ZeBu JTAG SWD Transactor User Manual

Version V-2024.03-SP1, February 2025

Feedback

Contents

Feedback

About This Manual

Overview

This manual describes how to use the ZeBu SWD transactor with your design being emulated in ZeBu.

Overview

This manual describes how to use the ZeBu SWD transactor with your design being emulated in ZeBu.

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Vertical Solutions User Guide	Provides basic information on ZeBu transactors and memory models and the associated features.
ZeBu Transactor Library Release Notes	Provides information on new features, enhancements, and limitations for a specific release.

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.

Feedback

1 Introduction

The ZeBu SWD transactor implements an interface to drive a SWD port of a design implemented in ZeBu. This SWD interface is managed by a software layer that controls the clock and data signals.

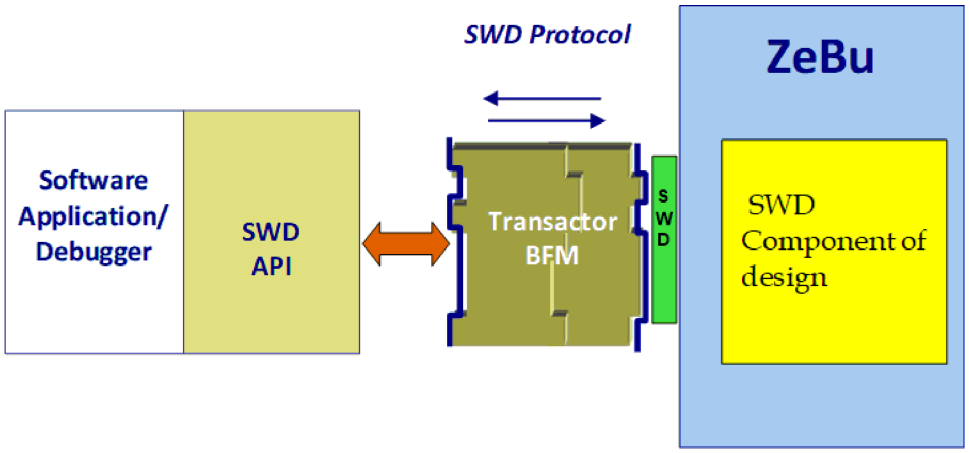
This section explains the following topics:

- Overview
- Features

Overview

The ZeBu JTAG SWD transactor provides an application layer interface compatible with most of the SWD low-level drivers. The SWD transactor enables you to directly connect a software debugger or a test application software to the DUT mapped in ZeBu. The SWD transactor can be used locally or remotely. The remote connection enables you to drive the SWD port with application software running on a remote location. The JTAG SWD transactor provides a set of APIs to create a client server setup, which can be used to manage remote connections to the transactor.

Figure 1 SWD Interface



Features

Following are the features of the JTAG SWD transactor:

- Real-time JTAG speed
- Compatible with most of the JTAG SWD controllers
- Supports the following configurable ports:
  - TRST optional port
  - RTCK optional port
  - CPU/JTAG TCK clock ratio
- Provides API methods to be integrated with specific debuggers Trace 32, DS5 and so on
- Supports remote connection from Linux or Windows platforms

2 Installation

This section explains the following topics:

Installation Procedure

To install the SWD transactor, perform the following steps:

1. Ensure that you have WRITE permissions on the IP directory and the current directory.
2. 1. Download the xtor_jtag_swd_svs compressed shell archive (.sh).
3. *Install*the SWD transactor as follows:

```
$sh xtor_jtag_swd_svs.<version>.sh install [ZEBU_IP_ROOT]
```

Where: *[ZEBU_IP_ROOT]* is the path to your ZeBu IP directory:

- If no path is specified, the *ZEBU_IP_ROOT* environment variable is used automatically.
- If the path is specified and a *ZEBU_IP_ROOT* environment variable is also set, the transactor is installed at the defined path and the environment variable is ignored.

The installation process is complete and successful when the following message is displayed:

```
xtor_jtag_swd_svs.<num> has been successfully installed
```

If there is an error during the installation, an error message is displayed. Here is an example error message:

```
ERROR: /path/directory is not a valid directory.
```

During installation, the symbolic links are created in the following directories for an easy access from all ZeBu tools.

- $ZEBU_IP_ROOT/include
- $ZEBU_IP_ROOT/lib
- $ZEBU_IP_ROOT/vlog

For example, zCui automatically looks for all transactor drivers in the $ZEBU_IP_ROOT/vlog/vcs directory when $ZEBU_IP_ROOT is set.

