Verification Continuum™

ZeBu® Lauterbach TRACE32 JTAG Transactor User Manual

Version S-2022.06, June 2022

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at *http://www.synopsys.com/company/legal/trademarks-brands.html*.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

List of Tables

List of Figures

About This Manual

This manual describes how to use the ZeBu JTAG T32 Transactor package with your design being emulated with ZeBu.

Related Documentation

The following table lists the reference document names and their availability.

TABLE 1 Related Documentation

Document Name	Description
ZeBu Release Notes	Provides information about the ZeBu supported features and limitations. Available in the ZeBu documentation package corresponding to your software version.
Using Transactors	Provides relevant information about the usage of the present transactor. Available in the training material.

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1 Introduction

This section explains the following topics:

- *Overview*
- *Features*
- *Requirements*
- *Limitations*

1.1 Overview

ZeBu JTAG T32 transactor (xtor_jtag_t32_svs) can be used to connect a SOC running in ZEBU emulator with Lauterbach TRACE32 debugger. As per below setup TRACE32 gets connected to the transactor via TCP/IP.

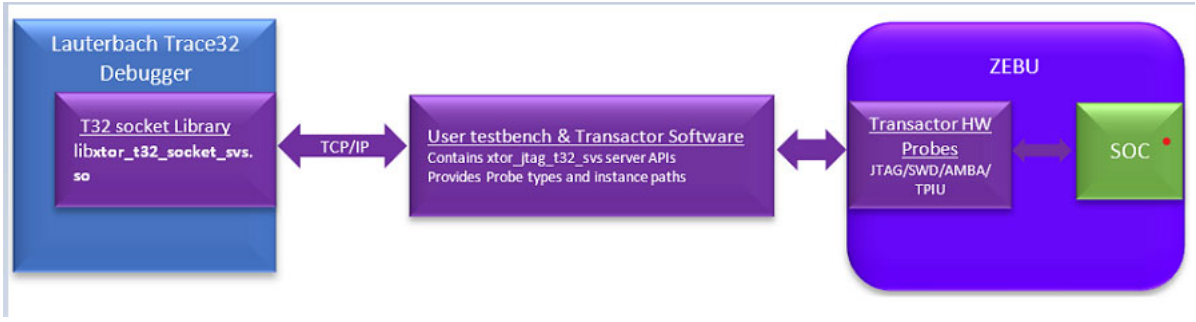

FIGURE 1. ZeBu JTAG T32 Transactor setup Overview

Lauterbach TRACE32 uses the 'T32 socket library' or 'libxtor_t32_scoket_svs.so' to connect with the xtor_jtag_t32_svs server running in the testbench. The library is part of the xtor_jtag_t32_svs package and it should be pointed correctly on your TRACE32 practice script (.cmm file).

1.2 Features

- xtor_jtag_t32_svs is a ZEMI3 transactor
- Supports TRACE32 version above **S.2015.07**
- Dynamic connection and disconnection to the debugger at runtime
- Simulation support (Please check example README for supported configurations)
- Following AP/DP interfaces are supported
  - JTAG-DP
  - SWD-DP
  - JTAG-SWD-DP
  - APB-AP
  - AHB-AP
  - AXI-AP
- Supports tracing of AMBA Trace Bus Interface with TPIU probe interface
- Multiple probes in a single T32 session are supported.
  - Probes are categorized into four independent probe classes.
  - Any combination of Probe Classes is supported.
  - Select any one interface from the list of supported interfaces for each probe class
  - More information is available in xtor_jtag_t32_svs.hh and example README

TABLE 2 Probe class and supported interfaces

Probe Class	Supported Probe Interfaces
JTAG Class	JTAG
DAP-1 class	SWD APB AHB AXI-3 AXI-4

TABLE 2 Probe class and supported interfaces

DAP-2 class	SWD APB AHB AXI-3 AXI-4
TPIU class	TPIU

- Multiple debugger sessions to multiple instances of T32 transactor instances are supported
- JTAG probe features
  - Configurable TCK/CPU clock ratio
  - Configurable TRSTn optional port
  - Configurable SRSTn optional port
  - Configurable RTCK optional port
  - Supports sending switch sequence to move design from SWD to JTAG. Dormant based and non-dormant based switch sequences are supported.
- SWD probe features
  - Supports sending switch sequences to move design from JTAG to SWD. Dormant based and non-dormant based switch sequences are supported.
  - Supports sticky overrun behavior
- AXI probe features
    - Supports 32/64 bus widths

1.3 Requirements

Transactor requires hw_xtormm_jtag FLEXnet license feature.

1.4 Limitations

2 Installation

This section explains the following topics:

- *Installing the xtor_jtag_t32_svs Transactor Package*
- *Package Structure and Content*

2.1 Installing the xtor_jtag_t32_svs Transactor Package

To install the JTAG T32 transactor package, ensure that the WRITE permissions on the IP directory and the current directory.

Download the transactor compressed shell archive (.sh) and install the package using the following command:

```
sh xtor_jtag_t32_svs.<version>.sh install [ZEBU_IP_ROOT]
```

For more information on how to install the transactor package, see ZeBu Vertical Solutions User Manual.

2.2 Package Structure and Content

After xtor_jtag_t32_svs transactor is successfully installed, it consists of the following items

- Shared library of the transactor (lib directory)

  $ZEBU_IP_ROOT/lib
- Header files of the transactor (include directory)

  $ZEBU_IP_ROOT/include
- Hardware module definitions

  $ZEBU_IP_ROOT/vlog
- Remote Validator tool executable

  $ZEBU_IP_ROOT/bin

3 Hardware Interface

This section explains the following topics:

- *JTAG probe*
- *SWD probe*
- *AMBA probes*
- *TPIU probe*
- *Examples*

3.1 JTAG probe

Module Name: `xtor_t32Jtag_svs`

Source File: `$ZEBU_IP_ROOT/vlog/vcs/xtor_t32Jtag_svs.v`

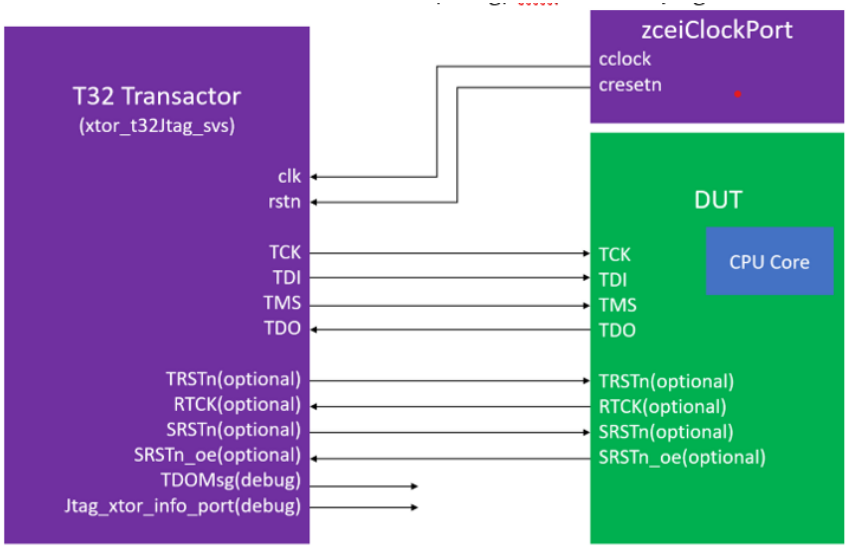

FIGURE 2. ZeBu JTAG T32 JTAG Probe Hardware Interface

Following table lists the signals in the transactor's hardware interface:

TABLE 3 Signals List for JTAG probe

Signal	Direction	Description
clk	Input	Clock input for the transactor
rstn	Input	Reset input for the transactor, active low
TCK	Output	JTAG Clock: Output of transactor
TDI	Output	JTAG Test Data In
TMS	Output	JTAG Test Mode Select
TDO	Input	JTAG Test Data Out
TRSTn	Output	JTAG TAP reset (active low) (Optional)
SRSTn	Input	System Reset detection (Optional)
SRSTn_oe	Output	System Reset output enable (Optional)
RTCK	Input	Optional input port to transactor to connect RTCK output of CPU (Optional)
TDOMsg	Output	Last 64 TDO bits buffered (Optional)
jtag_xtor_info_port	Output	Transactor debug port

