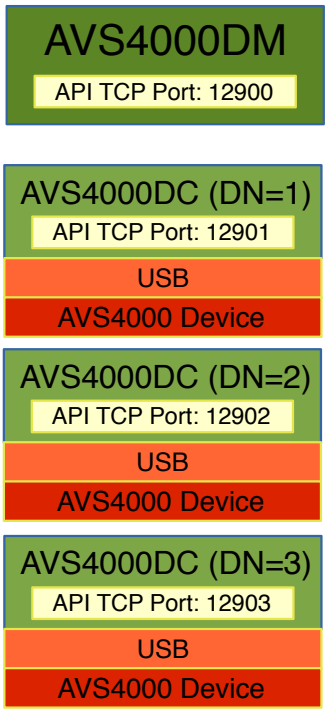
AVS4000 API

The AVS4000 transceiver is controlled by a software application called the AVS4000 Daemon (*AVS4000D*). *AVS4000D* manages all of the AVS4000 devices connected to the system. *AVS4000D* also provides a TCP based multiple control Application Programming Interfaces (API) that can be used to control the AVS4000 devices.

The following diagram describe the different processes that are created by *AVS4000D*.

AVS4000 Daemon



AVS4000DM (Device Manager) handles detecting and identifying all AVS4000 devices. For each AVS4000 device detected, an AVS4000DC (Device Controller) will be started.

The AVS4000DM API can be used detect which devices are available and identity information for each device.

For each device detected in the system a Device Number (DN) is assigned.

An AVS4000DC (Device Controller) process is create to handle the operation of a specific AVS4000 device. The DN (Device Number) specifies which device is being controlled.

Each AVS4000 Device has its AVS4000DC API available on its own TCP port. The TCP port for each device is 12900+DN.

Avid Control API

The AVS4000 provides Application Programming Interaces (*API*) that are based on the Avid Control API (*AVSAPI*). The basics of the *AVSAPI* is generic to support multiple products.

Requirements

- **Simple & Lightweight:** The overhead for implementing the protocol should be small.
- **Human Readable:** What goes over the wire doesn't have to be directly human readable, but there should be a standard human readable representation that can be displayed.
- **Extendable:** Should be able to add new commands or parameters without breaking out of date client or server.
- **Widely supported:** The protocol should use existing facilities available in scripting languages such as Python.
- **Transport Agnostic:** The command and control protocol should be able to use a variety of transports. UDP, TCP, ZeroMQ, Local Sockets/Named Pipes, etc...
- **Stability:** Implementation of the API should be robust and tolerate improperly formatted requests.

JSON (json.org)

Javascript Object Notation (JSON) is a lightweight data-interchange format. The EMCA-404 is a standard that defines JSON.

JSON meets the requirements listed above. JSON is widely supported on in multiple programming and scripting languages.

Data Types

The following are all possible data types in JSON.

- **String:** One or more Unicode characters
- **Number:** Any integer or floating point number
- **Boolean:** *true* or *false*
- **Null:** Presents the absence of a value
- **Array:** Ordered collection of values
- **Map:** Unordered name-value pairs, where *name* is a string.

NDJSON (ndjson.org)

Newline delimited JSON provides a specification for streaming JSON Objects. The idea is that the JSON of each message is in compact form (no line feeds). Each object is then separated by a LF (new line).

NDJSON could be used for JSON implementation over TCP, named pipes or any streaming interface.

Control API Design

The *AVSAPI* provides the ability to control and configure the AVS4000. This is done via a request (*REQ*) and response (*RSP*). The *REQ* consists of a command and parameters. The *RSP* contains a boolean, error information and return values. The boolean represents if the *REQ* succeeded.

The *REQ* is encoded and sAVS4000ent via a transport to the system. The encoding is JSON and the transport used is TCP/IP.

The *RSP* will be received over the same transport as the *REQ* and encoded in the same manner.

JSON encoded *REQ/RSP* are encoded in compact *JSON*, which means there are no line feeds (*LF*) or carriage returns(*CR*). For streaming transports such as *TCP*, line feeds (*LF*) are used to separate messages as described by NDJSON

REQ

The client creates a JSON message known as a request (*REQ*). The *REQ* is a JSON list. The first element in the list is always the command string. Optionally the second element is used as the command parameter(s).

