Multicore System Analyzer User's Guide

Literature Number: SPRUH43A
June 2011

Texas Instruments

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Communications & Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers & Peripherals	www.ti.com//computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy & Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics, & Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation & Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com	Wireless	www.ti.com/wireless
RF/IF & ZigBee® Solutions	www.ti.com/lprf	**TI E2E Community**	e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303 Dallas, Texas 75265

Preface

About This Guide

Multicore System Analyzer (MCSA) is a tool suite that provides visibility into the real-time performance and behavior of your software. It allows you to analyze the load, execution sequence, and timing of your single- or multi-core target applications. MCSA is made up of a number of components. Two key components of MCSA are:

- **DVT.** Various features of DVT provide the user interface for MCSA within Code Composer Studio (CCS).
- **UIA.** The Unified Instrumentation Architecture (UIA) target package defines APIs and transports that allow embedded software to log instrumentation data for use within CCS.

This document provides information about both the host-side and target-side components of MCSA.

Intended Audience

This document is intended for users of Multicore System Analyzer.

This document assumes you have knowledge of inter-process communication concepts and the capabilities of the processors available to your application. This document also assumes that you are familiar with Code Composer Studio, SYS/BIOS, and XDCtools.

See Section 3.1, *Different Types of Analysis for Different Users* for more about the categories of users for MCSA.

Notational Conventions

This document uses the following conventions:

❑ When the pound sign (#) is used in filenames or directory paths, you should replace the # sign with the version number of the release you are using. A # sign may represent one or more digits of a version number.

❑ Program listings, program examples, and interactive displays are shown in a `mono-spaced font`. Examples use **`bold`** for emphasis, and interactive displays use **`bold`** to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

❑ Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

Documentation Feedback

If you have comments about this document, please provide feedback at http://www.go-dsp.com/forms/techdoc/doc_feedback.htm?litnum=SPRUH43. This link is for reporting errors or providing comments about a technical document. Using this form for support or asking questions will delay getting a response to you.

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, DaVinci, the DaVinci logo, XDS, Code Composer, Code Composer Studio, Probe Point, Code Explorer, DSP/BIOS, SYS/BIOS, RTDX, Online DSP Lab, DaVinci, eXpressDSP, TMS320, TMS320C6000, TMS320C64x, TMS320DM644x, and TMS320C64x+.

MS-DOS, Windows, and Windows NT are trademarks of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

All other brand, product names, and service names are trademarks or registered trademarks of their respective companies or organizations.

June 9, 2011

Contents

Chapter 1

Overview of Multicore System Analyzer

This chapter provides an introduction to Multicore System Analyzer (MCSA) and its host-side and target-side components.

1.1 Introduction

Instrumenting software with print statements to provide visibility into the operation of the software at run-time has long been one of the keys to creating maintainable software that works. As devices become increasingly complex, the system-level visibility provided by software instrumentation is an increasingly important success factor, as it helps to diagnose problems both in the lab during development and in the field.

One of the key advantages of instrumented software is that, unlike debug sessions, the statements used are part of the code-base. This can help other developers figure out what is going on as the software executes. It can also highlight integration problems and error conditions that would be hard to detect otherwise.

As a result, many groups create their own logging APIs. Unfortunately, what often happens is that the logging APIs they create are closely tied to particular hardware and operating systems, use incompatible logging infrastructures, make assumptions about the acceptable amount of memory or CPU overhead, generate logs in a diverse range of formats, may not include timestamps, or may use different time-bases (ticks, cycles, wall-clock, etc.). All of these differences make it difficult to port code from one system to another, difficult to integrate software from different development groups, difficult or impossible to correlate events from different cores on the same device, and costly to create tooling to provide ease-of-use, analysis and visualization capabilities.

The Multicore System Analyzer (MCSA) tool suite provides a consistent and portable way to instrument software. It enables software to be re-used with a variety of silicon devices, software applications, and product contexts. It includes both host-side tooling and target-side code modules (the UIA software package). These work together to provide visibility into the real-time performance and behavior of software running on TI's embedded single-core and multi-core devices.

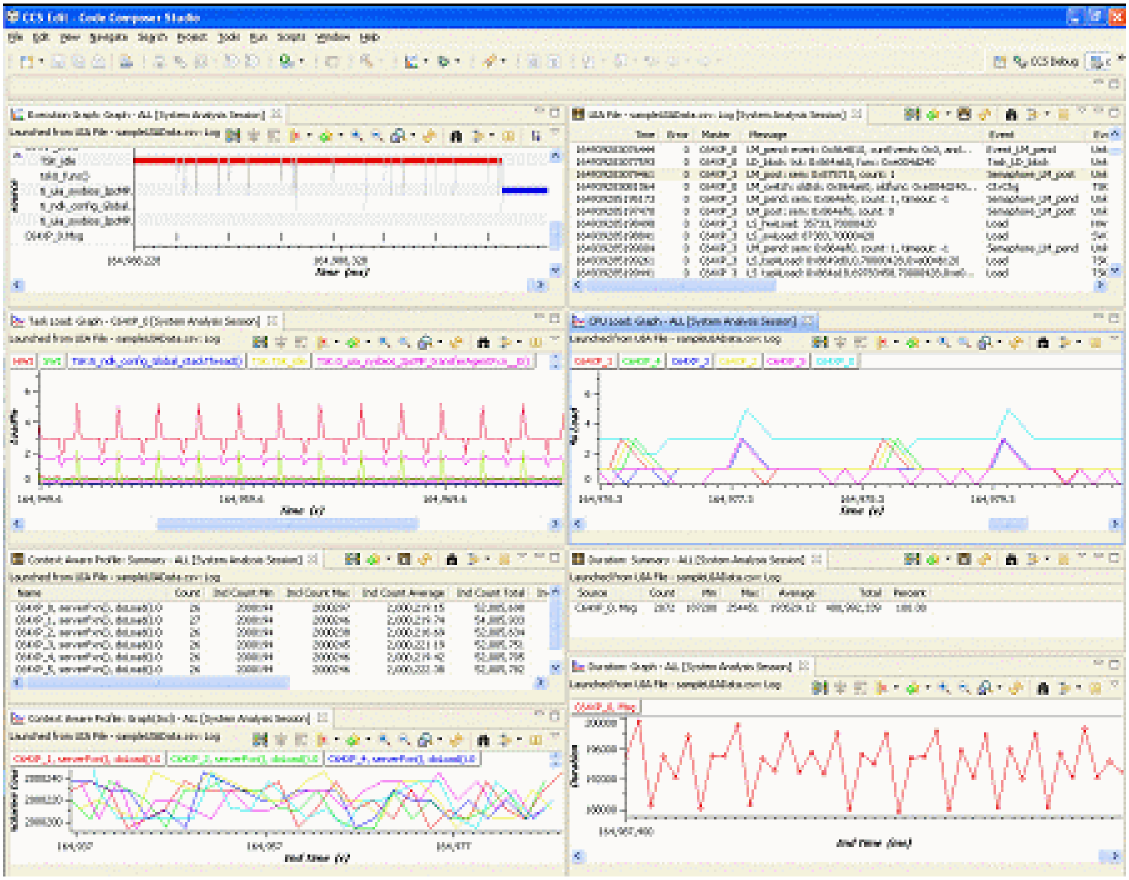
1.1.1 What Analysis and Visualization Capabilities are Provided?

The host-side MCSA tools use TI's Data Visualization Technology (DVT) to provide the following features for target applications that have been instrumented with the UIA target software package:

- ❏ **Advanced analysis features for data analysis and visualization.** Features include the ability to view the CPU and thread loads, the execution sequence, thread durations, and context profiling.
- ❏ **Multi-core event correlation.** Allows software instrumentation events from multiple cores on multi-core devices to be displayed on

the same timeline, allowing users to see the timing relationships between events that happened on different CPUs.

- **Run-time analysis.** For targets that support either the UIA Ethernet transport or real-time JTAG transport, events can be uploaded from the target to MCSA while the target is running without having to halt the target. This ensures that actual program behavior can be observed, without the disruption of program execution that occurs when one or more cores are halted.
- **Recording and playback of data.** You can record real-time event data and later reload the data to further analyze the activity. Both CSV and binary files are supported by MCSA.


CCS Edit - Code Composer Studio
CCS Debug
Task Load: Graph
CPU Load: Graph - ALL [System Analysis Session]
Context Aware Profile: Summary - ALL [System Analysis Session]
Duration: Summary - ALL [System Analysis Session]
Duration: Graph - ALL [System Analysis Session]
Context Aware Profile: Graph(Incl) - ALL [System Analysis Session]
Time (s)
End Time (ns)

1.1.2 What the UIA Target Software Package Provides

For the target, the Unified Instrumentation Architecture (UIA) target package, a component of MCSA, provides the following:

❑ **Software instrumentation APIs.** The xdc.runtime.Log module provides basic instrumentation APIs to log errors, warnings, events and generic instrumentation statements. A key advantage of these APIs is that they are designed for real-time instrumentation, with the burden of processing and decoding format strings handled by the host. Additional APIs are provided by the ti.uia.runtime package to support logging blocks of data and dynamic strings (the LogSnapshot module), context change events (the LogCtxChg module), and multicore event correlation information (the LogSync module).

❑ **Predefined software events and metadata.** The ti.uia.events package includes software event definitions that have metadata associated with them to enable MCSA to provide performance analysis, statistical analysis, graphing, and real-time debugging capabilities.

❑ **Event loggers.** A number of event logging modules are provided to allow instrumentation events to be captured and uploaded to the host over both JTAG and non-JTAG transports. Examples include LoggerCircBuf, which logs events to a circular buffer in memory, and LoggerSM, which logs events to shared memory and enables events to be decoded and streamed to a Linux console window.

❑ **Transports.** Both JTAG-based and non-JTAG transports can be used for communication between the target and the host. Non-JTAG transports include Ethernet, with UDP used to upload events to the host and TCP used for bidirectional communication between the target and the host.

❑ **SYS/BIOS event capture and transport.** For example, when UIA is enabled, SYS/BIOS uses UIA to transfer data about CPU Load, Task Load, and Task Execution to the host.

❑ **Multi-core support.** UIA supports routing events and messages across a central master core. It also supports logging synchronization information to enable correlation of events from multiple cores so that they can be viewed on a common timeline.

❑ **Scalable solutions.** UIA allows different solutions to be used for different devices.

❑ **Examples.** UIA includes working examples for the supported boards.

❑ **Source code.** UIA modules can be modified and rebuilt to facilitate porting and customization.

1.2 MCSA Terminology

You should be familiar with the following terms when using this manual.

- **Multicore System Analyzer (MCSA).** A suite of host-side tools that use data captured from software instrumentation, hardware instrumentation, and CPU trace to provide visibility into the real-time performance and behavior of target applications.
- **UIA.** Unified Instrumentation Architecture. A target-side package that provides instrumentation services.
- **DVT**. Data Visualization Technology. Provides a common platform to display real-time SYS/BIOS and trace data as lists and graphically. Used in the MCSA features. Also used in such CCS features as STM Logic and Statistics Analyzer and Trace Analyzer.
- **CCS**. Code Composer Studio. The integrated development environment (IDE) for TI's DSPs, microcontrollers, and application processors.
- **Analysis Feature**. A MCSA tool provided by DVT for use in the analysis of instrumentation data. A feature typically consists of several related views. For example, "CPU Load" is an Analysis Feature that includes summary, detail, and graph views.
- **Core.** An embedded processor. Also called a CPU.
- **Host**. The processor that communicates with the target(s) to collect instrumentation data. For example, a PC running Code Composer Studio.
- **Target**. A processor running target code. Generally this is an embedded processor such as a DSP or microcontroller.
- **UIA Packets**. Generic term for either Events or Messages. A UIA packet can hold multiple events or a single message.
- **Events**. Instrumentation data sent from the target to the host For example, Log records.
- **Messages**. Actions that are sent between the host and target. For example, commands, acknowledgements, and results.
- **Service**. A component that supplies some type of host/target interaction. There can be multiple services in a system. An example is the Rta Service that provides XDC Log information.
- **IPC**. Inter-Processor Communication. A software product containing modules designed to allow communication between processors in a multi-processor environment.

- **JTAG**. Joint Test Action Group. IEEE specification (IEEE 1149.1) for a serial interface used for debugging integrated circuits.
- **MADU**. Minimum Addressable Data Unit. Also called MAU. The minimum sized data that can be accessed by a particular CPU. Different architectures have different size MADUs. For the C6000 architecture, the MADU for both code and data is an 8-bit byte.
- **NDK**. Network Developer's Kit. Contains libraries that support the development of networking applications.
- **SYS/BIOS**. A real-time operating system for a number of TI's DSPs, microcontrollers, and application processors. Previously called DSP/BIOS.
- **SysLink**. Run-time software and an associated porting kit to simplify the development of embedded applications in which either General-Purpose microprocessors (GPPs) or DSPs communicate with each other.
- **RTSC.** Real-Time Software Components. A standard for packaging and configuring software components. XDCtools is an implementation of the RTSC standard.
- **UART**. Universal Asynchronous Receiver/Transmitter. A UART chip controls the interface to serial devices.
- **XDCtools**. A product that contains tools needed to create, test, deploy, install, and use RTSC components. RTSC standardizes the delivery of target content.
- **xdc.runtime.** A package of low-level target-software modules included with XDCtools that provides "core" services appropriate for embedded C/C++ applications, including real-time diagnostics, concurrency support, memory management, and system services.

1.3 Using MCSA with Your Application Software

MCSA provides flexible ways to instrument your application and to configure how logged events are uploaded to the host.

