Analog Devices, Inc.

AVESGold Tool Suite User Guide

Build 1.00.0232

AVESGold Tool Suite User Guide
8-29-2018

Table of Contents

Table of Figures

Introduction

This is an installation and user guide for AVESGold. It is subject to change.

AVESGold is a suite of tools used to evaluate Analog Devices Video Boards and devices. It contains register maps and the ability to run scripts or commands form a console.

This code was primarily designed for client/server connections to the DUT however it can, with the appropriate plugins do a direct connection to a DUT. Once the server has been created anybody can create their own program and connect to the DUT as long as they go through the server. As a matter of fact multiple clients can communicate with the DUT at the same time just like web pages are served up to multiple client browsers.

If you have any questions or comments please start a discussion at the Video Support Community found at Video Support Community.

Words and Abbreviations used

AVES – Advantiv Video Evaluation Software [1]. Software used to evaluate ADV products.

DUT – Device Under Test

I2C – Inter IC. A communication bus designed by Philips (now NXP) to allow easy communications between integrated circuits. It uses two wires for communications, a data lane and a clock lane [2].

IPC – Inter Process Communications

Setting up AVES Gold for full use

Installing AVES Gold

- Download the AvesGold zip file
- Unzip AvesGold to extract the .msi file
- Run the .msi file and follow dialogs

Once completed you will find the programs Startup menu -> All Programs -> Analog Devices -> AvesGold folder.

Applications:

- AvesGold – Main application.
- Probe – Probe application.
- TestPipeServer – and application to test custom servers.

Servers:

- ServerPipeNull – A pipe server that implements a null/loopback data storage.
- ServerPipeApolloI2C – A pipe server that connects to the target though and Apollo board.

Plugins

- Null – Simple null/loopback data storage meant for testing.
- Pipe – connection to the Pipe. Requires a pipe server to be running.
- ApolloI2C – Direct connection to the target via the Apollo board.

See the possible topology options see which programs need to be run.

As other clients, servers, Plugins and IPCs are created they will be added to the appropriate folder locations.

System Architecture

AvesBlue had taught me a lot about how people want to use Aves and the tools associated with it. Unfortunately the under lying architecture had a real hard time accommodating the usage model. So I started from scratch with the following data flow architecture.

Users wanted:

- Multiple clients (Aves, Probe…) accessing the DUT at the same time.
- IronPython accessing to the DUT while the other tools are up and running
- New users want to use other languages that cannot be integrated into AvesBlue.
- A console where users could enter single line command to be executed
- Block reads and writes specifically for SPI interfaces but good for I2C (FW loads).

I wanted:

- Break apart Aves so one program does not do it all. Divide and conquer.
- Keep legacy functionality
- Create an architecture that allows IPC connections instead of just direct connections.
- Allow any language to access a server for DUT access.

The following diagrams illustrate one solution for all these constraints.

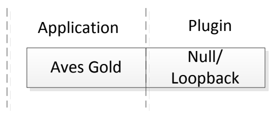


Figure 1 App/Null

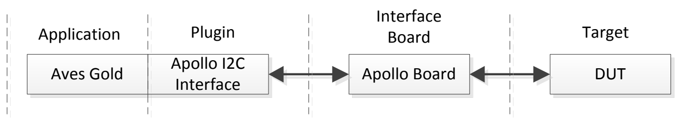


Figure 2 App/Plugin + Hardware + DUT

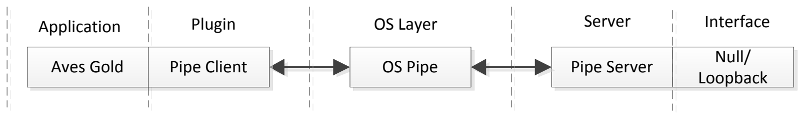


Figure 3 App/Plugin + Pipe + Server/Null

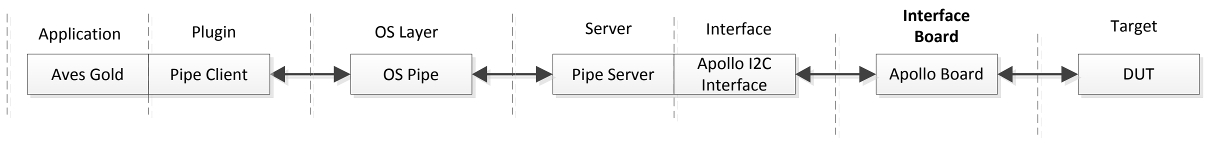


Figure 4 App/Plugin + Pipe + Server/Interface + Hardware + DUT

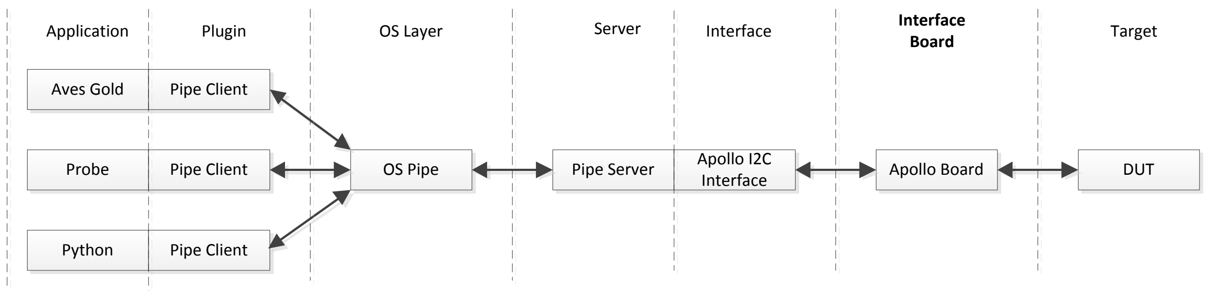


Figure 5 Multi App/Plugin + Pipe + Server/Interface + Hardware + DUT

Figure 1 above shows a topology using a simple Null/Loopback plugin for testing purposes. The Null/Loopback plugin executes the queries sent to it and returns the appropriate response. The Null plugin has a 256 x 65536 8-bit array to emulation the full address space of the command set.