Format:

```
[ "CMD", ARG ]
```

> **CMD:** String for REQ command
> **ARG:** Optional command argument (Any valid *JSON* value)

The following is a list of possible formats used for API requests:

```
[ "CMD" ]
[ "CMD", {} ]
[ "CMD", { "param1":value1, "param2":value2} ]
[ "CMD", [] ]
[ "CMD", [value1,value2] ]
[ "CMD", value1 ]
```

> **CMD:** String for REQ command
> **paramN:** String for command parameter name
> **valueN:** JSON value for command parameter

RSP

The response (*RSP*) message is a JSON message that represents the results of the previous *REQ*. It is always a JSON list. The first element of the list is always a boolean that indicates if the *REQ* was successful.

Format:

```
[ success, rval ]
[ success, errorCode, errorDetails ]
```

> **success**: The first value in the *RSP* list is always a boolean that indicates if the *REQ* was successful. If **success** is *true*, it may be followed by a return value (**rval**). If **success** is *false*, an **errorCode** and **errorDetails** are in the *RSP*.
> **rval**: Return value from successful *REQ*. Can be any valid JSON data type.
> **errorCode:** Integer an enumerated error code returned when *REQ* was unsuccessful.
> **errorDetails:** String that gives a description of the error.

API Command

The API Command is the equivalent to a function call into a C library.

- Commands are always JSON text string.
- Commands are sent in the *REQ* as the first element in the list.
- Commands are case insensitive.
- Standard commands will be available on all systems.
- Product specific commands may also be available.

API Parameters

API parameters are used for configuration and control the operation of the hardware, firmware or software contained in the Avid Product.

- Parameters have containers called, API *groups*.
- Each parameter *must* belong to an API group.
- Each parameter has a valid JSON data type.
- Parameter names are case insensitive.
- Parameter have an access type of: read-only, write-only or read-write.
- Parameters may be read using a **GET** command.
- Parameters may be changed using a **SET** command.
- Parameters are validated during a **SET** command.

- Multiple parameters can be changed within a single **SET** command
- A **SET** may be rejected due to one or more parameter's value being out of range.
- A **SET** will only return an error for 1 parameter per call. The first parameter error detected will be the error returned.
- A failed **SET** command will NOT make any changes to ANY parameters in the command.
- Validated parameter changes won't take effect until a **COMMIT** is issued.
- The **SET** command does any automatic **COMMIT** after all parameters have been successfully validated.
- The **SETN** command does NOT do an automatic **COMMIT**.
- The **GETP** command will get values of parameter that have pending changes (not committed).

API Group

An API Group is a collection of API parameters that are related. Typically these will be associated with a hardware/system component. Behind the scenes, the **COMMIT** is implemented on a per *group* basis. This means any changes committed to an API Group can perform any special procedures that are needed to make the change to the hardware interface.

- API group is a collection related parameters.
- API group names are case insensitive.

Process for changing API parameters

- One or more settings are changed via **SET** or **SETN** command
- Each new parameter value is validated.
- If ANY parameter values in the **SET** or **SETN** command are invalid, then an error will be returned.
- If ALL parameter values are valid, then the new values are stored in a temporary cache.
- A **COMMIT** command will write parameters stored in the temporary cache to the hardware.
- If the **SET** command was used, then an automatic **COMMIT** will be performed.
- A **DISCARD** command will empty the temporary cache, erasing any recent changes performed using **SETN** commands.

Commands

The following a list of standard *AVSAPI* commands.

COMMIT

Command that activates pending changes that have been performed using *SETN* commands.

Parameters: None
Return Value: None

Example:

```
        REQ: ["COMMIT",""]
        RSP: [true]
```

DISCARD

Command that discards pending changes that have been performed using *SETN* commands.

Parameters: None
Return Value: None

Example:

```
        REQ: ["DISCARD",""]
        RSP: [true]
```

GET

Command that gets current values for one or more API parameter groups.