1.3.1 Instrumenting Your Application Using UIA

There are a number of different ways to take advantage of the real-time visibility capabilities provided by MCSA and the UIA target software:

- ❏ SYS/BIOS modules provide built-in software instrumentation that can be enabled to provide visibility into CPU Load, Task Load, and Task Execution "out of the box". (See Section 3.2 and Section 3.3).
- ❏ The UIA and xdc.runtime.Log APIs can be used in your C or C++ code directly to log software events to instrument your application code. You don't have to write RTSC modules; just #include the appropriate header files in your software and call the provided APIs. Examples are provided in Section 5.4 as well as in the help files that ship with the UIA target content.
- ❏ Macros can be used to wrap the UIA and XDC event logging APIs so that they can be called using the same API signature as other event logging APIs your software may already be using for software instrumentation. More information is provided on the wiki page at http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer.

1.3.2 Capturing and Uploading Events Using UIA

UIA allows you to configure the infrastructure used to capture and upload software instrumentation events without having to change your application software C code. The LoggingSetup module in the ti.uia.sysbios package provides the following eventUploadMode configuration options, which can be configured by adding a couple of script statements or settings in XGCONF:

Table 1–1 Event Upload Modes

Mode	Description
Simulator	Events are uploaded from the simulator at the time the event is logged. Uses the same infrastructure as Probe Point event upload.
Probe Point	Events are uploaded over JTAG at the time the event is logged. The target is briefly halted while the event is uploaded.
JTAG Stop-Mode	Events are uploaded over JTAG when the target halts. This is the default.
JTAG Run-Mode	Events are streamed from the target to the host via JTAG while the target is running. (For UIA 1.0, this is only available on C6X targets).
Non-JTAG Transport	Events are uploaded over a non-JTAG transport such as Ethernet.

For details about the benefits and constraints for each of these modes, see *Configuring the Event Upload Mode*, page 5-11.

Other Communication Options

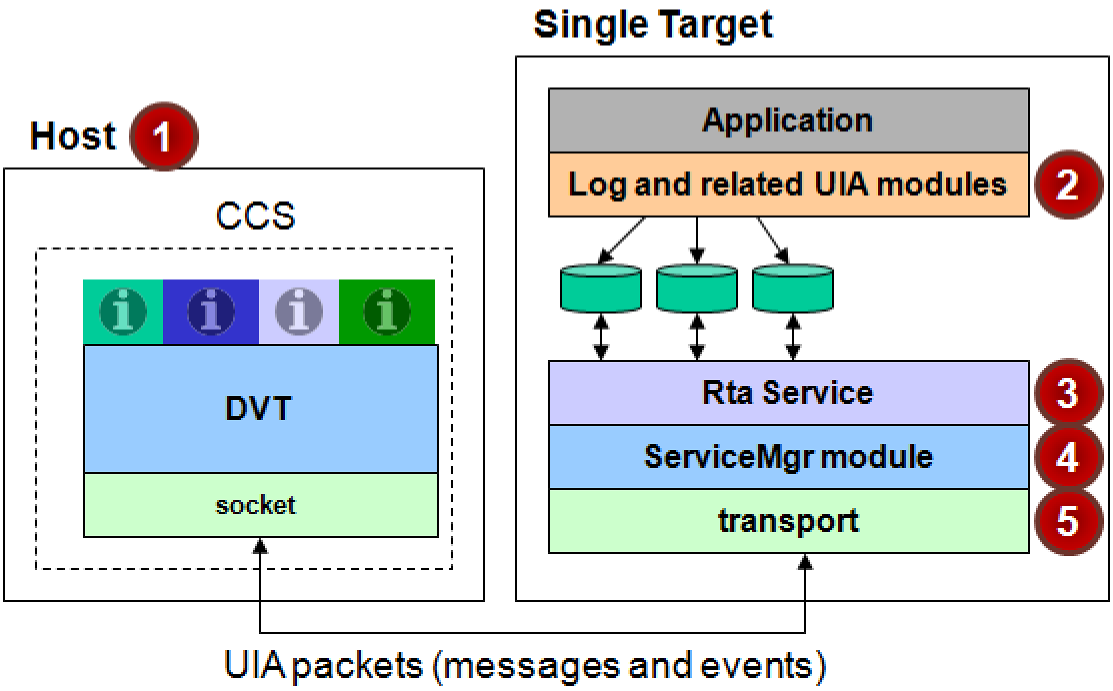
- ❑ **Specialized Logger.** Logger modules can implement a host-to-target connection in various ways. For example, the LoggerSM module provided with UIA uses shared memory reads and writes to directly communicate with a Linux application.
- ❑ **UIA ServiceMgr Framework.** UIA provides a full-featured pluggable framework. It supports both default SYS/BIOS instrumentation and extensive custom instrumentation. Communication via Ethernet, files over JTAG, and other methods can be plugged into this framework. The advantage of this technique is its power and flexibility. The disadvantage is that the code and data footprint on the target may be too large for memory-constrained targets. More information is provided in Section 1.4.

Note: UIA does not support RTDX (Real-Time Data eXchange). Please use JTAG Run-Mode instead.

1.4 How Does MCSA Communicate over Non-JTAG Transports?

UIA manages communication between the target(s) and the host by providing a ServiceMgr module that is responsible for sending and receiving packets between the services on the target and the host.

The following is simplified diagram of the components and connections involved in single-core target applications. The numbers correspond to the items in the numbered list after the diagram.



1) **Host.** The host is typically a PC running Code Composer Studio. Within CCS, the MCSA features provided by DVT (the "i" icons in the diagram) display and make sense of the UIA packets received via the socket connection.

2) **Target application.** The target runs an application that uses SYS/BIOS and/or XDCtools for configuration and APIs. Internally, the SYS/BIOS modules make API calls to log events related to threading. You can also add additional configuration and calls to make use of the logging, event-handling, and diagnostics support in UIA, SYS/BIOS, and XDCtools.

3) **Rta service.** UIA’s Rta module on the target collects events from the log written to by both the pre-instrumented SYS/BIOS threads and any custom instrumentation you have added. By default, it collects

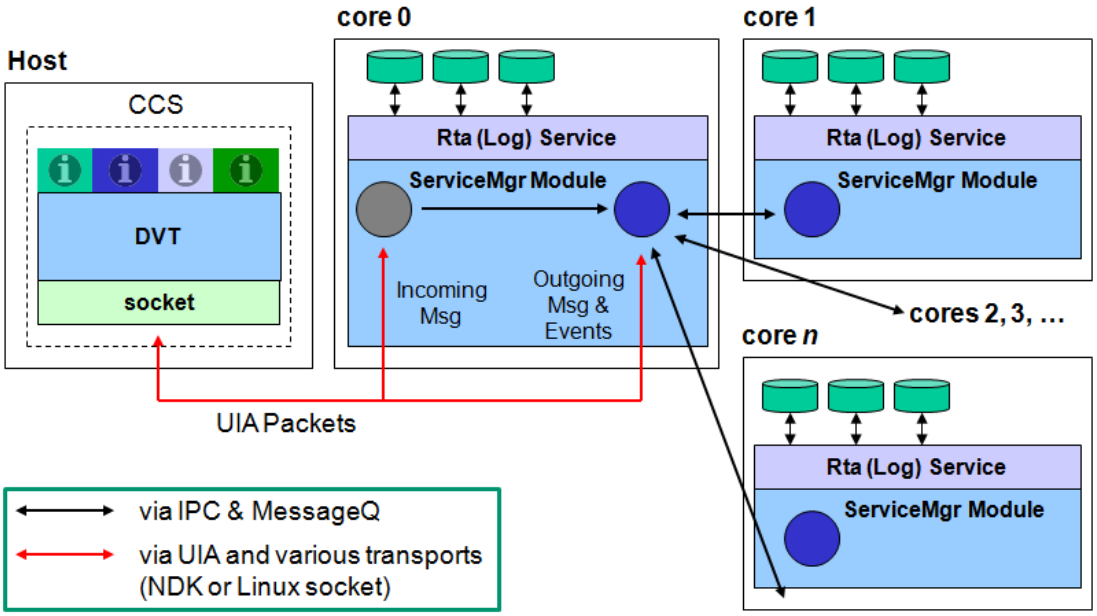
events every 100 milliseconds. Rta then sends the events on to the UIA ServiceMgr module.

4) **ServiceMgr module.** This module moves data off the target primarily in the background. You configure the ServiceMgr module to specify the following:

   - Whether you have a single-core or multi-core application.
   - If you have a multi-core application, which core is designated the master core, which communicates directly with the host.
   - The type of physical connection used for data transport between the master core and the host. Options are Ethernet, file (over JTAG), and user-defined (for custom connections).

5) **Transport.** By default, TCP is used to transport messages and UDP is used to transport events over an Ethernet connection. The application is responsible for setting up the Ethernet connection, for example by using the NDK on an EVM6472.

If there are multiple cores, the simplified diagram of the connections looks similar to the following:



In the multi-core case, the ServiceMgr module on each core is configured to identify the master core. UIA packets from other cores are sent to the master core by the ServiceMgr module via the MessageQ module, which is part of both IPC and SYSLink.

The master core sends the UIA packets on to the host via the Ethernet transport in the case of a master core that runs a SYS/BIOS application and via standard Linux socket calls in the case of an ARM master core.

1.4.1 Communication for EVM6472 Single-Core

If the target application runs on a single-core EVM6472, the ServiceMgr module on the target uses NDK as its transport. The NDK communicates with the host via sockets. The NDK transport functions are in the ti.uia.sysbios.TransportNdk module provided with UIA. See the <*uia_install*>\packages\ti\uia\sysbios directory.

1.4.2 Communication for EVM6472 Multi-Core

If the target application is running on multiple cores on an EVM6472, all non-master cores communicate to the “master” core via IPC’s MessageQ module.

The ServiceMgr module on the master core communicates with the host by using NDK as its transport. The NDK communicates with the host via sockets. The NDK transport functions, which are only used by the master core, are in the ti.uia.sysbios.TransportNdk module provided with UIA. See the <*uia_install*>\packages\ti\uia\sysbios directory.

1.4.3 Communication for EVMTI816x

If the target application is running on the ARM, DSP, and M3 cores of an EVMTI816x, the ServiceMgr module is used on all cores. The ARM core is configured to be the master core. The DSP and M3 cores communicate with the ARM core via SysLink’s MessageQ module. The ARM core communicates with the host via standard Linux socket calls. That is, the ARM core acts as a router for the UIA packets.

1.4.4 Communication for TCI6616

In the TCI6616 simulator the ti.uia.sysbios.TransportNyquistSim module provided with UIA uses WinPcap to send UIA packets to the host. See the release notes for the simulator for details. See the <*uia_install*>\packages\ti\uia\sysbios directory.

When the hardware is available, the ServiceMgr module on the master core will communicate with the host by using NDK as its transport.

1.5 About this User Guide

The remaining chapters in this manual cover the following topics:

- ❑ Chapter 2, "Installing Multicore System Analyzer", describes how to install the MCSA components.
- ❑ Chapter 3, "Tasks and Roadmaps for MCSA“, explains how to begin using MCSA.
- ❑ Chapter 4, “Using MCSA in Code Composer Studio“, describes the analysis features provided in Code Composer Studio for examining instrumentation data.
- ❑ Chapter 5, “UIA Configuration and Coding on the Target“, describes how to configure and code target applications using UIA modules.
- ❑ Chapter 6, “Advanced Topics for MCSA“, provides additional information about using MCSA components.

Note: Please see the release notes in the installation before starting to use MCSA. The release notes contain important information about feature support, issues, and compatibility information.

1.6 Learning More about Multicore System Analyzer

To learn more about MCSA and the software products used with it, refer to the following documentation:

- ❑ **UIA online reference help** (also called "CDOC"). Open with CCSv5 online help or run <*uia_install*>/docs/cdoc/index.html. Use this help system to get reference information about static configuration of UIA modules and C functions provided by UIA.
- ❑ **Tutorials.** http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer_Tutorials
- ❑ **TI Embedded Processors Wiki.** http://processors.wiki.ti.com
  - ■ **MCSA.** http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer
  - ■ **Code Composer Studio.** http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v5
  - ■ **SYS/BIOS.** http://processors.wiki.ti.com/index.php/Category:SYSBIOS
  - ■ **NDK.** http://processors.wiki.ti.com/index.php/Category:NDK
  - ■ **SysLink.** http://processors.wiki.ti.com/index.php/Category:SysLink

- ❑ **RTSC-Pedia Wiki.** http://rtsc.eclipse.org/docs-tip for XDCtools documentation.
- ❑ **TI E2E Community.** http://e2e.ti.com/
  - ■ For CCS and DVT information, see the Code Composer forum at http://e2e.ti.com/support/development_tools/code_composer_studio/f/81.aspx
  - ■ For SYS/BIOS, XDCtools, IPC, NDK, and SysLink information, see the SYS/BIOS forum at http://e2e.ti.com/support/embedded/f/355.aspx
  - ■ Also see the forums for your specific processor(s).
- ❑ **SYS/BIOS 6.x Product Folder.** http://focus.ti.com/docs/toolsw/folders/print/dspbios6.html
- ❑ **Embedded Software Download Page.** http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/index.html for downloading SYS/BIOS, XDCtools, IPC, and NDK versions.

Chapter 2

Installing Multicore System Analyzer

This chapter covers how to install the MCSA components.

Topic **Page**

2.1 MCSA Installation Overview

MCSA support is available for the targets listed in the release notes and at http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer. Specific example templates are provided for multi-core targets such as the evm6472 and the evmTI816x. In addition, pre-built libraries are provided for a number of single-core targets.

MCSA makes use of the following other software components and tools, which must be installed in order to use MCSA. See the release notes for the specific versions required for the target you are using.

