input variables elsewhere in the model or from the host interface to control the **Filter** quality factor, roll-off frequency, and gain. Pass the input signal audio to the filter only if the derived output gain is greater than 0.0.

The following parameters control **Filter**:

- *Enable*
- *FilterType*
- *Frequency*
- *QFactor*

Enable is specified by an externally controlled variable and an XOR logic gate control. XOR logic is applied to the external variable and the gate control to derive a final value for *Enable*. If no external variable is connected, the value of the logic gate control becomes *Enable* . If *Enable* is **FALSE**, no filtering is applied to the input signal.

FilterType determines the type of **Filter** applied to the input signal (e.g., **Off**, **Lowpass**, **High-pass**, or **Bandpass**). If *FilterType* is turned off, no filtering is applied to the input signal. *Frequency* provides the roll-off frequency of the **Filter** in Hertz. The meaning of the frequency depends on the **Filter** type:

- Filter type frequency meaning
- Off ignored
- Lowpass upper bound frequency
- Highpass lower bound frequency
- Bandpass center frequency with bandwidth controlled by *QFactor*

QFactor is the **Filter** quality factor, which effectively determines the **Filter** roll-off for low-pass and high pass **Filters** and the **Filter** bandwidth for bandpass **Filters**. *QFactor* is inversely proportional to the **Filter** roll-off or bandwidth.

The **Filter** parameters derive the filter coefficients for a two-pole infinite input response (IIR) filter. If *QFactor* is 0 or less, use a low value above 0 for the k calculation. If the **Filter** type is off, the filter output signal is the input signal.

An amplitude gain control is applied to the **Filter** output. This output gain control consists of a connection to an external control variable and a scale factor. The value of the external variable is multiplied by the scale factor to drive the output gain. If no external variable is connected, the value for the scale factor becomes the output gain.

Filter output audio is only active if the following conditions are **TRUE**:

1. The **Filter** output is active.
2. The derived output gain (i.e., *External Variable* × *Scale Factor*) is greater than 0.0.

Table 28, "Filter audio input and output" below lists and describes **Filter** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	The signal connection used as an input to the filter.
Audio Output			
OutSignal	**audio**	N/A	The output signal from **Filter**.

Table 28: Filter audio input and output

Table 29, "Filter control inputs" below lists and describes **Filter** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**FALSE**	Determines if the input signal is filtered. If no external control is connected to *Enable*, the x/modifier is the control value. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**
Frequency	**float32**	**1.0**	Provides the roll-off frequency of the **Filter**. For low-pass filters, *Frequency* acts as an upper bound frequency. For highpass filters, *Frequency* acts as a lower bound frequency. For bandpass, *Frequency* acts as a center frequency, and the filter bandwidth is determined by *QFactor*. If no external control is connected to *Frequency*, the scale factor is the *Frequency* value. The acceptable range is 0 to half of the model sample rate. • *Modifier*: multiply (*) • *Modifier_default*: **2000.0**
OutGain	**float32**	**1.0**	Applies amplitude gain control to the output signal. If no external control is connected to *OutGain*, the scale factor is the *OutGain* value. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
QFactor	**float32**	**1.0**	The **Filter** quality factor, which effectively determines the **Filter** roll-off for lowpass and highpass **Filter**s and the **Filter** bandwidth for bandpass **Filters**. *QFactor* is inversely proportional to the **Filter** roll-off or bandwidth. If the input is not connected, the scale factor is used as the *QFactor* value. • *Modifier*: multiply (*) • *Modifier_default*: **0.707100**

Table 29: Filter control inputs

Table 30, "Filter internal parameter" below shows the **Filter** internal parameter variable:

Name	Type	Default Value	Description
FilterType	**filter_type2**	**LowPass**	Determines the type of filter applied to the input signal: • **LowPass** • **HighPass** • **BandPass** • **LowPassQ** • **BandPassQ** • **HighPassQ** • **Notch** If no filtering is desired, set *FilterType* to **OFF**.

Table 30: Filter internal parameter

2.10 Lockout

Summary: **Lockout** allows sharing of a limited number of assets on a "first-come, first-served" basis.

Description: This component routes input audio streams (i.e., operators) to output audio streams (i.e., assets) based on when each operator triggers his or her press-to-talk (PTT) device. If the number of input streams is greater than the available outputs, operators who press the PTT button last are unable to access an output and are locked out. **Lockout** accepts up to eight separate operator inputs. Operators are assigned to the output assets based on when they PTT. The first operator to PTT gains access to Asset 1, the second to Asset 2, etc. If the number of available assets is exceeded, the next operator is locked out and unable to gain access to any of the assets.

Once an operator releases his or her PTT, the asset allocated to him or her is available to all operators once the release time elapses. The component is similar to an office phone system, whereby multiple outside lines are accessible by many people, but the number of simultaneous calls is limited.

Table 31, "Lockout audio input" below describes the **Lockout** audio input variable:

Name	Type	Default Value	Description
Operator1_Audio–Operator8_Audio	**audio**	N/A	Input audio stream that corresponds to the PTT of the same name. Operator audio is routed to asset audio based on when the PTT is pressed. The audio stream is always routed to the lowest numbered asset that is available.

Table 31: Lockout audio input

Table 32, "Lockout audio output" below describes the **Lockout** audio output variable:

Name	Type	Default Value	Description
Asset1_Audio–Asset8_Audio	**audio**	N/A	Operator audio is routed out of the component once a PTT is pressed. When a PTT is **TRUE**, the audio is always routed to the lowest available asset.

Table 32: Lockout audio output

Table 33, "Lockout control inputs" below lists and describes **Lockout** control input variables:

Name	Type	Default Value	Description
AssetMask	**byte**	**255**	Controls which assets the operator can use; the least significant bit corresponds to *Asset1*, and the most significant corresponds to *Asset8*. For example, an *AssetMask* of 15 would make *Asset1*, *Asset2*, *Asset3*, and *Asset4* available for the operators to use.
Operator1_PTT–Operator8_PTT	**Boolean**	**FALSE**	The operator PTT acts as a trigger for each of the eight input streams. A value of **TRUE** is used when an operator is trying to gain access to an asset. The operator PTT can also be used without the operator audio.
OperatorMask	**byte**	**255**	Controls which operators can be routed to the asset streams with the least significant bit corresponding to Operator1, and the most significant corresponding to Operator8. For example, an *OperatorMask* of 31 makes Operators 1, 2, 3, 4, and 5 available for use within **Lockout**.
ReleaseDelay	**float32**	**1.0**	The time, in seconds, that **Lockout** waits after a PTT is **FALSE** before allowing another operator to access an asset.

Table 33: Lockout control inputs

Table 34, "Lockout control outputs" below lists and describes **Lockout** control output variables:

Name	Type	Default Value	Description
Asset1_Operator–Asset8_Operator	**uint8**	**0**	Reports which input audio stream (operator) is assigned to the corresponding output (asset). For example, if *Asset1_Operator* is equal to 4, that indicates that the audio from *Operator4_Audio* is being routed to *Asset1_Audio*.
OperatorAssignedMask	**byte**	**0**	Reports if an operator is currently assigned to an asset. For example, a value of 3 indicates that *Operator1* and *Operator2* are both routed to one of the asset outputs.

Table 34: Lockout control outputs

2.11 LevelDCapture

Summary: **LevelDCapture** records audio for Level D compliance testing. The audio recorded by this component can be compared to the reference audio, in real time with the scope or offline with the spectral analysis capability accessible through the Telestra web interface.

Description: To record audio using **LevelDCapture**, link the record audio coming in through the ACU2 to **LevelDCapture**. In the sound library, add the aircraft reference recordings and set the path value and index number. In **LevelDCapture**, set the test number to match the file's index number, this selects the reference file to run with the test recordings. Record the audio. The recording then goes to a file on the Telestra server. The file number resembles **library000_group***yyyyy***_index00***X*, where *yyyyy* represents the Level D recording file number, and *X* represents the test number. Host control can direct the test number and the start variable.

In **LevelDCapture**, do the following:

1. Choose a library and group object (these are set in the Sound Library)
2. Set the ACU2 gain level to match the level of the reference recording.
3. Add the table function to calculate the gain level with the calibration number. This is the number calculated during calibration of the ACU2 with the microphone.

4. Set **X** to the test case number and **f(x)** to the calibration number. This value is sent to the ACU2.
5. Set the *RecordLength* function for each test case to match the amount of time the file was recorded.
6. Set the maximum record time.

After recording the audio, open the Telestra web interface and go to **Spectral Analysis**. Choose a wave set that contains the reference files, and the Telestra web interface automatically links the reference file to the recorded file. Select the box to compare the files and/or create a difference plot to show the margin of the difference. Select **Start Spectral Analysis**, and the Telestra web interface generates a PDF file of the results.

Table 35, "LevelDCapture audio inputs and outputs" below lists and describes **LevelDCapture** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
RecAudio	**audio**	N/A	The audio that typically comes in from a microphone linked to an ACU2. The audio is recorded; similar to **RecordReplay**.
Audio Output			
PSAudioOut	**audio**	N/A	The reference file audio for the specified test number.

Table 35: LevelDCapture audio inputs and outputs

Table 36, "LevelDCapture control inputs" on the facing page lists and describes **LevelDCapture** control input variables:

Name	Type	Default Value	Description
ACUGain	**function**	**<Select>**	Insert a table function containing the ACU2 gain corresponding to each test number.

Name	Type	Default Value	Description
RefAudio	**<Select>**	**0**	The reference audio for comparison. *Playsound_LibraryID* selects the library from the sound library containing the reference audio file to be played.
	playsound_ group	**0**	*Playsound_GroupID* selects a group from the sound library containing the reference audio file to be played.
	playsound_ sound	**0**	*Playsound_SoundID* selects a sound file from the sound library. ***Note:*** *Similar to* ***Playsound****.*
Start	**Boolean**	**FALSE**	Triggers playback of the reference file and recording of *RecAudio*.
TestNumber	**uint16**	**0**	Drives the index number of the reference file and the record file. Use one test number per test case.

Table 36: LevelDCapture control inputs

Table 37, "LevelDCapture control outputs" below lists and describes **LevelDCapture** control output variables:

Name	Type	Default Value	Description
ACUGainResult	**float32**	**1.0**	The gain applied to the ACU2 input audio. Links to **ACU2channel**.
RecState	**player_state**	**STOPPED**	Displays the **RECORD** state of **LevelDCapture**. When **START** becomes **TRUE**, it changes from **STOPPED** to **RECORDING**.

Table 37: LevelDCapture control outputs

Table 38, "LevelDCapture internal parameters" below lists and describes the **LevelDCapture** internal parameter variable:

Name	Type	Default Value	Description
RecordLength			
Function	**function**	**<Select>**	A table function that contains the record time corresponding to the current test number.
Result	**float32**	**0.0**	The length of the recording.
Max	**uint32**	**120**	Set the maximum record length in seconds.

Table 38: LevelDCapture internal parameters

2.12 MessageList

Summary: Plays through a list of sound files and sequences ATIS or warning messages.

Description: **MessageList** is similar to **Playsound** but accepts an array of sound indices to play. You can set the sound indices locally or drive them with a host computer. Sounds play sequentially. The next sound starts immediately after the current sound stops.

Table 39, "MessageList audio outputs" below lists and describes **MessageList** audio output variables:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	The output signal from the component.
RepeatCount	**uint16**	**0**	This number increments each time the same index plays in a row.

Table 39: MessageList audio outputs

Table 40, "MessageList control inputs" below lists and describes **MessageList** control input variables:

Name	Type	Default Value	Description
LibraryID	**playsound_library**	**<Select>**	Sound library selection from layout's repository.
GroupIndex	**playsound_group**	**0**	Selects a group from within the sound library.
SoundIndices	**message**	**[0,0,0...0]**	Array of sound indices to be played (up to 256).
Trigger	**Boolean**	**FALSE**	Trigger state; **TRUE** plays **MessageList**.
Reset	**Boolean**	**FALSE**	When **TRUE**, **MessageList** stops playing. When **FALSE**, **MessageList** resumes playing at the first sound index.
Pause	**Boolean**	**FALSE**	**TRUE** pauses playback of the current sound. When **FALSE**, play continues from the current file location.
PlayAll	**Boolean**	**FALSE**	Determines **MessageList** behavior when not triggered. **TRUE** forces the entire list to play; **FALSE** stops the **MessageList**.
Continuous	**Boolean**	**FALSE**	Determines **MessageList** behavior when *Trigger* is **TRUE** and it reaches the end of the list. If **TRUE**, **MessageList** loops to the beginning and continues playing. If **FALSE**, **MessageList** stops.
OutGain	**float32**	**1.0**	Applies the amplitude gain control to the output signal. If an external control is not connected, the scale factor equals the gain value.

Table 40: MessageList control inputs

Table 41, "MessageList control output" below lists and describes the **MessageList** control output variable:

Name	Type	Default Value	Description
CurrentSound	**playsound_sound**	**0**	Outputs the currently playing sound index.

Table 41: MessageList control output

Table 42, "MessageList internal parameter" below lists and describes the **MessageList** internal parameter variable:

Name	Type	Default Value	Description
Delay	**float32**	**0**	The length of time (in seconds) to wait before looping back to the beginning of the list when in *Continuous* mode.

Table 42: MessageList internal parameter

2.13 Mixer

Summary: **Mixer** provides controlled mixing of up to eight signals into a single, composite signal. **Mixer** determines which of the eight signals to mix with both individual and overall gain control. A ninth signal is always mixed into the output signal and can cascade multiple **Mixers** together.

Description: **Mixer** mixes up to nine signal sources into a single, composite output stream. *Control* mixes the first eight signals, determining which signal(s) to include in the output. *Control* is a single byte in which each bit controls the switch state of a signal. If a given bit is set, the signal corresponding to that bit is added to the **Mixer** output. If present, the ninth signal is always added to the **Mixer** output, regardless of *Control*'s value.

Each signal source has its own amplitude gain control. Each signal gain control connects to an external, controlled variable and a scale factor. The value of the external variable is multiplied by the scale factor to derive the signal gain. If no external variable is connected, the value of the scale factor becomes the signal gain.

The amplitude of the final **Mixer** output is controlled by the output gain control. Like the signal gains, the output gain is an external variable multiplied by a scale factor.

The **Mixer** output audio is only active if all of the following conditions are **TRUE**:

1. At least one if the signal sources (*Signal1–Signal8* or *InSignal*) is active.
2. The derived amplitude gain(s) (*External Variable* × *Scale Factor*) for the active signal source(s) is greater than 0.0.
3. The bit in *Control* corresponding to the active signal source must be set. This condition does not apply to *InSignal* because it is not affected by *Control*.
4. The final **Mixer** output gain control is greater than 0.0.

Table 43, "Mixer audio inputs and outputs" below lists and describes **Mixer** audio input and output variables:

Name	Type	Default Value	Description
Audio Inputs			
InSignal	**audio**	N/A	Additional signal added to output of **Mixer**; independent of the control variable. Allows multiple **Mixers** to cascade together, mixing the audio of more than eight signals.
Signal1–Signal8	**audio**	N/A	Eight signals mixed into composite output according to control selectors and gain values.
Audio Outputs			
OutSignal	**audio**	N/A	Composite output from **Mixer**.

Table 43: Mixer audio inputs and outputs

Table 44, "Mixer control inputs" on the next page lists and describes **Mixer** control input variables:

Name	Type	Default Value	Description
Control	**byte**	**255**	Switches eight signals on or off. Each bit in the control byte enables one of eight signals. When the least significant bit (i.e., Bit 0) is 1, *Signal1* is added to the **Mixer** output. Each bit in the sequence controls remaining signals. *Signal8's* control is the most significant bit (i.e., Bit 7). If no external variable is connected, *Control* uses the modifier's value. • *Modifier*: and (**&**) • *Modifier_default*: **255** • *Range*: **0–255**
InSig_Gain	**float32**	**1.0**	The amplitude gain control for *InSignal*. If no external variable is connected, *InSig_Gain* uses the scalar's value. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**

Name	Type	Default Value	Description
Sig1_Gain–Sig8_Gain	**float32**	**1.0**	Amplitude gain control for signals 1–8. If no external variable is connected, signal gain uses scalar's value. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
Out_Gain	**float32**	**1.0**	Amplitude gain control for the final **Mixer** output. If no external variable is connected, *Out_Gain* uses the scalar's value. • *Modifier*: multiply (*) • *Modifier_Default*: **1.0**

Table 44: Mixer control inputs

2.14 NoiseSource

Summary: **NoiseSource** generates a filtered white, pink, or brown noise signal. The type of filtering applied may be lowpass, bandpass, or highpass. The filter quality factor, roll-off frequency, and gain can be controlled by input variables from elsewhere in the model or from the host interface. The noise signal is an internal pseudo-random noise source, providing an improved noise source that are easier to tune.

Description: This component consists of a signal source and a filter. The source signal on which the filter acts is white Gaussian, pink, or brown noise. The noise signal is only active if the derived *OutGain* is greater than 0.0.

The following parameters control the filter:

- *FilterEnable*: specified by an externally controlled variable and an exclusive-OR (XOR) logic gate control. XOR logic is applied to the external variable value and the gate control to derive a final value for *FilterEnable*. If no external variable is connected, the value of the logic gate control equals *FilterEnable*. If *FilterEnable* is **FALSE**, no filtering is applied to the noise signal.
- *FilterType*: determines the type of filter applied to the input signal (e.g., **Off**, **Lowpass**, **Highpass**, or **Bandpass**). If **Type** is **Off**, no filtering is applied to the noise. *FilterFrequency* (in Hertz) provides the roll-off frequency or the center frequency of the filter, depending the selected filter's type.
- *FilterFrequency*: sets either the roll-off frequency or center frequency of the filter depending on the type of the filter selected:
  - **Off:** Ignored
  - **Lowpass:** upper bound frequency

  - **Highpass:** lower bound frequency
  - **Bandpass:** center frequency with bandwidth controlled by *FilterQFactor*

- *FilterQFactor*: the filter quality factor, which determines the filter roll-off for lowpass and highpass filters and the filter bandwidth for bandpass filters. *FilterQFactor* is inversely proportional to the filter roll-off or bandwidth. These parameters drive the filter coefficients for a two-pole infinite impulse response (IIR) filter.

  If *FilterQFactor* is 0 or less, use a very low, nonzero value for the k calculation. The output from the filter is only active if the noise source is active. If *FilterType* is **Off**, the filter output signal is the raw noise signal.

- *OutGain*: an amplitude gain control is applied to the filter output. This output gain control consists of a connection to an external controlled variable and a scale factor. The value of the external variable is multiplied by the scale factor to derive the output gain. If no external variable is connected, the value for the scale factor becomes the output gain. **NoiseSource** output audio is only active if the output from the filter is active, and the derived output gain (*External Variable* × *Scale Factor*) is greater than 0.0.

Table 45, "NoiseSource audio output" below lists and describes the **NoiseSource** audio output variable:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	Filtered audio signal. If the filter type is **OFF**, it does not produce audio.

Table 45: NoiseSource audio output

Table 46, "NoiseSource control inputs" on the next page lists and describes **NoiseSource** control input variables:

Name	Type	Default Value	Description
FilterEnable	**Boolean**	**FALSE**	Determines if the noise signal is filtered. If no external control is connected to *FilterEnable*, the **XOR** modifier is the *FilterEnable* value. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**

Name	Type	Default Value	Description
FilterFrequency	**float32**	**1.0**	*FilterFrequency* in Hertz provides roll-off frequency of the filter. For lowpass filters, *FilterFrequency* acts as an upper bound frequency. For highpass filter, *FilterFrequency* acts as a lower bound frequency. For bandpass, *FilterFrequency* acts as a center frequency. *FilterQFactor* determines filter bandwidth. If no external control is connected to *FilterFrequency*, scale factor the *FilterFrequency* value. An acceptable range is 0 to half of the model sample rate. • *Modifier*: multiply (*) • *Modifier_default*: **2000.0**
FilterQFactor	**float32**	**1.0**	*FilterQFactor*is the filter quality factor, which determines filter roll-off for lowpass and highpass filters and filter bandwidth for bandpass filters. *FilterQFactor* is inversely proportional to filter roll-off or bandwidth. This field cannot be less than or equal to 0. If no external control is connected to *FilterQFactor*, scale factor is the *FilterQFactor* value. • *Modifier*: multiply (*) • *Modifier_default*: **0.7071**

Table 46: NoiseSource control inputs

Table 47, "NoiseSource control output" below lists and describes the **NoiseSource** control output variable:

Name	Type	Default Value	Description
OutGain	**float32**	**1.0**	*OutGain* applies the amplitude gain control to the output signal. If no external control is connected to *OutGain*, the scale factor equals the *OutGain* value. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**

Table 47: NoiseSource control output

Table 48, "NoiseSource internal parameters" below lists and describes **NoiseSource** internal parameter variables:

Name	Type	Default Value	Description
FilterType	**filter_type2**	**LowPass**	Determines the two-pole filter type applied to the input signal: • *Lowpass* • *Highpass* • *Bandpass* • *LowPassQ* • *BandPassQ* • *HighPassQ* • *Notch* If no filtering is desired, *FilterType* should be **OFF**. Three *Q* filters are amplitude-adjusted so the filter has a unity gain at a roll-off frequency and maintains this gain as the quality factor increases. *Bandpass* filters have lowpass and highpass poles at the same roll-off frequency.
NoiseColor	**noise_color**	**White**	Determines type of noise applied to input signal: • *White Noise*: noise with a flat power spectral density; the intensity is the same at all frequencies within a given band. • *Pink Noise*: a random signal whose amplitude decreases as the frequency increases, preserving constant audio strength per frequency increment. • *Brown Noise*: if the spectral density is proportional to $1/f^2$, it has more energy at lower frequencies, even more than pink noise.

Table 48: NoiseSource internal parameters

2.15 PEnvelope

Summary: **Parametric Envelope** (**PEnvelope**) applies a parametric filter (**PFilter**) to an input signal based on three **MathFunctions**.

Description: **PEnvelope** applies the internal parametric filter to a given band of the input signal, which is determined by three parameters:

- Center frequency
- Bandwidth
- Desired gain

Table 49, "PEnvelope audio input and output" below lists and describes the **PEnvelope** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	Provides a connection to an audio signal for filtering.
Audio Output			
OutSignal	**audio**	N/A	The output audio signal after it is filtered.

Table 49: PEnvelope audio input and output

Table 50, "PEnvelope control inputs" below lists and describes **PEnvelope** component's control input variables:

Name	Type	Default Value	Description
Control	**float32**	**1.0**	Determines what the input is going to be to the **MathFunctions**.
Enable	**Boolean**	**FALSE**	When **TRUE**, the filter is turned on.
ControlFunction	N/A	**<Select>**	Defined in **MathFunction**.

Table 50: PEnvelope control inputs

Table 51, "PEnvelope internal parameters" below lists and describes **PEnvelope** internal parameter variables:

Name	Type	Default Value	Description
BandwidthFunction	**function**	**<Select>**	Defined in **MathFunction** to set the filter bandwidth in Hertz.
FreqFunction	**function**	**<Select>**	Defined in **MathFunction** to set the filter frequency.
GainFunction	**function**	**<Select>**	Defined in **MathFunction** to set the filter gain.

Table 51: PEnvelope internal parameters

2.16 PFilter

Summary: **Parametric Filter** (**PFilter**) is a single-band parametric equalizer that allows you to control the amplitude of a given band of the input signal.

Description: **PFilter** applies the internal parametric filter to a given band of the input signal which is determined by three parameters:

- Center frequency
- Bandwidth
- Gain

Table 52, "PFilter audio input and output" below lists and describes **PFilter** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	*InSignal* is the connection to the signal to be used as an input to the filter. The input signal comes from another component in the model.
Audio Output			
OutSignal	**audio**	N/A	The filtered audio signal. If *Enable* is **FALSE**, it passes original audio without filtering.

Table 52: PFilter audio input and output

Table 53, "PFilter control inputs" below lists and describes **PFilter** control input variables:

Name	Type	Default Value	Description
Bandwidth	**float32**	**1.0**	Sets the bandwidth around its center frequency. Units are in Hertz.
Enable	**Boolean**	**FALSE**	When **TRUE**, turns the filter on.
Frequency	**float32**	**1.0**	The center frequency. Units are in Hertz.
Gain_dB	**float32**	**1.0**	Sets the gain value for the filter. Units are in decibels (dB).

Table 53: PFilter control inputs

2.17 Playsound

Summary: **Playsound** plays digitally encoded sound files. Sounds that have no dynamically varying elements except for overall volume level should be as fixed, offline recorded sound files (e.g., missile launch). Host inputs do the following:

- Trigger playback
- Pause playback
- Set the start of file offset
- Set the end of file offset
- Adjust the overall output gain
- Set the file index number for grouped sound files

Description: *Trigger* and *Pause* control playback. The playback behavior is determined by *Loop* and *Playing* that are set as part of the sound file definition outside of **Playsound**. If the sound file is set for *Loop*, **Playsound** plays the file to its end, then starts at the beginning again until *Trigger* is removed. If the sound file is set for *PlayAll*, **Playsound** plays the file to its end when *Trigger* is removed.

If **Playsound** points to a sound group or library, *SoundIndex* determines which sound file within the group or library is played. *SoundIndex* must be greater than 0 for a sound file within a sound group to be played. **Playsound** does not play if *SoundIndex* is 0. If *Trigger* value is **TRUE**, and *SoundIndex* increments, the sound file plays as if *Trigger* were **TRUE** with the new *SoundIndex* value. If *Trigger* is **TRUE** and the *SoundIndex* increments before the current file is done playing, *PlayAll* and *Loop* determine the behavior.

Several levels of organization contain sound files. A library is a collection of groups. A group is a collection of sounds or play files. Within **Playsound**, set *LibraryId* must be set and is not modified by an external control (e.g., a host control). You can set the group ID locally within **Playsound** or set it with an external control. *SoundIndex* determines which sound file within the specified library and group is selected. A group value of 0 indicates that the sound file is not in a group, but directly under the library equivalent to **Model Builder** organization. *OutGain* controls the amplitude of the **Playsound** output. The output signal from a prerecorded sound file can connect to any component that accepts audio as an input, such as a **Mixer**.

Table 54, "Playsound audio output" below lists and describes the **Playsound** audio output variable:

Name	Type	Default Value	Description
Out	**audio**	N/A	The output signal from **Playsound**, which may connect to another component.

Table 54: Playsound audio output

Table 55, "Playsound control inputs" on the next page lists and describes **Playsound** control input variables:

Name	Type	Default Value	Description
BeginOffset	**float32**	**1.0**	A value between 0.0 and 1.0 adjusts the start position of the sound file. The sound must fully buffer for this variable to function; ensure the sound buffer setting is **TRUE** for the **Sound Repository**.
EndOffset	**float32**	**1.0**	A value between 0.0 and 1.0 adjusts the end position of the sound file. The position is calculated relative to the end of the file (i.e., a value of 0.3 results in an end position of 70 percent of the entire length of the file). If *EndOffset* is less than the start offset after the *BeginOffset* calculation is made, *EndOffset* results in a value of 0.0. The sound must fully buffer for this variable to function; ensure the sound buffer setting is **TRUE** in the **Sound Repository**.
GroupID	**playsound_group**	**0**	Selects the group containing the sound file.
LibraryId	**playsound_library**	**<Select>**	Selects the library containing the group and sound file.

Name	Type	Default Value	Description
OutGain	**float32**	**1.0**	Amplitude gain of the file replay source. If the gain connection is blank, then the gain scale factor is the gain value; otherwise, the gain is the scale factor multiplied by the output result of the control object. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: **0.0**–**Inf**
Pause	**Boolean**	**FALSE**	The PAUSE state. **TRUE** freezes the sound file playing; **FALSE** allows play to continue from the current file position. If no external variable is connected to *Pause*, the modifier equals the local value. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
Randomize	**Boolean**	**FALSE**	When **TRUE**, the **Playsound** signal has a counter running and starts the sound at the counter position when the sound triggers, resulting in a random start position. In **Sound Library**, the sound file buffer must be **TRUE** for *Randomize* to take effect.
SoundIndex	**playsound_sound**	**0**	Selects the sound file you want to play.
Trigger	**Boolean**	**FALSE**	Triggers the sound file to play.

Table 55: Playsound control inputs

Table 56, "Playsound control output" below lists and describes the **Playsound** control output variable:

Name	Type	Default Value	Description
Playing	**Boolean**	**FALSE**	**TRUE** if **Playsound** is actively playing audio.

Table 56: Playsound control output

2.18 Pulse

Summary: This signal source produces a **PulseStream** signal.

***Note**: A **Pulse** signal is similar to the square wave except it is limited to positive amplitudes.*

Description: This signal source produces a **PulseStream** signal. *Gain*, *Frequency* and *MarkSpaceRatio* can be controlled by input variables from elsewhere in the model or from the host interface. Other signals in the model can modulate *Frequency* and *Amplitude*.

The frequency of the output pulse signal is determined by a combination of *Frequency* and *FreqModSignal*. If no signal source is connected to *FreqModSignal*, only *Frequency* is used. If a signal source is connected to the *FreqModSignal*, the actual frequency of the waveform is varied according to the amplitude of the modulating signal. The degree to which the frequency varies is controlled by *FreqModDepth*. Use the following formula to calculate the actual frequency:

$Frequency \times [1 + (FreqModDepth \times FreqModSignal)]$

When *FreqModSignal* is **Off**, its amplitude is 0.0. When the waveform frequency is less than or equal to 0, **Wave** does not generate a signal.

Gain and *AmpModSignal* determine the output pulse signal's actual amplitude. If *AmpModSignal* has no signal source, only *Gain* determines the amplitude. If a signal source is connected to *AmpModSignal*, the output pulse's amplitude varies according to the modulating signal's amplitude. The modulation depth controls how much the amplitude varies. Use the following formula to calculate the actual amplitude:

$Gain \times [1 + (AmpModDepth \times AmpModSignal)]$

When *AmpModSignal* is **Off**, its amplitude is 0.0. When the waveform amplitude is less than or equal to 0, **Pulse** does not generate a signal.

Table 57, "Pulse audio input and output" below lists and describes **Pulse** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
AmpModSignal	**audio**	N/A	The amplitude modulation signal controls the amplitude of the final output pulse. To avoid unpredictable behavior care should be taken to ensure that the product of modulation depth and modulation signal does not span a range greater than -1.0 to +1.0.
FreqModSignal	**audio**	N/A	The frequency modulation signal controls the actual generated frequency. To avoid unpredictable behavior, ensure that the product of modulation depth and modulation signal does not span a range greater than -1.0 to +1.0.
Audio Output			
OutSignal	**audio**	N/A	*OutSignal* is the output signal from **Pulse**.

Table 57: Pulse audio input and output

Table 58, "Pulse control input" on the facing page lists and describes **Pulse** control input variables:

Name	Type	Default Value	Description
AmpModDepth	**float32**	**1.0**	Controls the effect of the frequency modulation signal. Typical ranges are 0 to 1.0, when used in conjunction with a unity gain modulation signal. If no external variable is connected to *AmpModDepth*, the scaler is *FreqModDepth*. To avoid unpredictable behavior, ensure the product of *AmpModDepth* and *AmpModSignal* does not span a range greater than -1.0 to + 1.0. • *Modifier*: multiply (*) • *Modifier_default*: **0.0** • *Range*: **0.0**–**1.0**

Name	Type	Default Value	Description
Frequency	**float32**	**1.0**	Frequency in Hertz of a wave generated by the waveform synthesizer. The frequency is the number of oscillations made per second for the given waveform. If no external variable is connected to *Frequency*, the value of the scaler is used. • *Modifier*: multiply (*) • *Modifier_default*: **0.0** • *Range*: **0.0** – **1/2** × *Sample Rate*
FreqModDepth	**float32**	**1.0**	Controls the effect of the frequency modulation signal. If no external variable is connected to *FreqModDepth*, the value of the scaler is the frequency modulation depth. To avoid unpredictable behavior, ensure the product of the modulation depth and modulation signal does not span a range greater than -1.0 to +1.0. • *Modifier*: multiply (*) • *Modifier_default*: **0.0** • *Range*: **0.0**–**1.0**
Gain	**float32**	**1.0**	Amplitude gain of the waveform. If no external variable is connected to *Gain*, the value of the scaler is used. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: **0.0**–**Inf**
MarkSpaceRatio	**float32**	**1.0**	The pulse width or mark-to-space ratio of the wave. The pulse is positive relative to the period of the wave. (e.g., a value of 0.5 means in a given period, the pulse output is +ve for half the time and 0 for the other half). If no external variable is connected to *MarkSpaceRatio*, the value of the scaler is used. • *Modifier*: multiply (*) • *Modifier_default*: **0.5** • *Range*: **0.0**–**1.0**

Table 58: Pulse control input

2.19 PulseSequence

Summary: **PulseSequence** can generate a repeating series of up to eight pulses of arbitrary pulse width, pulse amplitude, and timing.

Description: **PulseSequence** is a signal component that generates a repeating series of up to eight pulses of arbitrary pulse width, pulse amplitude, and timing. Typically, this signal is used to frequency or amplitude modulate other signals. Part (a) of Figure 8, "PulseSequence timing" below shows what the various parameters in **PulseSequence**. The paint count specifies the number of pulses, while the delays, amplitudes, and durations are specified as shown. An external signal can modulate the pulse sweep time. How the pulses within the sweep act depends on whether the paint times fixed or fractional. For fixed paint times, the paint times and durations are defined in terms of the number of microseconds after the initiation of the sweep. Modulating the sweep time does not affect the pulse time. If the sweep time cuts off a pulse, it does not generate.

Part (b) shows the same pulse sequence as part (a), but the sweep time shortens in fixed paint time mode. For fractional paint times, the paint times and durations are defined in terms of the fraction of the total sweep time. Modulating the sweep time compresses (or extends) the pulses and moves them closer together or farther apart. Part (c) shows the same pulse sequence as part (a), but the sweep time shortens in fractional paint time mode.

Figure 8, "PulseSequence timing" below shows **PulseSequence** timing:

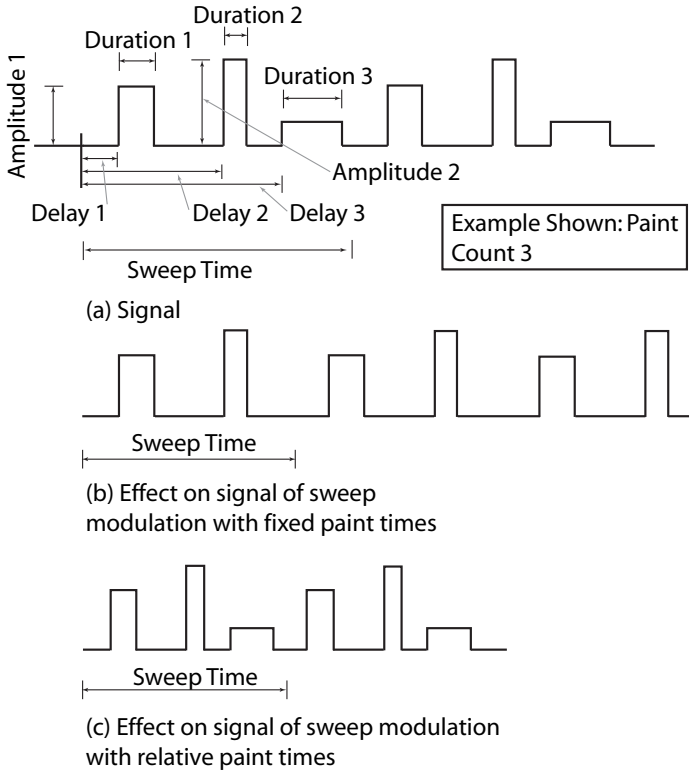


Figure 8: PulseSequence timing

Table 59, "PulseSequence audio inputs and output" below lists and describes **PulseSequence** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
GainMod	**audio**	N/A	Provides a connection to a signal that modulates the amplitude of the **PulseSequence**.
SweepMod	**audio**	N/A	Provides the audio input connection for the sweep time modulation.
Audio Output			
OutSignal	**audio**	N/A	Audio output signal from **PulseSequence**.

Table 59: PulseSequence audio inputs and output

Table 60, "PulseSequence control inputs" on the next page lists and describes the **PulseSequence** control input variables:

Name	Type	Default Value	Description
Duration1–Duration8	**float32**	**1.0**	Provides each pulse's width in microseconds (µs) (Fixed stage) or as fraction of sweep time (Fractional stage). Pulse's resolution duration is 1/sample rate. A 48 kHz sample rate = 20.833 µs.
Gain	**float32**	**1.0**	Amplitude gain of pulse. If blank, then the *Gain* scale factor is the *Gain* value. Otherwise, *Gain* = *Scale Factor* × *Output Result*, where *Output Result* represents the output result of control objects.
GainModEn	**Boolean**	**FALSE**	Control parameter enabling connection that provides amplitude gain control from elsewhere in model.
PaintCount	**uint8**	**0**	Number of pulses in a sweep. Allowable values are from 1 –8. A value of 0 disables **PulseSequence**.
SweepModEn	**Boolean**	**FALSE**	Control parameter enabling connection that provides sweep modulation control from elsewhere in model.

Name	Type	Default Value	Description
SweepModDepth	**float32**	**1.0**	Provides modulation depth for sweep modulation signal ranging from 0–1. If the modulation signal has a gain of 1, then the sweep time (or frequency) modulates between 0 and twice normal value. A modulation depth of 0 means no sweep modulation occurs.
SweepTime	**float32**	**1.0**	Provides length of sweep containing pulses in ms. For details, go to Figure 8, "PulseSequence timing" on page 48. If *PaintTimes* is **Fractional**, this variable is a frequency in Hertz. The sweep time's timing resolution = 1/sample rate. A 48 kHz sample rate has a 20.833 µs resolution.

Table 60: PulseSequence control inputs

Table 61, "PulseSequence internal parameters" below lists and describes **PulseSequence** internal parameter variables:

Name	Type	Default Value	Description
Amplitude1–Amplitude8	**float32**	**1.0**	Provides each pulse's height, ranging from 0–1.
Delay1–Delay8	**float32**	**1.0**	Delays pulse between the sweep and each pulse's start. If the pulse delay puts the pulse outside of the sweep, the pulse does not generate. *Delay's* resolution is 1/sample rate. An 8 kHz sample rate has a 125 ms resolution.
PaintTimes	**pulse_step_mode**	**fixed**	If **Fixed**, sweep time, delay times and durations are measured in microseconds. If **Fractional**, sweep equals a frequency in Hertz; duration/delays are fraction of sweep time.

Table 61: PulseSequence internal parameters

2.20 PulseStep

Summary: **PulseStep** can generate a repeating series of up to sixteen sequential pulses.

Description: **PulseStep** can generate a repeating series of up to sixteen sequential pulses of arbitrary pulse width and pulse amplitude. The pulses follow immediately one after the other. There is no space between them. Typically, this signal is used to frequency or amplitude modulate other signals.

The string of pulses can loop repeatedly with a specified loop time or can trigger in a "one-shot" sequence. When looped, another signal can modulate the loop time. If the loop time is modulated, it can shorten and lengthen the pulses (i.e., fractional step times) or keep it at a constant length (i.e., fixed step times). At the end of the pulses, the output of the signal goes to a constant amplitude, which is specified in *AmplitudeOff* (i.e., the amplitude when all the pulses are off).

In addition, an external signal can modulate the signal gain:

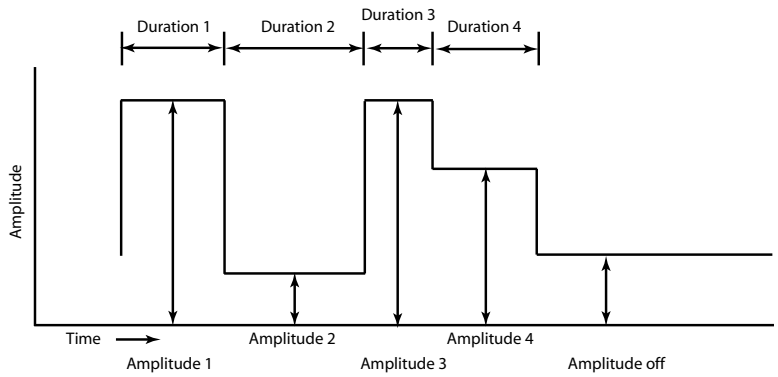


Figure 9: PulseStep signal

Table 62, "PulseStep audio inputs and output" below lists and describes **PulseStep** audio input and output variables:

Name	Type	Default Value	Description
Audio Inputs			
GainMod	**audio**	**View Scope**	Provides a connection to a signal that modulates the amplitude of the pulse sequence.
LoopMod	**audio**	N/A	Provides a connection to a control component that gives the loop modulation depth.
Audio Output			
OutSignal	**audio**	N/A	Audio output signal from **PulseStep**.

Table 62: PulseStep audio inputs and output

Table 63, "PulseStep control inputs" on the next page lists and describes **PulseStep** control input variables:

Name	Type	Default Value	Description
Amplitude1–Amplitude16	**float32**	**1.0**	Provides the amplitudes for the pulses. Go to Figure 9, "PulseStep signal" above.

Name	Type	Default Value	Description
Duration1–Duration16	**float32**	**1.0**	Provides the durations for the pulses in microseconds (i.e., fixed times) or as a fraction of the sweep time (i.e., fractional times). Go to Figure 9, "PulseStep signal" on the previous page.
Gain	**float32**	**1.0**	Amplitude gain of the signal. If *Gain* is blank, then the scale factor is the *Gain* value. Otherwise, *Gain* is calculated using the following formula: *Gain = Scale Factor × Output Result* where *Output Result* represents the control component's output result.
GainModEn	**Boolean**	**FALSE**	Enables the gain modulation.
LoopModDepth	**float32**	**1.0**	Provides the modulation depth for the loop modulation signal. This should be between 0 and 1. If the modulation signal has a gain of 1, then the loop time or frequency does modulate between zero and twice its normal value. A modulation depth of 0 means that no loop modulation occurs.
LoopModEn	**Boolean**	**FALSE**	Enables the loop modulation.
LoopTime (or frequency)	**float32**	**1.0**	*LoopTime* (i.e., fixed mode) or *LoopFrequency* (Hz) (i.e., fractional mode) determines how often the pulses repeat themselves. If *LoopTime* is 0, **PulseStep** goes into one-shot mode. In one-shot mode, the pulses generate only *StepCount* changes from 0 to a non-zero value. Otherwise, *AmplitudeOff* determines the output.
StepCount	**uint8**	**0**	An integer that provides the number of pulses. It must be between 1 and 16. Setting the value to 0 turns **PulseStream** off, so the signal output is *AmplitudeOff*. If pulse step signal is in one-shot mode, then changing *StepCount* from zero to a non-zero value triggers **PulseStream** to start.

Table 63: PulseStep control inputs

Table 64, "PulseStep internal parameters" below lists and describes **PulseStep** internal parameter variables:

Name	Type	Default Value	Description
AmplitudeOff	**float32**	**0.0**	Provides the signal's amplitude when no pulses are generated; occurs either because *StepCount* is zero or in “dead time” at the end of a loop when no more pulses exist.
PlayAll	**Boolean**	**FALSE**	Parameter is in one-shot mode. In normal mode, if the step count is toggled from 0 to a non-zero value and back to zero, pulses stop playing immediately when the *StepCount* returns to zero. In *PlayAll*, pulses finish playing to the end of sequence.
StepMode	**pulse_step_mode**	**fixed**	Determines whether pulse durations are given in absolute times in microseconds (i.e., **Fixed** mode) or as a fraction of the loop time (i.e., **Fractional** mode). When the loop time is modulated, **Fractional** mode causes pulses to shorten and lengthen in proportion to the loop time. In **Fixed** mode, pulses remain at their fixed durations when the loop time varies. If the loop time becomes less than the sum of pulse durations in **Fixed** mode, end pulses are cut off.

Table 64: PulseStep internal parameters

2.21 PulseStream

Summary: The **PulseStream** signal source is a sophisticated signal source that generates a stream of pulses. Like the **Pulse** signal, the pulses have an amplitude between 0 and 1.

Description: The **PulseStream** is typically used to frequency- or amplitude-modulate other signals, providing time-varying tones. Figure 10, "PulseStream signal" on the next page shows the pulse width and Pulse Repetition Interval (PRI). The PRI is usually the *Main_PRI* modulated by some other signal or function. There are several ways to modulate the timing between pulses. These different modulation methods are called **Pulse** types and are specified in *PulseType*. Each *PulseType* has a number from 1 to 255 and a name. For each *PulseType*, some of the parameters are ignored.

Figure 10, "PulseStream signal" below shows which *PulseTypes* use which parameters:

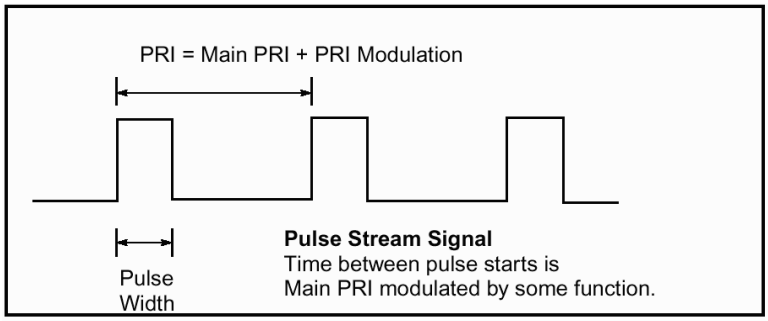
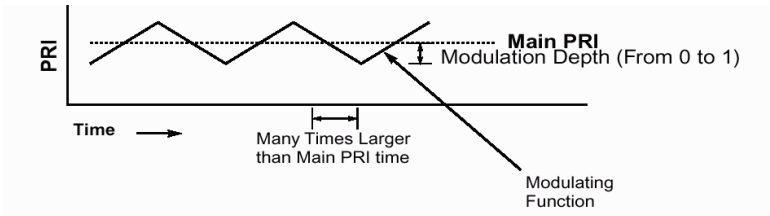


Figure 10: PulseStream signal

Figure 11, "Signal modulated" below shows how the first few *PulseTypes* work. The graph shows the PRI as a function of time. The specific example is a triangle modulated **PulseStream** (*PulseType* 3, Triangle.) Built into the object are sine, triangle, sawtooth, and square wave modulations.

To use an arbitrary signal to modulate the PRI, select *PulseType* 9 (external) and enter a signal in *Main_PRI*. For the built-in modulation signals, enter a modulation frequency and modulation depth. For the external *PulseType*, the *PRI_Mod_Freq* is ignored and *PRI_Mod_Depth* becomes a scale factor for the external modulation signal.

Figure 11, "Signal modulated" below shows the modulated signal:

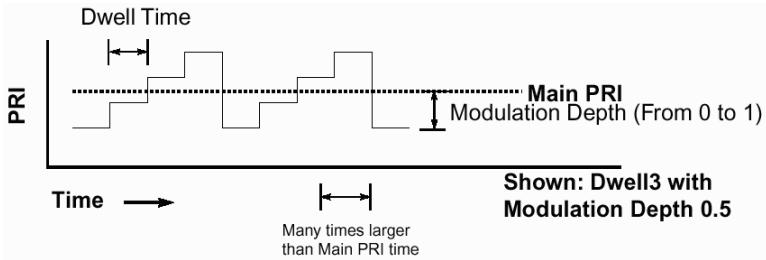


Value of PULSE TYPE field	Modulating Function	Note
2	Sine Wave	Frequency, Mod depth
3	Triangle Wave	Given by Fields in
4	Sawtooth Wave	PULSE STREAM object
5	Square Wave	
9	External Signal	Signal Connection
1,6-8,10-100,220-255	Steady - no modulation	

Figure 11: Signal modulated

PulseType 101 to 150 are called dwel1-to-dwell 50. The spacing between pulses (PRI) is modulated in a step like fashion. The length of time on each step is given by the dwell time. The number of steps is the dwell number + 1 (e.g., dwell5 has six steps). The steps are evenly spaced, and the modulation depth gives their height.

The PRI time varies between *Main_PRI* × (1 – *PRI_Mod_Depth*) and *Main_PRI* × (1 + *PRI_Mod_Depth*):



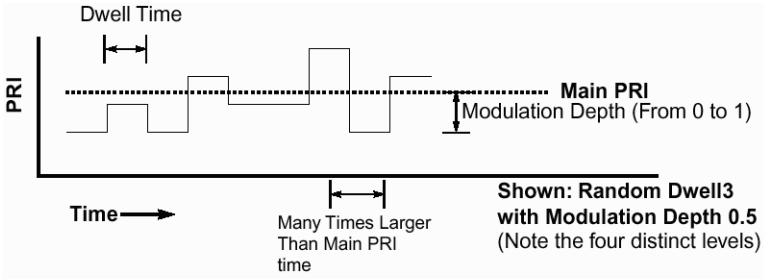
Value of PULSE TYPE field	Name	Note
101-150	Dwell1-Dwell50	Number of steps = Dwell number +1. Dwell Time and Mod Depth specified in Pulse Stream Object

Figure 12: Dwell

Figure 13, "Random Dwell" on the next page shows the Random Dwell *PulseType* (*PulseTypes* 151–199). The random dwell is the same as the dwell, except that instead of stepping sequentially through the levels it jumps randomly among them. The number of levels is given by the Random Dwell number + 1 (i.e., Random Dwell 5 would jump through six different PRI modulation times).