3.2 SWD probe

Module Name: `xtor_t32Jtag_swd_svs`

Source File: `$ZEBU_IP_ROOT/vlog/vcs/ xtor_t32jtag_swd_svs.v`

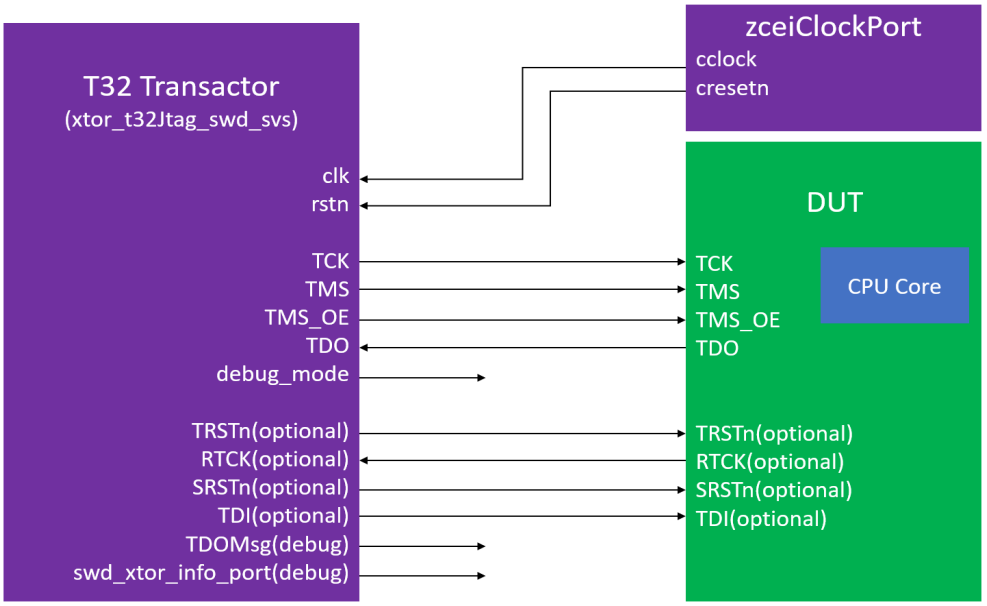

FIGURE 3. ZeBu JTAG T32 SWD Probe Hardware Interface

TABLE 4 Signals List for JTAG probe

Signal	Direction	Description
clk	Input	Clock input for the transactor
rstn	Input	Reset input for the transactor, active low
TCK	Output	Protocol clock generated by transactor
TMS	Output	SWD data out from transactor.
TMS_OE	Output	Indicates whether TMS generated by transactor is valid
TDO	Input	SWD data in to transactor. Used when TMS_OE is LOW

TABLE 4 Signals List for JTAG probe

Debug_mode	Output	Indicates whether XTOR is in JTAG(0) or SWD(1) mode
TRSTn	Output	JTAG TAP reset (Optional, valid for JTAG mode)
SRSTn	Output	System Reset detection (Optional, valid for JTAG mode)
TDI	Output	JTAG Test Data In (Optional, valid for JTAG mode)
RTCK	Input	Optional input port to transactor to connect RTCK output of CPU (Optional, valid for JTAG mode)
TDOMsg	Output	Last 64 TDO bits buffered (Optional, valid for JTAG mode)
swd_xtor_info_port	Output	Transactor debug port

3.3 AMBA probes

AMBA probes have the standard AMBA port interfaces and notations derived from ZEMI3 AMBA transactors.

TABLE 5 AMBA probe types

Intf	**T32 module name**	**AMBA xtor module name**
APB	xtor_t32apb_master_svs	xtor_apb_master_svs
AHB	xtor_t32ahb_master_svs	xtor_ahb_master_svs
AXI3	xtor_t32amba_master_axi3_svs	xtor_amba_master_axi3_svs
AXI4	xtor_t32amba_master_axi4_svs	xtor_amba_master_axi4_svs

3.4 TPIU probe

Module name: `t32Tpiu_dpi_driver`

Source file: `$ZEBU_IP_ROOT/vlog/vcs/t32Tpiu_dpi_driver.v`

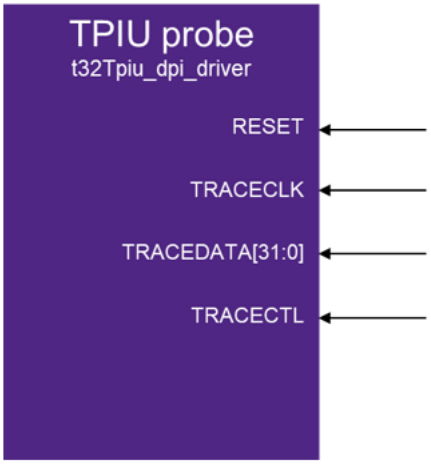

FIGURE 4. TPIU probe driver

TABLE 6 Signals List for TPIU probe

Pin	Type	Description
RESET	input	Active high reset
TRACECLK	input	Trace port clock provided by DUT
TRACEDATA[31:0]	input	Trace port data provided by DUT
TRACECTL	input	Trace port data valid provided by DUT

3.5 Examples

There are separate hardware top files for different use models of probes present in the area `$ZEBU_IP_ROOT/xtor_jtag_t32_svs/example/TapDap/src/env/`

Please check any of below files for reference.

- xtor_ahb_top.v
- xtor_ahb_vcs_top.v
- xtor_apb_top.v
- xtor_apb_vcs_top.v
- xtor_axi3_64_top.v
- xtor_axi3_64_vcs_top.v
- xtor_axi4_64_top.v
- xtor_axi4_64_vcs_top.v
- xtor_jtag_a72_vcs_top.v
- xtor_jtag_apb_ahb_top.v
- xtor_jtag_apb_apb_top.v
- xtor_jtag_apb_axi3_64_top.v
- xtor_jtag_apb_top.v
- xtor_jtag_axi3_32_top.v
- xtor_jtag_axi3_64_top.v
- xtor_jtag_axi4_64_top.v
- xtor_jtag_rtl_clk_top.v
- xtor_jtag_srst_top.v
- xtor_jtag_srst_vcs_top.v
- xtor_jtag_swd_ahb_top.v
- xtor_jtag_top.v
- xtor_jtag_tpiu.v
- xtor_jtag_vcs_top.v
- xtor_swd_a72_vcs_top.v
- xtor_swd_top.v

4 Software Interface

The ZeBu JTAG T32 transactor provides C++ APIs, which are included in the header file located in the `$ZEBU_IP_ROOT/include/xtor_jtag_t32_svs.hh` directory. Following table lists the APIs for the JTAG T32 transactor.

All APIs/Structs/Enums are part of the namespace `ZEBU_IP::XTOR_JTAG_T32_SVS`

The enum probe_t and struct t32_probes are used to define the probe types used.

```
enum probe_t
{
    T32_SWD,
    T32_APB,
    T32_AHB,
    T32_AXI3,
    T32_AXI4
};

struct t32_probes
{
    char*       probe_JTAG_path   = NULL;
    uint16_t    probe_JTAG_port   = 20010;

    char*       probe_DAP1_path   = NULL;
    uint16_t    probe_DAP1_port   = 20011;
    probe_t     probe_DAP1_type   = T32_SWD;

    char*       probe_DAP2_path   = NULL;
    uint16_t    probe_DAP2_port   = 20012;
```

```
    probe_t     probe_DAP2_type   = T32_APB;

    char*       probe_TPIU_path   = NULL;
    uint16_t    probe_TPIU_port   = 20014;
};
struct Tap_struct_cb
{
    uint64_t tdi_cb,tms_cb,tdo_cb,cyle_cb
};
```

TABLE 7 API & description

API	Description
Constructors	
xtor_jtag_t32_svs(svt_c_runtime_cfg * runtime, t32_probes probes)	Creates xtor_jtag_t32_svs transactor instance, arguments probe structure and runtime object. Returns an instance of transactor.
static xtor_jtag_t32_svs* getInstance(svt_c_runtime_cfg* runtime, t32_probes probes)	Gets the xtor_jtag_t32_svs transactor instance, arguments probe structure and runtime object.
~xtor_jtag_t32_svs()	Destructor. Closes any TCP connection and servers, clear memory
Server APIs	
void useLocalHostOnly(void)	It only use localhost IP for connection
bool runUntilReset ()	Wait for transactor to come out of reset (A blocking call)
bool startServer()	Starts the TCP servers based on the probe types provided
bool isConnected (void)	Provide the Connection status

