TEXAS INSTRUMENTS
20450 Century Boulevard
Germantown, MD 20874
Fax: (301) 515-7954

Debug and Trace Library

Software Design Specification (SDS)

Revision A

QRSA000XXXX

March 12th, 2015

Revision Record	
Document Title: **Software Design Specification**	
Revision	Description of Change
A	Initial Release

Note: Be sure the Revision of this document matches the QRSA record Revision letter. The revision letter increments only upon approval via the Quality Record System.

TABLE OF CONTENTS

1 Scope

This document describes the functionality, architecture, and operation of the Debug and Trace Library which allows applications to collect and transport logs and instrumentation data.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

Table 2.1 References

No	Referenced Document	Control Number	Description
1	Syslib User Guide		SYSLIB User Guide
2	Debug and Trace API Documentation		Debug and Trace (DAT) Doxygen API documentation
3	System Analyzer User Guide		UIA/System Analyzer documentation
4			

3 Definitions

Table 3.1 Definitions

Acronym	**Description**
API	Application Programming Interface
DSP	Digital Signal Processor
DAT	Debug and Trace
NetFP	Network Fast Path
UIA	Unified Instrumentation Architecture
PDK	Platform Development Kit

4 Debug and Trace Library

4.1 Introduction

The debug and trace (DAT) library provides interfaces to configure and control generation of trace messages. The library is based on producer-consumer architecture, and provides a framework to maintain and enforce verbosity levels for the system components.

Logs are produced by multiple entities in the system called producers, and the logs generated by a producer can be sent to debug stream or consumed by multiple consumers. DAT requires

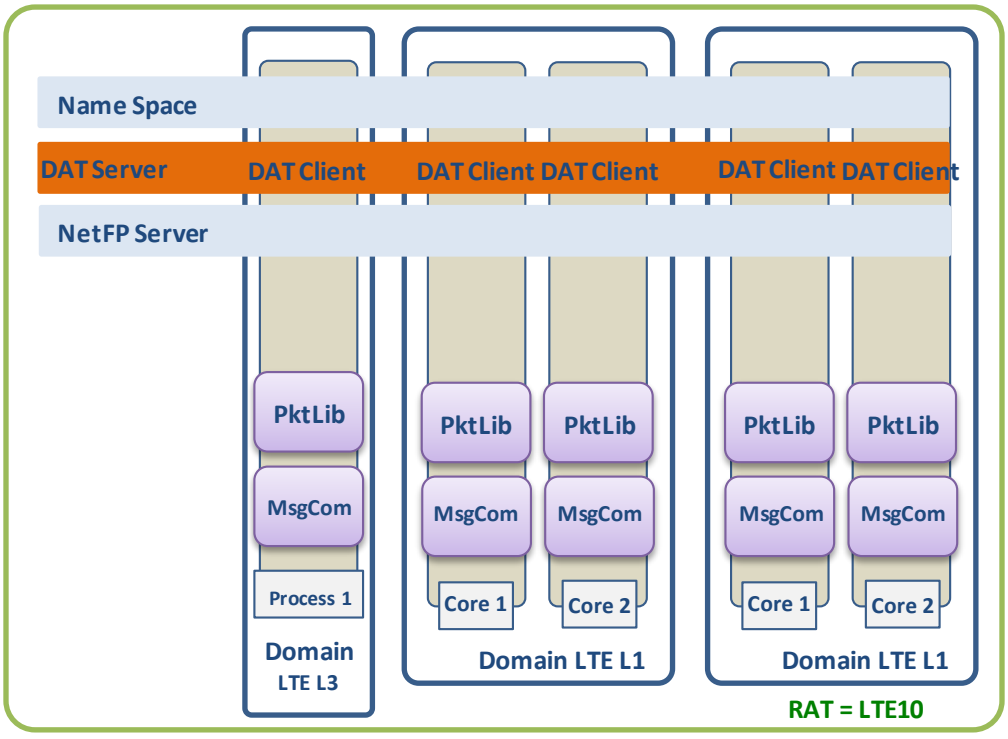
the log messages to be captured through logging APIs provided by the UIA LoggerStreamer2 module. Data buffers can also be logged through DAT using the general-purpose producer.

DAT manages a ring of buffers used for each producer, when buffer is full; buffer exchange is triggered to switch to a new buffer if available. Otherwise, it drops the contents of the current buffer and re-uses the buffer for the producer.

Generated logs can be sent to NetFP based debug stream, and/or consumed by multiple connected consumers. Log buffers can also be saved to pre-allocated memory buffers.

