STARC Compliant Models

Software for Semiconductors

Document History:

Date	Name	Comment	Version
2nd March, 2009	CircuitSutra	Created first draft	0.1

Table of Contents

1.Purpose

The purpose of this document is to explain the functionality and the architecture of models developed using the TLM2 library and STARC guidelines.

2.Scope

This document explains the modeling of the various IPs using STARC TLM guidelines. Following two models are created as per the STARC TL Guidelines.

- General purpose timer (based on 8254).

You can get the specs of the 8254 from:

www.stanford.edu/class/cs140/projects/pintos/specs/**8254**.pdf

- Programmable Interrupt controller (based on 8259)

You can get the specs of 8259 from:

www.ee.hacettepe.edu.tr/~alkar/ELE414/**8259**.pdf

Rest of this document explains the implementation of General purpose timer using TLM2.0 and STARC TL guidelines.

3.Timer 8254

3.1 Block Diagram of Timer

For the details of functionality please refer to the specification of 8254. Following is the high level block diagram of the timer 8254.This diagram shows the basic components in the 8254 timer, which includes three counters and a Control Word.

Block diagram of timer 8254[Signal level]

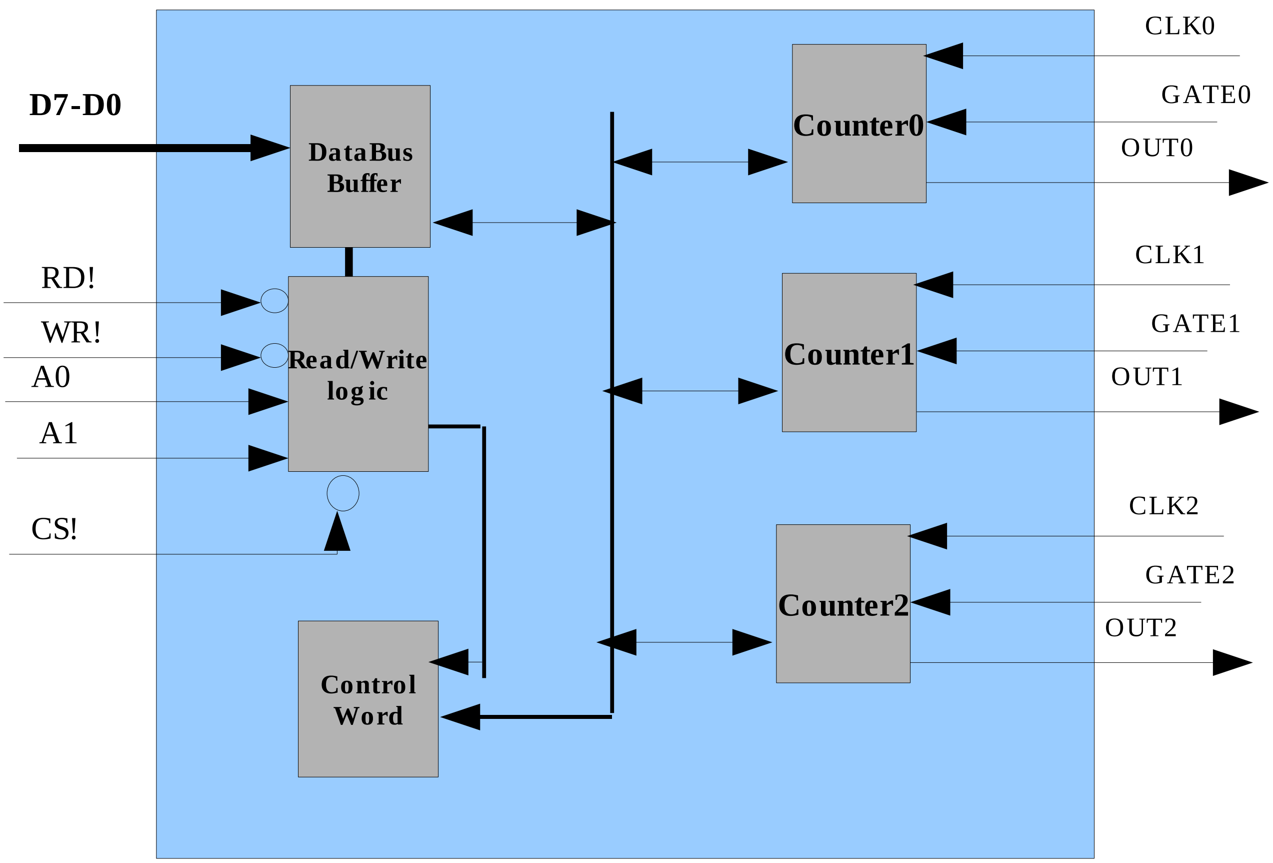


Figure 3.1

3.2 Architectural Diagram of Model:

Timer 8254 IP has been modeled using the TLM2 library. It has been modeled using the STARC guidelines and thus the computation and the communication part of the IP has been separated out as mentioned. The Model can handle both blocking and non-blocking transport calls. It has taken care of the reusability and interoperability in the model. Following block diagram gives a brief idea about important blocks of the Timer8254 IP.

Block diagram of timer 8254[TLM level]