Figure 2 above shows a topology similar to what we expect from AvesBlue where the application uses an Apollo plugin and the Apollo board connects to the target.

Figure 3 above shows a topology like Figure 2 except the application plugin is a pipe connection and a pipe Server with Null/loopback is started as a discrete application. Once both application and server are up and running it should perform the same way Figure 1 performs.

Figure 4 above is the same as the topology as Figure 3 except the Server plus Null/Loopback has been replaced by the Server plus Apollo board interface plus Apollo board plus DUT. This topology is basically the Application communicating with the DUT through a pipe.

Since the OS pipe is a standard IPC method and can handle multiple client connections, Figure 5 above shows a topology where multiple application can access the DUT. The accesses are time shared between all the applications, not in parallel.

Using a pipe to communicate with the DUT allows any number of types of applications to connect to the DUT as long as the application can connect with the pipe. This allows the developers to create any level complexity application using any language they prefer.

Also using the pipe to communicate with the DUT allows for different server/interfaces to be used so the final connection to the DUT can be through either Apollo board, SDP board, Aardvark or hardware of their choice in either I2C for SPI formats. It's just a matter of creating a new Server/Interface program as long as the server connects to the pipe. There would be no changes in the applications to support new hardware.

Another benefit to this topology, the pipe could be replaced with any method of IPC. All that would need to be created are a client and server for the new IPC. This bring up an interesting possible option of a TCP based IPC. This would allow Aves Gold running in Austin to connect to a DUT running in Limerick while Limerick is accessing the same DUT. (Webex at a hardware level ☺)

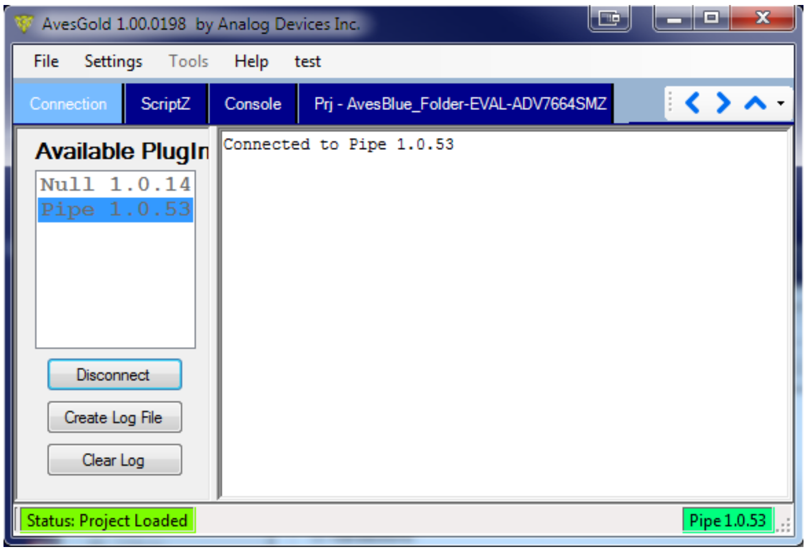
Basically I wanted a Lego style bits and pieces that I can just plug together what I need to make an operational system. Keep all the individual program as small and simple as possible.

Another benefit with rewriting Aves is to change the program to a true MVC model implementation. AvesBlue had several structural problems with keeping all the pop out windows synchronized and implementing the proxy servers that external Python scripts can connect to.

AVESGold

Aves Gold is a tool that performs similar to the AvesBlue program except with the Probe and Python portions removed.

Below is the AvesGold start up screen already connected to the pipe server and one project loaded.



```
Connected to Pipe 1.0.53
```

Status: Project Loaded

Pipe 1.0.53

Figure 6 AvesGold Start up screen

Initially the start screen consists of 3 or 4 tabs, Connection, ScriptZ, Console and a Project (Project may or may not be present depending on the state of the previous exit). As projects and devices are added more tabs will be added to contain the new information. The menu is self-explanatory.

The tab line has three blue arrows. The left and right arrows are used when there are more tabs then the line can handle. These 2 arrows act as a slider to bring the tabs into view. The up arrow will pop the selected tab out to a free floating window. Free floating windows can be returned to the main program by just closing them with top right ‘red X’. Due to the MVC nature of this program all pop out widows will be synchronized to the model data.

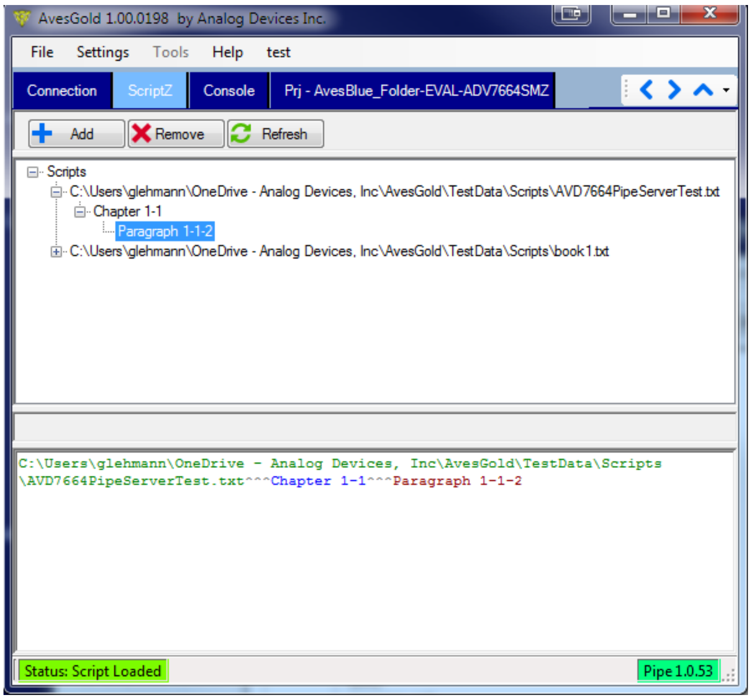
The Status box in the bottom left corner indicates what state the program is in as it runs and load projects and devices. The bottom right corner box shows the active plugin and whether it’s still connected or not (green/gray).