In addition, a Random Dwell with no number (i.e., *PulseType* 199). With this *PulseType*, the PRI time jumps randomly throughout its allowed range, staying at each PRI time for a dwell time.

The allowed range for the random dwell is between *Main_PRI* × (1 – *PRI_Mod_Depth*) and *Main_PRI* × (1 + *PRI_Mod_Depth*):

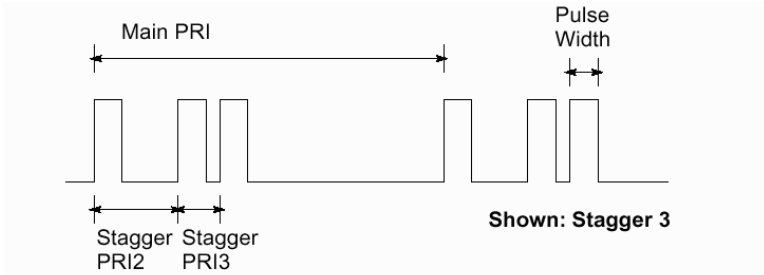


Value of PULSE TYPE field	Name	Note
151-198	Random Dwell1 to Random Dwell 48	Number of steps = Dwell number +1. Dwell Time, Mod depth specified in Pulse Stream Object
199	Random Dwell	Jumps to any level, up to Modulation Depth

Figure 13: Random Dwell

The stagger *PulseTypes* operate differently from the other *PulseType* mentioned before. Instead of modulating the PRI, up to eight PRI values define the pulse spacing. The number of pulses in the stream equals the stagger number (e.g., Stagger 4 has four pulses per cycle).

Main_PRI determines the cycle length. The stagger *PulseType* is the only one to use the Stagger PRI values. It ignores all of the PRI modulation variables, as shown in Figure 14, "Stagger" below:



Value of PULSE TYPE field	Name	Note
201-208	Stagger 1 to Stagger 8	Number of Pulses = Stagger Number.
200 209-219	Stagger 1 Stagger 8	Frequency modulation fields and dwell time are ignored

Figure 14: Stagger

Table 65, " PulseStream types" below summarizes each *PulseType* and its corresponding parameters:

Number	Name	Stagger PRIs	PRI Freq	Mod. Depth	Dwell Time
1	steady	N	N	N	N
2	sine	N	Y	Y	N
3	triangle	N	Y	Y	N
4	sawtooth	N	Y	Y	N
5	square	N	Y	Y	N
6–8	steady	N	N	N	N
9	external	N	N	Y	N
10–100	steady	N	N	N	N
101–150	dwell–dwell50	N	N	Y	Y
151–198	random dwell1–random dwell 48	N	N	Y	Y
199	random dwell	N	N	Y	Y
200	stagger1	Y	N	N	N
201–208	stagger1–stagger8	Y	N	N	N
209–219	stagger8	Y	N	N	N
220–255	steady	N	N	N	N

Table 65: PulseStream types

Table 66, "PulseStream audio inputs and output" below lists and describes **PulseStream** audio input and output variables:

Name	Type	Default Value	Description
Audio Inputs			
GainMod	**audio**	N/A	Connection to a signal which modulates the amplitude of **PulseStream**.
PRI_Mod	**audio**	N/A	Provides a connection to a signal that modulates *Main_PRI*. Only *PulseType* 9 (external) uses this variable.
Audio Outputs			
OutSignal	**audio**	N/A	*OutSignal* is the output signal from **PulseStream**, which may connect to another component or be directed to an output highway.

Table 66: PulseStream audio inputs and output

Table 67, "PulseStream control inputs" below describes **PulseStream** control input variables:

Name	Type	Default Value	Description
DwellTime	**float32**	**1.0**	Used by the dwell and random dwell pulse types (numbers 101–199). It gives the value in seconds that the PRI stays on a particular value.
Gain	**float32**	**1.0**	The amplitude gain of the **PulseStream** signal.
GainModEnable	**Boolean**	**FALSE**	Determines whether the amplitude gain modulation is enabled. Value is **TRUE** if a signal is connected to *GainMod*.
Main_PRI	**float32**	**1.0**	Provides the basic spacing between pulses, as measured from the beginning of successive pulses. This spacing can be modulated by the different pulse types. Different pulse types can modulate this spacing.
PRI_Mod_Depth	**float32**	**1.0**	A number between 0 and 1 that determines the modulation depth for the *Main_PRI* modulation. A value of 0 indicates no modulation. The range of PRI values is from *Main_PRI* × (1 – *PRI_Mod_Depth*) to *Main_PRI* × (1 + *PRI_Mod_Depth*). Steady and Stagger *PulseTypes* ignore this variable type.
PRI_Mod_Freq	**float32**	**1.0**	For *PulseTypes* 2–5 (i.e., sine, triangle, sawtooth, square), this variable provides the frequency of the signal modulating the *Main_PRI*. It is ignored for other *PulseTypes*.
PulseType	**uint8**	**0**	Determines the type of **PulseStream**. The *PulseType* is a number between 0 and 255. Each number has an associated name that appears next to the number. A *PulseTypes* of 0 turns off the **PulseStream**.
PulseWidth	**float32**	**1.0**	Provides the width of the pulses in the **PulseStream** in microseconds.
Stagger_PRI_2–Stagger_PRI_8	**float32**	**1.0**	Gives the stagger time in microseconds for the Stagger *PulseTypes*. These values are only used by the Stagger *PulseTypes* (i.e., 200–219).

Table 67: PulseStream control inputs

Table 68, "PulseStream internal parameter" below lists and describes the **PulseStream** internal parameter:

Name	Type	Default Value	Description
PulseTypeName	**string**	**OFF**	Displays the currently selected *PulseTypes*.

Table 68: PulseStream internal parameter

2.22 RecordReplay

Summary: With **RecordReplay**, you can record and replay digitally encoded sound files. This component records any signal source within a model. It also features host control inputs for controlling **RecordReplay** modes, file selection, file position, gain, and other parameters.

Description: **RecordReplay** lets you record an input signal and replay it from a prerecorded file on the Telestra server hard drive. You might use **RecordReplay** for after-action review (AAR) or in-flight replay of headset audio in the simulator.

Figure 15, "RecordReplay internal logic" below shows **RecordReplay's** internal logic:

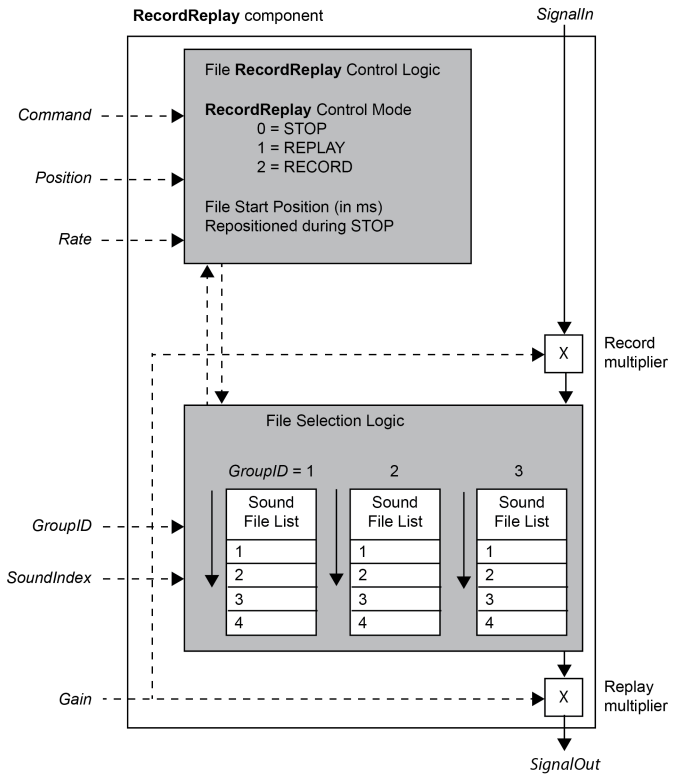


Figure 15: RecordReplay internal logic

During RECORD mode, *SignalIn* audio enters **RecordReplay**, and the signal is multiplied by *Gain*. The component stores the audio based on *Command*, *Position*, and *Rate*. It then saves the file on the Telestra server in the **/var/local/asti/recordreplay** folder and assigns it a *GroupID* and *SoundIndex* number.

GroupID and *SoundIndex* organize the sound files into groups and allow you to select a file to replay. *GroupID* represents a top-level folder storing multiple *SoundIndex* files. These files have the following name structure: **library000_group***GroupID***_index***SoundIndex***.tsr**.

During REPLAY mode, **RecordReplay** uses *GroupID* and *SoundIndex* to find the sound file. To adjust the volume, it multiplies the *SignalOut* audio by the *Gain* value. *SignalOut* then replays based on the *Command*, *Rate*, and *Position* parameters.

Position activates when *Command* leaves STOPPED (**0**) mode and begins RECORD (**2**) or REPLAY (**1**). Its meaning depends on whether the *Loop* variable is **TRUE** or **FALSE**. When *Loop* is **FALSE**, *Position* defines where the component starts recording or replaying in the file. Units (i.e., positions) are measured in milliseconds (ms).

Table 69, "Position behaviors" below defines *Position* behaviors when *Loop* is **TRUE** or **FALSE**:

Position Value	Loop=FALSE	Loop=TRUE
-1 and lower	From the end of the file, the audio position moves back the number of ms in the negative position number. For example, a -1 value produces a -1 ms offset.	From the stop position, the audio position moves back the number of ms defined in the negative position number. For example, a -1 value results in a -1 ms offset.
0	The audio resumes from the stop position.	The audio resumes from the stop position.
2	The audio plays from the beginning of the file. Set *Rate* to 48 kHz.	From the stop position, the audio position moves forward the number of ms in the positive position number. For example, a 17 value results in a +17 ms offset.
8	The audio plays from the beginning of the file. Set *Rate* to 16 kHz.	
16	The audio plays from the beginning of the file. Set *Rate* to 8 kHz.	
17 and higher	From the beginning of the file, the audio position moves forward the number of ms defined in the positive position number. For example, a 17 value produces a +17 ms offset.	

Table 69: Position behaviors

To replay the audio, convert the Telestra sound recording (.tsr) file into waveform (.wav) audio file format using the Telestra web interface. For more information about .tsr file conversion in the Telestra web interface, go to "Sound Files" in the *Telestra Web Interface User Guide*.

Telestra also includes a built-in tool that converts .tsr files into .wav files. To use this tool, go to the Telestra server's command line, and run the **tsr2wav.py** script. To convert a single .tsr file into a .wav file with a new name, run the following:

```
tsr2wav.py -i file_name.tsr -o new_file_name.wav -r NNNN
```

where *file_name* is the .tsr sound file (e.g., **library001_group002_index0001.tsr**), *new_file_name* is the converted .wav audio file (e.g., **ConvertedFile1.wav**), and *NNNN* is the input file's audio rate in hertz (Hz). The default rate is 8 kHz (i.e., 8000 Hz), but Telestra also supports 16 kHz and 48 kHz.

To convert all the .tsr files in a recording directory to .wav files, run the following:

```
tsr2wav.py -i directory_path/*.tsr -r NNNN -v
```

where *directory_path* is the location of the Telestra recordings (e.g., **/var/-local/asti/recordreplay**). If the script does not specify an output file, the converted .wav files use the original .tsr file names. The **-v** option provides additional information about the file conversion process (e.g., completion percentage, processing time in seconds), which may be useful if you are converting a large number of files.

To view a help message for the **tsr2wav.py** script, run **tsr2wave.py -h**.

Table 70, "RecordReplayaudio input and output" below lists and describes the **RecordReplay** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
SignalIn	**audio**	N/A	Audio linked to this variable is recorded. *SignalIn* is unused in REPLAY mode. The *Gain* variable controls the audio level.
Audio Output			
SignalOut	**audio**	N/A	Output signal of **RecordReplay**. *Gain* controls the audio level and reads from the raw audio **RecordReplay** file. The audio is only an output when **RecordReplay** is in REPLAY mode.

Table 70: RecordReplayaudio input and output

Table 71, "RecordReplay control input variables" on the facing page lists and describes **RecordReplay** control input variables:

Name	Type	Default Value	Description
Command	**uint8**	**0**	**RecordReplay** contains three modes: • ***0*** = STOP • ***1*** = REPLAY • ***2*** = RECORD *Command* must transition through STOP mode when it switches from RECORD to REPLAY or vice versa.
Gain	**float32**	**1.0**	Amplitude gain control for the RECORD or REPLAY audio. This variable affects both *SignalIn* and *SignalOut*.
GroupID	**uint16**	**0**	Identifies the **RecordReplay** file group. Each file is part of a specific group and may be stored on the hard drive. To point to a specific file, identify both the *GroupID* and the *SoundIndex*.
Length	**uint32**	**0**	Maximum duration of a nonlooped recording or the loop duration of a recording when *Loop* is **TRUE**. Units are in seconds.
Loop	**Boolean**	**FALSE**	Sets **RecordReplay** to continuously play or record in a loop. If **FALSE**, it stops recording or playing at the end of the file. If **TRUE**, **RecordReplay** starts recording or replaying at position 0 when it reaches the end of the file.
Position	**int32**	**0**	Activates when *Command* leaves STOPPED (**0**) mode and begins RECORD (**2**) or REPLAY (**1**) mode. Its meaning depends on whether the *Loop* variable is **TRUE** or **FALSE**. Units (i.e., positions) are measured in milliseconds (ms).
Rate	**rate_hz**	**rate_8kHz**	Record audio rate selection. Configuration options are 8 kHz, 16 kHz, and 48 kHz. Rate change requires a model reload.

Name	Type	Default Value	Description
Reset	**Boolean**	**FALSE**	Reloads the current file resource. *Reset* only works if **RecordReplay** is in STOPPED mode. It only triggers on the rising edge of the signal, when it changes from **FALSE** to **TRUE**. You might use *Reset* to swap or archive **RecordReplay** files on the server at run time. **RecordReplay** uses the following workflow: 1. Record a segment. 2. Stop recording. 3. Move the record files off the server (e.g., to archive). 4. Set *Reset* to **TRUE**. 5. Start recording again, which creates a new record file.
SoundIndex	**uint16**	**0**	Sets the recording's index number, letting you choose a specific sound file in a group. Every recording must have a unique *GroupID* and *SoundIndex*. To start replay, the value cannot be **0**.

Table 71: RecordReplay control input variables

Table 72, "RecordReplay control outputs" below lists and describes **RecordReplay** control output variables:

Name	Type	Default Value	Description
CurrentPosition	**int32**	**0**	Displays the file's position in milliseconds (ms). *CurrentPosition* updates as**RecordReplay** records or replays audio. If **RecordReplay** stops the audio, the variable holds its current value until **RecordReplay** begins to replay or record.
CurrentFileLength	**int32**	**0**	Displays the current file length in ms of the selected **RecordReplay** file.
FileMode	**player_state**	**STOPPED**	Displays the stopped, recording, or playing state of the file. It displays “RECORDING” when the file is recording and “REPLAY” when the file is replaying.
MaxFileLength	**int32**	**0**	The maximum file length is in ms, based on *Length*.
NewPosition	**int32**	**0**	The point in the file where it resumes if it begins recording or replaying.
PositionOffset	**int32**	**0**	Matches the position input and displays in ms.

Table 72: RecordReplay control outputs

2.23 SimpleMixer

Summary: The **SimpleMixer** provides quick and easy mixing of up to 32 input signals.

Description: This no-frills mixer component is the fastest way to mix audio. Up to 32 signals can be linked to the *InSignals* connection, and they are then mixed together equally. One overall *OutGain* is provided if the level needs to be moved up or down. **SimpleMixer** is most useful when central processing unit (CPU) processing is at a premium or when individual audio mixing controls are not necessary.

Table 73, "SimpleMixer audio input" below lists and describes the **SimpleMixer** audio input variable:

Name	Type	Default Value	Description
InSignals	**audio**	N/A	An input connection for up to 32 audio signals.

Table 73: SimpleMixer audio input

Table 74, "SimpleMixer control input and output" below lists and describes **SimpleMixer** control input and output variables:

Name	Type	Default Value	Description
Control Input			
OutGain	**float32**	**1.0**	Volume control for the output mix's overall level.
Control Output			
OutSignal	**audio**	N/A	The audio output of **SimpleMixer**, containing an equal amount of each input.

Table 74: SimpleMixer control input and output

2.24 Sequencer

Summary: The **Sequencer** plays audio files in a specified order.

Description: The **Sequencer** cycles through the indexes (i.e., sound files) when the trigger is set to **TRUE**. The **Sequencer** drives the **Playsound** component.

Table 75, "Sequencer control inputs" on the next page lists and describes **Sequencer** control input variables:

Name	Type	Default Value	Description
Continuous	**Boolean**	**TRUE**	Determines **Sequencer** behavior when *Trigger* is held **TRUE**. If this flag is **TRUE**, **Sequencer** continually steps through sequence. When it reaches an empty entry, it returns to the first entry and begins the sequence again. If flag is **FALSE**, **Sequencer** does only one pass through sequence. It stops sequencing when it reaches an empty entry.
Delay	**uint32**	**1**	Length of time (in seconds) between sequencing cycles; **Sequencer** is in *Continuous* mode.
Playall	**Boolean**	**TRUE**	When trigger is turned off, *Playall* determines if **Sequencer** stops immediately or plays to end of file. Defaults to **TRUE**, which plays to end of file.
Playing	**Boolean**	**TRUE**	Link from **Playsound**; informs **Sequencer** that the **Playsound** is playing.
Reset	**Boolean**	**TRUE**	If reset flag is set, **Sequencer** is set to **0.0**.

Name	Type	Default Value	Description
TriggerOut	**Boolean**	**TRUE**	**TRUE** initiates sequencing. *Continuous* mode flag determines how **Sequencer** behaves when *TriggerOut* is **FALSE**.

Table 75: Sequencer control inputs

Table 76, "Sequencer control outputs" below lists and describes **Sequencer** control output variables:

Name	Type	Default Value	Description
GroupID	**playsound_ group**	**0**	Selects groups in libraries.
SoundIdx	**playsound_ sound**	**0**	Value of the file index to the played.
TriggerOut	**Boolean**	**FALSE**	When set to **TRUE**, *TriggerOut* initiates **Sequencer**.

Table 76: Sequencer control outputs

2.25 StereoWavRecord

Summary: **StereoWavRecord** records two audio channels, allows users to name sound files in the component, and saves files in waveform (.wav) audio file format.

Description: **StereoWavRecord** records two audio channels and saves them to a file on the Telestra server hard drive. *Name* allows you to name the sound file in the user interface. The component also saves recordings as .wav files. Unlike **RecordReplay**, you can listen to audio via third-party software without converting it from Telestra sound recording (.tsr) format. **StereoWavRecord** does not have built-in replay capabilities.

Figure 16, "StereoWavRecord internal logic" below shows the **StereoWavRecord** component's internal logic:

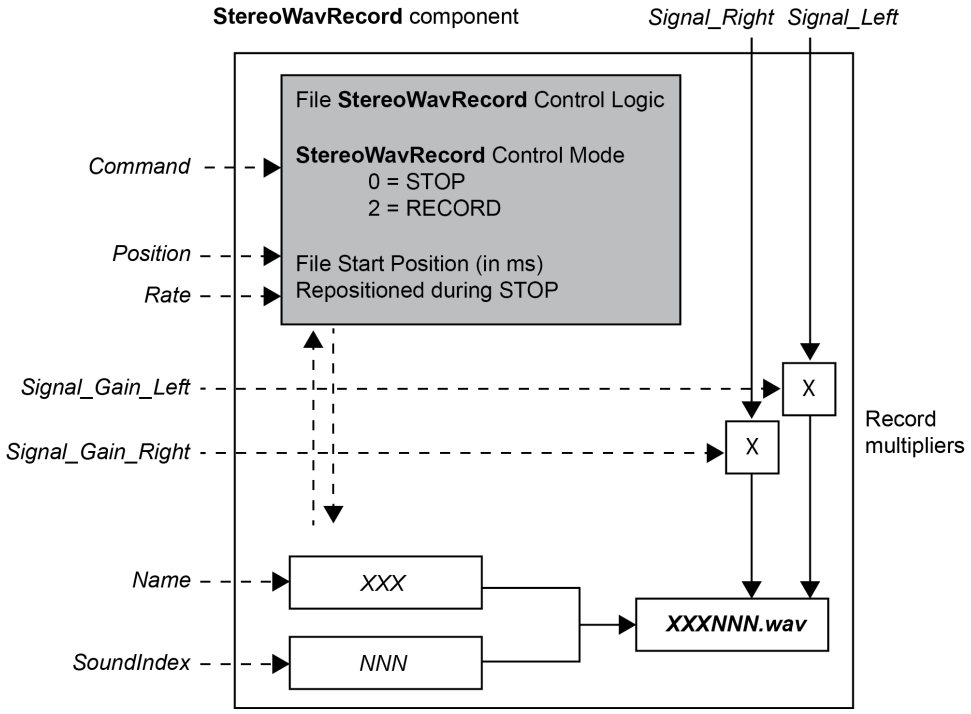


Figure 16: StereoWavRecord internal logic

During RECORD mode, *LeftInput* and *RightInput* audio enters **StereoWavRecord**, and the signals are multiplied by the *Signal_Gain_Left* and *Signal_Gain_Right* values. The component stores the audio based on *Command*, *Position*, and *Rate*. It then saves the file on the Telestra server in the default **/var/local/asti/recordreplay** folder and assigns it a *SoundIndex* number. *Name* and *SoundIndex* organize the sound files into sections, allowing you to choose a specific file. If you define a name, the file is called *NameSoundIndex***.wav** (e.g., KCR003). If *Name* is blank, the file is called **library000_group000_index***SoundIndex***.wav** by default.

Caution*: Before removing or copying files, stop RECORD mode. Otherwise, the header contains missing values.*

Position activates when *Command* leaves STOP (**0**) mode and begins RECORD (**2**) mode. *Position* defines where **StereoWavRecord** starts recording in the file. Units (i.e., positions) are measured in milliseconds (ms).

Table 77, "Position behaviors" below defines *Position* behaviors:

Position Value	Description
-1 and lower	From the end of the file, the audio position moves back the number of ms defined in the negative position number (e.g., a -1 value results in a -1 ms offset).
0	The audio resumes from the STOP position.
2	The audio plays from the beginning of the file; *Rate* must be set to 48 kHz.
4	The audio plays from the beginning of the file; *Rate* must be set to 16 kHz.
8	The audio plays from the beginning of the file; *Rate* must be set to 8 kHz.
9 and higher	From the beginning of the file, the audio position moves forward the number of ms defined in the positive position number (e.g., a 9 value results in a +9 ms offset).

Table 77: Position behaviors

Table 78, "StereoWavRecord audio inputs" below lists and describes **StereoWavRecord** audio input variables:

Name	Type	Default Value	Description
LeftInput	**audio**	N/A	Records audio from the left side of the headset. This input is divided into two subcategories: • *Signal_Gain_Left*: controls the volume for the left side. • *Signal_Left*: controls the audio input signal for the left side.
RightInput	**audio**	N/A	Records audio from the right side of the headset. This input is divided into two subcategories: • *Signal_Gain_Right*: controls the volume for the right side. • *Signal_Right*: controls the audio input signal for the right side.

Table 78: StereoWavRecord audio inputs

Table 79, "StereoWavRecord control inputs" below lists and describes **StereoWavRecord** control input variables:

Name	Type	Default Value	Description
Command	**uint8**	**0**	**StereoWavRecord** contains two modes: • ***0*** = STOP • ***2*** = RECORD
Gain	**float32**	**1**	Amplitude gain control for the RECORD audio. *Gain* affects both *LeftInput* and *RightInput*.
Length	**uint32**	**0**	Maximum duration of a nonlooped recording or the loop duration of a recording. Units are in seconds.
Name	**string**	N/A	Defines a name for the sound file.When this variable is defined, the file name is called *NameSoundIndex***.wav** (e.g., **KCR003.wav**). When blank, the file is called **library000_ group000_index***SoundIndex***.wav** by default. For best results, define a name for the file.
Position	**int32**	**0**	Activates when *Command* leaves STOP (**0**) mode and begins RECORD (**2**). Units (i.e., positions) are measured in ms.
Rate	**rate_hz**	**rate_8kHz**	Record audio rate selection. Configuration options are 8 kHz, 16 kHz, and 48 kHz. Rate change requires a model reload.
Reset	**Boolean**	**FALSE**	Reloads the current file resource. *Reset* only works if the component is in STOP mode. It only triggers on the rising edge of the signal, when changing from **FALSE** to **TRUE**. You might use *Reset* to swap or archive **StereoWavRecord** files on the server at run time. The following describes the **StereoWavRecord** workflow: 1. Record a segment. 2. Stop recording. 3. Move the record files off the server (e.g., to archive). 4. Set *Reset* to **TRUE**. 5. Start the recording again, which creates a new record file.
SoundIndex	uint16	**0**	Sets the recording's index number.

Table 79: StereoWavRecord control inputs

Table 80, "StereoWavRecord control outputs" below lists and describes **StereoWavRecord** control output variables:

Name	Type	Default Value	Description
CurrentPosition	**int32**	**0**	Displays the file's position in milliseconds. *CurrentPosition* updates as**StereoWavRecord** records audio. If the component stops the audio, the variable holds its current value until the component begins to record.
CurrentFileLength	**int32**	**0**	After a recording, displays the current file length in ms of the selected **StereoWavRecord** file.
FileMode	**player_state**	**STOP**	Displays the stopped or recording state of the file. Displays “RECORDING” when the file is recording.
MaxFileLength	**int32**	**0**	Displays the maximum file length in ms. *MaxFileLength* is based on *Length*.
NextPosition	**int32**	**0**	The point in the file where it resumes if it begins recording.
PositionOffset	**int32**	**0**	Matches the position input and displays in ms.

Table 80: StereoWavRecord control outputs

2.26 VolumeControl

Summary: **VolumeControl** provides a host control for overall volume level and muting in a model.

Description: **VolumeControl** sets the main volume level. When **Mute** is **TRUE**, the resultant volume level is 0. Table 81, "VolumeControl control inputs and output" below lists and describes **VolumeControl** control input and output variables:

Name	Type	Default Value	Description
Control Inputs			
Mute	**Boolean**	**FALSE**	When *Mute* is **TRUE**, it sets the volume level to 0.
Volume	**float32**	**1.0**	Sets main volume, overall controlled by host.
Control Output			
VolumeLevel	**float32**	**1.000000**	Resulting output of the volume setting.

Table 81: VolumeControl control inputs and output

2.27 Vox

Summary: **Vox** allows voice-activated or press-to-talk (PTT) control over an audio input signal. An optional filter may be applied to the input signal. If the filtered input signal level exceeds the **Vox** threshold level, **Vox** outputs the signal. Once the input signal level drops below the threshold, **Vox** remains active for the period of time specified by the **Vox** delay, and then it stops the output signal. Alternately, **Vox** can operate in PTT mode, where **Vox** outputs a signal only when *PTT* is set.

Description: **Vox** operates on an input signal. The source of the input signal is external audio that is connected to the **Vox** via *InSignal*. **Vox** first applies a filter (if enabled) to the input signal. The output from the filter is active only if the input signal source is active. The filter is controlled by the following parameters:

- *FilterType*
- *FilterFreq*
- *FilterQ*

FilterType determines the type of filter applied to the input signal:

- **OFF**
- **Lowpass**
- **Highpass**
- **Bandpass**
- **Notch**

If **Type** is set to **Off**, no filtering is applied to the input signal. *FilterFreq* (in Hertz) provides the roll-off frequency of the filter. The meaning of the *FilterFreq* depends on the filter type as follows:

- Filter type frequency meaning
- Off ignored
- Lowpass upper bound frequency
- Highpass lower bound frequency
- Bandpass center frequency with bandwidth controlled by *FilterQ*

FilterQ is the filter quality factor, which effectively determines the filter roll-off for lowpass and highpass filters and the filter bandwidth for bandpass filters. *FilterQ* is inversely proportional to the filter roll-off or bandwidth.

The filter parameters derive the filter coefficients for a two-pole *IIR* filter. If *FilterQ* is 0 or less, use a very low, non-zero value for the k calculation. If the **Filter Type** is **Off**, the filter output signal is the input signal.

If the filter output is active, **Vox** applies *Vox* or PTT control to the filtered signal. **Vox** includes *VoxEnable* and *PTT*. *VoxEnable* and *PTT* are specified by an externally controlled variable and an XOR logic gate control. XOR logic is applied to the external variable value and the gate control to derive a final value for the *VoxEnable* and *PTT* flags. If no external variable is connected, the value of the logic gate variable becomes the input value.

These two Boolean inputs control when the **Vox** outputs audio as follows:

1. *VoxEnable* = **FALSE**; *PTT* = **FALSE** – No Output
2. *VoxEnable* = **TRUE**; *PTT* = **FALSE** – Output active only when signal level exceeds the threshold
3. *VoxEnable* = **FALSE**; *PTT* = **TRUE** – Output active always (regardless of signal level)
4. *VoxEnable* = **TRUE**; *PTT* = **TRUE** – Output active always (regardless of signal level)

For Case 2, if the signal level drops below the threshold, **Vox** continues to output a signal for the duration specified in *VoxDelay*. The threshold level is specified by the **Vox** level control, which consists of a connection to an external control variable and an offset. The value of the external variable is added to the offset to derive the **Vox** threshold level. If no external variable is connected, the value for the offset becomes the **Vox** threshold level.

An amplitude gain control is applied to the **Vox** output. This output gain control consists of a connection to an external controlled variable and a scale factor. The value of the external variable is multiplied by the scale factor to derive the output gain. If no external variable is connected, the value for the scale factor becomes the signal gain.

Vox output audio should only be active if both of the following conditions are **TRUE**:

1. The output from the filter is active.
2. The derived output gain (*External Variable* × *Scale Factor*) is greater than 0.0.

Summary View contains two variables: *FilterSignalLevel* and *OutSignalLevel*:

- *FilterSignalLevel:* reflects the Root Mean Squared signal power at the **Vox** output.
- *OutSignalLevel:* the Root Mean Squared signal power of the output from the **Vox** component. This variable is only non-zero if the component output is active.

The *FilterSignalLevel* can be non-zero while the *OutSignalLevel* is zero. This would happen if the filtered signal does not exceed the **Vox** threshold. If the output signal is active, both variables have the same value.

Table 82, "Vox audio input and output" below lists and describes **Vox** input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	*InSignal* is the input signal into **Vox**. The filter and the **Vox** logic is applied to this signal to determine the output signal.
Audio Output			
OutSignal	**audio**	N/A	*OutSignal* is the output signal from **Vox**.

Table 82: Vox audio input and output

Table 83, "Vox control inputs" on the next page lists and describes **Vox** control input variables:

Name	Type	Default Value	Description
OutGain	**float32**	**1.0**	*OutGain* applies amplitude gain control to the primary output signal. If no external control is connected to *OutGain*, the scale factor is the *OutGain* value. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
PTT	**Boolean**	**FALSE**	In press-to-talk (PTT) mode, the input signal is the output only when *PTT* is set. In VOX mode, **Vox** behaves as if in HOT MIC mode (i.e., it always outputs the signal). If no external control is connected to *PTT*, the exclusive-or (XOR) modifier is the *PTT* value. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
VoxEnable	**Boolean**	**FALSE**	*VoxEnable* is the control variable for **Vox** mode. If this variable is **FALSE**, **Vox** is set for *PTT* mode. If no external control is connected to *VoxEnable*, the **XOR** modifier is the *VoxEnable* value. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**

Name	Type	Default Value	Description
VoxLevel	**float32**	**0.0**	*VoxLevel* provides the **Vox** threshold level. When **Vox** mode is enabled, the input signal level is compared to *VoxLevel* and fed through only if it exceeds it. If no external control is connected to *VoxLevel*, the offset is used as the *VoxLevel* value. • *Modifier*: add (**+**) • *Modifier_default*: **0.002**

Table 83: Vox control inputs

Table 84, "Vox control outputs" below lists and describes **Vox** control output variables:

Name	Type	Default Value	Description
FilterSignalLevel	**float32**	**0.0**	Indicates the sound level of the filtered signal.
OutSignalLevel	**float32**	**0.0**	The same as the filtered signal level. When the **Vox** is not outputting audio, the value is 0.

Table 84: Vox control outputs

Table 85, "Vox internal parameters" on the facing page lists and describes **Vox** internal parameters:

Name	Type	Default Value	Description
FilterFreq	**float32**	**2000.0**	*FilterFreq* (in Hertz) provides the roll-off frequency of the filter. For lowpass filters, *FilterFreq* acts as an upper bound frequency. For highpass filters, *FilterFreq* acts as a lower bound frequency. For bandpass, *FilterFreq* acts as a center frequency. *FilterQ* determines the filter bandwidth. • *Range*: (**0.0** – *Sample Rate*) ÷ **2**
FilterQ	**float32**	**0.707100**	*FilterQ* is the filter quality factor, which determines the filter roll-off for lowpass and highpass filters. The bandpass's filter bandwidth must be greater than or equal to 0.

Name	Type	Default Value	Description
FilterType	**filter_type2**	**LowPass**	Determines the filter type applied to the input signal (e.g., *Lowpass*, *Highpass*, *Bandpass*). Filtering occurs before **Vox** compares the **Vox** and signal levels. Turn *FilterType* off to disable filtering. Range: • **Off** • **LowPass** • **BandPass** • **HighPass** • **LowPassQ** • **BandPassQ** • **HighPassQ** • **Notch** Adjust the three *FilterQ*'s amplitudes so the filter has unity gain at the roll-off frequency and maintains this gain as the quality factor increases. The bandpass filters' lowpass and highpass poles are at the same roll-off frequency.
VoxDelay	**float32**	**2.0**	*VoxDelay* is the amount of time (in seconds) after the input signal level falls below the **Vox** level that **Vox** continues to output audio.

Table 85: Vox internal parameters

2.28 Wave

Summary: This signal source produces a waveform signal, which may be sine, sawtooth, triangle, or square.

Description: This signal source produces a waveform signal. Both amplitude and frequency can be controlled by input variables from elsewhere in the model or from the host interface. The frequency can also be modulated by another signal within the signal processor, with the model having control over the depth of modulation. The waveform signal type can be sine, sawtooth, triangle or square.

The waveform frequency is determined by a combination of *Frequency* and *FreqModSignal*. If no signal source is connected to the *FreqModSignal*, only *Frequency* is used.

If a signal source is connected to *FreqModSignal*, the actual frequency of the waveform varies according to the amplitude of the modulating signal. The degree to which the frequency varies is controlled by the modulation depth.

The following calculation determines the frequency:

ActualFreq = *Frequency* × [1 + (*FreqModDepth* × *FreqModSignal*)]

where *FreqModSignal* is the amplitude of the modulating signal. A modulating signal with an active state of **OFF** should be treated as an amplitude of 0.0.

When the waveform frequency is less than or equal to 0, **Wave** does not generate a signal. The amplitude of the final signal is controlled by *Gain*. When *Gain* is less than or equal to 0, **Wave** does not generate a signal.

When the waveform type is square, there is an additional control for the Mark/Space ratio. This ratio is the proportion of time, the square wave amplitude is positive to the time the square wave amplitude is negative in a given cycle. If the waveform type is not square, the Mark/Space ratio is ignored.

Table 86, "Wave audio inputs" below lists and describes the **Wave** audio input variable:

Name	Type	Default Value	Description
FreqModSignal	**audio**	N/A	Controls the actual generated frequency via the following formula: *ActualFreq* = *Frequency* × [1+ (*FreqModDepth* x *FreqModSignal*)] It usually falls between 0–1.0, when used in conjunction with a unity gain modulation signal. ***Important:*** *To avoid unpredictable behavior, ensure that the product of modulation depth and modulation signal does not span a range greater than -1.0 to +1.0.*

Table 86: Wave audio inputs

Table 87, "Wave audio outputs" below lists and describes the **Wave** audio output variable:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	*OutSignal* is the output signal from **Wave**, which maybe connected to another component or directed to an output gateway.

Table 87: Wave audio outputs

Table 88, "Wave control inputs" below lists and describes **Wave** control input variables:

Name	Type	Default Value	Description
Frequency	**float32**	**1.0**	Frequency (in Hertz) of the wave generated by the waveform synthesizer. *Frequency* equals the number of oscillations per second for the given waveform. If no external variable is connected to *Frequency*, the scalar value is used. • *Modifier*: multiply (*) • *Modifier_default*: **0.0** • *Range*: **0.0** – **1/2** × *Sample Rate*
FreqModDepth	**float32**	**1.0**	The frequency modulation depth value controls the effect of the frequency modulation signal: *ActualFreq* = *Frequency* × [1+ (*FreqModDepth* × *FreqModSignal*)] Usually falls in 0–1.0 range, when used in conjunction with unity gain modulation signal. If no external variable is connected to *FreqModDepth*, the scalar value is used as frequency modulation depth. To avoid unpredictable behavior, ensure modulation depth product and modulation signal does not span a range greater than -1.0 to +1.0. • *Modifier*: multiply (*) • *Modifier_default*: **0.0** • *Range*: **0.0**–**1.0**
Gain	**float32**	**1.0**	Amplitude gain of the waveform. If no external variable is connected to *Gain*, scalar value is used. • *Modifier*: multiply (*) • *Modifier_default*: **0.0** • *Range*: **0.0**–**Inf**
MarkSpaceInput	**float32**	**1.0**	*MarkSpaceInput* controls the duration of the square wave, plus the side relative to the wave period (e.g., value of 0.5 means in a given period, the square wave output is +ve for 1/2 time and -ve for other half). If no external variable is connected to *MarkSpaceInput*, the scalar value is used. • *Modifier*: multiply (*) • *Modifier_default*: **0.5** • *Range*: **0.0**–**1.0**

Table 88: Wave control inputs

Table 89, "Wave internal parameters" below lists and describes **Wave** internal parameter variables:

Name	Type	Default Value	Description
MarkSpaceUnits	**width_units**	**DUTY_ CYCLE**	This parameter is for the pulse and square inverse and defines the units for mark space input. The duty cycle range is 0–1. The value can also be set in seconds.
Wavetype	**waveshape**	**sine**	The type of the generated waveform. Range includes the following: • **Off** • **Sawtooth** • **Triangle** • **Sine** • **Square**

Table 89: Wave internal parameters

3.0 AudioIO

The **AudioIO** components retrieve audio from the **Highway 3D Service**. Typically, these components directly link to one of the hardware components (e.g., ACU2). The **AudioIO** components include the following:

- **Headphone3DOut**
- **HighwayOut**
- **SpeakerOut**

These components are described in the Section 9.0, "Highway 3D Service" on page 159.

4.0 CommPanel

The following section details **CommPanel** and the objects within them. **CommPanel** components include the following:

- **CommPanel4**
- **CommPanel8**
- **CommPanel16**
- **CommPanel32**
- **CommPanel8Stereo**
- **StereoCommPanel**

4.1 CommPanel 4, 8, 16, 32

Summary: Simulation of a generic communication control panel that links a single operator with up to 32 assets. A facility is provided for separate control of input, output, and sidetone routing to and from assets. Assets are linked to **CommPanel** components using the intercom service.

Description: Although many assets may be connected, only a single channel needs to be described since the remainder are simply copies of the first. The basic concept of the **CommPanel** is the connection of an operator input, usually microphone, often through a **Vox** object, through a *press-to-talk (PTT)* gate and gain stage with an optional control input and scaling factor via a control selector switch (i.e., *InControl*) to a bidirectional intercom channel provided by the Telestra intercom service. Connect the intercom channel to various other audio object types. The intercom channel can also function as a basic intercom that provides standard intercom voice communication.

The input is active if *PTT* is set (either by default value or external control), and the input audio is tagged as Active_Tx. The return signal from the service passes to the operator via an output control selector via an optional receive gain control and scaling factor.

CommPanel determines how sidetone is computed for each asset, depending on the setting of the *SidetoneLocal* mask. If the bit corresponding to a particular channel is set, then the gain is computed locally within **CommPanel**. Otherwise, sidetone is a function of the externally connected asset. *SidetoneLocal* is usually used for intercom connections between operators, while remote sidetone is used for connections to radio objects. Remote sidetone causes operators sharing an asset to hear each other's sidetone, while *SidetoneLocal* causes other operators to hear only transmitted sidetone.

Normal operation for the *SideGainControl* byte is for it to be AND'ed with the *OutControl* byte. **CommPanel**can connect to Tx/Rx intercom service channels, typically voice signals to or from radios and intercom between crew, or Rx-only signals (e.g., Nav receivers, tone sources).

Default operation is full duplex. Asset connection duplex behavior always forces local behavior, so connections to a normal radio forces half-duplex operation. *OutGain* and scaling factor determines the overall receive signal gain. *SidetoneGain* and scaling factor determine the overall sidetone signal gain. The power connection state determines if the object is operational. All operation is disabled when *Power* is turned off (**FALSE**).

Table 90, "CommPanel audio input" below lists and describes the **CommPanel** audio input variable:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	Input audio signal that **CommPanel** transmits.

Table 90: CommPanel audio input

Table 91, "CommPanel audio outputs" below lists and describes **CommPanel** audio output variables:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	Output audio signal generated by mixing all the received signals from the actively selected IC buses.
SideSignal	**audio**	N/A	Sidetone audio signal generated by mixing all the received side tones from the actively selected IC buses.

Table 91: CommPanel audio outputs

Table 92, "CommPanel4, 8, 16, 32 control inputs" on page 83 lists and describes control inputs for **CommPanels 4**, **8**, **16**, and **32**:

Name	Type	Default Value	Description
InControl	**byte**	**255**	Selects the intercom buses to transmit the *InSignal* (i.e., bit mask). For example, a value of 1 transmits on *Sig1*, a value of 2 on *Sig2*, 4 on *Sig3*, 8 on *Sig4*, 16 on *Sig5*, and 255 on all buses. • *Modifier*: and (**&**) • *Modifier_default*: **255** • *Range*: **0**–**255**

Name	Type	Default Value	Description
InGain	**float32**	**1.0**	Scales the *InSignal* audio signal. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: **0.0**–**1.0**
OutControl	**byte**	**255**	Selects the buses to receive audio from (bit mask). For example, a value of 1 receives from *Sig1*, a value of 2 on *Sig2*, 4 on *Sig3*, 8 on *Sig4*, 16 on *Sig5*, and 255 on all buses. • *Modifier*: and (**&**) • *Modifier_default*: **255** • *Range*: **0**–**255**
OutGain	**float32**	**1.0**	Volume control that scales the received *OutSignal* obtained after mixing the received signals from each intercom bus. • *Modifier*: multiply (*) • *Modifier_default*: 1.0 • *Range*: 0.0–1.0
Power	**Boolean**	**FALSE**	Controls the power of the **CommPanel**. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**
PTT	**Boolean**	**TRUE** for **CommPanel4**; **FALSE** for others	Controls **CommPanel** audio transmission. • *Modifier*: **XOR** • *Modifier_default*: **FALSE** for **CommPanel4**; **TRUE** for all others
SideControl	**byte**	**255**	Selects the buses from which to receive the sidetone signal. For example, a value of 1 receives from *Sig1*, a value of 2 from *Sig2*, from *Sig3*, 8 from *Sig4*, 16 from *Sig5*, and 255 from all buses. • *Modifier*: and (**&**) • *Modifier_default*: **255** • *Range*: **0**–**255**
SidetoneGain	**float32**	**0.6**	Scales the side (i.e., sidetone) signal obtained after mixing the received sidetone from each IC bus.
Sig1–SigN	**id**	**UNASSIGNED**	Selects the intercom bus handle.

Name	Type	Default Value	Description
Sig1_RxGain–SigN_RxGain	**float32**	**1.0**	Volume control that scales the signal received from each intercom bus prior to mixing.

Table 92: CommPanel4, 8, 16, 32 control inputs

Table 93, "CommPanel internal parameters" below lists and describes **CommPanel** internal parameter variables:

Name	Type	Default Value	Description
SideGainControl	**byte**	**255**	Selects which side tones are affected by the **CommPanel's** received signal gains. When the bit is high, the sidetone volume for the intercom bus is multiplied by the appropriate *SigN_RxGain*. Also when the bit is high, the *SideControl* value for the intercom bus becomes the logical AND of *SideControl* and *OutControl*.
SidetoneLocal	**byte**	**0**	Selects which side tones are generated locally instead of remotely. If more than one **CommPanel** is sharing the same intercom bus, this control determines if the sidetone is sent to the other panels.

Table 93: CommPanel internal parameters

4.2 CommPanel8Stereo

Summary: Simulation of a stereo communication control panel that links a single operator with up to eight assets. A facility is provided for separate control of input, output, and sidetone routing to and from assets. Assets are linked to **CommPanel** using the intercom service. **CommPanel8Stereo** has separate controls for left and right output and sidetone signals.

Description: This is a generic **CommPanel** component with stereo function. For more information, go to Section 4.0, "CommPanel" on page 80.

Table 94, "CommPanel8Stereo audio input" below describes the **CommPanel8Stereo** audio input variable:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	The input audio signal **CommPanel** transmits.

Table 94: CommPanel8Stereo audio input

Table 95, "CommPanel8Stereo audio outputs" below lists and describes **CommPanel8Stereo** audio output variables:

Name	Type	Default Value	Description
OutSignalL	**audio**	N/A	Left output audio signal generated by mixing all the received signals from the actively selected IC buses.
OutSignalR	**audio**	N/A	Right output audio signal generated by mixing all the received signals from the actively selected IC buses.
SideSignalL	**audio**	N/A	Left sidetone audio signal generated by mixing all the received sidetone signals from the actively selected IC buses.
SideSignalR	**audio**	N/A	Right sidetone audio signal generated by mixing all the received sidetone signals from the actively selected IC buses.

Table 95: CommPanel8Stereo audio outputs

Table 96, "CommPanel8Stereo control inputs" on the facing page lists and describes **CommPanel8Stereo** control input variables:

Name	Type	Default Value	Description
Power	**Boolean**	**TRUE**	Controls **CommPanel** power.
PTT	**Boolean**	**TRUE**	Controls **CommPanel** audio transmission.
InGain	**float32**	**1.0**	Scales *InSignal* audio signal.
OutGain	**float32**	**1.0**	Mixes received signals from each intercom bus and scales received *OutSignal*.
SidetoneGain	**float32**	**0.600**	Mixes received sidetone from each IC bus, and scales *SideSignal*.
InControl	**byte**	**255**	Selects intercom buses to transmit *InSignal* (i.e., bit mask). Value 1 transmits on *Sig1*, value 2 on *Sig2*, 4 on *Sig3*, 8 on *Sig4*, 16 on *Sig5*, and 255 on all buses.
OutControlL	**byte**	**255**	Selects buses receiving Rx signals routed to *OutSignalL* Value 1 receives on *Sig1*, value 2 on *Sig2*, 4 on *Sig3*, 8 on *Sig4*, 16 on *Sig5*, and 255 on all buses.

Name	Type	Default Value	Description
OutControlR	**byte**	**255**	Selects buses receiving Rx signals routed to the *OutSignalR*. Value 1 receives on *Sig1*, value 2 on *Sig2*, 4 on *Sig3*, 8 on *Sig4*, 16 on *Sig5*, and 255 on all buses.
SideControlL	**byte**	**255**	Selects buses receiving sidetone signal routed to *SideSignalL* (e.g., value 1 receives on *Sig1*, value 2 on *Sig2*, 4 on *Sig3*, 8 on *Sig4*, 16 on *Sig5*, and 255 on all buses).
SideControlR	**byte**	**255**	Selects buses receiving sidetone signal routed to *SideSignalR*. Value 1 receives on *Sig1*, value 2 on *Sig2*, 4 on *Sig3*, 8 on *Sig4*, 16 on *Sig5*, and 255 on all buses.
SidetoneLocal	**byte**	**0**	Selects which sidetone signals are generated locally vs. remotely. If more than one **CommPanel** is sharing the same intercom bus, this control determines if other panels receive sidetone.
SideGainControl	**byte**	**255**	Selects which sidetone signals are affected by the **CommPanel's** received signal gains. When the bit is high, the sidetone volume for intercom bus is multiplied by the appropriate *SigN_RxGainL* or *SigN_RxGainR*, and by *OutGain* bytes, which is masked by *OutControl* bytes.
Sig1_RxGainL–Sig8_RxGainL	**float32**	**1.0**	The volume control scaling signal received from each intercom bus before mixing into left output signal.
Sig1_RxGainR–Sig8_RxGainR	**float32**	**1.0**	The volume control scaling signal received from each intercom bus before mixing into the right output signal.
Sig1–Sig8	**id**	**UNASSIGNED**	Selects the intercom bus handle.

Table 96: CommPanel8Stereo control inputs

4.3 StereoCommPanel

ASTi created **StereoCommPanel** for a specific program. Contact ASTi for details.

Table 97, "StereoCommPanel audio input" below lists and describes the **StereoCommPanel** audio input variable:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	Input audio signal that the **CommPanel** transmits.