- ❏ Code Composer Studio (CCStudio) 5.0
- ❏ SYS/BIOS 6.30 or 6.31 or higher (installed as part of CCStudio)
- ❏ XDCtools 3.20 or higher (installed as part of CCStudio)
- ❏ IPC 1.21 or 1.22 or higher (version required depends on target)
- ❏ Code Generation Tools (version required depends on target)
- ❏ NDK required for evm6472
- ❏ PDK and simulator required for simTCI6616
- ❏ SysLink required for evmTI816x

2.2 Installing MCSA as Part of a Larger Product

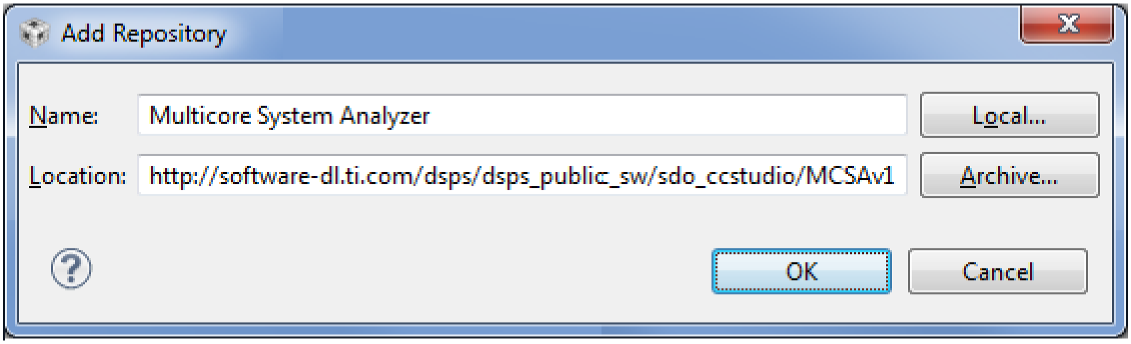
MCSA and the components it requires are automatically installed as part of the MCSDK installation. You do not need to perform additional installation steps in order to make it available.

MCSA and the components it requires will also be automatically installed as part of the Code Composer Studio v5 installation in the near future. MCSA will also be available through the CCS Update Installer.

2.3 Installing MCSA as a Software Update

To install or update MCSA from within CCSv5, follow these steps:

1) Choose **Help > Install New Software** from the CCS menus.
2) Click **Add** to the right of the Work with field.
3) In the Add Repository dialog, type "Multicore System Analyzer" as the **Name**.
4) In the **Location** field, type the following URL: `http://software-dl.ti.com/dsps/dsps_public_sw/sdo_ccstudio/MCSAv1` and then click **OK**.



Name: Multicore System Analyzer

Location: http://software-dl.ti.com/dsps/dsps_public_sw/sdo_ccstudio/MCSAv1

5) Check the box next to DVT, and click **Next**. (MSCA is installed as part of the Data Visualization Technology component of CCS.) Continue clicking **Next** as needed and accept the license agreement as prompted.
6) Click **Finish** to install or update the DVT software component. When the installation is complete, restart CCS as prompted.

2.4 Installing and Using UIA Outside CCS

You can also install the UIA target-side modules on a Linux machine for use outside the CCS environment. On a Linux machine, you should unzip the UIA target package in the same root directory where XDCtools and SYS/BIOS are installed.

If you want to build applications with UIA modules outside of CCS, add the UIA package path to your XDCPATH definition. The UIA package path is the /packages subfolder of the UIA target-side installation. For example, the package path may be the C:\Program Files\Texas Instruments\uia_1_#_#_#\packages folder.

Chapter 3

Tasks and Roadmaps for MCSA

This chapter explains how to begin using MCSA. It provides roadmaps for common tasks related to using MCSA.

3.1 Different Types of Analysis for Different Users

A variety of users make use of MCSA, but different users perform different types of analysis. To find tasks that apply to your needs, choose the use case that matches your needs best from the following list:

1) **Analyst for a deployed system.** You have an existing system for which you need a performance analysis. You do not need to know about the actual target code, and are interested in using the GUI features of MCSA to find answers about CPU utilization. You will want to use the CPU Load and possibly the Task Load analysis features.
2) **Linux developer.** You have a multi-core application with Linux on the master core and SYS/BIOS applications on other cores. You want data about how the SYS/BIOS applications are running, but do not want to modify these applications yourself. You should use the CPU Load, Task Load, and Execution Graph analysis features.
3) **SYS/BIOS application developer (simple case).** You want to analyze default information provided by SYS/BIOS, but do not want to add custom instrumentation code. You may be adding support for MCSA to a deployed application. You should use the CPU Load, Task Load, and Execution Graph analysis features.
4) **SYS/BIOS application developer (custom instrumentation).** You want to get additional information about threading and the time required to perform certain threads. In addition to the CPU Load, Task Load, and Execution Graph analysis features, you should use the Duration and Context Aware Profile features.
5) **SYS/BIOS application developer (custom communication).** You want to use MCSA on a multi-core platform with a setup that varies from the defaults. You may want to modify the transport or modify the behavior of the ServiceMgr module.

The following table shows tasks that apply to users in the previous list.

Table 3-1. Task Roadmaps for Various Users

User Type	Load Analysis	Execution Sequence Analysis	Benchmarking and Count Analysis	SYS/BIOS & UIA Configuration	SYS/BIOS & UIA API Coding	Multi-core IPC, NDK, or SysLink setup
1	Yes	No	No	No *	No	No
2	Yes	Yes	No	No *	No	Maybe

User Type	Load Analysis	Execution Sequence Analysis	Benchmarking and Count Analysis	SYS/BIOS & UIA Configuration	SYS/BIOS & UIA API Coding	Multi-core IPC, NDK, or SysLink setup
3	Yes	Yes	No	Yes	No	No
4	Yes	Yes	Yes	Yes	Yes	Maybe
5	Yes	Yes	Yes	Yes	Yes	Yes

***** A few SYS/BIOS configuration settings need to be modified and applications need to be rebuilt in order to use MCSA. Users who are not familiar with SYS/BIOS, should ask a SYS/BIOS application developer to make the configuration changes described in Section 5.1.

To learn about the tasks that apply to your needs, see the following sections:

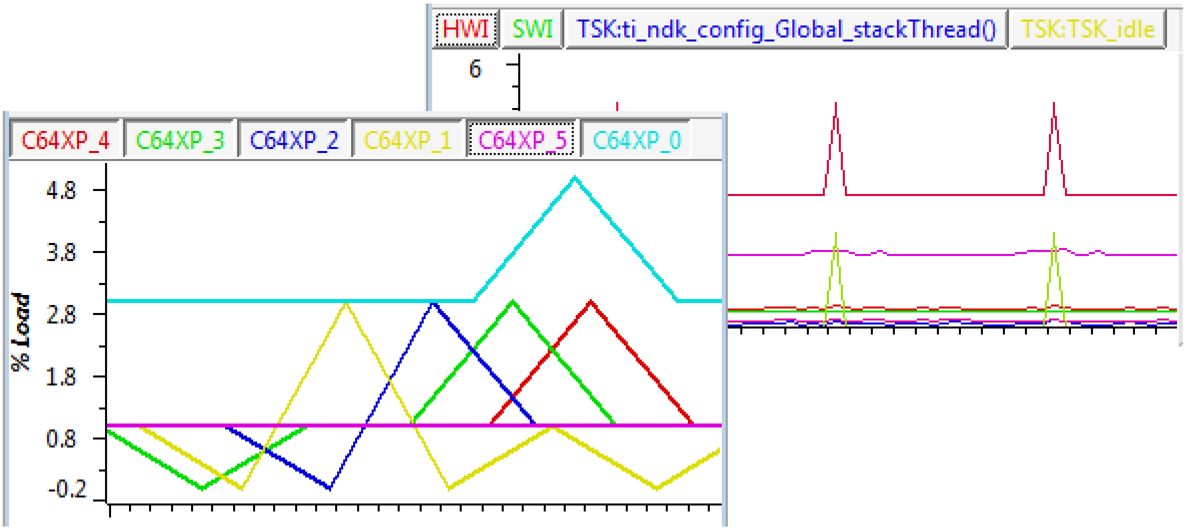
- **Load Analysis.** This includes using the CPU Load and Task Load analysis features. See Section 3.2 for a roadmap.
- **Execution Sequence Analysis.** This includes using the Execution Graph analysis feature. See Section 3.3 for a roadmap.
- **Benchmarking and Count Analysis.** This includes using the Context Aware Profile, Duration, and Count Analysis features. The target code needs to be modified in order to perform this type of analysis. See Section 3.5 for a roadmap.
- **SYS/BIOS and UIA Configuration.** This involves editing the *.cfg configuration file for the target application either with a text editor or with XGCONF in CCS. See Section 5.1 for the simple setup and Section 5.3 for custom configuration.
- **SYS/BIOS and UIA API Coding.** You can add C code to your target application to provide data to the Context Aware Profile and Duration analysis features. You can also add code for custom instrumentation. See Section 5.4 for details.
- **Multi-core IPC, NDK, or SysLink setup.** See Section 5.3.3, *Configuring ti.uia.runtime.ServiceMgr*, Section 5.3.6, *Configuring ti.uia.runtime.LogSync*, Section 5.3.7, *Configuring IPC*, and documentation for IPC, NDK, SysLink, etc.

3.2 Analyzing System Loading with MCSA

You can use MCSA to perform CPU and Task load analysis on SYS/BIOS applications.

- ❑ **CPU Load** is calculated on the target by SYS/BIOS and is based on the amount of time spent in the Idle thread. That is, the CPU Load percentage is the percent of time not spent running the Idle thread.
- ❑ **Task Load** is calculated on the target based on the amount of time spent in specific Task threads and in the Hwi and Swi thread-type contexts.

If configured to do so, the target application periodically logs load data on the target and transports it to the host. This data is collected and processed by MCSA, which can provide graph, summary, and detailed views of this data.



Performing Load Analysis

Follow these steps to perform load analysis for your application. Follow the links below to see detailed instructions for a particular step.

Step 1: Update your CCS installation to include MCSA and UIA if you have not already done so.

- ■ See Section 2.2, *Installing MCSA as Part of a Larger Product* or
- ■ See Section 2.3, *Installing MCSA as a Software Update*

Step 2: Configure your target application so that UIA logging is enabled. Causing your application to use UIA's LoggingSetup and Rta modules as described in the first link below automatically enables logging of events related to the CPU and Task load. You can skip the links to more detailed information that follow if you just want to use the default configuration.

- First, see Section 5.1, *Quickly Enabling UIA Instrumentation*.
- For more details, see Section 5.2.1, *Enabling and Disabling Load Logging*.
- For even more details, see Section 5.3, *Customizing the Configuration of UIA Modules*.

Note: If you are analyzing a deployed system or are integrating a system that includes SYS/BIOS applications, the step above may have already been performed by the application developer. If so, you can skip this step.

Step 3: If the application is not already loaded and running, build, load, and run your application.

Step 4: Start a CCS Debugging session with a target configuration to match your setup.

Step 5: Capture instrumentation data using MCSA. Note that when you start a session, you can choose to also send the data to a file for later analysis.

- See Section 4.2, *Starting a Live MCSA Session*.

Step 6: Analyze data using the CPU Load and/or Task Load Analyzer.

- See Section 4.7, *Using the CPU Load View with MCSA*.
- See Section 4.8, *Using the Task Load View with MCSA*.

See Also

❑ Section 4.7.3, *How CPU Load Works with MCSA*

❑ Section 4.8.3, *How Task Load Works with MCSA*

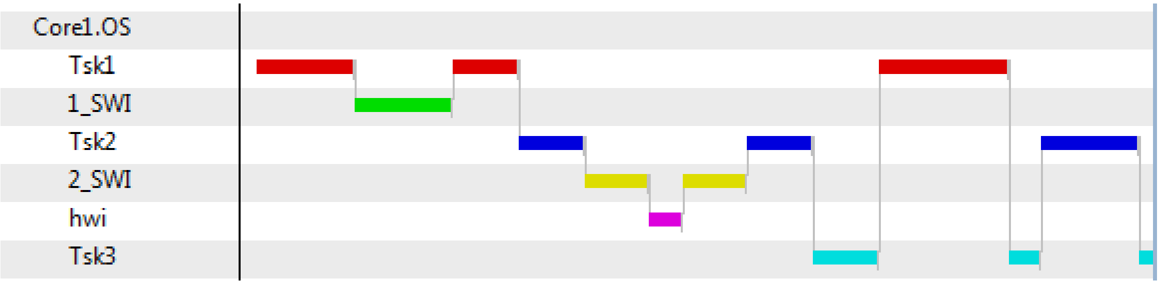
❑ Section 3.8, *Special Features of MCSA Data Views*

❑ To troubleshoot data loss: Section 3.6.3, *If MCSA Events are Being Dropped*

3.3 Analyzing the Execution Sequence with MCSA

You can use MCSA to perform execution sequence analysis on SYS/BIOS applications. The execution sequence and start/stop benchmarking events are shown in the Execution Graph.

If configured to do so, the target application periodically logs event data on the target and transports it to the host. This data is collected and processed by MCSA, which can provide a graph view of this data.



Performing Execution Sequence Analysis

Follow these steps to perform an execution sequence analysis for your application. Follow the links below to see detailed instructions for a particular step.

Step 1: Update your CCS installation to include MCSA and UIA if you have not already done so.

- See Section 2.2, *Installing MCSA as Part of a Larger Product* or
- See Section 2.3, *Installing MCSA as a Software Update*

Step 2: Configure your target application so that UIA logging is enabled. Causing your application to use UIA’s LoggingSetup and Rta modules as described in the first link for this step automatically enables logging of execution sequence events related to Task threads. You can enable execution sequence events for Swi and Hwi threads (which are off by default) as described at the second link. You can skip the links to more detailed information if you just want to use the default configuration.

- First, see Section 5.1, *Quickly Enabling UIA Instrumentation*.
- For more details, see Section 5.2.2, *Enabling and Disabling Event Logging*.
- For even more details, see Section 5.3, *Customizing the Configuration of UIA Modules*.

Note: If you are analyzing a deployed system or are integrating a system that includes SYS/BIOS applications, this step may have already been performed by the application developer. If so, you can skip this step.

Step 3: If the application is not already loaded and running, build, load, and run your application.