TABLE 7 API & description

bool close()	Closes the TCP servers
void setDebugLevel(DebugVerbosity_t lvl)	Set debug level of transactor and internal probes Use level as- DEBUG_NONE (means level 0), DEBUG_LOW (means level 1), DEBUG_MEDIUM (means level 2), DEBUG_HIGH (means level 3), DEBUG_FULL (means level 4)
TPIU probe APIs	
void enableTPIUTrace(uint32_t)	Enable tracing of TPIU data
void setTPIULog(File*,uint32)	Update TPIU log output
void setTPIURawLog(File*,uint32)	Update TPIU raw log output
Bool configTPIU(uint32_t)	Update TPIU data width
Transaction Monitoring APIs	
void printJTAGMsg(bool)	Enable printing every TMS, TDI, TDO messages
void printFSMChanges(bool)	Enable tracing of JTAG FSM state changes
void printIRnDR(bool)	Prints IR and DR register changes
void printARM(bool)	Decodes IR and DR registers as per ARM architecture *Experimental feature
Callbacks	
void registerT32CB(void(*t32_CB)(void* xtor,ZEBU_IP::Tap_struct_cb t32_cb_data),void* context=NULL);	Register callback to trigger whenever a JTAG message is sent to HW. TDI,TMS,TDO and Cycles are provided back
UAPI Constructor approach	

TABLE 7 API & description

xtor_jtag_t32_svs(const char * xtorTypeName, const char * driverName, svt_c_runtime_cfg * runtime, XtorScheduler * sched, bool isDPI = true);	UAPI constructor aligned to other Xtors
Static Xtor* getUAPIInstance(const char * xtorTypeName, const char * driverName, svt_c_runtime_cfg * runtime, XtorScheduler * sched, bool isDPI = true);	UAPI get method
Static void Register (const char* xtorTypeName)	Register call to register Xtor type
Void initUAPII(t32_probes probes)	Init call to provide probe structure

The ZeBu T32 transactor provides following transactor libraries

- libxtor_jtag_t32_svs.so - Transactor library for xtor_jtag_t32_svs server
- libxtor_t32_socket_svs.so - Library which is used by the debugger to connect with ZeBu

5 Testbench Examples

Following are few commonly used testbench configurations. Please find more testbench configurations in example/TapDap/src/bench/server/

5.1 JTAG probe only

```
    ZEBU_IP::XTOR_JTAG_T32_SVS::t32_probes probes;
    ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs* t32_server;
    probes.probe_JTAG_path  =   "xtor_jtag_top.jtag";
    probes.probe_JTAG_port  =   20010;


   t32_server =
ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs::getInstance(runtime
,probes);
   t32_server ->setDebugLevel(DEBUG_LOW  );
   t32_server ->runUntilReset ();
   t32_server->startServer();
```

5.2 SWD probe only

```
    ZEBU_IP::XTOR_JTAG_T32_SVS::t32_probes probes;
    ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs* t32_server;
    probes.probe_DAP1_path  =   "xtor_swd_top.swd0";
    probes.probe_DAP1_port  =   20011;
    probes.probe_DAP1_type  =
ZEBU_IP::XTOR_JTAG_T32_SVS::T32_SWD;


   t32_server =
ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs::getInstance(runtime
,probes);
```

```
    t32_server ->setDebugLevel(DEBUG_LOW);
    t32_server ->runUntilReset();
    t32_server->startServer();
```

5.3 JTAG, APB and AHB probes

```
    ZEBU_IP::XTOR_JTAG_T32_SVS::t32_probes probes;
    ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs*  t32_server;

    probes.probe_JTAG_path  =   "xtor_jtag_apb_ahb_top.jtag";
    probes.probe_JTAG_port  =   20010;

    probes.probe_DAP1_path  =
"xtor_jtag_apb_ahb_top.apb_master_U0";
    probes.probe_DAP1_port  =   20011;
    probes.probe_DAP1_type  =
ZEBU_IP::XTOR_JTAG_T32_SVS::T32_APB;

    probes.probe_DAP2_path  =
"xtor_jtag_apb_ahb_top.ahb_master_U0";
    probes.probe_DAP2_port  =   20012;
    probes.probe_DAP2_type  =
ZEBU_IP::XTOR_JTAG_T32_SVS::T32_AHB;

    t32_server =
ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs::getInstance(runtime
,probes);
    t32_server ->setDebugLevel(DEBUG_LOW);
    t32_server ->runUntilReset();
    t32_server->startServer();
```

5.4 JTAG and TPIU probes

```
    ZEBU_IP::XTOR_JTAG_T32_SVS::t32_probes probes;
    ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs* t32_server;
    FILE* fd = fopen("tpiu_trace.log","w");

    probes.probe_JTAG_path  =   "xtor_jtag_tpiu_top.jtag";
    probes.probe_JTAG_port  =   20010;

    probes.probe_TPIU_path  =   "xtor_jtag_tpiu_top.u_tpiu";
    probes.probe_TPIU_port  =   20013;

    t32_server =
ZEBU_IP::XTOR_JTAG_T32_SVS::xtor_jtag_t32_svs::getInstance(runtime
,probes);

    //TPIU settings
    t32_server->enableTPIUTrace(1);
    t32_server->setTPIULog(fd, 2);
    t32_server->configTPIU(32);
    // Start both servers
    t32_server ->setDebugLevel(DEBUG_LOW);
    t32_server ->runUntilReset();
    t32_server->startServer();
```

6 Tutorial

Following section describes connecting the transactor with Lauterbach Trace32 debugger.

6.1 Getting transactor connected with the debugger

1. Integrate the transactor with your design and bring up emulation.

Note *If you wish to try transactor example, use below commands*

```
% cd $ZEBU_IP_ROOT/xtor_jtag_t32_svs/example/TapDap/zebu
% make compile ZEBU=1 CONFIG=10_JTAG
% make run_t32_server ZEBU=1 CONFIG=10_JTAG
```

Note *If you wish to try the example in simulation, use below command*

```
% make run_t32_server SIMULATOR=1 CONFIG=10_JTAG
```

You should be able to see below messages once the transactor starts TCP server.

```
VJTAG [INFO]: JTAGXTOR Daemon waiting for connection of
client socket port: 20010
```

2. Launch Trace32 debugger

Please update your PATH variable with T32 installation directory as below

```
setenv PATH <Trace32_Installation>/bin/pc_linux64/:$PATH
```

For in house tests please reach to JTAG R&D team for latest T32 installation

Once PATH is set correctly, try below command to launch the debugger.

Note *Please note that a configuration file should be provided to the debugger command with -c option. The config file content indicates to the debugger that a virtual JTAG target is being used*

```
% t32marm64 -c $ZEBU_IP_ROOT/xtor_jtag_t32_svs/example/
TapDap/src/env/config_test.t32
```