Connect Tab

Referencing the above image the connect tab contains a list of available plugins and a transaction list. Just select a plugin and hit the ‘Connect’ button. The ‘Create Log File’ will copy the contents of the transaction window to a file. The ‘Clear Log’ will erase the transaction window.

ScriptZ Tab

The ScriptZ tab allows loading and running of various script files.



```
Scripts
  C:\Users\glehmann\OneDrive - Analog Devices, Inc\AvesGold\TestData\Scripts\AVD7664PipeServerTest.txt
    Chapter 1-1
      Paragraph 1-1-2
  C:\Users\glehmann\OneDrive - Analog Devices, Inc\AvesGold\TestData\Scripts\book1.txt
```

```
C:\Users\glehmann\OneDrive - Analog Devices, Inc\AvesGold\TestData\Scripts
\AVD7664PipeServerTest.txt^^^Chapter 1-1^^^Paragraph 1-1-2
```

Figure 7 AvesGold ScriptZ Tab

The ‘Add’ button will add script files to the first panel.

The ‘Remove’ button will remove the selected script file from the first panel

The ‘Refresh’ button will reload all top level scripts.

Double clicking on any paragraph will run that script and place a link to in the bottom panel.

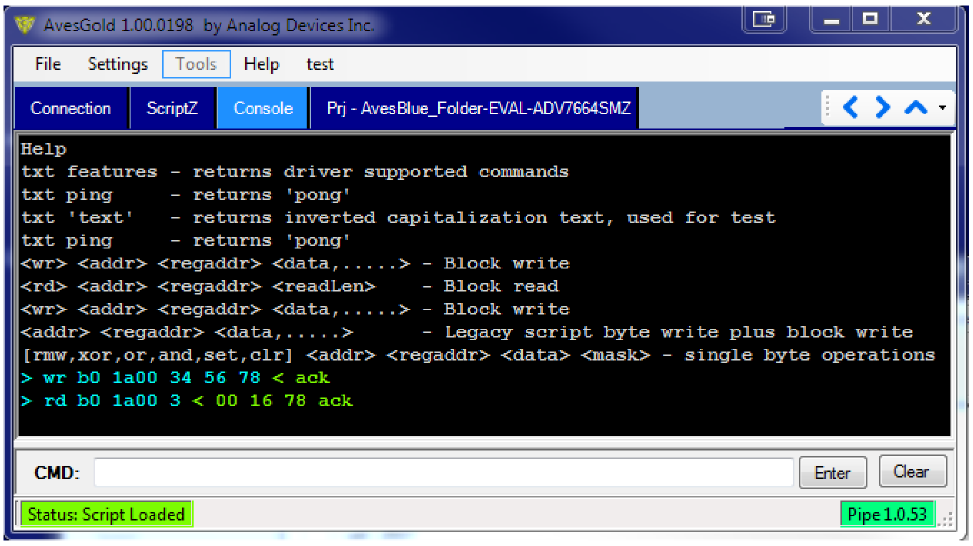
Double clicking on the link in the bottom will also run script.

If the script is edited by an external program, any changes saved by the editor will be reloaded in this tab. Only the top level scripts are watched for changes. ‘Include’ paragraphs are not automatically reload upon change, therefore the need for the ‘Refresh’ button which re-loads and validates all scripts.

All scripts files are validated before they are allowed to load into the script database. Some parsing error information is provided.

Console Tab

The console tab allows the user to enter any ScriptZ sentence or single line command.



```
Help
txt features - returns driver supported commands
txt ping     - returns 'pong'
txt 'text'   - returns inverted capitalization text, used for test
txt ping     - returns 'pong'
<wr> <addr> <regaddr> <data,.....> - Block write
<rd> <addr> <regaddr> <readLen>    - Block read
<wr> <addr> <regaddr> <data,.....> - Block write
<addr> <regaddr> <data,.....>      - Legacy script byte write plus block write
[rmw,xor,or,and,set,clr] <addr> <regaddr> <data> <mask> - single byte operations
> wr b0 1a00 34 56 78 < ack
> rd b0 1a00 3 < 00 16 78 ack
```

Figure 8 AvesGold Console Tab

The console box contains color coded text. In this example the light grey color implies informational text. The aqua color is the outbound information that when through the plugin. The green color is the information that returned from the plugin.