Step 4: Start a CCS Debugging session with a target configuration to match your setup.

Step 5: Capture instrumentation data using MCSA. Note that when you start a session, you can choose to also send the data to a file for later analysis.

- ■ See Section 4.2, *Starting a Live MCSA Session*.

Step 6: Analyze data using the Execution Graph.

- ■ See Section 4.9, *Using the Execution Graph with MCSA*.
- ■ In addition to the Execution Graph, you may find the Count columns in the CPU Load and Task Load summary views useful for analyzing the execution sequence. See Section 4.7.1, *Summary View for CPU Load* and Section 4.8.1, *Summary View for Task Load*.

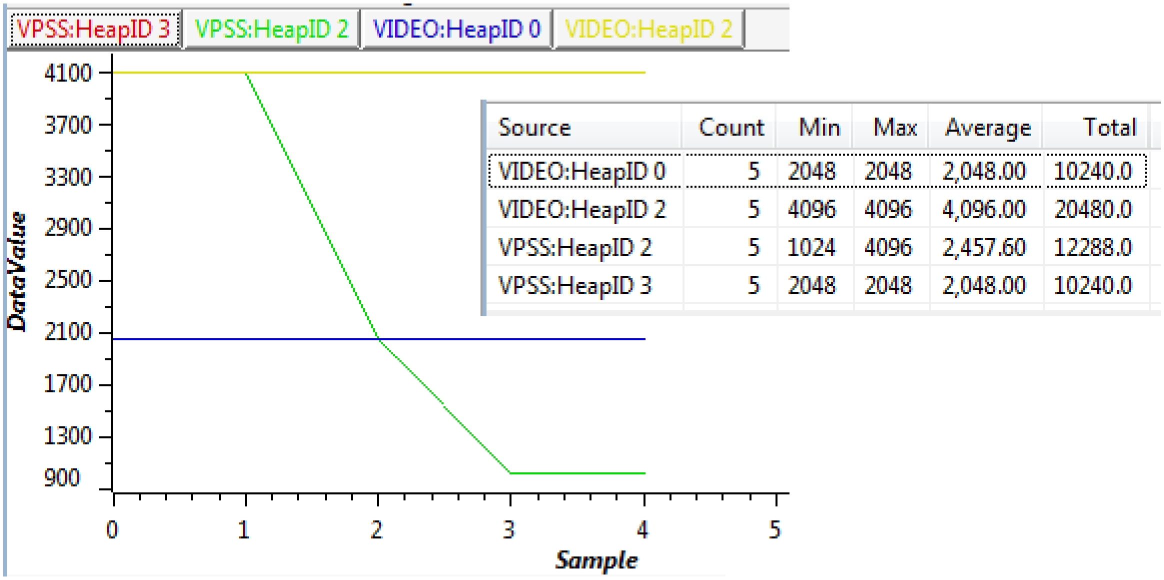
See Also

- ❏ Section 4.9.1, *How the Execution Graph Works with MCSA*
- ❏ Section 3.8, *Special Features of MCSA Data Views*
- ❏ To troubleshoot data loss: Section 3.6.3, *If MCSA Events are Being Dropped*

3.4 Performing Count Analysis with MCSA

You can use MCSA to perform count analysis on SYS/BIOS applications. For example, you might want to use Count Analysis to analyze how a data value from a peripheral changes over time. Or, you might want to find the maximum and minimum values reached by some variable or the number of times a variable is changed. The results are shown in the Count Analysis feature.

In order to use this feature, you will need to add code to your target to log data values for one or more sources. If you do this the target application transports the data to the host. This data is collected and processed by MCSA, which can provide graph, summary, and detailed views of this data.



Source	Count	Min	Max	Average	Total
VIDEO:HeapID 0	5	2048	2048	2,048.00	10240.0
VIDEO:HeapID 2	5	4096	4096	4,096.00	20480.0
VPSS:HeapID 2	5	1024	4096	2,457.60	12288.0
VPSS:HeapID 3	5	2048	2048	2,048.00	10240.0

Performing Count Analysis

Follow these steps to perform a count analysis for your application. Follow the links below to see detailed instructions for a particular step.

Step 1: Update your CCS installation to include MCSA and UIA if you have not already done so.

- See Section 2.2, *Installing MCSA as Part of a Larger Product* or
- See Section 2.3, *Installing MCSA as a Software Update*

Step 2: Configure your target application so that UIA logging is enabled. Causing your application to use UIA's LoggingSetup and Rta modules as described in the first link for this step automatically enables logging of execution sequence events related to Task threads. You can enable execution sequence events for Swi and Hwi threads (which are off by default) as described at the second link. You can skip the links to more detailed information if you just want to use the default configuration.

- First, see Section 5.1, *Quickly Enabling UIA Instrumentation*.
- For more details, see Section 5.2.2, *Enabling and Disabling Event Logging*.
- For even more details, see Section 5.3, *Customizing the Configuration of UIA Modules*.

Note: If you are analyzing a deployed system or are integrating a system that includes SYS/BIOS applications, this step may have already been performed by the application developer. If so, you can skip this step.

Step 3: Add code to your target application that logs the UIAEvt_intWithKey event.

- See Section 4.10.3, *How Count Analysis Works with MCSA*.

Step 4: Build, load, and run your application.

Step 5: Start a CCS Debugging session with a target configuration to match your setup.

Step 6: Capture instrumentation data using MCSA. Note that when you start a session, you can choose to also send the data to a file for later analysis.

- Section 4.2, *Starting a Live MCSA Session*.

Step 7: Analyze data using the Count Analysis feature.

- Section 4.10, *Using the Count Analysis Feature with MCSA*.
- If you want to perform statistical analysis on the primary and auxiliary data values, export records from the Count Analysis Detail view to a CSV file that can be opened with a spreadsheet. To do this, right-click on the view and choose **Data > Export All**.

See Also

❑ Section 3.8, *Special Features of MCSA Data Views*

❑ To troubleshoot data loss: Section 3.6.3, *If MCSA Events are Being Dropped*

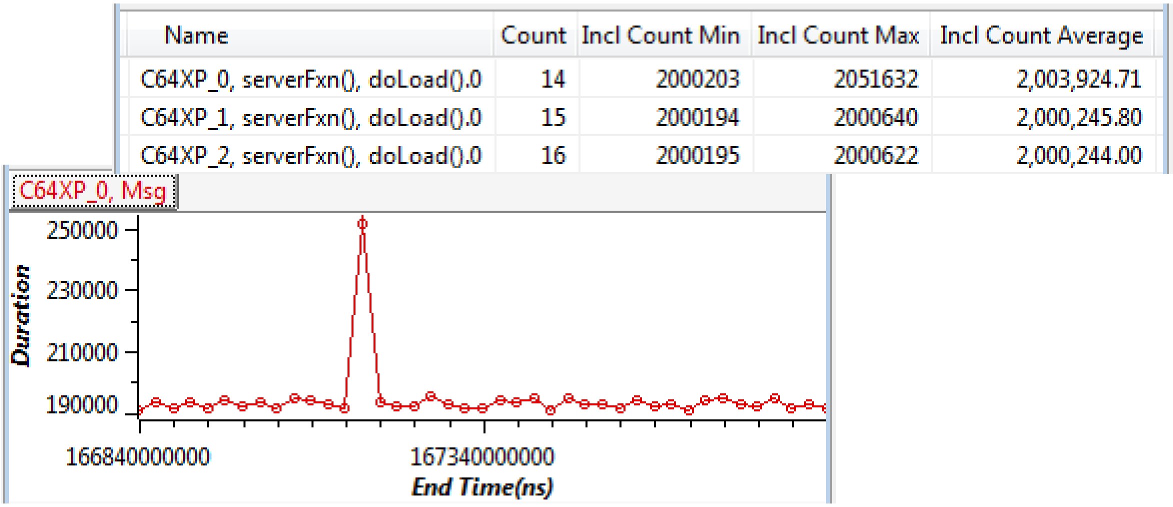
3.5 Benchmarking with MCSA

You can use MCSA to perform benchmarking analysis on SYS/BIOS applications. The results are shown in the Duration and Context Aware Profile features.

❏ **Duration Benchmarking.** Use this type of benchmarking if you want to know the absolute amount of time spent between two points in program execution.

❏ **Context Aware Profiling.** Use this type of benchmarking if you want to be able to measure time spent in a specific thread's context vs. time spent in threads that preempt or are yielded to by this thread.

In order to use these features, you will need to add code to your target to start and stop the benchmarking timer. If you do this the target application transports the data to the host. This data is collected and processed by MCSA, which can provide graph, summary, and detailed views of this data.



Name	Count	Incl Count Min	Incl Count Max	Incl Count Average
C64XP_0, serverFxn(), doLoad().0	14	2000203	2051632	2,003,924.71
C64XP_1, serverFxn(), doLoad().0	15	2000194	2000640	2,000,245.80
C64XP_2, serverFxn(), doLoad().0	16	2000195	2000622	2,000,244.00

Performing Benchmarking Analysis

Follow these steps to perform a benchmarking analysis for your application. Follow the links below to see detailed instructions for a particular step.

Step 1: Update your CCS installation to include MCSA and UIA if you have not already done so.

- See Section 2.2, *Installing MCSA as Part of a Larger Product* or
- See Section 2.3, *Installing MCSA as a Software Update*

Step 2: Configure your target application so that UIA logging is enabled. Causing your application to use UIA's LoggingSetup and Rta modules as described in the first link for this step automatically enables logging of execution sequence events related to Task threads. You can enable execution sequence events for Swi and Hwi threads (which are off by default) as described at the second link. You can skip the links to more detailed information if you just want to use the default configuration.

- First, see Section 5.1, *Quickly Enabling UIA Instrumentation*.
- For more details, see Section 5.2.2, *Enabling and Disabling Event Logging*.
- For even more details, see Section 5.3, *Customizing the Configuration of UIA Modules*.

Note: If you are analyzing a deployed system or are integrating a system that includes SYS/BIOS applications, this step may have already been performed by the application developer. If so, you can skip this step.

Step 3: Add benchmarking code to your target application.

- For duration benchmarking, see Section 4.11.3, *How Duration Analysis Works with MCSA*.
- For context aware profiling, see Section 4.12.3, *How Context Aware Profiling Works with MCSA*.

Step 4: Build, load, and run your application.

Step 5: Start a CCS Debugging session with a target configuration to match your setup.

Step 6: Capture instrumentation data using MCSA. Note that when you start a session, you can choose to also send the data to a file for later analysis.

- Section 4.2, *Starting a Live MCSA Session*.

Step 7: Analyze data using Duration and Context Aware Profile features.

- Section 4.11, *Using the Duration Feature with MCSA*.
- Section 4.12, *Using Context Aware Profile with MCSA*

See Also

- ❑ Section 3.8, *Special Features of MCSA Data Views*
- ❑ To troubleshoot data loss: Section 3.6.3, *If MCSA Events are Being Dropped*

3.6 Troubleshooting MCSA Connections

The following sections describe issues that might occur as you use MCSA and UIA.

3.6.1 If You Cannot Connect to the Target with MCSA

If you cannot connect to the target, check the following items:

- ❏ Verify that the UIA configuration specifies the correct transports.
- ❏ Verify that the configuration code for the target application includes the ti.uia.services.Rta module. You can use the **Tools > ROV** menu command in a CCS debugging session to confirm this.
- ❏ Verify that the correct transport functions were selected. You can do this by looking at the ti.uia.sysbios.Adaptor (or IpcMP) transport functions.

3.6.2 If No Events are Shown in MCSA Features

If you can connect to the target, but no events are shown in the Log view, check the following items:

- ❏ Confirm that the endpoints are configured properly in the MCSA Session Manager (**Tools > Multicore System Analyzer > Multicore System Analyzer Config**). Be sure that the .out and .xml filenames are correct.
- ❏ Confirm that the target application uses LoggerCircBuf.
- ❏ Confirm that events are being logged. You can check this by using the ROV tool to look at the ti.uia.runtime.LoggerCircBuf module. The "serial" field should be non-zero and increasing.
- ❏ Confirm that the UIA task is not being starved. You can check this by using the ROV tool to look at the ti.uia.runtime.ServiceMgr module. The "runCount" in the Proxy tab should be incrementing.
- ❏ Confirm that you’ve enabled logging by setting the common$.Diags mask accordingly in your configuration file. See Section 5.2.2.

3.6.3 If MCSA Events are Being Dropped

If you can connect to the target and events are shown in the Log view, events may still be dropped. The status bars in MCSA views tell how many records are shown and how many gaps occurred.

If events are being dropped, first confirm that events are being logged by the logger. You can check this by using the ROV tool to look at the ti.uia.runtime.LoggerCircBuf module. If the "numDropped" field is incrementing, then events are being dropped. If the "numDropped" field is not incrementing, then UIA packets are being dropped, and you should see Section 3.6.4.

To prevent events from being dropped, try one or more of the following:

- Increase the logger buffer size.
- Increase the frequency of Rta by lowering its period. The minimum is 100ms.
- Reduce the number of logged events.
- If this is a multi-core application, increase the number of event packets on the non-master processors. This allows UIA to move the records off in a faster manner. For example:

```
ServiceMgr.numEventPacketBufs = 4;
```

3.6.4 If MCSA Packets are Being Dropped

If UIA packets are being dropped, examine your configuration of IPC, NDK, or other communications software.

3.6.5 If Events Stop Being Show Near the Beginning

For a multi-core system, check the status message at the bottom of Log View. If the message says "Waiting UIA SyncPoint data", it is possible that the critical SyncPoint events were dropped in transport. Try using the **Disconnect** and **Connect** commands or the **Restart** command.

3.6.6 If MCSA Events Do Not Make Sense

If the events listed in the MCSA features do not make sense, confirm that the endpoints are configured properly in the MCSA Session Manager (**Tools > Multicore System Analyzer > Multicore System Analyzer Config**). Be sure that the .out and .xml filenames are correct.