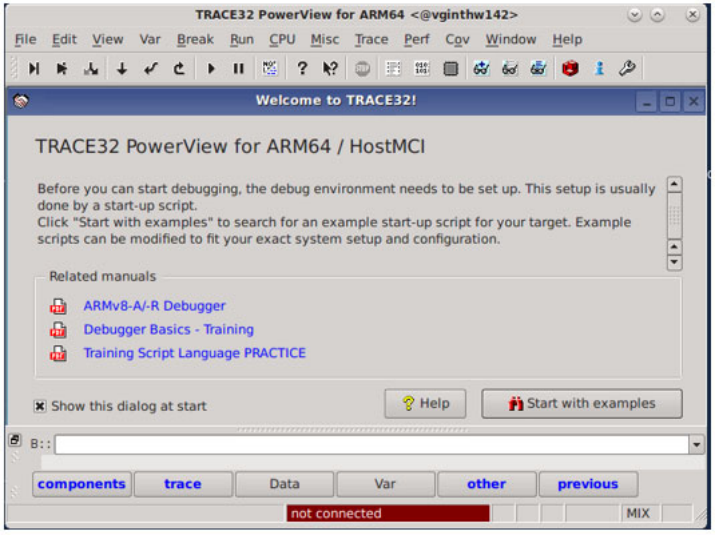

FIGURE 5. TRACE32 PowerView for ARM64/HostMCI

3. Load and execute practice script (.cmm) you wish to execute with the debugger.

Note *First few lines of the practice script should be added with the configurations to connect with the transactor. Please refer below .cmm file.*

```
; ----------must include below for Xtor connections-------
sys.CONFIG.DEBUGPORT GTL0

; ---configurations, please check table below for all configurations available.--
sys.gtl.modelconfig
"NODE=127.0.0.1|PORT_JTAG=20010|PORT_DAP0=20011|PORT_DAP1=20012|PORT_TPIU=20013|CLK
_RATIO=2"

; ------T32 socket library.-----
sys.gtl.LIBname "$ZEBU_IP_ROOT/lib/libxtor_t32_socket_svs.so" ;

; -----Probe configurations. Uncomment/modify as required-----
;sys.gtl.JTAGPROBENAME "JTAGPROBE0" ; uncomment when using JTAG probe

;SYStem.CONFIG.DAPNAME  "DAP0_SWD0" ; uncomment when using SWD probe

;SYStem.CONFIG.AXINAME "DAP000_AXI" ; uncomment when using AXI3 as DAP-1 probe

;SYStem.CONFIG.AXINAME "DAP111_AXI4"; uncomment when using AXI4 as DAP-1 probe

;SYStem.CONFIG.APBNAME "apb_DAP0"    ; uncomment when using APB as DAP-0

;SYStem.CONFIG.AHBNAME "ahb_DAP1"    ; uncomment when using AHB as DAP-0

;system.gtl.TRACENAME "TPIU"         ; uncomment when using TPIU port

; -----Connect with transactor-----
;sys.gtl.connect
```

Refer the below screen shots of loading and running the .cmm practice script:

1. File >OpenScript

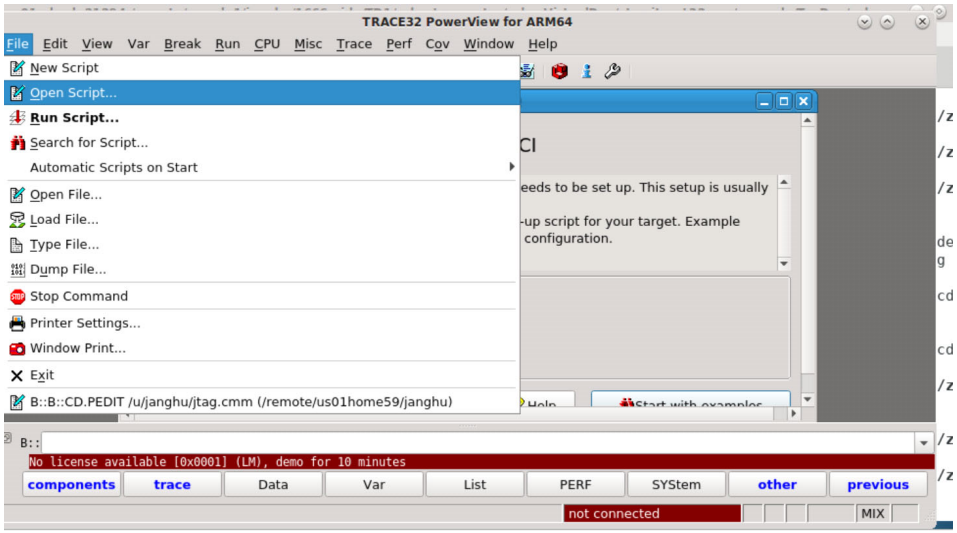

FIGURE 6. Opening Practice Script

2. Select the Desired Script or can take Reference from /example/TapDap/src/env

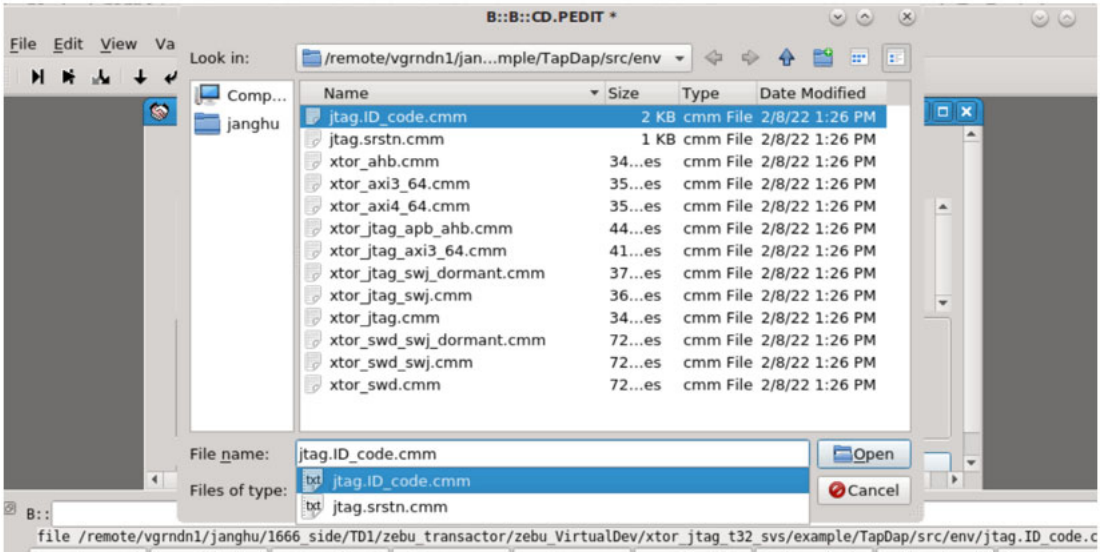

FIGURE 7. Selecting Practice Script

3. Click on Debug option

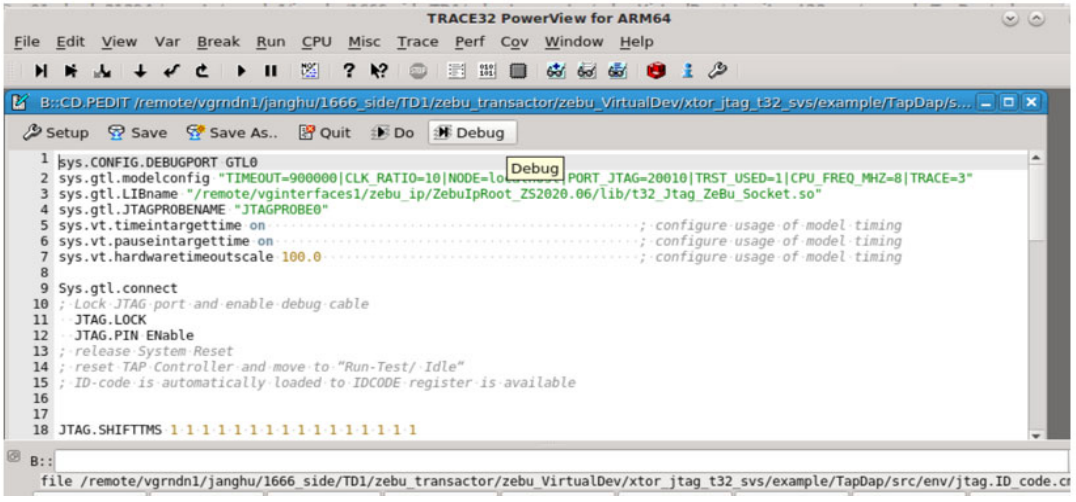

FIGURE 8. Practice Script

4. To Execute the Command line by line keep pressing on step option

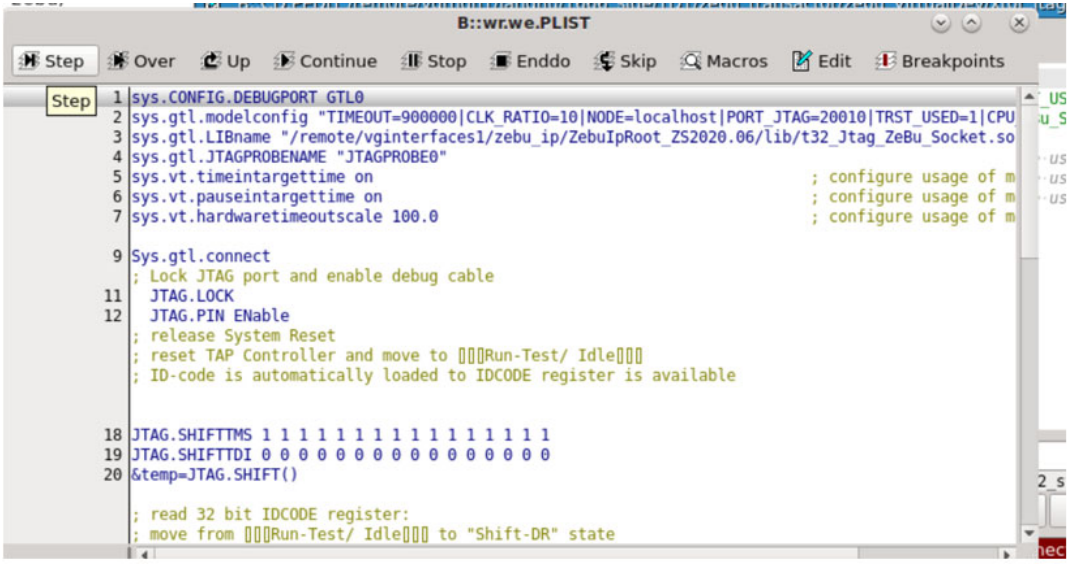

FIGURE 9. Execution of practice Script

Once connection is successful you should be able to see below message from the transactor. Then TRACE32 can be used with the transactor for debugging your SOC like a real target

```
VJTAG [INFO]: JTAGXTOR server: got connection from 127.0.0.1
```