4.2 DAT library Architecture

DAT library uses server-client model to handle producer/consumer/trace objects reside on DSPs and ARM. DAT server has a centralized database to keep track of DAT entities - Producers/Consumers and global trace objects created from all DAT clients. DAT clients communicate with other clients through DAT server. Application interfaces with DAT library through DAT clients.

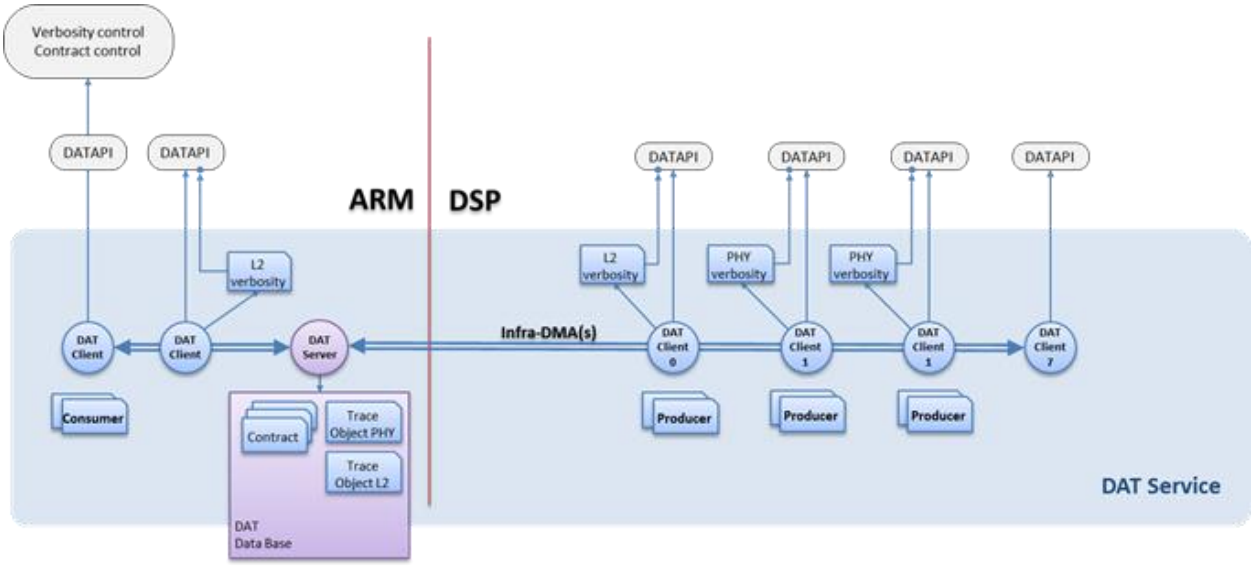


As shown in the above diagraph, in each RAT, one DAT server works with DAT clients within the RAT from DSP cores and ARM applications. DAT client is operational only once it has been initialized and registered with the DAT server. DAT server and DAT clients are communicated through msgCom queue DMA channels.

The DAT library provides the following features, which are explained in the subsequent sections:

- Create/delete producers(UIA and General Purpose) and consumers
- Set up Producer debug stream through NetFP for log buffers to be transmitted over Ethernet
- Set up Producer memory logging for log buffers to be save in DDR memory
- Send periodic LogSync messages to system analyzer for multicore event correlation
- Create DAT trace objects that contain default verbosity levels, defines list of verbosity levels
- Provide APIs to modify/query verbosity at runtime

The following picture demonstrates the main entities - producer/consumer/traceObject in a system.



4.3 DAT Producer

DAT Producers are entities that generate the logs and deliver the logs. Logs can be sent to Ethernet or be saved in on-board memory. There are two types of producers supported by the DAT library:

- UIA producer
- General-purpose producer

4.3.1 UIA producer

A UIA producer is an entity that provides and manages buffers used by the UIA library's LoggerStreamer2 module. The UIA producer maintains a circular ring of N equal-sized buffers. The UIA logging APIs write log messages to one of these buffers at any given time. When the

buffer is full, an exchange function is called by the LoggerStreamer2 that closes this buffer and fetches a new buffer. The location and management of these buffers are hidden from the application.

4.3.2 General-purpose producer

General-purpose producers provide and manage buffers used to collect statistics, protocol events or any other data. Here again, a circular ring of N equal sized buffers is used. In this case, the application can get the head of the ring and populate the data being logged. When the application is ready to ship this data buffer, the buffer exchange function can be called to close the current buffer and fetch a new one. The DAT library inserts a UIA header in this buffer so that it can be displayed in the System Analyzer. The library allocates memory for these buffers by taking into account the UIA header size, but the buffer pointer provided to the application skips the header room (see **Figure 4:1**).