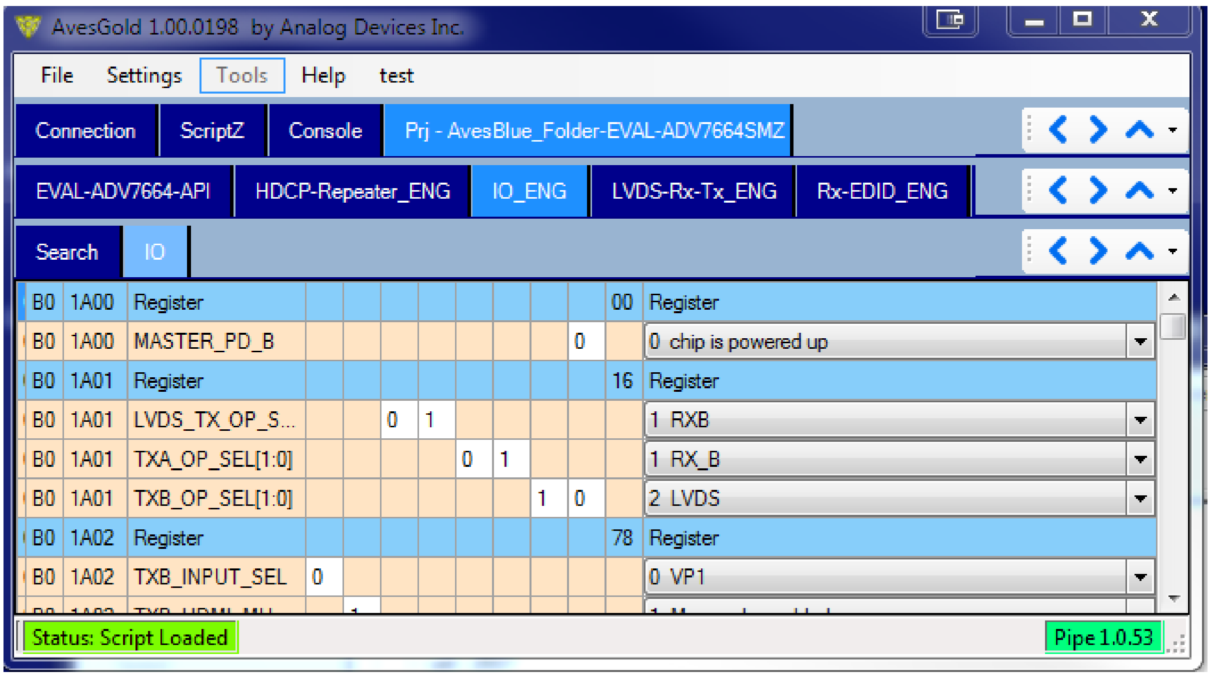
The command box is where you enter the command.

The ‘Enter’ button or Enter in the command box executes the command.

The ‘Clear’ button clears the console box.

Project Tab

The project tab has contains the register/field maps defined be the device XML file.



B0	1A00	Register									00	Register
B0	1A00	MASTER_PD_B								0		0 chip is powered up
B0	1A01	Register									16	Register
B0	1A01	LVDS_TX_OP_S...			0	1						1 RXB
B0	1A01	TXA_OP_SEL[1:0]					0	1				1 RX_B
B0	1A01	TXB_OP_SEL[1:0]							1	0		2 LVDS
B0	1A02	Register									78	Register
B0	1A02	TXB_INPUT_SEL	0									0 VP1

Figure 9 AvesGold Project TAB

Operation and interactions with the fields, bits and meta-fields are the same as previous versions of Aves so I will not go into any deep description here.

Settings

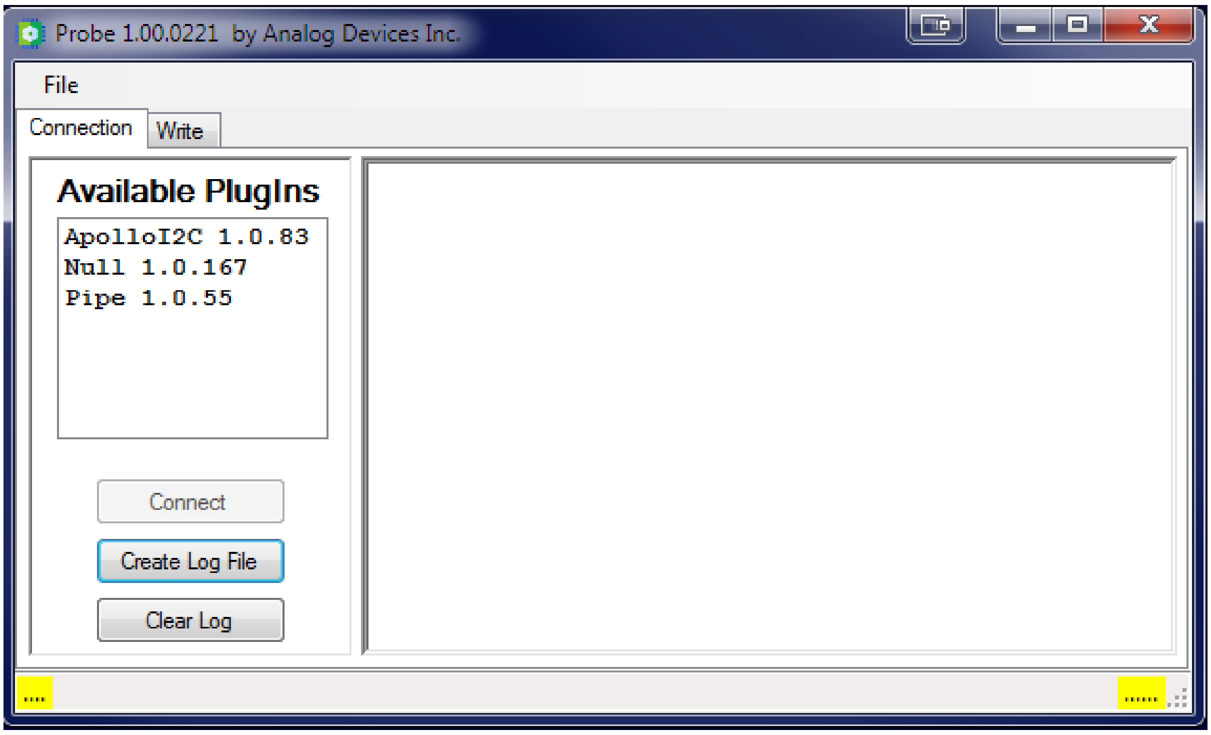
Preferences

Allows you to select various load and color options.