Table 97: StereoCommPanel audio input

Table 98, "StereoCommPanel audio outputs" below lists and describes **StereoCommPanel** audio output variables:

Name	Type	Default Value	Description
OutSignalL	**audio**	N/A	Left output audio signal generated by mixing all the received signals from the actively selected IC buses.
OutSignalR	**audio**	N/A	Right output audio signal generated by mixing all the received signals from the actively selected IC buses.
OutSignalM	**audio**	N/A	Monitor output audio signal generated by mixing all the received signals from the actively selected IC buses.
SideSignalL	**audio**	N/A	Left sidetone audio signal generated by mixing all the received sidetone signals from the actively selected IC buses.
SideSignalR	**audio**	N/A	Right sidetone audio signal generated by mixing all the received sidetone signals from the actively selected IC buses.
SideSignalM	**audio**	N/A	Monitor sidetone audio signal generated by mixing all the received sidetone signals from the actively selected IC buses.

Table 98: StereoCommPanel audio outputs

Table 99, "StereoCommPanel control inputs" on page 89 lists and describes **StereoCommPanel** control input variables:

Name	Type	Default Value	Description
InControlL	**byte**	**255**	Selects intercom buses to transmit *InSignal* (i.e., bit mask). Value 1 transmits on *Sig1*, value 2 on *Sig2*, value 4 on *Sig3*, value 8 on *Sig4*, 255 on all IC buses. Sidetone feeds to *SideSignalL*.
InControlR	**byte**	**255**	Selects intercom buses to transmit *InSignal* (i.e., bit mask). Value 1 transmits on *Sig1*, value 2 on *Sig2*, value 4 on *Sig3*, value 8 on *Sig4*, 255 on all IC buses. Sidetone feeds to *SideSignalR*.
InGain	**float32**	**1.0**	Scales *InSignal* audio signal. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: **0.0**–**1.0**
OutControlL	**byte**	**255**	Selects intercom buses to receive from (i.e., bit mask). Value 1 receives from *Sig1*, 2 from *Sig2*, 4 from *Sig3*, 8 on *Sig4*, 255 from all IC buses. Mixed audio feeds to *OutSignalL*.
OutControlR	**byte**	**255**	Selects intercom buses to receive from (i.e., bit mask). Value 1 receives from *Sig1*, 2 from *Sig2*, 3 from *Sig4*, 8 from *Sig4*, 255 from all IC buses. Mixed audio feeds to *OutSignalR*.
OutControlM	**byte**	**255**	Selects intercom buses to receive from (i.e., bit mask). Value 1 receives from *Sig1*, 2 from *Sig2*, 4 from *Sig3*, 8 from *Sig4*, 255 from all IC buses. Mixed audio feeds to *OutSignalM*.
OutGainL	**float32**	**1.0**	Scales the received *OutSignalL* signal obtained after mixing the received signals from each IC bus. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: **0.0**–**1.0**
OutGainR	**float32**	**1.0**	Scales the received *OutSignalR* signal after mixing the received signals from each IC bus. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: **0.0**–**1.0**

Name	Type	Default Value	Description
OutGainM	**float32**	**1.0**	Scales the received *OutSignalM* signal after mixing the received signals from each IC bus. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: **0.0**–**1.0**
Power	**Boolean**	**FALSE**	Controls the power bus of the **CommPanel**. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**
PTTL	**Boolean**	**FALSE**	Controls **CommPanel** audio transmission. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**
PTTR	**Boolean**	**FALSE**	Controls **CommPanel** audio transmission. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**
SideControlL	**byte**	**255**	Selects the intercom buses to receive side tones from (i.e., bit mask). Value 1 receives from *Sig1*, 2 from *Sig2*, 4 from *Sig3*, 8 from *Sig4*, 255 from all IC buses. Mixed audio feeds to *SideSignalL*.
SideControlR	**byte**	**255**	Selects intercom buses to receive side tones from (i.e., bit mask). Value 1 receives from *Sig1*, 2 from *Sig2*, 4 from *Sig3*, 8 from *Sig4*, 255 from all IC buses. Mixed audio feeds to *SideSignalR*.
SideControlM	**byte**	**255**	Selects intercom buses to receive side tones from (i.e., bit mask). Value 1 receives from *Sig1*, 2 from *Sig2*, 4 from *Sig3*, 8 from *Sig4*, 255 from all IC buses. Mixed audio feeds to *SideSignalM*.
SidetoneGainL	**float32**	**1.0**	Scales sidetone signal from *SideSignalL* audio: • *Modifier*: multiply (*) • *Modifier_default*: **0.6** • *Range*: **0.0**–**1.0**
SidetoneGainR	**float32**	**1.0**	Scales sidetone signal for *SideSignalR* audio. • *Modifier*: multiply (*) • *Modifier_default*: **0.6** • *Range*: **0.0**–**1.0**

Name	Type	Default Value	Description
SidetoneGainM	**float32**	**1.0**	Scales sidetone signal for *SideSignalM* audio. • *Modifier*: multiply (*) • *Modifier_default*: **0.6** • *Range*: **0.0**–**1.0**
Sig1_RxGainL–*Sig8_RxGainL*	**float32**	**1.0**	Scales received signals from each IC bus for *OutSignalL* audio.
Sig1_RxGainR–*Sig8_RxGainR*	**float32**	**1.0**	Scales received signals each IC bus for *OutSignalR* audio.
Sig1_RxGainM–*Sig8_RxGainM*	**float32**	**1.0**	Scales received signals from each IC bus for *OutSignalM* audio.

Table 99: StereoCommPanel control inputs

Table 100, "StereoCommPanel internal parameters" below lists and describes **StereoCommPanel** internal parameter variables:

Name	Type	Default Value	Description
SideGainControlL	**byte**	**255**	Selects which left side tones are affected by the **CommPanel's** received signal gains.
SideGainControlR	**byte**	**255**	Selects which right side tones are affected by the **CommPanel's** received signal gains.
SideGainControlM	**byte**	**255**	Selects which right side tones are affected by the **CommPanel's** received signal gains.
SidetoneLocalL	**byte**	**0**	Selects which left side tones are affected by the **CommPanel's** received signal gains.
SidetoneLocalR	**byte**	**0**	Selects which right side tones are affected by the **CommPanel's** received signal gains.
SidetoneLocalM	**byte**	**0**	Selects which monitor side tones are affected by the **CommPanel's** received signal gains.

Table 100: StereoCommPanel internal parameters

5.0 Control

Control components provide the logic for driving the functions of the objects in the model. This section documents the following **Control** components:

- **BitToByte**
- **ByteToBit**
- **ByteMerger**
- **ByteSplitter**
- **Counter**
- **Delay**
- **Ident**
- **Incrementer**
- **IntCompare**
- **IntFlexTable**
- **IntTable**
- **Latch**
- **LogicTable**
- **MathFunction**
- **NumToString**
- **PassThrough**

5.1 BitToByte

Summary: **BitToByte** combines up to eight Boolean controls into a single byte-wide value.

Description: The first input (*Input0*) becomes the least significant bit, while the eighth input (*Input7*) becomes the most significant bit. Each Boolean control may be inverted at the component input. *Gain* can function as a multiplier for the output byte if **BitToByte** operates as another conversion type function.

Table 101, "BitToByte control inputs" below lists and describes **BitToByte** control input variables:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	Overall gain applied to the output result. If *Gain* is blank, the scaler functions as *Gain*. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
Input0–Input7	**Boolean**	**FALSE**	Controls Boolean values that assembles into an 8-bit byte. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**

Table 101: BitToByte control inputs

Table 102, "BitToByte control output" below lists and describes the **BitToByte** control output variable:

Name	Type	Default Value	Description
Output	**float32**	**0.0**	Normally, an integer value equaling the sum of the binary weighting of the bits input to the component. This value may function as a floating point number if *Gain* is a floating point value.

Table 102: BitToByte control output

5.2 ByteToBit

Summary: **ByteToBit** splits an input byte into its eight, individual bit values.

Description: **ByteToBit** converts a single input byte into its eight, individual bit values.

Table 103, "ByteToBit control output" below lists and describes the **ByteToBit** control output variable:

Name	Type	Default Value	Description
BitOut0–*BitOut7*	**Boolean**	**FALSE**	The eight individual bit outputs resulting from the input byte word. *BitOut0* is the least significant, and *BitOut7* is the most significant.

Table 103: ByteToBit control output

5.3 ByteMerger

Summary: **ByteMerger** accepts four **uint8** inputs, interprets them as a multibyte value (i.e., **int16**, **uint16**, **int32**, **uint32**, or **float32**), and outputs the result as a **float64**.

Description: **ByteMerger** reconstructs an integer or float from a series of bytes. The input type specifies how the input byte should integrate. For example, if reconstructing a two-byte signed integer, link the least significant byte to *In0* and the most significant byte to *In1*.

Figure 17, "ByteMerger pinout" below shows the **ByteMerger** pinout:

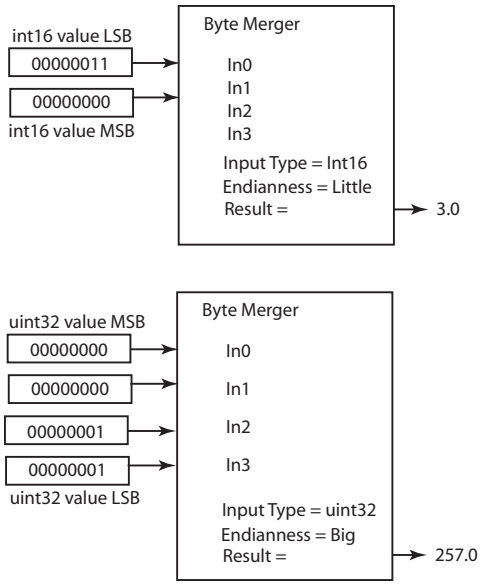


Figure 17: ByteMerger pinout

Table 104, "ByteMerger control inputs" below lists and describes **ByteMerger** control input variables:

Name	Type	Default Value	Description
In0	**uint8**	**0**	The first byte of the multibyte value.
In1	**uint8**	**0**	The second byte of the multibyte value.
In2	**uint8**	**0**	The third byte of the multibyte value.
In3	**uint8**	**0**	The fourth byte of the multibyte value.

Table 104: ByteMerger control inputs

Table 105, "ByteMerger control outputs" below lists and describes **ByteMerger** control output variables:

Name	Type	Default Value	Description
Endianness	**endianness**	**LITTLE**	The input byte order. If little endian, then the least significant byte links to *In0*. If big endian, then the most significant byte links to *In0*.
InputType	**types**	**UINT_32**	Specifies how the input bytes should be interpreted.
Result	**float64**	**0.0**	The result of merging the input bytes based on endianness and input type.

Table 105: ByteMerger control outputs

5.4 ByteSplitter

Summary: **ByteSplitter** decomposes an input value into byte-sized chunks, allowing for data type conversion at either end.

Description: When **ByteSplitter** accepts an integer or float type, you can choose the specific input type. If you choose an integer type, **ByteSplitter** only uses the value from *InInt*. If a float type is selected, **ByteSplitter** only uses the value from *InFloat*. The unused value in either case is set to 0. *InInt* and *InFloat* are constrained to **uint64** and **float64** respectively. Depending on input type (e.g., **int64** linked to the **uint64**), the result might be misrepresented. Internally, the component is aware of the actual representation of the input data, as specified by *InputType*.

Figure 18, "ByteSplitter pinout" below shows the **ByteSplitter** pinout:

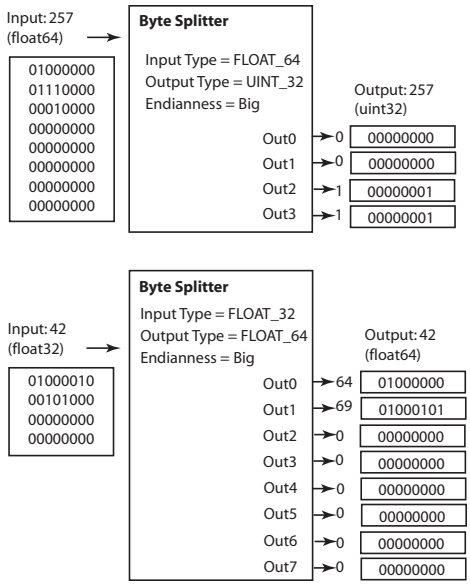


Figure 18: ByteSplitter pinout

Table 106, "ByteSplitter control inputs" below lists and describes **ByteSplitter** control input variables:

Name	Type	Default Value	Description
InInt	any integer type	0	Input port for integer-typed inputs.
InFloat	any float type	0	Input port for float-typed inputs

Table 106: ByteSplitter control inputs

Table 107, "ByteSplitter control output" below lists and describes the **ByteSplitter** control output variable:

Name	Type	Default Value	Description
Out0–Out7	**uint8**	**0**	The output of the data in byte-sized chunks. The size of the data type determines how many output ports are used.

Table 107: ByteSplitter control output

Table 108, "ByteSplitter internal parameters" below lists and describes **ByteSplitter** internal parameter variables:

Name	Type	Default Value	Description
Endianness	**endianness**	**LITTLE**	Sets the endianness of the output data. Toggling this variable reorders the data in the *Out* array.
InputType	**types2**	**INT_16**	Selects which input port (i.e., *InInt* or *InFloat*) to use and lets the component know how to interpret the bit-level representation of the data coming in on the input port. This variable allows you to input any integer or float type into the input ports.
OutputType	**types2**	**INT_16**	Allows you to convert the input value to a different data type before outputting the data in byte-sized chunks.

Table 108: ByteSplitter internal parameters

5.5 Counter

Summary: **Counter** provides a time-based, general-purpose event or continuous ramping function. In single-shot mode, **Counter** can provide an externally triggered function lookup suitable for amplitude or frequency control of dynamically generated audio sources for sound effects, such as explosions or touchdown thumps that play once for a fixed time.

When set to *Continuous* mode, **Counter** can provide a table-driven modulation of waveforms, where the modulation rate is slower than the overall model execution rate (i.e., 0 to 100 Hertz). **Counter** also provides a generic timer function for control logic within the model.

Description: The operation of **Counter** is as follows:

1. When triggered, **Counter** counts from *RangeStart* to *RangeEnd* over the period of time specified by *Duration*.
2. The current count value is passed as input *X* into an f(x) function.
3. The output of **Counter** is the function result.

As described above, *RangeStart* and *RangeEnd* define the counting sequence. If these parameters are set to the same value, **Counter** outputs 0.0. If *RangeEnd* is less than *RangeStart*, **Counter** counts down. If the *RangeEnd* is greater than the *RangeStart*, **Counter** counts up. When **Counter** is not counting, its value is 0.0.

***Note**: **Counter's** output may not necessarily be 0.0 when **Counter** is 0.0. **Counter's** output depends on the specified function.*

If *Duration* is less than zero, **Counter** uses a value of zero for **Counter** duration.

When *Continuous* is **TRUE**, **Counter** continuously cycles through the count sequence. When it reaches *RangeEnd*, it waits for the period of time specified by **Delay**. It then begins the count sequence again. If *Continuous* is **FALSE**, **Counter** resets to 0.0 when it reaches the end of the sequence.

When *CountAll* is **TRUE**, **Counter** proceeds to the end of the count sequence, even if *Trigger* is removed. At the end of the sequence, **Counter** resets to 0.0. If this variable is **FALSE**, **Counter** resets to 0.0 as soon as *Trigger* expires.

Table 109, "Counter control inputs" below lists and describes **Counter** control input variables:

Name	Type	Default Value	Description
Delay	**float32**	**1.0**	The length of time in seconds between counting cycles when *Continuous* is **TRUE**. • *Modifier*: multiply (*) • *Modifier_default*: **0.0**
Duration	**float32**	**1.0**	The length of time in seconds for **Counter** to go from *RangeStart* to *RangeEnd*. • *Modifier*: multiply (*)
Pause	**Boolean**	**FALSE**	**Counter** pauses at its current value when *Pause* is **TRUE**. • *Modifier*: **XOR**
Trigger	**Boolean**	**FALSE**	**TRUE** initiates counting. *Continuous* determines how **Counter** behaves when *Trigger* is **FALSE**. • *Modifier*: **XOR**

Table 109: Counter control inputs

Table 110, "Counter control output" below lists and describes the **Counter** control output variable:

Name	Type	Default Value	Description
Result	**float32**	**0.0**	Displays the final **Counter** result.

Table 110: Counter control output

Table 111, "Counter internal parameters" below lists and describes **Counter** internal parameter variables:

Name	Type	Default Value	Description
Continuous	**Boolean**	**FALSE**	*Continuous* determines **Counter** behavior when the *Trigger* is **TRUE**. If **TRUE**, **Counter** continually steps through the counting sequence. When it reaches *RangeEnd*, it returns to *RangeStart* and begins the counting sequence again. If **FALSE**, **Counter** only does one pass through the counting sequence. It stops counting when it reaches *RangeEnd*.
CountAll	**Boolean**	**FALSE**	Determines **Counter** behavior when *Trigger* transitions from **TRUE** to **FALSE**. If **TRUE**, **Counter** continues counting until it reaches the *RangeEnd* value when *Trigger* goes **FALSE**. If **FALSE**, **Counter** immediately stops counting when *Trigger* goes **FALSE**.
Function	**function**	**<Select>**	This service handle determines the final output based on the current **Counter** value. *Function* may be of any of the types supported by *Function* service. The most commonly used **Counter** functions are *Table* and *Comparator*. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
FunctionGain	**float32**	**1.0**	The overall gain applied to the *Function* output result.
RangeEnd	**float32**	**1.0**	Specifies the ending value of the **Counter** sequence. When triggered, **Counter** counts from *RangeStart* to *RangeEnd*.
RangeStart	**float32**	**0.0**	Specifies the starting value of the **Counter** sequence. When triggered, **Counter** counts from *RangeStart* to *RangeEnd*.
UpDown	**Boolean**	**FALSE**	Controls if **Counter** should count from *RangeStart* to *RangeEnd* then back to *RangeStart* within the *Duration* period. This feature is useful for inserting windup and wind down effects to dynamically generated aural cue sounds.

Table 111: Counter internal parameters

5.6 Delay

Summary: Postpones the input control value.

Description: **Delay** postpone a value by the specified number of k-frames. Each k-frame is 9.333 milliseconds (msec). The maximum settable delay is 750 k-frames, or approximately 7 seconds. **Delay** postpones control data (i.e., host data) within the model and may be useful for creating certain state logic.

Table 112, "Delay control inputs" below lists and describes **Delay** control input variables:

Name	Type	Default Value	Description
DelayKFrames	**int32**	**0**	The number of k-frames to delay the value. Each k-frame is 9.333 msec. Maximum setting is 750 k-frames or 7 seconds.
Enable	**Boolean**	**FALSE**	When **TRUE**, the input value is delayed. When **FALSE**, the input value is not delayed.
ValueIn	**float32**	**0.0**	An input value from another component in the model. This value is delayed and sent to *ValueOut*.

Table 112: Delay control inputs

Table 113, "Delay control output" below lists and describes the **Delay** control output variable:

Name	Type	Default Value	Description
ValueOut	**float32**	**0.0**	The delayed value, which is based on *ValueIn* and *DelayKFrames*.

Table 113: Delay control output

Table 114, "Delay internal parameter" below lists and describes the **Delay** internal parameter variable:

Name	Type	Default Value	Description
DelayMsec	**float32**	**0**	The k-frame delay expressed in milliseconds.

Table 114: Delay internal parameter

5.7 Ident

Summary: **Ident**, much like **MorseKeyer**, provides an interface between the *HostIn* packet and the model for any four-character ASCII string identifier sequence.

Description: **Ident** decodes the incoming zero-terminated ASCII string into the correct sequence of on and off pulses required for ident code communication. In addition to the usual letters and numbers defined in Morse code, it also includes the characters (* and -) to represent individual dot and dash combinations. Offset to input variable from beginning of Ethernet packet in bytes.

Table 115, "Ident control inputs" below lists and describes **Ident** control input variables:

Name	Type	Default Value	Description
CarrierMode	**Boolean**	**FALSE**	Provides control of carrier wave state gaps in the identifier. When **TRUE**, the identifier has spaces appended to the front and back of the Morse string. The carrier wave is on when not keying Morse string.
Ident	**ident**	N/A	Connection to feed *Ident* into the component.
Inverted	**Boolean**	**FALSE**	Provides local logic inversion of keying.
RepeatRate	**float32**	**1**	Repeat period in seconds for retransmission of Morse code string. • *Modifier*: **5**
WordRate	**uint8**	**1**	Determines the rate at which the word is played. Units are in dots per second. The faster the rate, the higher the number. • *Modifier*: **8**

Table 115: Ident control inputs

Table 116, "Ident control output" below lists and describes the **Ident** control output variable:

Name	Type	Default Value	Description
Result	**Boolean**	**FALSE**	Output of the Morse code toggle tone.

Table 116: Ident control output

5.8 Incrementer

Summary: Tracks the number of times an event has occurred.

Description: **Incrementer** adds one to an output count when the trigger transitions, which can track basic state information. The output reverts to zero on model load or when *Reset* triggers. *ResetAtValue* and *TimeToReset* can automatically reset the output.

Table 117, "Incrementer control inputs and output" below lists and describes **Incrementer** control input and output variables:

Name	Type	Default Value	Description
Control Inputs			
Reset	**Boolean**	**FALSE**	Resets the output count to zero when **TRUE**.
Trigger	**int32**	**0**	Increments the result by 1 when transitioning to a higher or lower number.
Control Output			
Result	**float32**	**0.0**	The output count increments by 1 when *Trigger* transitions.

Table 117: Incrementer control inputs and output

Table 118, "Incrementer internal parameters" below lists and describes **Incrementer** internal parameters:

Name	Type	Default Value	Description
ResetAtValue	**uint32**	**0**	A ceiling for the result. When *Result* reaches *ResetAtValue*, it immediately sets to zero. If *ResetAtValue* remains at 0, then *Result* counts up indefinitely.
TimeToReset	**float32**	**0.0**	A timer in seconds that begins every time *Result* increments. If the time expires before the next increment, the *Result* resets to 0. If *TimeToReset* is 0, then no timed reset occurs.
TriggerType	**trigger_ type**	**RISING_ EDGE**	Determines when the count should increment. If set to **RISING_EDGE**, *Result* increments when the trigger increases (0 to 1). If set to **FALLING_ EDGE**, *Result* increments when *Trigger* decreases (1 to 0). If **ON_TRANSITION**, *Result* increments in either case.
IncrementerValue	**uint3**	**0**	**Incrementer** value presented to function.

Table 118: Incrementer internal parameters

5.9 IntCompare

Summary: **IntCompare** checks eight input values and compares them against a range of integers. Each integer is compared against a different range. The result of all eight comparisons outputs as a single byte bit mask and a Boolean value.

Description: **IntCompare** compares multiple values at the same time. For example, compare a single frequency against multiple frequency bands to determine if you should change a radio's mode. The component replaces many **IntTables** or **In-Range MathFunctions**. Each input integer (i.e., *InputA–InputH*) is compared against its own range (i.e., *RangeA–RangeH*). *Result* is **TRUE** when all controlled inputs are within range. *Output* also provides a bit mask so you can use the result of all eight comparisons in the model. The ranges are all-inclusive.

Table 119, "IntCompare control inputs" below lists and describes **IntCompare** control input variables:

Name	Type	Default Value	Description
InputA–InputH	**uint32**	**0**	Compared against its matching range. For example, *InputA* is compared against *RangeA*, and *InputB* is compared against *RangeB*. The range for each integer is inclusive. If the input value is within range, it sets the corresponding bit in *Output* to 1. For example, if *InputA* is in range, the least significant bit of *Output* is 1; if *InputA* is out of range, it is 0. *InputB's* comparison is the second bit, and *InputH* is the most significant bit. • *Modifier*: add (**+**) • *Modifier_Default*: **0**
ControlByte	**uint8**	**255**	A bit mask that determines which input integers are used for *Result* and *Output*. When equal to 255, all eight inputs must be in range for *Result* to be **TRUE**. If equal to 1, **IntCompare** only uses *InputA* and *RangeA*. • *Modifier*: and (**&**) • *Modifier_Default*: **255**

Table 119: IntCompare control inputs

Table 120, "IntCompare control outputs" below lists and describes **IntCompare** control output variables:

Name	Type	Default Value	Description
Output	**byte**	**255**	A bit mask that contains the result of all eight comparisons. Each *Output* bit corresponds to one of the eight inputs. If the bit is 1, then the input is within range. If the bit is 0, then the input is out of range. For example, if *InputA* is within *RangeA* but the other seven inputs are out of range, *Output* equals 1.
Result	**Boolean**	**TRUE**	The master result of all comparisons. All controlled integers must be in range for *Result* to be **TRUE**. If *ControlByte* is less than 255, only the selected integers must be in range for *Result* to be **TRUE**.

Table 120: IntCompare control outputs

Table 121, "IntCompare internal parameters" below lists and describes **IntCompare** internal parameter variables:

Name	Type	Default Value	Description
RangeA–RangeH			
RangeLower	**uint32**	**0**	The lower end of each range comparison. If the input equals or is greater than *RangeLower* but less than *RangeUpper*, the comparison is in range. Each input only looks at its corresponding *RangeLower*. *InputA* looks at *RangeA*, *InputB* looks at *RangeB*, and so on.
RangeUpper	**uint32**	**0**	The upper end of each range comparison. If the input equals or is less than *RangeUpper* but greater than *RangeLower*, the comparison is in range. Each input only looks at the corresponding *RangeUpper*. *InputA* looks at *RangeA*, *InputB* looks at *RangeB*, etc.

Table 121: IntCompare internal parameters

5.10 IntFlexTable

Summary: **IntFlexTable** provides a simple lookup function for integer values. It is similar to **IntTable** with the added feature of having the lookup values as control inputs rather than internal parameters.

Description: **IntFlexTable** receives an integer input from an external connection. If this input is between 1 and 16, the result is the corresponding lookup value. For example, if *Index* is 7, *Lookup7* is used. If *Index* is outside the range, the result value is set in *IndexNot1–16*. You can set each lookup value from an external connection. Output then takes the resulting lookup value, adds the *ResultOffset* to it, and multiplies it by *Gain*.

Table 122, "IntFlexTable control inputs, output, and internal parameter" below lists and describes **IntFlexTable** control input, output, and internal parameter variables:

Name	Type	Default Value	Description
Control Input			
Gain	**float32**	**1.0**	Overall gain applied to the output result.
Index	**int32**	**0**	Index value used for table lookup. For example, when *Index* = 7, *Lookup7* is used.
Lookup1–Lookup16	**int32**	**0**	Corresponding entries for when *Index* is between 1 and 16 inclusive. If *Index* equals 7, **IntFlexTable** uses *Lookup7*. • *Modifier*: add (**+**) • *Modifier_default*: **0**
Control Output			
Output	**float32**	**0**	Final output value of **IntFlexTable**. *Output* = *Lookup* + *ResultOffset* × *Gain*.
Internal Parameter			
IndexNot1–IndexNot16	**int32**	**0**	This value is used when *Index* is less than 1 or greater than 16.

Table 122: IntFlexTable control inputs, output, and internal parameter

5.11 IntTable

Summary: **Integer Table** (**IntTable**) provides a simple lookup function for integer values. You can use **IntTable** to play a sound file index or receive/transmit selections in a communication panel.

Description: **IntTable** receives an integer input from an external connection and drives an index to add an offset. If the derived index is between 1 and 16, the result is the corresponding entry in the lookup table. If the derived index is outside of this range, the result is the *OutofRange* value. *Lookup_Table* values are hard-coded as internal parameters.

IntTable then adds an offset to the result and multiplies that value by a gain multiplier. An external connection may drive *ResultOffset*. This input provides a means to effectively change all of the lookup table values by a fixed amount dynamically. This functionality is useful if the lookup value set varies under certain logic conditions. Finally, **IntTable** applies a gain multiplier to derive the output value.

Table 123, "IntTable control inputs" below lists and describes **IntTable** control input variables:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	The overall gain applied to the *Output* result. If the gain connection is blank, then the scaler is the *Gain* value. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
Index	**int32**	**0**	*Index* used for table lookup. If no external variable is connected to *Index*, *ResultOffset* is used. • *Modifier*: add (**+**) • *Modifier_default*: **0**
ResultOffset	**int32**	**0**	An offset value added to the value in the table. If no external variable is connected to *ResultOffset*, **IntTable** uses the modifier value. • *Modifier*: add (**+**) • *Modifier_default*: **0**

Table 123: IntTable control inputs

Table 124, "IntTable control output" below lists and describes the **IntTable** control output variables:

Name	Type	Default Value	Description
Output	**float32**	**0.0**	Final output value from **IntTable**.

Table 124: IntTable control output

Table 125, "IntTable internal parameters" below lists and describes **IntTable** internal parameter variables:

Name	Type	Default Value	Description
IndexNot1–IndexNot16	**int32**	**0**	Lookup table value for result index less than 1 and greater than 16.
Lookup_Table	**int32[16]**	**1–16**	The lookup table value for result *Index1–Index16*.

Table 125: IntTable internal parameters

5.12 Latch

Summary: **Latch** holds an input value for a specified time or indefinitely.

Description: **Latch** holds a control value inside the model for other components to use. As a result, you can store an input that varies over time and send it elsewhere in the model.

Latch is ideal for sampling a rapidly changing control value such as the **Random Number MathFunction**, or for building state logic into the model in conjunction with **Incrementer**. **Latch** reads the *Input* value and sets it as the *Output* value.

The component then hold that value for the duration of *LatchTime*. After *LatchTime* expires, a new *Input* value is taken and used as the *Output* again. Alternatively, setting *LatchTime* to **0.0** holds the *Input* value indefinitely once *Enable* is **TRUE**.

Table 126, "Latch control inputs, output, and internal parameter" below lists and describes **Latch** control input, control output, and internal parameters variables:

Name	Type	Default Value	Description
Control Inputs			
Enable	**Boolean**	**FALSE**	Causes **Latch** to hold and store the current input value. When **FALSE**, the output is equal to the input. When **TRUE**, it holds the *Input* value as the *Output* value, even if *Input* changes state. If *LatchTime* is **0.0**, the *Output* does not change until *Enable* is **FALSE**. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
Input	**float32**	**1.0**	The control value that is latched and held. *Enable* must be **TRUE** to store the *Input*. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
Control Output			
Output	**float32**	**1.0**	The result of **Latch**. If *Enable* is **FALSE**, this value is equal to the *Input*. Otherwise, *Output* is equal to the value of *Input* at the time that *Enable* is **TRUE**. If *LatchTime* is not zero and *Enable* is **TRUE**, *Output* latches again after the duration of *LatchTime*. This continues to happen as long as *Enable* is **TRUE**.
Internal Parameter			
LatchTime	**float32**	**0.0**	The time, in seconds, for **Latch** to hold the *Input* value before latching a new value. If *Enable* goes **FALSE** before *LatchTime* is complete, the *Output* reverts to *Input*. A *LatchTime* of 0 causes **Latch** to store the *Input* value as long as *Enable* is **TRUE**.

Table 126: Latch control inputs, output, and internal parameter

5.13 LogicTable

Summary: **LogicTable** a mechanism for combining up to four Boolean controls into a single function. The four inputs combine to form a four-bit number that acts as an index into a 16-value array or lookup table. This array contains floating point values, combining control functions in a simple fashion.

Description: The four Boolean inputs into the component are combined into a single *Index* value as follows:

$Index = (Input0 \times 1) + (Input1 \times 2) + (Input2 \times 4) + (Input3 \times 8)$

The resulting index value is used as an *Index* into the 16-value array. The array values are set as parameters. **LogicTable** determines the array value at the given index. It then adds the array value to the chain value to derive the final output.

Table 127, "LogicTable control inputs" below lists and describes **LogicTable** control input variables:

Name	Type	Default Value	Description
Chain	**float32**	**1.0**	Value added to gain scaled *Output* from the lookup table. • *Modifier*: multiply (*) • *Modifier_default*: **0.0**
Gain	**float32**	**1.0**	Scales the *Output* from the lookup table. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**
Input0	**Boolean**	**FALSE**	A Boolean input whose value is assigned to Bit0 in the derived index value. If **TRUE**, a value of 1 is added to the derived index value. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
Input1	**Boolean**	**FALSE**	A Boolean input whose value is assigned to Bit1 in the derived index value. If **TRUE**, a value of 2 is added to the derived index value. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
Input2	**Boolean**	**FALSE**	A Boolean input whose value is assigned to Bit2 in the derived index value. If **TRUE**, a value of 4 is added to the derived index value. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
Input3	**Boolean**	**FALSE**	Boolean input whose value is assigned to Bit3 in the derived index value. If **TRUE**, a value of 8 is added to derived index value. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**

Table 127: LogicTable control inputs

Table 128, "LogicTable control output" below lists and describes the **LogicTable** control output variable:

Name	Type	Default Value	Description
Output	**float32**	**0**	Final output value from **LogicTable**.

Table 128: LogicTable control output

Table 129, "LogicTable internal parameter" below lists and describes the **LogicTable** internal parameter variable:

Name	Type	Default Value	Description
Lookup_Table	**float32[16]**	**0–15.0**	Lookup table value for result index 0–15.

Table 129: LogicTable internal parameter

5.14 MathFunction

Summary: **MathFunction** calls a user-specified function (e.g., lookup table, add, subtract, multiply, divide, etc.) that acts on the input(s). The output is the result of the specified function acting on the input variables. **MathFunction** links to a variable inside the data sink component.

Description:

MathFunction provides a variety of modifier features:

1. One, two, or three data source inputs.
2. A link to the **Function Service**, providing access to a one-variable function (i.e., f(x)), a two-variable function (i.e., f(x,y)) or a three-variable function (i.e., f(x, y, z)).

MathFunction has X, Y, and Z inputs and serves the following functions:

- *One-variable function*: links one input to a value, and a one-variable function (i.e., f(x)) is selected. The two unused inputs assume values of 1 or are unused.
- *Two-variable function*: two inputs link to values, and a two-variable function (i.e., f (x,y)) is selected. The unused input assumes a value of 1 or is unused.
- *Three-variable function*: all three inputs link to values, and a three-variable function (i.e., f(x, y, z)) is selected. Any unused input is ignored. Nondriven inputs assume a value of 1. Each input has its own constant scaling factor and multiplier operand. Each of the input scaling factors are user-selectable.

Choose the function operating on the X, Y, and Z inputs is selected using the function handle. The function handle specifies an f(x), f(x, y) or f(x, y, z) function from the Function Service. Function handles include the following:

- Table (x)
- Scale/Limit (x)
- Log/Antilog (x)
- Lag Filter (x)
- Add/ Subtract/ Multiply/ Divide (x, y)
- Random Number (x)
- Comparator / MaxMin (x)
- Switch (x, y, z)

You can modify the function evaluation with a scaling factor and multiplier operand. *Gain* is also available as a final-stage scaling factor—it modifies the scaled evaluation of the function. *Gain* has its own constant scaling factor and multiplier operand. The final result of **MathFunction** equals the scaled *Gain* value multiplied by the scaled function evaluation.

Result data types include:

- *Float*: output result of **MathFunction** and a floating point value.
- *Integer*: output result of **MathFunction**, an integer value, and a rounding of the floating point result.
- *Boolean*: output result of **MathFunction** as a Boolean. The Boolean is a digital comparison of the float value based on a 0.3 and 0.7 low and high threshold value. Below 0.3 is turned off, above 0.7 is turned on, the 0.4 difference provides a hysteresis region.

Table 132, "MathFunction internal parameter" on page 112 lists and describes **MathFunction** control input variables:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	Links to another control source to provide overall gain control of the **MathFunction** evaluation. If no external variable is connected to *Gain*, **MathFunction** uses the scaler value. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: min.–max. values of each type

Name	Type	Default Value	Description
Input_X	**float32**	**1.0**	Links to another control source (e.g., **HostIn** or **MathFunction**), which provides the first variable for **MathFunction** to use (i.e., the first stack element). If no external variable is connected to *Input_X*, the value of the scaler is used. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: minimum to maximum values of each type
Input_Y	**float32**	**1.0**	Links to another control source (e.g., **HostIn** or **MathFunction**), which provides the second variable for **MathFunction** to use (i.e., the second stack element). If no external variable is connected to *Input_Y*, the value of the scaler is used. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: minimum to maximum values of each type
Input_Z	**float32**	**1.0**	Links to another control source (**HostIn** or **MathFunction**), which provides the third variable for **MathFunction** to use (i.e., the third stack element). If no external variable is connected to *Input_Z*, the value of the scaler is used. • *Modifier*: multiply (*) • *Modifier_default*: **1.0** • *Range*: minimum to maximum values of each type

Table 130: MathFunction control inputs

Table 132, "MathFunction internal parameter" on the next page lists and describes the **MathFunction** control output variable:

Name	Type	Default Value	Description
Result	**float32**	**1.0**	The result of the input values, acted on by the specified function and internal multiplier scaling factors.

Table 131: MathFunction control output

Table 132, "MathFunction internal parameter" below lists and describes the **MathFunction** internal parameter:

Name	Type	Default Value	Description
Function	**function**	**<Select>**	The function service handle selections include: **Table** (x), **Scale** (x,y), **Random Number** (x), **Comparator /MaxMin** (x), and **Switch** (x, y, z). Range: • **Table** • **Scale** • **Limit** • **Log** • **Antilog** • **Lag filter** • **Add** • **Subtract** • **Multiply** • **Divide** • **Random** • **Comparator** • **Maxmin** • **Switch**

Table 132: MathFunction internal parameter

5.15 NumToString

Summary: **NumToString** takes a set of input numbers and converts them into a single string for use with **Radio Transceiver**. This functionality allows host control of common Distributed Interactive Simulation (DIS) parameters for radios by setting the *DomainNameIn* and *ProtocolIDIn* fields of the Radio Transceiver.

Description: **NumToString** takes up to four input numbers and combines them together to form a string. A prefix, suffix, and character separator help fit the *ProtocolID* and *DomainName* syntax for radios. The most common use of **NumToString** is to set the exercise ID of the radio by creating a string (e.g., DIS:5).

The string above can be used in the **Transceiver's** *DomainNameIn* and sets the radio's DIS exercise ID to 5. As a result, *Input0* is 5, *Prefix* is "DIS:" and *InputCount* is 1. Another component can also set *Input0*, allowing you to change the radio's exercise ID to as needed.

NumToString also produces strings such as DIS:80.1.3.17.

This string can be used in a **Transceiver's** *ProtocolIDIn* and sets Site ID to 80, Application ID to 1, Entity ID to 3, and the Radio ID to 17. With this method, another component or **HostIn** can dynamically set *Input0–Input3*, allowing you to change any of the DIS IDs for a radio on an as needed basis. To create the string above, set the prefix to "DIS:" and *InputCount* to 4. Additionally, set *Input0–Input3* to the digits used above, and use a period for the separator.

Table 133, "NumToString control inputs" below lists and describes **NumToString** control input variables:

Name	Type	Default Value	Description
Input0	**int32**	**0**	Becomes part of the output string based on conditional parameters. If used for *Domain Name: Domain Name* equals the DIS Exercise ID. If used for *Protocol ID: Protocol ID* equals the DIS Entity ID or DIS Site ID.
Input1	**int32**	**0**	Becomes part of the output string based on conditional parameters. If used for *Protocol ID*, it equals the DIS Radio ID or DIS Application ID.
Input2	**int32**	**0**	Becomes part of output string based on conditional parameters. If used for *Protocol ID*, it equals DIS Entity ID.
Input3	**int32**	**0**	Becomes part of the output string based on conditional parameters. If used for *Protocol ID*, it equals DIS Radio ID.

Table 133: NumToString control inputs

Table 134, "NumToString control output" below lists and describes the **NumToString** control output:

Name	Type	Default Value	Description
Output	**string**	**0000**	Outputs string values, maximum number of string characters = 32.

Table 134: NumToString control output

Table 135, "NumToString internal parameters" on the next page lists and describes **NumToString** internal parameter variables:

Name	Type	Default Value	Description
InputCount	**int32**	**0**	Sets the number of values defined by inputs, typically 4.
KCycles	**int32**	**0**	Currently not required.
Prefix	**string**	N/A	Attaches to front of string. Typically used for “DIS” or colon “ : ”.

Name	Type	Default Value	Description
Separator	**string**	N/A	Attaches to string to separate values. Typically used for “ . ” or space “ ”.
Suffix	**string**	N/A	Attaches to end of string and may be any string value.

Table 135: NumToString internal parameters

5.16 PassThrough

Description: **PassThrough** passes variables from **HostIn** to **HostOut**. Do not use this component for any other purpose. For example, you may want to transfer unmodified host input variables through the model and out to a separate host computer or touchscreen tablet.

6.0 Dynamics

The following section details **Dynamics** components and the objects within them, which are used to control the volume or level of signals. The **Dynamics** group includes the following components:

- **AGC** keeps a signal near a “target” level.
- **CompressorLimiter** prevents signals from getting too loud.
- **Expander** reduces the volume of quiet signals like background noise between speech.
- **Gate** only allows a signal through if it is loud enough.

Common to all four of these components are controls measured in dB and dB/sec. The dB controls such as target and threshold specify signal levels relative to 1.0 (as seen in the signal scope). Table 136, "Linear equivalents to dB values" below shows a few dB values and their linear scale equivalents.

dB Value	Linear Value
0	1.0
-6	0.5
-12	0.25
-18	0.125
---	---
-40	0.01
-60	0.001

Table 136: Linear equivalents to dB values

Speech, for example, might be around -20 dB in the system (depending on the microphone and input gains), while background noise might be around -60 dB. *Attack* and *Release* gain controls are measured in dB/sec, which is one way of specifying how quickly the gain can change. An attack of 6 dB/sec, for example, means the gain could double every second. Generally, practical *Attack* and *Release* rates are much quicker, typically in the range of 50–1000 dB/sec.

6.1 AGC

Summary: **AGC** for automatic gain control attempts to keep audio at a consistent, specified volume.

Description: **AGC** controls the signal's volume to keep the output close to a target volume. This component contains two stages: AGC and LIMITER. In the AGC stage, the component reduces signals above the target and increases signals below the target (and above the threshold). LIMITER then prevents any high peaks that AGC may have induced.

AGC can be useful for a variety of volume-leveling tasks, such as evening out microphone volumes before the input of the speech recognition system.

Table 137, "AGC audio input and output" below lists and describes **AGC** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	The signal that you will filter. This input links into **AGC** from somewhere else in the model.
Audio Output			
OutSignal	**audio**	N/A	The processed signal that has gone through AGC and LIMITER stages.

Table 137: AGC audio input and output

Table 138, "AGC control inputs" on the facing page lists and describes **AGC** control input variables:

Name	Type	Default Value	Description
Attack	**float32**	**1.0**	Determines how quickly the gain increases when the level drops below the threshold. Values are in decibels per second. Higher attack increase rate of gain change. • *Modifier:* **250.0**
Enable	**Boolean**	**TRUE**	Controls whether any gain adjustment occurs.
LimEnable	**Boolean**	**TRUE**	Turns the LIMITER stage on or off in the **AGC**. If **TRUE**, the peak output levels of the **AGC** does not go above *LimThreshold*.

Name	Type	Default Value	Description
LimThreshold	**float32**	**1.0**	Maximum peak level in dB allowed in output signal. A value of -18 corresponds to 0.125 linear. • *Modifier:* **-18.0**
OutGain	**float32**	**1.0**	Applied to the signal after AGC stage.
Ratio	**float32**	**1.0**	Controls how much the signal level advances to the *Target*. A 4.0 ratio (4:1) means a 4 dB signal from the target leaves **AGC** only 1 dB away. Higher ratios have more push. • *Modifier:* **4.0**
Release	**float32**	**1.0**	Determines how quickly gain decreases when signal level goes above *Target*. Values are in decibels per second. Higher releases aggressively reduce the Root Mean Squared gain. • *Modifier:* **100**
Target	**float32**	**1.0**	The “goal” output signal level in the Telestra web interface, measured in dB, relative to 1.0 level. As with any dB control, negative values are quieter; positive values are louder. • *Modifier:* **-24.0**
Threshold	**float32**	**1.0**	Level above which automatic gain control occurs. Adjust above noise floor, so background noise does not increase. Values are in dB, relative to the Telestra web interface 1.0 level. • *Modifier:* **-70.0**

Table 138: AGC control inputs

6.2 CompressorLimiter

Summary: **CompressorLimiter** reduces the volume of loud sounds.

Description: **CompressorLimiter** has two stages that work together to keep volume (i.e., signal levels) under control. In the first stage, COMPRESSOR reduces the volume of the signal when its average Root Mean Squared level exceeds *Threshold*. In the second stage, LIMITER then prevents peak levels from exceeding the *LimThreshold*. This functionality makes it a useful tool for preventing sound levels or clipping that could cause damage to equipment or your hearing.

Table 139, "CompressorLimiter audio inputs" below lists and describes **CompressorLimiter** audio input variables:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	The processed signal. This input links to **CompressorLimiter** from elsewhere in the model.
SideChain	**audio**	N/A	A second audio input which is used for level calculations if *SideChainEnable* is **TRUE**. In this mode, the *InSignal* is compressed only if the *SideChain* exceeds the threshold. This is useful for controlling the relative volumes of two sounds.

Table 139: CompressorLimiter audio inputs

Table 140, "CompressorLimiter audio output" below lists and describes the **CompressorLimiter** audio output variable:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	The processed signal that has gone through the COMPRESSOR and LIMITER stages.

Table 140: CompressorLimiter audio output

Table 141, "CompressorLimiter control inputs" on the facing page lists and describes **CompressorLimiter** control input variables:

Name	Type	Default Value	Description
Attack	**float32**	**1.0**	Determines how quickly the gain is turned down when the signal level goes above threshold. Values are in dB/sec. Higher attacks produce faster gain reductions. • *Modifier:* **350.0**
CompOutGain	**float32**	**1.0**	The gain applied to the signal after the COMPRESSOR stage but before the LIMITER stage.
Enable	**Boolean**	**TRUE**	Determines if the COMPRESSOR stage adjusts the gain.
LimAllowClip	**Boolean**	**FALSE**	If **TRUE**, signal peaks are hard clipped at *LimThreshold*; produces radio distortion-like effects.

Name	Type	Default Value	Description
LimEnable	**Boolean**	**TRUE**	Turns the LIMITER stage on or off in **CompressorLimiter**. If **TRUE**, its peak output levels do not exceed the *LimThreshold*.
LimRelease	**float32**	**1.0**	Rate in dB/sec that determines how quickly the LIMITER responds to drop-in signal levels. Higher releases cause a faster increase in gain to ensure the quiet signals do not reduce in volume. • *Modifier:* **100.0**
LimThreshold	**float32**	**1.0**	Maximum peak level in dB allowed in output signal. A value of -12 corresponds to 0.25 linear, or 1/4 of “full scale” signal. • *Modifier:* **-12.0**
OutGain	**float32**	**1.0**	Gain applied to signal after both the COMPRESSOR and LIMITER stages.
Ratio	**float32**	**1.0**	Controls the aggressiveness of the COMPRESSOR. A high ratio is more aggressive and means more gain reduction occurs when the signal goes above threshold. Specifically, a 4.0 ratio (4:1) means a 4 dB signal above the threshold leaves **CompressorLimiter** only 1 dB above the threshold. A 20 ratio or higher is equivalent to (∞:1), meaning the signal's average level never exceeds the threshold. • *Modifier:* **4.0**
Release	**float32**	**1.0**	Determines how quickly the gain turns when the level goes below the threshold. Values are in dB/sec. Higher releases produce more aggressive gain rate increase. • *Modifier:* **50.0**
SideChainEnable	**Boolean**	**FALSE**	If **TRUE**, *SideChain* audio input controls gain reduction on *InSignal*, rather than *InSignal* itself.
Threshold	**float32**	**1.0**	The average Telestra web interface signal level above which gain reduction occurs. Values are in dB. • *Modifier:* **-24.0**

Table 141: CompressorLimiter control inputs

6.3 Expander

Summary: **Expander** is used reduces the volume of background noise between speech.

Description: **Expander** increases the dynamic range of a signal by making quiet sounds quieter. If the average Root Mean Squared level of the signal is below the threshold, **Expander**smoothly reduces the gain on the signal. The key to using **Expander** is to adjust the threshold to be just above the noise floor. This component is useful in hot mic situations to reduce background noise transmission.

Table 142, "Expander audio inputs and outputs" below lists and describes **Expander** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	The processed signal; this input is linked into the component from somewhere else in model.
Audio Output			
OutSignal	**audio**	N/A	The processed signal that went through the **Expander** stage.

Table 142: Expander audio inputs and outputs

Table 143, "Expander control inputs" on the facing page lists and describes **Expander** control input variables:

Name	Type	Default Value	Description
Attack	**float32**	**250.0**	Controls how quickly the gain changes in response to a signal level increase. **Expander** controls how quickly the gain is restored to 1.0 when speech resumes.
Enable	**Boolean**	**TRUE**	Controls whether any gain adjustment occurs.
OutGain	**float32**	**1.0**	The gain applied after the **Expander** stage.
Ratio	**float32**	**2.0**	Controls how aggressively sounds below the threshold reduce in volume. A ratio of 2.0 (2:1) means a signal of 1 dB below exits the **Expander** 2 dB below.