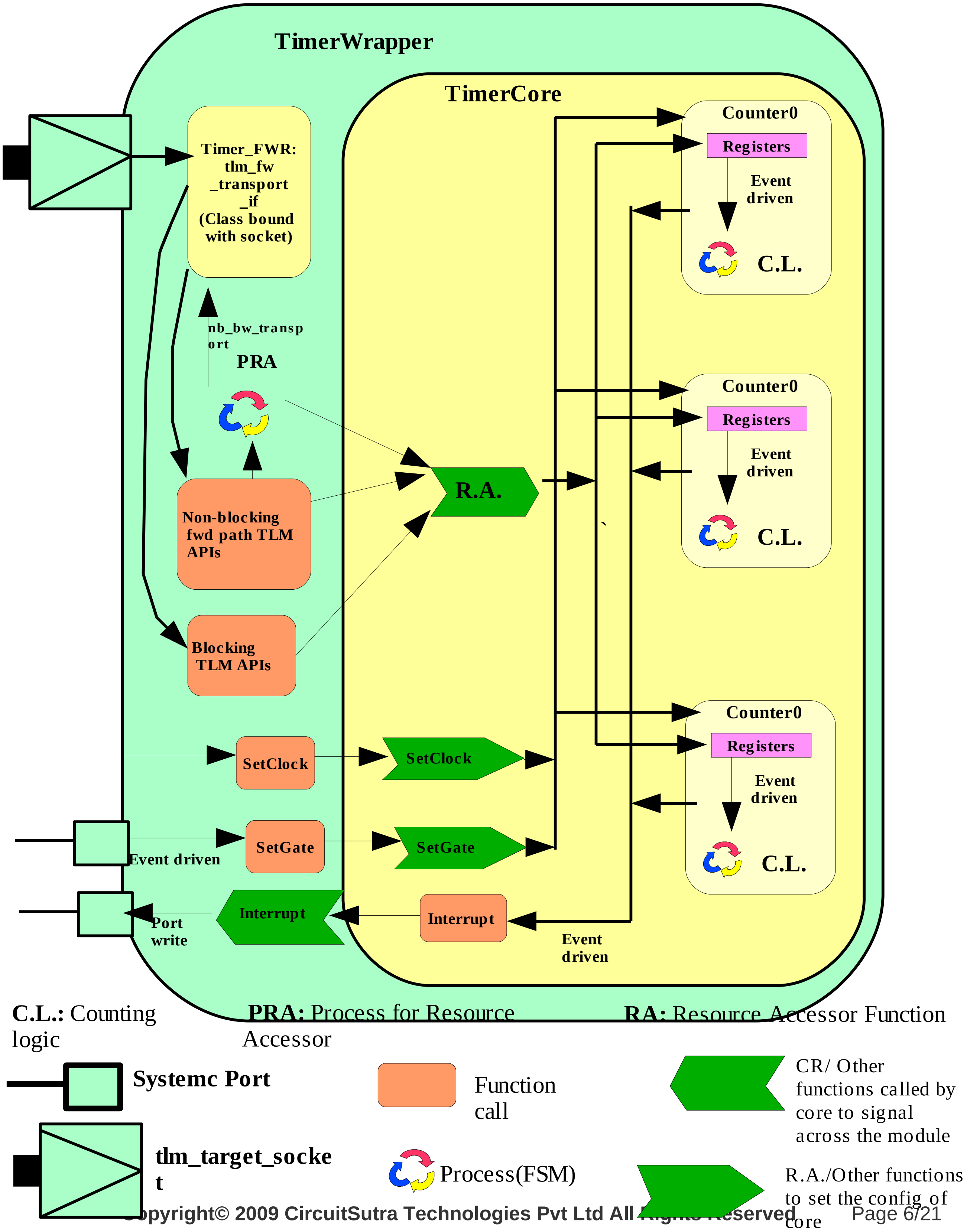

TimerWrapper
TimerCore
Timer_FWR: tlm_fw _transport _if (Class bound with socket)
nb_bw_transport
PRA
Non-blocking fwd path TLM APIs
Blocking TLM APIs
R.A.
Counter0
Registers
Event driven
C.L.
Counter0
Registers
Event driven
C.L.
Counter0
Registers
Event driven
C.L.
SetClock
SetClock
Event driven
SetGate
SetGate
Port write
Interrupt
Interrupt
Event driven
C.L.: Counting logic
PRA: Process for Resource Accessor
RA: Resource Accessor Function
Systemc Port
Function call
CR/ Other functions called by core to signal across the module
tlm_target_socket
Process(FSM)
R.A./Other functions to set the config of core

Figure 3.2

Block Diagram of Initiator:

Following section gives a brief idea about the important functional blocks of the Initiator IP, which has been used for this model. While creating initiator also, STARC guidelines have been followed properly. The implementation of initiator is specific to the user who is using this timer IP. It may be changed according to the requirement.