Probe

The probe is a standalone application to replace the probe feature that was in AvesBlue. The probe can only do all reads or all writes (slightly different from AvesBlue's probe function). However since we are now using IPC to connect to the target we can bring up multiple instances of the probe each one doing only reads or only writes.

Figure 10 below shows the initial start-up screen of the Probe. Here you are able to select this connection path to the target, monitor/save log files, save/load probes files or create new ones. Overall the look and operation is very similar to the AvesGold application.



ApolloI2C 1.0.83
Null 1.0.167
Pipe 1.0.55

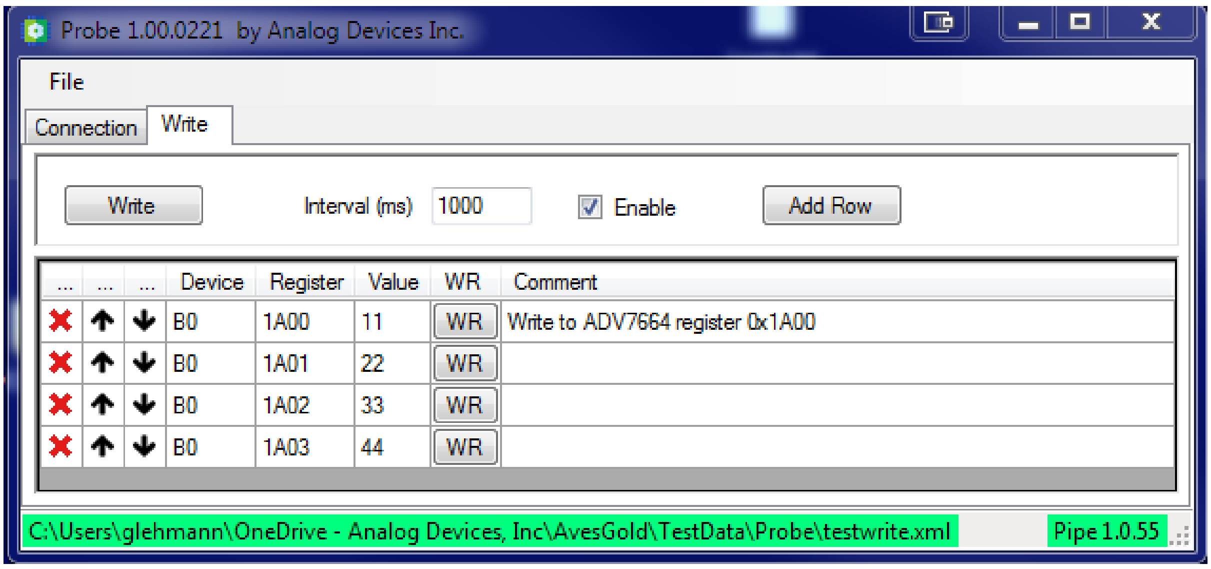
Figure 10 Probe Startup Screen

Figure 11 below shows the program with the Write tab loaded and visible. The top panel contains global controls

- Write button – forces all registers to be written out
- Interval text box – sets the write interval
- Enable check box – enable automatic global write at the defined interval
- Add Row button – add a new to the data grid

The bottom panel contains individual register read lines

- 1st column – Delete the line
- 2nd column – Move line up one
- 3rd column – Move line down one
- 4th column – Sets device address
- 5th column – Sets register address. If register < 0x100 then transaction will use 8-bit register addressing only. If register >= 0x100 then transaction will us 16-bit register addressing.
- 6th column – Sets value to write out
- 7th column – Write button that writes this line only
- 8th column – Comment for this line



Probe 1.00.0221 by Analog Devices Inc.

File

Connection | Write

Write | Interval (ms) 1000 | ☑ Enable | Add Row

...	...	...	Device	Register	Value	WR	Comment
✗	↑	↓	B0	1A00	11	WR	Write to ADV7664 register 0x1A00
✗	↑	↓	B0	1A01	22	WR	
✗	↑	↓	B0	1A02	33	WR	
✗	↑	↓	B0	1A03	44	WR	

C:\Users\glehmann\OneDrive - Analog Devices, Inc\AvesGold\TestData\Probe\testwrite.xml Pipe 1.0.55