6.2 TRACE32 GTL model configurations

Following table contains all the switches that can be provided with the command `'sys.gtl.modelconfig'`.These options are used to send configurations to the transactor.

TABLE 8 Switches and the commands

Option	Description
NODE	Indicates the IP address (or localhost in case of a local session) of the workstation running the ZeBu TRACE32 JTAG server. Eg: `NODE=127.0.0.1`
PORT_JTAG	Indicates the IP port number used by the ZeBu TRACE32 JTAG server.
PORT_DAP0	Indicates the IP port number used by the ZeBu TRACE32 DAP-1 class
PORT_DAP1	Indicates the IP port number used by the ZeBu TRACE32 DAP-2 class
TRST_USED	Indicates whether the `TRSTn` pin of the JTAG transactor is used or not: 1:`TRSTn` pin is used. 0:`TRSTn` pin is not used
SRST_USED	Indicates whether the SRSTn pin of the JTAG transactor is used or not: • 1: SRSTn pin is used. • 0: SRSTn pin is not used. Please check the SRST example configuration for SRST management.
RTCK_USED	Indicates whether the `RTCK` pin of the JTAG transactor is used or not: 1: `RTCK` pin is used. 0: `RTCK` pin is not used.
CLK_RATIO	Indicates minimum ratio b/w CPU CLK and TCK.
SWITCH_SEQ *Valid only for SWD probe	Specify this option when HW state needs to be changed from JTAG to SWD when using the SWD probe.
SWITCH_SEQ_TO_DORMANT *Valid only for SWD probe	Specify this option when HW state needs to be changed from JTAG to SWD using Dormant state-based switch sequences when using the SWD probe.

TABLE 8 Switches and the commands

SWITCH_SEQ_TO_JTAG *Valid only for JTAG probe	Specify this option when HW state needs to be changed from SWD to JTAG when using the JTAG probe.
SWITCH_SEQ_TO_JTAG_DORMANT *Valid only for JTAG probe	Specify this option when HW state needs to be changed from SWD to JTAG using Dormant state-based switch sequence when using the JTAG probe.
READ_ID *Valid only for SWD probe	Read DPIDR register just making the connection
WAIT_REPEAT_COUNT *Valid only for SWD probe	Repeat the SWD transaction for specified number of times at max if WAIT response is received.
TRACE	Increase verbosity of the debug output of TRACE32.

6.3 Other helpful configurations

- When transferring more than 255 bytes from Lauterbach to external files, specify the following in the .cmm file

```
System.option.Mem11StatusCheck On
```

- Turn off any time-outs in TRACE32

```
sys.vt.timeintargettime on
sys.vt.pauseintargettime on
sys.vt.hardwaretimeoutscale 100.0
```

6.4 Release examples

Please check the release example at `$ZEBU_IP_ROOT/ xtor_jtag_t32_svs/example/TapDap/zebu/README`

Commands for emulation

```
make compile            ZEBU=1  CONFIG=<CONFIG_NAME>
make run_t32_server     ZEBU=1  CONFIG=<CONFIG_NAME>
```

Commands for simulation:* (Only few configs enabled for simulation)

```
make run_t32_server     SIMULATOR=1  CONFIG=<CONFIG_NAME>
```

When run_t32_server is executed, JTAG and DAP servers will be launched based on the config selected.

Once the servers are spawned, user can try running Trace32 as below.

```
% t32marm64 -c <src/env folder>/config_test.32
```

Sample .cmm files are provided in <src/env folder> which can be tried from trace32.

Details of different configurations

==========

Following configurations are available

```
> 1_AXI3_64_JTAG : 64bit AXI3 as DAP0 along with JTAG probe.
| works with ZEBU=1

  > 2_AXI4_64_JTAG : 64bit AXI4 as DAP0 along with JTAG probe.
| works with ZEBU=1

  > 3_AXI3_32_JTAG : 32bit AXI4 as DAP0 along with JTAG probe.
| works with ZEBU=1                      | RTL Clock

  > 4_AXI3_64    : 64bit AXI3 as DAP0 without JTAG probe.
| works with ZEBU=1, SIMULATOR=1

  > 5_AXI4_64    : 64bit AXI4 as DAP0 without JTAG probe.
| works with ZEBU=1, SIMULATOR=1

 *> 6_APB_JTAG   : 32bit APB  as DAP0 along with JTAG probe.
| works with ZEBU=1

  > 7_APB       : 32bit APB  as DAP0 without JTAG probe.
| works with ZEBU=1 ,SIMULATOR=1

   > 8_APB_APB_JTAG : 32bit APB  as DAP0 ,32bit APB as DAP1 along
with JTAG probe. | works with ZEBU=1

   > 9_APB_AXI3_JTAG: 32bit APB  as DAP0 ,64bit AXI3 as DAP1 along
with JTAG probe.| works with ZEBU=1                      | zRci based
```

```
   *> 10_JTAG       : JTAG probe only.
| works with ZEBU=1 ,SIMULATOR=1

    > 11_APB_AHB_JTAG: 32bit APB  as DAP0 ,32bit AHB as DAP1 along
with JTAG probe. | works with ZEBU=1

   > 12_AHB       : 32bit AHB  as DAP0 without JTAG probe.
| works with ZEBU=1 ,SIMULATOR=1

   > 13_TPIU_JTAG   : TPIU probe and JTAG probe.
| works with ZEBU=1

   *> 14_JTAG_ZRCI   : JTAG probe in ZRCI mode
| works with ZEBU=1

   > 15_JTAG_SWD_AHB: JTAG probe with SWD as DAP0 and AHB as DAP1
| works with ZEBU=1

   *> 16_SWD        : SWD only
| works with ZEBU=1

   > 17_JTAG_RTL_ZRCI: JTAG probe with RTL CLOCK in ZRCI mode
| works with ZEBU=1                        | zRci based,RTL Clock

   > 18_JTAG_SRST  : JTAG probe, with System Reset Logic DUT (SRSTn)
| works with ZEBU=1, SIMULATOR=1

   > 19_JTAG_ARM_IP : JTAG probe with A72.
| works with SIMULATOR=1

   > 20_SWD_ARM_IP  : SWD probe with A72.
| works with SIMULATOR=1

   > 21_JTAG_DMTCP  : JTAG probe with DMTCP support
| works with ZEBU=1

   > 22_SWD_DMTCP   : SWD with DMTCP support
| works with ZEBU=1

*Frequently used configs
```

Example cmm files

==========

Please find few example cmm files in /example/TapDap/src/env/ for quick bringup.

```
    xtor_jtag.cmm                | Can be used with  CONFIG=10_JTAG
```

```
    xtor_jtag_swj.cmm           | Can be used with  CONFIG=10_JTAG
    xtor_jtag_swj_dormant.cmm   | Can be used with  CONFIG=10_JTAG
    xtor_axi3_64.cmm            | Can be used with  CONFIG=4_AXI3_64
    xtor_axi4_64.cmm            | Can be used with  CONFIG=5_AXI4_64
    xtor_jtag_apb_ahb.cmm       | Can be used with
CONFIG=11_APB_AHB_JTAG
    xtor_jtag_axi3_64.cmm       | Can be used with
CONFIG=1_AXI3_64_JTAG
    xtor_ahb.cmm                | Can be used with  CONFIG=12_AHB
    xtor_swd.cmm                | Can be used with  CONFIG=16_SWD
    xtor_swd_swj.cmm            | Can be used with  CONFIG=16_SWD
    xtor_swd_swj_dormant.cmm    | Can be used with  CONFIG=16_SWD
```

jtag.ID_code.cmm | Can be used with any config with JTAG DAP enabled, Please use this .cmm file to make sure the communication is properly happening between trace32<->transactor<->dut. Once the .cmm is executed you will be able to see the ID code of target CPU being printed on Trace32 window

ZRCI scripts

==========

zRci scripts for following configs

```
> 14_JTAG_ZRCI : /example/TapDap/src/env/xtor_jtag_zrci.tcl
```

SWJ switching Sequences

==========

If the design has SWJ interface and if user wishes to use

5. JTAG: Refer to config 10_JTAG for hardware and software connections

   Refer xtor_jtag.cmm for getting connected from Trace32

If DUT is stuck at SWD mode prior to the connection, please use SWITCH_SEQ_TO_JTAG=1 or SWITCH_SEQ_TO_JTAG_DORMANT=1 on the .cmm to get it to JTAG mode during connection.