Parameters:

- *None*: All API groups are returned
- *String*:

    - *Group*: If the string is a *group* name, then the entire parameter group will be returned.
    - *Group.Param*: If the string is of the form *group.param*, then a single parameter will be returned.

- *List*: List of strings where each string is a *group* name, returns multiple groups

Return Value: Object with API parameter values

Example:

```
        REQ: ["GET","master"]
        RSP: [true,
             {"master":{"SampleRate":40000000,
              "SampleRateMode":"Manual"}}]
```

GETP

Command that gets *pending* values for one or more API parameter groups.

Parameters:

- *None*: All API groups are returned
- *String*: Returns parameters for a single group that matches name
- *List*: List of strings returns multiple groups

Return Value: Object with API parameter values

Example: The following shows a *GETP* command following the *SETN* example listed below.

```
        REQ: ["GETP"]
        RSP: [true,
             {"ddc":{},"duc":{},
              "master":{"SampleRate":42000000},
              "ref":{},"rx":{},"rxdata":{},
              "tx":{},"txdata":{}}]
```

GETERR

Command that returns a complete list of possible error code used by the AVSAPI.

Parameters: None
Return Value: List of Error Code & Error Description pairs.

Example:

```
        REQ: [ "GETERR" ]
        RSP: [true,[[0,"Success"],[1,"Syntax Error"],[2,"Invalid Command"],
             [3,"Missing Command"],[4,"Invalid Parameter"],[5,"Missing Parameter"],
             [6,"Parameter Invalid Type"],[7,"Parameter Invalid Value"],
             [8,"Parameter Out of Range"],[9,"Parameter Read Only"],
             [10,"Invalid Config Group"],[11,"Invalid Config Parameter"],
             [12,"Timeout"],[13,"Failure"],[14,"Partial Commit"]]]
```

GETCMD

Command that returns a list of supported commands for the system.

Parameters: None
Return Value: List of commands & description pairs.

Example:

```
        REQ: [ "GETCMD" ]
        RSP: [true,
             [["GET","Get values of config parameters"],
             ["SET","Set values of config parameters and commit changes"],
             ["GETP","Get values of pending config parameters"],
             ["SETN","Set values of config parameters (NO Commit)"],
             ["COMMIT","Commit pending parameter changes."],
             ["DISCARD","Discard pending config changes"],
             ["GETCMD","Get list of available commands"],
             ["GETERR","Get list of defined error codes"],
             ["INFO","Get information about parameters"]]]
```

INFO

Command that returns a list of descriptions for one or all API parameter groups.

Parameters:

- *None*: All API groups are returned
- *String*: Returns information for a single group that matches name

Example: The following example shows information for the parameters in the group *Master*.

```
        REQ: ["INFO","master"]
        RSP: [true,
             {"master":{
                "SampleRate":"Sample Rate (Hz) [2.5e6 to 61.44e6]",
                "SampleRateMode":"Sample Rate Mode (Str) [Auto,Manual]"
                }
            }]
```

SET

Command that sets the values for one or more API parameter groups. Once each parameter has been successfully validated, then a *COMMIT* is automatically performed.

Parameters:

- *Map of group objects*: Each group object contains one or more parameter/value pairs of parameters that are to be changed. (*required*)

Return Value: None

Example: The following example shows changing a parameter, *SampleRate* in group *Master*

```
        REQ: ["SET",{"master":{"SampleRate":44e6}}]
        RSP: [true]
```

SETN

Command that sets the values for one or more API parameter groups. After parameter validation, *NO* commit is performed. To activate changes, an explicit *COMMIT* must be performed.