Name	Type	Default Value	Description
Release	**float32**	**1.0**	Controls how quickly the gain changes in response to a decrease in signal level. High releases causes a quick reduction in background noise when speech stops. • *Modifier:* **50.0**
Threshold	**float32**	**1.0**	Level in dB below that expansion (i.e., gain reduction) occurs. Adjust threshold to just above loudest sound that reduces in volume; corresponds to just above background noise level. • *Modifier:* **-55.0**

Table 143: Expander control inputs

6.4 Gate

Summary: **Gate** only allows a signal through if it is loud enough to break the threshold.

Description: This component is similar to **Audio**/**Vox** because it only lets audio through (to a radio, etc.) if the audio level is above a set level. For example, you might adjust the threshold just below the quietest speech allowed through. **Gate** reacts to peak levels in the signal, unlike **Vox**, which looks at average levels. As a result, **Gate** is more responsive, especially to the first sounds as speech starts.

Table 144, "Gate audio input and output" below lists and describes **Gate** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	The filtered signal; links into **Gate** from somewhere else in the model.
Audio Output			
OutSignal	**audio**	N/A	Processed signal that went through **Gate**.

Table 144: Gate audio input and output

Table 145, "Gate control inputs" below lists and describes **Gate** control input variables:

Name	Type	Default Value	Description
Depth	**float32**	**1.0**	Amount of gain reduction in dB that is applied when the signal dips below the threshold. • *Modifier:* **40.0**
Enable	**Boolean**	**TRUE**	Controls whether any gain adjustment occurs.
Hold	**float32**	**1.0**	Delay in milliseconds before reducing gain on a signal after it goes below the threshold; allows the signal through **Gate** for hold time, even though it is below the threshold. • *Modifier:* **1000.0**
OutGain	**float32**	**1.0**	Applied after **Gate** stage.
PTT	**Boolean**	**FALSE**	Opens **Gate** (i.e., let the signal through), even if it is below the threshold.
Threshold	**float32**	**1.0**	Level in dB below which a signal attenuates in volume or is barred from the **Gate**. • *Modifier:* **-45.0**
VoxMode	**Boolean**	**FALSE**	If **TRUE**, **Gate** acts like the **Vox** component and causes an audio stream to go inactive if it is below the threshold. If **FALSE**, signals below the threshold attenuates according to the depth setting but remain active.

Table 145: Gate control inputs

7.0 Environmental Cue

The following section details the **Environmental Cue** components and the objects within them. The **Environmental Cue** components include:

- **5BandFilter**
- **Engine**
- **EngineLevelD**
- **FilterBank**
- **JetEngine**
- **MultiFilter**
- **PropRotor**
- **SpeakerEQ**
- **VibrationCapture**

7.1 5BandFilter

Summary: Combines five **MultiFilter** components into one component.

Description: This component essentially connects five **MultiFilter** components in series. This component therefore consumes more processing power and should be used for complex filtering schemes. The advantage of using this component over five **MultiFilter** components is that all inter-filter routing is handled within the component. For more information about **MultiFilter** components, go to Section 7.4, "MultiFilter" on page 131.

Table 146, "5BandFilter audio input and output" below lists and describes **MultiFilter** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	The audio signal to be filtered from somewhere else in the model.
Audio Output			
OutSignal	**audio**	N/A	The signal after it is filtered.

Table 146: 5BandFilter audio input and output

Table 147, "5BandFilter control inputs" on the facing page lists and describes **5BandFilter** control input variables:

Name	Type	Default Value	Description
Filter1_Enable1–Filter5_Enable5	**Boolean**	**TRUE**	Determines if the signal is filtered. If **FALSE**, the signal is not filtered, but *OutGain* is still applied. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**
Filter1_Frequency1–Filter5_Frequency5	**float32**	**1.0**	Specifies the frequency in Hertz. For *BandPass*, *BandPassUnityGain*, and *PeakingEQ*, this variable specifies passband's center frequency. For *Notch* filter, this variable specifies stopband's center frequency. For *LowPass* and *HighPass*, this variable specifies corner (-3 dB) frequency. For *LowShelf* and *HighShelf*, this variable specifies midpoint frequency. Range of is [20, 24000). • *Modifier*: multiply (*) • *Modifier_default*: **500**
Filter1_Gain_dB1–Filter5_Gain_dB5	**float32**	**0.0**	Specifies the gain value in dB. For *PeakingEQ*, *HighShelf*, and *LowShelf*, this variable specifies gain applied to passband. Range is [-50,50]. • *Modifier*: add (**+**) • *Modifier_default*: **0.0**
OutGain1–OutGain5	**float32**	**0.0**	Specifies gain in linear scale. For all filters, this variable specifies gain applied to post-filtered signal. This variable takes effect even if *Enable* results in **FALSE** and if *FilterType* is **OFF**. The range of this variable is [0, 316.2277], and a setting of 1.0 corresponds to a gain of 0 dB. These variables are only visible in **Full View**, not **Filter View**. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**

Name	Type	Default Value	Description
Filter1_QFactor1–Filter5_QFactor5	**float32**	**1.0**	Specifies bandwidth for *BandPass*, *BandPassUnityGain*, *Notch*, and *PeakingEQ* ; the higher this value, the smaller the bandwidth. For *HighPass*, *LowPass*, *HighShelf*, and *LowShelf*, this variable specifies the roll-off or corner frequency gain; the higher this value, the steeper the roll-off and higher the gain at corner frequency. This variable's range is (0, 16), and setting this variable to a 0 value or lower results in a default of 0.707107. • *Modifier*: multiply (*) • *Modifier_default*: **0.7071**

Table 147: 5BandFilter control inputs

Table 148, "5BandFilter internal parameters" below lists and describes **5BandFilter** internal parameter variables:

Name	Type	Default Value	Description
Filter1_FilterType1–Filter5_FilterType5	**filter_type3**	**LowPass**	Determines the type of filter applied to the input signal. If no filtering is desired, set *FilterType* to **OFF**.
Filter1_Order1–Filter5_Order5	**filter_order**	**_12dB_Per_Octave**	Determines the order of the filter applied to the input signal. The second, fourth, and sixth orders are available, corresponding to 12, 24, and 36 dB per octave roll-off in *LowPass*, *HighPass*, *LowShelf*, and *HighShelf* filters and 6, 12, and 18 dB per octave roll-off in *BandPass*, *BandPassUnityGain*, *Notch* and *PeakingEQ* filters, assuming *QFactor* is fixed at 0.7071. Generally, the higher this variable, the steeper the roll-offs.

Table 148: 5BandFilter internal parameters

7.2 Engine

Summary: **Engine** recreates the tones for a single jet engine.

Description: **Engine** provides a composite sound for an engine. The component includes two principal sources for noise and three independent whine tones. You can tune the overall sound based on manipulating the driving parameters for the noises, whines, and overall gain control.

Table 149, "Engine audio output" below lists and describes the **Engine** audio output variable:

Name	Type	Default Value	Description
OutputSignal	**audio**	N/A	The audio output signal from **Engine**.

Table 149: Engine audio output

Table 150, "Engine control inputs" on page 128 lists and describes **Engine** control input variables:

Name	Type	Default Value	Description
Noise1Filter1			
NoiseFilter1_GainInput	**float32**	**1.0**	Control connection for the Noise 1 gain.
GainFunction	**function**	**<Select>**	Connection to selected table or function for controlling noise gain based on *NoiseFilter1_GainInput*.
GainFunctionScalar	**float32**	**1.0**	Scaling factor for the Noise 1 gain control.
GainResult	**float32**	**1.0**	Final gain factor for the Noise Source 1.
NoiseFilter1_FrequencyInput	**float32**	**1.0**	Control connection for the Noise 1 frequency.
FrequencyFunction	**function**	**<Select>**	Connection to selected table or function for controlling noise frequency based on the *NoiseFilter1_FrequencyInput*.
FrequencyFunctionScalar	**float32**	**1.0**	Scaling factor for the Noise 1 frequency control.
FrequencyResult	**float32**	**10.0**	Final frequency factor for the Noise Source 1.
Noise2Filter2			
NoiseFilter2_GainInput	**float32**	**1.0**	Control connection for the Noise 2 gain.
GainFunction	**function**	**<Select>**	Connection to selected table or function for controlling noise gain based on the *NoiseFilter2_GainInput*.
GainFunctionScalar	**float32**	**1.0**	Scaling factor for the Noise 2 gain control.
GainResult	**float32**	**1.0**	Final gain factor for the Noise Source 2.
NoiseFilter2_FrequencyInput	**float32**	**1.0**	Control connection for the Noise 2 frequency.

Name	Type	Default Value	Description
FrequencyFunction	**function**	**<Select>**	Connection to selected table or function for controlling noise frequency based on the *NoiseFilter2_FrequencyInput*.
FrequencyFunctionScalar	**float32**	**1.0**	Scaling factor for the Noise 2 frequency control.
FrequencyResult	**float32**	**10.0**	Final frequency factor for the Noise Source 2.
WhineTone1			
WhineTone1_GainInput	**float32**	**0.0**	Control connection for the Whine 1 gain.
GainFunction	**function**	**<Select>**	Connection to selected table or function for controlling whine gain based on the *WhineTone1_GainInput*
GainFunctionScalar	**float32**	**1.0**	Scaling factor for the Whine 1 gain control.
GainResult	**float32**	**0.0**	Final gain factor for the Whine 1.
WhineTone1_Frequency	**float32**	**1.0**	Frequency in Hertz of the Whine 1 triangle wave.
WhineTone2			
WhineTone2_GainInput	**float32**	**0.0**	Control connection for the Whine 2 gain.
GainFunction	**function**	**<Select>**	Connection to selected table or function for controlling whine gain based on the *WhineTone2_GainInput*.
GainFunctionScalar	**float32**	**1.0**	Scaling factor for the Whine 2 gain control.
GainResult	**float32**	**0.0**	Final gain factor for the Whine 2.
WhineTone2_Frequency	**float32**	**1.0**	Frequency in Hertz of the Whine 2 triangle wave.
WhineTone3			
WhineTone3_GainInput	**float32**	**0.0**	Control connection for the Whine 3 Gain.
GainFunction	**function**	**<Select>**	Connection to selected table or function for controlling whine gain based on the *WhineTone3_GainInput*.
GainFunctionScalar	**float32**	**1.0**	Scaling factor for the Whine 3 gain control.
GainResult	**float32**	**0.0**	Final gain factor for the Whine 3.

Name	Type	Default Value	Description
WhineTone3_Frequency	**float32**	**1.0**	Frequency in Hertz of the Whine 3 triangle wave.
OutGain	**float32**	**1.0**	Applies amplitude gain control to output signal. If no external control is connected to *OutGain*, scale factor functions as *OutGain*.

Table 150: Engine control inputs

7.3 EngineLevelD

Summary: **EngineLevelD** is a high-fidelity engine component that can recreate intake hisses, combustion roar, blade and tip buzz, and characteristic whine tones.

Description: **EngineLevelD** provides a composite sound for engines at the highest fidelity level required by the FAA. The component includes five principal sources for noise, including the following:

- The impact of the blade movement through air
- The effects of air velocity
- The noise effect based on aircraft altitude
- The noise effect of engine fuel consumption
- The noise effect of the engine ignition

To tune the overall sound, manipulate the revolutions per minute (RPM), airspeed, fuel flow, and overall gain control.

Table 151, "EngineLevelD control inputs" on the facing page lists and describes **EngineLevelD** control input variables:

Name	Type	Default Value	Description
Airspeed	**float32**	**1.0**	Provides the control of the airspeed, which drives the intake hiss sound.
Altitude	**float32**	**1.0**	Provides a gain effect based on altitude; input affects the overall output of **EngineLevelD**.
EngineLit	**Boolean**	**TRUE**	Controls when the igniters light the fuel in the engine.

Name	Type	Default Value	Description
FuelFlow	**float32**	**1.0**	The rate of fuel consumed by the engine, typically pounds per hour.
N1_RPM	**float32**	**1.0**	Provides control for the first stage of blade RPM.
N2_RPM	**float32**	**1.0**	Provides control for the second stage of blade RPM.
OutputGain	**float32**	**1.0**	Applies the overall composite gain control.

Table 151: EngineLevelD control inputs

Table 152, "EngineLevelD internal parameters" on page 131 lists and describes **EngineLevelD** internal parameter variables:

Name	Type	Default Value	Description
Altitude			
Gain_Table	**function**	**<Select>**	Pointed to a table to control the altitude effect based on the *Altitude* input.
Gain	**float32**	**1.0**	Gain result for the *Altitude* effect.
Combustion_Fuelflow			
Freq_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the combustion effect based on FuelFlow.
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the combustion effect based on FuelFlow.
Frequency	**float32**	**10.0**	Frequency result for the combustion effect.
Gain	**float32**	**1.0**	Gain result for the combustion effect.
ExhaustRoar_FuelFlow			
Freq_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the exhaust roar effect based on FuelFlow.
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the exhaust roar effect based on FuelFlow.
Frequency	**float32**	**10.0**	Frequency result for the exhaust roar effect.
Gain	**float32**	**1.0**	Gain result for the exhaust roar effect.
IntakeHiss_Airspeed			
Freq_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the intake hiss effect based on airspeed.

Name	Type	Default Value	Description
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the intake hiss effect based on airspeed.
Frequency	**float32**	**1.0**	Frequency result for the intake hiss based on airspeed effect.
Gain	**float32**	**1.0**	Gain result for the intake hiss based on air-speed effect.
IntakeHiss_RPM			
Freq_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the intake hiss effect based on N1 RPM.
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the intake hiss effect based on N1 RPM.
Frequency	**float32**	**1.0**	Frequency result for the intake hiss based on RPM effect.
Gain	**float32**	**1.0**	Gain result for the intake hiss based on RPM effect.
RotatingMachinery_RPM			
Freq_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the rotating machinery effect based on N1 RPM.
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the rotat-ing machinery effect based on N1 RPM.
Frequency	**float32**	**1.0**	Frequency result for the rotating machinery based on RPM effect.
Gain	**float32**	**1.0**	Gain result for the rotating machinery based on RPM effect.
TurboFan1_RPM			
Freq_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the Fan effect based on N1 RPM.
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the Fan effect based on N1 RPM.
Frequency	**float32**	**1.0**	Frequency result for the Fan based on RPM effect.
Gain	**float32**	**1.0**	Gain result for the Fan based on RPM effect.
TurboFan2_RPM			

Name	Type	Default Value	Description
Freq_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the Fan effect based on N1 RPM.
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the Fan effect based on N1 RPM.
Frequency	**float32**	**1.0**	Frequency result for the Fan based on RPM effect.
Gain	**float32**	**1.0**	Gain result for the Fan based on RPM effect.
Whine_1–Whine_6			
Frequency_Table	**function**	**<Select>**	Pointer to a table to control the frequency of the Whine effect based on N1 or N2 RPM.
Gain_Table	**function**	**<Select>**	Pointer to a table to control the gain of the Whine effect based on N1 or N2 RPM.
Frequency	**float32**	**1.0**	Frequency result for the Whine based on RPM effect.
Gain	**float32**	**1.0**	Gain result for the Whine based on RPM effect.
Freq_N1(0)_Drive, Freq_N2(1)_Drive	**float32**	**0.0**	Switch to select N1 or N2 as source RPM for *Frequency_Table* drive.
Gain_N1(0)_Drive, *Gain_N2(1)_Drive*	**float32**	**0.0**	Switch to select N1 or N2 as source RPM for *Gain_Table* drive.

Table 152: EngineLevelD internal parameters

7.4 MultiFilter

Summary: Expanded version of the **Filter**. **MultiFilter** has a variety of basic filter types and the ability to define the filter order.

Description: **MultiFilter** filters a signal using one of the basic filter types described below:

- *Off*: no filtering occurs.
- *OutGain*: affects *OutSignal* amplitude.
- *LowPass*: basic *LowPass* filter; frequency content below the specified frequency is passed through and content above is attenuated.
- *HighPass*: basic *HighPass* filter; frequency content below the specified frequency is attenuated, content above is passed through.

- *BandPass*: basic *BandPass* filter; frequency content ranging from *StartBand* to *EndBand* is passed through; content outside of this range is attenuated.
- *BandPassUnityGain*: same as the *BandPass* filter except the amplitude of the *BandPassFilter* is scaled by *1/QFactor* resulting in a 0 dB peak gain, whereas the normal *BandPassFilter* has a peak gain of Q.
- *Notch*: basic *Notch* filter; frequency content at the specified frequency is attenuated.
- *AllPass*: basic *AllPass* filter; no frequency is attenuated.
- *PeakingEQ*: basic *Peak* filter; frequency content near the specified frequency is boosted or attenuated by the *Gain_dB* control.
- *LowShelf*: basic *LowShelf* filter; frequency content below the specified frequency is boosted or attenuated by the *Gain_dB* control.
- *HighShelf*: basic *HighShelf* filter; frequency content above the specified frequency is boosted or attenuated by the *Gain_dB* control.

If the filter is not enabled, the incoming *InSignal* is passed through unprocessed but potentially modified by the *OutGain* control. For a true bypass of **MultiFilter**, ensure *OutGain* results in 1.0 and *Enable* or *FilterType* results in **FALSE** or **OFF**.

Table 153, "MultiFilter audio input and output" below lists and describes **MultiFilter** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	The audio signal to be filtered.
Audio Output			
OutSignal	**audio**	N/A	The signal after it is filtered.

Table 153: MultiFilter audio input and output

Table 154, "MultiFilter control inputs" on the next page lists and describes **MultiFilter** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**TRUE**	Determines if the signal is filtered. If this variable results in **FALSE**, the signal is not filtered, but *OutGain* is still applied. • *Modifier*: **XOR** • *Modifier_default*: **FALSE**
Frequency	**float32**	**1.0**	Specifies the frequency in Hertz. For *BandPass*, *BandPassUnityGain*, and *PeakingEQ*, this variable specifies the passband's center frequency. For *Notch*, this variable specifies the stopband's center frequency. For *LowPass* and *HighPass*, this variable specifies corner (-3 dB) frequency. For *LowShelf* and *HighShelf*, this variable specifies the midpoint frequency. This variable's range is [20, 24000). • *Modifier*: multiply (*) • *Modifier_default*: **500**
Gain_dB	**float32**	**0.0**	Specifies a gain value in dB. For *PeakingEQ*, *HighShelf*, and *LowShelf*, this variable specifies gain applied to the passband. This control's range is [-50, +50]. • *Modifier*: add (**+**) • *Modifier_default*: **0.0**
OutGain	**float32**	**0.0**	Specifies the gain in linear scale. For all filters, this control specifies the gain applied to the post-filtered signal. This variable takes effect even if *Enable* is **FALSE** and *FilterType* is **OFF**. This control's range is [0, 316.2277]; a 1.0 setting corresponds to 0 db gain. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**

Name	Type	Default Value	Description
QFactor	float32	1.0	Filter quality factor that specifies the bandwidth for *BandPass*, *BandPassUnityGain*, *Notch*, and *PeakingEQ*; the higher this value, the smaller the bandwidth. *StartBand* and *EndBand* adjust according to *QFactor* and are not adjustable in display. For *HighPass*, *LowPass*, *HighShelf*, and *LowShelf*, this variable specifies roll-off/corner frequency gain; the higher this value, the steeper the roll-off and the higher the gain at the corner frequency. This control's range is (0, 16). Setting this control to a value of 0 or lower defaults the control to 0.707107. • *Modifier*: multiply (*) • *Modifier_default*: 0.7071

Table 154: MultiFilter control inputs

Table 155, "MultiFilter internal parameters" below lists and describes **MultiFilter** internal parameter variables:

Name	Type	Default Value	Description
EndBand	**float32**	**965.4772**	Specifies the end of the bandwidth affected by the filter settings. Not adjustable. For *BandPass*, *BandPassUnityGain* and *Notch*, this variable specifies the right -3 dB corner frequency in Hertz. For *PeakingEQ* filters, this specifies the midpoint frequency in Hertz.
FilterType	**filter_type3**	**LowPass**	Determines the type of filter applied to the input signal. To disable filtering, set *FilterType* to **OFF**.
Order	**filter_order**	**_12dB_Per_Octave**	Determines the order of the filter applied to the input signal. Second, fourth, and sixth orders are available, corresponding to 12, 24, and 36 dB per octave roll-off in *LowPass*, *HighPass*, *LowShelf*, and *HighShelf* filters and 6, 12, and 18 dB per octave roll-off in *BandPass*, *BandPassUnityGain*, *Notch* and *PeakingEQ*, assuming *QFactor* is fixed at 0.7071. Generally, the higher this control, the steeper the roll-offs.
StartBand	**float32**	**258.9393**	Specifies the start of the bandwidth affected by the filter settings. Not adjustable. For *BandPass*, *BandPassUnityGain*, and *Notch*, this specifies the left -3 dB corner frequency in Hertz. For *PeakingEQ*, this variable specifies the midpoint frequency in Hertz.

Table 155: MultiFilter internal parameters

7.5 PropRotor

Summary: **PropRotor** generates the composite sound for a rotating helicopter blade.

Description: **PropRotor** includes the three principal sources of noise:

- Air noise from the movement of air over the blades
- Force noise from the impact of the blade with the air medium
- Thickness of noise due to the dual edge sound sources on a blade

Tune the overall sound based upon blade parameters, such as radius and blade count. Overall gain control is based on both revolutions per minute (RPM) and blade angle.

Table 156, "PropRotor audio input and output" below lists and describes the **PropRotor** audio input and output variables:

Name	Type	Default Value	Description
Audio Input			
InSignal	**audio**	N/A	Link to external noise source. To be used instead of the **PropRotor** internal noise source if desired.
Audio Output			
OutputSignal	**audio**	N/A	Audio output from **PropRotor**.

Table 156: PropRotor audio input and output

Table 157, "PropRotor control inputs" below lists and describes **PropRotor** control input variables:

Name	Type	Default Value	Description
Angle	**float32**	**1.0**	Provides the blade angle value.
ForceGain	**float32**	**1.0**	The gain for the principle component of the blade sound due to the air force on the blade. If a function connection is not made, this gain equals the scale factor.
LagFilter	**float32**	**1.0**	Provides a limiting filter used for fade-in and fade-out effects.
NoiseGain	**float32**	**1.0**	The gain for the air noise part of the blade sound. If the function connection is not made, then *Noise Gain* = *Scale Factor*.
OutputGain	**float32**	**1.0**	Amplitude gain of prop/rotor output. If the gain connection is blank, then the gain scale factor equals the gain value. Otherwise, *Gain* = *Scale Factor* × *Output*, where *Output* represents the output result of the control component.
Radius	**float32**	**1.0**	Scales the tip mach speed based on the blade radius (m).
RPM	**float32**	**1.0**	Frequency in RPM of the rotor shaft.
ThicknessGain	**float32**	**1.0**	Gain of the airflow sound determined by blade thickness.
QFactor	**float32**	**1.0**	Quality factor for the air noise filter. If the *QFactor* connection is blank then the *Q* scale factor equals the *Q* value. *Q* = *Scale Factor* × *Output*, where *Output* represents the output result of the control object.

Table 157: PropRotor control inputs

Table 158, "PropRotor internal parameters" below lists and describes **PropRotor** internal parameter variables:

Name	Type	Default Value	Description
FilterFreqScale	**float32**	**4.0**	The scale factor for noise filter roll-off frequency.
FilterType	**filter_type2**	**BandPassQ**	Selects a two-pole filter type from the following: • *Lowpass* • *Bandpass* • *Highpass* • *LowpassQ* • *BandpassQ* • *HighpassQ* The latter three are amplitude-adjusted such that the filter has unity gain at the roll-off frequency and maintains this gain as the quality factor increases. For the bandpass filters, the lowpass and highpass poles have the same roll-off frequency.
MachLimit	**float32**	**.950**	A limit for the calculated tip mach speed. This keeps the maximum mach speed to a predetermined maximum. Usually 0.95 and 0.99, depending how dominant a thickness noise is required. The sound model is not accurate above 0.99 since supersonic effects start to dominate the sound spectrum.
NumberOfBlades	**uint8**	**6**	Number of blades on the shaft.

Table 158: PropRotor internal parameters

7.6 SpeakerEQ

Summary: **SpeakerEQ** automatically equalizes a speaker by comparing referenced and measured sound responses.

Description: **SpeakerEQ** has three different modes of operation. PASS_THROUGH mode sends the *ModelIn* input signal directly to the speaker, disabling the filters. Use this mode to initially test the component during model setup. A *ModeControl* value of **0** initiates PASS_THROUGH mode.

ANALYZE_REFERENCE mode compares reference audio (i.e., *ReferenceIn*) to microphone audio (i.e., *MicIn*) and generates filters for speaker equalization. In other words, **SpeakerEQ** changes the filter settings applied to the *ModelIn* signal and analyzes how those changes impact the *MicIn* signal. A *ModeControl* value of **1** sets the component to ANALYZE_ REFERENCE mode.

During ANALYZE_SPEAKER mode, **SpeakerEQ** routes *ModelIn* audio from the simulation's sound model to the speaker. A reference microphone connected to an ACU2 collects and returns the speaker frequency response via the *MicIn* input. **SpeakerEQ** then compares the microphone audio to the pink noise *ReferenceIn* signal, which represents a flat frequency response. **SpeakerEQ** applies a set of filters to the *MicIn* signal so the speaker's frequency response matches *ReferenceIn* and *ModelIn*. It then routes the equalized sound back to the speaker via *OutSignal*. By default, analysis takes 90 seconds to complete and automatically transitions to PROCESS mode.

***Note**: While a flat frequency response is ideal in most cases, the component can match any reference response curve.*

Figure 19, "SpeakerEQ Analyze mode" below shows **SpeakerEQ's** feedback loop during ANALYZE mode:

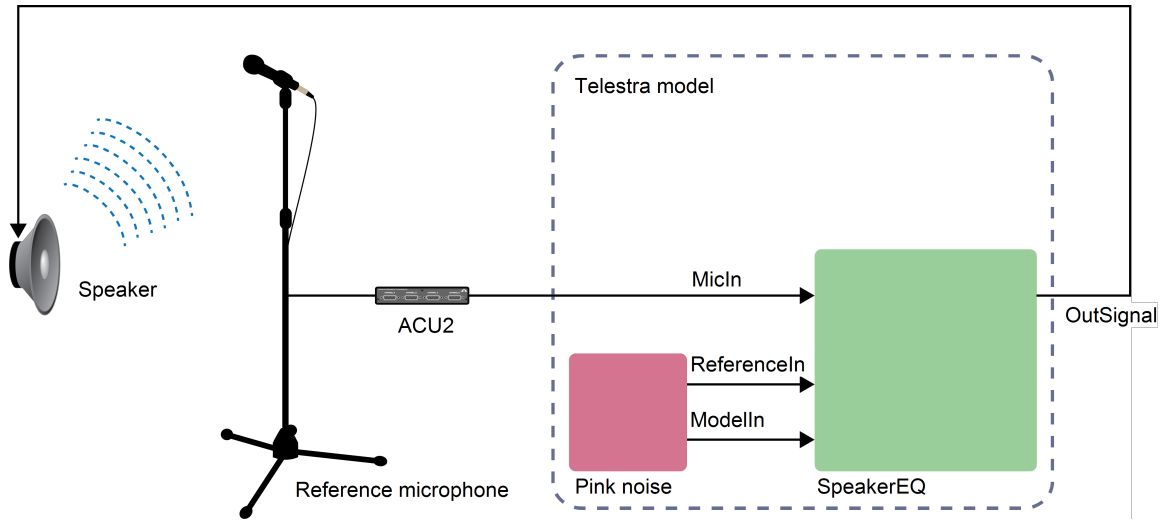


Figure 19: SpeakerEQ ANALYZE mode

PROCESS mode maintains the speaker's equalization. During PROCESS mode, the component applies a bank of audio filters to the *ModelIn* signal and sends the result to *OutSignal*. During PROCESS mode, it saves the filters' frequencies and gains to a file and recalls them during project installation. Back up and restore these speaker-specific features in the Telestra web interface. A *ModeControl* value of **3** initiates PROCESS mode.

Table 159, "SpeakerEQ audio inputs" below lists and describes **SpeakerEQ's** audio input variables:

Name	Type	Default Value	Description
MicIn	**audio**	N/A	Audio coming from the spectral analysis microphone.
ReferenceIn	**audio**	N/A	A signal with a frequency response curve. This input sets a reference signal used in the analysis. Most of the time, *ReferenceIn* uses pink noise, which represents a flat frequency response (i.e., feedback with minimal distortion from the speaker). *ReferenceIn* is only used for tuning.
ModelIn	**audio**	N/A	Simulation audio from the **AuralCue** sound model to which you will apply the filter.

Table 159: SpeakerEQ audio inputs

Table 160, "SpeakerEQ audio output" below lists and describes **SpeakerEQ's** audio output variable:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	Filtered audio output that usually routes to the speaker.

Table 160: SpeakerEQ audio output

Table 161, "SpeakerEQ control inputs" below lists and describes **SpeakerEQ** control input variables:

Name	Type	Default Value	Description
ModeControl	**uint8**	**0**	A value of **1** runs **SpeakerEQ** through two states: • ANALYZE: compares reference audio to microphone audio • PROCESS: applies a filter yielding a flat response A value of **0** sets *ModeControl* in PASS_THROUGH mode, which disables the current filter. A value of **3** sets **SpeakerEQ's** state to PROCESS using the default saved filter, if one exists.
MaxBoost_dB	**float64**	**6.0**	The maximum filter boost that the filter applies for any given third octave frequency band, measured in decibels. If you notice a steady state difference between the filtered audio and reference audio, increase the value by that number.

Table 161: SpeakerEQ control inputs

Table 162, "SpeakerEQ control output" below lists and describes the **SpeakerEQ** control output variable:

Name	Type	Default Value	Description
ModeOut	**ModeOut**	N/A	**SpeakerEQ's** current state: • **0** = PASS_THROUGH • **1** = ANALYZE • **3** = PROCESS Use this variable to control host feedback. When equalization is complete, *ModeOut* automatically changes to PROCESS.

Table 162: SpeakerEQ control output

Table 163, "SpeakerEQ internal parameters" below lists and describes **SpeakerEQ** internal parameters and debugging variables:

Name	Type	Default Value	Description
SpeakerNumber	**int32**	**0**	A unique number that identifies each speaker.
NumTuningPasses	**int32**	**10**	The total number of passes. For higher accuracy, let *NumTuningPasses* cycle more than 10 times.
PassDuration	**float64**	**9.0**	The approximate duration of each pass in seconds.
AnalysisCutoffFreqHigh	**int32**	**0**	Above a certain frequency, does not apply the correction filter to the signal; measured in hertz (Hz). For example, a typical high cutoff frequency for a subwoofer is 200 Hz.
AnalysisCutoffFreqLow	**int32**	**0**	Below a certain frequency, this variable does not apply the correction filter to the signal; measured in Hz. For example, a typical low cutoff frequency for a bookshelf speaker is 150 Hz.

Table 163: SpeakerEQ internal parameters

Table 164, "SpeakerEQ debugging variables" below lists and describes **SpeakerEQ** debugging variables:

Name	Type	Default Value	Description
CurrentAPass	**int32**	**0**	Displays a number ranging from 1 to (*NumTuningPasses* value × 10), indicating the progress of each pass.
CurrentTPass	**int32**	**0**	Displays current pass number.

Table 164: SpeakerEQ debugging variables

7.6.1 Set up and run SpeakerEQ

To set up and run **SpeakerEQ**, follow these steps:

1. In **Studio Project Manager**, on the **Icon View** tab, go to **Telestra Server** ( ).

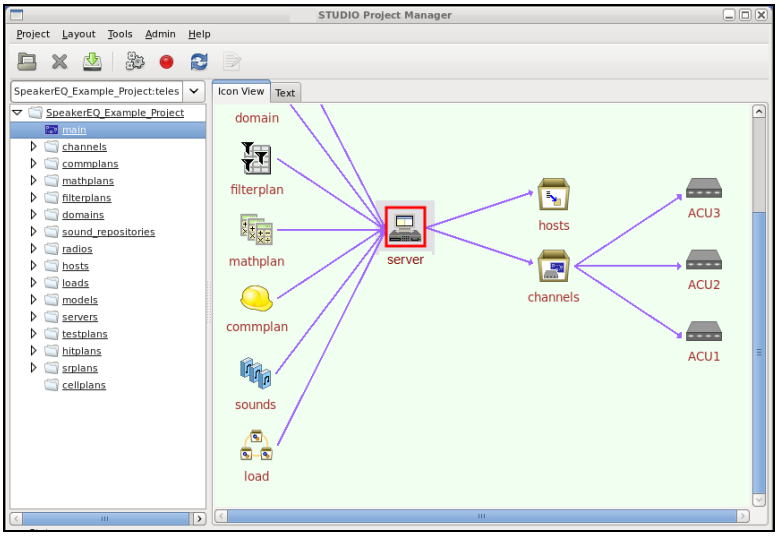


Figure 20: Open Studio Viewer Telestra Server

2. In **Studio Viewer Telestra Server**, under **System Load**, choose a simulation model for **SpeakerEQ** (e.g., **AuralOuts**).

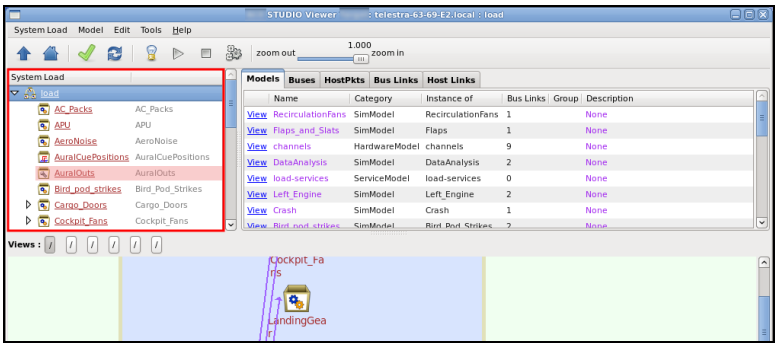


Figure 21: Select simulation model

3. Right-click in **Views**, and select **Add** from the list.
4. In **Add to Model :** *Model Name*, expand **EnvCue**, and select **SpeakerEQ**.
5. *(Optional)* In **Item Name**, enter a name for the **SpeakerEQ** component.

6. Select **Add**, and wait for the component to install. When “Added Successfully” displays, close the window.

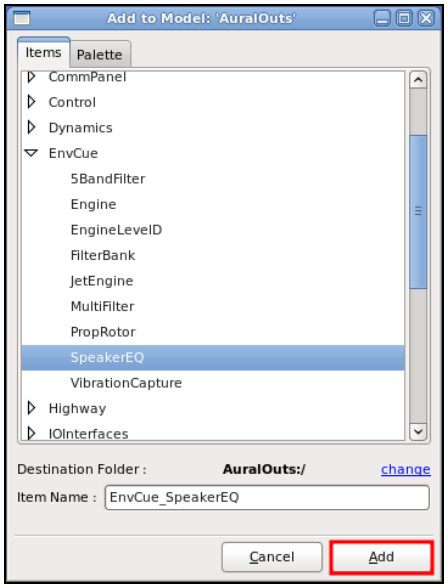


Figure 22: Add SpeakerEQ component to model

7. In **Views**, open the new **SpeakerEQ** component.
8. From **Data Viewer**, next to *SpeakerNumber*, double-click in the **Value** column.
9. In **Set Value**, enter an ID number for the speaker, and select **OK**.

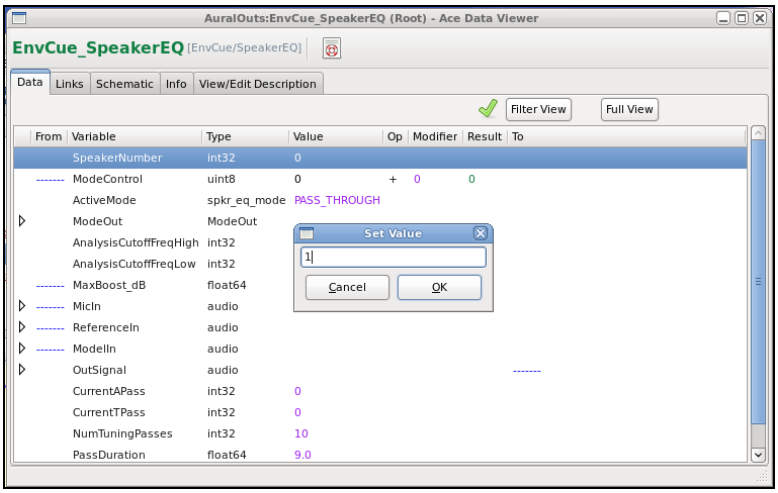


Figure 23: Set SpeakerNumber value

10. Set the *AnalysisCutoffFreqHigh* and *AnalysisCutoffFreqLow* values applicable to your speaker type. Values are measured in Hertz (e.g., enter **10,000** for 10 kHz).

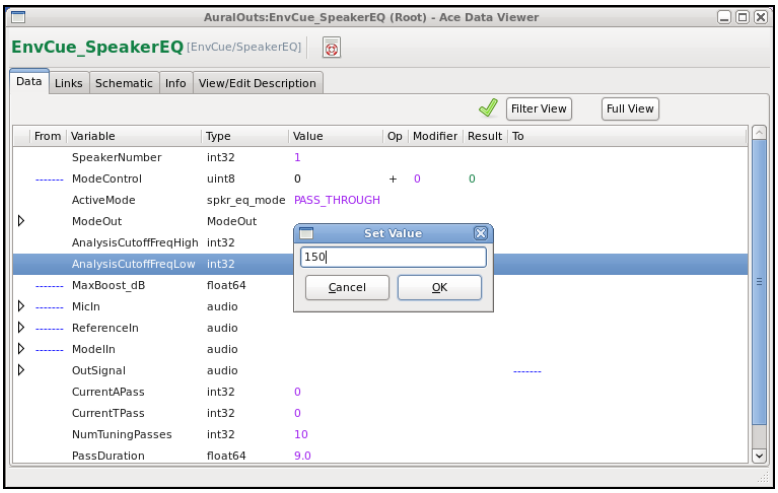


Figure 24: Set AnalysisCutoffFreq values

11. Link *MicIn* to an existing **ACU2channel**. For more information about **ACU2channel**, go to Section 11.4, "ACU2channel" on page 181.

***Caution**: To avoid damaging your speakers, do not run **SpeakerEQ** without connecting a microphone to MicIn.*

12. Link *ReferenceIn* and *ModelIn* to an existing **Playsound** or **NoiseSource** component with pink noise. For a flat frequency response, ensure *ReferenceIn* and *ModelIn* match.

> ***Note**: **Playsound** or **NoiseSource** serve as a substitute during tuning. Once tuning is complete, you may want to replace the pink noise with a component containing audio from your simulation model.*

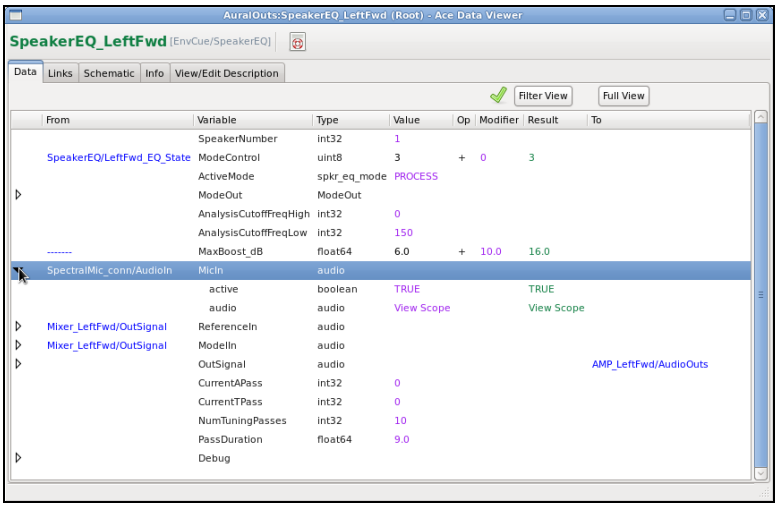


Figure 25: Set MicIn, ReferenceIn, and ModelIn audio inputs

For more information about **Playsound**, go to Section 2.17, "Playsound" on page 42. For more information about **NoiseSource**, go to Section 2.14, "NoiseSource" on page 36.

13. Next to *ModeControl*, double-click in **Modifier**. In **Set Modifier Value**, enter **1**, and select **OK**.

14. Wait for **SpeakerEQ** to cycle through the set number of passes in *CurrentTPass* **Value**. By default, it runs through 10 passes. When finished, *ActiveMode* displays "PROCESS," and *CurrentAPass* and *CurrentTPass* display "0."

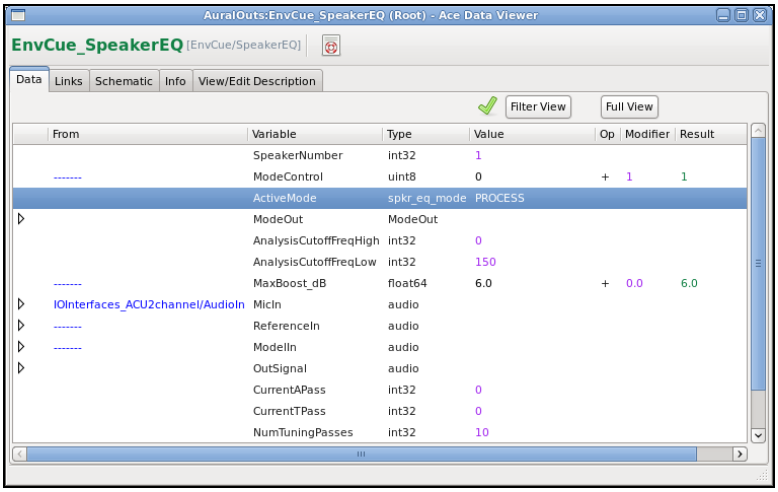


Figure 26: PROCESS mode

7.6.2 Tune SpeakerEQ

After you set and run **SpeakerEQ**, compare the *MicIn* and *ReferenceIn* signals for audio differences. If needed, adjust their frequencies, and rerun the component. To tune **SpeakerEQ**, follow these steps:

1. In **Data Viewer**, expand *MicIn*. Next to **audio**, in the **Value** column, open **View Scope**.

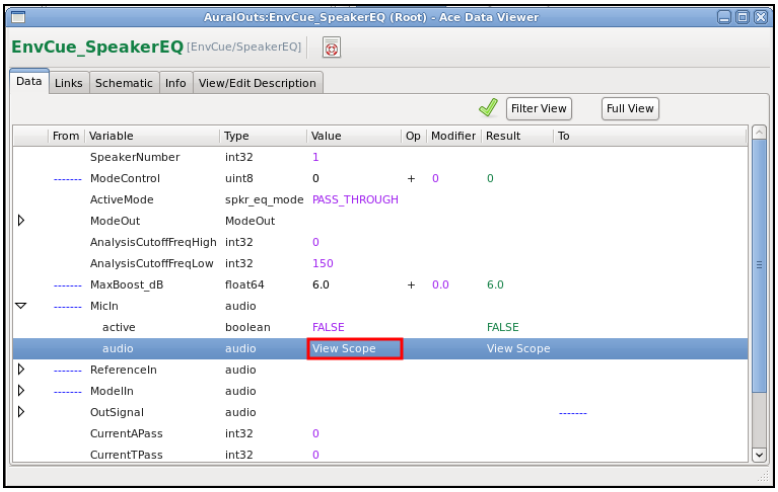


Figure 27: View MicIn scope

The /*Component* **- AnalyzerIn(Result).audio** window opens, displaying the *MicIn* signal.

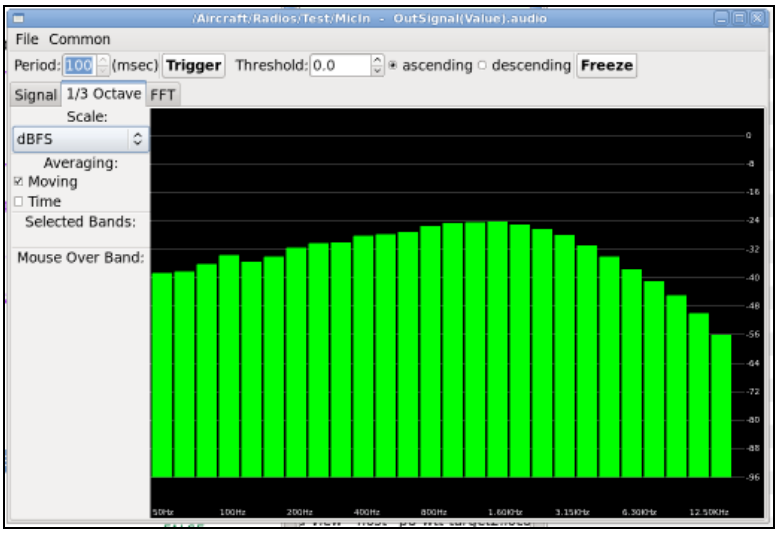


Figure 28: MicIn audio signal

2. In **Data Viewer**, expand *ReferenceIn*. Next to **audio**, in the **Value** column, right-click **View Scope**, point to **Add to Scope**, and select the *ReferenceIn* audio file.

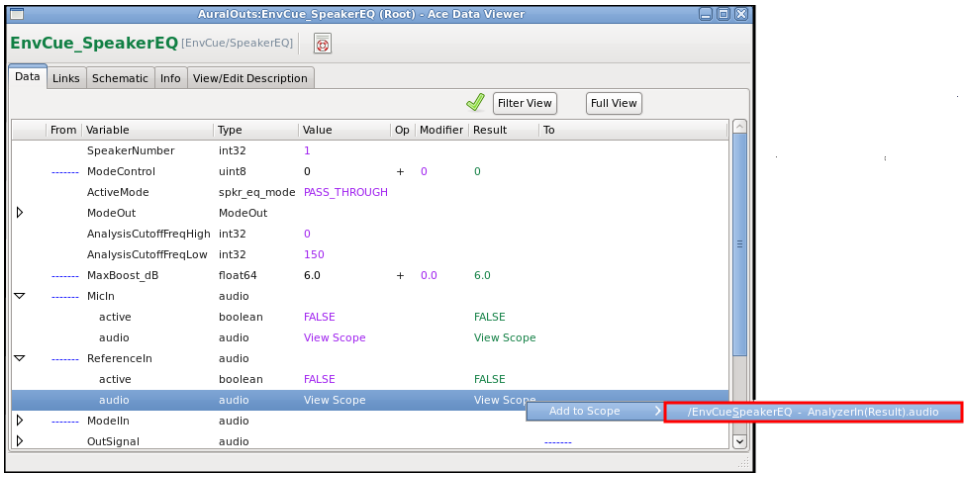


Figure 29: Add ReferenceIn to scope

3. In the **Scope**, on the **File** menu, select **Change Layout**.

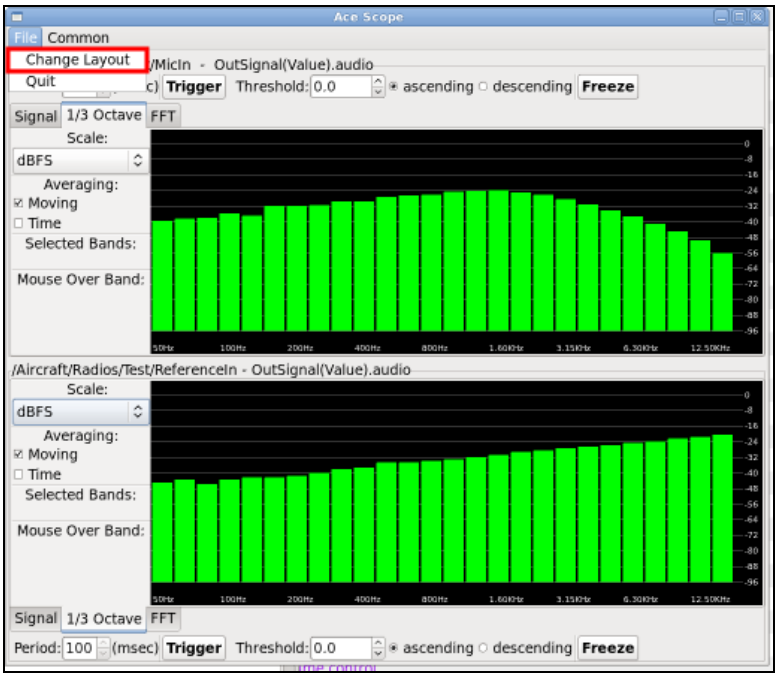


Figure 30: Change Layout

4. In **Change Layout**, select the list, and then select **Comparison**. In **Comparison**, select **OK**.

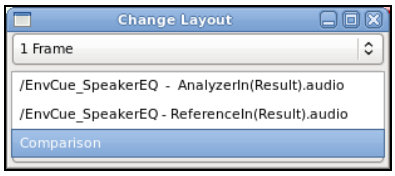


Figure 31: Comparison layout

5. The scope view now displays a comparison of the *MicIn* and *ReferenceIn* signals. In **Y Range**, enter **20**, and press Enter.