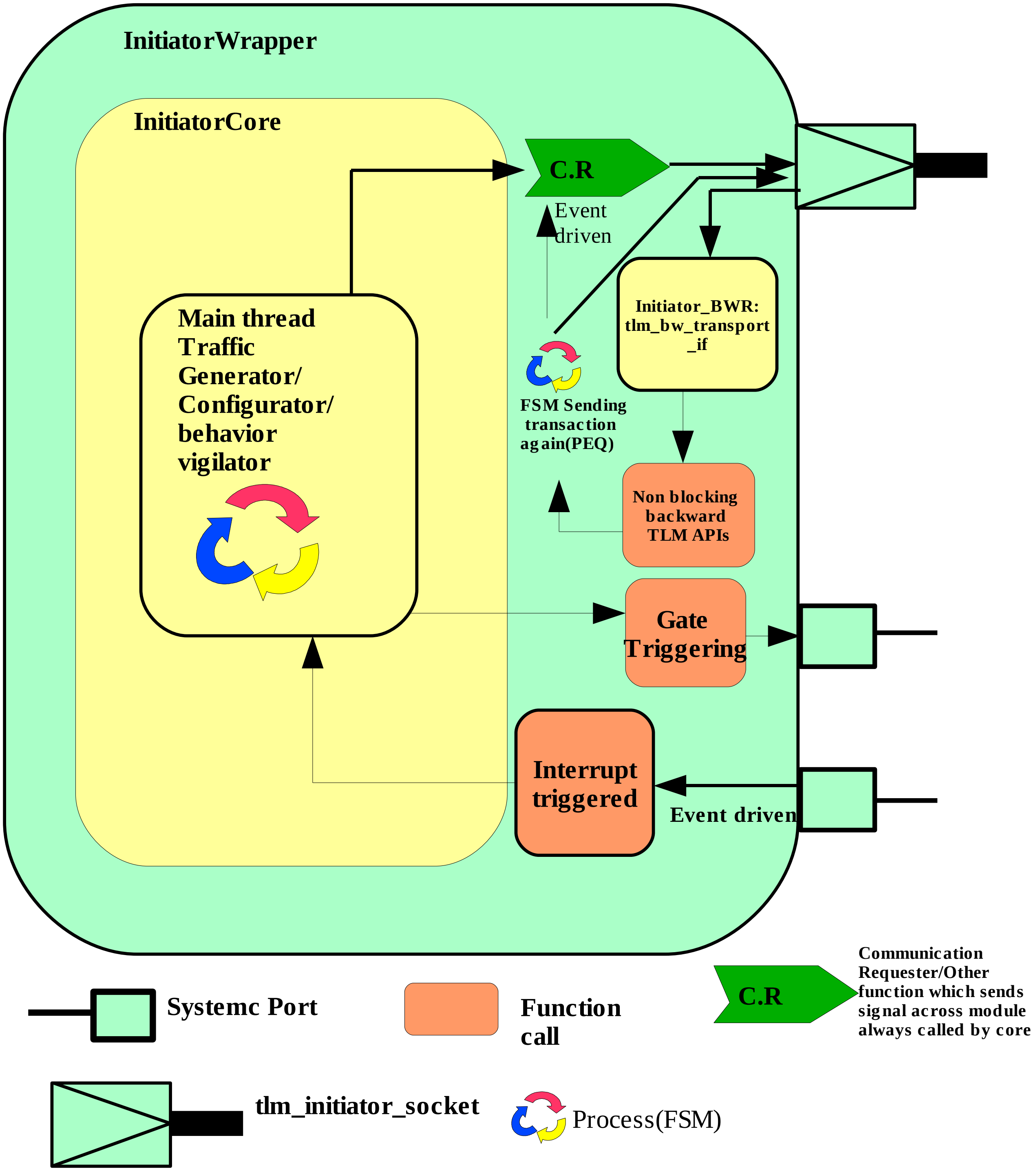

InitiatorWrapper
InitiatorCore
C.R
Event driven
Main thread Traffic Generator/ Configurator/ behavior vigilator
Initiator_BWR: tlm_bw_transport _if
FSM Sending transaction ag ain(PEQ)
Non blocking backward TLM APIs
Gate Triggering
Interrupt triggered
Event driven
Systemc Port
Function call
C.R
Communication Requester/Other function which sends signal across module always called by core
tlm_initiator_socket
Process(FSM)

Figure 3.3

Connections of the timer with the Initiator IP:

Following diagram gives a brief idea about how the timer will be connected to the Initiator IP. The initiator will basically configure the timer into various modes, generate the read/write requests traffic and observe the behavior of timer as per configuration. It may generate notify about any error conditions also.

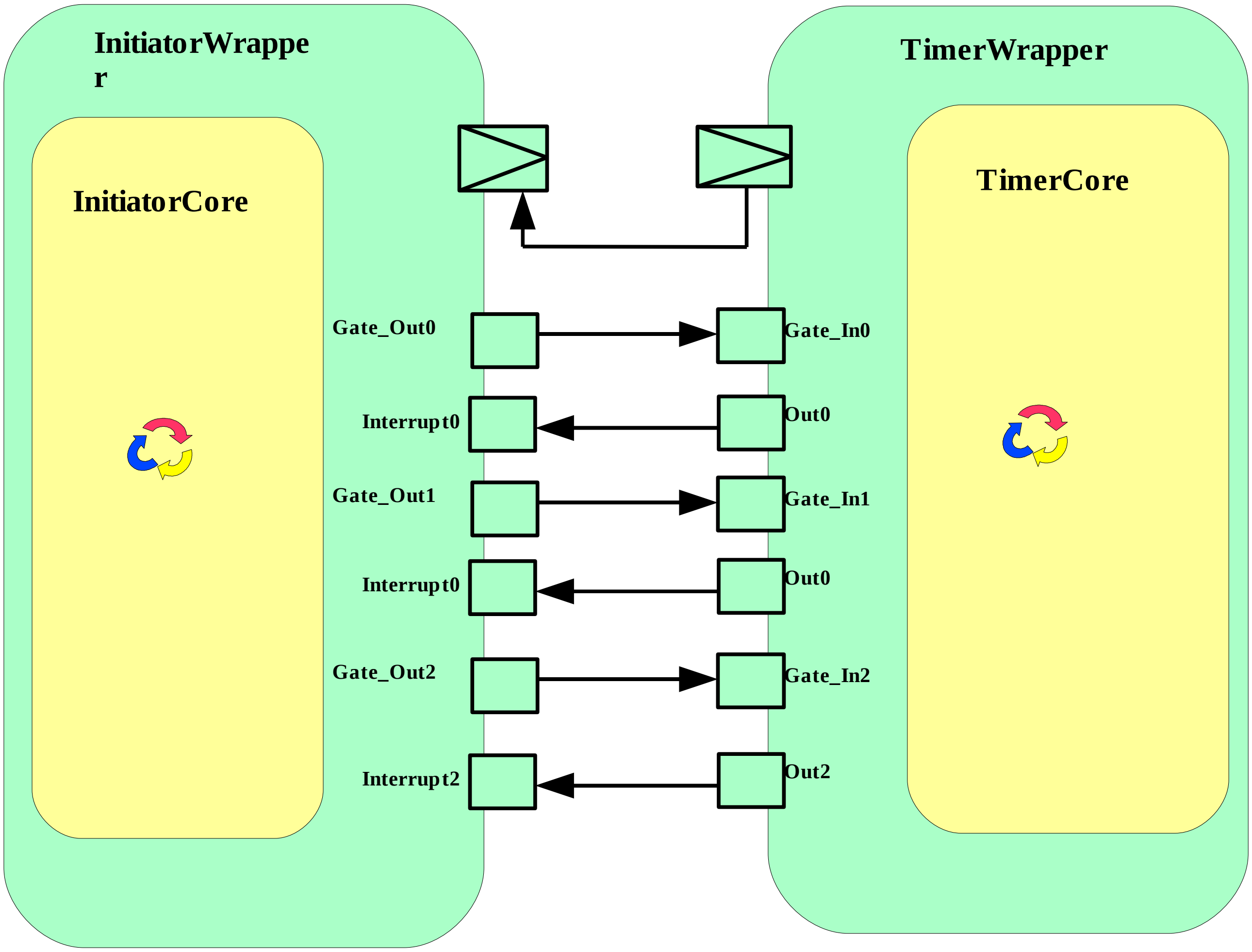


Figure 3.4

4.STARC TL Guidelines followed in the model

4.1 Separation of Communication and Computation:

As STARC guidelines suggest to separate out the computation and the communication to increase the reusability of the computation part. The computation is defined in the core and the communication is defined in the wrapper. Both Wrapper and the core are bind with each other. We have followed the guidelines by the STARC section 5.3.2 here.