Package Description

After the SWD Master transactor package is successfully installed, it provides the following elements:

- .so library of the transactor (*lib* directory)
- Header files of the transactor in the *include* directory.

File Tree

The following is an example of the SWD transactor file tree after package installation:

```
├── doc
│   ├── foss
│   │   └── ZX-XTOR-Library_20210329_FOSS.pdf
│   ├── ZeBu_VS_UM.pdf
│   └── ZeBu_XTOR_JTAG_SWD_svs_UM.pdf
├── example
│   ├── src
│   │   ├── bench
│   │   │   ├── client_test.cc
│   │   │   ├── tb_top.cc
│   │   │   ├── tb_zRci.cc
│   │   │   └── tests
│   │   │   ├── ts.server_test.hh
│   │   │   └── ts.trivial_test.hh
│   │   ├── dut
│   │   │   ├── dut.v
│   │   │   ├── hw_top.sv
│   │   │   └── tb_top.sv
│   │   └── env
│   │   ├── designFeatures
│   │   ├── designFeatures_sem
│   │   ├── project_rtl_clk.utf
│   │   ├── project_sem_rtl_clk.utf
│   │   ├── project_sem.utf
│   │   └── project.utf
│   └── zebu
│   ├── Makefile
│
```

```
│   └── zrci_script.tcl
├── include
│   ├── svt_c_runtime_cfg.16.0.hh
│   ├── svt_c_runtime_cfg.hh -> svt_c_runtime_cfg.16.0.hh
│   ├── svt_c_threading.16.0.hh
│   ├── svt_c_threading.hh -> svt_c_threading.16.0.hh
│   ├── svt_hw_platform.16.0.hh
│   ├── svt_hw_platform.hh -> svt_hw_platform.16.0.hh
│   ├── svt_message_port.16.0.hh
│   ├── svt_message_port.hh -> svt_message_port.16.0.hh
│   ├── svt_pthread_threading.16.0.hh
│   ├── svt_pthread_threading.hh -> svt_pthread_threading.16.0.hh
│   ├── svt_report.16.0.hh
│   ├── svt_report.hh -> svt_report.16.0.hh
│   ├── svt_report_uvm.16.0.hh
│   ├── svt_report_uvm.hh -> svt_report_uvm.16.0.hh
│   ├── svt_simulator_platform.16.0.hh
│   ├── svt_simulator_platform.hh -> svt_simulator_platform.16.0.hh
│   ├── svt_systemc_threading.16.0.hh
│   ├── svt_systemc_threading.hh -> svt_systemc_threading.16.0.hh
│   ├── svt_systemverilog_threading.16.0.hh
│   ├── svt_systemverilog_threading.hh ->
svt_systemverilog_threading.16.0.hh
│   ├── svt_zebu_platform.16.0.hh
│   ├── svt_zebu_platform.hh -> svt_zebu_platform.16.0.hh
│   ├── Xtor.16.0.hh
│   ├── xtor_cb_svs.Q-2021.03-RC.svi
│   ├── xtor_cb_svs.svi -> xtor_cb_svs.Q-2021.03-RC.svi
│   ├── Xtor_defines.16.0.hh
│   ├── Xtor_defines.hh -> Xtor_defines.16.0.hh
│   ├── Xtor.hh -> Xtor.16.0.hh
│   ├── xtor_if_svs.Q-2021.03-RC.svi
│   ├── xtor_if_svs.svi -> xtor_if_svs.Q-2021.03-RC.svi
│   ├── xtor_jtag_swd_client.hh -> xtor_jtag_swd_client.Q-2021.03-RC.hh
│   ├── xtor_jtag_swd_client.Q-2021.03-RC.hh
│   ├── xtor_jtag_swd_svs.hh -> xtor_jtag_swd_svs.Q-2021.03-RC.hh
│   ├── xtor_jtag_swd_svs.Q-2021.03-RC.hh
│   ├── XtorScheduler.16.0.hh
│   ├── XtorScheduler.hh -> XtorScheduler.16.0.hh
│   ├── xtor_swd_common.hh -> xtor_swd_common.Q-2021.03-RC.hh