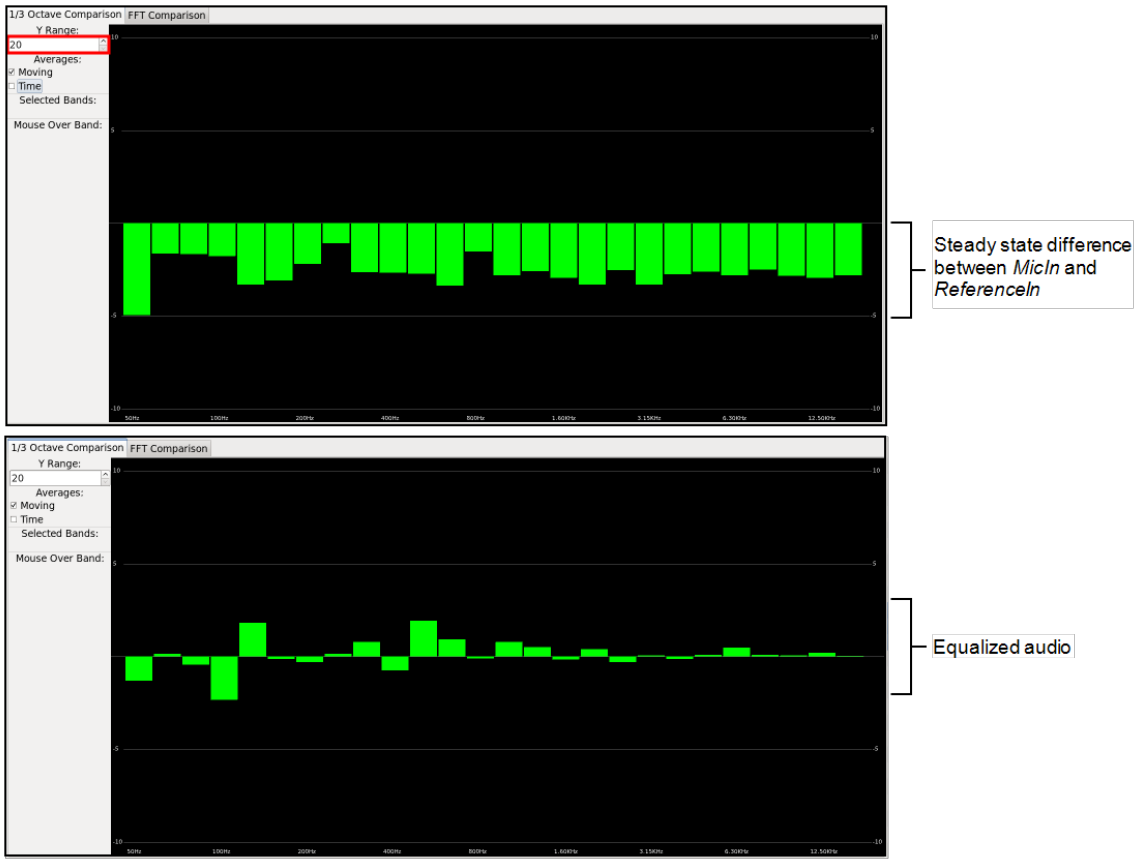


Figure 32: MicIn/ReferenceIn comparison

In the example above, the top comparison shows a steady state difference of 5 decibels (dB), whereas the bottom comparison shows equalized audio.

6. If a difference greater than 2 dB exists, check the recommended high and low cutoff frequencies for your speaker type.

7. Increase *MaxBoost_dB* by the difference amount (e.g., 5 dB), and rerun **SpeakerEQ**.

***Caution**: To avoid damaging the speaker, check your speaker's limitations before adjusting MaxBoost_dB.*

8. If a difference still exists, lower *ReferenceIn*'s gain, and rerun **SpeakerEQ**.

9. If your speaker setup changes after tuning, reprocess **SpeakerEQ** to ensure the speakers are still equalized correctly. To put the component in PROCESS mode, set *ModeControl* to **3**, and save your changes.

7.7 VibrationCapture

Summary: The **VibrationCapture** component records an input signal specifically for use with ASTi's Vibration Analysis tool kit.

Description: This component is a simplified version of **RecordReplay**. The capture length defaults to 30 seconds, and continuous recording is not allowed. The *Record* parameter controls the recording, when true recording begins.

The captured file results in a Telestra sound recording (.tsr) file with 32-bit float values recorded at 48 kHz with a name based on the *GroupID* and *SoundIndex*. The file is written to the hard drive under **/var/local/asti/recordreplay**. **VibrationCapture** always starts recording at the beginning of the file, and it cannot start in the middle. The recording always overwrites the file from the beginning.

Table 165, "VibrationCapture audio input" below lists and describes the **VibrationCapture** audio input variable:

Name	Type	Default Value	Description
SignalIn	**audio**	N/A	Audio linked into this position is recorded and written to the record file on the hard drive.

Table 165: VibrationCapture audio input

Table 166, "VibrationCapture control inputs" below lists and describes **VibrationCapture** control input variables:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	Amplitude gain control for the capture file, which can be set via the host or internally.
SoundIndex	**uint16**	**0**	Identifies the file. Each file is organized as part of a group with a specific index that names the Tracker Status Report file. The file is written to the hard drive under **/var/local/asti/recordreplay**.

Table 166: VibrationCapture control inputs

Table 167, "VibrationCapture control outputs" below lists and describes **VibrationCapture** control output variables:

Name	Type	Default Value	Description
FileMode	**player_state**	**STOPPED**	Displays the recording state of the file. Possible states are STOPPED and RECORDING.
Record	**Boolean**	**FALSE**	When **TRUE**, the component starts recording.

Table 167: VibrationCapture control outputs

Table 168, "VibrationCapture internal parameter" below lists and describes the **VibrationCapture** internal parameter variable:

Name	Type	Default Value	Description
Length	**uint32**	**30**	Set to the number of seconds to record vibration at a 48 kHz sample rate. A value of 30 is the expected length for the Vibration Analysis.

Table 168: VibrationCapture internal parameter

7.8 FilterBank

Summary: FilterBank is a simple method of installing a series bank of 15 configurable filters in a signal path.

Description: FilterBank points to a filter stored in a **FilterPlan** available for viewing and adjustment from the **Project Manager**. Each filter is comprised of 15 filters in series. Each of these 15 filters is fully configurable from the **FilterPlan**.

Table 169, "FilterBank audio input and output" below lists and describes **FilterBank** audio input and output variables:

Name	Type	Default Value	Description
Audio Inputs			
InSignal	**audio**	N/A	Audio input that **FilterBank** filters.
Audio Outputs			
OutSignal	**audio**	N/A	Filtered audio output from **FilterBank**.

Table 169: FilterBank audio input and output

Table 170, "FilterBank control inputs" below lists and describes **FilterBank** control inputs:

Name	Type	Default Value	Description
FilterName	**filter_name**	N/A	Pointer to a bank of 15 filters in the **FilterPlan**.
MasterGain	**float32**	**1.0**	Gain applied to **FilterBank**'s audio output.

Table 170: FilterBank control inputs

7.9 FilterPlan

Summary: FilterPlan is a storage and management location for a set of filters that the **FilterBank** component may reference.

Description: In **FilterPlan**, store filters by name. Each filter contains 15 filter sections, which you can designate as one of the following types:

- *LowPass*
- *HighPass*
- *BandPass*
- *BandPassUnityGain*
- *Notch*
- *AllPass*
- *PeakingEQ*
- *LowShelf*
- *HighShelf*

In each filter section, store the following values:

- *CenterFrequency*
- *QFactor*
- *Gain (dB)*

You can also set and store the set's *MasterGain*. To implement **FilterPlan** edits when applying changes, select **Notify Telestra Server**. As a result, you won't need to save and reload the layout after every change.

8.0 Highway Service

Summary: **Highway Service** provides audio distribution for aural cues from components in Telestra to the outside world. The need for a specialized service to handle aural cue audio is driven by the fact that aural cues are generated by multiple sources whose output is typically sent to multiple channels (e.g., left speaker, right speaker). **Highway Service** handles both the mixing of audio from multiple sources and the routing of the composite audio to multiple channels.

Highway Service supports 128 independent output channels. The service receives audio from a set of **Balancers**. Each **Balancer** sends a stream of audio to the service with applied gains for the audio on each of the 128 highways. The service mixes the audio sources for a given highway to create a composite audio stream for each highway. The service applies highway gains that it receives from **HighwayGain**. This set of gains includes an overall output gain that affects all highways and individual highway gains that affect all audio on a given highway. The service sends the composite audio to **HighwayOut**, which is tuned to one and only one highway. **Highway Service** supports input connections from only **Balancer** and **HighwayGain** and output connections to only **HighwayOut**.

As with all service components, you cannot manually instigate the creation of the service object. Instead, the loader automatically creates service objects on demand when it detects that a component has a **Highway Service** port connection. Only one **Highway Service** is loaded in a model based on the first found need for a service of this type.

Description: **Highway Service** receives an audio input and an array of 128 gain values, one for each highway, and from **Balancer**. The **Highway Service** can connect to up to 1,024 **Balancers**. The connection from **Balancer** to **Highway Service** is unidirectional for audio data. The **Balancer** is always the audio source while the highway service is always the audio sink.

For each **Balancer** connection, the highway service makes 128 copies of the audio and applies a different gain value to each copy. The gain values come from the gain value array that **Balancer** provides. The output from this step is a set of 128 audio streams, one for each highway, per **Balancer**.

Highway Service mixes the audio from the various **Balancers** for a given highway into a final audio output for that highway.

Highway Service also receives an input connection from **HighwayGain**, which provides an overall output gain and 128 individual highway gains. The highway service applies the gains it receives from **HighwayGain** to the composite audio on each highway. Each model may contain only one **HighwayGain**. The model loader throws an error you attempt to add a second **HighwayGain**. If there is a **HighwayGain** in the model, the highway service uses a value of 1.0 for these gains.

The output connection from **Highway Service** goes to a **HighwayOut**. This output connection is unidirectional for audio data. **Highway Service** is always the audio source while **HighwayOut** is always the audio sink. Multiple **HighwayOuts** may point to the same highway channel.

The service must also support the flow through *Active_TX* to indicate the state of audio activity on a highway. Only set *Active_TX* if audio is present and a non-zero gain value at a given point in the path.

Highway Service also supports diagnostic facilities to allow you to view the use of a highway within the model and the current highway gain settings. This functionality scans the model for all **HighwayOuts** and **HighwayGains**, and extracts the information into a table-like view, as demonstrated in the following example:

The handle view displays all **HighwayOuts**, including paths connected to each handle and gains associated with each highway.

The following text shows an example of displayed information:

```
Overall Output Gain = 1.0

Left_Speaker (Gain = 0.5) --->

Aural_Cues / Left_Speaker_Sounds > HighwayOutput_A

Aural_Cues / Left_Speaker_Sounds > HighwayOutput_B

Seat_Shaker (Gain = 0.8) --->

Some_Folder1 / Some_Folder2 / Vibration_Output > HighwayOutput_C
```

8.1 AuralCue

Summary: Inputs audio into the **Highway Service** and associates the audio with a name or "cue ID."

Description: **AuralCue** places an audio signal on a bus in the speaker service. From here, **AuralCuePosn** references that particular bus and gives it a position in the 3D audio environment. At this point, **SpeakerOutput** retrieves the audio at the correct amplitude and phases to position the sound correctly in the 3D environment.

Table 171, "AuralCue audio input" below lists and describes the **AuralCue** audio input variable:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	The audio signal routed to a specific position.

Table 171: AuralCue audio input

Table 172, "AuralCue control input" below lists and describes the **AuralCue** control input:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	A gain applied to the input audio streams.

Table 172: AuralCue control input

Table 173, "AuralCue internal parameter" below lists and describes the **AuralCue** internal parameter:

Name	Type	Default Value	Description
Cue	**ID**	**UNASSIGNED**	A name given to this audio source. For example, add and select a cue called *Engine* and route all engine audio into **AuralCue**. ***Note**: Generate meaningful names for better readability in the model. Reference this cue name in a corresponding component.*

Table 173: AuralCue internal parameter

8.2 AuralCuePosn

Summary: Adds the selected sound to the sound field at the specified (X, Y, Z) position.

Description: In an aircraft model, you might have an **AuralCuePosn** with a *CueID* called “LeftEngine” positioned at X= -3.0, Y= 3.0, Z= 0.0. This means that the audio fed into *CueID* is positioned to the back-left of the reference point. Once a cue is added to the sound field with this component, add either a **Highway** > **SpeakerOutput** or an **AudioIO** > **SpeakerOutput** to get audio from the sound field.

Table 174, "AuralCuePosn Position control inputs" below lists and describes **AuralCuePosn** *Position* control input variables:

Name	Type	Default Value	Description
X	**float64**	**0.0**	X coordinate relative to the reference point. Positive *X* is in front and negative *X* is behind reference point.
Y	**float64**	**0.0**	Y coordinate relative to the reference point. Positive *Y* is to left, and negative *Y* is to right of reference point.
Z	**float64**	**0.0**	Z coordinate relative to the reference point. Positive *Z* is above, and negative *Z* is below the reference point.

Table 174: AuralCuePosn Position control inputs

Table 175, "AuralCuePosn internal parameter" below lists and describes the **AuralCuePosn** internal parameter variable:

Name	Type	Default Value	Description
CueID	**id**	**UNASSIGNED**	Selects *CueID* specified in **AuralCue** for positioning.

Table 175: AuralCuePosn internal parameter

8.3 SpeakerOutput

Summary: Retrieves audio from the **Highways** sound field intended for a speaker at the specified X, Y, Z position.

Description: **SpeakerOutput** creates a speaker in **Highway Service** with the specified X, Y, and Z coordinates relative to the sound field reference point, extracting the audio from the service for the speaker. The X, Y, and Z position should correspond to the speaker's physical location. Typically, *AudioOut* links directly to an **AmpOut** audio out.

For example, for an eight-channel amplifier to drive an eight-speaker setup, the model should contain eight **SpeakerOutputs** linked to eight **AmpOuts**.

Table 176, "SpeakerOutput audio output" below lists and describes **SpeakerOutput** audio output and internal parameter variables:

Name	Type	Default Value	Description
AudioOut	**audio**	N/A	Audio for this speaker.

Table 176: SpeakerOutput audio output

Table 177, "SpeakerOutput internal parameters" below lists and describes **SpeakerOutput** internal parameters:

Name	Type	Default Value	Description
Position	**worldposition_geocentric**	**(0,0,0)**	The X, Y, and Z position of the speaker relative to the reference point. The coordinate system is the same as the one used in **Highway** > **AuralCuePosn** (i.e., +X is forward, +Y is to the left, +Z is up).
SourceSpeaker	**id**	**UNASSIGNED**	Selects the speaker for sound positioning.

Table 177: SpeakerOutput internal parameters

9.0 Highway 3D Service

The **Highway 3D Service** is a collection of components used for mixing and routing audio to hardware output channels. The service provides two approaches for audio mixing:

- Gain-Mixing
- 3D Soundfield Reconstruction

These approaches are useful for a range of applications, from routing communications audio to operator headsets to immersing listeners in a 3D audio environment.

Gain-mixing is the simpler approach and represents the more traditional way of mixing audio. Sounds are sent to a set of output “highways” using a set of gains that are driven by the host or calculated in the model. Gain-mixing using the **Highway 3D Service** allows you to easily route audio from simulation models to hardware models, making it an attractive alternative to using connectors and buses.

The 3D Soundfield Reconstruction (SFRC) is a sophisticated method of processing sounds to make them appear to come from somewhere in 3D space around the listener(s). Sounds are assigned (X, Y, Z) positions relative to the listening or reference point. The **Highway 3D Service** then filters and mixes each sound based on its position in order to encode the sound into a virtual 3D sound environment or sound field.

Once all the sounds are encoded, the sound field is decoded for playback in stereo headphones or an array of speakers. If decoding to headphones, two output streams are output for the left and right earphones. The head's orientation is used in this calculation if it is available from a head-tracking system. If decoding to a speaker array, the service outputs one audio stream for each speaker, taking into account the speaker's (X, Y, Z) position.

All positions used in the service must be specified in meters relative to the listening or reference position. The reference position should be the central listening point of the simulator, such as the midpoint between the pilot's and copilot's heads. This approach ensures the service renders the sounds in a way that sounds best to all listeners.

Use the following coordinate system when calculating **RelativePositions** of sounds and speakers:

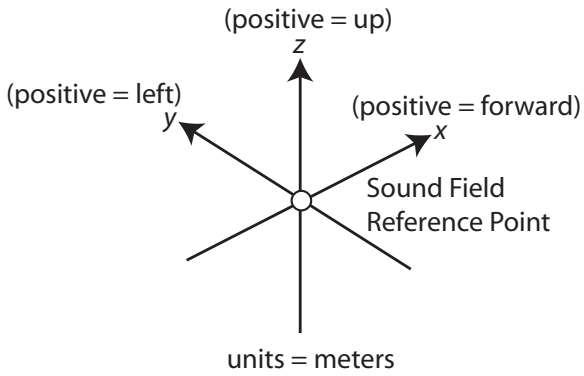


Figure 33: RelativePositions in sound coordinate system

The 3D positioning algorithm used by the service calculates a contribution from each speaker for each sound, even if the speaker is not precisely in the right direction. This results in realistic and immersive reproduction of many types of sounds.

If a sound outputs only through one speaker, use gain-mixing or connectors and buses instead of SFRC.

Highway 3D Service requires a three-step process that includes entering audio into the service, determining how it is mixed or processed, and extracting audio for the outputs from the service.

The **Highway 3D Service** components include the following:

For input into the service: **Audio Feed**

- For controlling gain-mixing or 3D positioning of a sound:
  - **Feeders** > **Balancer1**, **4**, **8**, **16**
  - **Feeders** > **AuralCuePosn**
- For extracting audio from the service:
  - **AudioIO** > **HighwayOut**
  - **AudioIO** > **SpeakerOut**
  - **AudioIO** > **Headphone3DOut**

To set **Highway 3D Service**, follow these steps:

1. Add **AudioFeed** in a simulation model to input a sound into the service. Add more **AudioFeeds** for other sounds if they need mixing or independent positioning. Create and select new sound names or “cue IDs” to identify the sound in the service.

2. Add **Balancer***N* in a **Feeders** model to gain-mix a sound to the output highways. Select the cue ID of the sound used for mixing, and choose or create the highways. Mix the sound's output highways to determine the **Balancer's** size. Link the gains from the host or elsewhere in the load if desired, or just set static gains by hand.

3. Add **AuralCuePosn** to a **Feeders** model to add a sound to the 3D sound field at a specified position. Select the cue ID of the sound that is then added to the sound field. Link to or set the (X, Y, Z) position inputs to determine the location of the sound in the sound field.

4. Add **HighwayOut** in a hardware model to extract audio from one of the output highways. Choose a highway and link the audio output of the component to a hardware output. This component only retrieves sounds added to the highways via **Balancer**, not **AuralCuePosn**.

5. Only sounds put onto the highways using **Balancer** components are retrieved by this component, not sounds that were positioned in the sound field using the **AuralCuePosn** component.

6. Add **SpeakerOuts** in a hardware model for each speaker in the simulator. Enter the (X, Y, Z) position of the speaker, and link the audio output to a hardware output. The service decodes the sound field to this speaker based on the specified position.

7. Add **Headphone3DOut** in a hardware model for operators that should have 3D audio in their stereo headphones. If head tracker information is available from the host, link it to the head azimuth and elevation inputs. Link the component's left and right audio outputs directly to the output channels for the left and right ear cups of the operator's headphones. Avoid adding other components between the **Headphone3DOut** and hardware components. The presence of other components on these audio streams can alter the timing and distort the apparent 3D position of the sound.

HighwayOut and **SpeakerOut** can be added to autogenerated hardware models using **Channel Helper**. Notice the *Highway*, *Speaker*, and *Speaker Position* options when creating ACE-RIU, ACU2, and amplifier channels in **Channel Helper**. Enter a highway name to create a **HighwayOut**, select the highway, and link the audio to the channel's audio out. Enter a speaker name and position to create a **SpeakerOut** created at that position with its audio linked to the channel's audio out.

Figure 34, "Gain-mixing" below shows how to use the **Highway 3D Service** for gain-mixing warning tones to a set of output highways:

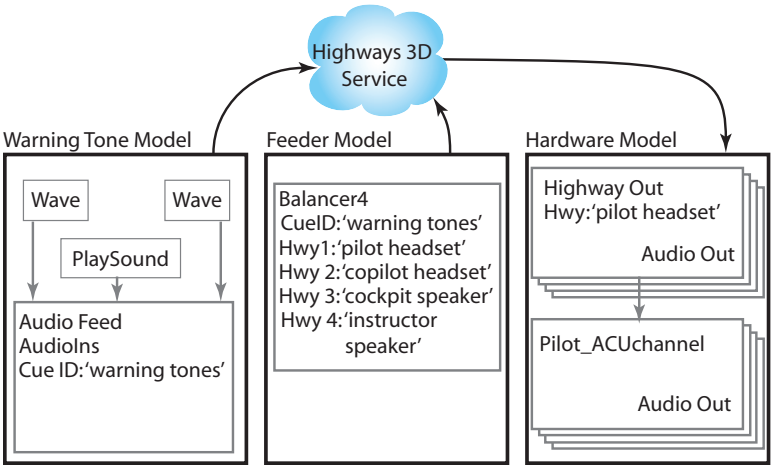


Figure 34: Gain-mixing

Figure 35, "Sound field reconstruction" below shows an example of using the **Highway 3D Service** for position weapons sounds in 3D and rendering the sound field to both a speaker array and a stereo headset:

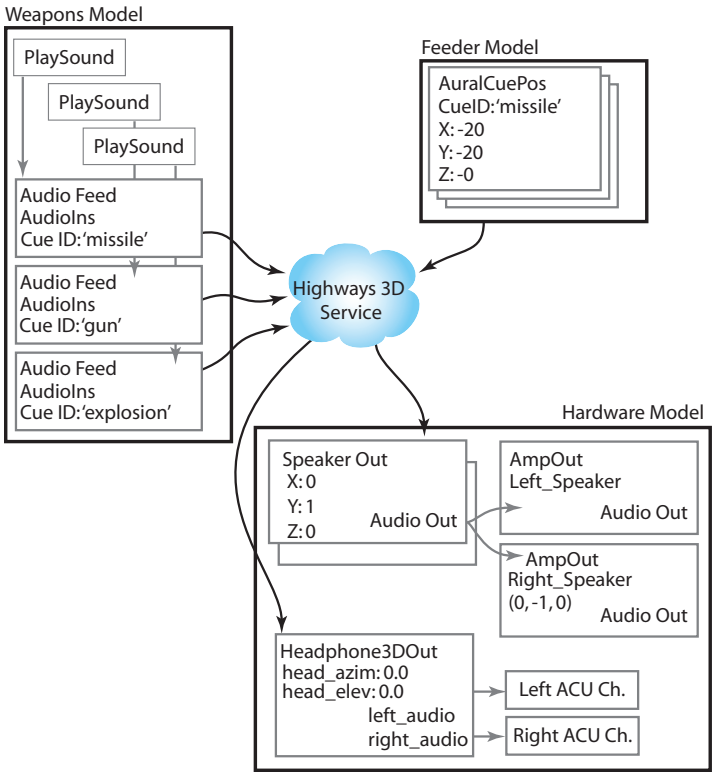


Figure 35: Sound field reconstruction

For more information, go to "Highway 3D Service" in the *Studio Technical User Guide*.

9.1 Audio > Audio Feed

Summary: Inputs audio into the **Highway 3D Service** and associates the audio with a name or “cue ID.”

Description: **AudioFeed** is the first step when using the **Highway 3D Service**. First audio is linked to **AudioFeed**, and the cue ID is created and selected. Then this audio is mixed in a 3D or non-3D way using the other **Feeders**.

Table 178, "AudioFeed audio inputs" below lists and describes **AudioFeed** audio input variables:

Name	Type	Default Value	Description
InSignals	**audio[64]**	N/A	An array of audio inputs. Up to 64 links can connect to this input. All of the incoming audio is mixed in this component.
InSignals	**audio**	N/A	Provides a view of all the mixed audio inputs.

Table 178: AudioFeed audio inputs

Table 179, "AudioFeed control input" below lists and describes the **AudioFeed** control input variable:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	A gain applied to the input audio streams.

Table 179: AudioFeed control input

Table 180, "AudioFeed internal parameter" below lists and describes the **AudioFeed** internal parameter variable:

Name	Type	Default Value	Description
Cue	**ID**	**UNASSIGNED**	A name given to this audio source. For example, add and select a *Cue* called **Engine**, and route all engine audio into this **AudioFeed**. ***Note**: Generate meaningful names for better readability in the model. Reference this Cue name in a corresponding **Balancer** or **AuralCuePosn**.*

Table 180: AudioFeed internal parameter

9.2 Feeders > AuralCuePosn

Summary: Adds the selected sound cue bus to the sound field at the specified (X, Y, Z) position.

Description: In an aircraft model, you might have an **AuralCuePosn** with a *CueID* called "Left Engine" positioned at X= -3.0, Y= 3.0, Z= 0.0. This means that the audio fed into this *CueID* with **AudioFeed** is positioned to the back-left of the reference point.

Once a cue is added to the sound field with this component, add either an **AudioIO** > **Headphone3DOut** or **AudioIO** > **SpeakerOut** to get audio from the sound field.

Table 181, "Feeders > AuralCuePosn control inputs" below lists and describes **Feeders > AuralCuePosn** control input variables:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	Linear gain attached to the *Cue*.
X	**float32**	**0.0**	The *X* coordinate relative to the reference point. Positive *X* is in front and negative *X* is in back of the reference point.
Y	**float32**	**0.0**	The *Y* coordinate relative to the reference point. Positive *Y* is left, and negative *Y* is right of the reference point.
Z	**float32**	**0.0**	The *Z* coordinate relative to the reference point. Positive *Z* is above and negative *Z* is below of the reference point.
W_Adjust	**float32**	**1.0**	Focus of the located sound.

Table 181: Feeders > AuralCuePosn control inputs

Table 182, "Feeders > AuralCuePosn internal parameters" below lists and describes **Feeders > AuralCuePosn** internal parameters:

Name	Type	Default Value	Description
Cue	**ID**	**UNASSIGNED**	Selects the sound cue bus for positioning.

Table 182: Feeders > AuralCuePosn internal parameters

9.3 Feeders > Balancer1, 4, 8, 16

Summary: Mixes the selected sound cue bus to the selected highways with a set of gains. **Balancer1** mixes to one **Highway**, **Balancer4** to four **Highways**, etc.

Description: After inputting the audio into the **Highway 3D Service** using **AudioFeed**, **Balancers** mix the audio to output highways. **Balancers** provide simple gain mixing to the highways. No 3D calculations are involved. If positioning in 3D is desired, use **Feeders** > **AuralCuePosn**. For example, use **Balancer4** to mix a tone to four different highways where the audio from each highway is routed to an operator's headset.

Table 183, "Balancer control inputs" below lists and describes **Balancer** control input variables:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	The gain applied to the cue's audio before it is mixed to the highways.
Gain1–Gain8	**float32**	**1.0**	An additional gain applied when mixing the cue's audio to each highway. *Gain1* affects mixing on the first selected highway, *Gain2* affects mixing on the second selected highway, etc.

Table 183: Balancer control inputs

Table 184, "Feeders > Balancer1, 4, 8, 16 internal parameters" below lists and describes **Feeders** > **Balancer1**, **4**, **8**, **16** internal parameter variables:

Name	Type	Default Value	Description
Cue	**id**	**UNASSIGNED**	Selects the sound for mixing.
Highways	**id[8]**	**UNASSIGNED**	Selects which highways will receive audio. The number of highways varies based on the **Balancer** used (i.e., 1, 4, 8, or 16).

Table 184: Feeders > Balancer1, 4, 8, 16 internal parameters

9.4 AudioIO > Headphone3DOut

Summary: **Headphone3DOut** retrieves the sound field from the service and filters it to reproduce the 3D environment in a pair of audio outputs for a stereo headset.

Description: **Headphone3DOut** generates 3D audio in the headphones of an operator. The operator's head is located at the sound field reference point (0, 0, 0). The component drives head azimuth and head elevation from the host based on values from a head tracking system. These values define the head's orientation relative to straight forward and cause the sound field to rotate accordingly.

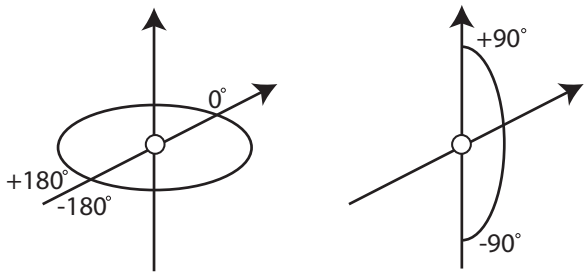


Table 185, "Headphone3DOut control inputs" below lists and describes **Headphone3DOut** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**TRUE**	Enables and disables 3D filtering. If disabled, the sound field is mixed in mono to the left and right headphone channels.
HeadAzim	**float32**	**0.0**	The orientation of the head in the horizontal plane. 90 degrees is directly to the left, and -90 degrees is directly to the right.
HeadElev	**float32**	**0.0**	The orientation of the head in the vertical plane, 90 degrees is directly straight up and -90 degrees is directly straight down.

Table 185: Headphone3DOut control inputs

Table 186, "Headphone3DOut audio outputs" below lists and describes **Headphone3DOut** audio output variables:

Name	Type	Default Value	Description
LeftAudio	**audio**	N/A	The audio intended for the left ear cup of the headphones.
RightAudio	**audio**	N/A	The audio intended for the right ear cup of the headphones.

Table 186: Headphone3DOut audio outputs

9.5 AudioIO > HighwayOut

Summary: Extracts the specified highway's audio from the **Highway 3D Service**. *AudioOut* typically links directly to a hardware component's (ACE-RIU, ACU2, etc.) audio out.

Description: To create new highways, double-click **SourceHighway**, and add new buses in the service window. For example, add a highway for each speaker or headset. Name the highways based on the corresponding hardware that it is linked to (e.g., "PilotHeadset"). To put audio onto the highways, use **AudioFeed** and **Balancer**. **Balancer** takes audio from **AudioFeed** and puts it onto one or more highways.

Table 187, "HighwayOut audio output" below lists and describes the **HighwayOut** audio output:

Name	Type	Default Value	Description
AudioOut	**audio**	N/A	The highway's audio.

Table 187: HighwayOut audio output

Table 188, "HighwayOut control input" below lists and describes the **HighwayOut** control input:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	A gain applied to the audio output.

Table 188: HighwayOut control input

Table 189, "HighwayOut internal parameter" below lists and describes the **HighwayOut** internal parameter:

Name	Type	Default Value	Description
SourceHighway	**ID**	**UNASSIGNED**	Specifies which highway to extract audio from.

Table 189: HighwayOut internal parameter

9.6 AudioIO > SpeakerOut

Summary: Retrieves audio from the Highways 3D sound field intended for a speaker at the specified X, Y, Z position.

Description: **SpeakerOut** creates a speaker in **Highway 3D Service** with the specified X, Y, and Z coordinates relative to the soundfield reference point, extracting the audio from the service for the speaker. The X, Y, and Z position should correspond to the speaker's physical location. Typically, **SpeakerOut** links directly to an **AmpOut** audio out.

For example, if an eight-channel amplifier drives an eight-speaker setup, the model should contain eight **SpeakerOuts** linked to eight **AmpOuts**.

Table 190, "SpeakerOut audio output" below lists and describes **SpeakerOut** audio output variable:

Name	Type	Default Value	Description
AudioOut	**audio**	N/A	Audio for this speaker.

Table 190: SpeakerOut audio output

Table 191, "SpeakerOut control inputs" below lists and describes **SpeakerOut** control input variables:

Name	Type	Default Value	Description
Gain	**float32**	**1.0**	A gain applied to the audio output.
Position	**worldposition_geocentric**	**(0.0, 0.0, 0.0)**	The X, Y, and Z position of the speaker relative to the reference point. The coordinate system is the same as the one used in **AuralCuePosn**, where X is forward, +Y is left, and +Z is up.

Table 191: SpeakerOut control inputs

10.0 HRTFService

Description: The **HRTFService** provides 3D capability for headphone audio. Both environmental and communications audio can be mixed in 3D. Head-related transfer functions (HRTFs) position audio streams at specified 3D (i.e., azimuth and elevation) positions.

Figure 36, "HRTFService 3D communications configuration" on the next page depicts a 3D communications configuration for a single operator. The operator has two **CommPanels**: one for radios and the other for intercoms. Since both **CommPanel** components have the same HRTF bus selected (i.e., Op1), they contribute audio to the same four audio positions. Behind the scenes, **HRTFService** mixes audio from **CommPanels** with the same HRTF bus. Then **HRTFOut4** applies 3D filters according to the specified azimuth and elevation positions. **HRTFOut4's** left and right audio outputs connect to an ACENet channel to go to the operator's stereo headset. Each additional 3D operator may have a nearly identical configuration but must use a unique HRTF bus for independent audio and 3D positioning.

Table 192, "HRTFService internal parameter" below lists and describes the **HRTFService** internal parameter variables:

Name	Type	Default Value	Description
Position	**worldposition_geocentric**	**(0,0,0)**	The X, Y, and Z position of the speaker relative to the reference point. The coordinate system is the same as the one used in **AuralCuePosn** (i.e., +X is forward, +Y is to the left, +Z is up).

Table 192: HRTFService internal parameter

Figure 36, "HRTFService 3D communications configuration" below shows a **HRTFService** 3D communications configuration:

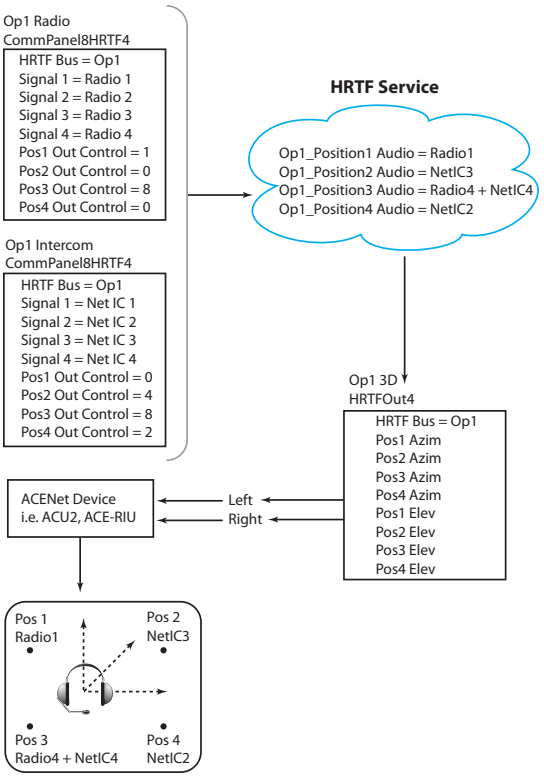


Figure 36: HRTFService 3D communications configuration

10.1 HRTFOut4

Summary: Outputs a 3D mix of audio for up to four sound positions.

Description: This component receives up to four audio streams from **HRTFService**, such as **CommPanel8HRTF4**. Each stream is positioned at a specific azimuth and elevation in 3D space relative to the listener. Valid azimuth values range is from -180 to 180 degrees. Valid elevation range is from -40 to 90 degrees. Head-related transfer function (HRTF) measurements are available in increments of a few degrees. The azimuth and elevation inputs jumps to the nearest position for which a filter is available. The resulting azimuth and elevation is then shown in **Result** for each variable.

An azimuth and elevation position of (0,0) corresponds to directly in front of the listener. Positive azimuths position sounds to the right and negative azimuths position sounds to the left. An azimuth of 90 corresponds to a position in line with the left ear. An elevation of 90 degrees means straight up.

The HRTF uses time (i.e., delay) and intensity (i.e., gain) differences to synthesize how a sound or channel appears to come from a particular point in space. **HRTFOut4** *Left* and *Right* audio outputs must directly connect to the ACENet channel(s), since that is the final output stage before the digital-to-analog (D/A) conversion. For example, if the HRTF is using an ACU2, **HRTFOut4** should directly link to **ACU2channel** rather than going through a series of mixers or other components. This ensures that the HRTF timing and gains are maintained. Other audio sources (i.e., sidetone, etc.) can be mixed into **ACENet** as needed, given **ACENet** performs implicit mixing at the final output stage.

Table 193, "HRTFOut4 audio inputs" below lists and describes **HRTFOut4** audio input variables:

Name	Type	Default Value	Description
OutLeft	**audio**	N/A	3D audio output for the left ear. ***Caution**: OutLeft must be directly connected to **ACENet** (i.e., ACU2, ACE-RIU, etc.).*
OutRight	**audio**	N/A	3D audio output for the right ear. ***Caution**: OutRight must be directly connected to an ACENet component (e.g., ACU2, ACE-RIU.).*

Table 193: HRTFOut4 audio inputs

Table 194, "HRTFOut4 control inputs" below lists and describes **HRTFOut4** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**TRUE**	Turns 3D filtering on for all four audio positions. If **FALSE**, the audio outputs are a mono mix of the four positions.
Pos1Azim–Pos4Azim	**int32**	**0**	Azimuth in degrees of audio position *N* (i.e., 1–4).
Pos1Elev– Pos4Elev	**int32**	**0**	Elevation in degrees of audio position *N* (i.e., 1-4).
Pos1Gain– Pos4Gain	**float32**	**1.0**	Gain applied to position *N* (i.e., 1-4) audio before mixing.
OutGain	**float32**	**1.0**	Gain applied to the left and right audio outputs, affecting all positions.

Table 194: HRTFOut4 control inputs

Table 195, "HRTFOut4 status variable" below lists and describes the **HRTFOut4** status variable:

Name	Type	Default Value	Description
Pos1Audio– Pos4Audio	**audio**	N/A	Displays each position's audio stream prior to 3D mixing.

Table 195: HRTFOut4 status variable

Table 196, "HRTFOut4 internal parameter" below lists and describes the **HRTFOut4** internal parameter variable:

Name	Type	Default Value	Description
HRTFBus	**id**	**UNASSIGNED**	Selects **HRTFService** bus of this component. Components sending audio to this component (e.g.,**CommPanel8HRTF4**) must have a matching bus.

Table 196: HRTFOut4 internal parameter

10.2 CommPanel8HRTF4

Summary: Operates the same as the **CommPanel8** but outputs audio with the **HRTFService** for 3D positioned communications.

Description: **CommPanel8HRTF4**, much like a standard **CommPanel**, transmits and receives on multiple intercoms or radios. The component's *InSignal* and *SideSignal* behavior is identical to those of the generic **CommPanel**. However, instead of one *OutSignal* link, four output signals are generated and sent to the **HRTFService** for 3D mixing. The four signals correspond to four positions in 3D space.

This component should be paired with **HRTFOut4**, which receives the four output signals and does the 3D filtering. To establish this connection, the components must have matching *HRTFBus* variables. The four *PosNOutControl* masks determine which intercom buses contribute receive audio to each 3D position. Multiple **CommPanel8HRTF4s** can share the same *HRTFBus* to contribute audio to the same set of four positions.

Table 197, "CommPanel8HRTF4 audio input" below lists and describes the **CommPanel8HRTF4** audio input variable:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	Input audio that **CommPanel** transmits.

Table 197: CommPanel8HRTF4 audio input

Table 198, "CommPanel8HRTF4 audio output" below lists and describes the **CommPanel8HRTF4** audio output variable:

Name	Type	Default Value	Description
SideSignal	**audio**	N/A	Sidetone audio generated by mixing all received sidetone signals from buses selected in the *SideControl* bit mask.

Table 198: CommPanel8HRTF4 audio output

Table 199, "CommPanel8HRTF4 internal parameters" below lists and describes **CommPanel8HRTF4** internal parameter variables:

Name	Type	Default Value	Description
HRTFBus	**id**	**UNASSIGNED**	Selects **HRTFService** bus. **HRTFOut4** with a matching bus receives this **CommPanel's** four position audio streams.
Sig1–Sig8	**id**	**UNASSIGNED**	Selects intercom bus handle.

Table 199: CommPanel8HRTF4 internal parameters

Table 200, "CommPanel8HRTF4 control inputs" on the next page lists and describes **CommPanel8HRTF4** control input variables:

Name	Type	Default Value	Description
InControl	**byte**	**255**	Bit mask that selects intercom buses to transmit *InSignal*. For example, a value of 1 transmits on *Sig1*, a value of 2 on *Sig2*, 4 on *Sig3*, and 255 on all buses.
InGain	**float32**	**1.0**	Scales input audio.
Pos1OutControl–Pos4OutControl	**byte**	**255**	Bit mask that selects which buses contribute their output signal to position *N* (i.e., 1-4) output.
Power	**Boolean**	**TRUE**	Controls **CommPanel's** power. If **FALSE**, no audio is received or transmitted.
PTT	**Boolean**	**FALSE**	Controls **CommPanel** audio transmission.
SideControl	**byte**	**255**	Bit mask that selects intercom buses that contributes to *SideSignal*. For example, a value of 1 receives from *Sig1*, a value of 2 from *Sig2*, 4 from *Sig3*, and 255 from all buses.

Name	Type	Default Value	Description
SideToneGain	**float32**	**1.0**	Scales sidetone mix just before *SideSignal* output.
SideGainControl	**byte**	**255**	Selects which side tones are affected by receive signal gains. When a bit is high, the intercom bus's sidetone volume is multiplied by the appropriate *SigN_RxGain*. *SideControl* value for intercom bus becomes logical and of *SideControl* and *OutControl*.
SidetoneLocal	**byte**	**255**	Selects which side tones are generated locally versus remotely. If more than one **CommPanel** is sharing the same intercom bus, this variable determines if **CommPanel** shares the sidetone with other panels.
Sig1_RxGain–Sig8_RxGain	**float32**	**1.0**	Receive volume for each intercom bus.

Table 200: CommPanel8HRTF4 control inputs

11.0 IOInterfaces

The following section details the **IOInterfaces** components and the objects within them. The **IOInterfaces** components include:

- **ACE_RIU_channel**
- **ACE_RIU_SerialByteOut**
- **ACUchannel**
- **ACU2channel**
- **ACU2_SerialByteOut**
- **AmpOut**
- **RTPStream**
- **SerialPort_IO**
- **VoisusChannel**

11.1 ACE_RIU_channel

Summary: The ACE-RIU device provides remote digital-to-analog audio and I/O distribution between Telestra servers and audio peripherals. The **ACE_RIU_channel** component assigns audio inputs and outputs for models.

Description: The **ACE_RIU_channel** component connects the software model audio to one specified channel on the ACE-RIU device. It also handles the digital in and out from the device channel. Up to 32 audio streams can be mixed together and linked to the ACE-RIU channel output. The component accepts a single audio in source. The audio gain is adjusted through the audio in and out gains. The overall channel volume is adjusted through the volume input.

Two serial devices can be connected to one ACE-RIU device. The serial devices are associated with ACE-RIU channels A and C. When connecting to a state machine device such as an handheld terminal (HHT) or SINCGARS panel, only use channels A and C.

Table 201, "ACE_RIU_channel audio input" below lists and describes **ACE_RIU_channel** audio input variable:

Name	Type	Default Value	Description
AudioOut	**audio[32]**	N/A	Mixes up to 32 audio streams before it is routed out of the ACE-RIU device as an analog signal.

Table 201: ACE_RIU_channel audio input

Table 202, "ACE_RIU_channel audio output" below and describes the **ACE_RIU_channel** audio output:

Name	Type	Default Value	Description
AudioIn	**audio**	N/A	Audio routed in from the ACE-RIU device and sent to other components in the model.

Table 202: ACE_RIU_channel audio output

Table 203, "ACE_RIU_channel control input" below lists and describes the **ACE_RIU_channel** control input variable:

Name	Type	Default Value	Description
DigitalIn	**uint8**	**0**	Allows the **ACE_RIU_channel** to drive a digital input to the software. Each channel has one digital input.

Table 203: ACE_RIU_channel control input

Table 204, "ACE_RIU_channel control outputs" below lists and describes **ACE_RIU_channel** control output variables:

Name	Type	Default Value	Description
AudioOutGain	**float32**	**1.0**	Provides the audio output's gain control.
AudioInGain	**float32**	**1.0**	Provides the audio input's gain control.
DigitalOut	**Boolean**	**FALSE**	Allows the software to drive a digital output on the ACE-RIU. Each ACE-RIU channel has one digital output.
Volume	**float32**	**1.0**	Sets the ACE-RIU channel's main volume.

Table 204: ACE_RIU_channel control outputs

Table 205, "ACE_RIU_channel internal parameters" on the facing page lists and describes **ACE_RIU_channel** internal parameter variables:

Name	Type	Default Value	Description
Channel	**riu_channel**	**None**	Selects the ACE-RIU channel.
Identifier	**device_id**	**<Select>**	Selects the name of the ACE-RIU device.

Name	Type	Default Value	Description
StateMachine			Defines variables related to a state machine device associated with the specified ACE-RIU channel such as an HHT or SINCGARS panel. State machine devices can only connect to channels A and C. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
Name	**string**	**<Edit>**	Identifies state machine instance in the initialization (i.e., .ini) file. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
EntityName	**string**	**<Edit>**	Identifies the state machine .ini file. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
Type	**sme_type**	**NONE**	Type of state machine device. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*

Table 205: ACE_RIU_channel internal parameters

11.2 ACE_RIU_SerialByteOut

Summary: This component transmits up to eight bytes of data from an ACE-RIU ACENet device using the serial port interface.

Description: This device transmits a range of bytes (i.e., 0–8) from the ACENet device's serial port interface. The bytes transmit when *MsgSize* updates and is not 0. The bytes also transmit when the *Char* numbers update and the updated *Char* number is in the range of bytes transmitted, as specified by *MsgSize*.

Table 206, "ACE_RIU_SerialByteOut control inputs" below lists and describes **ACE_RIU_SerialByteOut** control input variables:

Name	Type	Default Value	Description
MsgSize	**uint32**	**0**	Specifies the number of bytes that the *CharsOut* array transmits. The range for this variable is [0,8]. When this control changes, the corresponding number of bytes transmit in order. • *Modifier*: add (**+**) • *Modifier_default*: **0**
CharsOut	**uint8**	**0**	An array of eight bytes (uint8). The resulting bytes in this array transmit in order from *Char0* to the number specified in *MsgSize*. If *MsgSize* is 0, no bytes are transmitted. If *MsgSize* is not 0 and a byte updates in the range [*Char1*, *Char*{*MsgSize*}], the whole range transmits. • *Modifier*: add (**+**) • *Modifier_default*: **0**

Table 206: ACE_RIU_SerialByteOut control inputs

Table 207, "ACE_RIU_SerialByteOut internal parameters" below lists and describes **ACE_RIU_SerialByteOut** internal parameter variables:

Name	Type	Default Value	Description
Identifier	**device_id**	**<Select>**	The name of the ACE-RIU.
Channel	**riu_channel**	**A**	Select the **ACUchannel** to transmit serial data. Serial Port A requires channel A; Serial Port B requires channel C.
BaudRate	**serial_baud_rate**	**Baudrate_4800**	Select the serial data's baud rate.

Table 207: ACE_RIU_SerialByteOut internal parameters

11.3 ACUchannel

Summary: The software-configurable component for the ACU hardware device.

Description: The **ACUchannel** connects the software model audio to a channel of the physical ACU device. The control outputs provide the press-to-talk (PTT) channel selection for Channels 0–3. Adjust audio gain with the *AudioOuts* and *AudioIn* gains. The volume controls the overall channel's volume.

Table 208, "ACUchannel audio input" below lists and describes **ACUchannel** audio input variable:

Name	Type	Default Value	Description
AudioOuts	**audio[32]**	N/A	Mixes up to 32 audio streams before routing out of the ACU device as an analog signal.

Table 208: ACUchannel audio input

Table 209, "ACUchannel audio output" below lists and describes **ACUchannel** audio output variable:

Name	Type	Default Value	Description
AudioIn	**audio**	N/A	Audio routed in from the ACU device on the specified channel that becomes available to other components in the model.

Table 209: ACUchannel audio output

Table 210, "ACUchannel control inputs" below lists and describes **ACUchannel** control input variables:

Name	Type	Default Value	Description
AudioInGain	**float32**	**1.0**	Provides *AudioIn's* gain control.
AudioOutGain	**float32**	**1.0**	Provides *AudioOut*'s gain control.
DigitalOut	**Boolean**	**FALSE**	The ACU digital output allows the software to drive a digital output for **ACUchannel**. One digital output exists per channel.
Volume	**float32**	**1.0**	Sets the **ACUchannel's** main volume.

Table 210: ACUchannel control inputs

Table 211, "ACUchannel control outputs" on the next page lists and describes **ACUchannel** control output variables:

Name	Type	Default Value	Description
PTT	**Boolean**	**FALSE**	When **TRUE**, *PTT* is on for transmission.
PTTselect	**uint8**	**255**	Digital in for *PTT*. Ranges 1–255.
PTT1	**Boolean**	**FALSE**	When **TRUE**, the *PTT* is on for transmission.