Parameters:

- *Map of group objects*: Each group object contains one or more parameter/value pairs of parameters that are to be changed. (*required*)

Return Value: None

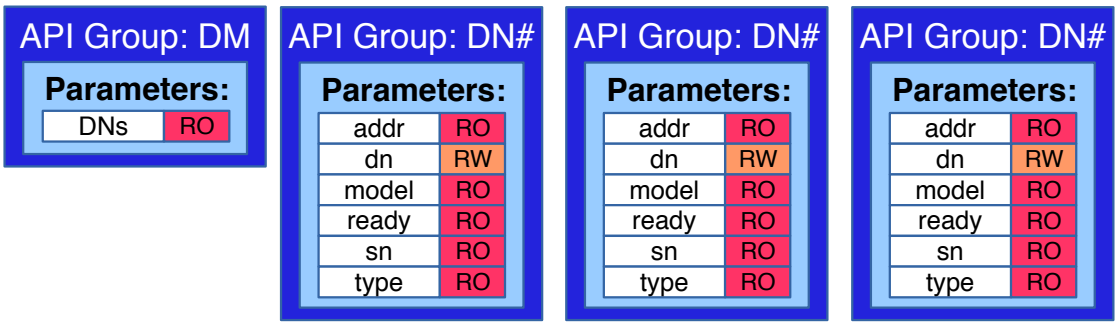
Example: The following example shows changing a parameter, *SampleRate* in group *Master*

```
        REQ: ["SETN",{"master":{"SampleRate":44e6}}]
        RSP: [true]
```

AVS4000DM API Parameters

The AVS4000 DM (Device Manager) API uses the standard AVSAPI design. The following section includes a list of the DM API parameters used to control the AVS DM.

AVS4000DM API



API Group: DM

The following parameters are part of the *DM* (device manager) group of API parameters.

Parameter	Type	Access	Description
DNs	list	RW	List of Active Device Numbers

API Group: DN#

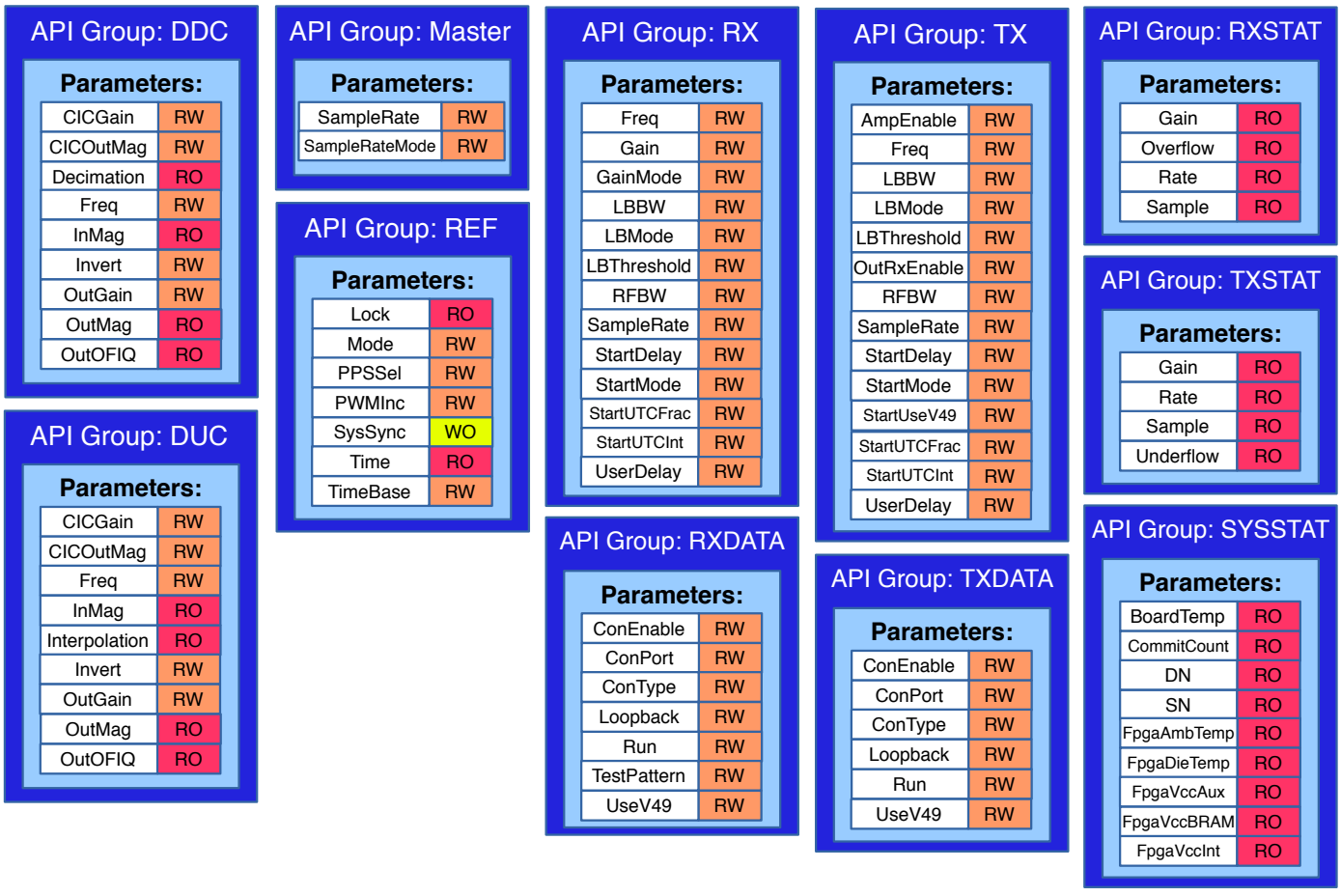
An API group is created for each active device in the system. The group name for each active device is of the form *DN#* where # is the device number for the device. For example, for DN=1, the group name would be *DN1*.