│   ├── xtor_swd_common.Q-2021.03-RC.hh
│   ├── ZebuIpRoot.16.0.hh
│   ├── ZebuIpRoot.hh -> ZebuIpRoot.16.0.hh
│   ├── ZFSDB.16.0.hh
│   └── ZFSDB.hh -> ZFSDB.16.0.hh
├── lib -> lib64
├── lib_1
│   ├── libxtor_jtag_swd_client.Q-2021.03-RC.so
│   ├── libxtor_jtag_swd_client.so ->
libxtor_jtag_swd_client.Q-2021.03-RC.so
│   ├── libxtor_jtag_swd.Q-2021.03-RC.so
│   ├── libxtor_jtag_swd_svs.so -> libxtor_jtag_swd.Q-2021.03-RC.so
│   ├── libZebuXtor.16.0.so
│   ├── libZebuXtorSim.16.0.so
│   ├── libZebuXtorSim.so -> libZebuXtorSim.16.0.so
│   ├── libZebuXtor.so -> libZebuXtor.16.0.so
│   ├── libZebuXtorUVM.16.0.so
│   └── libZebuXtorUVM.so -> libZebuXtorUVM.16.0.so
├── lib64
│   ├── libxtor_jtag_swd_client.Q-2021.03-RC.so
│   ├── libxtor_jtag_swd_client.so ->
libxtor_jtag_swd_client.Q-2021.03-RC.so
│   ├── libxtor_jtag_swd.Q-2021.03-RC.so
│   ├── libxtor_jtag_swd_svs.so -> libxtor_jtag_swd.Q-2021.03-RC.so
│   ├── libZebuXtor.16.0.so
│   ├── libZebuXtorSim.16.0.so
│   ├── libZebuXtorSim.so -> libZebuXtorSim.16.0.so
│   ├── libZebuXtor.so -> libZebuXtor.16.0.so
│   ├── libZebuXtorUVM.16.0.so
│   └── libZebuXtorUVM.so -> libZebuXtorUVM.16.0.so
└── vlog
├── gtech
├── gtech_lib.16.0.v
├── gtech_lib.v -> gtech_lib.16.0.v
├── svt_dpi.16.0.sv
├── svt_dpi_globals.16.0.sv
├── svt_dpi_globals.sv -> svt_dpi_globals.16.0.sv
├── svt_dpi_report_uvm.16.0.sv
├── svt_dpi_report_uvm.sv -> svt_dpi_report_uvm.16.0.sv
├── svt_dpi.sv -> svt_dpi.16.0.sv
├── svt_systemverilog_threading.16.0.sv
├── svt_systemverilog_threading.sv -> svt_systemverilog_threading.16.0.sv
└── vcs
├── vs_fifo_udpi.16.0.sv
├── vs_fifo_udpi.sv -> vs_fifo_udpi.16.0.sv
├── vs_memory.16.0.sv
├── vs_memory.sv -> vs_memory.16.0.sv
├── xtor_jtag_swd_svs.sv
├── zebu_vs_apb_master_udpi.16.0.sv
├── zebu_vs_apb_master_udpi.sv -> zebu_vs_apb_master_udpi.16.0.sv
├── zebu_vs_complex_hwtosw_fifo_udpi.16.0.sv
├── zebu_vs_complex_hwtosw_fifo_udpi.sv ->
zebu_vs_complex_hwtosw_fifo_udpi.16.0.sv
├── zebu_vs_complex_rst_and_clk_udpi.16.0.sv
├── zebu_vs_complex_rst_and_clk_udpi.sv ->
zebu_vs_complex_rst_and_clk_udpi.16.0.sv
├── zebu_vs_complex_swtohw_fifo_udpi.16.0.sv
├── zebu_vs_complex_swtohw_fifo_udpi.sv ->
zebu_vs_complex_swtohw_fifo_udpi.16.0.sv
├── zebu_vs_from_sw_fifo.16.0.sv
├── zebu_vs_from_sw_fifo.sv -> zebu_vs_from_sw_fifo.16.0.sv
├── zebu_vs_simple_hwtosw_fifo_udpi.16.0.sv
├── zebu_vs_simple_hwtosw_fifo_udpi.sv ->
zebu_vs_simple_hwtosw_fifo_udpi.16.0.sv
├── zebu_vs_simple_rst_and_clk_udpi.16.0.sv
├── zebu_vs_simple_rst_and_clk_udpi.sv ->
zebu_vs_simple_rst_and_clk_udpi.16.0.sv
├── zebu_vs_simple_swtohw_fifo_udpi.16.0.sv
├── zebu_vs_simple_swtohw_fifo_udpi.sv ->
zebu_vs_simple_swtohw_fifo_udpi.16.0.sv
├── zebu_vs_to_sw_fifo.16.0.sv
└── zebu_vs_to_sw_fifo.sv -> zebu_vs_to_sw_fifo.16.0.sv
```

