Trace framework User Guide

Version 0.9

Revision Record

Version No.	Date	Comments
0.1	February 13, 2012	Initial Version
0.2	February 15, 2012	Updated the ladder diagram with the specific example use case
0.3	February 17, 2012	Updated the document after the code review inputs are implemented
0.4	March 19, 2012	Updated the document for API changes in producer/consumer libraries
0.5	January 7, 2013	Updated the document for new APIs in contract creation
0.6	January 15, 2013	Updated the UG for new APIs for performance Optimization of Buffer Exchange and Multi instance UIA support
0.7	February 10, 2014	Updated for new Examples
0.8	February 19, 2014	Enhanced steps on how to execute unit test
0.9	March 27, 2014	Added System Analyzer Configuration for CCS

Table of Contents

1. Concept

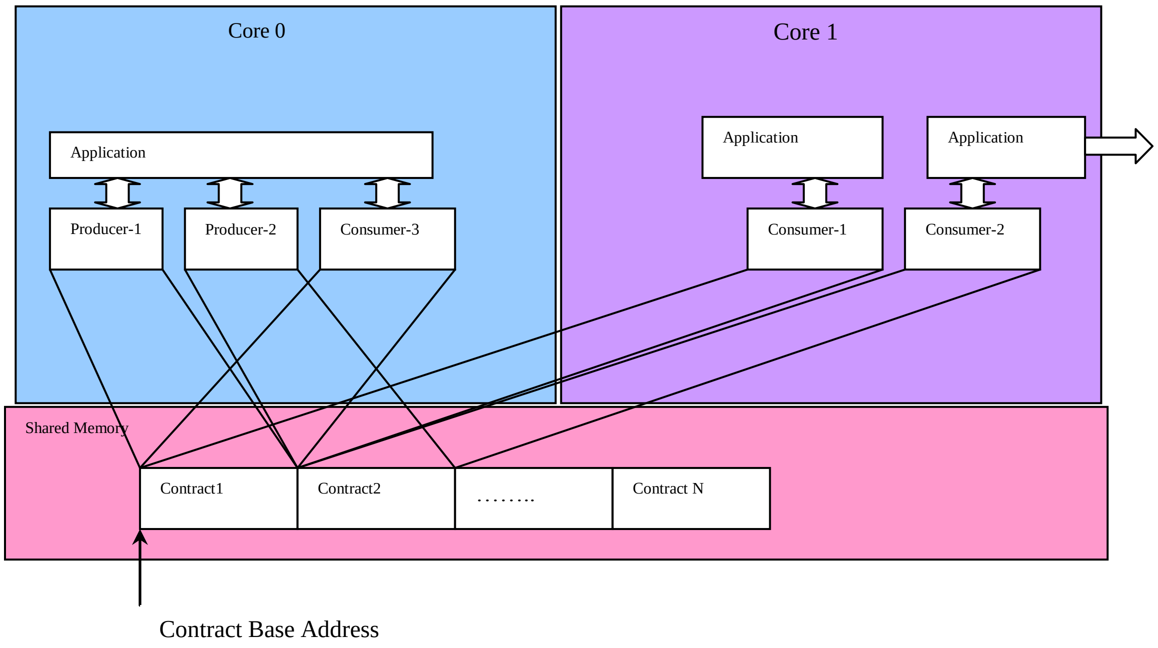
Introduction

In a multicore environment there would be data generated by a producer in one core and would need to be consumed by multiple consumers in other cores/same core.
Example: UIA log information to be consumed by ARM and System Analyzer.

The trace framework provides a mechanism to send the information to multiple consumers for a given producer.