Parameter	Type	Access	Description
addr	uint	RO	Device class specific addres that uniquely identifies the device.
dn	uint	RW	Device number for the device
model	string	RO	Model name for the device
ready	bool	RO	Ready state of the device (true=device is ready)
sn	string	RO	Model specific serial number for the device
type	string	RO	Type associated with the device

AVS4000DC API Parameters

The AVS4000DC (Device Controller) API uses the standard AVSAPI design. The following section includes a list of the API parameters used to control a Specific AVS4000. The TCP port number associated with a specific device is 12900+DN, where DN is the device number for the device.

AVS4000DC API



API Group: DDC

The following parameters are part of the *DDC* group of API Parameters.

Parameter	Type	Access	Description
CICGain	float	RW	CIC Gain (dB)
CICOutMag	uint	RO	CIC Output Average Magnitude (dBFS)
Decimation	uint	RO	Decimation [1 to 8192]
Freq	int	RW	Tuning Freq (Hz) [-50e6 to 50e6]
InMag	int	RO	Input Average Magnitude (dBFS)
Invert	bool	RW	Invert Spectrum
OutGain	float	RW	Output Gain (dB) [-72.2471 to 30.1029]
OutMag	float	RO	Output Average Magnitude (dBFS)
OutOFIQ	bool	RO	CIC Output Overflow (UInt) [I=lower 16-bit,Q=upper 16-bit]

API Group: DUC

The following parameters are part of the *DUC* group of API Parameters.

Parameter	Type	Access	Description
CICGain	float	RW	CIC Gain (dB)
CICOutMag	uint	RO	CIC Output Average Magnitude (dBFS)
Freq	int	RW	Tuning Freq (Hz) [-50e6 to 50e6]
InMag	int	RO	Input Average Magnitude (dBFS)
Interpolation	uint	RO	Interpolation [1 to 8192]
Invert	bool	RW	Invert Spectrum
OutGain	float	RW	Output Gain (dB) [-72.2471 to 30.1029]
OutMag	float	RO	Output Average Magnitude (dBFS)
OutOFIQ	bool	RO	CIC Output Overflow [I=lower 16-bit,Q=upper 16-bit]

API Group: MASTER

The following parameters are part of the *Master* group of API Parameters.

Parameter	Type	Access	Description
SampleRate	float	RW	Sample Rate (Hz) [1e6 to 50e6]
SampleRateMode	string	RW	Sample Rate Mode [Auto,Manual]

API Group: REF

The following parameters are part of the *Ref* group of API Parameters for controlling the system reference.