4.1.1: Cores:

STARC explains how to create cores and what essential blocks should we keep in the core in section 5.3.

In this model, the initiator core does not have any port or TLM socket. It is simply bound with the wrapper which has systemc ports and the TLM sockets. The main traffic generation part is inside the initiatorcore and it calls the **CR[communication requester]** functions defined in the wrapper. Even for writing/reading some value on the ports defined in the Wrapper, the core calls some port access functions of the wrapper.

At target side, the TimerCore has been created, which is also bound with the TimerWrapper. Whenever any Read/Write requests come to the TimerWrapper it calls the **RA[Resource Accessor]** functions of the TimerCore. In this model, the TimerCore has instances of Counter, which is the main computation logic. The RA function of the TimerCore basically access the registers of Counter and perform the required operation. The Counter starts counting depending upon the values written on the control registers.

4.1.2: Wrappers:

While creating the wrappers for the timer and the initiator, **STARC** guidelines(**section 5.4)** have been used.

InitiatorWrapper instantiates the **tlm_initiator_socket** and provides the implementation of **tlm_bw_transport_if** APIs. InitiatorWrapper contains the **CR** functions which are called by the core for executing any transaction. **TLM API** calls through sockets have been used in the **CR** functions.

TimerWrapper instantiates the **tlm_target_socket** and provides the implementation of **tlm_fw_transport_if** APIs. TimerWrapper whenever receives any transport call, it calls the **RA** functions of the TimerCore.

4.2 Separate class for TLM APIs:

As recommended in the STARC guidelines, it is better to define the set of TLM APIs into a separate class instead of implementing them into the Wrapper class. This allows to have multiple set of TLM APIs if there are multiple sockets. This class is instantiated into the Wrappers.

On the initiator side, we have created **Initiator_BWR** class, which is instantiated inside the initiatorWrapper and bound with the **tlm_initiator_socket**. This class **Initiator_BWR** is derived from the **tlm_bw_transport_if** interface and provides the definition of backward tlm APIs.

On the target side, we have **Timer_FWD** class, which is instantiated inside the **TimerWrapper** and bound with the **tlm_target_socket**. This **Timer_FWD** is derived from **tlm_fw_transport_if** and provides the implementation of fwd tlm APIs.

The TLM protocol is standard and is independent of the functionality of IP. On the target side we have further seperated the TLM API implementation from the IP. The TLM api implementation is done outside of target wrapper, and the same implementation is used in both Timer and IP. Similarly a general purpose TLM implementation can be used on the intiator side also.

4.3 Abstraction Levels (Data granularity):

The model supports both the TR(transaction) level and the BP(Bus Phase) level of data granularity. In the initiator examples, there is a variable which can be set to indicate which abstraction level (data granularity) is to be used. In case of TR the initiator uses the use the blocking TLM calls to communicate with the timer IP, in case of BP non-blocking TLM calls are used. For setting the data granularity, just call this SetAbstraction() with proper value and it will set the data granularity either to TR/ BP. The number of phases in the BP, depends upon various latencies.

4.4 Abstraction Levels (Timing):

This model supports both untimed and the approximate timed models. It basically supports UTTR, ATTR and ATBP abstraction levels. The timing abstraction is based on the latencies. The user of Timer IP can set three latencies for the timer, as listed below:

1) Request Accept Delay: It is the time taken by the target(Timer in this model), to acknowledge the initiator after it has received the request from the Initiator. Or it may be defined as the interval between the beginning and the end of the request phase in non-blocking mode of communication.

2)Read Response Latency: This is the time taken by target IP in processing of the Read request. It is the interval between the end of request phase and the beginning of the response phase while the request is of read type.

3) Write Response Latency: This is the time taken by target IP in processing of the Write request. It is the interval between the end of request phase and the beginning of the response phase while the request is of write type.

On the initiator side there can be one latency:

1) Response Accept Delay: It is the time required by the initiator to accept the response form the target. Target should not send the response of next transaction unless initiator has accepted the response of previous transaction. It is the interval between the beginning and the end of the response phase.

We have created the model in such a way, that depending on the various combinations of these latencies, it automatically switch to the appropriate abstraction level.

If the Data granularity abstraction level is TR, and all the latencies are zero, then it behaves as UTTR. If the data granularity abstraction level is TR, and some or all of the latencies are non zero then it behaves as ATTR.

In case of data granularity abstraction of BP, the various TLM sequences are realized based on the combination of latencies. STARC section 6.2 suggests how to introduce various phases at ATBP level. The complete sequence of a transaction at ATBP level is a follows:

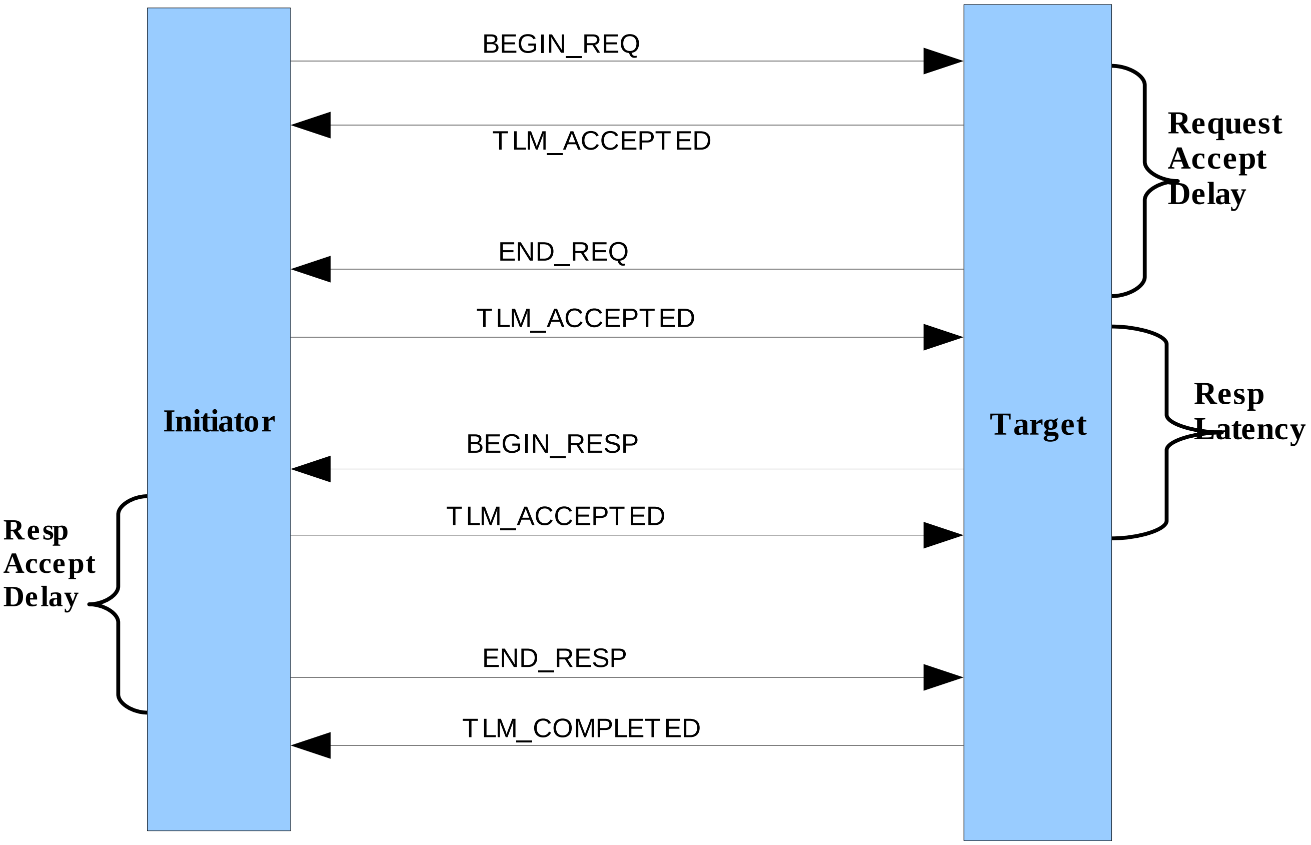


Figure 4.1

This is the complete sequence of the phases when all the latencies have been defined.but when any of them is missing, the transaction follows any of the shortcut mentioned in the STARC section 4.1 (page 4-18).

Following table shows the various shortcuts which will be realized in the various conditions of latencies.

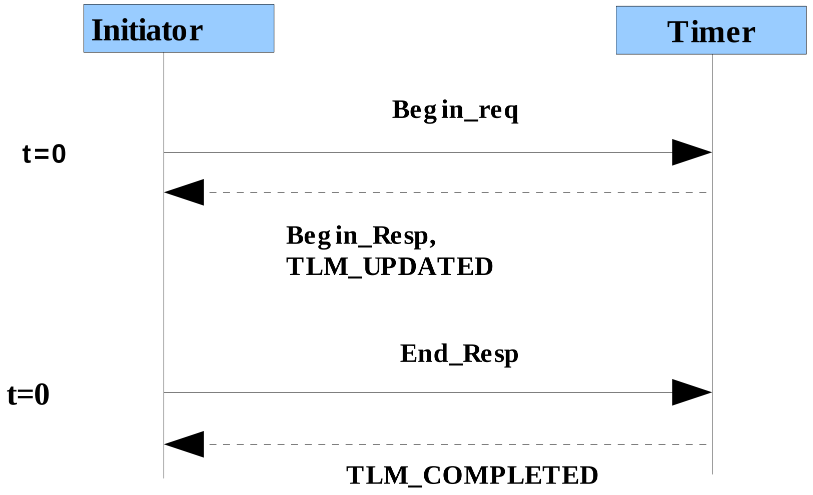
Case	Request Accept Delay	Response Latency	Response Accept Delay	ShortCut
Case1	0	0	0	Shortcut 5
Case2	0	0	1	Shortcut 5
Case3	0	1	0	Shortcut4
Case4	0	1	1	Shortcut5
Case5	1	0	0	Shortcut3
Case 6	1	0	1	Shortcut5
Case7	1	1	0	Shortcut4
Case8	1	1	1	Complete Sequence

Note:1 means non-zero value, here it is not representing the exact value of time.

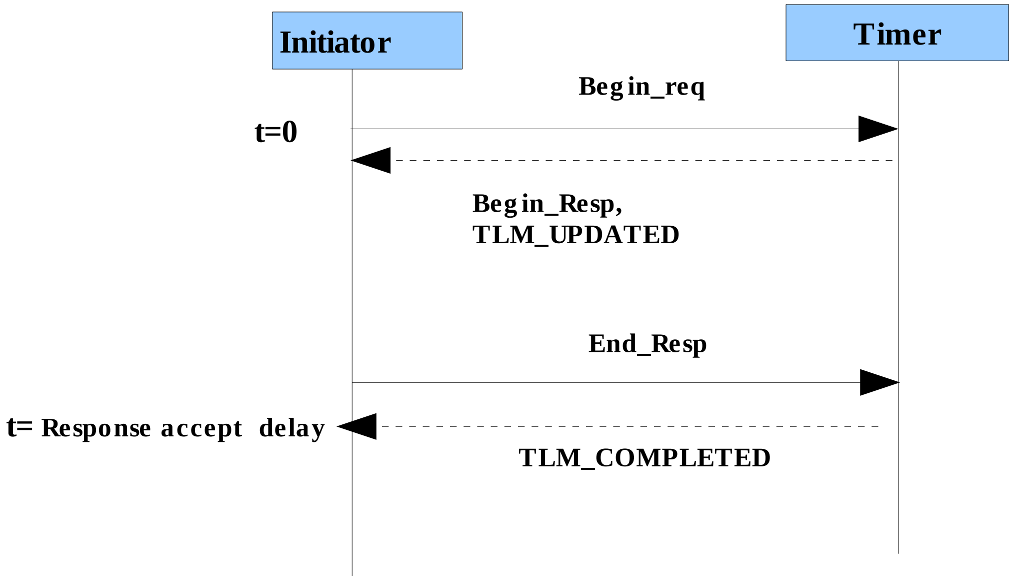
Note: In this examples we have implemented various shortcut sequences just on the basis of latencies specified by the user. However, there may be several other criteria for choosing a particular sequence depending on the functionality of IP modeled.