Figure 11 Probe Write Tab

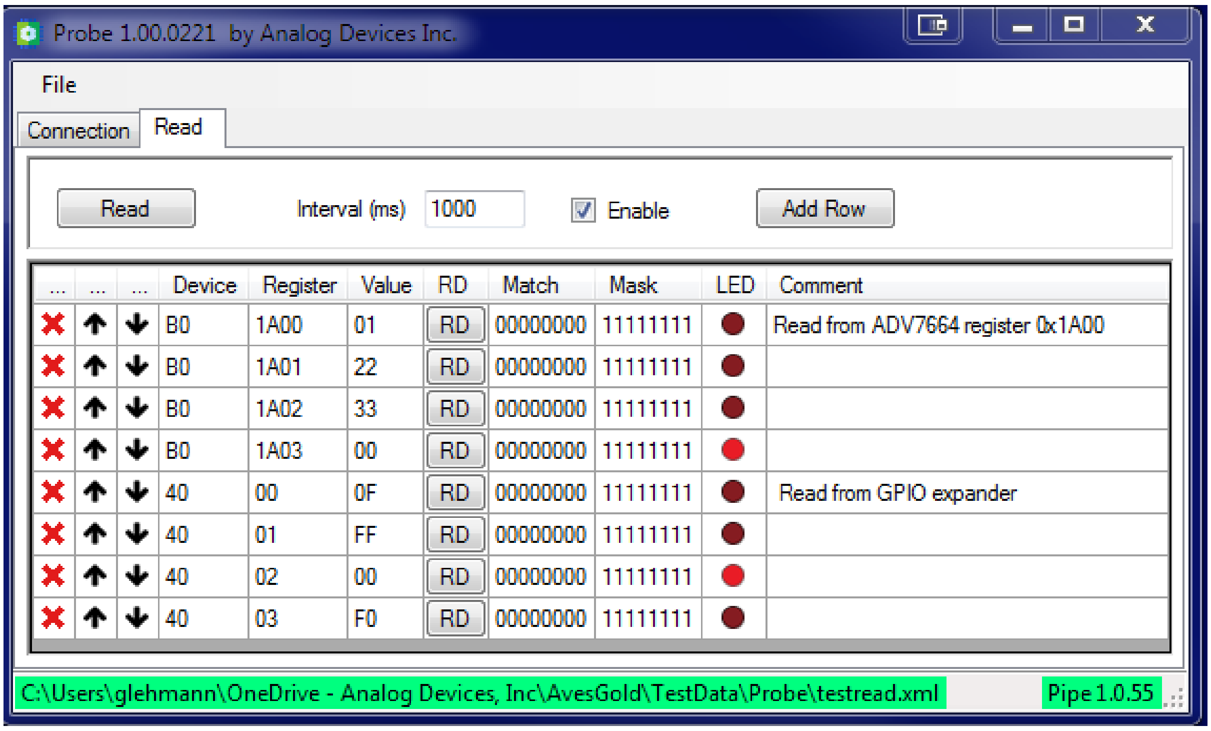
Figure 12 below shows the program with a read tab and loaded probe file. This tab is mostly the same as the write tab however three extra columns have been added

- Match column – contains byte we want to match to
- Mask column – contains a mask to use against Match and Value
- LED – indicator that a match has occurred

The LED will light up when the following conditions are met:

LED on = (Value & Mask) == (Match & Mask)

This allows you to easily check for specific bit states.



Probe 1.00.0221 by Analog Devices Inc.

File

Connection | Read

Read | Interval (ms) 1000 | ☑ Enable | Add Row

...	...	...	Device	Register	Value	RD	Match	Mask	LED	Comment
✗	↑	↓	B0	1A00	01	RD	00000000	11111111	●	Read from ADV7664 register 0x1A00
✗	↑	↓	B0	1A01	22	RD	00000000	11111111	●	
✗	↑	↓	B0	1A02	33	RD	00000000	11111111	●	
✗	↑	↓	B0	1A03	00	RD	00000000	11111111	🔴	
✗	↑	↓	40	00	0F	RD	00000000	11111111	●	Read from GPIO expander
✗	↑	↓	40	01	FF	RD	00000000	11111111	●	
✗	↑	↓	40	02	00	RD	00000000	11111111	🔴	
✗	↑	↓	40	03	F0	RD	00000000	11111111	●	

C:\Users\glehmann\OneDrive - Analog Devices, Inc\AvesGold\TestData\Probe\testread.xml Pipe 1.0.55

Figure 12 Probe Read Tab

General Comments

- These applications are designed to handle block reads and writes however the user must keep in mind that some devices do not implement auto-increment addresses so block reads and writes will not work for them. Script files must keep this in mind when writing to devices like the PCA9554.
- It is possible to run two probes using direct access to the same target via the Apollo interface. However what is happing is the USB connection to the Apollo board is being shared at a relatively high swapping rate. This can and does cause crashes. Using 2 USB connections to the same target is not recommended. The pipe solution handles this topology better.
- With the Apollo/pipe implementation I have tested transaction rates up to 120 transactions/second. There is a built-in delay in the Apollo driver which limits the maximum transaction rate to 200/second. I will be examining this limit in the future but the current setting is safe. This 200 transactions/second limit applies to both direct connections and pipe connections.
- The XML packet is designed to transfer multiple transactions within one packet. AvesGold and Probe only implement one transaction per packet. This is on the queue to be re-examined also.
- More clients and servers will be created on a priority/time bases.
- Current the pipe defaults to 'AvesPipe' however any pipe name can be used with any application or server. This allows the topology to work with multiple pipes.

FAQ

Appendix A – ScriptZ

ScriptZ is a superset of ScriptX implemented for AvesBlue. ScriptX is a superset of the original Script language used by XRC and Aves3. ScriptZ is still a line based scripting language with added feature to handle block read and writes.

Syntax

A script file can be thought of as a book containing chapters, paragraphs and sentences. This analogy makes it easier to understand the script file format and is used within the program to syntax check and run the scripts.

Definitions:

Book	a script file comprised of multiple chapters. Referenced by filename.txt
Chapter	multiple Paragraphs or Pages delineated with a '##', Remaining text of the line is the chapter name. Chapters are ended by the next Chapter marker or end of file (EOF)
Paragraph	set of sentences. Delineated with a ':' in first column, remaining text of the line is the paragraph name. Paragraphs are ended with 'End'. Also known as one script, the unit that runs when selected.
This	refers to this file/book, not an external file/book
Sentence	one action. Delineated as one line in the file. Mask is optional will default to 0xFF.