*Refer xtor_jtag_swj.cmm for deprecated switching sequence

*Refer xtor_jtag_swj_dormant.cmm for dormant switching sequence

6. SWD: Refer to config 16_SWD for hardware and software connections

   Refer xtor_swd.cmm for getting connected from Trace32

If DUT is stuck at JTAG mode prior to the connection, please use SWITCH_SEQ=1 or SWITCH_SEQ_TO_DORMANT=1 on the .cmm to get it to SWD mode during connection.

*Refer xtor_swd_swj.cmm for deprecated switching sequence

*Refer xtor_swd_swj_dormant.cmm for dormant switching sequence

7 Troubleshooting

The troubleshooting steps include the following:

7.1 JTAG/SWD Remote Validator Tool

The GUI tool JTAG/SWD Remote Connection Validator tool can be tried with T32 transactor. Tool provides basic sanity tests, identifies communication issues and provide suggestions.

Execute the validator tool from below location

```
$ZEBU_IP_ROOT/bin/zebu_jtag_tester_tool
```

JTAG Mode SWD Mode

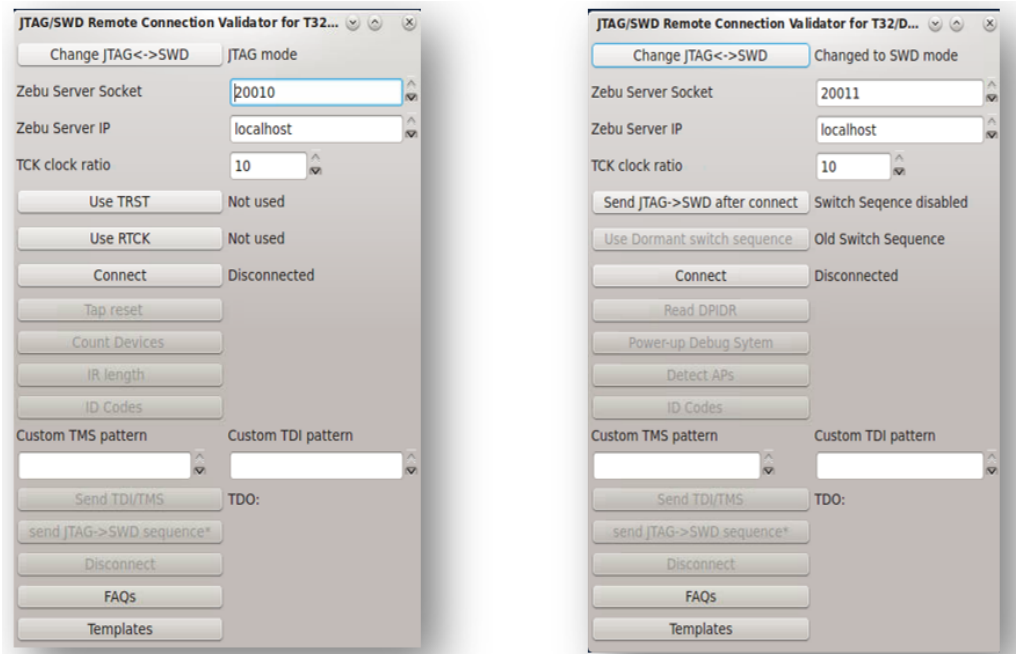

FIGURE 10. JTAG/SWD remote modes

7.1.1 JTAG Mode functionality

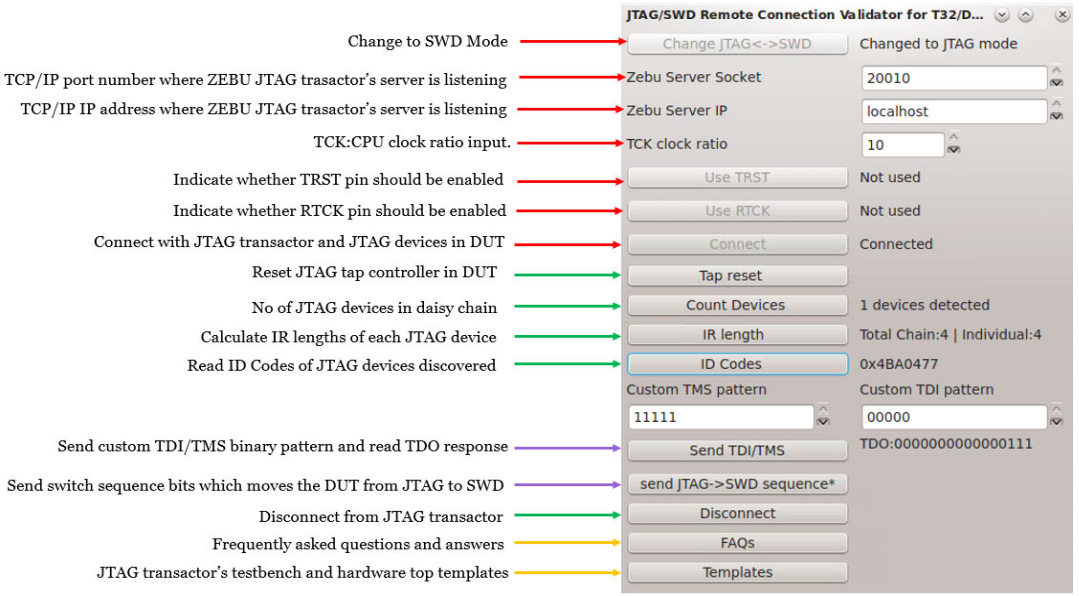

FIGURE 11. JTAG mode functionality

7.1.2 SWD Mode functionality

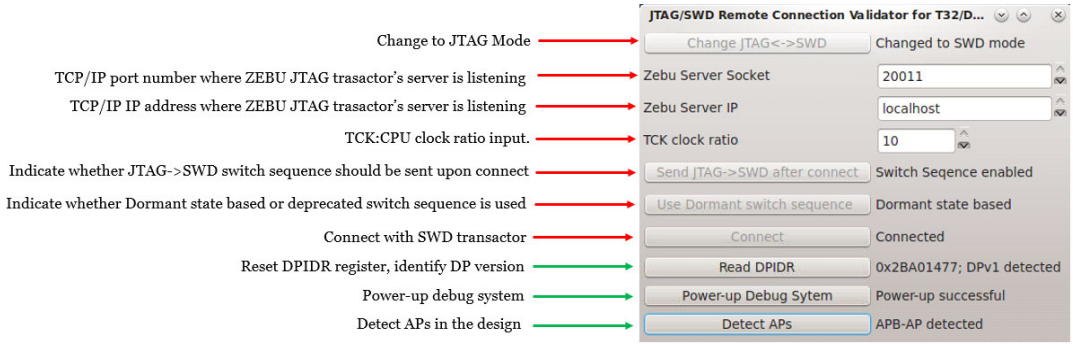

FIGURE 12. SWD mode functionality

7.1.3 Issue detection

The tool provides issue detection and suggests solutions to fix the issues. Tool can identify issues which can occur while connecting with the transactor, issues in TDO response and timeouts while reading the responses from the design.

7.1.4 JTAG count devices

The tool will try to identify how many JTAG TAP devices are present in the JTAG chain. Tool will identify issues and provide suggestions incase issues are detected.