Parameter	Type	Access	Description
Lock	bool	RO	Reference Lock
Mode	string	RW	Reference Mode [Internal,InternalStatic,External,GPSDO,PPS]
PPSSel	string	RW	PPS Select [Internal,External,GPS]
PWMInc	uint	RW	PWM Increment [0 - 0xFFFF]
SysSync	bool	WO	System Sync
Time	uint	RO	Time since the UTC Epoch (ms)
TimeBase	string	RW	Timebase [GPS,Host]

API Group: RX

The following parameters are part of the *RX* group of API Parameters for controlling receive portion of the transceiver.

Parameter	Type	Access	Description
Freq	uint	RW	Tuning Freq (Hz) [50e6 to 6e9]
Gain	int	RW	RF Gain (dB) [-3 to 71]
GainMode	string	RW	RF Gain Mode [Manual,FastAGC,SlowAGC,HybridAGC]
LBBW	string	RW	Low Band Bandwidth [Narrow,Wide]
LBMode	string	RW	Low Band Mode [Auto,Enable,Disable]
LBThreshold	uint	RW	Low Band Threshold (Hz) [5e6 to 5e9]
RFBW	uint	RW	RF Bandwidth [100e3 to 56e6, 0=auto]
SampleRate	uint	RW	Sample Rate (Hz) [1e6 to 50e6]
StartDelay	uint	RW	Start Delay (Sec) [1 to 300]
StartMode	string	RW	Start Mode [Immediate,OnPPS,OnFracRoll,OnTime]
StartUTCFrac	uint	RW	Start UTC Fracional
StartUTCInt	uint	RW	Start UTC Integer (Sec)
UserDelay	uint	RW	User Delay

API Group: RXDATA

The following parameters are part of the *RXDATA* group of API Parameters for controlling RX Data stream.

Parameter	Type	Access	Description
ConEnable	bool	RW	Enable/Disable Conection [true=connect,false=disconnect]
ConPort	uint	RW	Connection Port [0 - 0xFFFF]
ConType	string	RW	Connection Type [TCP]
Loopback	bool	RW	Enable/Disable Loopback
Run	bool	RW	Enable/Disable RX Data [true=start,false=stop]
TestPattern	bool	RW	Enable/Disable Test Pattern
UseV49	bool	RW	Enable/Disable Vita 49 Packets

API Group: RXSTAT

The following parameters are part of the *RXSTAT* group of API Parameters for status about RX Data stream.

Parameter	Type	Access	Description
Gain	float	RO	Roll-up Gain for Stream (dB)
Overflow	uint	RO	Number of overflow errors for stream
Rate	string	RO	Current Transfer Data Rate (MB/s)
Sample	uint	RO	Current Transfer sample count

API Group: TX

The following parameters are part of the *TX* group of API Parameters for controlling transmit portion of the transceiver.

Parameter	Type	Access	Description
AmpEnable	bool	RW	TX Amp Enable
Freq	uint	RW	Tuning Freq (Hz) [50e6 to 6e9]
LBMode	string	RW	Low Band Mode [Auto,Enable,Disable]
LBThreshold	uint	RW	Low Band Threshold (Hz) [5e6 to 5e9]
OutRxEnable	bool	RW	Out RX Enable
RFBW	uint	RW	RF Bandwidth [100e3 to 56e6, 0=auto]
SampleRate	uint	RW	Sample Rate (Hz) [1e6 to 50e6]
StartDelay	uint	RW	Start Delay (Sec) [1 to 300]
StartMode	string	RW	Start Mode [Immediate,OnPPS,OnFracRoll,OnTime]
StartUseV49	bool	RW	Use Vita49 for Start Time
StartUTCFrac	uint	RW	Start UTC Fracional
StartUTCInt	uint	RW	Start UTC Integer (Sec)
UserDelay	uint	RW	User Delay

API Group: TXDATA

The following parameters are part of the *TXDATA* group of API Parameters for controlling TX Data stream.

Parameter	Type	Access	Description
ConEnable	bool	RW	Enable/Disable Conection [true=connect,false=disconnect]
ConPort	uint	RW	Connection Port [0 - 0xFFFF]
ConType	string	RW	Connection Type [TCP]
Run	bool	RW	Enable/Disable RX Data [true=start,false=stop]
UseV49	bool	RW	Enable/Disable Vita 49 Packets

API Group: TXSTAT

The following parameters are part of the *TXSTAT* group of API Parameters for status about TX Data stream.