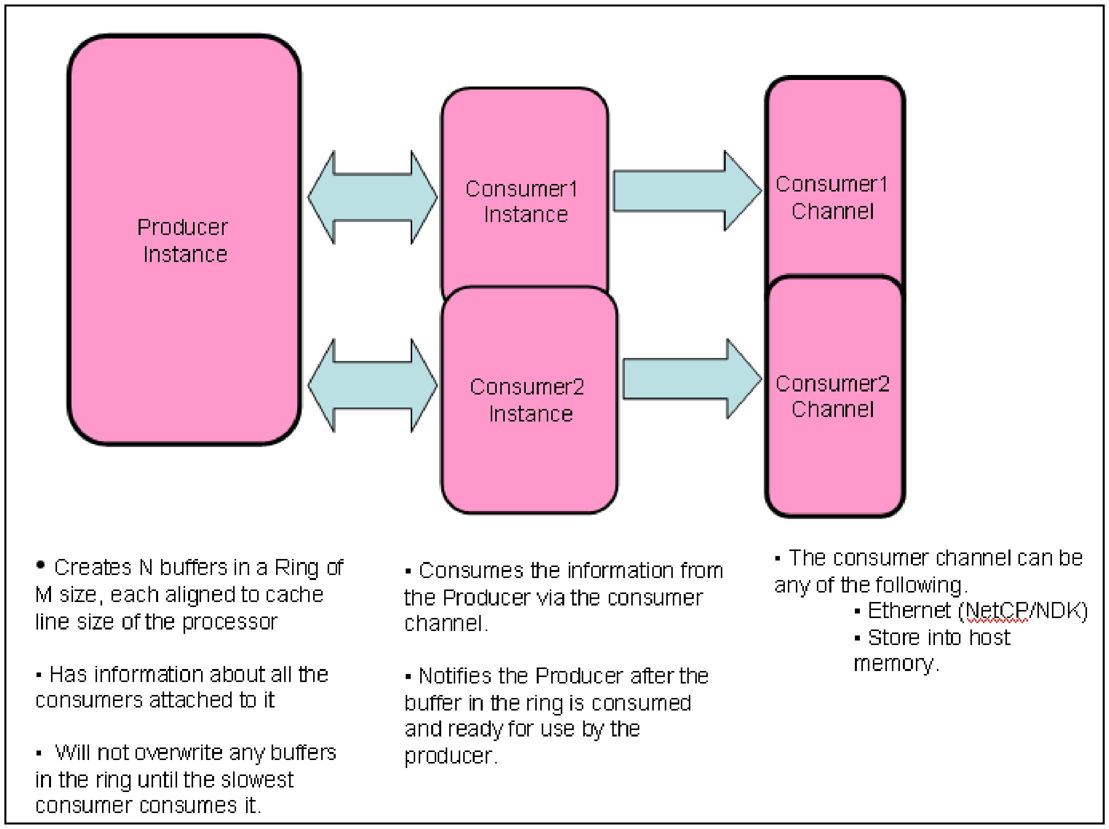
In trace framework, Producer populates the associated Ring buffer. Consumers consume the ring buffer and send the data to external actual consumers (like System Analyzer in CCS).

The below figure is an example system showing the producer, consumer and the transport interconnected via the contract.
In the system core 0 has two producer instances which produce information on two different rings. Core 0 also has a consumer instance (3) for producer-1. Core 2 has two consumers, consumer (1) for core0's producer 1, consumer (2) for core0's producer 2.



The information exchange synchronization between producers and consumers are done via the contract memory.

Each core can have multiple consumer Instances and Producer Instances. Each Producer instance would be tied to a ring buffer and many consumer instances can consume data from a ring buffer. Every consumer needs to have its own channel to send the data out, which triggers the draining of the ring buffer.


Producer Instance
Consumer1 Instance
Consumer2 Instance
Consumer1 Channel
Consumer2 Channel
• Creates N buffers in a Ring of M size, each aligned to cache line size of the processor
• Has information about all the consumers attached to it
• Will not overwrite any buffers in the ring until the slowest consumer consumes it.
• Consumes the information from the Producer via the consumer channel.
• Notifies the Producer after the buffer in the ring is consumed and ready for use by the producer.
• The consumer channel can be any of the following.
• Ethernet (NetCP/NDK)
• Store into host memory.