7.1.5 JTAG IR Length

The tool will try to identify the total instruction register length in entire JTAG scan chain. Tool will provide individual IR lengths assuming equal IR lengths in each device.

7.1.6 JTAG IDCode

The tool will try to identify the ID Codes in the entire JTAG chain. Tool will identify issues and provide suggestions incase issues are detected. In case you see an unexpected ID codes, please dump the waves and check the TDO response and JTAG FSM signals.

7.1.7 JTAG send TDI/TMS

User can provide any kind of bit pattern to custom TDI, TMS text boxes and hit send, tool will bring TDO output for the same. You can utilize this tool incase there is some BITS pattern that needs to be tried.

7.1.8 SWD switch sequences

The tool provides sending JTAG->SWD switch sequences to bring your design to SWD mode. There are two types of switch sequences

7. Dormant state based
8. Regular sequence (deprecated in newer devices)

Newer designs require Dormant state-based switch sequence. If you are not aware of the exact sequence required, please select the regular sequence, try connecting with the device and try to read DPIDR. Tool will detect the DP version if the switching is successful. If its not you can disconnect, select the Dormant state based sequence, connect and try read DPIDR.

7.1.9 SWD Read IDR

Reading IDR provides you the DP version being used in your debug subsystem. Successful DP version identification indicates that SWD-DP accesses are working as expected. If the DP discovery is unsuccessful, please dump the waves and check the SWD response generated by the debug system, check whether ACK signal is correct and also the data phase of SWD packet is correct.

7.1.10 SWD power-up debug system

The tool will assert CSYSPWRUPREQ and CDBGPWRUPREQ registers and checks until CSYSPWRUPACK and CDBGPWRUPACK are asserted. Once done tool will indicate successful Power-up. If the system is not getting powered-up please dump the waves and check CSYSPWRUPACK and CDBGPWRUPACK signals in your debug subsystem.

7.1.11 SWD Detect APs

The tool reads AP-IDR register to identify the APs attached to the design. Once identified the AP type is displayed on the tool. If the tool is able to detect the APs successfully, it indicates that the SWD AP access is happening correctly. If its unsuccessful AP detection, please dump the waves and check the AP read results in your debug subsystem.

7.1.12 FAQ

This section includes frequently asked questions related with JTAG T32 transactor. Try checking out the questions and solutions provided for better understanding.

7.1.13 Templates

This section provides you references to correctly connect the pins of HW transactor and provide API references to setup the transactor in testbench.

7.2 Transaction monitoring APIs

There can be scenarios where the debug system might behave incorrect deep into the emulation. In such cases debugging with JTAG bit stream from a log or from a wave dump is difficult. As a solution following list of APIs are present in latest T32 transactor package which gives you high level information.

Note *These APIs provide correct results only when there is a single JTAG device in your JTAG scan chain. These APIs will be enhanced to support multiple device in a future release*

TABLE 9 Transaction Monitoring APIs

Transaction Monitoring APIs	
void printJTAGMsg(bool)	Enable printing every TMS, TDI, TDO messages
void printFSMChanges(bool)	Enable tracing of JTAG FSM state changes

TABLE 9 Transaction Monitoring APIs

void printIRnDR(bool)	Prints IR and DR register changes
void printARM(bool)	Decodes IR and DR registers as per ARM architecture *Experimental feature

7.2.1 void PrintJTAGMsg(bool enable)

Once enabled, transactor prints JTAG packets received by the debugger and corresponding TDO output in hex format

Sample message printed:

JTAG MSG: cycles:7 tms=0x3 tdi=0x4f tdo=0x2e

Waves corresponding to above line:

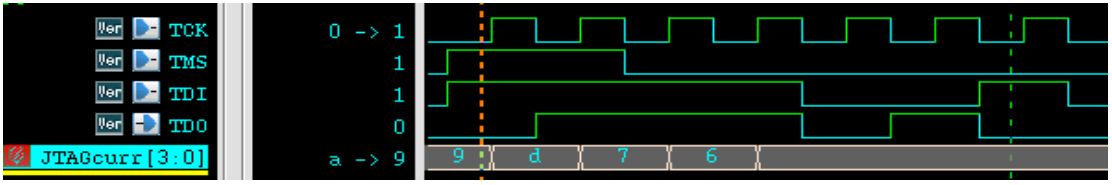

FIGURE 13. Waveform: JTAGMsg

TDI and TMS values are sent from LSB to MSB on each TCK. This is done for ‘cycles’ number of times. Furthermore, TDO is captured on each posedge, and buffered from LSB to MSB

7.2.2 void PrintFSMChanges(bool enable)

Once enabled, transactor prints each jtag cycle transition with the FSM state change.

Sample messages printed:

```
JTAG FSM: | tms_bit:1 | tdi_bit:0 | tdo_bit:1 | state:Test-Logic-Reset
JTAG FSM: | tms_bit:0 | tdi_bit:0 | tdo_bit:1 | state:Run-Test/Idle
```

```
JTAG FSM: | tms_bit:0 | tdi_bit:0 | tdo_bit:1 | state:Run-Test/Idle
JTAG FSM: | tms_bit:1 | tdi_bit:0 | tdo_bit:1 | state:Select-DR-Scan
JTAG FSM: | tms_bit:1 | tdi_bit:1 | tdo_bit:1 | state:Select-IR-Scan
JTAG FSM: | tms_bit:0 | tdi_bit:1 | tdo_bit:1 | state:Capture-IR
JTAG FSM: | tms_bit:0 | tdi_bit:1 | tdo_bit:1 | state:Shift-IR
```

7.2.3 void printIRnDR(bool enable)

Once enabled, transactor prints three types of messages. When IR register gets updated, when DR register is updated, and DR register is read.

Type 1: Sample message printed when IR register is updated

```
JTAG REG: IR-REGISTER-INPUT=0xb LENGTH=4
```

Waves corresponding to above line:

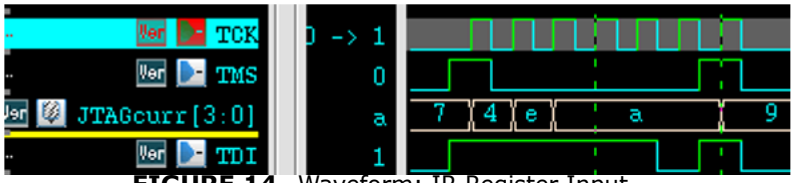

FIGURE 14. Waveform: IR Register Input

Here the FSM state is denoted by the signal JTAGcurr and its value is ‘0xa’ during shift-IR. The instruction fed is 0xb(‘b1011) which is sent from LSB to MSB on TDI pin. The instruction length is denoted by LENGTH

Type 2: Sample message printed when DR register is updated

```
JTAG REG: DR-REGISTER-INPUT=0x4 LENGTH=35
```

Waves corresponding to above line:

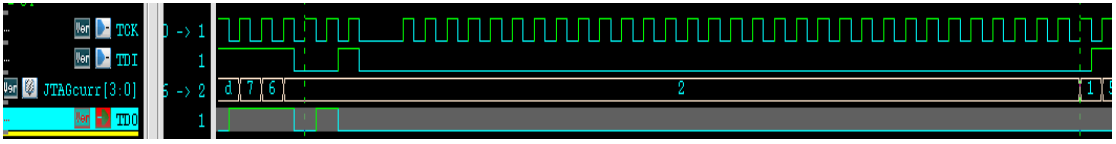

FIGURE 15. Waveform: DR Register Input

Here the FSM state is denoted by the signal JTAGcurr and its value is ‘0x2’ during shift-DR. The Data Value fed is 0x4(‘b100) which is sent from LSB to MSB on TDI pin. The data register length is denoted by LENGTH.

Type 3: Sample message printed when DR register is read

JTAG REG: DR-REGISTER-OUTPUT=0x2 LENGTH=35

Waves corresponding to above line:

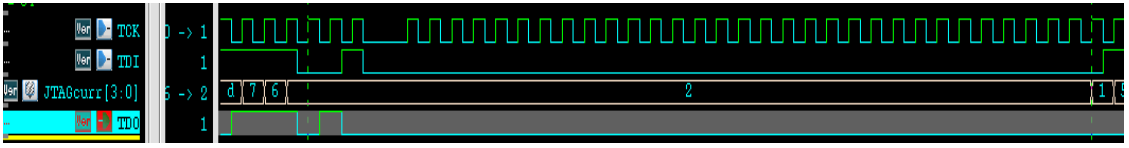

FIGURE 16. Waveform: DR Register Output

Here the FSM state is denoted by the signal JTAGcurr and its value is ‘0x2’ during shift-DR. The Data Value is read from TDO pin each bit buffered from LSB to MSB. Here the TDO pattern is 0->1->0->0….->0 which gives 0x2 as the DR value read.