Feedback

3

Hardware Interface

This section explains the following topics:

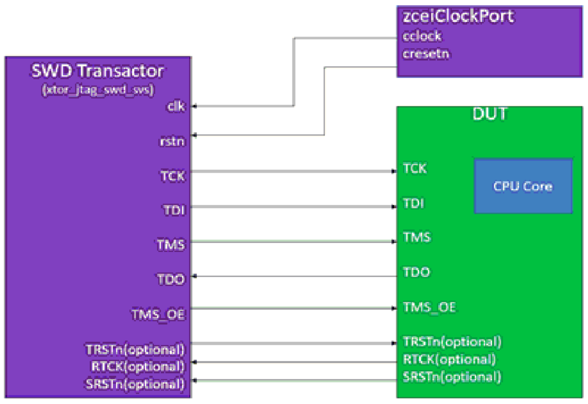
- Introduction
- Signal List
- Connecting Signals to the Design

Introduction

The ZeBu SWD transactor connects to the DUT port through a directional interface.

The following figure illustrates the hardware interface for the ZeBu SWD transactor:

Figure 2   SWD Transactor Hardware Interface



Signal List

The following table lists the signals for the SWD transactor:

Table 1 Signal List

Signal	Direction	Description
clk	Input	Clock input for the transactor
rstn	Input	Reset input for transactor; active low.
TCK	Output	Output from transactor, generated from CPU clock depending on clock ratio configuration in software.
RTCK	Input	Optional input port to transactor to connect RTCK output of CPU.
TMS	Output	Data output from the transactor.
TMS_OE	Output	Output enable signal from transactor which is active when transactor is driving data.
TDO	Input	Input data to the transactor.
TRSTn	Output	Optional reset signal which can be input to design, software configuration *TRSTn_used* must to be set to 1.
SRSTn	Input	System Reset detection (optional)
debug_mode	Output	Output signal: 0 when transactor is in JTAG mode and 1 if it is in SWD mode.
TDI	Output	Output from transactor: valid only in JTAG mode.
swd_xtor_info_port	Output	Output debug port of the transactor.

In the above table:

- The bidirectional data pin specified by the protocol is managed by the transactor using TMS, TDO and TMS_OE, where TMS is output data from transactor when TMS_OE is 1 and TDO is input to transactor when TMS_OE is 0. Therefore, TDO and TMS can be connected to bidirectional port of CPU using TMS_OE.
- Debug mode output signal indicates if transactor is in JTAG or SWD mode. SWD transactor must work in JTAG mode to send initial switching sequence to move CPU from JTAG to the SWD state.

Connecting Signals to the Design

The following code snippet describes the connection between the SWD transactor and the design:

```
module hw_top();
wire TCK;
wire TMS;
wire TDO;
wire TRST;
wire debug_mode;
wire [63:0] TDOMsg;
wire [77:0] swd_xtor_info_port;
  wire hwtop_clk;
  wire hwtop_rstn;
zceiClockPort zceiClockPort_top (
.cclock (hwtop_clk),
.cresetn (hwtop_rstn)
);
xtor_jtag_swd_svs swd_node_i(
.clk(hwtop_clk),
.rstn(hwtop_rstn),
.TCK(TCK),
.TMS(TMS),
.TMS_OE(TMS_OE),
.TDI(TDI),
.TDO(TDO),
.RTCK(1'b0),
.TRSTn(TRSTn),
.SRSTn(SRSTn),
.TDOMsg(TDOMsg),
.debug_mode (debug_mode),
.swd_xtor_info_port (swd_xtor_info_port)
);
top dut0(
  //--- SWD ---
   .tck_swd_i(TCK),
   .tms_swd_i(TMS)   ,
   .tms_oe_swd_i(TMS_OE)   ,
   .tdi_swd_i(TDI)   ,
   .tdo_swd_o(TDO)   ,
   .trstn_swd_i(1'b1)
);
endmodule
```

Debug mode output signal indicates if transactor is in the JTAG or SWD mode. SWD transactor must work in JTAG mode to send initial switching sequence to move CPU from JTAG to the SWD state.

Feedback

4

Software Interface

The ZeBu xtor_jtag_swd_svs transactor provides C++ APIs. They are included among the header files located in $ZEBU_IP_ROOT/include.