Parameter	Type	Access	Description
Gain	float	RO	Roll-up Gain for Stream (dB)
Rate	string	RO	Current Transfer Data Rate (MB/s)
Sample	uint	RO	Current Transfer sample count
Underflow	uint	RO	Number of underflow errors for stream

API Group: SYSSTAT

The following parameters are part of the *SYSSTAT* group of API Parameters for status about the system.

Parameter	Type	Access	Description
BoardTemp	float	RO	Board Temperature in celsius
CommitCount	uint	RO	Number of configuration changes since startup.
DN	uint	RO	Device Number assigned by Device Manager
SN	string	RO	Serial Number
FpgaAmbTemp	float	RO	FPGA ambient temperature in celsius
FpgaDieTemp	float	RO	FPGA die temperature in celsius
FpgaVccAux	float	RO	FPGA Vcc Aux voltage
FpgaVccBRAM	float	RO	FPGA Vcc BRAM voltage
FpgaVccInt	float	RO	FPGA Vcc Internal voltage

AVS4000 API Testing

The *AVS4000D* is the application that implements a test *AVS4000 API* and controls the AVS4000. In order to use the *AVS4000 API* the *AVS4000D* application must be running.

SOCAT

SOCAT is a Linux command line based utility that establishes bi-directional communication between sockets. This simple tool provides a way to test the *AVS4000 API* implemented in the *AVS4000D*.

Install: The following can be used to install *SOCAT* on Ubuntu.

```
$ sudo apt-get install socat
```

Testing: The following is an example of how *SOCAT* can be used to test the *AVS4000 API* (for DN=1) using the TCP JSON socket implementation. JSON *REQ* are typed in by the user and the *RSP* can immediately be seen. In the output below, a number of errors have been intentionally generated.

```
$ socat - tcp:localhost:12901
abc
[false,1,"Parse Error"]
[[abc
[false,1,"Parse Error"]
get
[false,1,"Parse Error"]
[get]
[false,1,"Parse Error"]
["getcmd"]
[true,[["GET","Get values of config parameters"],
["SET","Set values of config parameters and commit changes"],
["GETP","Get values of pending config parameters"],
["SETN","Set values of config parameters (NO Commit)"],
["COMMIT","Commit pending parameter changes."],
["DISCARD","Discard pending config changes"],
["GETCMD","Get list of available commands"],
["GETERR","Get list of defined error codes"],
["INFO","Get information about parameters"]]]
```

> In the above output additional line feeds have been inserted to make the output more readable. However, in real tests only a single linefeed is output at the end of each *RSP*.

Testing: The following is an example of how *SOCAT* can be used to test the *AVS4000 DM API* (device manager) using the TCP JSON socket implementation. JSON *REQ* are typed in by the user and the *RSP* can immediately be seen.

```
$ socat - tcp:localhost:12900
["get"]
[true,{
"DN1":{
    "addr":259,
    "dn":1,
    "model":"AVS4000",
    "ready":true,
    "sn":"SN0008",
    "type":"USB"
    },
"dm":{
    "DNs":[1]
    }
}]
```

> In the above output additional line feeds have been inserted to make the output more readable. However, in real tests only a single linefeed is output at the end of each *RSP*.

AVS4000 Data Streams

The AVS4000 RX and TX Data streams use TCP sockets. Data is transmitted over the socket as either *Raw IQ* data or *Vita49* packets. In *Raw IQ* mode, the first 16 bits is I and the second 16-bits is Q. The I & Q are signed 16-bit integers. All data for the data stream is little endian.