7.2.4 void printARM(bool enable)

Once enabled, transactor prints DPACC and APACC register updates. Valid only for ARM based devices and the full set of ARM register decoding will be provided in a future release.

Sample messages printed:

JTAG ARM: DR-REGISTER-INPUT -> DPACC Request:: RnW:Write | A:0x8 | Datain:0x0

JTAG ARM: DP-REGISTER-INPUT -> DPACC REGISTER:: SELECT

```
JTAG ARM: DR-REGISTER-INPUT -> SELECT REGISTER UPDATE:: DPBankSel=0x0
| APBankSel=0x0 | APSel=0x0
JTAG ARM: DR-REGISTER-OUTPUT -> DPACC Result:: ACK:OKAY | Dataout:0x0
```

7.3 Turning off timeouts in TRACE32

It is recommended to turn off the debugger timeouts using below commands in your .cmm before calling sys.gtl.connect

```
sys.vt.timeintargettime on
sys.vt.pauseintargettime on
sys.vt.hardwaretimeoutscale 100.0
sys.gtl.connect
```

7.4 Printing debug Information on TRACE32

If user wants to check the JTAG bit stream handed over to transactor by the debugger, use TRACE=3 with sys.gtl.modelconfig on your .cmm

```
sys.gtl.modelconfig
"TIMEOUT=6000000|CLK_RATIO|NODE=localhost|PORT_JTAG=20010|PO
RT_DAP0=20011|PORT_DAP1=20012|CPU_FREQ_MHZ=5|PRINT_STAT=0|TR
ACE=3"
```

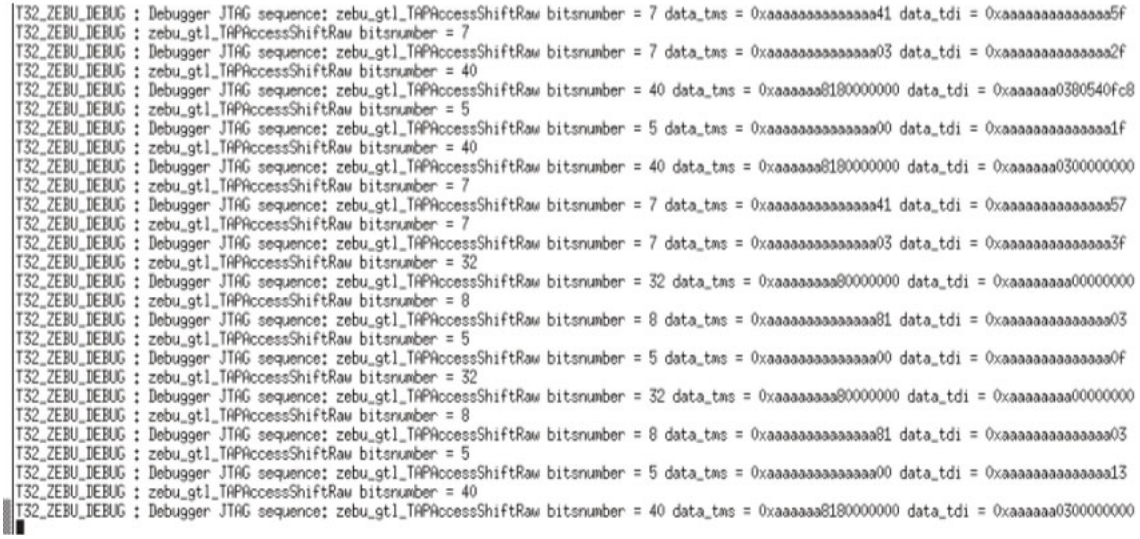

FIGURE 17. Print debug information on TRACE32

7.5 Reading ROM table in Trace32

ROM table provides information about the SOC structure and the base address for each component. If user is not aware about the components and base address settings, try below command to read the ROM table

sys.detect.DAP

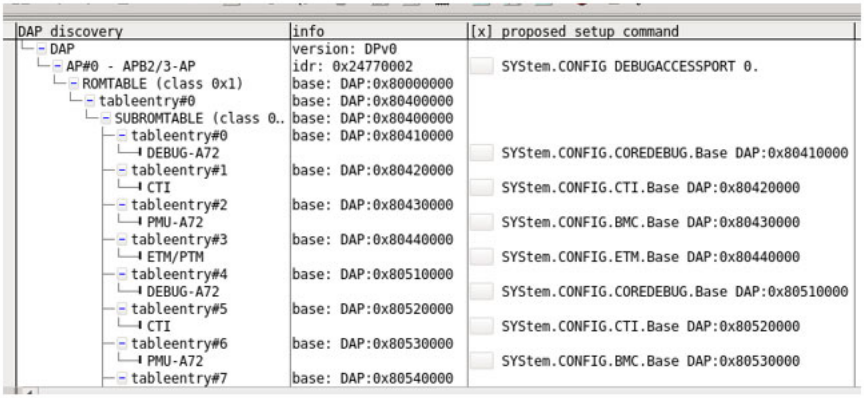

FIGURE 18. Read ROM table

7.6 Waveform and debug log based analysis

- If user is not seeing correct TDO and suspects low-level JTAG communication problems, please use below log and wave analysis method.
- Enable wavedump and increase transactor debug level to 3

You should be see debug messages like below

```
XTOR_JTAG_TAP_SVS :: sending data cycle counter = 0x4, tms
=0x2 and data_tdi = 0x0

XTOR_JTAG_TAP_SVS::JTAG TDO output =0xf
```

Corresponding waveform snippet is highlighted below.

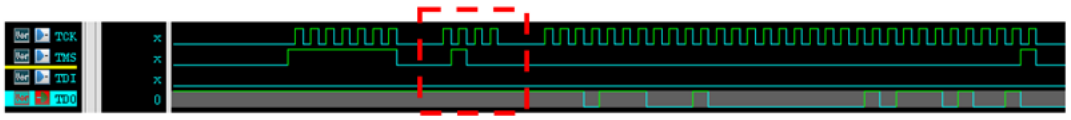

FIGURE 19. Waveform Debug

- For a single debug message printed, a group of TCKs are emitted by the transactor
  - ❒ Cycle counter= number of TCK pulses

- ❒ tms = TMS pattern sent from LSB to MSB
- ❒ tdi =TDI pattern sent from LSB to MSB
- ❒ TDO= TDO pattern captured from LSB to MSB

- You can try to correlate the debug message with the waveform to check if TCK,TMS,TDI signals are driven correctly.
- If any of the TCK/TMS/TDI are not getting driven, please check for
  a. Multiple drivers on the connections TCK/TDI/TMS/TDO.
  b. Check whether you are connecting with bidirectional pins of the DUT, in such cases check if tristate logic is correctly updated.
- If waves are not as per expectation, then check below debug signals to check whether transactions actually occur between hardware and software

  jtag_xtor_info_port[7]: Asserts when debugger sends a request, De-asserts when the debugger request is complete in HW

  jtag_xtor_info_port[50:44]: Gives you how many bits are driven out of the debugger's software message.

FIGURE 20. Debug Signals

7.7 How to identify Multiple drivers

Check backend_default/zTopBuild_report.log

Manipulating multi-driven net:

- xtor_jtag_top.tms_i with conflict resolution WAND with 1 drivers and 1 constant drivers

- xtor_jtag_top.tck_dpi with conflict resolution WAND with 1 drivers and 1 constant drivers

7.8 Checking Slowness in APACC/DPACC

If user is experiencing slowness while connected to CPU it could be due to slowness in the AP buses inside the coresight module. Please try below API on the testbench to observe the ACK value being received from the DUT.

7.8.1 void printARM(bool enable)

The API will print below lines, check if ACK:WAIT responses are received frequently.

JTAG ARM: DR-REGISTER-OUTPUT -> DPACC Result:: ACK:WAIT | Dataout:0x0

If more and more wait responses are occurring user needs to check AP buses inside the coresight module for any slowness in AMBA transactions.