Name	Type	Default Value	Description
PTT1select	**uint8**	**255**	Digital in for the *PTT*. Ranges 1–255
PTT2	**Boolean**	**FALSE**	When **TRUE**, the *PTT* is on for transmission.
PTT2select	**uint8**	**255**	Digital in for the *PTT*. Ranges 1–255
PTT3	**Boolean**	**FALSE**	When **TRUE**, the *PTT* is on for transmission.
PTT3select	**uint8**	**255**	Digital in for the *PTT*. Ranges 1–255
DigitalIn1	**uint8**	**0**	Allows the ACU to drive a digital input to the software and allows direct connection of *PTT*. Each ACU has three digital inputs per channel.
DigitalIn2	**uint8**	**0**	Allows the ACU to drive a digital input to the software and allows direct connection of *PTT*. Each ACU has three digital inputs per channel.
DigitalIn3	**uint8**	**0**	

Table 211: ACUchannel control outputs

Table 212, "ACUchannel internal parameters" below lists and describes **ACUchannel** internal parameter variables:

Name	Type	Default Value	Description
Channel	**acenet_channel**	**None**	Selects the **ACUchannel**.
Identifier	**device_id**	**<Select>**	Selects the name of the ACU device.
StateMachine	N/A	N/A	Defines variables related to a state machine device associated with the specified **ACUchannel**, such as a handheld terminal (HHT) or SINCGARS panel. State machine devices can only connect to channels A and C. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
Name	**string**	**<Edit>**	Identifies state machine instance in the initialization (.ini) file. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
EntityName	**string**	**<Edit>**	Identifies state machine .ini file. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
Type	**string**	**None**	Type of state machine device. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*

Table 212: ACUchannel internal parameters

11.4 ACU2channel

Summary: The software-configurable component for the ACU2 hardware device.

Description: The **ACU2channel** connects the software model audio to a channel of the physical ACU2 device. The control outputs provide the press-to-talk (PTT) channel selection for channels 0–3. Adjust audio gain with the *AudioOut* and *AudioIn* gains. The volume controls the overall channel's volume. The ACU2 has four stereo audio inputs and outputs that provide support for multiple mono or stereo operators and audio equipment.

Table 213, "ACU2channel audio inputs" below lists and describes **ACU2channel** audio input variables:

Name	Type	Default Value	Description
AudioOut	**audio[32]**	N/A	Mono audio output of the **ACU2channel**. Mixes up to 32 audio streams.
AudioOutsLeft	**audio[32]**	N/A	Audio that routes out the left stereo channel. This audio is mixes with the *AudioOut* audio streams. Mixes up to 32 audio streams.
AudioOutsRight	**audio[32]**	N/A	Audio that is routed out the right stereo channel. This audio also mixes with the *AudioOut* audio streams. Mixes up to 32 audio streams.

Table 213: ACU2channel audio inputs

Table 214, "ACU2channel audio output" below lists and describes the **ACU2channel** audio output variable:

Name	Type	Default Value	Description
AudioIn	**audio**	N/A	Audio routed in from the ACU2 device on the specified channel that becomes available to other components in the model.

Table 214: ACU2channel audio output

Table 215, "ACU2channel control inputs" below lists and describes **ACU2channel** control input variables:

Name	Type	Default Value	Description
AudioOutGain	**float32**	**1.0**	The *AudioOutGain* gain control.
AudioOutGainL	**float32**	**1.0**	The gain control for the *AudioOutsLeft* audio output.
AudioOutGainR	**float32**	**1.0**	The gain control for the *AudioOutsRight* audio output.
AudioInGain	**float32**	**1.0**	The *AudioIn* gain control.
DigitalOut	**Boolean**	**FALSE**	The ACU2 digital output allows the software to drive a digital output for the **ACU2channel**. There is one digital output per channel.
Volume	**float32**	**1.0**	Sets the main volume of the ACU2 outputs.

Table 215: ACU2channel control inputs

Table 216, "ACU2channel control outputs" below lists and describes **ACU2channel** control output variables:

Name	Type	Default Value	Description
AnalogIn1	**uint8**	**0**	Corresponds to *DigitalIn1*with a range of 1–255. Digital in for *PTT*. Used for four-channel selector *PTT*.
AnalogIn2	**uint8**	**0**	Corresponds to *DigitalIn2*with a range of 1–255. Used for four-channel selector *PTT*.
AnalogIn3	**uint8**	**0**	Corresponds to *DigitalIn3*with a range of 1–255. Digital in for the *PTT*. Used for four-channel selector *PTT*.
DigitalIn1–DigitalIn3	**Boolean**	**FALSE**	Allows the ACU2 to drive a digital input to the software and allows direct connection of *PTT*. Each ACU2 has three digital inputs per channel.
PTT	**Boolean**	**FALSE**	When **TRUE**, *PTT* is on for transmission.
PTTselect	**uint8**	**255**	Digital in for *PTT*; matches the **CommPanel** *InControl*/*OutControl* variables. Valid values are 0, 1, 2, 4, and 8.

Table 216: ACU2channel control outputs

Table 217, "ACU2channel internal parameters" on the next page lists and describes **ACU2channel** internal parameter variables:

Name	Type	Default Value	Description
Channel	**acu2_channel**	**None**	Select the ACU2 channel A–D.
Identifier	**device_id**	**<Select>**	Select the name of the ACU2 device.
StateMachine			Defines variables related to a state machine device associated with the specified **ACU2channel**, such as an HHT or SINCGARS panel. State machine devices can only connect to channels A and C. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*

Name	Type	Default Value	Description
Name	**string**	**<Edit>**	Identifies state machine instance in the initialization (.ini) file. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
EntityName	**string**	**<Edit>**	Identifies state machine initialization file. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*
Type	**Type**	**NONE**	Type of state machine device. ***Important:*** *This variable is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*

Table 217: ACU2channel internal parameters

11.5 ACU2_SerialByteOut

Summary: This component transmits up to eight bytes of data from an ACU2 ACENet device using the serial port interface.

Description: This device transmits a range of bytes (i.e., 0–8) from the ACENet device's serial port interface. The bytes transmit whenever *MsgSize* updates and is not 0. The bytes also transmit whenever *Char* numbers are updated and as long as the updated *Char* number is in the range of bytes transmitted, as specified by *MsgSize*.

Table 218, "ACU2_SerialByteOut control inputs" below lists and describes **ACU2_SerialByteOut** control inputs variables:

Name	Type	Default Value	Description
MsgSize	**uint32**	**0**	Specifies the number of bytes that transmit from *CharsOut*. The range for this variable is [0,8]. When this variable changes, the corresponding number of bytes transmit in order. • *Modifier*: add (**+**) • *Modifier_default*: **0**
CharsOut	**uint8**	**0**	An array of eight bytes (i.e., uint8). The resulting bytes in this array transmit in order from *Char1* to the number specified in *MsgSize*. If *MsgSize* is 0, no bytes transmit. If *MsgSize*is not 0 and a byte is updated in the range [*Char1*, *Char<MsgSize>*], the whole range transmits. • *Modifier*: add (**+**) • *Modifier_default*: **0**

Table 218: ACU2_SerialByteOut control inputs

Table 219, "ACU2_SerialByteOut internal parameters" below lists and describes **ACU2_SerialByteOut** internal parameter variables:

Name	Type	Default Value	Description
Identifier	**device_id**	**<Select>**	The name of the ACU2 goes here.
Channel	**acu2_channel**	**A**	Choose an ACU2 channel to transmit serial data. Serial Port A requires Channel A, while Serial Port B requires Channel C.
BaudRate	**serial_baud_rate**	**baudrate_4800**	Choose the baud rate of the serial data.
SerialSigType	**serial_signal_type**	**RS422**	Choose the serial signal type.

Table 219: ACU2_SerialByteOut internal parameters

11.6 AmpOut

Summary: The software-configurable component for the amplifier hardware devices.

Description: **AmpOut** drives up to 32 audio streams from within the model to a specified channel of the amplifier.

Table 220, "AmpOut audio input" below lists and describes the **AmpOut** audio input variable:

Name	Type	Default Value	Description
AudioOuts	**audio[32]**	N/A	Mixes up to 32 audio streams before routing out of the amplifier channel.

Table 220: AmpOut audio input

Table 221, "AmpOut control inputs" below lists and describes **AmpOut** control input variables:

Name	Type	Default Value	Description
AudioOutGain	**float32**	**1.0**	Provides the audio output's overall gain control.
LimiterEnable	**Boolean**	**TRUE**	Turns LIMITER on or off. If **TRUE**, **AmpOut's** peak output levels do not exceed *LimiterThreshold*.
LimiterRelease	**float32**	**100.0**	The rate in dB/sec that determines how fast LIMITER responds to a signal-level drop. Higher releases cause a faster gain increase, ensuring quiet signals do not get quieter.
LimiterThreshold	**float32**	**0.0**	The maximum peak level in dB allowed in the output signal. A value of -12 corresponds to 0.25 linear or 1/4 a full-scale signal.
Volume	**float32**	**1.0**	Sets the amplifier's normal volume.

Table 221: AmpOut control inputs

Table 222, "AmpOut internal parameters" below lists and describes **AmpOut** internal parameter variables:

Name	Type	Default Value	Description
Channel	**acenet_channel**	**None**	Selects the amplifier channel.
Identifier	**device_id**	**<Select>**	Select the amplifier's name.

Table 222: AmpOut internal parameters

11.7 RTPStream

Summary: The **RTPStream** component sends and receives audio over the network using Real-time Transport Protocol (RTP).

Description: To set up an **RTPStream**, complete the following steps:

1. Define RTP streams in an **RTP Stream Map** (.rtp) file, as described in Section 11.7.1, "Add an RTP Stream Map" on page 189.
2. Link the new **RTP Stream Map** to a Telestra server, as described in Section 11.7.2, "Assign the RTP Stream Map to a Telestra server" on page 191.
3. Configure **RTPStream** variables using the variable descriptions below.

Table 223, "RTPStream audio inputs and outputs" below lists and describes **RTPStream** audio inputs and outputs:

Name	Type	Default Value	Description
Audio Input			
TxAudio	**audio**	N/A	The **RTPStream's** audio source.
Audio Output			
RxAudio	**audio**	N/A	The received network RTP audio.

Table 223: RTPStream audio inputs and outputs

Table 224, "RTPStream control inputs" below lists and describes **RTPStream** control inputs:

Name	Type	Default Value	Description
NameIn	**string**	N/A	Determines which stream configuration to use in the **RTP Stream Map**. This input overrides the *Name* internal parameter variable.
TxEnabled	**boolean**	**FALSE**	When **TRUE**, **RTPStream** streams audio. If the input audio signal is inactive or no signal is connected, **RTPStream** sends packets of silent audio. When **FALSE**, **RTPStream** does not stream audio.

Table 224: RTPStream control inputs

Table 225, "RTPStream internal parameters" below lists and describes **RTPStream** internal parameters:

Name	Type	Default Value	Description
Name	**string**	N/A	Determines which stream configuration to use in the **RTP Stream Map**.
PacketLength	**packet_length**	**_20_ms**	Sets the audio length to send per packet.
PayloadFormat	**payload_format**	**Mu_Law_8k**	Sets the payload's audio-encoding format.

Table 225: RTPStream internal parameters

In RTP, the payload type is a 7-bit number that represents the type of encoding that the stream uses. Table 226, "RTPStream payload formats" below lists and describes **RTPStream** payload formats:

Payload Format	Type	Description
Mu_Law_8k	0	8-bit, 8 kHz μ-law algorithm
A_Law_8k	8	8-bit, 8 kHz A-law algorithm
A_Law_16k	92	8-bit, 16 kHz A-law algorithm
PCM_16_16k	93	16-bit, 16 kHz PCM
Float_48k	94	32-bit, 48 kHz floating point
PCM_16_48k	95	16-bit, 48 kHz PCM

Table 226: RTPStream payload formats

Table 227, "RTPStream packet lengths" below describes the length of the audio sample **RTPStream** sends in a single packet:

Packet Length	Description
_4_ms	The packet contains 4 ms of audio.
_8_ms	The packet contains 8 ms of audio.
_12_ms	The packet contains 12 ms of audio.
_16_ms	The packet contains 16 ms of audio.
_20_ms	The packet contains 20 ms of audio.

Table 227: RTPStream packet lengths

11.7.1 Add an RTP Stream Map

An **RTP Stream Map** is an a configuration file that defines RTP streams and their corresponding parameters (i.e., the destination IP address, local port numbers, and remote port numbers). To add an **RTP Stream Map** file and define RTP streams, follow these steps:

1. From the left menu, go to **servers**.

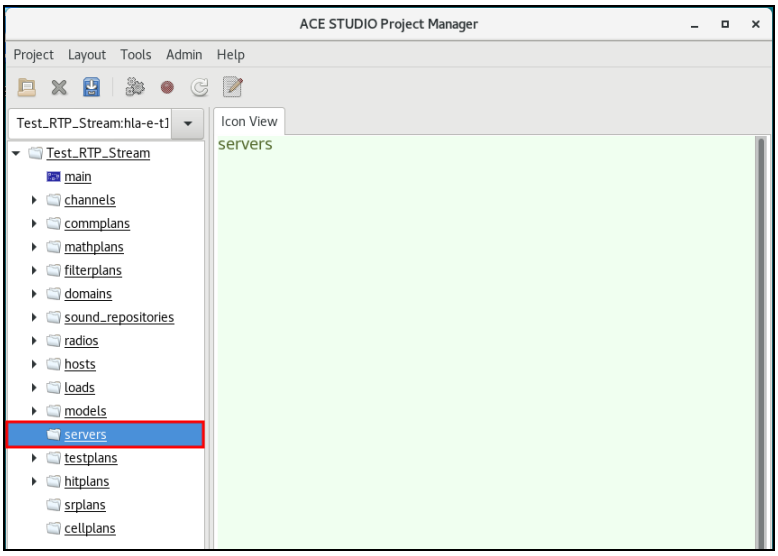


Figure 37: servers navigation

2. Right-click the **servers** canvas, and select **Add**.
3. Select the **New Item** drop-down list, choose **RTP Stream Map**, and enter a unique name.
4. Select OK.
5. Under **servers**, select the new .rtp file, and then select Edit File.

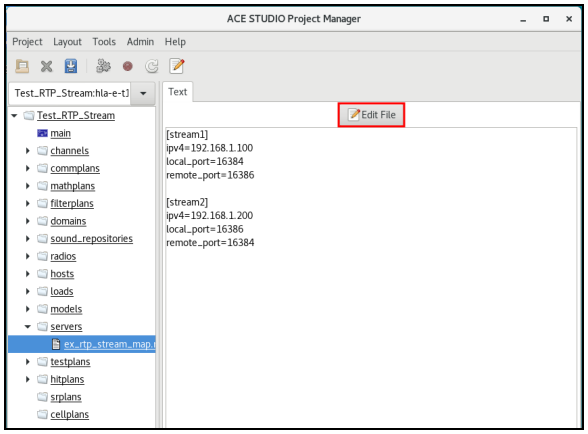


Figure 38: Edit RTP Stream Map .rtp file

6. In the text editor, enter the following parameters for each stream:

   - ***[StreamName]***: where *StreamName* is the name of the stream defined in brackets (e.g., [stream1], [stream2]). **RTPStream** uses this name to choose a configuration. Stream names must follow initialization (.ini) file format.
   - ***ipv4***: In **IP Address**, enter *xxx.xxx.xxx.xxx*, where *xxx.xxx.xxx.xxx* is the IPv4 address or hostname of an ED-137 radioendpoint on the network. In a typical configuration, this radioendpoint serves as the destination for outgoing streams and the source for incoming streams.
   - ***local_port***: the port that **RTPStream** uses to listen for incoming streams.
   - ***remote_port***: the remote port's streaming destination.

   An **RTP Stream Map** .rtp file might resemble the following:

```
[stream1]
ipv4=192.168.1.100
local_port=16384
remote_port=16386

[stream2]
ipv4=192.168.1.200
local_port=16386
remote_port=16384
```

7. Select Save.

11.7.2 Assign the RTP Stream Map to a Telestra server

To assign the **RTP Stream Map** to a Telestra server, follow these steps:

1. From the left menu, choose a layout (e.g., **main**).

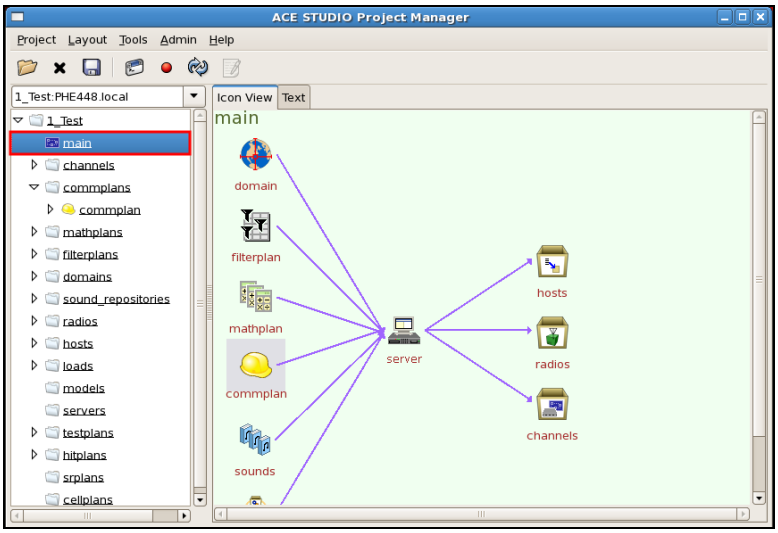


Figure 39: Layout navigation

2. In **Icon View**, right-click a Telestra server (), and choose **Edit**.
3. In **Telestra Editor**, go to **SIM SERVER**.
4. Select **RTP Plan**, and choose the **RTP Stream Map** you created in Section 11.7.1, "Add an RTP Stream Map" on page 189.

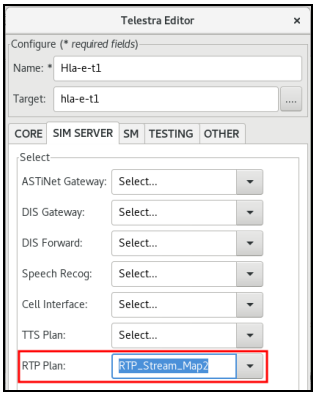


Figure 40: Assign the RTP Stream Map to RTP Plan

5. Select Update.
6. From the toolbar, select **Project** > **Save**.

7. From the toolbar, select **Install Layout** ( ).

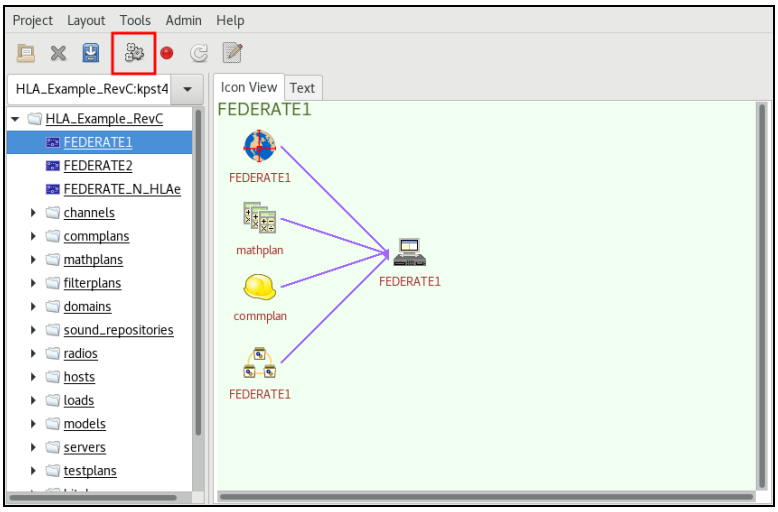


Figure 41: Install Layout

11.8 SerialPort

Summary: This component is an internal ASTi component that debugs ACE-RIU channels.

11.9 VoisusChannel

Summary: **VoisusChannel** provides the audio communication and radio control for a remote client.

Important: *This feature is unavailable in Red Hat Enterprise Linux (RHEL) 7 and later.*

Description: **VoisusChannel** works with **CommPanels** and multiple radio components to provide remote radio control. Use **VoisusChannel** to provide press-to-talk (PTT), output, radio net selection, and volume controls for the remote client. This component accepts receive and transmit states and **commplan** configuration for the remote client radios. Several settings per radio (i.e., 1–16) define how the remote operator controls the radio.

Table 228, "VoisusChannel audio inputs" below lists and describes **VoisusChannel** audio input s variables:

Name	Type	Default Value	Description
AudioOut	**audio**	N/A	Audio sent to the remote client.
Sidetone	**audio**	N/A	Audio sent to the remote client as the sidetone signal.

Table 228: VoisusChannel audio inputs

Table 229, "VoisusChannel audio output" below lists and describes VoisusChannel audio output variable:

Name	Type	Default Value	Description
AudioIn	**audio**	N/A	Audio routed from the remote client on the specified channel.

Table 229: VoisusChannel audio output

Table 230, "VoisusChannel control inputs" below lists and describes **VoisusChannel** control input variables:

Name	Type	Default Value	Description
Fill	**string**	N/A	Identifies the Comm Plan fill from the connected radio (e.g., one input per radio, Radio 1–16).
NetIn	**uint32**	**0**	The current net selection of the specified radio (e.g., one input per radio, Radio 1–16).
Receiving	**Boolean**	**FALSE**	When **TRUE**, the specified radio actively receives (e.g., one input per radio, Radio 1–16).
Transmitting	**Boolean**	**FALSE**	When **TRUE**, the specified radio actively transmits (e.g., one input per radio, Radio 1–16).

Table 230: VoisusChannel control inputs

Table 231, "VoisusChannel control outputs" below lists and describes **VoisusChannel** control output variables:

Name	Type	Default Value	Description
PTT	**Boolean**	**FALSE**	When **TRUE**, audio actively transmits.
InControlA	**uint8**	**0**	Selects the radios (i.e., 1–8) in the **CommPanel** to transmit on (i.e., bit mask).
InControlB	**uint8**	**0**	Selects the radios (i.e., 9–16) in the **CommPanel** to transmit on (bit mask).
NetOut	**int32**	**0**	Net selection from the remote client (e.g., one output per radio, Radio 1–16).
OutControlA	**uint8**	**0**	Selects the radios (i.e., 1–8) to receive from the bit mask.
OutControlB	**uint8**	**0**	Selects the radios (9–16) to receive from the bit mask.
Volume	**float32**	**0.0**	Sets the radio receive gain (e.g., one output per radio, Radio 1–16).

Table 231: VoisusChannel control outputs

Table 232, "VoisusChannel internal parameters" below lists and describes **VoisusChannel** internal parameter variables:

Name	Type	Default Value	Description
DefaultNet	**uint8**	**0**	The initial net that the radio tunes to after an installation (e.g., one parameter per radio, radios 1–16).
DefaultRxState	**uint8**	**0**	The initial receive/transmit settings after an install (e.g., one parameter per radio, radios 1–16).
DeviceName	**string**	**<Edit>**	The name given to the operator instance in the model.
NetLock	**Boolean**	**FALSE**	Controls whether the remote client can change the default net selection (e.g., one parameter per radio, Radio 1–16)
NumRadios	**uint8**	**0**	Number of attached radios that the remote client controls.
RxLock	**Boolean**	**FALSE**	Controls whether the remote client can change the default receive state (e.g., one parameter per radio, Radio 1–16).

Table 232: VoisusChannel internal parameters

12.0 Intercom

The **Intercom** components provide an audio bus service to which other components can connect, distributing audio throughout a model. Additionally, the intercom service and components can simulate the intercom bus structures of real aircraft and other training applications. This section provides details on the following intercom components:

- **IcomRx**
- **IcomTx**
- **Intercom_Bus_Power**

This section also describes the **IntercomBusService**.

12.1 IcomBalancer8

Summary: Aural cue applications with multiple speakers that distribute the same sound can use **IcomBalancer8** to individually set each speaker's volume.

Description: IcomBalancer8 places a single audio signal on multiple intercom channels. To adjust the aural cue's apparent location relative to its speaker position, set the individual gain of each intercom bus. Each component can route an audio signal to eight or fewer intercom buses.

__Note__: To route audio from the intercom buses to the speakers, use the __IcomRx__ component. To learn more about __IcomRx__, go to Section 12.2, "IcomRx" on the next page.

Table 233, "IcomBalancer8 audio input" below lists and describes the **IcomBalancer8** audio input:

Name	Type	Default Value	Description
InSignals	**audio**	N/A	An input mixer with 32 inputs; connects to one or more audio signal(s) that you will distribute to the specified intercom buses.

Table 233: IcomBalancer8 audio input

Table 234, "IcomBalancer8 control inputs" below lists and describes **IcomBalancer8** control inputs:

Name	Type	Default Value	Description
InGain	**basic/float32**	N/A	A gain applied to the sum of the *InSignals*.
Gain1–Gain8	**basic/float32**	**0**	Multipliers to the assigned *OutBus* audio bus signals. **IcomBalancer8** multiplies each audio buses signal amplitude by its corresponding gain value before transmitting it to the audio bus.

Table 234: IcomBalancer8 control inputs

Table 235, "IcomBalancer8 internal parameters" below lists and describes **IcomBalancer8** internal parameter variables:

Name	Type	Default Value	Description
InSignalResult	**audio**	N/A	The audio signal that **IcomBalancer8** will distribute to the specified intercom buses; displays the InSignals result with InGain applied.
OutBus1–OutBus8	**id**	N/A	Selects the intercom bus handle.

Table 235: IcomBalancer8 internal parameters

12.2 IcomRx

Summary: **Intercom Receiver** (**IcomRx**) provides a connection from an intercom bus within the model. The purpose of this component is to provide a simplified means of retrieving audio from an intercom bus without using a **CommPanel**. This capability is useful for monitoring audio on a particular intercom bus within the model. **IcomRx** cannot transmit audio to an intercom bus.

This component also provides the bus with a power state. If there is not an intercom object included for a particular bus, then the bus is fully operational by default.

Description: **IcomRx** provides power conditions a specific bus and a direct output tap of the bus composite audio.

This component maps to the same connection required for a radio object. These two service-side connections have values of **1.0** when the intercom object power is **TRUE** and **0.0** when the power is **FALSE**. Basically, the intercom bus is off if the power is off.

The intercom bus parameter defines which bus the audio is pulled from. This parameter maps to the handle of the intercom bus.

The **IcomRx** output is the audio that comes from the intercom bus. *OutGain* controls the amplitude of the output signal. If this variable is 0.0, **IcomRx** does not output a signal.

Table 236, "IcomRx audio output" below lists and describes the **IcomRx** audio output variable:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	Outputs audio from the selected intercom bus.

Table 236: IcomRx audio output

Table 237, "IcomRx control input" below lists and describes the **IcomRx** control input variable:

Name	Type	Default Value	Description
OutGain	**float32**	**1.0**	*OutGain* applies gain control to the signal that is retrieved from the intercom bus. • *Modifier*: multiply (*) • *Modifier_default*: **1.0**

Table 237: IcomRx control input

Table 238, "IcomRx internal parameter" below lists and describes IcomRx inputs, outputs, and internal parameters:

Name	Type	Default Value	Description
Channel	**id**	**UNASSIGNED**	*Channel* selects the intercom bus identifier.

Table 238: IcomRx internal parameter

12.3 IcomTx

Summary: **Intercom Transmitter** (**IcomTx**) connects to an intercom bus within the model. This component provides a simple means of placing audio on an intercom bus without a **CommPanel**. This capability is useful for distributing audio from a single source that should be heard through multiple comm panels in the model. The source audio for transmission comes from an external signal connection into **IcomTx**. This component cannot retrieve audio from an intercom bus.

Description: **IcomTx** provides power conditioning of a specific bus and an input for an auxiliary signal to be placed onto the bus. Only one intercom object may connect to any individual intercom bus within the intercom service. Any other configuration is invalid. **IcomTx** maps exactly to the same connections required for a radio transmitter object. The parameter intercom bus determines onto which bus the audio is injected. This parameter maps to the handle input to the intercom service. Since **IcomTx** cannot retrieve audio from a bus, no output signal is required.

Table 239, "IcomTx audio input" below lists and describes the **IcomTx** audio input variable:

Name	Type	Default Value	Description
InSignal	**audio**	N/A	Connects a signal into the selected intercom bus for transmission.

Table 239: IcomTx audio input

Table 240, "IcomTx control input" below describes the **IcomTx** control input:

Name	Type	Default Value	Description
InGain	**float32**	**1.0**	Applies gain control to *InSignal* before it transmits on the intercom bus.

Table 240: IcomTx control input

Table 241, "IcomTx internal parameter" below describes the **IcomTx** internal parameter:

Name	Type	Default Value	Description
Channel	**id**	**UNASSIGNED**	Channel selects the intercom bus identifier.

Table 241: IcomTx internal parameter

12.4 Intercom_Bus_Power

Summary: Provides power for **IcomRx** and **IcomTx** to play audio over an intercom bus.

Description: **Intercom_Bus_Power** selects an intercom bus to control power. *Power* toggles **Intercom_Bus_Power**. *IntercomBus* is turned on by default.

Table 242, "Intercom_Bus_Power audio input" below describes the **Intercom_Bus_Power** audio input variable:

Name	Type	Default Value	Description
Power	**Boolean**	**FALSE**	Toggles the intercom bus power. • *Modifier*: **XOR** • *Modifier_default*: **TRUE**

Table 242: Intercom_Bus_Power audio input

Table 243, "Intercom_Bus_Power internal parameter" below describes **Intercom_Bus_Power** internal parameter:

Name	Type	Default Value	Description
IntercomBus	**id**	**UNASSIGNED**	Selects the intercom bus to toggle power.

Table 243: Intercom_Bus_Power internal parameter

12.5 IntercomBusService

Summary: **IntercomBusService** provides an invisible audio connection between objects that are attached to a common intercom channel. All connectivity between a specific operator's CommPanel and the source and source/sink objects is carried out via intercom service links. The service supports multiple simultaneous channels that operate in isolation from any others. Inputs to a particular channel mix. Outputs from a channel are common to all destination objects, with one exception related to the operation of sidetone. Sidetone is the return signal from an object that has transmit and receive capabilities, such as a radio. Go to the description below for a full definition of the sidetone characteristic. Both half and full duplex connectivity between objects is supported depending on the source/sink object type.

***Note**: As with all service-type components, you do not manually create the service object; the loader automatically creates the service object when it detects that a component has an intercom service port connection. Only one service component of the appropriate type loads based on the first found need for a service of this type.*

Description: **IntercomBusService** provides an *N* channelized audio mixer and distribution service. **IntercomBusService** provides a centralized point for the generation of sidetone signal return to source objects. The majority of voice systems receive a portion of the source signal as a return, ensuring that the system is functional. The sidetone must generate cleanly since there are many potential issues related to multiple signal returns to an operator. When you should hear sidetone, the receive signal should not contain the same source. A return path must provide a local loop for the sidetone to ensure that the source signal subtracts from the receive signal. This action must also accommodate any processing delays within the signal paths.

This service also includes some intelligence, allowing it to condition the return of sidetone based on state data returned from a connected object, specifically when used to link an operator to a radio or network intercom object. The radio object must provide a sidetone state return that conditions whether any sidetone returns to any operators on the channel.

IntercomBusService touches many other components throughout a model. As a result, define the nature of the service-edge primitive. These primitives provide the component interface to and from the **IntercomBusService**. This interface passes the input audio to the service, receives the return audio from the service, and passes additional data into the service. The service includes the channel index set by the model developer **Channel Handle**, sidetone gain, audio activity status, and component type. The component type determines whether the sidetone gain acts as a local loop gain or gate value for local radios or the net intercom. The Channel Handle is the only user-defined input to this primitive. No other inputs or settings are required.

Within the tool, a handle (i.e., bus name) is defined, and the tool assigns an internal index number to the channel. By default, the tool assigns the channel numbers incrementally. Internally, **IntercomBusService** uses the channel number to link inputs and outputs. Channel numbers do not display. All displays related to the handles must be in alphabetic order to help you find required handles for assignment.

Intercom also supports diagnostic facilities, allowing you to view intercom channels in the model. This function scans the model for all objects using the service and extracts the information into a table-like view.

The **Assigned Buses** view displays all components connected to each handle:

Component Name	Var	Bus
/Intercom_Intercom_Bus_Power	IntercomBus	bus_number1
/Intercom_IcomRx1	Channel	bus_number1
/Intercom_IcomRx1	IcBus.channel_Id	bus_number1
/Intercom_IcomTx1	Channel	bus_number1
/Intercom_IcomTx1	IcBus.channel_Id	bus_number1
/Intercom_IcomRx2	Channel	bus_number2
/Intercom_IcomRx2	IcBus.channel_Id	bus_number2
/Intercom_IcomTx2	Channel	bus_number2
/Intercom_IcomTx2	IcBus.channel_Id	bus_number2

Table 244: Components connected to Assigned Buses

13.0 Platform

The following section details **Platform** and the objects within them. **Platform** components include the following:

- **Detonation**
- **Entity**
- **Fire**
- **GeocentricWorldPosition**
- **GeodeticWorldPosition**
- **RelativePosition**

13.1 Detonation

Summary: **Detonation** looks at **Detonation** protocol data units (PDUs) on the Distributed Interactive Simulation (DIS) network and outputs their values for use within the model.

Description: This component is used primarily for triggering sounds and events based on incoming **Detonation** PDUs from the DIS network. **Detonation** presents relevant variables needed to determine the sound, while it ignores other parts of the PDU. It holds all values until it detects another **Detonation** PDU. *Serial* increments with every **Detonation** PDU detected on the network. The **PDU Type** must equal 3 for **Detonation** to present values. All nomenclature is based on the DIS standard for **Detonation** PDUs.

Table 245, "Detonation control outputs" below lists and describes **Detonation** control output variables:

Name	Type	Default Value	Description
Serial	**uint32**	**0**	Increments with each incoming **Detonation** PDU, such as Type 3 DIS PDUs. All other values in **Detonation** update with the *Serial* value based on the incoming packet.
LocationX, LocationY, LocationZ	**float32**	**0.0**	The geocentric location in meters.
VelocityX, VelocityY, VelocityZ	**float32**	**0.0**	The geocentric velocity in meters per second.

Table 245: Detonation control outputs

13.2 Entity

Summary: Entity creates a fully Distributed Interactive Simulation (DIS)-compliant entity protocol data unit (PDU) at a given location.

Description: This component is used primarily to allow the creation of DIS Entity PDUs at a given location without Radio PDUs. It informs other entities of an ownship's location for targeting or other purposes.

Table 246, "Entity control outputs" below lists and describes **Entity** control output variables:

Name	Type	Default Value	Description
EntityNameIn	**string**	**<Edit>**	The entity name. This variable can either be hard -coded or driven from a host interface.
DomainNameIn	**string**	**<Edit>**	The domain name. This variable can either be hard -coded or driven from a host interface.
ProtocolIdIn	**string**	**<Edit>**	The protocol ID. This variable can either be hard- coded or driven from a host interface.
WorldPositionBus	**id**	**UNASSIGNED**	Links to a defined world position.
ForceCenterOfEarth	**Boolean**	**FALSE**	When set to **TRUE**, *WorldPositionBus* is ignored, and the world position is set to the center of the earth.
MyLocationX, MyLocationY, MyLocationZ	**float32**	**0.0**	These variables display and output the current world position.

Table 246: Entity control outputs

13.3 Fire

Summary: **Fire** looks at **Fire** protocol data units (PDUs) on the Distributed Interactive Simulation (DIS) network and outputs their values for use within the model.

Description: **Fire** triggers sounds and events based on incoming Fire PDUs from the DIS network. **Fire** presents relevant fields needed to determine the sound, while other parts of the PDU are ignored. All values are held until another **Fire** PDU is detected. *Serial* increments with every **Fire** PDU detected on the network. **PDU Type** must equal two for **Fire** to present the values. All nomenclature is based on the DIS standard for **Fire** PDUs.

Table 247, "Fire control outputs" below lists and describes **Fire** control output variables:

Name	Type	Default Value	Description
Serial	**uint32**	**0**	Increments with each incoming **Fire** PDU, such as Type 2 DIS PDUs. All other values in **Fire** update with PDU Value 2, based on the incoming packet.
LocationX, *LocationY*, *LocationZ*	**float32**	**0.0**	The geocentric location in meters.
VelocityX, *VelocityY*, *VelocityZ*	**float32**	**0.0**	The geocentric velocity in meters per second.

Table 247: Fire control outputs

13.4 GeocentricWorldPosition

Summary: **GeocentricWorldPosition** provides a simple location feature that positions radios and transmitters.

Description: **GeocentricWorldPosition** of the transmitter and receiver compute the diminishing power and occult by the Earth for line-of-sight transmissions. The standard model of the Earth is a smooth ellipsoid. A terrain server may be used for accurate modeling.

GeocentricWorldPosition is identical to **GeodeticWorldPosition**, except that the input values for the position are given in terms of X, Y, and Z coordinates from the center of the Earth in meters.

If the world position is 0, 0, 0 (i.e., the center of the Earth), then the ranging effects of any attached radio are turned off, and the radio clearly receives all transmissions on its frequency.

Table 248, "GeocentricWorldPosition control inputs" below lists and describes the **GeocentricWorldPosition** control input variables:

Name	Type	Default Value	Description
X	**float64**	**0.0**	The *X* world position coordinate; units are in meters.
Y	**float64**	**0.0**	The *Y* world position coordinate; units are in meters.
Z	**float64**	**0.0**	The *Z* world position coordinate; units are in meters.

Table 248: GeocentricWorldPosition control inputs

Table 249, "GeocentricWorldPosition internal parameter" below describes the **GeocentricWorldPosition** internal parameter:

Name	Type	Default Value	Description
WorldPositionBus	**id**	**UNASSIGNED**	Selects the world position bus to assign a position.

Table 249: GeocentricWorldPosition internal parameter

13.5 GeodeticWorldPosition

Summary: The **GeodeticWorldPosition** component provides a simple location feature for the radio and transmitter positioning.

Description: The world positions of the transmitter and receiver are used to compute diminishing power and occulting by the earth for the line of sight transmissions. The model of the earth is a smooth ellipsoid (model WGS84). The **GeodeticWorldPosition** is specified in altitude in meters with latitude, and longitude in degrees.

Table 250, "GeodeticWorldPosition control inputs" below lists and describes **GeodeticWorldPosition** control input variables:

Name	Type	Default Value	Description
Elevation	**float64**	**0.0**	The altitude world position coordinate, units are in meters.
Latitude	**float64**	**0.0**	The latitude world position coordinate, units are in degrees.
Longitude	**float64**	**0.0**	The longitude world position coordinate; units are in degrees.

Table 250: GeodeticWorldPosition control inputs

Table 251, "GeodeticWorldPosition internal parameters" below and describes **GeodeticWorldPosition** internal parameter variables:

Name	Type	Default Value	Description
WorldPositionBus	**id**	**UNASSIGNED**	Selects the world position bus to assign a position.
Position	**woldposn_geocentric**		Selects the world position X, Y, Z coordinates.
X	**float64**	**6378137**	
Y	**float64**	**0.0**	
Z	**float64**	**0.0**	

Table 251: GeodeticWorldPosition internal parameters

13.6 RelativePosition

Summary: **RelativePosition** calculates the relative coordinates and velocities of an entity, used with the **Highway 3D Service** and other orientation-dependent sound models. Figure 42, "NED coordinate planes" below shows **RelativePosition** NED coordinate planes:

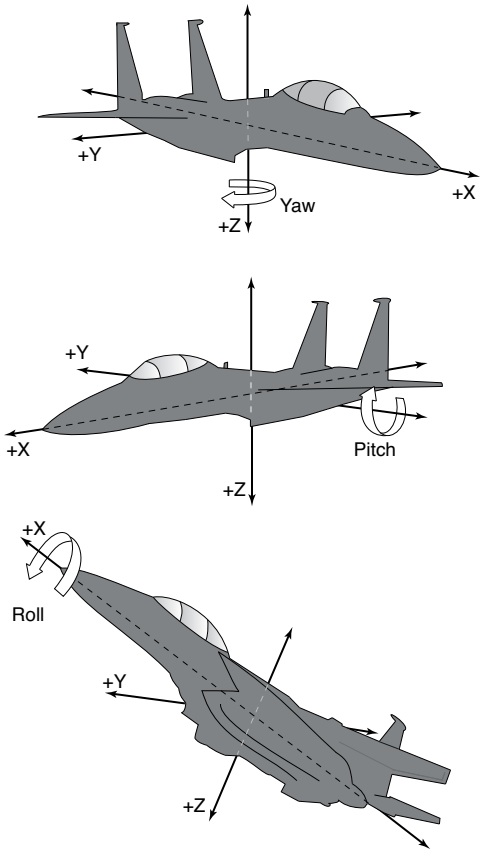


Figure 42: NED coordinate planes

Figure 43, "RelativePosition earth plane" below shows the **RelativePosition** earth plane:

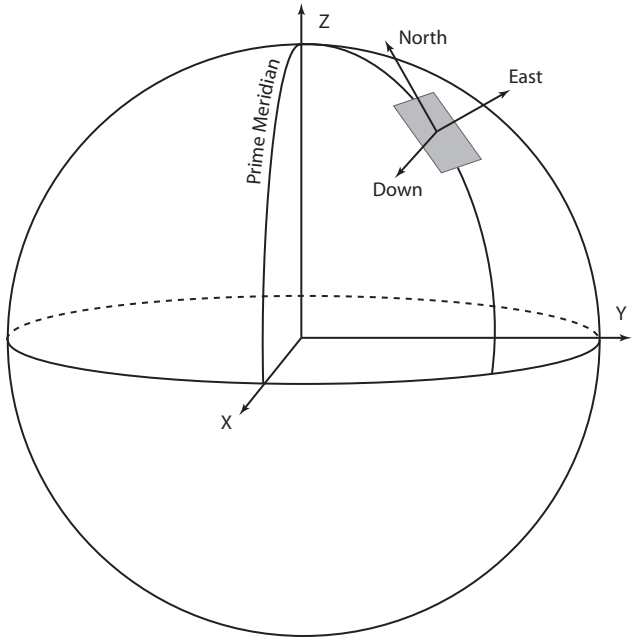


Figure 43: RelativePosition earth plane

Table 252, "RelativePosition control inputs" on the facing page lists and describes **RelativePosition** control input variables:

Name	Type	Default Value	Description
Ent_Alt	**float64**	**1.0**	The altitude of the entity position.
Ent_Lat	**float64**	**1.0**	The latitude of the entity position.
Ent_Lon	**float64**	**1.0**	The longitude of the entity position.
Ref_Alt	**float64**	**1.0**	The altitude of the reference position.
Ref_Lat	**float64**	**1.0**	The latitude of the reference position.
Ref_Lon	**float64**	**1.0**	The longitude of the reference position.
Ref_Yaw	**float64**	**1.0**	The measure of the angle formed from True North to the entity's +X axis. This angle is specified in degrees and is positive clockwise along the +Z axis.
Ref_Pitch	**float64**	**1.0**	The measure of the angle between the reference plane and the entity's +X axis. This angle is specified in degrees and is positive above the reference plane (i.e., away from the ellipsoid).

Name	Type	Default Value	Description
Ref_Roll	**float64**	**1.0**	The measure of the angle between the reference plane and the entity's +Y axis along a plane perpendicular to the entity's X axis. It is the angle of rotation about the X axis after applying the yaw and pitch. Roll is specified in degrees and is positive clockwise along the +X axis.

Table 252: RelativePosition control inputs

Table 252, "RelativePosition control inputs" above lists and describes **RelativePosition** control output variables:

Name	Type	Default Value	Description
Distance	**float64**	**0.0**	Distance from the reference point to the entity in meters.
ApproachSpeed	**float32**	**0.0**	The rate at which the entity and reference positions approach each other in meters per second.
Ent_X	**float64**	**6378137**	The X position of the entity in the geocentric coordinate system.
Ent_Y	**float64**	**0.0**	The Y position of the entity in the geocentric coordinate system.
Ent_Z	**float64**	**0.0**	The Z position of the entity in the geocentric coordinate system.
Ent_Speed	**float32**	**0.0**	The instantaneous speed of the entity in meters per second.
Ref_X	**float64**	**6378137**	The X position of the reference point in the geocentric coordinate system. Units are in meters.
Ref_Y	**float64**	**0.0**	The Y position of the reference point in the geocentric coordinate system. Units are in meters.
Ref_Z	**float64**	**0.0**	The Z position of the reference point in the geocentric coordinate system. Units are in meters.
Rel_X_Pos	**float64**	**0.0**	The X position of the entity relative to the position and orientation of the reference frame.
Rel_Y_Pos	**float64**	**0.0**	The Y position of the entity relative to the position and orientation of the reference frame.

Name	Type	Default Value	Description
Rel_Z_Pos	**float64**	**0.0**	The Z position of the entity relative to the position and orientation of the reference frame.
Rel_X_Vel	**float32**	**0.0**	The relative velocity in the X direction between the reference and entity positions.
Rel_Y_Vel	**float32**	**0.0**	The relative velocity in the X direction between the reference and entity positions.
Rel_Z_Vel	**float32**	**0.0**	The relative velocity in the X direction between the reference and entity positions.

Table 253: RelativePosition control outputs

Table 254, "RelativePosition debugging variables" below lists and describes **RelativePosition** debugging variables:

Name	Variable	Type	Default Value	Description
E_Axis	*X*	**float64**	**0.0**	The vector pointing east in the local geographic frame of the reference position.
	Y		**1.0**	
	Z		**0.0**	
D_Axis	*X*	**float64**	**-1.0**	The vector pointing down in the local geographic frame of the reference position.
	Y		**0.0**	
	Z		**0.0**	
N_Axis	*X*	**float64**	**0.0**	The vector pointing north in the local geographic frame of the reference position.
	Y		**0.0**	
	Z		**1.0**	
Vec_To_Entity	*X*	**float64**	**0.0**	Shows the direction to the entity from the reference position.
	Y		**0.0**	
	Z		**0.0**	
X_Axis	*X*	**float64**	**0.0**	The vector pointing forward from the reference position (e.g., the vector pointing through the nose of the airplane, if the airplane is the reference).
	Y		**0.0**	
	Z		**1.0**	
Y_Axis	*X*	**float64**	**0.0**	The vector pointing down from the reference position.
	Y		**1.0**	
	Z		**0.0**	
Z_Axis	*X*	**float64**	**-1.0**	The vector pointing down from the reference position.
	Y		**0.0**	
	Z		**0.0**	

Table 254: RelativePosition debugging variables

14.0 Host Control

Before adding **HostIns** and **HostOuts**, you must first add a host model. A host model is required to create the host I/O packets. Inside the host model, add a **HostIn** or **HostOut**, and add the host I/O components, also known as packets.

To debug the host control, figure out if packets are coming in off the network from the host. To determine packet activity, pay close attention to two items in the host interface:

- *Live Capture*: continuously displays packet activity from the host.
- *Controller*: displays packet statistics.

In *Controller* (i.e., **HostIn Viewer**), **Fail Count** always shows data if the host is working. The numbers continue to increment in a loop. **Total Packets Received** increments over time if the host control is working properly. *TestIn* and *TestOut* are for ASTi internal testing. Contact ASTi for more information.

14.1 HostIn

Summary: **HostInput** (**HostIn**) is a user interface wrapper for control data fields extracted from external sources, including host Ethernet control User Datagram Protocol (UDP) packets and state machine cells, providing a means to apply incoming control data to other components in the model. This section describes **HostIn** as used with the host Ethernet.

You can create and modify **HostIn** with **IO Packet Editor**. With this tool, define the source of the data packet by the UDP port. Extracted individual **HostIns** to other model components, serving as control variables.

Description: **IO Packet Editor** provides a means to functionally disassemble a host UDP packet into individual fields. **HostIn** is the composite collection of individual data fields in a specific packet. Other model components (i.e., data sinks) sometimes use **HostIn's** control variables.

HostIn is analogous to the various **Control** objects in **Model Builder**. There is one fundamental difference between **HostIn** and an (MB) control object: MB includes a separate control object for each data type, while **HostIn** accommodates all available data types.

The process for creating and using **HostIn** follows:

1. In **IO Packet Editor**, enter the UDP port name of the host packet and the most-least significant data order (i.e., big or little endian).
2. To create a variable within **HostIn**, enter the parameters defining the data field: offset byte location within the packet (or in the case of a Boolean field, offset byte and bit), the data type, and the initial value for the field.

After creating a **HostIn** data source, link it to a variable (i.e., data sink) inside another component. The action of making a link is done on the sink side of the link. The link is visible from the source side (i.e., the **HostIn Used By** shows the sink component name) and from the sink side (i.e., the sink component **From** shows **HostIn** name).