Here are the sequence diagrams showing the forward and backward calls as per the above table:

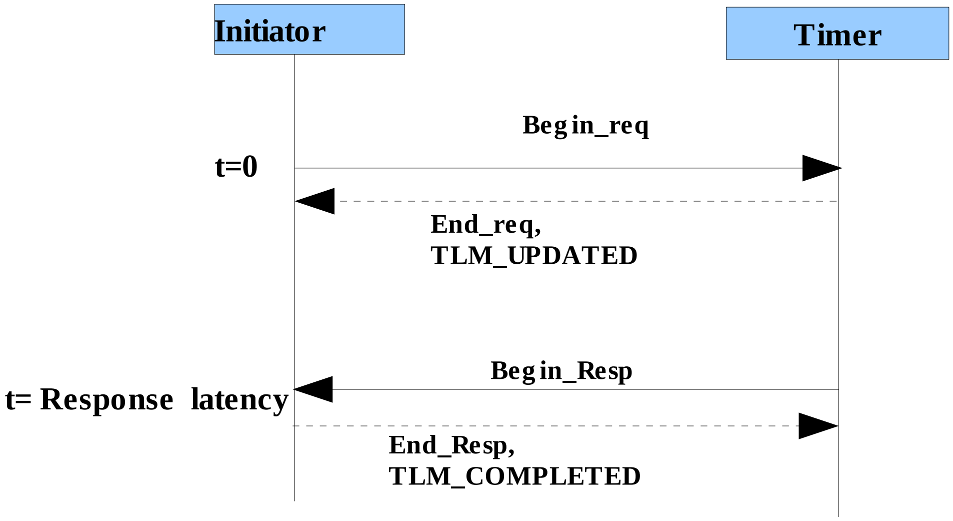
Case 1:[Request accept delay=0, response latency=0, response accept delay=0]



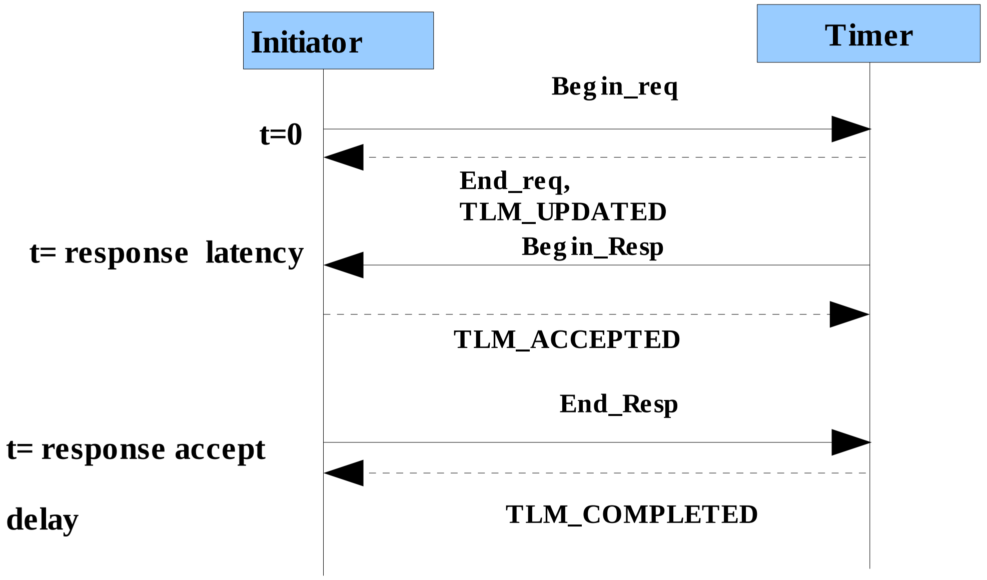
Case 2:[request accept delay=0, response latency=0, response accept delay =1]



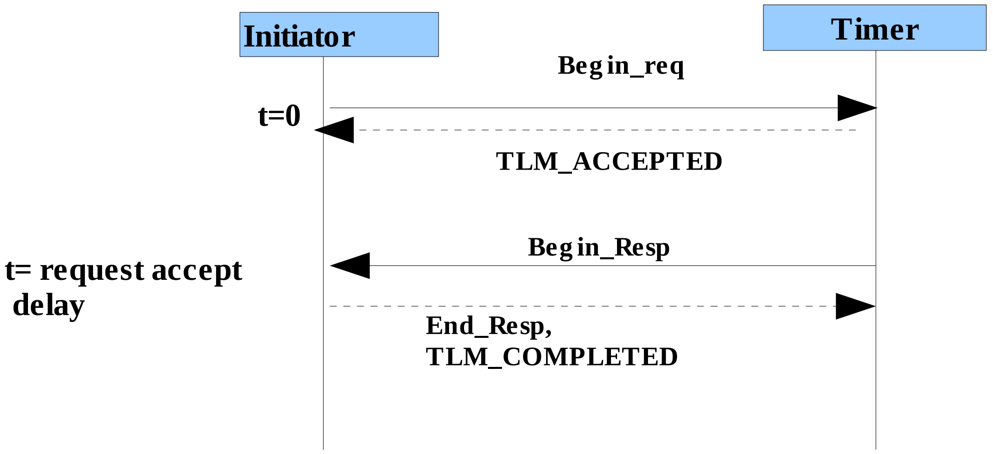
Case 3: [request accept delay=0 , response latency=1, response accept delay=0]