The following table describes the header files for the ZeBu SWD API:

Table 2 Header Files for SWD Transactor

Header Files	Description
xtor_jtag_swd_svs.hh	Contains top-level APIs for driving and receiving data. Also, contains APIs to make SWD work in server mode where data is driven from client through TCP IP.
xtor_jtag_swd_client.hh	Contains client-side APIs of SWD transactor which can be used to send and receive packets through TCP/IP via transactor server.
xtor_swd_common.hh	Common enumerations, structs and configurations used in SWD transactor.

This section explains the following topics:

- SWD APIs
- Transactor Parameters
- Client APIs
- Connecting and Initializing the Transactor
- Driving and Receiving Data

SWD APIs

The following table lists the APIs for the SWD transactor.

Table 3 SWD API List

API	Description
xtor_jtag_swd_svs* getInstance(const char* hdl_path, svt_c_runtime_cfg* runtime = 0);	Gets xtor_jtag_swd_svs transactor instance, hdl_path represents the path of transactor instance in hardware. Runtime object with platform, threading set must to be passed from the testbench. Returns instance of the transactor.
bool Configure(XtorParameter param, uint32_t val)	Sets transactors parameters to the specified values. Refer to the table below to see supported param's. returns false if successful and true if error occurs.
bool getParam(XtorParameter param, uint32_t & val)	Gets the value of transactor parameter specified by parameter, value is returned in argument val.
bool printConfig()	Prints transactor configuration.
bool runUntilReset()	Blocking call to transactor, waits for transactor to come out of reset state.
bool is_reset_active()	Non-blocking call to get the value of reset from the transactor.1: reset0: not reset
bool runClk(uint32_t numClk)	Blocking call to wait for a specified number of cycles.
bool setDebugLevel(uint8_t level)	Sets the verbosity of displayed messages by the transactor. Value can range from 0 to 4¾that is, lowest to highest respectively.
bool setDebugFile(const char* file_name,int file_control)	Controls the log file to be used for dumping debug messages of the transactor.
void waitTCKCycles(uint64_t nb_cycle)	Runs for TCK clock cycles specified by the argument.
void setCPUFrequency(uint32_t frequency)	Sets CPU frequency specified by argument for wait function calculation.
uint32_t getCPUFrequency()	Returns CPU frequency set by the user.
bool getMode()	Returns current mode of transactor; 1 for SWD and 0 for JTAG.
uint64_t getCPUCycle()	Returns current CPU cycle number.
void waitCycles(uint64_t cycleNum)	Waits for number of CPU cyles specified in the argument.
bool waitMicroseconds(uint32_t time)	Waits amount of time specified by the argument; returns true if ok, false if CPU frequency has not been set

Table 3 SWD API List (Continued)

API	Description
bool forceSignals(uint8_t signals,uint8_t mask)	Function for setting the value of TRST,TDI,TMS, and SRST signals.
bool getSignals(uint8_t &signal)	Returns value of TRST,TDI,TMS and SRST signals.
bool Add(int TMSbit, int TDIbit)	Adds TDI and TMS shift to send to the transactor.
void AddSequenceAndStream(unsigned int size, unsigned int char* tms,unsigned int char* tdi)	Adds a sequence of TDI and TMS data to transactor, param size is the size of stream.
bool AddStream(unsigned int size,unsigned char* tdi_stream)	Adds a stream of TDI, param size is size of stream.
bool Send(int sendReceive=SEND_RECEIVE)	Function used to send TDI and TMS added in the stream (previously). TDO is captured if param is SEND_RECEIVE, ignore if param is SEND_ONLY.
bool SkipTDOBits(unsigned int skip)	Function used to skip number of TDO bits specified in the param.
int GetTDOBit()	Function used to get first TDO bit from tdo chain received.
bool GetTDOBits(unsigned int size,unsigned char* chainOut)	Function used to get chain of TDO bits from tdo bits received, param size is length of TDO bits returned in chainOut argument.
bool AddSequence(int tdi,const char* tms_seq, int state=0)	Function used to add a sequence of TDI and TMS data to the transactor.
bool readyToSave()	Legacy API from Zcei (to be deprecated in the future release)
bool save()	Legacy API from Zcei (to be deprecated in the future release)
bool configRestore()	Legacy API from Zcei (to be deprecated in the future release)
void init(Board* zebu)	Legacy API from Zcei (to be deprecated in the future release)
Server Mode APIs	
void quitSession()	Quits server session