3.6.7 If Data is Not Correlated for Multi-Core System

The following situations can cause correlation (out-of-sequence) errors:

❑ **The clock setting is not correct.** Each core/endpoint has clock settings (local and global) that are used to convert from local to global time. If any setting is incorrect, the global time conversion will be off and will affect the system-level correlation. Check the clock settings on the target side and UIA endpoint configuration.

❑ **SyncPoint is not logged properly.** For a multi-core platform, there must be a common global timer that each core can reference. If there is no global timer available or it is not configured properly, the converted global time in each core may not be correct. Also, since most global timers have a lower clock frequency, time precision may be lost with respect to the core's local timer. Check the SyncPoint events reported at the beginning of the log.

❑ **Transport delay.** Under certain conditions, some logs may be transported to the host computer with a huge delay. In this case, some old data may be received after newer data has been reported. Check the transport, especially when using UDP. If the transport is not reliable for a live data stream, specify a binary file to contain the live data. After the data has been captured, open the binary file to analyze the results.

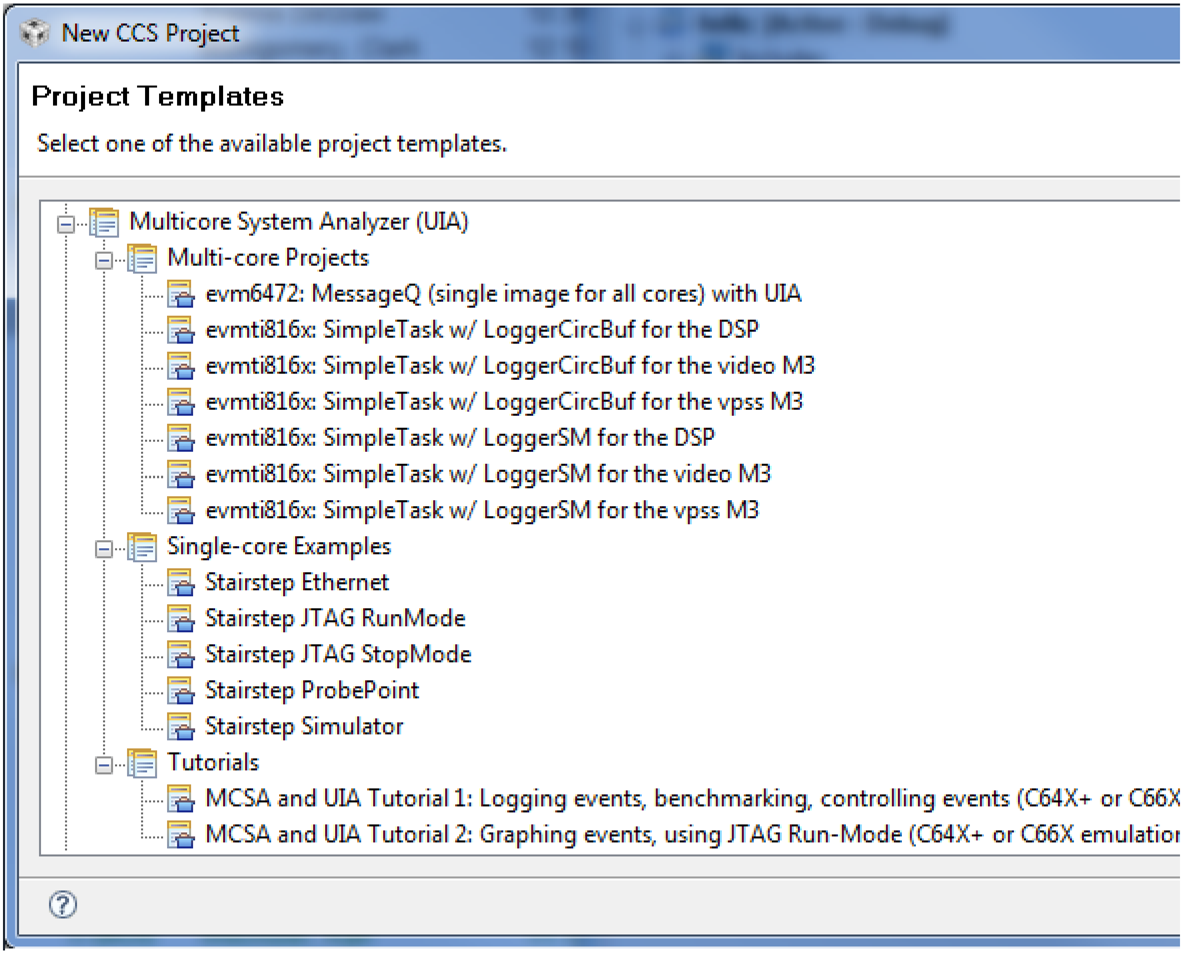
3.6.8 If the Time Value is Too Large

If the Time value shown in the logs is much larger than you expect, you should power-cycle the target board or perform a system reset before testing the application.

3.7 Creating Sample MCSA Projects

A number of project templates for use in CCS with MCSA and UIA are provided.

To use a project templates, begin creating a new CCS project by choosing **File > New > CCS Project** from the menus. When you reach the Project Templates page in the New Project wizard, expand the **Multicore System Analyzer (UIA)** item to see the list of available templates.



New CCS Project

Project Templates

Select one of the available project templates.

- Multicore System Analyzer (UIA)
  - Multi-core Projects
    - evm6472: MessageQ (single image for all cores) with UIA
    - evmti816x: SimpleTask w/ LoggerCircBuf for the DSP
    - evmti816x: SimpleTask w/ LoggerCircBuf for the video M3
    - evmti816x: SimpleTask w/ LoggerCircBuf for the vpss M3
    - evmti816x: SimpleTask w/ LoggerSM for the DSP
    - evmti816x: SimpleTask w/ LoggerSM for the video M3
    - evmti816x: SimpleTask w/ LoggerSM for the vpss M3
  - Single-core Examples
    - Stairstep Ethernet
    - Stairstep JTAG RunMode
    - Stairstep JTAG StopMode
    - Stairstep ProbePoint
    - Stairstep Simulator
  - Tutorials
    - MCSA and UIA Tutorial 1: Logging events, benchmarking, controlling events (C64X+ or C66X
    - MCSA and UIA Tutorial 2: Graphing events, using JTAG Run-Mode (C64X+ or C66X emulatior

When you select a project template, a description of the project is shown to the right. Finish creating the project and examine the *.c code files and *.cfg configuration file. All required products and repositories are pre-configured.

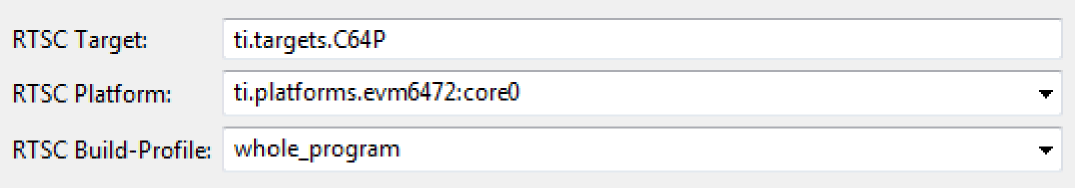
Multi-core project templates are available for the EVM6472 and the EVMTI816x. Single-core project templates that use the "stairstep" example from SYS/BIOS are available for a number of supported transports described on page 5–11. Additional tutorial examples are provided; these are described on the Texas Instruments Embedded Processors Wiki.

See the sections that follow for any specific notes about settings or changes you need to make to the project files before building, loading, and running it.

3.7.1 Notes for EVM6472 MessageQ Project Templates

On the Project Templates page of the New CCS project wizard, select the "evm6472: MessageQ" template. This example shows how to use IPC's MessageQ module with UIA. The same image must be loaded on all cores.

The RTSC Configuration Settings page of the wizard automatically has the correct RTSC Target, Platform, and Build-Profile set.



RTSC Target: ti.targets.C64P

RTSC Platform: ti.platforms.evm6472:core0

RTSC Build-Profile: whole_program

After creating the project, examine the message.c and message.cfg files.

In the message.c file, notice the two calls to Log_write2() in tsk0_func(), which runs only on CORE0. The calls to Log_write2() pass event types of UIABenchmark_start and UIABenchmark_stop. These are used to bracket the code that uses MessageQ to send and receive a message from a remote processor.

In the message.cfg file, notice that the LoggingSetup module is configured to use the UploadMode_NONJTAGTRANSPORT mode. This mode uses Ethernet as the default transport to move Log records to CCS via the UIA ServiceMgr framework. This example configures the ServiceMgr module to use a multi-core topology. All the cores route their data to the ServiceMgr module running on Linux. The configuration also contains a section that configures the NDK, which is used by the Ethernet transport.

UIA ships pre-built EVM6472 Ethernet drivers. The libraries are in the <*uia_install*>\packages\ti\uia\examples\evm6472\ndkdrivers directory. These libraries were copied out of the PDK_1_00_00_05 package. This was done to make building the examples easier.

Within the configuration file of EVM6472 example, the following line gets the pre-built Ethernet libraries and includes them in the build. If you have an updated PDK, simply remove this statement and add the libraries into the project (or follow the instructions with the PDK).

```
var ndkdrivers =
   xdc.loadPackage('ti.uia.examples.evm6472.ndkdrivers');
```

Note that the NDK currently supports only the COFF format.

You can use the following MCSA analysis features when running this example: CPU Load, Task Load, Execution Graph, Duration, and Context Aware Profile.

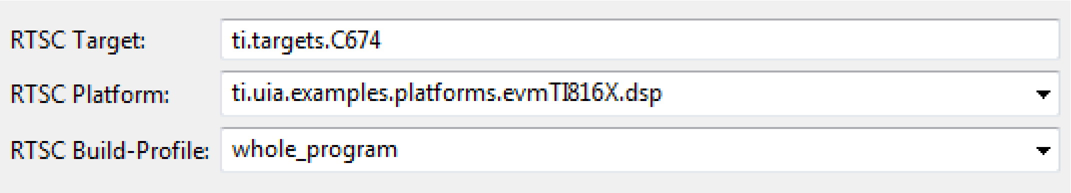
3.7.2 Notes for EVMTI816x SimpleTask Project Templates

On the Project Settings page of the New CCS project wizard, be sure to select the correct Device Variant (e.g. C674X or CortexM3).

On the Project Templates page of the New CCS project wizard, select one of the "evmti816x: SimpleTask" templates. These examples use LoggerCircBuf or LoggerSM (shared memory) to log benchmark events. Different projects are provided for the DSP, video M3, and vpss M3.

On the RTSC Configuration Settings page of the wizard, make sure to check the box for SysLink package in the Products and Repositories list. Use the **Add** button to add the repository if it is not shown.

The RTSC Configuration Settings page of the wizard automatically has the correct RTSC Target, Platform, and Build-Profile set. For example:



RTSC Target: ti.targets.C674

RTSC Platform: ti.uia.examples.platforms.evmTI816X.dsp

RTSC Build-Profile: whole_program

After creating the project, examine the simpleTask.c and *.cfg files.

In the simpleTask.c file, notice the two calls to Log_write1() in the taskLoad() function. The calls to Log_write1() pass event types of UIABenchmark_start and UIABenchmark_stop. These are used to bracket the code that reverses the bits in a buffer.

The configuration filename is dependent on the core and the logger implementation. For example, for the LoggerCircBuf version of the DSP application, the configuration file is called dspLoggerCircBuf.cfg. All versions of the configuration files for these examples include the simpleTask.cfg.xs configuration file. This shared file configures Clock, Semaphore, and Task objects. It also configures IPC and the shared memory region.

The non-shared configuration files cause the LoggingSetup module to use the UploadMode_NONJTAGTRANSPORT mode. This mode uses Ethernet as the default transport to move Log records to CCS via the UIA ServiceMgr framework. This example configures the ServiceMgr module to use a multi-core topology.

You can use the following MCSA analysis features with these examples: CPU Load, Task Load, Execution Graph, Duration, and Context Aware Profile.

See the <*uia_install*>\packages\ti\uia\examples\evmti816x directory for a readme.txt file with details on how to run the example. The source code and a Makefile to build the Linux application are also included in the <*uia_install*>packages\ti\uia\examples\evmti816x directory.

3.7.3 Notes for Single-Core Stairstep Project Templates

On the Project Templates page of the New CCS project wizard, expand the **Multicore System Analyzer > Single-core Examples** list and choose a "Stairstep" template. These examples use Hwi, Swi, and Task threads run to add to the CPU load of the system. This example periodically generates log events.

Each of the examples uses a different transport mode. These modes are configured by setting the LoggingSetup.eventUploadMode parameter.

The following list provides notes that apply to specific versions of this example:

- ❑ **Stairstep Ethernet.** This template is configured for use on the EVM6472 with NDK. Within the configuration file, the following line gets the pre-built Ethernet libraries and includes them in the build. If you have an updated PDK or are using a different device, simply remove this statement and add the libraries into the project (or follow the instructions with the PDK). See Section 3.7.1 for more about using the NDK with an application for the EVM6472.

```
var ndkdrivers =
  xdc.loadPackage('ti.uia.examples.evm6472.ndkdrivers');
```

- ❑ **Stairstep JTAG RunMode.** This mode is only supported on CPUs that support real-time JTAG access. This support is provided on the C64x+ and C66x CPUs. When the UploadMode_JTAGRUNMODE is used, the UIA ServiceMgr framework and NDK are not used.
- ❑ **All other Stairstep templates.** The JTAG StopMode, ProbePoint, and Simulator templates are not-platform specific. These templates do not use the UIA ServiceMgr framework or the NDK.

In the Stairstep example, the cpuLoadInit() function gets the CPU frequency and fills arrays with load values corresponding to 0, 25, 50, 75, and 95 percent CPU loads. The timerFunc() function is a Hwi thread that runs every 100ms to launch a Hwi, Swi, and Task thread. Each thread then performs a doLoad() function before relinquishing the CPU. After staying at each load setting for 5 seconds, timerFunc() calls the step() function to advance to the next set of Hwi, Swi, and Task load values. The cycle repeats after reaching the 95 percent load.