****example sentences****

Single write,	EX. 72 1 0 ; single byte write
Block write,	Ex. 72 1 23 45 67 ; 3 byte write
Block write,	Ex. wr 72 1 23 45 67 ; same as above
Block read,	Ex. rd 72 1 2; not useful in script files
ReadModifyWrite,	Ex. rmw 72 22 <data> <mask>
XOR,	Ex. xor 72 23 <data> <mask>
OR,	Ex. or 72 34 <data> <mask>
AND,	Ex. and 72 45 <data> <mask>
SET,	Ex. set 72 56 <mask>
CLR,	Ex. clr 72 67 <mask>

Include … … … ; see below

Delay 1000 ; see below

Line types:

';' comments, all characters after ';' are ignored regardless of position within the line

'##' begins chapters, remainder of the line is the chapter name, a book can have multiple chapters

':' begins a paragraph, remainder of the line is the paragraph name, each paragraph is terminated by 'End' line. A chapter can have multiple paragraphs

#SETTINGS# and #REVISION# paragraphs will be ignored regardless of their location

'include <[book,this]> <chapter> <paragraph>', 3 arguments, this line will parse into scripts/books to find and run the specified chapter:paragraph.

'delay 1000', delays processing by 1000 ms

'end' of paragraph or any '##' region

Line Rules:

Blank lines are ignored

Leading and trailing white spaces are removed before parsing

All actions are case insensitive

Chapter, and Paragraph names can be quoted with ' " ' to include white spaces.

All lines before the first chapter are ignored

'delay' and 'include' are only valid within a paragraph and are case insensitive

'this' refers to the same file this script is in and is case insensitive

In register write sentences, 2nd argument, 4 hex digits implies 16 bit register indexes, 2 hex digits implies 8 bit register indexes.

For legacy scripts '##' will be converted to '#' before processing.

****this is an example book/script file****

```
;Book1
descriptive text

##SETTINGS##
End

#Chapter 1-1# ; comment
:Paragraph 1-1-1:
90 01 00 01 02 03; abc
91 0123 45 ; xyz
End

:Paragraph 1-1-2:
92 01 55
delay 1000
include this "Chapter 1-3" "Paragraph 1-2-1"
include book2.txt "Chapter 2-1" "Paragraph 2-1-1"
rmw 72 22 3C 0C        ; change register bits[3:2]
xor 72 23 3C FF        ; toggle register bits[5:2]
or 72 34 0c 0c         ; set register bits[3:2]
and 72 45 0c 3f        ; and register bits[5:2] with data[5:2]
set 72 67 c0           ; sets register bits[7:6]
clr 72 67 c0           ; clears register bits[7:6]
wr 92 04 55 66 77      ; dev=94, regAddr=04, write 3 byte block
rd 92 04 3             ; dev=94, regAddr=04,reads 3 byte block, useless in script file
End

##Chapter 1-3##
:Paragraph 1-2-1:
95 01 22
95 01 33
End
```

Appendix B – XML packets

When client/server connection method are used all communications are done via XML packets in ASCII string format. This allows easier packet debugging. The client creates a query packet and the server will return a response packet. A transaction is defined as a set of query/response packets. If the client does not receive a response within a timeout period, the client shall assume it is a NACK response.

Example of a query packet:

```
<aves>
        <version>1.0</version>
        <originator>Pipe 1.0.53</originator>
        <timestamp> 2108.07.26.02.35.880</timestamp>
        <uid>2</uid>
        <transactions>
                <transaction>
                        <cmd>txt</cmd>
                        <dev>0</dev>
                        <addr>0</addr>
                        <addrsize>8</addrsize>
                        <mask>0</mask>
                        <rdlen>0</rdlen>
                        <wrdata>0</wrdata>
                        <rddata>0</redata>
                        <query>features</query>
                        <response></response>
                </transaction>
                <transaction>
                        …
                </transaction>
        </transactions>
</aves>
```

Tag definitions:

```
'aves'         - root tag
'version'      - defines packet version, optional
'originator'   - program that created the packet, optional
'timestamp'    - utc timestamp of packet creation, optional
'uid'          - unique identifier, optional
'transactions' - wraps a group of transactions, required
'transaction'  - wraps on transaction, at least one is required
'cmd'          - transaction type, required, can be 'txt rd wr rmw xor or and set clr'
'dev'          - I2C/SPI chip/bus address, required for any byte transaction
'addr'         - register address, required for any byte transaction
'addrsize'     - defines 8/16 bit register addressing, required 8/16 for byte transactions
'mask'         - defines mask used with binary operations. Required for binary operations
'rdlen'        - block read length, required for block reads
'wrdata'       - write data block, required for write blocks, Ex. '55 66 77'
'rddata'       - returned block reads, Ex. '44 55 66 77'
'query'        - query part of 'cmd'==txt, optional, currently can be 'ping' or 'features'
'response'     - contains response to a query, known responses are 'pong' or 'feature list'
```

Comments

Binary operators, "rmw,xor,or,and,set,clr" require the mask and one byte in the 'wrdata' tag.

var d = read_dut_byte()

var x = (d & ~mask) | (d & mask) binOp wrdata[0]

write_dut_byte(x)

The response pack will be a clone of the query packet except the appropriate 'rddata' and 'response' fields filled in with the correct information.

All numbers are hexadecimal.