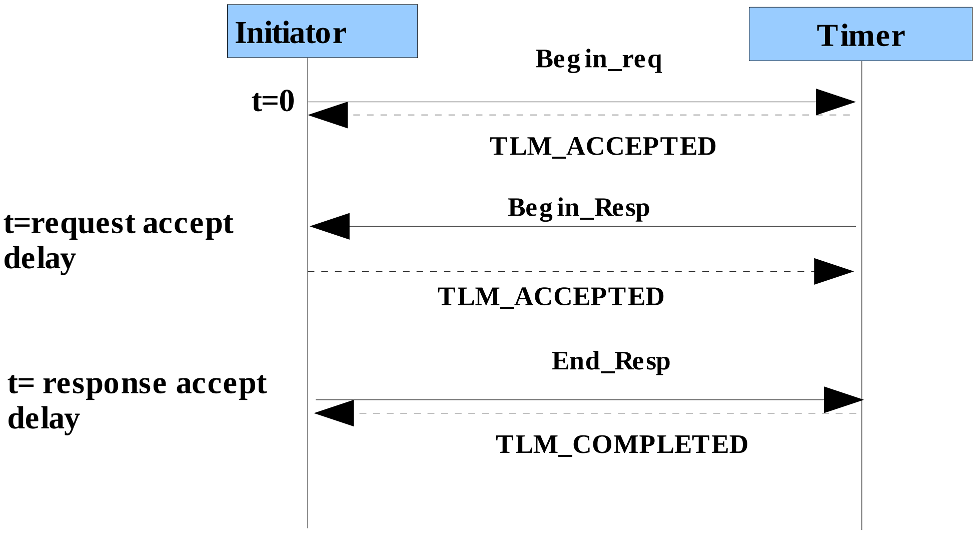
Case 4: [request accept delay=0, response latency=1, response accept delay=1]



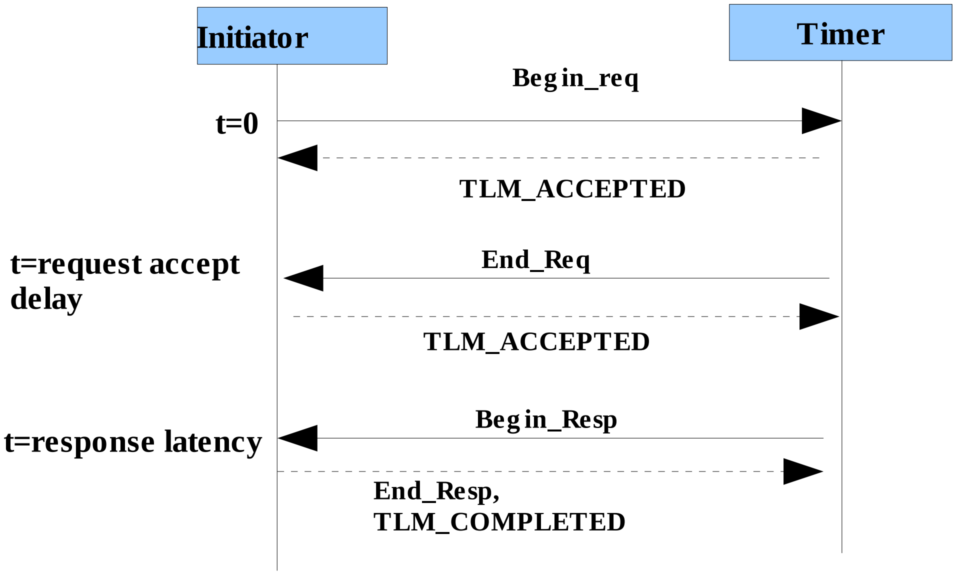
Case 5:[request accept delay=1, response latency=0, response accept delay=0]



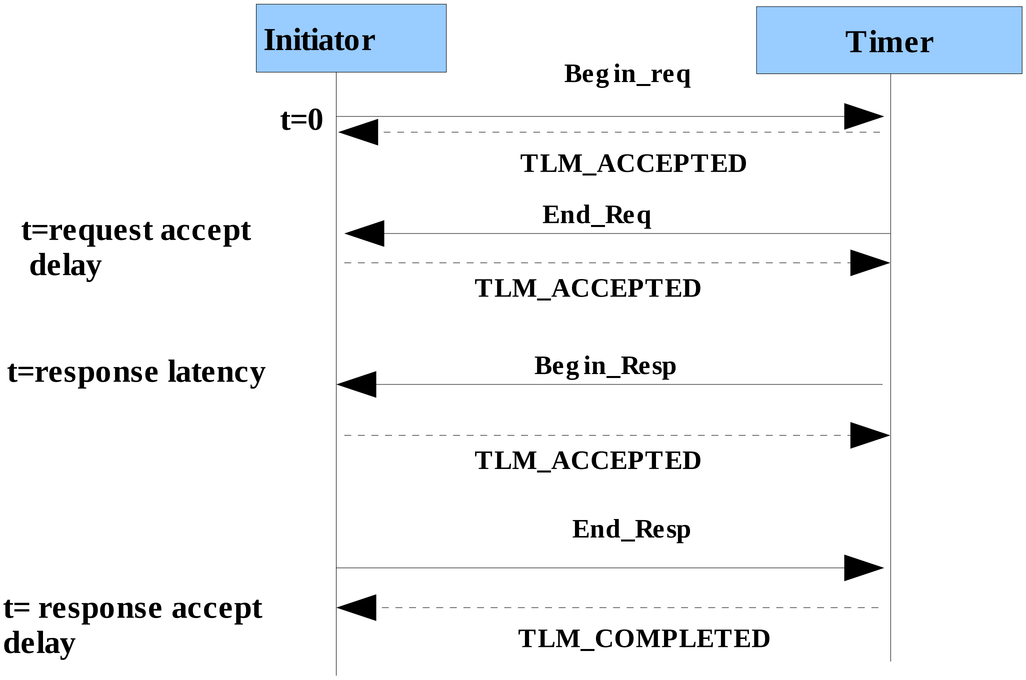
Case 6:[request accept delay =1, response latency=0, response accept delay=1]



Case 7:[request accept delay=1, response latency=1, response accept delay=0]



case 8: [request accept delay=1, response latency=1, response accept delay=1]



4.5 Concurrent Execution of Communication and Computation:

STARC section 6.2.3 discusses the concurrent execution of communication and computation on both initiator side and target. By concurrency, it means that at initiator it will be achieved by keeping different processes for sending requests and receiving responses. Similarly at the timer end, the requests should be accepted in different process and the response should be sent into separate process.

Initiator:

In current model, on Initiator side, basic TLM handling code for ATBP mode supports concurrency by having different process for request and response. It provides the provision to initiate the next transaction even if the previous transaction is not yet completed, thus there can be various concurrent transactions. It has to follow the basic TLM2 rule:

-Initiator can not send another request until it has received the END_REQ/BEGIN_RESP phase of previous transaction.[**ref TLM2 User guide sec 7.2.4 b**].