You can use the following MCSA analysis features when running these examples: CPU Load, Task Load, and Execution Graph.

3.7.4 Notes for MCSA Tutorial Project Templates

You can create projects using the MCSA and UIA tutorials. See http://processors.wiki.ti.com/index.php/Multicore_System_Analyzer_Tutorials

- ❑ **Tutorial 1:** This template is intended for use on a C64x+ or C66x simulator. This tutorial shows how to log errors, warnings, and informational events, benchmark code, and control which events are logged.
- ❑ **Tutorial 2:** This template is intended for use on a C64x+ or C66x emulator. This tutorial shows how to log data that can be graphed and analyzed for minimum, maximum, and average statistics.

3.8 Special Features of MCSA Data Views

Multicore System Analyzer provides three types of data views for working with collected data. Each type of data view has some power features you can use to navigate, analyze, and find points of interest. The sections that follow provide help on using the special features of these data views.

- ❑ **Table Views** are used to display data in a table. Table views are used for Summary and Detail views in MCSA.
- ❑ **Line Graphs** are used for x/y plotting, mainly for viewing changes of a variable against time. MCSA uses line graphs for all graphs except the Execution Graph.
- ❑ **DVT Graphs** depict state transitions and events against time. Groups of related states form a line with a common key value on the y-axis against time. Different types of data are assigned different colors. MCSA uses this graph type for the Execution Graph.

Special features provided for these view types are as follows:

- ❑ **Zoom (graphs only)** adjusts the scaling of the graph. See Section 3.8.1.
- ❑ **Measurement Markers (graphs only)** measure distances in a graph. See Section 3.8.2.
- ❑ **Bookmarks** highlight certain rows and provide ways to quickly jump to marked rows. See Section 3.8.3.
- ❑ **Groups and Synchronous Scrolling** causes several views to scroll so that data from the same time is shown. See Section 3.8.4.
- ❑ **Find** lets you search this view using a field value or expression. See Section 3.8.5.
- ❑ **Filter** lets you display only data that matches a pattern you specify using the Set Filter Expression dialog. See Section 3.8.6.
- ❑ **Export** sends selected data to a CSV file. See Section 3.8.7.
- ❑ **Scroll Lock** controls scrolling due to updates. See Section 3.8.8.
- ❑ ▲ **Sort** controls the record sequence in a table.
- ❑ **Auto Fit** adjusts table column widths to display complete values.
- ❑ **Tree Mode** toggles between flat and tree mode on y-axis labels. See Section 4.9.
- ❑ **Refresh** triggers a refresh of the view.

Also see page 4–6 and page 4–19 for additional descriptions of toolbar icons, including those shown only in the Log view.

3.8.1 Zoom (Graphs Only)

Zooming is only available in graph views. You can zoom in or out on both the x- and y-axis in line graphs. For DVT graphs (like the Execution Graph), you can only zoom on the x-axis.

You can zoom using any of these methods:

Using the Mouse

- Hold down the **Alt** key and drag the mouse to select an area on the graph to expand.
- Drag the mouse to the left or below the graph where the axis units are shown (without holding the Alt key) to select a range to expand.
- Click on the x-axis legend area below the graph and use your mouse scroll wheel to zoom in or out.

Using the Keyboard

- Press **Ctrl +** to zoom in.
- Press **Ctrl -** to zoom out.

Using the Toolbar

- The **Zoom In** toolbar icon increases the graph resolution to provide more detail. It uses the zoom direction and zoom factor set in the drop-down.
- The **Zoom Out** toolbar icon decreases the graph resolution to provide more detail. It uses the zoom direction and zoom factor set in the drop-down.
- The **Reset Zoom** toolbar icons resets the zoom level of the graph to the original zoom factor.
- The **Select Zoom Options** drop-down next to the Reset Zoom icon lets you select the zoom factor and directions of the zoom for a line graph. By default, zooming affects both the x- and y-axis and zooms by a factor of 2. You can choose options in this drop-down to apply zooming to only one axis or to zoom by factors of 4, 5, or 10.

Note: When you use the keyboard, scroll-wheel, or toolbar icons for zooming, the cursor position is used as the center for zooming. If there is no current cursor position, the center of the graph is used. To set a cursor position, click on the point of interest on the graph area. This places a red line or cross-hair on the graph, which is used for zooming.

3.8.2 Measurement Markers (Graphs Only)

Use the **Measurement Marker Mode** toolbar icon to add a measurement marker line to a view. A measurement marker line identifies the data value at a location and allows you to measure the distance between multiple locations on a graph.

Click the icon to switch to Measurement mode. Then, you see marker lines as you move the mouse around the graph. You can click on the graph to add a marker at that position. You stay in the "add marker" mode until you add a marker or click the Measurement Marker icon again.

The legend area above the graph shows the X and Y values of markers. Right-click inside the graph to enable or disable the **Legend** from the shortcut menu.

If you create multiple measurement markers, the legend also shows the distance (or delta) between consecutive data points. For example, as:

```
X2-X1 = 792  Y1-Y2 = 2.4
```

To add a marker, move the mouse to a location on the graph, right-click and select **Insert Measurement Mark**.

To move a marker to a different location on the graph, hold down the Shift key and drag a marker to a new location.

To remove a marker from the view, right-click on the graph, select **Remove Measurement Mark** and click on an individual marker. Or, double-click on a measurement marker to remove it. To remove all the markers, right-click on the graph and select **Remove All Measurement Marks**.

The drop-down menu to the right of the Measurement Marker icon allows you to select the following marker modes:

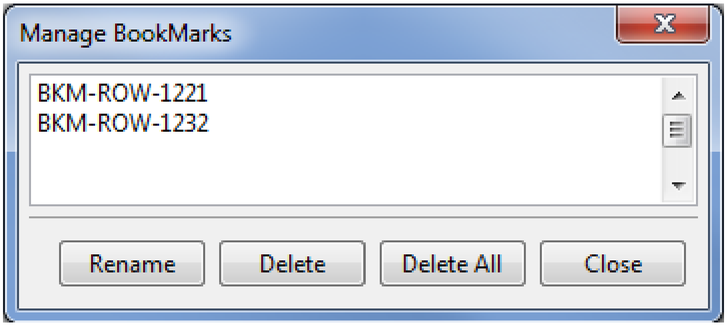
- **Freeform** is the default mode, which lets you add a marker line at any point on the view.
- **Snap to Data** forces you to add markers only at data points. When you move the mouse over the graph in this mode, you see circles on the four closest data points and a dot on the closest data point. Click on the graph to add a marker at the closest data point.
- **X-axis/Y-axis/Both** determines whether placing a marker adds lines that intersect the x-axis, the y-axis, or both axes.

3.8.3 Bookmarks

Use the **Bookmarks** toolbar icon to create a bookmark on any data point of a graph or table. The bookmark will be displayed as a vertical red dashed line in a graph or a row with a red background in a table.

You can use the drop-down next to the icon to jump to a previously created bookmark. Each bookmark is automatically assigned an ID string. A bookmarks applies only to the view in which you created it.

Choose **Manage the Bookmarks** from the drop-down list to open a dialog that lets you rename or delete bookmarks.



BKM-ROW-1221
BKM-ROW-1232

3.8.4 Groups and Synchronous Scrolling

You can group data views together based on common data such as time values. Grouped views are scrolled synchronously to let you easily navigate to interesting points. For example, if you group the CPU load graph with the Log view, then if you click on the CPU Load graph, the Log view displays the closest record to where you clicked in the graph.

To enable grouping, toggle on the **View with Group** icon on the toolbar. Then, simply move the cursor in a grouped table or on a graph as you normally would.

For graphs, the x-axis is used for the common reference value. For tables you can define the reference column. Also, you can use the drop-down to define multiple view groups.

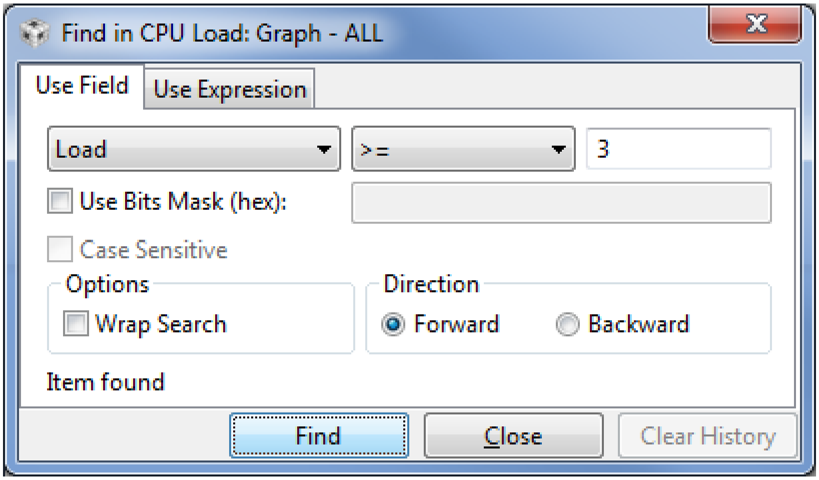
In graphs you can use the **Align Horizontal Center** and **Align Horizontal Range** icons to determine whether this view should be grouped according to the center value currently displayed on the x-axis or the full range of values displayed on the x-axis.

3.8.5 Find

Click to open a dialog that lets you locate a record containing a particular string in one of the fields or a record whose fields satisfy a particular expression. Clicking **Find** repeatedly moves you through the data to each instance of the desired value or string.

The **Use Field** tab is best for simple searches that compare a field value using common operators such as ==, <, != etc. Follow these steps in the **Use Field** tab:

1) Click the **Find** icon in the toolbar.
2) Select the **Use Field** tab.



Find in CPU Load: Graph - ALL

Use Field | Use Expression

Load ▾ | >= ▾ | 3

□ Use Bits Mask (hex):

□ Case Sensitive

Options: □ Wrap Search

Direction: ● Forward ○ Backward

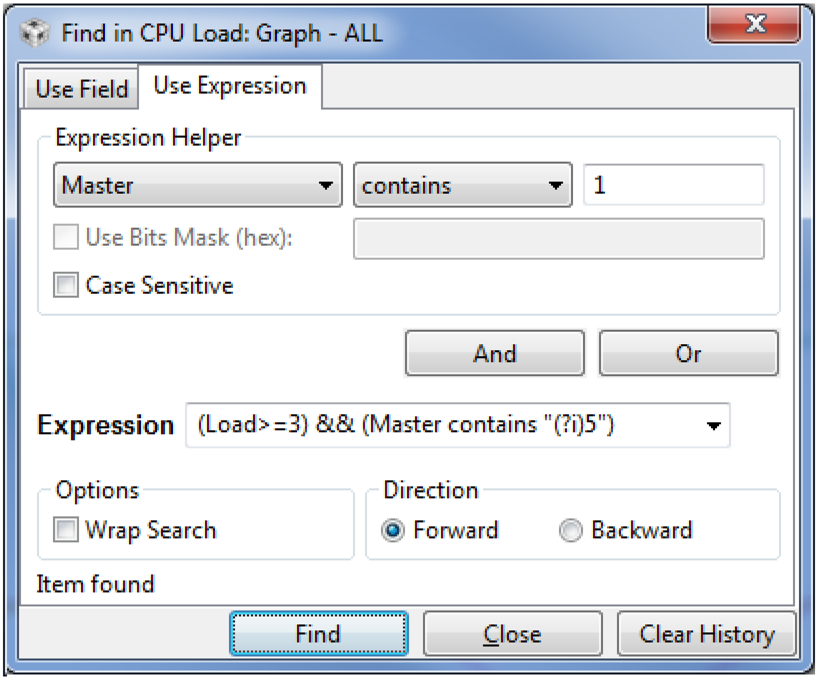
Item found

3) Select a field name from the left drop-down list. This list shows all the data columns used in the detail view for this analysis feature.
4) Select an operation from the middle drop-down list. The operators depend on the datatype for the field you selected.
5) Type a field value for the comparison in the text box.
6) [Optional] Check the **Use Bits Mask (hex)** box and specify a hexadecimal bit mask in the adjacent field if you want to exclude a portion of a value from consideration.
7) [Optional] Check the **Case Sensitive** box if you want a case-sensitive search.
8) [Optional] Check the **Wrap Search** box if you want to continue searching from the top of the table once the end is reached.
9) [Optional] Select a **Direction** option for the search.
10) Click **Find** to start the search.

The **Use Expression** tab lets you enter a regular expression for pattern matching and lets you combine expressions with Boolean operators. Follow these steps in the **Use Expression** tab:

1) Click the **Find** icon in the toolbar.

2) Select the **Use Expression** tab.



Master ▾ contains ▾ 1

□ Use Bits Mask (hex):

□ Case Sensitive

Expression (Load>=3) && (Master contains "(?i)5") ▾

□ Wrap Search

● Forward ○ Backward

Item found

3) Create a regular expression within the **Expression** text box. Visit the link for info on creating expressions used to find data. You can type a regular expression directly or use the Expression Helper to assemble the expression. To use the Expression Helper, follow these sub-steps:

   - Select a field name from the left drop-down list. This list shows all data columns used in the detail view for this analysis feature.
   - Select an operation from the middle drop-down list. The operators depend on the datatype for the field you selected.
   - Type a field value for the comparison in the text box.
   - [Optional] Check the **Use Bits Mask (hex)** box and specify a hexadecimal bit mask in the adjacent field if you want to exclude a portion of a value from consideration.
   - [Optional] Check the **Case Sensitive** box if you want a case-sensitive search.

- Click **And** or **Or** to create the regular expression and add it to the existing statement in the **Expression** text box.