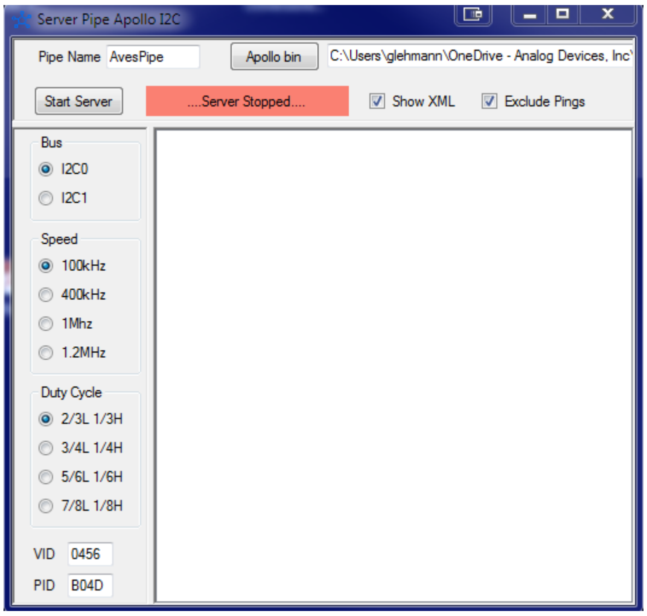
All non-txt response packets will contain ack/nack text in the 'response' tag indicating the status of the transaction with regards to the server interaction with the DUT.

'transactions' can contain multiple 'transaction'. All server processing will handle one 'transaction' at a time and will return the response packet with cloned multiple 'transaction'. Multiple 'transaction' packet will become valuable if/when we create TCP servers. Transaction speed appears to be fast enough for pipe communications.

In the C# environment, these packets are represented as classes and use XML serialization/deserialization to easily convert from XML packets in objects the program can easily work with. Other languages should have similar functionality to create/process these ASCII based XML packets.

Appendix C – Server Pipe Apollo I2C

The ServerPipeApolloI2c is a standalone program that interfaces between an OS pipe and Apollo hardware (041317) generating I2C transactions with the DUT. The image below shows what the start-up screen looks like.


Server Pipe Apollo I2C

Pipe Name: AvesPipe | Apollo bin: C:\Users\glehmann\OneDrive - Analog Devices, Inc\

Start Server | ....Server Stopped.... | ☑ Show XML | ☑ Exclude Pings

Bus: [✓ I2C0] I2C1

Speed: [✓ 100kHz] 400kHz 1Mhz 1.2MHz

Duty Cycle: [✓ 2/3L 1/3H] 3/4L 1/4H 5/6L 1/6H 7/8L 1/8H

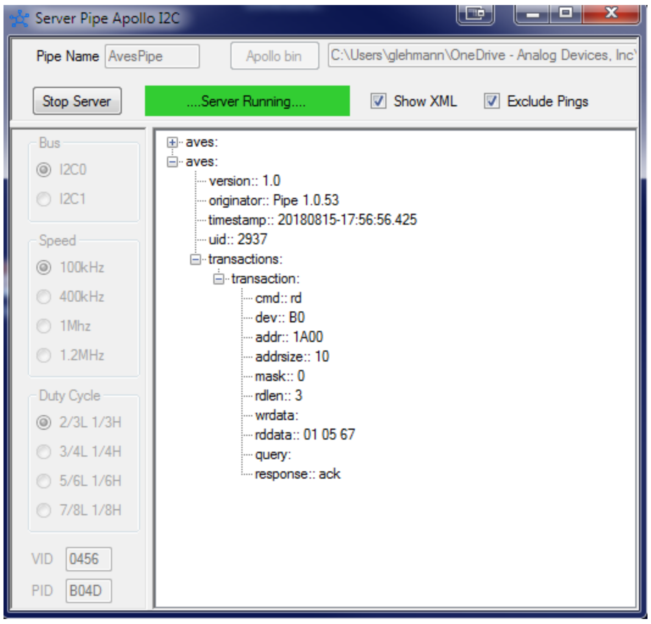
VID: 0456

PID: B04D

Figure 13 Server Pipe Apollo

The left panel sets the configuration required by the Apollo driver. The top panel sets the location of the Apollo bin file to load. Normally leave this to the default location. A default bin will be included in the installer. The Pipe Name box allows you to change the referenced pip name. For now leave it to default. The Start Server button begins the server. The Exclude Pings check box reduces the number of transactions inserted into the tree view. The Show Xml check box, when checked inserts all transactions into the tree view.

Below is the screen after the server has started running and several commands from the client have been handled.


Server Pipe Apollo I2C

Pipe Name: AvesPipe | Apollo bin: C:\Users\glehmann\OneDrive - Analog Devices, Inc\

Stop Server | ....Server Running.... | ☑ Show XML | ☑ Exclude Pings

Bus: [✓ I2C0] I2C1

Speed: [✓ 100kHz] 400kHz 1Mhz 1.2MHz

Duty Cycle: [✓ 2/3L 1/3H] 3/4L 1/4H 5/6L 1/6H 7/8L 1/8H

VID: 0456

PID: B04D

```
⊞ aves:
⊟ aves:
    version:: 1.0
    originator:: Pipe 1.0.53
    timestamp:: 20180815-17:56:56.425
    uid:: 2937
  ⊟ transactions:
    ⊟ transaction:
        cmd:: rd
        dev:: B0
        addr:: 1A00
        addrsize:: 10
        mask:: 0
        rdlen:: 3
        wrdata:
        rddata:: 01 05 67
        query:
        response:: ack
```

Figure 14 Server Pipe Apollo running

The green box indicates the server has connected to the Apollo board, updated the Apollo board's FW, created the pipe and is ready to accept transactions over the pipe.

The tree view shows 2 transaction. The open transaction shows a tree view version of the server response from the console command “rb b0 1a00 3” or reading a 3 byte block starting at B0:1A00. The tree view is mostly meant for debugging.

Normally the server is configured, started and ignored with both Show XML unchecked and Exclude Pings checked. All setting are persistent. The client should periodically ‘ping’ the server to make sure it’s still running. Currently I have the client ping rate set for every 5 seconds.

Any program that can connect to the pipe and generate/decode XML packets should be able to use this server.