Steps for Receiving Data

1. Configure RX parameters: tune, sample rate, gain, etc..
2. Enable RX Data Connection: Specify TCP type and port.
3. Start RX Data
4. Connect to TCP RX Data socket
5. Receive & process data from socket
6. Stop RX Data

Example Python Script for Receiving Data to File

```
import socket
import json
import os

DN=1
HOST='localhost'
PORT=12900+DN
RX_PORT=12700+DN

# Connect to AVS4000 API socket
sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
sock.connect((HOST,PORT));

# Create REQ
P={};                    # Set REQ Map
G={};                    # RXDATA Group Map
G["conEnable"]=True;     # enable RX Data Connection
G["conType"]="tcp";      # Specify to use TCP
G["conPort"]=RX_PORT;    # Specify TCP Port to listen on
G["useV49"]=False;       # Receive Raw IQ Data
G["run"]=True;           # Start the stream
P["rxdata"]=G;           # Add RXDATA Group Map to REQ
REQ=['set',P];           # Create REQ for SET Command

sock.send(json.dumps(REQ)); # Convert REQ to JSON and send on socket
str=sock.recv(8192);        # Receive the RSP
RSP=json.loads(str);        # Parse JSON to Python list
print (RSP);                # Print the RSP

# Use SOCAT utitiltiy to connect to TCP RX Data Socket
# save data to a new file 'rx.out'.
os.system("socat -u TCP:localhost:12914 CREATE:rx.out &")
```

Example Python Script to Stop Receiving Data

```
#!/usr/bin/python
import socket
import json
import os

DN=1
HOST='localhost'
PORT=12900+DN
RX_PORT=12700+DN

# Connect to AVS4000 API socket
sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
sock.connect((HOST,PORT));

# Create REQ
P={};                  # Set REQ Map
G={};                  # RXDATA Group Map
G["conEnable"]=False; # Disable RX Data Connection
G["run"]=False;       # Stop the RX Data stream
P["rxdata"]=G;        # Add RXDATA Group Map to REQ

REQ=['set',P];        # Create REQ for SET Command

sock.send(json.dumps(REQ)); # Convert REQ to JSON and send on socket
str=sock.recv(8192);        # Receive the RSP
RSP=json.loads(str);        # Parse JSON to Python list
print (RSP);                # Print the RSP
```

Steps for Transmitting Data

1. Configure TX parameters: tune, sample rate, gain, etc..
2. Enable TX Data Connection: Specify TCP type and port.
3. Start TX Data
4. Connect to TCP TX Data socket
5. Transmit Data to socket
6. Stop TX Data

Example Python Script to Transmit Data from File

```
#!/usr/bin/python
import socket
import json
import os

DN=1
HOST='localhost'
PORT=12900+DN
TX_PORT=12800+DN

# Connect to AVS4000 API socket
sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
sock.connect((HOST,PORT));

# Create REQ
P={};                    # Set REQ Map
G={};                    # TXDATA Group Map
G["conEnable"]=True;     # enable TX Data Connection
G["conType"]="tcp";      # Specify to use TCP
G["conPort"]=TX_PORT;    # Specify TCP Port to listen on
G["useV49"]=False;       # Send Raw IQ Data
G["run"]=True;           # Start the stream
P["txdata"]=G;           # Add TXDATA Group Map to REQ

REQ=['set',P];           # Create REQ for SET Command

sock.send(json.dumps(REQ)); # Convert REQ to JSON and send on socket
str=sock.recv(8192);        # Receive the RSP
RSP=json.loads(str);        # Parse JSON to Python list
print (RSP);                # Print the RSP

# Use SOCAT utitiltiy to connect to TCP TX Data Socket
# read data from file 'tx.out'.
os.system("socat -u FILE:tx.out TCP:localhost:12924 &")
```

Example Python Script to Stop Transmit Data

```
#!/usr/bin/python
import socket
import json
import os

DN=1
HOST='localhost'
PORT=12900+DN
TX_PORT=12800+DN

# Connect to AVS4000 API socket
sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
sock.connect((HOST,PORT));

# Create REQ
P={};                  # Set REQ Map
G={};                  # TXDATA Group Map
G["conEnable"]=False; # Disable TX Data Connection
G["run"]=False;       # Stop the TX Data stream
P["txdata"]=G;        # Add TXDATA Group Map to REQ

REQ=['set',P];        # Create REQ for SET Command

sock.send(json.dumps(REQ)); # Convert REQ to JSON and send on socket
str=sock.recv(8192);        # Receive the RSP
RSP=json.loads(str);        # Parse JSON to Python list
print (RSP);                # Print the RSP
```