Table 3 SWD API List (Continued)

API	**Description**
void t32quitSession()	Disconnects server session
void loop ()	Listens to server
void connect(int socketNum)	Connects server on socket number specified by arg.
void disconnect()	Quits server session
void t32disconnect()	Disconnects server session
uint8_t getDMABusWidth()	Returns the width of DMA bus if attached to T32.
uint64_t getDMABusSize()	Returns the size of DMA bus if attached to T32.
bool getDMAWrite(uint64_t offset,uint8_t * data, uint64_t size)	Function to write data into DMA of specified size
bool getDMAWrite(uint64_t offset,uint8_t * data, uint64_t size)	Function to read data from DMA of specified size from offset in arg.
bool connectDMA()	Connects to DMA for T32
void RegisterCallBacks(DMACallBacks* cb)	Register callbacks related to DMA functions for T32.

Transactor Parameters

The following table lists the parameters for the SWD transactor:

Table 4 SWD Transactor Parameters

Parameter	Description
TRST_USED_SWD	Must be configured if optional TRSTn port is being used.
RTCK_USED_SWD	Must be configured if optional RTCK port is being used.
CPUCLK_TCK_RATIO_SWD	Default is 4, must be set to a desired ratio of TCK to CPU CLK.
XTOR_DEBUG_MODE	Used to configure debug mode of the transactor.

Client APIs

The following lists the APIs available for *xtor_jtag_swd_client.hh* and used to control client side of the transactor:

Table 5 Client APIs for SWD Transactor

API	Description
xtor_jtag_swd_client()	Constructor for client
~xtor_jtag_swd_client()	Destructor for client
void init()	Function used to initialize communication from the transactor.
void reset()	Function used to reset transactor.
bool getMode()	Gets debug mode; 1 if SWD and 0 if JTAG
void waitTCKCycles(uint64_t nb_cycle)	Runs specified number of TCK cycles.
void setCPUFrequency(uint32_t frequency)	Sets CPU frequency specified by the argument for wait function calculation.
uint32_t getCPUFrequency()	Returns CPU frequency set by the user.
bool getMode()	Returns current mode of transactor; 1 for SWD and 0 for JTAG.
uint64_t getCPUCycle()	Returns the current CPU cycle number.
void waitCycles(uint64_t cycleNum)	Waits for number of CPU cycles specified in the argument.
bool waitMicroseconds(uint32_t time)	Waits amount of time specified by the argument; returns true if ok, false if CPU frequency is not set.
bool setDebugLevel(uint8_t level)	Sets the verbosity of displayed messages by the transactor. Value can range from 0 to 4¾that is, lowest to highest respectively.
bool forceSignals(uint8_t signals,uint8_t mask)	Function for setting value of TRST,TDI,TMS, and SRST signals.
bool getSignals(uint8_t &signal)	Returns value of TRST,TDI,TMS and SRST signals.
bool Add(int TMSbit, int TDIbit)	Adds TDI and TMS shift to send to the transactor.

Table 5 Client APIs for SWD Transactor (Continued)

API	Description
bool Send(int sendReceive=SEND_RECEIVE)	Function used to send TDI and TMS added in stream (previously). TDO is captured if param is SEND_RECEIVE, ignore if param is SEND_ONLY.
bool SkipTDOBits(unsigned int skip)	Function used to skip number of TDO bits specified in param.
int GetTDOBit()	Function used to get first TDO bit from tdo chain received.
bool GetTDOBits(unsigned int size,unsigned char* chainOut)	Function used to get chain of TDO bits from tdo bits received, param size is length of TDO bits returned in chainOut argument.
void AddSequenceAndStream(unsigned int size, unsigned int char* tms,unsigned int char* tdi)	Adds a sequence of TDI and TMS data to transactor, param size is the size of stream.
bool AddStream(unsigned int size,unsigned char* tdi_stream)	Adds a stream of TDI, param size is the size of stream.
bool Configure(XtorParameter param, uint32_t val)	Sets transactors parameters to specified values. Refer to the table above to see supported params. Returns false if successful and true if error occurs.
uint64_t getDMABusSize()	Returns the size of DMA bus if attached to T32.
bool getDMAWrite(uint64_t offset,uint8_t * data, uint64_t size)	Function to write data into DMA of specified size.
bool getDMAWrite(uint64_t offset,uint8_t * data, uint64_t size)	Function to read data from DMA of specified size from offset in arg.
bool connectDMA()	Connects to DMA for T32
bool readyToSave()	Legacy API from Zcei (to be deprecated in the future release)
bool save()	legacy API from Zcei (to be deprecated in the future release)
bool configRestore()	legacy API from Zcei (to be deprecated in the future release)