4) [Optional] Check the **Wrap Search** box if you want to continue searching from the top of the table once the end is reached.
5) [Optional] Select a **Direction** option for the search.
6) Click **Find** to start the search.

To clear the drop-down list of previously searched items in the **Expression** field, click **Clear History**.

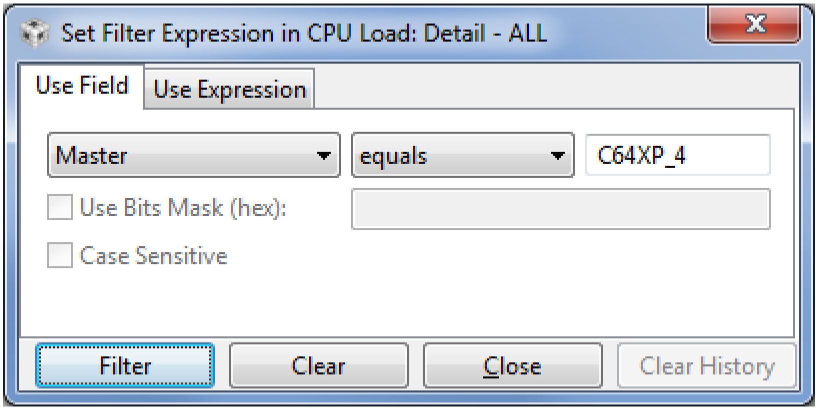
Information about regular expression syntax is widely available on the web.

3.8.6 Filter

Click to open a dialog that filter the view to display only records that contain a particular string in one of the fields or records whose fields satisfy a particular expression.

The **Use Field** tab is best for simple filters that compare a field value using common operators such as ==, <, != etc. Follow these steps in the **Use Field** tab:

1) Click the **Filter** icon in the toolbar.
2) Select the **Use Field** tab.



Set Filter Expression in CPU Load: Detail - ALL

Master ▾ equals ▾ C64XP_4

□ Use Bits Mask (hex):

□ Case Sensitive

3) Select a field name from the left drop-down list. This list shows all the data columns used in the detail view for this analysis feature.
4) Select an operation from the middle drop-down list. The operators depend on the datatype for the field you selected.
5) Type a field value for the comparison in the text box.

6) [Optional] Check the **Use Bits Mask (hex)** box and specify a hexadecimal bit mask in the adjacent field if you want to exclude a portion of a value from consideration.

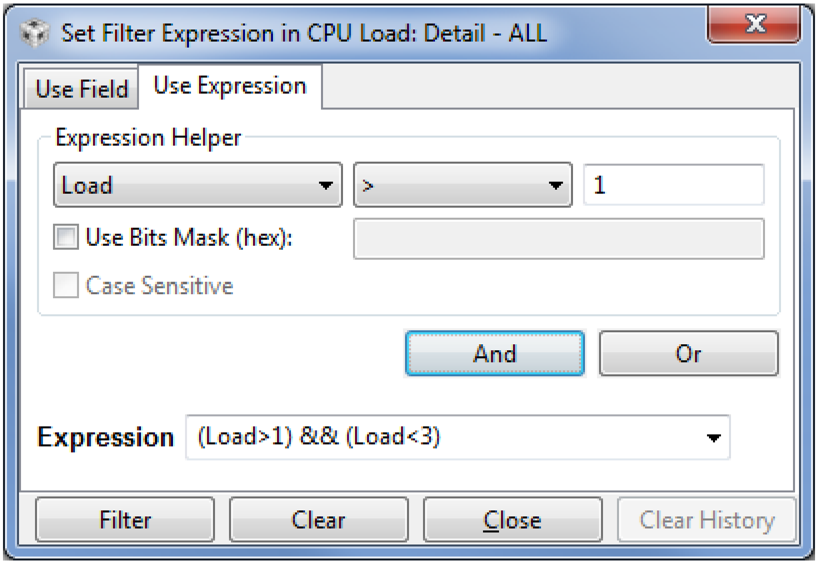
7) [Optional] Check the **Case Sensitive** box if you want a case-sensitive filter.

8) Click **Filter** to limit the records or data points displayed.

The **Use Expression** tab lets you enter a regular expression for pattern matching and lets you combine expressions with Boolean operators. Follow these steps in the **Use Expression** tab:

1) Click the **Filter** icon in the toolbar.

2) Select the **Use Expression** tab.



Load | > | 1

☐ Use Bits Mask (hex):

☐ Case Sensitive

Expression (Load>1) && (Load<3)

3) Create a regular expression within the **Expression** text box. Visit the link for info on creating expressions used to filter data. You can type a regular expression directly or use the Expression Helper to assemble the expression. To use the Expression Helper, follow these sub-steps:

- Select a field name from the left drop-down list. This list shows all data columns used in the detail view for this analysis feature.
- Select an operation from the middle drop-down list. The operators depend on the datatype for the field you selected.
- Type a field value for the comparison in the text box.

- [Optional] Check the **Use Bits Mask (hex)** box and specify a hexadecimal bit mask in the adjacent field if you want to exclude a portion of a value from consideration.
- [Optional] Check the **Case Sensitive** box if you want a case-sensitive search.
- Click **And** or **Or** to create the regular expression and add it to the existing statement in the **Expression** text box.

4) Click **Filter** to limit the records or data points displayed.

To clear the drop-down list of previously searched items in the **Expression** field, click **Clear History**.

Information about regular expression syntax is widely available on the web.

3.8.7 Export

You can save data in a table or graph view to an external file by using the **Data > Export** commands. All columns contained in the table (not just the displayed columns) and the displayed graph numbers are placed into a comma-separated value file format (*.csv filename extension).

Numeric values are stored in the CSV format using a general format. You can use spreadsheet software such as Microsoft Excel to perform additional computations or create annotated charts from the exported information.

To export data to an external CSV file:

1) Select a table or a graph view.
2) If you want to export only some rows from a table view, hold down the Shift key and select a range of rows or hold down the Ctrl key while selecting multiple rows.
3) Right-click on the table or graph and select **Data > Export All** or **Data > Export Selected** from the right-click menu.
4) In the Save As dialog, browse for the location where you want to save the file and type a filename. Click **Save**.
5) Open the file you created using a spreadsheet or other software program. Alternately, you can later reopen the CSV file in an MCSA session as described in Section 4.4.1.

3.8.8 Cursor and Scroll Lock

Data views scroll to the end whenever new data is received. If you click on a point in a graph or table while data is updating, automatic scrolling is stopped, even though data is still being added at to the end.

To continue scrolling to the end automatically, toggle off the **Scroll Lock** button on the toolbar.

Note that if you have enabled grouping (the icon is toggled on), the scroll lock icon does not lock the scrolling of grouped views.

Use the **Freeze** icon on the toolbar to freeze the data updates and automatic refreshing completely.

Chapter 4

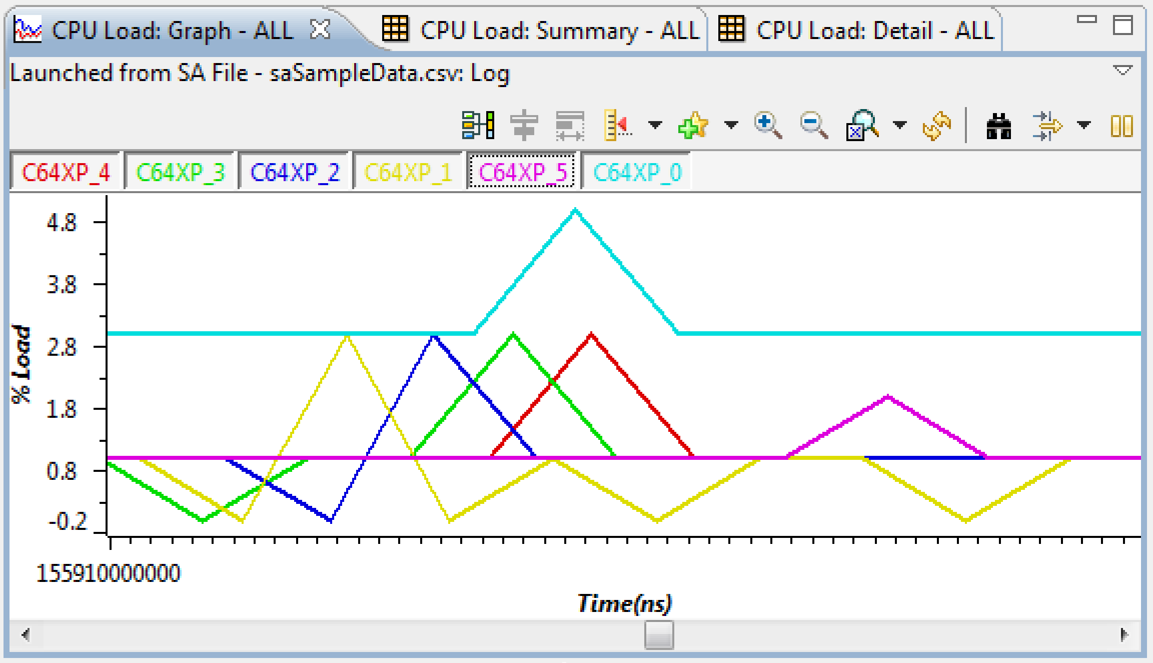
Using MCSA in Code Composer Studio

This chapter describes the host-side analysis features provided in Code Composer Studio for examining instrumentation data sent to the host.

Topic **Page**

4.1 Overview of Multicore System Analyzer (MCSA) Features

The DVT upgrade to support MCSA includes analysis features for viewing the CPU and thread loads, the execution sequence, thread durations, and context profiling. The features include graphs, detailed logs, and summary logs.



You can use these features at run-time and can also record the run-time data for later analysis.

- **Run-time analysis.** Real-time data analysis can be performed without the need to halt the target program. This ensures that actual program behavior can be observed, since halting multiple cores can result in threading that differs from real-time behavior.
- **Recording and playback of data.** You can record real-time data and later reload the data to further analyze the activity. The MCSA let you record and playback using both CSV and binary files.

This chapter describes how to set up, start, and stop data collection and analysis. It also describes how to use specific MCSA features.

4.2 Starting a Live MCSA Session

To gather live MCSA data, you need an instrumented application running on the target(s). If it has not already been done, you need to enable UIA logging by configuring the application as described in Section 5.1, *Quickly Enabling UIA Instrumentation*. Once you have enabled UIA logging, SYS/BIOS applications provide UIA data to the MCSA features in CCS.

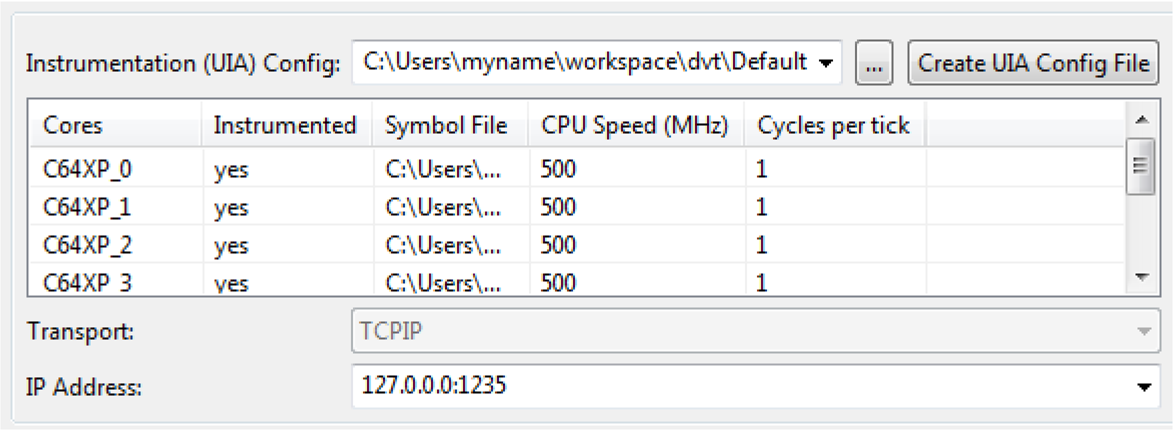
To start a live MCSA session, follow these steps:

1) If the application is not already loaded and running, load it in CCS.

2) Create a CCS target configuration and start a debugging session, This enables MCSA to auto-configure your session. (Alternatively, you can create a UIA configuration and save it to a file as described in Section 4.3. If you use a configuration from a file, you do not need to be running a CCS debugging session, because MCSA data is collected via Ethernet transports rather than streaming JTAG.)

3) In CCS, choose the **Tools > Multicore System Analyzer (MCSA) > Live** menu command. You will see the Live Parameters dialog.

4) The top section of the dialog lets you customize the UIA configuration, which controls how MCSA connects to the target.



Instrumentation (UIA) Config: C:\Users\myname\workspace\dvt\Default

Cores	Instrumented	Symbol File	CPU Speed (MHz)	Cycles per tick
C64XP_0	yes	C:\Users\...	500	1
C64XP_1	yes	C:\Users\...	500	1
C64XP_2	yes	C:\Users\...	500	1
C64XP 3	yes	C:\Users\...	500	1

Transport: TCPIP

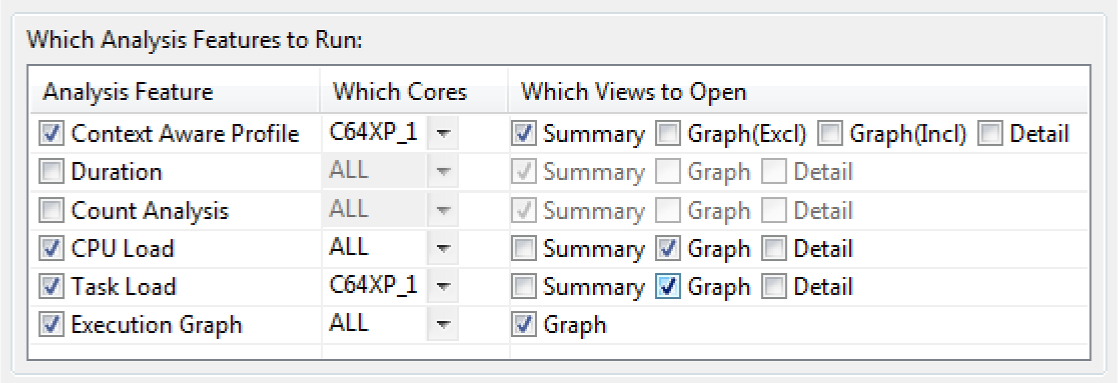
IP Address: 127.0.0.0:1235

- **Instrumentation (UIA) Config:** By default, the UIA configuration is automatically generated using the current target configuration for the CCS debug session, the .out file that is currently loaded, and auto-detected IP addresses.