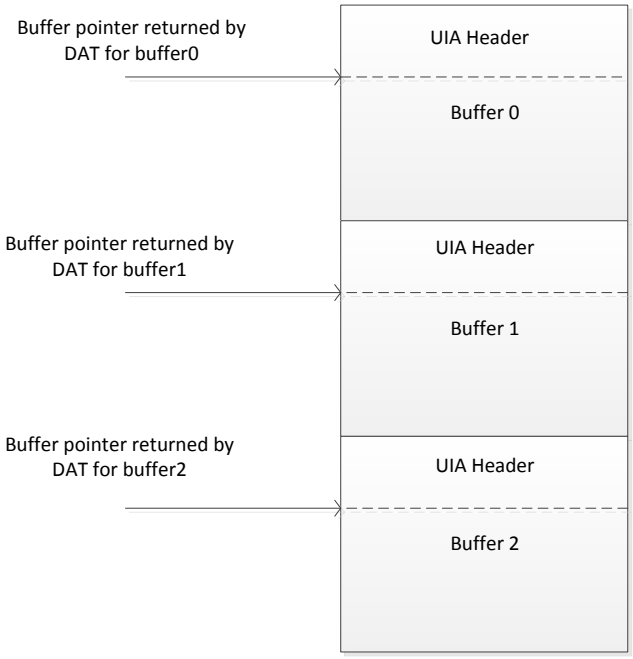


Figure 4:1 A ring buffer for general-purpose producer

For every DAT producer created (UIA or general-purpose), a corresponding LoggerStreamer2 instance has to be created and this handle is used by the DAT library for internal bookkeeping. LoggerStreamer2 instance can be created by application and provided to DAT library during producer creation. Or application can ask DAT library to create loggerStreamer2 instance by setting the logger handle to NULL. Multiple producers of different types can be created on each core.

Due to existing limitations with the general-purpose producer implementation, the maximum size of a buffer is restricted to 2000 bytes. If buffers of larger sizes are used, system analyzer may crash.

4.3.3 Log Buffer shipment

Log buffer exchange puts the old buffer in a pending queue to be processed in DAT producer background task. Producer background task will

1. retrieve the buffers in the pending queue,
2. write back the buffer content
3. and deliver the buffer to the configured one or multiple destinations.

The destination of a buffer can be a consumer, debug stream (NetFP based), and memory logging.

4.4 Consumer

A DAT consumer is an entity who is interested in the log buffer content a producer generates, such as PHY/LTE stack measures. A consumer can be created from any DAT clients. Producer buffers are delivered to **connected** consumers through Cppi DMA. Hence a heap needs to be provided to DAT library when creating a consumer. Application is responsible to call DAT API (Dat_processConsumer()) periodically to retrieve log buffers and free the buffer after consume the buffer. This has to be called on a per-consumer basis with consumer handle.

4.5 Background Actions

The following operations are running as background actions in DAT. They are normally called from a low-priority task. When Domain is available in the system, a low-priority task is created by Domain to handle DAT Client background actions. If Domain is not created, then executing these background actions should be handled by application.

1. Send synchronization messages to the system analyzer (using UIA's LogSync module) to correlate messages coming from different cores and display them in the right order. LogSync message is sent at the interval configured at DAT client initialization. The period can be overwritten at the runtime through DAT API.
2. execute the producer background actions :
    - Buffer closure procedure where cache coherence operations on log buffer are performed,
    - Shipment of log buffers to connected consumers, debug stream and/or memory logging.

4.6 Flush buffers during exceptions

When a software exception occurs, the application can call the DAT API to flush the active buffer and buffers in the pending Queue. These buffers will be delivered to the configured trace destination.

4.7 Memory logging

4.7.1 Save logs in memory

Memory logging can be enabled for producers during producer creation if a MEMLOG channel is provided. MEMLOG channel is created through MEMLOG module APIs. It creates a heap by allocating **memory buffers** from the memory block given by an application. It utilizes CPPI DMA based msgcom channel to copy the producer buffer to these pre-allocated memory buffers without ARM intervention. The memory information of the memory buffers are saved in named database and can be retrieved through MEMLOG controller.

4.7.2 Retrieve logs from memory

On ARM side, logs can be dumped from pre-allocated memory to a file at any time through a **MEMLOG controller**. A MEMLOG controller has memory logging information for a DAT producer in named database. To avoid log corruption, APIs are provided to stop/start logging into the memory on producer side.

5 Trace object & Verbosity

DAT also provides a framework to maintain logging levels for different components in the system.