However, in the current examples, we do not initiate concurrent transactions, the control from initiator wrapper reaches to core only after completion of a request.

Timer:

Concurrency in target side is achieved in two ways:
1. The counting logic(computation part) of the timer runs exclusive and independent of the communication in process.

2. In case of ATBP, the request and response are handled by separate process. We have used PEQs(payload event queues) to store the pending requests and the processes which processes these requests and send responses to the Initiator. At timer also, following rule is needed:

- Target shall not respond with BEGIN_RESP phase until it has not received the END_RESP phase/TLM_COMPLETED for the previous transaction[**ref TLM2 user guide sec 7.2.4 c**]

4.6. Unit Testing:

STARC suggests that for functional verification of the model, not only integration test environment is created but unit tests are also created. All the significant features of the Timer8254 have been covered in unit testing. For unit testing, a common top level module Testbench has been created and then at the testcase level there is testcase class. The testbench class is common to all the testcases.

Testbench: In the testbench module; initiator and timer are instantiated and then bound together. Apart from binding, testbench class has some public functions which can be called from outside or derived class for setting all the latencies, base addresses, clock frequencies and data-granularity(TR/BP) for the initiator and the timer IP.It has a virtual run() method for which the derived class may provide specific defintion.

Testcase: Each Testcase class is derived from Testbench. Each testcase provides its own definition of the run method of parent class. In each testcase, the configuration of latencies and clock frequencies can be different. For each feature of the TimerIP, there is separate testcase and they can be build on both linux and windows machine.

5. Directory Structure:

The STARC_models directory has several subdirectories, just for the ease of the user.

Dirs:

commonCode: This directory contains the code which is being used by different IPs.

Timer_8254: This directory contains the implementation of the Timer8254, documentation, testcases and examples.

/IP: This sub-directory contains the main implementation of timerIP and its counters.

/common: This subdirectory contains the implementation of the initiator which is main traffic generator. The same InitiatorWrapper and core have been used for examples and unittests.

/unit_test: This subdirectory contains exhaustive testcases for various features of the timer8254.

/examples: This subdirectory contains various examples of timer model. The ReadMe file explains more about it. The model has shown the behavior of the model at different abstraction and data granularity level.

/ docs: It contains the featurelist which maps with testcases covered for each feature of the timer IP.

PIC_8259: This directory contains the implementation of the PIC 8259, documentation, test cases and examples. It shows the re-usability of code across various IPs.

6.How to build and run:

This package provides some sets of unit tests and examples directory for the user to understand the implementation and model. Please follow the instructions mentioned below for building them. Note that including path for Boost headers is only required for PIC_8259 model.

6.1 Windows:

6.1.1 Using Solution file:

1) In each directory of example, there is a directory build-msvc.

2) Open the solution file in MSVC++9.

3) Right-click on the Project, go to properties.Now click C/C++ ->General->Additional Include directories.

4) Now add the SystemC headers(systemc.h) here.

5) Add the path for the tlm headers (tlm.h).

6) Add the path for Boost Headers(function.hpp)[required for PIC_8259 only]

7) Now click linker->general->Additional Library Directories.

8) Add the path for the systemc.lib here.

9) Now build the solution.

10) Run the executable generated in either Debug/release directory as per the configuration set in your project.

Note: Alternative solution for Step 3,4,5,6,7 is to follow the steps given below:

1) go to tools->options->Projects and Solutions->VC++ Directories.

2) In the Combobox “Show Directories For”, select Include files and add the path for systemc.h, tlm.h and

boost header(function.hpp)[boost headers required for PIC_8259 only]

3) Then again in the same Combobox select “Library files” and add the path for the library “ systemc.lib”.

6.1.2 Using Makefile:

1) In the directory examples/config_msvc, there is Makefile.config, which is being used by all the examples. For unit tests the file name is Makefile_msvc.config in the directory unit_test/.

2)In this file change the following variables as per your systemc environment and installation:

a) SYSTEMC_HOME

b) TLM_HOME

c) BOOST_DIR[required for PIC_8259 only]

d) FLAGS, add the Visual studio required variables like “Microsoft Visual Studio 9.0\VC\include”, “Microsoft Visual Studio 9.0\SDK”

e)LDFLAGS change the path for the Visual Studio SDK library (“Microsoft Platform SDK\Lib”)as per your system

3) Now simple go to Programs->Visual C++ 9.0 Express Edition ->Visual Studio tools-> Visual Studio 2008 command prompt.

4) Now go to the particular testcase(for e.g: unit_test/Test1/Test1.1) or build-msvc directory of particular example(for e.g: examples/Example_nb_000/build-msvc) which you want to build.

5) now simply run the nmake utility

```
>nmake
```

For unit tests, you will have to run

```
>nmake /F Makefile_msvc.
```

It will create all the object files and the required executable.

6.2 Linux:

The procedure for linux is easier one. Follow the steps below:

1) Go to the unit_test/ directory or examples/config-linux/ directory.

2) Edit the Makefile.config present there and change the SYSTEMCDIR and TLMDIR variables as per your system. Also edit the SYSTEMCLIB.

3) Now simply go to the particular testcase(for e.g: unit_test/Test5) or build-linux directory of the particular example(for e.g: examples/Example_nb_000/build-linux) that you want to build.

Run make command

```
>make
```

4) You will get the executable in the same directory.

Note: All the testcases can be run in a single go using the perl script "**runAll.pl**". This script is valid for both linux and windows. This script searches for the makefile in all the individual testcase directories and then run it, which creates the necessary object file and executable. This script run all the testcases and store their results in **runAllStatus**. So just by viewing the **runAllStatus,** one can analyze how many tests have been passed and how many have failed.It also specifies which testcases have failed and which have passed.

References:

1. **OSCI TLM2 User Manual (version JA22)**
   [http://www.systemc.org/home]

2. **STARC TLM Guide (second edition)**
   [http://www.starc.jp/index-e.html]

3. **Specs of Programmable Interrupt Timer8254**
   [www.stanford.edu/class/cs140/projects/pintos/specs/8254.pdf]