2. Trace Framework

2.1. Ring Producer

The ring producer produces the trace/log information in a ring buffer and notifies the consumers hooked to that ring when the ring buffers are ready. The producer fills up the information in the ring buffer sequentially, servicing the slowest possible consumer for that ring. So, a slowest consumer in the system can potentially make the fast consumers to drop messages.

The producers and consumers handshake through a shared memory region called as “Contract Memory”.

2.2. Consumer

The ring consumer reads the ring buffer from the producer and outputs it to the appropriate transport channel. The consumer updates the necessary control messages to the producer via the Contract.

2.3. Transport Channel for Consumer

Every Consumer instance for the ring buffer needs to have a transport channel. The transport channel triggers the drain for the ring buffers thereby allowing a free ring buffer to the producers.

2.4. Contract

This is a shared memory area between the producer and multiple consumers. Trace framework library provides the APIs to create the contracts, producer and consumers for the contract.

Please refer to *docs/Doxygen/html/index.html* for details on the API user guide and on rebuilding trace framework libraries and examples.

3. Migration Guide to UIA 1.1.2.23 or Higher

3.1. New APIs Added

1. Void * tf_getLoggerStreamerContext(LoggerStreamer_handle) – API added to get the producer handle to support optimal UIA buffer exchange performance for LoggerStrreamer Interface
2. void * tf_getLoggerStreamer2Context(LoggerStreamer2_handle) – API added to get the producer handle to support optimal UIA buffer exchange performance for LoggerStrreamer2 Interface
3. Ptr tf_uiaProducerBufExchange2(LoggerStreamer2_Handle handle, uint8_t *full) – API added to support multi instance UIA (loggerStreamer2 interface)
4. void tf_prodNotifyConsumers(tf_producer_HANDLE pHandle) – API added to support optimal UIA buffer Exchange scheame with notification to registered consumers

3.2. Changes in the Application for LoggerStreamer

1. Source files

   Changes may be needed to align to performance optimizations of trace framework as out lined in section 5.3.

2. .cfg file

Needs to add below lines in the.cfg file for trace framework library along with using the LoggerStreamer2 module and necessary configurations for LoggerStreamer module.

```
var LoggerStreamer2 = xdc.useModule('ti.uia.sysbios.LoggerStreamer2');
var LoggerStreamer = xdc.useModule('ti.uia.sysbios.LoggerStreamer');
```

3.3. Changes in the Application for LoggerStreamer2

- Application needs to migrate to UIA 1.1.2.23 or higher versions for the System Analyzer
- Application's .cfg file would now need to be configured to use both loggerSTreamer and LoggerStreamer2 as below

```
/* UIA 1.1.2.X onwards supports legacy loggerstreamer or multiintance
    loggerstreamer2 - use one of them in the configurations */
    var LoggerStreamer2 = xdc.useModule('ti.uia.sysbios.LoggerStreamer2');
    var LoggerStreamer  = xdc.useModule('ti.uia.sysbios.LoggerStreamer');
```

- Application should make sure either LoggerStreamer or LoggerStreamer2 is created for the system. There can be unpredicted behavior if both of them are created in the system.
- Sample .cfg configurations
  a. Sample Configurations for creating LoggerStreamer2 Objects with this new UIA (Please refer to UIA Multi Instance NetCpConsumer example project's .cfg file)
  b. Sample Configuration for Creating LoggerStreamer Object with this new UIA (Please refer to uiaprod_netcpconsumer example project's .cfg file).
  c. Sample configuration for creating the non UIA producer with Trace framework – please refer to uia prod4Arm example project's .cfg file

4. Running Unit Test

Trace framework provides DSP and ARM unit tests. Following use cases are supported.