Trace Object is an entity used to query/modify trace verbosities. Trace Object instance is a copy of a trace object and its verbosity settings on a DAT client. DAT server maintains the database of trace objects and trace levels for its components. The main feature of Trace object is as follows:

- Trace object with default trace levels are **created/deleted** through DAT client.
- **Query/modification** of a trace object's trace levels can be triggered from any DAT client. The results of a "query" will be returned from DAT server. Modification request is sent to DAT server first, DAT server will propagate the change to all the clients that has an instance of this trace object.

The following sections illustrate this framework with the LTE system as an example.

5.1 Trace classes and components

The LTE software system has different layers such as the PHY, L2 and L3. From the logging system perspective, PHY, L2 and L3 are considered Trace classes. The different components under the class are Trace components, each with configurable logging.

Each class owner maintains an independent list of enums called Trace component IDs to characterize the different components, and these will be published.

Example:
Trace Class: PHY; Components: Equalizer, PRACH, PUCCH etc.

```
typedef enum
{
        TRACE_ COMPONENT_PHY_EQUALIZER  = TRACE_COMPONENT_PHY_START,
        TRACE_ COMPONENT_PHY_PRACH,
        TRACE_ COMPONENT_PHY_PUCCH,
        ...
        TRACE_ COMPONENT_PHY_PDSCH  = TRACE_COMPONENT_PHY_END
} TRACE_COMPONENT_TYPE;
```

5.2 Trace verbosity levels

The verbosity levels are in the form of two 16-bit masks (focused and non-focused) for each trace component; one 16-bit common mask for all components which is ORed with individual component's mask and one 16-bit mask for the class (to control SYS/BIOS logging etc).

5.2.1 Component mask

If there are multiple instances of a component, then one or more of them can be in 'focus' while the others are not. The implementation of how to set the focus level of a trace component is left to the class owner.

DAT internal representation of a component mask:

DAT stores each component mask as 32 bits with the higher 16 bits of the mask corresponding to the 'focused' level of component verbosity, while the lower 16 bits for the 'normal' or non-focused level.

31 ..... 16 15 0

<Focused mask>	**<Normal mask>**

16-bit masks with the levels of verbosity are published. The verbosity levels with corresponding bit masks are listed in the table below.

Trace Level	Description	Bit Mask
DAT_COMP_LEVEL_EXCEPTION	Critical errors or exceptions	0x0001
DAT_COMP_LEVEL_ERROR	Errors that can be recovered from	0x0002
DAT_COMP_LEVEL_WARNING	Warning messages	0x0004
DAT_COMP_LEVEL_PE0	High-level debug messages or protocol events (PE); important information printed	0x0008

	occasionally	
DAT_COMP_LEVEL_PE1	Mid-level debug messages; printed less frequently (example: less than once per TTI)	0x0010
DAT_COMP_LEVEL_PE2	Low-level debug messages; printed frequently (example: many per TTI)	0x0020
DAT_COMP_LEVEL_BENCHMARK0	Benchmarks for the component	0x0040
DAT_COMP_LEVEL_BENCHMARK1	Benchmarks for the component	0x0080
DAT_COMP_LEVEL_USERDEF0	User-defined trace level. Can be uniquely defined for each component.	0x0100
DAT_COMP_LEVEL_USERDEF1	User-defined trace level. Can be uniquely defined for each component.	0x0200
DAT_COMP_LEVEL_USERDEF2	User-defined trace level. Can be uniquely defined for each component.	0x0400
DAT_COMP_LEVEL_USERDEF3	User-defined trace level. Can be uniquely defined for each component.	0x0800
DAT_COMP_LEVEL_USERDEF4	User-defined trace level. Can be uniquely defined for each component.	0x1000
DAT_COMP_LEVEL_USERDEF5	User-defined trace level. Can be uniquely defined for each component.	0x2000
DAT_COMP_LEVEL_USERDEF6	User-defined trace level. Can be uniquely defined for each component.	0x4000
DAT_COMP_LEVEL_USERDEF7	User-defined trace level. Can be uniquely defined for each component.	0x8000

Table 5:1 Component verbosity levels

The implementation of the trace API call (macros/inline/etc.) can be proprietary. The production is expected to use Log_iwriteUCn().

5.2.2 Common component mask

The common component mask has no concept of a focused versus non-focused mask. During filtering operations based on logging level, the common component mask is ORed with the individual component's focused or non-focused mask as applicable. The common component mask will allow high level configurability of all components in that class. If user wants to enable DAT_COMP_LEVEL_BENCHMARK0 for all components in the class, this can be achieved by the setting the corresponding bit in the common component mask, instead of enabling this for each component individually.