Table 255, "HostIn interface elements" on page 213 lists and describes HostIn interface elements:

Setting Name	Description
Name	A descriptor for the component.
Offset	The location of the data field within the packet, defined by the byte offset from the start of the data field. The location of a packed byte input is further defined by the bit position within a particular byte.
MsgLength	Message length in bytes.
Type	**HostIn** can be set to one of these data types: • Unsigned integer, 8-bit (uint8) • Unsigned integer, 16-bit (uint16) • Unsigned integer, 32-bit (uint32) • Unsigned integer, 64-bit (uint64) • Signed integer, 8-bit (int8) • Signed integer, 16-bit (int16) • Signed integer, 32-bit (int32) • Signed integer, 64-bit (int64) • Floating point, 32-bit (float32) • Floating point, 64-bit (float64) • Boolean, (byte) True_False • Character string (cstring) • Nav identifier (ident) • Packed byte (bits8_0–bits8_7) • Binary message (message) • Text file path (pathstring)
Init. Value	The data field's initial value can be set by the application should the source packet not be present due to failure to receive host packets. This is directly analogous to the initial value featured in MB Control objects.
Function	Add a basic function, including **Add**, **Linear**, **Logical-and**, **Logical-or**, **Logical-xor**, **Multiple**, and **Subtract**.
Y	Value used if a function is used. For more information, go to Section 5.14, "MathFunction" on page 109.
Z	Value used if a function is used. For more information, go to Section 5.14, "MathFunction" on page 109.
Rscale	Adds a scale factor to apply to the output of the function; works as the gain control. For more information, go to Section 5.14, "MathFunction" on page 109.

Setting Name	Description
Test Mode	A local test mode can override the incoming value from the host packet and manually set the value of a variable. Test modes for various fields can be selectively enabled. A master test mode enable switch activates the test mode for all test mode-enabled boxes. All boxes that are not test-mode enabled continue to get their value from either the host packet or their initial value if selected to do so and no host packets are being received. The application checks test mode values for correct range. If an out-of-range test value is entered, the application returns the default value. Selections = **ON** or **OFF**
Integers, all types	• **Test Mode Boolean**, default value =**OFF** • **Value Integer**, default value = 0 • **Ramp Rate (Hz) Floating Pt.**, default value = 0.0 • **Min Ramp Value Integer**, default value = 0 • **Max Ramp Value Integer**, default value = 0
Floats, all types	• **Test ModeBoolean**, default value = **OFF** • **ValueFloating Pt**, default value = 0 • **Ramp Rate (Hz) Floating Pt.**, default value = 0.0 • **Min Ramp Value Floating Pt**, default value = 0.0
Boolean	• **Test Mode Boolean**, default value = **OFF** • **Value Boolean**, default value = 0 • **Toggle Rate (Hz) Floating Pt**, default value = 0.0 • **Mark/Space Floating Pt**, default value = 0.5
For integers and floats	If **Test Mode** = ON and **Ramp Rate** = 0, the value sets a static test value. If **Test Mode** = ON and **Ramp Rate** > 0, the value is overridden, and a dynamic test value periodically ramps from **Min Ramp** to **Max Ramp** and back to **Min Ramp**. The **Ramp Rate** is in Hertz.
For Booleans	If **Test Mode** = ON and **Toggle Rate** = 0, the value sets a static test value. If **Test Mode** = ON and **Toggle Rate** >0, the value is overridden, and a dynamic test value toggles between 1 and 0. The **Toggle Rate** is in Hertz. The **Mark/Space** ratio controls how long the Boolean is 1 relative to the cycle duration. The Boolean is 0 for the period of the entire 1/0 toggle cycle (e.g., a value of 0.5 means that in a given period, the Boolean output is 1 for half of the time and 0 for the other half).
Test Value	Sets the desired test value the component will use.
Used By	Defines connection to a control sink point (i.e., variable) in another component. ***Note**: You can modify **HostIn's** value (e.g., inverted, scaled, offset) at the link's sink point.*
Other	Edits the dynamic test parameters, specifically the **<ramp>** setting, including mode, frequency, minimum, and maximum.

Setting Name	Description
Description	Used for comments.

Table 255: HostIn interface elements

Table 256, "HostIn control output" below lists and describes the **HostIn** control output variable:

Name	Type	Default Value	Description
N/A	Control data	N/A	The **HostIn** output value is based on parameters that you set (i.e., offset byte, data type, etc.).

Table 256: HostIn control output

Table 257, "HostIn internal parameters" on the next page lists and describes **HostIn** internal parameter variables:

Name	Type	Default Value	Description
Endianess	List	**Little Endian**	Changes the byte order. The endianness defines the byte order for the data in a packet.
Use Init Value	List	**Never**	This list box decides what should happen in the event of a loss of reception of UDP packets. • ***Never***: if UDP data stops, the outputs from **HostIn** go to zero. • ***Load***: when the project loads, **HostIn** outputs **Init. Value** values until it receives UDP packets. • ***Load/Source Fail***: upon loss of UDP packets for a period exceeding the value in **Timeout(s)**, **HostIn** outputs the values in **Init. Value**.
Enable Testing	Check box	**Selected**	Allows operation of **Test Mode**, **Test Value**, and **Other** for local testing. When cleared, these settings are disabled.
Timeout (s)	Entry box	**1**	Sets the threshold for assumed communication failure when a packet has not arrived within the set amount of time in seconds.
Clear Testmode	Radio button	N/A	This radio button resets any **On** entries in **Test Mode** to **Off**.

Name	Type	Default Value	Description
Align Offset	Radio button	N/A	Byte-aligns entries in the ICD. ***Note:*** *This variable does not work with packed bytes.*
Add To Connector	Radio button	N/A	Links selected entries in the ICD to a specified connector.
Live Capture	Select	N/A	Opens a window showing the current values for each entry in the ICD in real time. You can also capture and save the display as a comma-separated value (.csv) file.
Import ICD	Select	N/A	Select a .csv file to fill the entries in the **Host Packet Editor**.
Controller	Select	N/A	Opens the **HostIn Viewer** for the associated UDP interface for this packet.

Table 257: HostIn internal parameters

14.2 HostOut

Summary: **Host Output** (**HostOut**) provides a way to send control values from the model to a destination host computer via the system's Ethernet interface.

Create and modify **HostOuts** with the **IO Packet Editor**. With this tool, define the outgoing host data packet by the user datagram protocol (UDP) port, then link model control sources to individual control variables within the outgoing packet. Model control sources include any component with a control output:

- Host input
- Math components
- Component status fields

Description: **IO Packet Editor** provides a means to functionally assemble a host UDP packet from individual model controls. **HostOut** is a composite collection of specific data fields from model controls, packed into an outgoing Ethernet packet.

HostOut is analogous to the various control objects in **Model Builder**. There is one fundamental difference between **HostOut** and **Model Builder**:

1. Whereas the **Model Builder** included a separate control object for each data type, **HostOut** encompasses all variables within an outgoing packet.
2. Additionally **HostOut** accommodates all available data types.

The process for creating and using **HostOut** follows:

1. In **IO Packet Editor**, enter the UDP port of the destination host computer and the most-least significant data order (big or little endian).
2. To create a field within **HostOut**, define the parameters defining the data field (i.e., offset byte location within the packet, or in the case of a Boolean field, offset byte and bit and the data type) and link to a model control source.

Each field in **HostOut** is defined by the following:

- *Name*: contains a description of the component.
- *Offset*: the location of the data field within the packet, defined by the byte offset from the start of the data field. The location of a packed-byte input is further defined by the bit position within the byte.
- *MsgLen*: message length in bytes.
- *Type*: **HostOut** can be set to one of these data types:
  - Unsigned Integer, 8-bit (uint8)
  - Unsigned Integer, 16-bit (uint16)
  - Unsigned Integer, 32-bit (uint32)
  - Unsigned Integer, 64-bit (uint64)
  - Signed Integer, 8-bit (int8)
  - Signed Integer, 16-bit (int16)
  - Signed Integer, 32-bit (int32)
  - Signed Integer, 64-bit (int64)
  - Floating Point, 32-bit (float32)
  - Floating Point, 64-bit (float64)
  - Boolean, 1-byte
  - Character string (cstring)
  - Nav identifier (ident)
  - Packet byte (bits8_0–bits8_7)
  - Binary message (message)
  - Text file path (pathstring)

- *Used by*: the source field defines the link to a model control source.
- *Description*: comments field.

Table 258, "HostOut control output" below describes the **HostOut** control output variable:

Name	Type	Default Value	Description
N/A	Control data	N/A	The **HostOut** value is based on parameters (i.e., offset byte, data type, source, etc.).

Table 258: HostOut control output

Table 259, "HostOut internal parameters" below lists and describes **HostOut** internal parameters:

Name	Type	Default Value	Description
Endianness	List	**Little Endian**	Changes the byte order. The endianness defines the byte order for the data in a packet.
Timeout (s)	Entry box	**1**	Sets the threshold for assumed communication failure when a packet has not arrived within the set amount of time in seconds.
AlignOffset	Radio button	N/A	Correctly byte-aligns entries in the ICD.
AddToConnector	Radio button	N/A	Links selected entries in the ICD to a specified connector.
LiveCapture	Select	N/A	Opens a window showing the current values for each entry in the ICD in real time. You can also capture and save the display as a comma-separated value (.csv) file.
ImportICD	Select	N/A	Lets you upload a .csv file to fill **Host Packet Editor** entries.
Controller	Select	N/A	Opens the **HostOut Viewer** for the associated UDP interface for this packet.

Table 259: HostOut internal parameters

14.3 CellService

Summary: CellService provides an invisible data connection between objects that declare they are attached to a common cell channel. All connectivity between a specific cell channel and the source and source or sink objects is carried out via cell service links.

***Note**: As with all service-type components, you do not manually instigate the creation of the service object; this is done automatically on demand by the loader when it detects that a component has a cell service port connection. Only one service component of the appropriate type will be loaded based on the first found need for a service of this type.*

Description: The only user-defined input to this primitive will be the selected bus handle. No other inputs or settings are required. Available buses for use are declared through the **CellService** tool. Within the tool, a bus name is defined and the tool assigns an internal index number as a bus ID. By default, the tool assigns the bus ID numbers incrementally. Internally the cell service uses the bus ID number to link inputs and outputs.

14.3.1 CellIn

Summary: **Cell Input** (i.e., **CellIn**) serves as a user interface wrapper for control data fields extracted from external sources. These sources include host Ethernet control User Datagram Protocol (UDP) packets and state machine cells that apply incoming control data to other components in the model.

Description: **CellIn** is a collection of 40 individual data bytes in a specific packet. The individual bytes within **CellIn** (i.e., data sources) are used as control variables by other model components (i.e., data sinks).

Table 260, "CellIn control inputs" below lists and describes **CellIn** control input variables:

Name	Type	Default Value	Description
CellData	**uint8**	**0**	A collection of 40 unsigned bytes that connect to other components in the model.
NewCellData	**Boolean**	**FALSE**	When **TRUE**, new cell data UDP packets are recognized. This input was created for specific use by the LS653 state machine.

Table 260: CellIn control inputs

Table 261, "CellIn internal parameters" below lists and describes **CellIn** internal parameter variables:

Name	Type	Default Value	Description
CellBusId	**id**	**UNASSIGNED**	Assigns to **CellService**, linking components together.
CellReceiveCount	**uint32**	**0**	The count of the number of cells received from the cell daemon.

Table 261: CellIn internal parameters

14.3.2 CellOut

Summary: **CellOut** transmits bytes of data generated by specific components.

Description: **CellOut** provides a way to send control values from the model to a destination host computer via the system, using an Ethernet interface. **CellOut** is a composite collection of specific data bytes from model controls, packed into an outgoing Ethernet packet.

Table 262, "CellOut control output" below describes the **CellOut** control output variable:

Name	Type	Default Value	Description
CellData	**uint8**	**0**	*CellData* can only be a collection of 40 unsigned bytes, each of which can be used as a control to connect to other components within the model.

Table 262: CellOut control output

Table 263, "CellOut internal parameters" below lists and describes **CellOut** internal parameters:

Name	Type	Default Value	Description
CellBusId	**id**	**UNASSIGNED**	Assigns to Cell Bus Service, linking components together.
CellTransmitCount	**uint32**	**0**	The count of the number of cells transmitted from the cell daemon.

Table 263: CellOut internal parameters

15.0 Radio

Building radios in Telestra is slightly different from the traditional modeling used in other ASTi generations of software. A basic radio is now made up of two components: a **Transceiver** and a Radio Control Unit (RCU). The two objects are analogous to live radios on many aircraft. Typically, a base unit is responsible for the over-the-air transmission known as the radio transceiver (RT) and control heads that serve as the interface for the pilot or copilot.

The idea in Telestra is that every radio requires use of **Transceiver**, a master object which can be used for any radio, but it contains a minimum of settings. This way, if you want to simulate two different types of radios, the RCU is used to customize the **Transceiver** to be an ARC-210 versus an ARC-232.

The third piece of the radio is the fill that it receives from the **commplan** located in the project **Layout** of the project. The **commplan** provides a flexible way of adjusting internal radio parameters such as **Modulation Type** or the **Transmit Gain**. Previous generations of ASTi software accomplished the fill as a **Mode Table** within **Radio**. The fills in the Comm Plan make it easy to share mode settings across several radios in a model. In addition to the different mode settings, the fill of a radio can also set default frequencies.

The most commonly set radio uses two components:

- **Transceiver**
- **RCUbasic**

The RCU provides a fill from the **commplan** as well as a suite of parameters that can override their counterparts from within the fill. For example, **RCUbasic** has a loaded fill using Net 1, which sets the frequency of the radio to 101 megahertz (MHz). This information is sent down to **Transceiver** via *Transceiver ID* located in both components. To keep every setting in the net the same except for a new frequency of 105 MHz, set the frequency of the RCU to 105 MHz, which overrides the fill, and **Transceiver** acts accordingly.

A radio can switch between any number of nets, defined in the **commplan**, each of which can be custom-tailored to provide control over parameters such as modulation, noise, bandwidth, cryptosystem and key, frequency-hopping net ID, satellite communications (SATCOM), and other parameters. The default **commplan** fill supports the most common modes used by real radio frequency (RF) radios, including ultra high frequency (UHF), very high frequency (VHF), high frequency (HF), Single Channel Ground and Airborne Radio System (SINCGARS), and HAVEQUICK (HQ). This allows you to get started quickly while retaining the flexibility to further fine tune the simulation. **Transceiver** is also capable of receiving and transmitting Tactical Data Link (TDL) messages. Various voice-encoding schemes are also supported including Continuously Variable Slope Delta (CVSD), mu-law and pulse-code modulation (PCM).

In general, settings that relate to how the radio appears on the network are in **Transceiver**, and settings that customize the radio for simulation are placed in the RCU.

The following section details **Radio** components and the objects within them. **Radio** components include the following:

- **ColocatedBeacon**
- **GenericControl**
- **HfServer**
- **ICU**
- **IntercomTransceiver**
- **MarkerTone**
- **MorseKeyer**
- **RCUbasic**
- **RCUcryptokey**
- **RCUfrequency**
- **RCUhavequick**
- **RCUoverride**
- **RCUsincgars**
- **RCUtxpower**
- **Receiver**
- **Relay**
- **Satellite**
- **Transceiver**
- **Transmitter**
- **VORTAC_Controller**

15.1 ColocatedBeacon

Summary: **ColocatedBeacon** creates the audio to simulate colocated *VOR* and *TACAN* transmitters. Morse identifier tones generate for both transmitters.

Description: Allocated *VOR* and *TACAN* transmitters present a special case when simulating navigational aids. Both transmitters produce a Morse identifier that constantly broadcast over the air, but the IDs broadcast in sequence. As a result, **ColocatedBeacon** is actually two separate **MorseKeyers** that never play at the same time.

Figure 44, "ColocatedBeacon timing diagram" below shows basic timing:

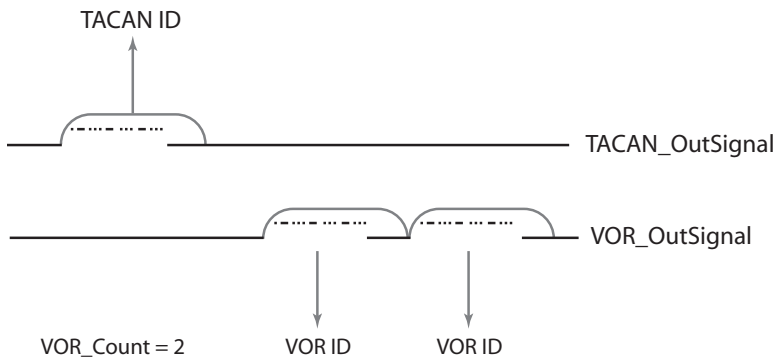


Figure 44: ColocatedBeacon timing diagram

Table 264, "ColocatedBeacon audio outputs" below lists and describes **ColocatedBeacon** audio output variables:

Name	Type	Default Value	Description
TACAN_OutSignal	**audio**	N/A	Outputs the keyed word for *TACAN* radio at appropriate intervals when **ColocatedBeacon** is enabled.
VOR_OutSignal	**audio**	N/A	Outputs the keyed word for *VOR* radio at appropriate intervals when **ColocatedBeacon** is enabled.

Table 264: ColocatedBeacon audio outputs

Table 265, "ColocatedBeacon control inputs" on the next page lists and describes **ColocatedBeacon** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**TRUE**	**TRUE** enables **ColocatedBeacon**.
Ident	**ident**	N/A	The ASCII characters driving **MorseKeyer**. *Ident* is special variable type defined as four ASCII characters combined together. Only **HostIn** can drive *Ident*.
Interval	**float32**	**1.0**	Sets delay between words in seconds.
TACAN_Frequency	**float32**	**1.0**	Provides frequency (i.e., pitch) of *TACAN* Morse tone. Units are in Hertz.
TACAN_Gain	**float32**	**1.0**	Scales strength of *TACAN* output signal's generated tone.

Name	Type	Default Value	Description
TACAN_ILSPause	**Boolean**	**FALSE**	Places space between *Ident's* first and second characters. When **FALSE**, *TACAN* keyer uses normal Morse code inter-character timing (i.e., three-dot gap between each *Ident* character). When **TRUE**, *TACAN* keyer uses standard Morse code interword timing between first and second characters (i.e., seven-dot gap). Inter-character timing for subsequent *Ident* characters use standard Morse code intercharacter spacing (i.e., three-dot gap).
VOR_Count	**uint16**	**1**	Number of times *VOR* identifier plays before keying *TACAN* identifier. *VOR* and *TACAN* identifier tones are mutually exclusive.
VOR_Frequency	**float32**	**1.0**	Sets *VOR* Morse tone's frequency (i.e., pitch). Units are in Hertz.
VOR_Gain	**float32**	**1.0**	Scales strength of *VOR* output signal's generated tone.
VOR_ILSPause	**Boolean**	**FALSE**	Places a space between the first and second characters of the *Ident*. When **FALSE**, the *VOR* keyer uses normal Morse code intercharacter timing, which is a three-dot gap between all *Ident* characters. When **TRUE**, the *VOR* Keyer uses a standard Morse code inter-word timing between the first and second characters (i.e., a seven-dot gap). Inter-character timing for subsequent *Ident* characters use standard Morse code inter-character spacing (i.e., a three-dot gap).
Wordrate	**uint8**	**1**	Determines the rate that the word keys in dots per second. Faster rates produce higher numbers.

Table 265: ColocatedBeacon control inputs

Table 266, "ColocatedBeacon control outputs" on the facing page lists and describes **ColocatedBeacon** control output variables:

Name	Type	Default Value	Description
TACAN_Active	**Boolean**	**FALSE**	**TRUE** when keying a dot/dash, mimicking the keyer's beep.

Name	Type	Default Value	Description
VOR_Active	**Boolean**	**FALSE**	When keying a dot or a dash, this variable becomes **TRUE**, mimicking the keyer's beep.
TACAN_Busy	**Boolean**	**FALSE**	Indicates if the *TACAN* keyer is actively producing Morse audio.
VOR_Busy	**Boolean**	**FALSE**	This flag indicates if the *VOR* keyer is actively producing Morse audio.

Table 266: ColocatedBeacon control outputs

15.2 GenericControl

Summary: Generic radio control for the radio simulated environment.

Description: **GenericControl** is a Radio Control Unit (RCU) designed for use where multiple RCUs are hooked onto the same **Transceiver**. It mimics **RCUbasic** with one key exception: any updates to the parameters of the RCU are sent to **Transceiver** on a change instead of all the time. This process allows the model to have two **GenericControls**, each with a different frequency, correctly driving **Transceiver**. At any given time, **Transceiver** takes the last sent value for a parameter and ignore those same settings on other RCUs.

Figure 45, "GenericControl timing diagram" below illustrates **GenericControl** changing the frequency of **Transceiver** in time:

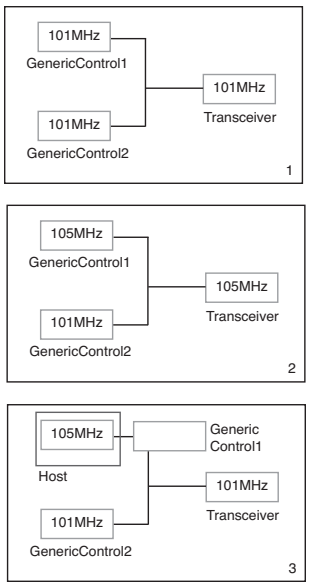


Figure 45: GenericControl timing diagram

Table 267, "GenericControl control inputs" on the facing page lists and describes **GenericControl** control input variables:

Name	Type	Default Value	Description
CryptoKey	**uint16**	**0**	If two radios are using encryption, then they must have matching crypto keys for the crypto modes. This variable must match for proper radio tuning.
Fill	**fill**	**<Select>**	Insert the fill created in the Comm Plan by double-clicking **Value**.
FreqHopNetID	**uint16**	**0**	Identifies the frequency-hopping network.
FreqHopSyncTOD	**uint32**	**0**	Identifies the time of day used in frequency hopping.
FreqHopTranSecKey	**uint16**	**0**	Identifies the transmission security key used in generating hopping patterns.
FreqHop HopSetWOD	**uint16**	**0**	Identifies the set of frequencies used in hopping pattern.
FreqHopLockOutId	**uint16**	**0**	Identifies the set of frequencies that are excluded from the hopping pattern.
FreqHopSystem	**uint16**	**0**	Sets the system type to enable frequency hopping, such as HAVEQUICK or SINCGARS. When the value is 0, hopping is disabled.
Frequency	**uint64**	**0**	The current radio tune frequency in Hertz.
Net	**uint32**	**1**	Net defines the core radio features, including frequency, Tx frequency, waveform, crypto, and frequency hopping. *Net* parameters are set in the **commplan**.
TransceiverId	**id**	**UNASSIGNED**	*TransceiverID* tells the RCU which **Transceiver** it should control.
TxFrequency	**uint64**	**0**	The transmit frequency in Hertz. For use only if the transmit frequency differs from the receive frequency.

Name	Type	Default Value	Description
TxPower	**float32**	**0.0**	The transmit power for the radio in Watts.

Table 267: GenericControl control inputs

15.3 HfServer

Summary: **HfServer** allows control of the **HfServer** application, providing real-time, high-fidelity modeling of HF radios.

Description: **HfServer** computes propagation effects between virtual radios, taking into account such things as transmitter-receiver global position, season, time of day (i.e., day-night terminator), and solar activity. To properly simulate solar activity and seasonal and circadian effects on the ionosphere and high frequency (HF) radio signal propagation, **HfServer** requires that the *Smoothed Sunspot Number (SSN)* and *Time-of-Day* offset be set. These parameters are set through use of **HfServer**.

***Note**: Set configuration parameters related to the **HfServer's** network behavior, (e.g., the interface and User Datagram Protocol (UDP) port number to send or receive HF requests) via the Distributed Interactive Simulation (DIS) gateway configuration in the project.*

Use *Offset* to force **HfServer** to return results for a night mission, even though the exercise takes place during the day. Likewise, use seasonal effects typical of winter propagation in July.

For example, if you're conducting an exercise on the East Coast of the US during summer (GMT +5) at 8:00 a.m. (08:00 Eastern Standard Time), **HfServer's** local clock reads 13:00 GMT. If the simulated scenario takes place on the West Coast of the US, during summer (GMT +8) at 3:30 p.m. (15:30 Pacific Standard Time), then a clock in the simulated world reads 23:30 GMT. The difference between the simulated GMT (23:30) and the real-world GMT (13:00) is the time-of-day *Offset* (i.e., +10.5 hours).

Table 268, "HfServer control inputs" below lists and describes **HfServer** control input variables:

Name	Type	Default Value	Description
SSN	**uint32**	**100**	Smoothed Sunspot Number. Typical values of the *SSN* range from 0–250, depending on past and current sunspot activity.
Offset	**int32**	**0**	Time-of-day offset in seconds between the **HfServer** clock (local time in GMT) and the simulation time (in GMT).
DomainNameIn	**string**	N/A	Lets a host platform set the domain name string remotely. This input typically works in conjunction with **NumToString**. Go to the *DomainName* string definition for syntax.

Table 268: HfServer control inputs

Table 269, "HfServer internal parameter" below describes the **HfServer** internal parameter variable:

Name	Type	Default Value	Description
DomainName	**string**	**<Edit>**	(Required) Part of the ASTiNet property set that defines a common communication environment. Radios in the same domain can communicate, while radios in separate domains cannot. It's analogous to a DIS exercise ID or an HLA Federation name. As an ASCII string, **Transceiver** is a native ASTiNet radio. When that string matches a defined DIS domain in a project, it maps the domain name to a DIS exercise ID. To define a DIS exercise ID , enter “DIS:*N*”, where *N*=1–255. For example, “DIS:1” places the radio in DIS exercise ID #1. ***Important**: DomainName specifies which HF configuration is valid. Use this variable to control multiple domains with multiple **HfServers**.*

Table 269: HfServer internal parameter

15.4 IntercomTransceiver

Summary: **IntercomTransceiver** is a simple version of **Transceiver** that works with **Intercom Control Unit** (**ICU**) to create a network intercom.

Description: **IntercomTransceiver** connects to an **ICU** to create a network intercom that broadcasts audio over the Distributed Interactive Simulation (DIS) network. Unlike the **Transceiver**, the **IntercomTransceiver** operates as an intercom only. It does not have the full radio features.

Table 270, "IntercomTransceiver control inputs" below lists and describes **IntercomTransceiver** control input variables:

Name	Type	Default Value	Description
DomainNameIn	**string**	N/A	Allows a host platform to set the domain name string remotely. This input is typically used in conjunction with **NumToString**.
LocalPTT	**Boolean**	**FALSE**	When **FALSE**, the radio transmits when it receives active audio and stops transmitting when the audio stops. When **TRUE**, the radio transmits persistently, even if it does not receive active audio.
PowerIn	**Boolean**	**TRUE**	Enables or disables power for **IntercomTransceiver**. If disconnected, the default value is **TRUE**.
ProtocolIdIn	**string**	N/A	Allows a host platform to set the *ProtocolIdIn* string remotely. This input is typically used in conjunction with **NumToString**.

Table 270: IntercomTransceiver control inputs

Table 271, "IntercomTransceiver control outputs" on the next page lists and describes **IntercomTransceiver** control output variables:

Name	Type	Default Value	Description
TxStatus			
TxAudio	**audio**	N/A	Audio input feed functioning Tx audio for **Transceiver**.
TxActive	**Boolean**	**FALSE**	Shows if **IntercomTransceiver** is transmitting.
RxStatus			
RxAudio	**audio**	N/A	Received audio fed to the intercom service.
RxActive	**Boolean**	**FALSE**	Shows if **IntercomTransceiver** is receiving.

Name	Type	Default Value	Description
LocalAudio			
RxTuneTone	**audio**	N/A	Audio input that functions as audio received from **Transceiver's** receiver stage.

Table 271: IntercomTransceiver control outputs

Table 272, "IntercomTransceiver internal parameters" below describes **IntercomTransceiver** internal parameters:

Name	Type	Default Value	Description
RCUParameters			
NetName	**string**	**<Edit>**	Net name of **RCUbasic**.
Audio			
encoding-rate	**uint32**	**8000**	Encoding rate of **RCUbasic**.
encoding-type	**EncodingType**	**MuLaw**	Encoding type of **RCUbasic**.
TxStatus			
TxChannel	**uint64**	**0**	Reflects the transmit channel set in the ICU.
TxPower	**float32**	**1.0**	Not used.

Table 272: IntercomTransceiver internal parameters

Table 273, "Other IntercomTransceiver internal parameters" on page 230 lists and describes other **IntercomTransceiver** internal parameter variables:

Name	Type	Default Value	Description
DomainName	**string**	**<Edit>**	Part of the ASTiNet property set and defines a common communication environment. All **Transceivers** in the same domain have the ability to communicate, and **Transceivers** in separate domains can never communicate. It is analogous to a DIS exercise ID or a high-level architecture (HLA) Federation name. As an ASCII string, **Transceiver** is an ASTiNet Radio. When that string matches a defined DIS domain as the project level, it maps the domain name to a DIS exercise ID. As a shortcut, define a DIS exercise ID by entering **DIS:***N*, where *N* = 1–255. For example, **DIS:1** puts the **Transceiver** in DIS exercise ID #1.
IntercomBus	**id**	**UNASSIGNED**	Assigns **IntercomTransceiver** to **IntercomBusService**. **IntercomBusService** allows for input audio, sidetone audio, and output audio to be passed around among components that are connected to the bus.
IntercomName	**string**	**<Edit>**	Set an identification name for the **IntercomTransceiver**.
Mode	**TunerMode**	**None**	Displays whether **IntercomTransceiver** is in Voice over Internet Protocol (VoIP) or intercom mode. Radio modes are not supported.
NetName	**string**	**<Edit>**	Indicates the set net name.
PowerBus	**id**	**UNASSIGNED**	Connects to *PowerService* to receive power instead of using a control.
PowerState	**Boolean**	**TRUE**	Displays the power state of **IntercomTransceiver**.

Name	Type	Default Value	Description
ProtocolID	**string**	**<Edit>**	Turns a radio into a DIS radio by setting the DIS identifiers (e.g., Host ID, Radio ID). *ProtocolID* sets **Marking Field**, labeling the radio “local” and unpublished on the network. • *DIS:#.#.#.#*: sets the site, application, entity, and radio IDs. • *DIS:#.#*: sets the entity and radio IDs. • *DIS:#*: sets the radio ID. • *DIS*: automatically sets all four IDs. • *LCL*: sets the radio as “local” and unpublished on the network. If any are blank, the variable automatically generates remaining identifiers. It sets the site and application ID via the IP address's last two octets. It also automatically generates the entity and radio IDs. You can append the above examples with another colon and the DIS marking field. For example, DIS:100.3:RedForce1 sets the **Entity ID** to **100**, the **Radio ID** to **3**, and the **Marking Field** to **RedForce1**.
RxChannel	**uint64**	**0**	Reflects the receive channel set in the ICU.
TransceiverID	**id**	**UNASSIGNED**	Defines the connection between the **IntercomTransceiver** and the ICU.

Table 273: Other IntercomTransceiver internal parameters

15.5 ICU

Summary: **ICU** works with **IntercomTransceiver** to create a network intercom.

Description: The network intercom provides a simple method of running networked communications and is the easiest way to connect two operators over a network. The ICU component is a control unit for **Transceivers** and removes most of the radio functionality such as bandwidth or propagation. Two intercoms must be in INTERCOM mode per the net and fill and have a matching channel, in the same way two radios must have a matching frequency.

The **ICU** helps to extend intercom capability beyond local operators and goes across a local area network (LAN) or wide area network (WAN) allowing multiple Telestra servers to communicate with each other when full fidelity radios are not required or desired. The **ICU** also uses fewer credits than a network radio.

***Note**: The **ICU** component can be used with the **Transceiver**, but that is not a recommended configuration.*

Table 274, "ICU control inputs" below lists and describes **ICU** control input variables:

Name	Type	Default Value	Description
Fill	**fill**	**<Select>**	Insert the fill created in the **Comm Plan** by double-clicking **Value**.
Net	**uint32**	**1**	Set in the **Comm Plan**. Set all nets to **Intercom** mode or use the default **Clearcom** net.
TransceiverID	**id**	**UNASSIGNED**	*TransceiverID* tells the ICU which **IntercomTransceiver** it should control.
Channel	**uint64**	**0**	Defines the **ICU** channel number. In order to receive over the network, both channel numbers must be equal. Valid values are 0–99,999.

Table 274: ICU control inputs

15.6 MarkerTone

Summary: **MarkerTone** add navigational beacons to a model.

Description: **MarkerTone** adds the Outer Marker, Middle Marker, Inner Marker, and Fan Marker navigational beacons to a model. The default values are set to FAA standards. While the component provides four separate beacons, the component can only consider one of the beacons at a time. An input control selects which beacon to use.

Table 275, "MarkerTone audio outputs" below describes the **MarkerTone** audio output variable:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	*OutSignal* is the audio output signal from **MarkerTone**.

Table 275: MarkerTone audio outputs

Table 276, "MarkerTone control inputs" on page 234 describes **MarkerTone** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**TRUE**	When **TRUE**, **MarkerTone** is enabled.
Frequency1	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is **1**. The default is 400 Hz for the Outer Marker. • *Modifier:* **400.0**
Frequency2	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is **2**. The default 1300 Hz for the Middle Marker. • *Modifier:* **1300.0**
Frequency3	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is set to **3**. The default 3000 Hz for the Inner Marker. • *Modifier:* **3000.0**
Frequency4	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is set to **4**. The default is 3000 Hz for the Fan Marker. • *Modifier:* **3000.0**
Frequency5	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is **5**. • *Modifier:* **0.0**
Frequency6	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is **6**. • *Modifier:* **0.0**
Frequency7	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is **7**. • *Modifier:* **0.0**
Frequency8	**float32**	**1.0**	Sets the pitch for the keyed tone when *IdentIndex* is **8**. • *Modifier:* **0.0**
Gain	**float32**	**1.0**	Gain applies a linear volume control to the **MarkerTone** audio output.
Interval1	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **1**. Units are in seconds.
Interval2	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **2**. Units are in seconds.

Name	Type	Default Value	Description
Interval3	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **3**. Units are in seconds.
Interval4	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **4**. Units are in seconds.
Interval5	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **5**. Units are in seconds.
Interval6	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **6**. Units are in seconds.
Interval7	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **7**. Units are in seconds.
Interval8	**uint16**	**0.0**	Sets the delay between repeating the keyed word when *IdentIndex* is **8**. Units are in seconds.
IdentIndex	**uint8**	**1.0**	Selects which marker beacon (*IdentChar*) is transmitted. A value of **0** selects none. A value of **1** selects *Ident1* (i.e., Outer Marker). A value of **2** selects *Ident2* (i.e., Middle Marker). A value of **3** selects *Ident3* (i.e., Inner Marker). A value of **4** selects *Ident4* (i.e., Fan Marker).
StrictTiming	**Boolean**	**TRUE**	When **TRUE**, **MarkerTone** uses proper **MarkerTone** intercharacter timing, which is a one-dot gap between **Ident** characters. When **FALSE**, **MarkerTone** uses standard Morse code intercharacter timing, which is a three-dot gap between **Ident** characters.
Wordrate1	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **1**. • *Modifier:* **11**
Wordrate2	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **2**. • *Modifier:* **10**
Wordrate3	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **3**. • *Modifier:* **18**
Wordrate4	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **4**. • *Modifier:* **8**

Name	Type	Default Value	Description
Wordrate5	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **5**. • *Modifier:* **11**
Wordrate6	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **6**. • *Modifier:* **10**
Wordrate7	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **7**. • *Modifier:* **18**
Wordrate8	**uint8**	**1**	Determines the rate at which the word is keyed when *IdentIndex* is **8**.

Table 276: MarkerTone control inputs

Table 277, "MarkerTone control outputs" below lists and describes **MarkerTone** control output variables:

Name	Type	Default Value	Description
Active	**Boolean**	**FALSE**	When keyer is keying a dot or a dash, this goes **TRUE**, essentially mimicking the beep of the keyer.
Busy	**Boolean**	**FALSE**	When *OutSignal* is active audio, *Busy* is **TRUE**.

Table 277: MarkerTone control outputs

Table 278, "MarkerTone internal parameters" on the facing page lists and describes **MarkerTone** internal parameter variables:

Name	Type	Default Value	Description
Ident1	**ident**	**T**	The identifier the keyer plays when *IdentIndex* is **1**. A value of **T** plays a single dash in Morse code.
Ident2	**ident**	**A**	The identifier the keyer plays when *IdentIndex* is **2**. A value of **A** plays a dot dash in Morse code.
Ident3	**ident**	**E**	The identifier the keyer plays when *IdentIndex* is to **3**. A value of **E** plays a single dot in Morse code.
Ident4	**ident**	**I**	The identifier the keyer plays when *IdentIndex* is **4**. A value of **I** plays a dot dot in Morse code.

Name	Type	Default Value	Description
Ident5	**ident**	**<Edit>**	The identifier the keyer plays when *IdentIndex* is **5**; useful if you want to create your own marker.
Ident6	**ident**	**<Edit>**	The identifier the keyer plays when *IdentIndex* is **6**; useful if you want to create your own marker.
Ident7	**ident**	**<Edit>**	The identifier the keyer plays when *IdentIndex* is **7**; useful if you want to create your own marker.
Ident8	**ident**	**<Edit>**	The identifier the keyer plays when *IdentIndex* is **8**; useful if you want to create your own marker.

Table 278: MarkerTone internal parameters

15.7 MorseKeyer

Summary: **MorseKeyer** creates a four-letter Morse code sequence.

Description: **MorseKeyer** translates four-letter words into Morse code. The component controls the word rate, interval, and the frequency (i.e., pitch) of the generated tone. In addition to the usual letters and numbers defined in Morse code, it also includes the characters * and - to represent individual dot and dash combinations.

Table 279, "MorseKeyer audio output" below lists and describes the **MorseKeyer** audio output variable:

Name	Type	Default Value	Description
OutSignal	**audio**	N/A	Outputs keyed word when press-to-talk (PTT) control is enabled.

Table 279: MorseKeyer audio output

Table 280, "MorseKeyer control inputs" on the next page lists and describes **MorseKeyer** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**TRUE**	When **TRUE**, it enables the **MorseKeyer**. When **FALSE**, the **MorseKeyer** is disabled.
Frequency	**float32**	**1.0**	Provides the frequency (i.e., pitch) of the Morse tone. Units are in Hertz.
Gain	**float32**	**1.0**	Scales the strength of the generated tone.

Name	Type	Default Value	Description
Ident	**ident**	N/A	The four-letter word converted into Morse code. *Ident* is a special variable type defined as four combined ASCII characters. **HostIn** drives *Ident*.
Interval	**float32**	**1.0**	Total cycle in seconds before the **Ident** message starts playing, including pause. To calculate this number, use the following formula: *Word Rate Time* + (*Wordrate* - *Interval)* The interval number must be larger than the word rate.
ILSPause	**Boolean**	**FALSE**	When **FALSE**, **MorseKeyer** uses normal Morse code inter-character timing, which is a three-dot gap between all **Ident** characters. When **TRUE**, **MorseKeyer** uses standard Morse code interword timing between first and second characters (i.e., a seven-dot gap). Inter-character timing for subsequent **Ident** characters use standard Morse code inter-character spacing (i.e., a three-dot gap).
Wordrate	**uint8**	**1**	Determines rate at which word is keyed.

Table 280: MorseKeyer control inputs

Table 281, "MorseKeyer control outputs" below lists and describes **MorseKeyer** control output variables:

Name	Type	Default Value	Description
Active	**Boolean**	**FALSE**	**TRUE** when the keyer is keying dot or dash, mimicking the keyer's beep.
Busy	**Boolean**	**FALSE**	Indicates if **MorseKeyer** actively produces audio.

Table 281: MorseKeyer control outputs

15.8 RCUbasic

Summary: **RCUbasic** pairs with **Transceiver** to model radios for the radio simulated environment.

Description: The most commonly set radio uses two components:

- **Transceiver**
- **RCUbasic**

The Radio Control Unit (RCU) basic requires a **commplan** fill to access radio modes, waveforms, default settings, etc. If the control inputs are set from an external source (e.g., host control), they override the default values in the fill. For example, **RCUbasic** has a loaded fill and is using Net 1, which defaults the frequency of the radio to 101 megahertz (MHz). This information is sent to the paired **Transceiver** that shares the same Transceiver ID as **RCUbasic**. For example, to use the net default with a new frequency of 105 MHz, set the RCUbasic component's frequency to 105 MHz. The new frequency overrides the net default, and **Transceiver** sets its frequency to 105 MHz.

In general, settings that relate to how the radio appears on the network are in **Transceiver**, and settings that controls the radio for simulation are placed in **RCUbasic**.

Table 282, "RCUbasic control inputs" on the next page lists and describes **RCUbasic** control input variables:

Name	Type	Default Value	Description
CryptoKey	**uint16**	**0**	If two radios are encrypted, they must have matching crypto keys to communicate.
CryptoSys	**uint16**	**0**	Sets the radio's crypto type. Acceptable values include the following: • *0:* other • *1:* KY-28 • *2:* KY-58 • *3:* NSVE • *4:* WSVE • *5:* SINCGARS ICOM
CodecType	**uint16**	**0**	When the radio mode is DIGITAL, this number must match the receiver to transmitter.
Frequency	**uint64**	**0**	Current radio tune frequency in Hertz.
FreqHopHopSetWOD	**uint16**	**0**	Identifies the set of frequencies used in hopping pattern. Default of 0 = match all.
FreqHopLockOutId	**uint16**	**0**	Identifies the set of frequencies excluded from hopping pattern. Default of 0 = match all.
FreqHopNetId	**uint16**	**0**	Identifies the frequency-hopping net ID. Frequency hopping radio must have non-zero Net ID to be considered actively hopping.
FreqHopSyncTOD	**uint32**	**0**	Identifies the frequency-hopping time of day. A default of 0 = match all.
FreqHopSystem	**uint16**	**0**	Not used. ASTi recommends setting of 0.

Name	Type	Default Value	Description
FreqHopTranSecKey	**uint16**	**0**	Identifies the frequency-hopping transmission security key that generates hopping patterns. Default of 0 = match all.
NetIndicate	**uint32**	**1**	Selects net from the **commplan** fill. Net defines core radio parameters, including frequency, Tx frequency, waveform, crypto, and frequency hopping. Set net parameters in the **commplan**.
TxFrequency	**uint64**	**0**	**Transmitter** frequency of **Transceiver** in Hz. Only use if Tx frequency differs from Rx frequency.
TxPower	**float32**	**0.0**	Transmit power for radio in Watts.

Table 282: RCUbasic control inputs

Table 283, "RCUbasic control outputs" below lists and describes **RCUbasic** control output variables:

Name	Type	Default Value	Description
FillName	**string**	**<Edit>**	Reports fill contents. Typically not connected to outputs.
NetIndicate	**uint32**	**0**	Reports *TransceiverId* contents. Typically not connected to output.

Table 283: RCUbasic control outputs

Table 284, "RCUbasic internal parameters" below lists and describes **RCUbasic** internal parameter variables:

Name	Type	Default Value	Description
Fill	**fill**	**<Select>**	IDs the **commplan** fill to use with **Transceiver**. To insert the **commplan** fill, double-click **Value**. A radio needs a fill to operate.
TransceiverId	**id**	**UNASSIGNED**	Tells **RCUbasic** which **Transceiver** to control. The **Transceiver's** *TransceiverId* must match.

Table 284: RCUbasic internal parameters

15.9 Receiver

Summary: **Receiver** communicates with **Transceiver** to force receive capability.

Description: **Receiver** is an RCU in the **Radio** group. It communicates with **Transceiver** to force Rx capability only. It's a subset of the features and controls in **RCUbasic**.

Receiver charges fewer credits than **RCUbasic** and other Radio Control Units (RCUs), but it also provides less functionality. The component replicates guard receivers and navigational aids, such as an automatic direction finder (ADF) receiver or a Tactical Air Navigation (TACAN) receiver. For more information about RCUs and **Transceiver**, go to Section 15.8, "RCUbasic" on page 236.

Table 285, "Receiver control inputs" below describes **Receiver** control input variables:

Name	Type	Default Value	Description
CryptoKey	**uint16**	**0**	If two radios use encryption, they must have matching crypto keys for the crypto modes; must match for proper radio tuning.
Frequency	**uint64**	**0**	The current radio tune frequency in Hz.
FreqHopNetId	**uint16**	**0**	Identifies the frequency-hopping network.
FreqHopSyncTOD	**uint32**	**0**	Identifies the time of day used in frequency hopping.
FreqHopTranSecKey	**uint16**	**0**	Identifies the transmission security key used to generate hopping patterns.
FreqHopHopSetWOD	**uint16**	**0**	Identifies the set of frequencies used to generate hopping patterns.
FreqHopLockOutId	**uint16**	**0**	Identifies the set of frequencies that are excluded from hopping pattern.
Net	**uint32**	**1**	Net defines the core radio features, including frequency, Tx frequency, waveform, crypto, and frequency hopping. *Net* parameters are set in the **Comm Plan**.
CryptoSys	**uint16**	**0**	Sets the radio's crypto type. Acceptable values include the following: • *0:* other • *1:* KY-28 • *2:* KY-58 • *3:* NSVE • *4:* WSVE • *5:* SINCGARS ICOM

Table 285: Receiver control inputs

Table 286, "Receiver internal parameters" below lists and describes **Receiver** internal parameter variables:

Name	Type	Default Value	Description
Fill	**fill**	**<SELECT>**	To insert the **commplan** fill, double-click in **Value**.
TransceiverId	**id**	**UNASSIGNED**	*TransceiverId* tells the RCU which **Transceiver** it should control.

Table 286: Receiver internal parameters

15.10 Relay

Summary: **Relay** links any radio pairs from a bank of eight radios.

Description: **Relay** connects up to eight **Transceivers** together to form radio relays. A radio relay takes all received audio from a radio and retransmits from another radio. **Relay** allows you to make up to four pairs of relays across eight radios. In the most basic case, link the received audio from a transceiver to *ReceiveAudio1* of **Relay**. *RadioSelector1* sets which radio receives the audio, where 2 = *TransmitAudio2, 3* = *TransmitAudio3*, etc.

In the example below, *Transceiver2* and *Transceiver3* are a relay pair, and *Transceiver1* and *Transceiver4* are also a pair. As in the real world, radios in **Relay** mode have different frequencies.

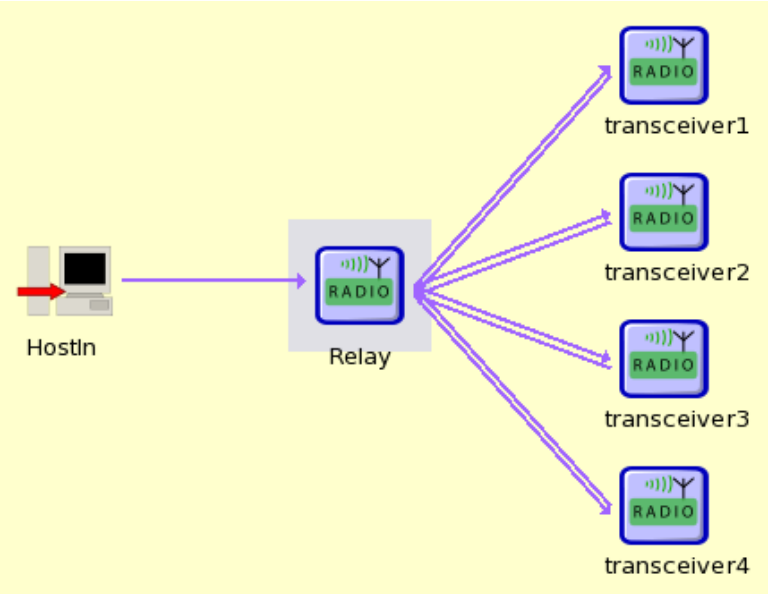


Figure 46: Basic Relay model example

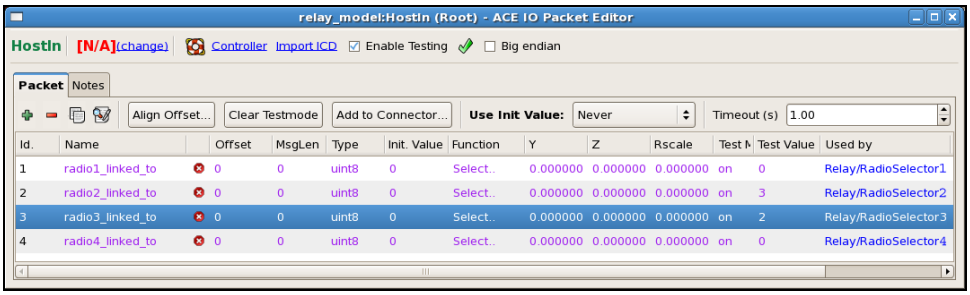


Figure 47: Relay host control

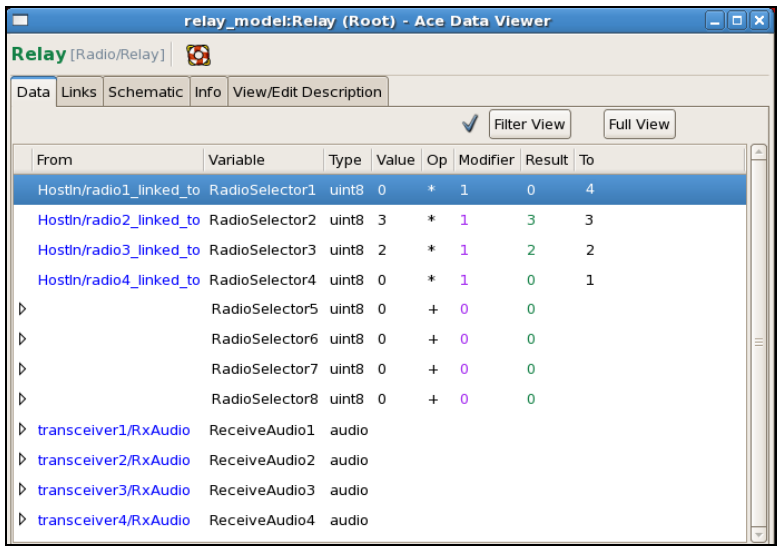


Figure 48: Relay component

Table 287, "Relay audio inputs and outputs" below lists and describes **Relay** audio input and output variables:

Name	Type	Default Value	Description
Audio Inputs			
ReceiveAudio1–ReceiveAudio8	**audio**	N/A	The *ReceiveAudio* from the selected **Transceiver** routed into **Relay**.
Audio Outputs			
TransmitAudio 1–TransmitAudio8	**audio**	N/A	A link to the transmit audio input of the selected **Transceiver**. Audio is routed from the *ReceiveAudio* lines based on the *RadioSelectors*.