Connecting and Initializing the Transactor

You can connect and initialize the transactor in the following modes:

- Standalone Mode
- Server Mode

Standalone Mode

The following code snippet shows the transactor connection and initialization in the standalone mode without any server client communication:

```
#include <string.h>
#include "xtor_jtag_swd_svs.hh"
#include <queue>
#include "stdio.h"
using namespace ZEBU_IP;
using namespace XTOR_JTAG_SWD_SVS;
using namespace ZEBU;
class trivial_test
{
public:
int main_phase()
{
svt_c_runtime_cfg* runtime = svt_c_runtime_cfg::get_default();
xtor_jtag_swd_svs *swd =
 xtor_jtag_swd_svs::getInstance("hw_top.swd_node_i", runtime);
swd->setDebugLevel(4);
//Configure SWD
swd->runUntilReset();
swd->Configure(XTOR_DEBUG_MODE, 1);
swd->Configure(CPUCLK_TCK_RATIO_SWD, 3);
```

Server Mode

Initialize the server using the following code:

```
#include <string.h>
#include "xtor_jtag_swd_svs.hh"
#include <queue>
#include "stdio.h"
using namespace ZEBU_IP;
using namespace XTOR_JTAG_SWD_SVS;
using namespace ZEBU;

class server_test
{
```

```
public:
int main_phase()
{
svt_c_runtime_cfg* runtime = svt_c_runtime_cfg::get_default();
xtor_jtag_swd_svs *swd_server  = new
 xtor_jtag_swd_svs("hw_top.swd_node_i",runtime);

swd_server->setDebugLevel(1);
swd_server->connect(20010);
while(run_active) {
swd_server->loop();
}
swd_server->disconnect();
delete swd_server;
}
};
```

Also, you can initialize the client using the following code:

```
#include <string.h>
#include "xtor_jtag_swd_client.hh"
#include <queue>
#include "stdio.h"
using namespace ZEBU_IP;
using namespace XTOR_JTAG_SWD_SVS;
using namespace ZEBU;

int main (int argc, char *argv[])
{
xtor_jtag_swd_client *swd_client  = new xtor_jtag_swd_client();

//Configure SWD
swd_client->init();
swd_client->Configure(XTOR_DEBUG_MODE, 1);
swd_client->Configure(CPUCLK_TCK_RATIO_SWD, 3);
}
};
```

Driving and Receiving Data

After completing the initialization and configuration, SWD packets are generated per the following configuration:

```
tms_pt[0] =  start_bit;
tms_pt[0] |= APnDP << 1;
tms_pt[0] |= rd_wr << 2;
tms_pt[0] |= Add_lower << 3;
tms_pt[0] |= Add_Upper << 4;
tms_pt[0] |= parity  << 5;
```

```
tms_pt[0] |= stop_bit << 6;
tms_pt[0] |= park_bit << 7;
tms_pt[1] = 4;
tms_pt[2] = 5;
tms_pt[3] = 6;
tms_pt[4] = 7;

tms_tmp = tms_pt[1];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
tms_tmp = tms_pt[2];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
tms_tmp = tms_pt[3];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
tms_tmp = tms_pt[4];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
//std::cout <<"data_parity is "<<data_parity;
tms_pt[5] = data_parity;

if(rd_wr) {
std::cout<<"\nSending Transaction->"<<a<<" Type: Read ";
} else {
std::cout<<"\nSending Transaction "<<a<<" Type: Write ";
std::cout<<"\n\tData 0 byte "<<std::hex<<int(tms_pt[1])<<" ";
std::cout<<"\n\tData 1 byte "<<std::hex<<int(tms_pt[2])<<" ";
std::cout<<"\n\tData 2 byte "<<std::hex<<int(tms_pt[3])<<" ";
std::cout<<"\n\tData 3 byte "<<std::hex<<int(tms_pt[4])<<" ";
}
swd->AddSequenceAndStream(53,(unsigned char*)tms_pt,(unsigned
 char*)tdi_pt);
swd->Send(0);
uint8_t *out_bit = new uint8_t [8] ;

if(swd->GetTDOBits(53,out_bit))
{
std::cout<<"Error in reading ";
}
uint8_t *result = new uint8_t [8];
uint8_t ack;
```

```
tms_pt[0] =  start_bit ;
tms_pt[0] |= APnDP << 1;
tms_pt[0] |= rd_wr << 2;
tms_pt[0] |= Add_lower << 3;
tms_pt[0] |= Add_Upper << 4;
tms_pt[0] |= parity  << 5;
tms_pt[0] |= stop_bit << 6;
tms_pt[0] |= park_bit << 7;
tms_pt[1] = 4;
tms_pt[2] = 5;
tms_pt[3] = 6;
tms_pt[4] = 7;
tms_tmp = tms_pt[1];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
tms_tmp = tms_pt[2];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
tms_tmp = tms_pt[3];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
tms_tmp = tms_pt[4];
while(tms_tmp) {
data_parity =!data_parity;
tms_tmp = tms_tmp & (tms_tmp-1);
}
//std::cout <<"data_parity is "<<data_parity;
tms_pt[5] = data_parity;

if(rd_wr){
std::cout<<"\nSending Transaction->"<<a<<" Type: Read ";
} else {
std::cout<<"\nSending Transaction "<<a<<" Type: Write ";
std::cout<<"\n\tData 0 byte "<<std::hex<<int(tms_pt[1])<<" ";
std::cout<<"\n\tData 1 byte "<<std::hex<<int(tms_pt[2])<<" ";
std::cout<<"\n\tData 2 byte "<<std::hex<<int(tms_pt[3])<<" ";
std::cout<<"\n\tData 3 byte "<<std::hex<<int(tms_pt[4])<<" ";

}
swd->AddSequenceAndStream(53,(unsigned char*)tms_pt,(unsigned
 char*)tdi_pt);
swd->Send(0);
uint8_t *out_bit = new uint8_t [8] ;
```