5.2.3 Class mask

The class mask has no concept of a focused mask. The class mask will be used for any trace setting that can only be done at core or class level and not at component level. An example for this would be SYS/BIOS logging controls.

The verbosity levels for the class mask are shown in the table below.

Trace Level	Description	Bit Mask
DAT_CLASS_LEVEL_BIOSLOAD	Enable logging of load information for the CPU, thread types (Hwi, Swi) and Task functions	0x0001
DAT_CLASS_LEVEL_BIOSTASK	Enable event logging used to display the Execution Graph for SYS/BIOS Task threads	0x0002
DAT_CLASS_LEVEL_BIOSMAIN	Enable benchmarking information from pre-instrumented objects and any events added to the target code.	0x0004
DAT_CLASS_LEVEL_HWI	Enable event logging used to display the Execution Graph for SYS/BIOS Hwi threads	0x0008
DAT_CLASS_LEVEL_SWI	Enable event logging used to display the Execution Graph for SYS/BIOS Swi threads	0x0010
DAT_CLASS_LEVEL_USERDEF0	User-defined trace level. Can be uniquely defined for each class.	0x0020
DAT_CLASS_LEVEL_USERDEF1	User-defined trace level. Can be uniquely defined for each class.	0x0040
DAT_CLASS_LEVEL_USERDEF2	User-defined trace level. Can be uniquely defined for each class.	0x0080
DAT_CLASS_LEVEL_USERDEF3	User-defined trace level. Can be uniquely defined for each class.	0x0100
DAT_CLASS_LEVEL_USERDEF4	User-defined trace level. Can be uniquely defined for each class.	0x0200
DAT_CLASS_LEVEL_USERDEF5	User-defined trace level. Can be uniquely defined for each class.	0x0400
DAT_CLASS_LEVEL_USERDEF6	User-defined trace level. Can be uniquely defined for each class.	0x0800
DAT_CLASS_LEVEL_USERDEF7	User-defined trace level. Can be uniquely defined for each class.	0x1000
DAT_CLASS_LEVEL_USERDEF8	User-defined trace level. Can be uniquely defined for each class.	0x2000
DAT_CLASS_LEVEL_USERDEF9	User-defined trace level. Can be uniquely defined for each class.	0x4000
DAT_CLASS_LEVEL_USERDEF10	User-defined trace level. Can be uniquely defined	0x8000

	for each class.	

Table 5:2 Class verbosity levels

DAT internal implementation of class and common component masks:

The lower order 16 bits corresponds to the common mask for components and the higher 16 bits is the class mask.

31 ..... 16 15.....0

<Mask controlling class level logging>	<Common mask controlling all components>

5.3 Trace Objects

Each class has a trace object associated with it which stores the verbosity levels of the class and all its components. A trace object is created through a DAT API (`Dat_createTraceObject`) as part of which the default verbosity levels for class and components are specified along with the set of component names. When there is a query for current verbosity levels, the component names along with verbosity levels are provided to the user.

A trace object is created on DAT server, its trace verbosity levels are saved in the database maintained by DAT server. In order to use the trace object, an instance of the trace object needs to be created on a DAT client. Each trace object is characterized by a unique name, and multiple instances of the trace objects can be created from different DSP cores and ARM. When creating a **trace object instance**, a copy of trace verbosity level is saved in memory allocated locally. It is recommended to use fast local memory (such as L2SRAM on DSP) to for performance benefits.

For example, if a class like PHY is executing on two cores, and a trace object for PHY is created and one instance of this trace object can be created on each core using its local L2SRAM to store the verbosity levels. All filtering operations for logging would be done off the local instance and will be more efficient.

5.4 Filtering

When application intends to log a message, it has to first check if the level of verbosity required for that message is enabled or not. This filtering capability is provided through the `Dat_filter` API, where application passes parameters such as the trace object body from a trace object instance, component ID, logging level and focused or non-focused flag. The filtering service checks the trace object instance to see if the particular logging level for the component is set in the trace object, either in the individual component's mask or the common component mask, and tells the application whether the event can be logged or not.

5.5 Verbosity modification and querying

Verbosity levels can be modified and queried from any DAT client with a trace object instance handle.

5.5.1 Trace verbosity Query

Query request is sent from DAT client to DAT server. DAT server will send the query results back to the questing DAT client with information from its database.

5.5.2 Trace verbosity Modification

Modification request is sent from DAT client to DAT server. DAT server will modify the trace verbosity in its database and propagate and change to all the registered trace object instances running on different DSP cores and ARM.