Table 287: Relay audio inputs and outputs

Table 288, "Relay control inputs" on the next page lists and describes **Relay** control input variables:

Name	Type	Default Value	Description
RadioSelector1	**uint8**	**0**	*RadioSelector1* chooses which transmit audio stream *ReceiveAudio1* routes to. A value of **0** means *ReceiveAudio1* does not relay to any other radio.
RadioSelector2	**uint8**	**0**	*RadioSelector2* chooses which transmit audio stream *ReceiveAudio2* routes to. A value of **0** means *ReceiveAudio2* is not relayed to any other radio.
RadioSelector3	**uint8**	**0**	*RadioSelector3* chooses which transmit audio stream *ReceiveAudio3* routes to. A value of **0** means *ReceiveAudio3* does not relay to any other radio.
RadioSelector4	**uint8**	**0**	*RadioSelector4* chooses which transmit audio stream *ReceiveAudio4* routes to. A value of **0** means *ReceiveAudio4* does not relay to any other radio.
RadioSelector5	**uint8**	**0**	*RadioSelector5* chooses which transmit audio stream *ReceiveAudio5* routes to. A value of **0** means *ReceiveAudio5* does not relay to any other radio.
RadioSelector6	**uint8**	**0**	*RadioSelector6* chooses which transmit audio stream *ReceiveAudio6* routes to. A value of **0** means *ReceiveAudio6* does not relay to any other radio.
RadioSelector7	**uint8**	**0**	*RadioSelector7* chooses which transmit audio stream *ReceiveAudio7* routes to. A value of **0** means *ReceiveAudio7* does not relay to any other radio.
RadioSelector8	**uint8**	**0**	*RadioSelector8* chooses which transmit audio stream *ReceiveAudio8* routes to. A value of **0** means *ReceiveAudio8* does not relay to any other radio.

Table 288: Relay control inputs

15.11 Satellite

Summary: **Satellite** is a simulated satellite that handles uplink/downlink relay of signals transmitted from a simulated radio in SATCOM mode, including simulation of mode-dependent delays.

Description: **Satellite** represents a simulated satellite and can be set to relay uplink signals from radios set in SATCOM mode. Realistic simulation of **Satellite** parameters, including delays based on SATCOM sub-modes, world position, uplink and downlink bands are included. Add multiple **Satellites** to a model to simulate multiple, independent satellites.

SATCOM submodes include the following:

- 5k Dedicated
- 5k DASA
- 5k DAMA
- 25k Dedicated
- 25k DC DASA
- 25k AC DAMA
- 25k DC DAMA

Satellite can also include propagation effects, such as smooth Earth occulting and terrain occulting through use of a terrain server:

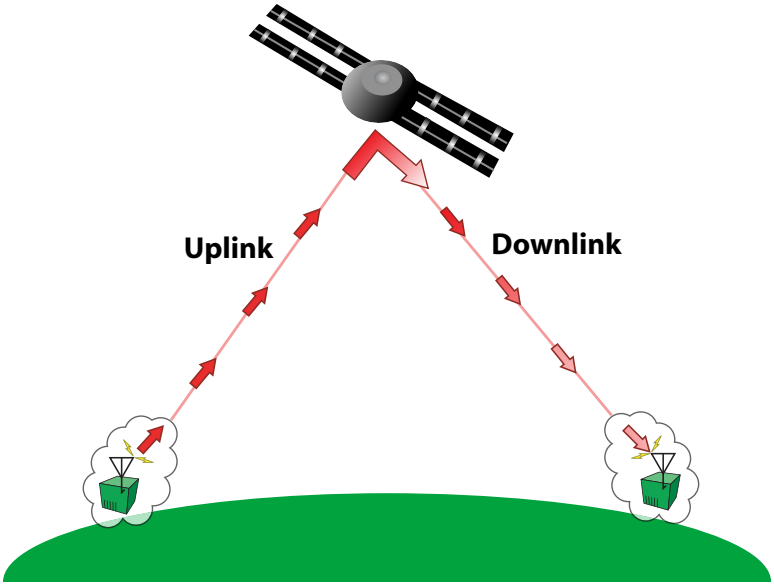


Table 289, "Satellite control inputs" on page 245 lists and describes **Satellite** control input variables:

Name	Type	Default Value	Description
UplinkFrequency	**uint64**	**0**	Base frequency of **Satellite** uplink frequency band in Hertz.
DownlinkFrequency	**uint64**	**0**	Base frequency of **Satellite** downlink frequency band in Hertz.
Passband	**uint64**	**0**	Defines the frequency range of **Satellite** uplink and downlink bands in hertz.
TxPower	**float32**	**1.0**	The transmit power for **Satellite** downlink transmitter in Watts.
Channels	**uint32**	**1**	Number of simultaneous channels that can be supported by **Satellite**.
FixedDelay	**uint32**	**0**	Overrides mode-based delays and forces a fixed delay value. Specified in milliseconds.
Delay5k	**uint32**	**0**	Delay applied between when **Satellite** receives an uplink signal, and it transmits the downlink signal when in *SATCOM5k* mode. Specified in milliseconds.
Delay5kDASA	**uint32**	**0**	Delay applied between when **Satellite** receives an uplink signal and when it transmits the downlink signal when in *SATCOM5kDASA* mode. Specified in milliseconds.
Delay5kDAMA	**uint32**	**0**	Delay applied from when **Satellite** receives an uplink signal and when it transmits the downlink signal when in *SATCOM5kDAMA* mode. Specified in milliseconds.
Delay25k	**uint32**	**0**	Delay applied from when **Satellite** receives an uplink signal and when it transmits the downlink signal when in *SATCOM25k* mode. Specified in milliseconds.
Delay25kDCDASA	**uint32**	**0**	Delay applied from when **Satellite** receives an uplink signal and when it transmits the downlink signal when in *SATCOM25kDCDASA* mode. Specified in milliseconds.

Name	Type	Default Value	Description
Delay25kACDAMA	**uint32**	**0**	Delay applied from when the **Satellite** receives an uplink signal and when it transmits the downlink signal when in *SATCOM25kACDAMA* mode. Specified in milliseconds.
Delay25kDCDAMA	**uint32**	**0**	Delay applied from when **Satellite** receives an uplink signal and when it transmits the downlink signal when in *SATCOM25kDCDAMA* mode. Specified in milliseconds.
DomainNameIn	**string**	N/A	Allows a host platform to set the *DomainName* string remotely. This input is typically used in conjunction with the **Control** > **NumToString**.
ProtocolIdIn	**string**	N/A	Allows a host platform to set the *ProtocolId* string remotely. This input is typically used in conjunction with **NumToString**.
ForceCenter OfEarth	**Boolean**	**FALSE**	A manual override that places **Satellite** at 0, 0, 0 geocentric world position. This variable disables all propagation effects on transmit or receive.
TerrainEnable	**Boolean**	**FALSE**	When **TRUE**, allows input terrain server to process line-of-sight transmissions.

Table 289: Satellite control inputs

Table 290, "Satellite internal parameters" on page 247 lists and describes **Satellite** internal parameter variables:

Name	Type	Default Value	Description
SatelliteName	**string**	**<Edit>**	Name of the component.

Name	Type	Default Value	Description
DomainName	**string**	**<Edit>**	The *DomainName*is part of the ASTiNet property set and defines a common communications environment. All radios in the same domain can communicate. Radios in separate domains can never communicate. It is analogous to a Distributed Interactive Simulation (DIS) exercise ID or a high-level architecture (HLA) Federation name. When defined as an ASCII string **Transceiver** is an ASTiNet radio. When that string matches a defined DIS domain at the project level, it maps the domain name to a DIS exercise ID. To define a DIS exercise ID, enter **DIS:***N*, where *N*=1–255. For example, “DIS:1” puts the radio in DIS exercise ID #1. In **Satellite**, this action specifies the domain where the satellite are active. Leaving this variable blank causes it to act as a wild card. **Satellite** is active in all domains.
ProtocolID	**string**	**<Edit>**	Optional configuration for when you have a non-ASTiNet radios. DIS is currently the only supported protocol. The syntax for setting up a DIS radio is the string format of DIS:*site.application.entity.radio*, DIS:*entity.radio*, or DIS:*radio*. If the site and application are excluded from the string, the radio environment assigns the default site and app from the DIS file in **Domain Editor**. If entity is excluded, the radio environment assigns a random/unique number entity ID for the radio. This variable can also be set in the radio helper. It must be unique within the DIS network. For example: DIS:100.100.1.1 sets: Site=100, App=100, Entity=1, and Radio=1 In **Satellite**, *ProtocolID* affects the DIS ID of the temporary transmitters setup to retransmit the signal.

Name	Type	Default Value	Description
WorldPositionBus	**id**	**UNASSIGNED**	Optional configuration that allows you to assign a **Satellite** to a world position bus. If undefined, **Satellite** defaults to X,Y,Z = 0,0,0. Alternatively, if an entity exists on the DIS network that matches the *Site.Application*. *Entity ID* within **Satellite** *ProtocolID*, **Satellite** inherits that entity's world position information. When a bus name is defined, **Satellite** is located (X,Y,Z) based on **Platform**, which is tied to the same bus name used in conjunction with **Platform**/ **GeocentricWorldPosition** or **GeodeticWorldPosition**.

Table 290: Satellite internal parameters

15.12 Transceiver

Summary: **Transceiver** works in conjunction with **Radio Control Unit** (**RCU**) to create a simulated radio.

Description: **Transceiver** models radio transmission and radio reception. Used in conjunction with **RCU**, **Intercom Control Unit** (**ICU**), **Receiver**, or **Transmitter**, it forms the basis of a **Radio**, **Network Intercom**, **Receiver**, or **Transmitter** respectively.

Some of the core features of the **Transceiver** include the following:

- Host control of core radio parameters
- Modulation matching, which includes amplitude modulation (AM), frequency modulation (FM), upper sideband (USB), lower sideband (LSB), etc.
- Radio Frequency (RF) propagation modeling based on world position, frequency, etc.
- Crypto state and sound modeling
- Frequency-hopping modeling
- Jamming support
- Squelch control
- Terrain support, in conjunction with an external terrain server
- Antenna Gain, Cable Loss, RXTuneTone and other advanced radio parameters
- Audio and Tactical Data Link support
- Multiprotocol support including local, ASTiNet, Distributed Interactive Simulation (DIS) and high-level architecture (HLA)

Transceiver provides a generic, high-level radio simulation, which includes transmit and receive operations, frequency-tuning effects, AM and FM modulation modes, signal-strength variation due to range, transmit power, antenna and receiver gain, RF and internal noise, and propagation path loss, sidetone signal return, and support for crypto and frequency hop behaviors, and more.

At the simplest level, **Transceiver** provides a simulation of the interface between the signal and data flow within a radio, and the simulated radio frequency environment. Just as in the real world, the signals passed between a real **Transceiver** and the remainder of the radio subsystem provide information to and from **Transceiver**, and include the transmit and receive audio signals (i.e., voice or tones), transmit and receive data message signals, including the following:

- Link-16
- Interval data messages (IDM)
- Aircraft Communications Addressing and Reporting System (ACARS)
- Control signals that determine the behavior of the **Transceiver** (e.g., tuned frequency, modulation mode, bandwidth)

Transceiver has two bidirectional information interfaces and a single, one-way control interface. One of the bidirectional information interfaces operates internally within the simulated radio and/or vehicle base-band. The other bidirectional interface operates in the RF bands and provides wireless transmission and reception. The control interface determines the translation between the base-band and RF environments.

Transceiver must be used with an **RCU** object to form an operational radio **RCU** provides the greater portion of the control interface data values. Applicable **RCU** subclass objects that may be used with the **Transceiver** include the following components:

- **RCU**
- **ICU**
- **Transmitter**
- **Receiver**

The type of **RCU** determines the available **Transceiver's** capabilities. For example, selection of a receiver **RCU** object limits the **Transceiver's** operation to the reception of RF-originated signals.

The **Transceiver** can implement various fidelity levels of RF modeling from simple frequency matching to full-fidelity simulation of specific radio types, including propagation and ranging, bandwidth overlap, antenna gain, etc. Ranging behavior requires that each radio has a world position describing the location of the radio on or above the Earth's surface. If a radio is set to operate using the DIS networking protocol, another option available is to attach a radio to an externally simulated entity using **Entity Attach**. Once attached, the **Transceiver** position is a slave to the selected entity, and all range calculations are based on the supplied entity position.

The received signal strength is computed for all in-tune radios based on the power of the transmitter, the antenna gains of **Transmitter** and **Receiver**, and the relative world positions. If the terrain interface is installed, the gain factor for the in-tune radios is factored in the calculator. If frequency hopping or encryption is enabled, **Transceiver** compares the parameters of the **Transmitter** and **Receiver** to see if the audio is received.

***Note**: In frequency-hopping mode, frequency is ignored. The frequency is implied in the selected Net ID hopset.*

If multiple **Transmitters** are broadcasting on the same frequency, **Transceiver** typically does one of two things. For AM signals, the received RF power is combined, and the received audio is a sum of the transmitted signals in proportion to their signal strength. For FM signals, only the strongest received signal is included.

Once the received power is determined, the RF signal-to-noise (SNR) is calculated. The noise level is determined by thermal noise, internal radio noise, and other parameters, which are set in **Transceiver**. The RF SNR is then compared to the squelch level. If the ratio is less than the squelch level, the signal is not received. Setting the squelch to zero disables the squelch.

After determining that the signal is received, the signal power and noise power are affected by automatic gain control (AGC). Additionally, when squelch is off, the maximum AGC determines the background noise when **Transceiver** doesn't receive a signal. The received audio is routed out of the component locally and/or onto the **IntercomBusService**, where an operator can hear it, typically through a communications panel component.

Transceiver can transmit as well as receive. When the radio transmits, the reception is cut off, assuming half-duplex operation.

To support crypto, simulation radios use a library of crypto tones (i.e., sound files) with each **Transceiver**. This library greatly simplifies the encrypted radio simulation by automatically playing tones (e.g., preamble, postamble, or mismatch tone) at the appropriate times during a secure radio transmission or reception.

Radio jamming occurs when a receiving radio operating in FM is blocked by a strong transmission, most often used a mismatched modulation mode (e.g., **Pulse**), causing any desired signal to be masked by the unwanted jammer. This capability is supported in **Transceiver**.

Receiver automatically determines if it is jammed. The audio associated with **Receiver** being jammed is implemented through **Transceiver** using the sound file library system to organize the jamming sounds. Each **Transceiver** can point at a particular jamming library and jamming group. The sound file indexes populated within the sound library or group map to possible modes of **Receiver**.

The receiving radio automatically chooses which sound file to play when it is jammed. The sound file (i.e., index) chosen is determined by the RECEIVER mode. Table 291, "Receiver mode sounds" below specifies which sound index is played based on the mode:

Receiver Mode	Sound File Index Played
FM	2
AM	3
SATCOM	4
CW	8
USB	9
LSB	10
Pulse	11
SSBF	13

Table 291: Receiver mode sounds

If the terrain interface is set, the radio environment determines in-tune **Transmitter** and **Receiver** pairs and generates data packets containing **Transmitter Receiver** world positions. The host computing system and a suitable terrain database may process these packets to determine accurate line-of-sight terrain obscuration checks in addition to the range calculations. Without the terrain package, ranging is limited by a calculation based upon a WGS-84 model of the Earth's curvature.

***Note**: For frequencies between 1 and 100,000, no background noise or signal attenuation effects are simulated. These frequencies provide a clear channel of communication, regardless of transmission power, world position, etc. The frequencies are typically reserved for network intercom communications (i.e., INTERCOM mode).*

***Note**: **Transceiver** defaults to a world position of geocentric X, Y, Z = 0, 0, 0 (i.e., the center of the Earth). At this position, a radio receives any radios using the same frequency without any signal loss or occulting. This feature can model a radio that monitors a particular radio band, without regard to position or transmit power.*

Table 292, "Transceiver audio inputs" below lists and describes **Transceiver** audio input variables:

Name	Type	Default Value	Description
ExternalNoise	**audio**	N/A	Attach an input signal to replace the internal **NoiseSource**. Overrides the white noise generator if a signal is attached and active.
RxTuneTone	**audio**	N/A	Local audio connection mixed into the receive audio path of the radio.
TxAudio	**audio**	N/A	Local audio connection to **Transceiver** transmit audio stream.

Table 292: Transceiver audio inputs

Table 293, "Transceiver audio outputs" below lists and describes **Transceiver** audio output variables:

Name	Type	Default Value	Description
RxAudio	**audio**	N/A	The receiving audio **Transceiver** picks up from in-tune transmitting radios).
TxCryptoAudio	**audio**	N/A	The audio the cryptotone player generates for transmit states only.

Table 293: Transceiver audio outputs

Table 294, "Transceiver control inputs" on page 254 lists and describes **Transceiver** control input variables:

Name	Type	Default Value	Description
AntennaGain	**float32**	**1.0**	The linear antenna gain applied to the radio receive signal.
Cableloss	**float32**	**1.0**	The loss factor representing the antenna cable; a value of 1.0 represents no loss.
BFOGain	**float32**	**1.0**	The gain associated with the beat frequency oscillator (BFO) audio.
BFOFrequency	**float32**	**0.0**	BFO frequency in **Transceiver**. The BFO has a tone strength that is proportional to the received carrier strength, generally used for detecting the Morse code keying present on a continuous wave beacon.
CryptoGroup	**playsound_group**	**1**	Allows for host control between different crypto groups in the sound library. For example, change from a KY-28 to a KY-58.
CryptoGain	**float32**	**1.0**	The gain level of the crypto sound.
CryptoEnable	**Boolean**	**TRUE**	Disables crypto even when crypto parameters are enabled in the **Transceiver's** associated **RCU**.
CryptoOnly	**Boolean**	**FALSE**	When **TRUE** and if the radio is in crypto mode (i.e., system and key are both not zero), the radio does not receive clear transmissions and or play *RX_Clear Playsound*. If **TRUE** and the radio is in CLEAR mode (i.e., system or key are 0), the radio receives clear transmissions.
DomainNameIn	**string**	N/A	Allows a host platform to set the *DomainName* string remotely. This input is typically used with **NumToString**.
ForceCenterOfEarth	**Boolean**	**FALSE**	A manual override places **Transceiver** at 0, 0, 0 geocentric world position. When **TRUE**, the radio is located at the center of the Earth (0, 0, 0). Set to the center of the Earth to disable propagation effects.

Name	Type	Default Value	Description
InterferenceLibrary	**playsound_library**	**<Select>**	The **Playsound** library with **Playsound** that plays when the radio senses a jammed. For more information about jamming, go to Section 15.12, "Transceiver" on page 247.
InterferenceGroup	**playsound_group**	**1**	The **Playsound** group with **Playsound** that plays when the radio senses a jam.
InterferenceGain	**float32**	**1.0**	The gain applied to jamming audio.
NoiseGain	**float32**	**0.1**	Gain associated with the internal **Transceiver** noise.
PowerIn	**Boolean**	**TRUE**	Enable or disable power for **Transceiver**. If not connected to **From**, the default value is **TRUE**.
ProtocolId	**string**	N/A	Allows a host platform to set the *ProtocolId* string remotely. This input is typically works with **NumToString**.
ReceiveGain	**float32**	**1.0**	The gain applied to the received audio. Equivalent to a volume knob.
RxALCEnable	**Boolean**	**FALSE**	Enables auto-level control (ALC) on **Transceiver's** received signal. Loud signals reduce in volume and quiet signals boost. Other ASTi-networked radios receive at a consistent volume, regardless of which system the radios are using.
RxDataThreshold	**float32**	**0.200**	Equivalent to a squelch level but for data reception.
RxEnable	**Boolean**	**TRUE**	If **TRUE**, then receive is enabled for the **Transceiver**.
SideFxEnable	**Boolean**	**TRUE**	If **TRUE**, then any applied voicing effects are heard in the sidetone returned from this **Transceiver**. Enable voicing effects in the **commplan** waveform.

Name	Type	Default Value	Description
SquelchLevel	**float32**	**0.200**	When the received signal-to-noise ratio (SNR) is less than the squelch value, the gain is 0, suppressing the background noise. To disable the squelch set the level to zero. To calculate the squelch level in dB, multiply the squelch result displayed in the component by 20. This is useful when you want to compare the squelch level to signal level in dB.
SquelchTail	**uint32**	**75**	The duration of the squelch tail in milliseconds.
TransmitAudioGain	**float32**	**1.0**	Gain applied to the Radio Bridge's transmit signal before it enters the radio environment.
TxALCEnable	**Boolean**	**FALSE**	Enables the ALC on the **Transceiver's** transmit signal. This behavior attempts to maintain a consistent and specified audio volume, reducing volume in loud signals and boosting the volume for quiet signals. As a result, other ASTi-networked radio systems hear this radio at a consistent volume.
TxEnable	**Boolean**	**TRUE**	If **TRUE**, then transmit is enabled for **Transceiver**.

Table 294: Transceiver control inputs

Table 294, "Transceiver control inputs" above lists and describes **Transceiver** control output variables:

Name	Type	Default Value	Description
Jammed	**Boolean**	**FALSE**	Indicates if **Transceiver** is jammed.
Range	**float32**	**0.0**	The distance to the **Transmitter** currently being received from in meters.
RxActive	**Boolean**	**FALSE**	Indicates if **Transceiver** is receiving.
RxAudioActive	**Boolean**	**FALSE**	When **TRUE**, the **Transceiver** receive output is non-zero. **TRUE** if squelch is broken or audio is received.
RxCrypt	**Boolean**	**FALSE**	**TRUE** when the radio is actively receiving a secure transmission.

Name	Type	Default Value	Description
RxCryptoSoundIdx	**uint16**	**0**	When no **Rx_Preamble** sound file is playing, this number is 0. When any **Rx_Preamble** is playing, this number reflects the sound index as configured in the sound library. This variable appears only in **Full View**.
RxDataDropped	**int32**	**0**	The number of data packets that the radio detected coming from an in-tune transmitting radio when *RxDataThresholdMet* is **FALSE** and is not forwarded to the local host.
RxDataForwarded	**int32**	**0**	The number of data packets successfully received from an in-tune, transmitting radio while *RxDataThresholdMet* is **TRUE** and is forwarded to the local host.
RxDataThresholdMet	**Boolean**	**FALSE**	**TRUE** when **Receiver** is in tune with a **Transmitter** and the SNR in dB of the path from that **Transmitter** to the **Receiver** is at or above the threshold set using the *RxDataThreshold* value.
RxFreq	**uint64**	**0**	The **Transceiver** receive frequency.
RxPathFactor	**float32**	**0.0**	The pathloss factor associated with the transmission path; could come from an external terrain server or pathloss server, internal calculations, or path factor. A value of 1.0 represents no loss.
RxPower	**float32**	**-270.0**	Indicates the receive power in dBm.
RxTxLocationX	**float64**	**0.0**	When the radio receives, it populates the transmitters location. This is the X position in a geocentric coordinate system of the radio that the **Transceiver** is currently receiving from.

Name	Type	Default Value	Description
RxTxLocationY	**float64**	**0.0**	When the radio receives, it populates the transmitters location. This is the Y position in a geocentric coordinate system of the radio that the **Transceiver** is currently receiving from.
RxTxLocationZ	**float64**	**0.0**	When the radio receives, it populates the **Transmitter's** location. This is the Z position in a geocentric coordinate system of the radio that **Transceiver** is currently receiving from.
RxVoiceTransmissionActive	**Boolean**	**FALSE**	When **TRUE**, **Transceiver** is receiving audio from another radio.
SNR	**float32**	**-270.0**	The signal-to-noise ratio of the received signal.
TxActive	**Boolean**	**FALSE**	Indicates whether **Transceiver** is transmitting.
TxVoice	**Boolean**	**FALSE**	Indicates when **Transceiver** is transmitting audio.
TxTDL	**Boolean**	**FALSE**	Indicates when **Transceiver** is transmitting Tactical Data Link (TDL).
TxCrypt	**Boolean**	**FALSE**	Indicates when **Transceiver** is in crypto mode and actively transmitting.
TxCryptoSoundIdx	**uint16**	**0**	When no **Tx_Preamble** sound file is playing, this number is 0. When a **Tx_Preamble** sound file is playing, this number reflects the sound index as configured in the sound library. This variable only appears in **Full View**.
TxFreq	**uint64**	**0**	Indicates **Transceiver** transmit frequency.

Table 295: Transceiver control outputs

Table 296, "Transceiver internal parameters" on page 259 lists and describes **Transceiver** internal parameter variables:

Name	Type	Default Value	Description
DomainName	**string**	**<Edit>**	*(Required)* Part of the ASTiNet property set that defines a common comms environment. All **Transceivers** in the same domain can talk. It's similar to a DIS Exercise ID or an HLA Federation name. When an ASCII string, **Transceiver** is natively an ASTiNet radio. When that string matches a defined DIS domain in a project, it maps the *DomainName* to a DIS exercise ID. To define a DIS exercise ID, enter **DIS:***N*, where *N*=1–255. For example, **DIS:1** puts **Transceiver** in DIS Exercise ID 1. Ensure another DIS domain is not already using the exercise ID.
PowerBus	**id**	**UNASSIGNED**	Connects to *PowerService* to receive power instead of using a control.
ProtocolId	**string**	**<Edit>**	Turns a radio into a DIS radio by setting the DIS identifiers (e.g., Host ID, Radio ID). **Marking Field** considers the radio "local" and unpublished on the network. • *DIS*:#.#.#.#: sets the Site, Application, Entity, and Radio IDs • *DIS*:#.#: sets the Entity and Radio ID • *DIS*:#: sets the Radio ID • *DIS*: sets all four IDs automatically • *LCL*: sets the radio as local and not published on the network. Blank identifiers generate automatically. Site and Application ID are the last two octets of the DIS IP address. Entity and Radio IDs are random. Append the above examples with a colon and **Marking Field**. For example, DIS:100.3:RedForce1 sets **Entity ID** to **100**, **Radio ID** to **3**, and **Marking Field** to **RedForce1**. The IP address sets Site ID and Application ID.

Name	Type	Default Value	Description
RadioBus	**id**	**UNASSIGNED**	Assigns **Transceiver** to **IntercomBusService**. This variable is most commonly used with **CommPanel**, which also uses **IntercomBusService**. **IntercomBusService** allows components connected to the bus to pass around input audio, sidetone audio, and output audio.
RadioName	**string**	**<Edit>**	The radio name that describes what **Transceiver** is modeling (e.g., UHF_Radio1). ***Important**: This parameter is required.*
RxState	**CryptoStateRx**	**RxCryptoOff**	This reports the current crypto receive state. Possible states include: • **RxCryptoOff** • **RxCryptoPreamble1** • **RxCryptoPreamble2** • **RxCryptoPostAmble** • **RxCryptoClear** • **RxCryptoMatch** • **RxCryptoMismatch** • **RxCryptoMismatchPreamble** Each one corresponds to a different sound file played from the crypto sound library. Crypto receive states include the following: • **0** // play no sound • **1** // play match preamble 1 • **2** // play match preamble 2 • **3** // play match post amble • **4** // play clear tone • **5** // play match tone • **6** // play mismatch tone • **7** // play mismatch tone
TransceiverId	**id**	**UNASSIGNED**	The connection to the radio control service. This bus defines the connection between the **Transceiver** and its **RCU**/**ICU**. ***Important**: This parameter is required.*

Name	Type	Default Value	Description
TxState	**CryptoStateTx**	**TxCryptoOff**	Reports the current crypto transmit state. Possible states include the following: • **TxCryptoOff** • **TxCryptoPreamble1** • **TxCryptoPreamble2** • **TxCryptoPostAmble** • **TxCryptoClear** • **TxCryptoCrypt** Each state corresponds to a different sound file in the crypto sound library. Crypto transmit states include following: • **0** // play no sound • **11** // play crypto preamble 1 • **12** // play crypto preamble 2 • **13** // play crypto post amble • **14** // play clear tone • **15** // play crypto tone
WorldPositionBus	**id**	**UNASSIGNED**	*(Optional)* Assigns **Transceiver** to a world position bus. If undefined, **Transceiver** defaults to X,Y,Z = 0,0,0. Alternatively if an entity exists on the DIS network that matches *Site.App.Entity ID* within the *ProtocolId* variable of **Transceiver**, **Transceiver** inherits that entity's world position information. When a bus name is defined, **Transceiver** is located (X,Y,Z) based on **Platform**, which is tied to the same bus name used with **Platform**/**GeocentricWorldPosition** or **Platform**/**GeodeticWorldPosition**.

Table 296: Transceiver internal parameters

Table 296, "Transceiver internal parameters" on the previous page lists and describes **Transceiver** internal display variables:

Name	Type	Default Value	Description
Mode	**TunerMode**	**None**	Displays the current **Transceiver** tuner mode (e.g., AM, FM, USB, LSB, etc.).
PowerState	**Boolean**	**TRUE**	Displays the power state of the **Transceiver**.
RxFrequency	**uint64**	**0**	Displays the current **Transceiver** receiver frequency.
TxFrequency	**uint64**	**0**	Displays the current **Transceiver** transmitter frequency.

Table 297: Transceiver internal display variables

15.13 Transmitter

Summary: **Transmitter** creates a transmit-only radio.

Description: **Transmitter** is a **Radio Control Unit** (**RCU**) included in the **Radio** group. It communicates with **Transceiver**, forcing transmit capability only. It is a subset of the features and controls located in **RCUbasic**.

The benefit of using **Transmitter** over **RCUbasic** or other all-purpose RCUs is that it charges a reduced number of credits for the reduced functionality. As a result, the component is useful for replicating navigational aids, such as very high frequency (VHF) Omnidirectional range (VOR) or nondirectional beacon (NDB) **Transmitters**.

For more information about using the **RCU** and **Transceiver**, go to Section 15.8, "RCUbasic" on page 236.

Table 298, "Transmitter control inputs" on the facing page lists and describes **Transmitter** control input variables:

Name	Type	Default Value	Description
CryptoKey	**uint16**	**0**	If two radios are using encryption then they must have matching cryptokeys for the crypto modes. This variable must match for proper radio tuning.
FreqHopNetID	**uint16**	**0**	Identifies the frequency-hopping net ID. A frequency-hopping radio must have a non-zero net ID to actively hop.

Name	Type	Default Value	Description
FreqHopSyncTOD	**uint32**	**0**	Identifies the time of day used in frequency hopping.
FreqHopTranSecKey	**uint16**	**0**	Identifies the transmission security key used to generate hopping patterns.
FreqHopHopSetWOD	**uint16**	**0**	Identifies the set of frequencies used in hopping patterns.
FreqHopLockOutId	**uint16**	**0**	Identifies the set of frequencies excluded from hopping patterns.
Frequency	**uint64**	**0**	The current transmitter tune frequency in Hz.
Net	**uint32**	**1**	Defines core radio features, including the following: • Frequency • Transmit frequency • Waveform • Crypto • Frequency hopping *Net* parameters in **commplan**.
TxPower	**float32**	**1.0**	Sets the transmit power for the **Transmitter**.

Table 298: Transmitter control inputs

Table 299, "Transmitter internal parameters" below lists and describes **Transmitter** internal parameter variables:

Name	Type	Default Value	Description
Fill	**fill**	**<Select>**	Insert the fill created in **commplan** by double-clicking in **Value**.
TransceiverID	**id**	**UNASSIGNED**	Tells **Transmitter** which **Transceiver** it should control.

Table 299: Transmitter internal parameters

15.14 VORTAC_Controller

Summary: **VORTAC_Controller** is designed to control embedded identifier tone elements to simulate very high frequency (VHF) omnidirectional range (VOR) and Tactical Air Navigation (TACAN) radios.

Description: **VORTAC_Controller** handles the timing of the VOR and TACAN tones to prevent overlap. Seven components are needed to implement a complete VORTAC simulation in a model. The TACAN and VOR radios each consist of a **Transceiver**, **RCU**, and **MorseKeyer**. Both **MorseKeyers** are driven by one **VORTAC_Controller**. **VORTAC_Controller** connects to *Enable* of **VOR** and **TACAN MorseKeyers** to keep proper timing of the Morse code transmission.

MorseKeyer reports busy to **VORTAC_Controller** to ensure accurate timing of the Morse code transmissions. **MorseKeyer's** *Busy* variable must be connected to *VOR_Busy* and *TACAN_Busy* in **VORTAC_Controller**.

***Note**: In some versions of Studio, VOR_Busy and TACAN_Busy are only visible in **Full View**.*

Figure 49, "VORTAC_Controller data flow" below shows the **VORTAC_Controller's** data flow:

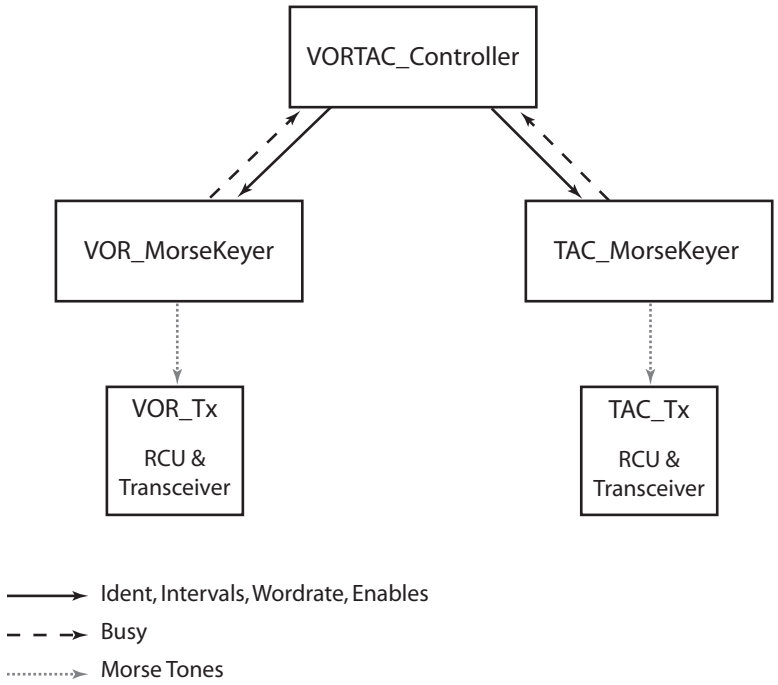


Figure 49: VORTAC_Controller data flow

Table 300, "VORTAC_Controller control inputs" on the facing page lists and describes **VORTAC_Controller** control input variables:

Name	Type	Default Value	Description
Enable	**Boolean**	**TRUE**	When **TRUE**, **VORTAC_Controller** is enabled.
Ident	**ident**	N/A	The ASCII characters that drive the **MorseKeyer**. *Ident* is a special variable type defined as ASCII characters concatenated together. **HostIn** must drive this parameter.
Interval	**uint16**	**1**	Used to set the delay between words. Units are in seconds.
TACAN_Busy	**Boolean**	**TRUE**	Connects to VOR **MorseKeyer** busy to be informed when the keyer is keying.
VOR_Busy	**Boolean**	**TRUE**	Connects to VOR **MorseKeyer** busy to be informed when the keyer is keying.
VOR_Count	**uint16**	**1**	Number of keyed identifiers to skip before keying *TACAN* identifier. *VOR* and *TACAN* identifier tones are mutually exclusive.
Wordrate	**uint8**	**1**	Determines rate at which the word is played. Units are in dots per second. The faster the rate, the higher the number.

Table 300: VORTAC_Controller control inputs

Table 301, "VORTAC_Controller control outputs" below lists and describes **VORTAC_Controller** control output variables:

Name	Type	Default Value	Description
IdentOut	**ident**	N/A	Drives VOR and TACAN MorseKeyer identifiers.
IntervalOut	**uint16**	**1**	Drives VOR and TACAN MorseKeyer intervals.
TACAN_Enable	**Boolean**	**TRUE**	Connects to TACAN MorseKeyer enable.
VOR_Enable	**Boolean**	**FALSE**	Connects to VOR MorseKeyer enable.
WordrateOut	**uint8**	**1**	Drives VOR and TACAN MorseKeyer word rates.

Table 301: VORTAC_Controller control outputs

16.0 Speech

The following sections describe the functionality and variables **SpeechFeed** and **TextToSpeech**.

16.1 SpeechFeed

Summary: Streams audio into the speech recognition engine.

Description: This component links an audio stream containing an operator's speech to the **SpeechFeed** to perform speech recognition on the audio. Once a stream ID is created and selected, the audio is fed into the speech recognition engine and processed according to the speech recognition configuration and grammars in the project's SR Plans folder. To set up speech recognition streams and grammars, go to the *Studio Technical User Guide*.

Table 302, "SpeechFeed audio input" below describes the **SpeechFeed** audio input:

Name	Type	Default Value	Description
AudioIn	**audio**	N/A	The audio stream directed to the speech recognition engine.

Table 302: SpeechFeed audio input

Table 303, "SpeechFeed internal parameter" below describes the **SpeechFeed** internal parameter:

Name	Type	Default Value	Description
StreamID	**id**	**UNASSIGNED**	Name or identifier given to the speech recognition stream. Double-click **UNASSIGNED** to open the speech service window, and select **New Bus** to add a speech recognition stream. Select **Set Value** to set the stream in **SpeechFeed**.

Table 303: SpeechFeed internal parameter

16.2 TextToSpeech

Summary: **TextToSpeech** attaches to the ID of the specified stream and outputs the audio.

Description: **TextToSpeech** selects up to four streams, or however many your license allows. The gain is applied to the selected stream, and the audio routes out of the component.

Table 304, "TextToSpeech audio output" below lists and describes the **TextToSpeech** audio output variable:

Name	Type	Default Value	Description
AudioOut	**audio**	N/A	The audio stream out signal.

Table 304: TextToSpeech audio output

Table 305, "TextToSpeech control outputs" below lists and describes **TextToSpeech** control output variables:

Name	Type	Default Value	Description
Gain	**float32**	**0.0**	The strength of volume applied to the audio out signal.
ActivePlaying	**Boolean**	**FALSE**	When **TRUE**, **TextToSpeech** is active.

Table 305: TextToSpeech control outputs

Table 306, "TextToSpeech internal parameter" below lists and describes the **TextToSpeech** internal parameter variable:

Name	Type	Default Value	Description
StreamID	**uint8**	**0**	The ID of the audio streams accepts multiple audio streams.

Table 306: TextToSpeech internal parameter

17.0 Remote Control

The following section describes **URC-200's** remote control capabilities.

17.1 URC-200

Summary: **URC-200** interfaces with a single URC-200 live radio through an ACE-RIU channel.

Description: **URC-200** provides a low-level interface to an URC-200 radio through an ACE-RIU serial port. The interface presented in the data viewer is similar to a combination **Radio**/**RCU** + **Radio**/**Transceiver** data viewer. The component contains **control** and **status** sections.

Control accepts user input and drives the radio settings when applicable. Radio items in **control** are generally prefixed with an **RCU** string, as in **RCU_Preset**. Items in **status** do not have a prefix string, as in **Preset**.

If the live radio fails to respond to a query for a specific status item (e.g., a query the current preset), **URC-200** displays an error indicator (e.g., -1 or “Unavailable”).

Input to the items in **control** are naively checked for validity. For example, **URC-200** does not know which frequency ranges are valid during live radio operation. On the contrary, **URC-200** assumes that you completely understand the constraints on parameters for different modes of operation. If you set an invalid input, **Transceiver** most likely returns a NAK, which *ErrorMask* eventually flags as an error.

For a more intuitive interface, go to the Remote Control in the Telestra web interface. The ideal interface to use with **URC-200** is the **Live Radio Remote Control** page of the Telestra web interface.

A Python API for interfacing with **Remote Control** in a model also exists; this API allows programmatic control over live radio entities present in the system. Contact ASTi for more information.

Table 307, "URC-200 command list" below shows the complete list of commands sent to the live radio RS232 serial interface from **URC-200**:

Command Name	Command Code
Zap	Z
Set Squelch	$
Set Preset	P
Set Frequency	R
Set Transmit Frequency	T
Set Modulation Mode	M
Set Tx Modulation Mode	N
Set Text Mode	X
Store Preset	Q
Set Power Level	#
Set Beacon Mode	*
Set Keypad Control	+
Set Transmit Mode	B
Set Receive Mode	E
Synth Lock/Unlock	?01
Receive Sig Strength	?03
SW Version	?08
Squelch Level Setting	?09
Current Preset Status	?10
Text Mode Status	?11
General Status	?11
General Mode Status	?12
Squelch Status	?13

Table 307: URC-200 command list

***Note**: For more information about URC-200 commands, go to the URC-200 operator manual.*

Table 308, "URC-200 control inputs" on page 270 shows **URC-200** control input variables:

Name	Type	Default Value	Description
ControlEnable	**Boolean**	**TRUE**	When **TRUE**, remote control is enabled. Control messages are passed between the ACE-RIU and the live radio.
LoadPreset	**Boolean**	**TRUE**	When **TRUE**, **URC-200** attempts to load presets but does not attempt to modify preset settings: • **Frequency** • **Tx Frequency** • **Modulation Mode** • **Tx Modulation Mode** • **Power Level** • **Text Mode** Consequently, the live radio's current setting overrides **Result**.
StorePreset	**Boolean**	**FALSE**	Triggers a **Store Preset** command on the remote radio, saving the current preset's state to the radio's internal memory. Go to the radio's operator manual for more information. The save command sends when this variable is first set to **TRUE** from a **FALSE** state.
ExternalPTT	**Boolean**	**FALSE**	When **TRUE**, the control message exchange between the ACE-RIU and live radio are halted. Keep **TRUE** when the live radio is transmitting, so the radio is not simultaneously transmitting and processing remote control messages. This might cause excessive noise on the transmitted signal.
RCU_Preset	**uint8**	**0**	Sets the live radio preset to switch to. Valid presets are in the range of 0–9.
RCU_Freq	**uint64**	**0**	Set the live radio frequency to tune to. Valid frequencies vary depending on other factors; go to the radio's operator manual for more information.

Name	Type	Default Value	Description
RCU_TxFreq	**uint64**	**0**	Sets the live radio transmit frequency to tune to. **URC-200** cannot set the frequency and transmit frequency together. If the two frequencies must match, set both of them separately. Valid frequencies varies depending on other factors; for more information, go to the radio's operator manual.
RCU_Squelch	**uint8**	**255**	Sets the live radio squelch level. Values in the range 0–255 are valid, with 255 being the highest level.
RCU_PowerLevel	**uint8**	**0**	Sets the live radio power level. Three settings exist: • *0*: LO • *1*: MED • *2*: HI Some settings are not possible; go to the radio's operator manual for information on setting the power level.
RCU_TextMode	**uint8**	**0**	Set the live radio text mode. Two settings exist: • *0*: plain text (PT) • *1*: cipher text (CT)
RCU_ModMode	**uint8**	**0**	Set the live radio to receive modulation mode. Two settings exist: • *0*: AM • *1*: FM Some settings are not possible in all cases; to set the modulation mode, go to the radio's operator manual.
RCU_TxModMode	**uint8**	**0**	Sets the live radio transmit modulation mode. Two settings exist: • *0*: AM • *1*: FM Some settings are not possible in all cases; to set the modulation mode, go to the radio's operator manual.

Name	Type	Default Value	Description
RCU_Operating Mode	**uint8**	**0**	Set the live radio's operating mode. Four possible settings exist: • **0**: BYPASS • **1**: RECEIVE • **2**: TRANSMIT • **3**: BEACON In BYPASS mode, **URC-200** does not attempt to control the operating mode. This is useful if another entity is driving the live radio's operating mode. For example, if you use a handset with the radio while the remote control system is connected, BYPASS. For more information about the other three modes, go to the radio's operator manual. ***Caution**: TRANSMIT and BEACON mode imply that **Transceiver** is actively keyed and transmitting over the air. Ensure the radio has antennae or equivalent load available.*

Table 308: URC-200 control inputs

Table 308, "URC-200 control inputs" above lists and describes **URC-200** control output variables:

Name	Type	Default Value	Description
Interface	**string**	N/A	A helpful status message about the state of the live radio remote control interface. Example messages include the following: • "Remote Control Off:" signifies that the serial messaging is disabled for the interface, most likely because *ControlEnable* is **FALSE**. • "Err: Bad device name/chan:" an invalid ACE-RIU device or channel has been specified. • "Err: check mask / log:" a control query might be failing consistently; check the component log. For more information, go to the *radio.log_level* control. • "Okay:" normal operation

Name	Type	Default Value	Description
SuccessMask	**uint64**	**0**	A 64-bit mask for tracking the outcome of commands to the live radio. To find specific commands in the mask, go to **Command List**. When a command succeeds, as determined by receiving an ACK from the live radio and or receiving a valid response, the bit for the command is 1. The bit is 0 when the command is sent again.
ErrorMask	**uint64**	**0**	A 64-bit mask that tracks the outcome of commands to the live radio. To find specific commands in the mask, go to Table 307, "URC-200 command list" on page 267. Two conditions cause a command to fail: • **URC-200** receives an NAK from the live radio. • **URC-200** receives a response that does not comply with the protocol defined in the *General Dynamics URC-200 V2 Manual, Document No. 99- P42304K)*. When a command fails three times, the corresponding bit in *ErrorMask* is 1. If the same command succeeds once, the bit resets to 0.
Option	**string**	**None**	A string detailing the options installed on the URC-200 transceiver. Valid strings include: • *None*: no options installed. • *30_90*: the EBN-30 option is installed. • *420*: the EBN-400 option is installed. • *30_90 && 420*: both EBN-30 and EBN-400 options are installed. • *Unavailable*: displayed on error.
SquelchStatus	**int8**	**0**	The live radio's squelch status; two values are valid: • *0*: the transceiver is squelched. • *1*: the transceiver's squelch has broken. • *-1*: shown on error

Name	Type	Default Value	Description
Overtemp	**int8**	**0**	The live radio's overtemp status; two values are valid: • *0*: temperature is okay • *1*: overtemp condition • *1-*: error
SynthLock	**int8**	**0**	The live radio's synthesizer lock status; two values are valid: • *0*: synthesizer is unlocked • *1*: synthesizer is locked • *-1*: error
SWVersion	**string**	**None**	The live radio's software version; an "Unavailable" displays on error.
Preset	**int8**	**0**	The live radio's preset number; valid values are in the range 0–9. A -1 value is shown on error.
Freq	**int64**	**0**	The live radio's receive frequency. A -1 value is shown on error.
TxFreq	**int64**	**0**	The live radio's transmit frequency. A -1 value is shown on error.
Squelch	**int16**	**0**	The live radio's squelch level, valid values are in the range of 0–255. A -1 value is shown on error.
PowerLevel	**int8**	**0**	The live radio's power level. Valid values are: • *0*: LO • *1*: MED • *2*: HI • *-1*: error
TextMode	**int8**	**0**	The live radio's text mode. Valid values are: • *0*: plain text (PT) • *1*: cipher text (CT) • *-1*: error
ModMode	**int8**	**0**	The live radio's receive modulation mode. Valid values are: • *0*: AM • *1*: FM • *-1*: error

Name	Type	Default Value	Description
TxModMode	**int8**	**0**	The live radio's transmit modulation mode. Valid values are: • *0*: AM • *1*: FM • *-1*: error
OperatingMode	**int8**	**0**	The live radio's operating mode. Valid values are: • *1*: Receive • *2*: Transmit • *3*: Beacon • *-1*: Error
RxSignalStrength	**int16**	**0**	The live radio's receive signal strength; valid values are in the range 0–255. A -1 value is shown on error.

Table 309: URC-200 control outputs

Table 310, "URC-200 internal parameters" below lists and describes **URC-200** internal parameter variables:

Name	Type	Default Value	Description
DeviceName	**deviceid**	**<Select>**	Select the name of the ACE-RIU device.
DeviceChannel	**riu_channel**	**None**	Select the ACE-RIU serial channel; serial channels can be either A or C.
RadioHandle	**string**	**<Edit>**	Enter a useful handle to help identify this radio object. The handle is useful when using the Remote Control feature in the Telestra web interface.
radio.log_level	**uint32**	**0**	Set this parameter to affect how much logging **URC-200** outputs. Only a level of 1 is supported. Set this parameter to 0 if you are not actively debugging **URC-200**, or the live radio interface as spurious log output is produced. Looking at the log is useful for identifying specific, unresponsive commands. This parameter is only accessible in **Full View**.

Table 310: URC-200 internal parameters