Feedback

```
if(swd->GetTDOBits(53,out_bit))
{
std::cout<<"Error in reading ";
}
uint8_t *result = new uint8_t [8];

uint8_t ack;
ack = 0;             uint8_t pos =0;

if(rd_wr)
pos=2;
else
pos=6;

ack = out_bit[pos] >> 2 & 0x1;
ack |= ((out_bit[pos] >>3) & 0x1) << 1;
ack |= ((out_bit[pos] >>4)& 0x1 )<< 2;

if(ack==1) {
std::cout<<"\nAck for Transaction "<<a<<" is OK \n";
} else if (ack==2) {
std::cout<<"\nAck for Transaction "<<a<<" is WAIT \n";
} else {
std::cout<<"\nAck for Transaction "<<a<<" is ERROR \n";  }
```

5 Tutorial

The transactor package is delivered with several examples, which describe how to use the ZeBu SWD transactor in conjunction with a DUT. The testbench is a C++ program driving the transactor which:

- Creates ZeBu SWD by instantiating xtor_jtag_swd_svs object
- Configures the transactors
- Sends and receives data

This section explains the following topics:

- Example Structure
- Running on ZeBu

Example Structure

The following is the directory structure available in the Example directory for the *xtor_jtag_swd_svs* transactor.

- *src*: Contains the following sub-directories
    - *dut* : Contains test hardware top files
    - *bench*: Contains testbench top and common files used by the testbench under the following sub-directory:

tests: Contains all tests case files

- *env*: Contains the environment files needed for compile
- *zebu*: Contains Makefile.

Running on ZeBu

To use the example, ensure that your transactor is installed correctly and the *ZEBU_IP_ROOT* environment variable is set accordingly.

Also, you can compile and run the example on any ZeBu Server system, as explained in the following sections:

- Compiling for ZeBu
- Running the Testcase

Compiling for ZeBu

The compilation flow is available in the Makefile provided in the *example/zebu* directory. You can launch the compilation using the following compilation target:

```
$ make ZEBU=1 compile
```

The project files are available in the *example/src/env* directory.

Running the Testcase

After successful compilation, use the following command in the *example/zebu* directory.

```
$ make ZEBU=1 run TESTNAME=<test case name>
```

Example: $ make ZEBU=1 run TESTNAME=trivial_test

Test Cases on ZeBu

The *example/src/bench* directory contains test cases for the ZeBu SWD transactor. The following table lists the test cases with description:

Table 6 TestCases

Test Case	Description
ts.trivial_test.hh	Test case with SWD transactor in standalone mode, sending read/write packets.
ts.server_test.hh	Test case with SWD transactor initialized in server mode.
client_test.cc	Test to connect to transactor running in server mode and send packets over TCP IP.