Type	Producer	Consumer	Test Executable	
			DSP	ARM
UIA Producer/Consumer	DSP	DSP/ARM	tfw_Uia_UnitTest_XXXXBiosTestProject.out	tfwUiaArmConsumer.out
UIA Multi instance Producer/Consumer	DSP	DSP/ARM	tfw_UiaMinst_UnitTest_XXXXBiosTestProject.out	tfwUiaMinstArmConsumer.out
General Producer/Consumer	DSP	DSP	tfw_GenProd_UnitTest_XXXXBiosTestProject.out	N/A
CUIA Producer/Consumer	ARM	ARM	N/A	tfwcUiaProdConsumers

Note: Please note that ARM producer and DSP consumers use case is not supported in trace framework.
Note: ARM executables are not applicable for KeyStone1 devices like C6678, C7760 and C6657.

4.1 General guidelines to run the examples

At this time, it is assumed that the DSP and ARM executables mentioned in the above table are already built/available.

- Please update */traceframework/test/TFUnitTest/TFUnitTest_input.txt* file for Destination PC IP address and MAC Address.
- Load the DSP executable
- Setup System Analyzer every time a new DSP executable is loaded onto CCS. Please refer to System Analyzer user guide on how to setup System Analyzer.
- Run the corresponding ARM executable from the ARM user space (if applicable).
- Run the corresponding DSP executable.
- Results are printed on the console for DSP and ARM executables.

4.2 CCS Instructions

1. DSP Producers/DSP Consumers along with optional ARM consumers.
    a. Modify the IP and MAC addresses in TFUnitTest_input.txt (under *traceframework/TFUnitTest* folder)
    b. Import the Trace framework CCS projects that are built using *pdkProjectCreate* script.
    c. Build the Trace Framework Imported CCS projects and load the executables.
    d. Create a UIA configuration file with these DSP test executables as the end points and set up the event transport as IP/UDP where IP = uiaSystemAnalyzerIPAddress and UDP port = 1235.
    e. Load the executables through CCS on at least DSP Core0 and Core1

Note: Corresponding ARM consumer applications like tfwUiaArmConsumer.out and tfwUiaMinstArmConsumer.out can be run before we execute the DSP side corresponding producer application with the same name.

2. ARM Producers and ARM consumers
   a. Modify the IP and MAC addresses in TFUnitTest_input.txt (under *traceframework/TFUnitTest* folder)
   b. Build the traceframework ARM examples and copy to file system.
   c. Copy over the “cUIA Executable” executable that was created in the above steps and store it in a folder.
   d. In the Debug Perspective, open the System Analyzer Live Dialog (Tools / System Analyzer / Live) create a UIA config file by clicking on the "Create UIA Config File" button Create UIA Config File
   e. If there is no event transport defined, right click on the table and select "Add an event transport".
   f. Double click on the event transport and select UDP as the Transport Type, the IP address of the PC running System Analyzer as the address, and set the port number to 1235.
   g. If there is a control and status transport entry, and its type is not "None", double click on it and set the transport type to "None".
   h. If there is no endpoint defined, right click on the table and select "Add an Endpoint".
   i. Double click on the endpoint entry and enter the following:
      - Name = CPU0 (or some other useful identifier that indicates which CPU core will be running the test program)
      - Endpoint Address = 0
      - .out file - click the . button and navigate to the local copy of the “cUIA Executable” executable
      - .uia.xml file - click the . button and navigate to the local copy under */traceframework/metadata* folder and select cuia.uia.xml
      - .rta.xml file - click the . button and navigate to the local copy under */traceframework/metadata* folder select cuia.rta.xml
      - Clock Freq. (MHz): enter the speed of the timestamp you are reading in the main.c file's getTimestamp function. (If this is returning 0, then set it to the CPU clock speed in MHz - e.g. 1000 for a 1GHz device)
      - Set Cycles per tick to the number of CPU cycles each tick of the timestamp you are reading in the main.c file's getTimestamp function (if this is returning 0, then set it to 1).
      - Click OK to close the endpoint dialog
   j. Click Save to save the UIA config file to some location (e.g. the folder containing the local copy of the executable that you are running)

Back in the System Tools Live dialog, click ... button to the left of the "Create UIA Config File" button and select the .usmxml file you just created.

Ensure the "Until data transfer is manually paused" radio button is selected, and click Run.