- **Browse:** You can click "..." to browse for a UIA configuration you have saved to a *.usmxml file. See Section 4.3 for more about creating UIA Configuration files.

- **Create UIA Config File:** This button opens a dialog that lets you create and save a configuration.
- **Cores:** The auto-detected target cores or cores defined in the UIA configuration file are shown below the UIA Config field.
- **Transport:** The transport that is auto-detected or specified in the UIA Config file you selected is shown here.
- **IP Address:** The IP address of the target board is autodetected and shown here. You can type a different value if the value shown is incorrect.

5) The next section of the dialog lets you select how to view the data that will be collected.

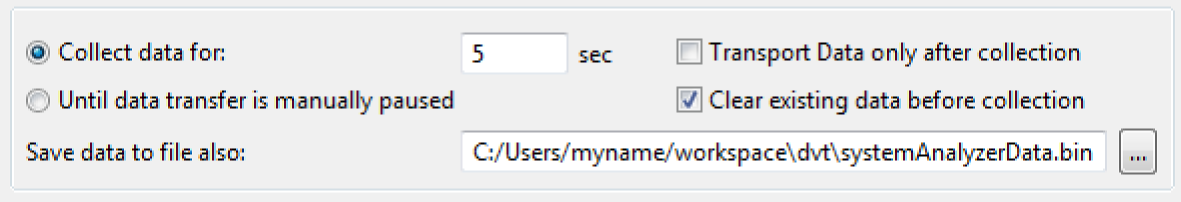


Which Analysis Features to Run:

Analysis Feature	Which Cores	Which Views to Open
☑ Context Aware Profile	C64XP_1	☑ Summary □ Graph(Excl) □ Graph(Incl) □ Detail
□ Duration	ALL	☑ Summary □ Graph □ Detail
□ Count Analysis	ALL	☑ Summary □ Graph □ Detail
☑ CPU Load	ALL	□ Summary ☑ Graph □ Detail
☑ Task Load	C64XP_1	□ Summary ☑ Graph □ Detail
☑ Execution Graph	ALL	☑ Graph

- **Analysis Feature:** Choose features you want to use. These features will process events that apply to them. (You can run additional analysis features after starting the session.)
- **Which Cores:** Choose whether to display events from ALL cores or a single core. The drop-down list shows the core names for endpoints in your current UIA configuration along with core names for any active CCS target configuration for a debugging session. For the Context Aware Profile and Task Load features, a specific core name is required (not ALL), and you can select or type the name.
- **Which Views to Open:** Choose the view types you want to open automatically. You can later open more views, but these checkboxes provide an easy way to open a number of them.

6) The last section of the dialog lets you specify how the data will be collected and stored.



● Collect data for: 5 sec

○ Until data transfer is manually paused

□ Transport Data only after collection

☑ Clear existing data before collection

Save data to file also: C:/Users/myname/workspace\dvt\systemAnalyzerData.bin

- **Collect data for:** Type the number of seconds to collect data. The default is 5 seconds. This time is measured on the host unless you check the **Transport Data only after collection** box. Choose **Until data transfer is manually paused** if you want to collect data until you pause or stop the data collection or halt the application.

- **Transport Data only after collection:** Check this box if you want to get event data from the target only after the collection time expires. That is, the target will run for the specified time without transferring data; the target restarts the transfer at the end of the time. Using this option reduces the intrusiveness of the transport while data is being collected. However the amount of data that can be collected is limited to the log buffer size. This option is supported only if you are using an Ethernet transport (that is, you are using Upload_NONJTAGTRANSPORT mode for the LoggingSetup.eventUploadMode property) and if your target supports control messages.

- **Clear existing data before collection:** Check this box if you want to erase any log records stored in the log buffers on the target when data collection starts. This option is not available if you are using a JTAG-based transport.

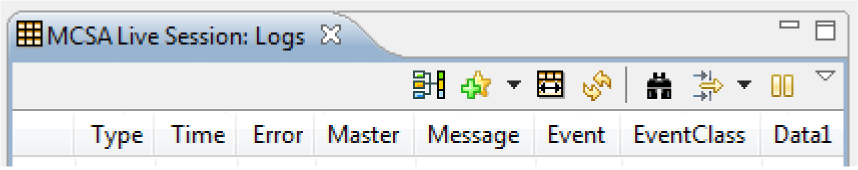
- **Save data to file also:** By default, live data is shown in the Log view and is saved to a binary file called systemAnalyzerData.bin. Later, you can reopen the file to analyze it further. You can browse to select a different filename or file location. If you do not want to create a file, clear the file path in this field. If you choose a binary file that contains records from a previous run, the records will be cleared from the file at the start of the run.

Note: If you let MCSA collect events for a significant amount of time (or even for a short period of time if the data rate is very high), your system resources will get used up resulting in slow performance.

Note: Depending on the data rate, multi-core event correlation may not work well when viewing live data. If events seem out of order, save the data to a binary file, then open the binary file for later processing to improve the event correlation.

Note: Multi-core event correlation for events uploaded via JTAG transports is an unsupported feature in UIA 1.0.

7) Click **Run** to connect to the target and show live data collection in the Log view. See Section 4.5 for information about using the Log view.



If you start a new live session when a live session is already running, the current session is closed so the new session can be opened. You can run only one live session or binary file session at a time. (You can have multiple CSV files open along with a single live session or binary file.)

If you want to save instrumentation data to a CSV file (instead of a binary file), right-click on the Log view and choose **Data > Export All.** For information about using binary and CSV files that store instrumentation data, see Section 4.4, *Opening CSV and Binary Files Containing MCSA Data*.

4.2.1 Managing an MCSA Session

You can manage MCSA sessions—including live sessions and binary files—with commands in the Log view toolbar and right-click menu and the **Tools > Multicore System Analyzer (MCSA) >** *session* menu from the main CCS menu bar. You can use the commands from the following list that apply to the current state of the session:

- **Connect/Disconnect** establishes or disconnects a connection to the target(s) using the UIA configuration during a live session. If you are using a binary file, the Connect command reruns the file. Data is cleared from the MCSA views when you use the Connect command.
- **Pause/Resume Data Decoding** halts and resumes processing of records received from the target. Available only for live sessions.
- **Clear Log View** erases all records from the Log view. Available only for live sessions.

- **Restart** causes data transfer to start over. This has the same effect as using the Disconnect command followed by the Connect command. Data is cleared from the MCSA views when you use this command. Available only for live sessions.
- **Pause/Resume Transfer** halts and resumes the transfer of records from the target(s) to the host. This command is supported only if you are using an Ethernet transport (that is, you have set LoggingSetup.eventUploadMode to Upload_NONJTAGTRANSPORT) and if your target supports control messages. Available only for live sessions.
- **Freeze/Resume Data Update** halts and resumes updates to the current view as a result of decoding data. If you pause data updates with this icon, data decoding continues in the background.
- **Scroll Lock** lets you examine records as data is being collected without having the display jump to the end whenever new records are added to the view. See Section 3.8.8.

Data is processed in the following sequence: data transfer, then data decoding, then data updates. So, for example, if you use **Pause Transfer**, data that has already been transferred to the host will be decoded and updated in the display; once all transferred data has been processed, the data decoding and updates will need to wait for more data. Similarly, if you **Pause Data Decoding**, data updates will need to wait once all the decoded records have been displayed.

The following commands are available only from the Log view right-click menu for CSV and binary files.

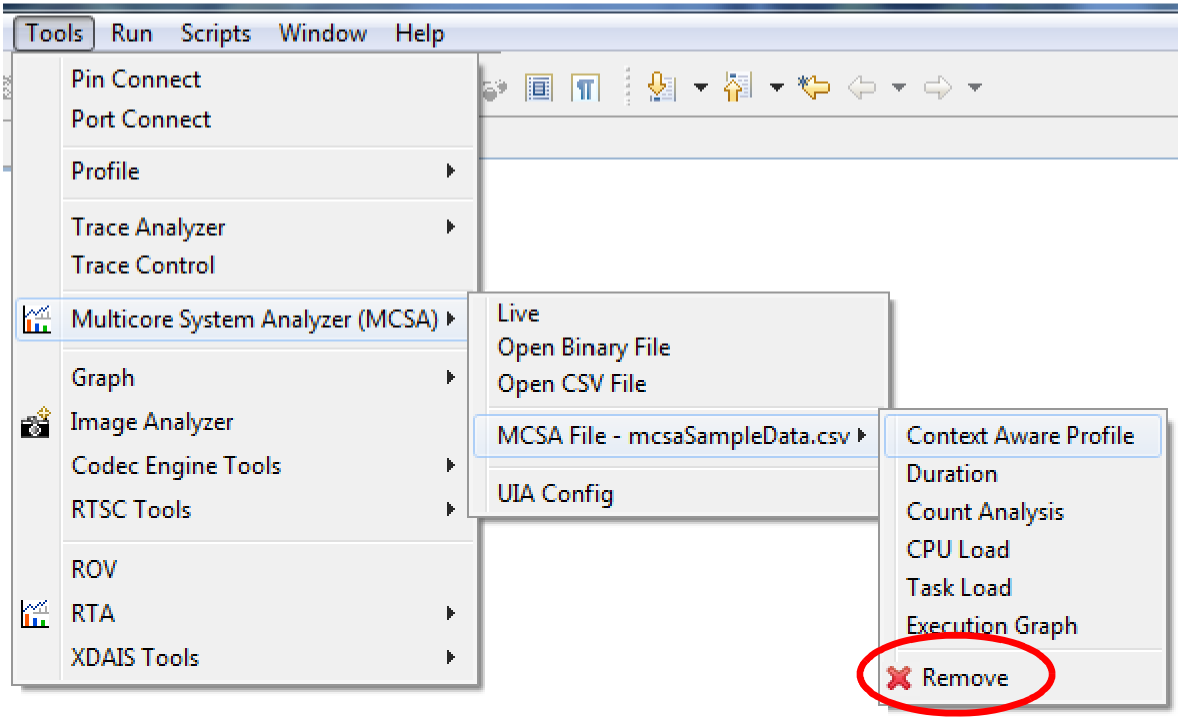
- **Stop** halts processing of the current file.
- **Clear Data** clears all data in all open MCSA feature views.
- **Open File** lets you select a CSV or binary file to open and process.

For descriptions of more toolbar icons, see page 3–20 and page 4–19.

4.2.2 Removing an MCSA Session

MCSA sessions—including live sessions, binary files, and CSV files—remain open even if you close all the views of the session including the Log view. You may want to close a session, for example, in order to open a different live session or a binary file, since only one of these can be open at a time.

In order to completely close a session, choose **Tools > Multicore System Analyzer (MCSA) >** *session_name* **> Remove** from the main CCS menu bar.

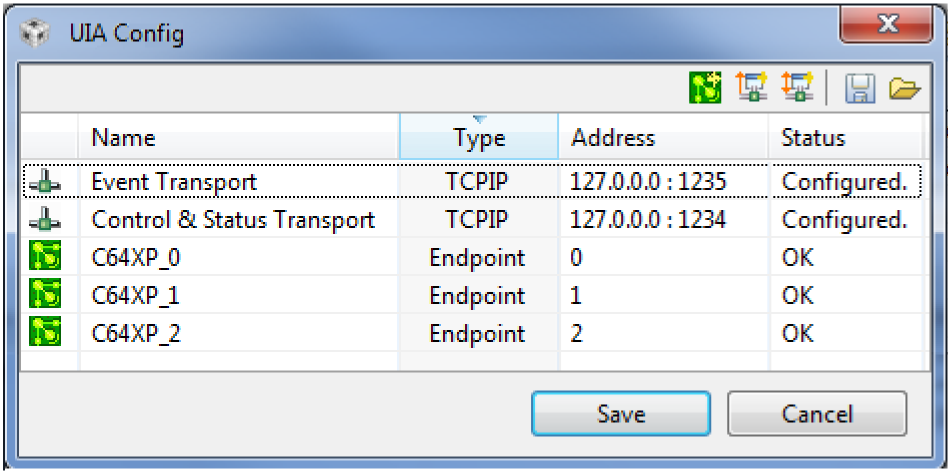


4.3 Configuring MCSA Transports and Endpoints

MCSA data is collected via an Ethernet transport that needs a configuration within CCS in order to collect data. The easy way to configure MCSA is to start a Debugger session in CCS with the target configuration for your platform. MCSA can usually get enough information from the .out file that was loaded, the target configuration, and the auto-detected IP address of the target to auto-configure the information it needs.

Alternatively, you can create a UIA configuration and save it to a file as described here. If you use later collect data using a configuration file, you do not need to be running a CCS debugging session, because MCSA data is collected via Ethernet transports rather than streaming JTAG.

A UIA configuration specifies the transports used to send logs and commands between the target and host. It also specifies the cores that will be sending data to MCSA. You can save a configuration after you create it for later use.



Name	Type	Address	Status
Event Transport	TCPIP	127.0.0.0 : 1235	Configured.
Control & Status Transport	TCPIP	127.0.0.0 : 1234	Configured.
C64XP_0	Endpoint	0	OK
C64XP_1	Endpoint	1	OK
C64XP_2	Endpoint	2	OK

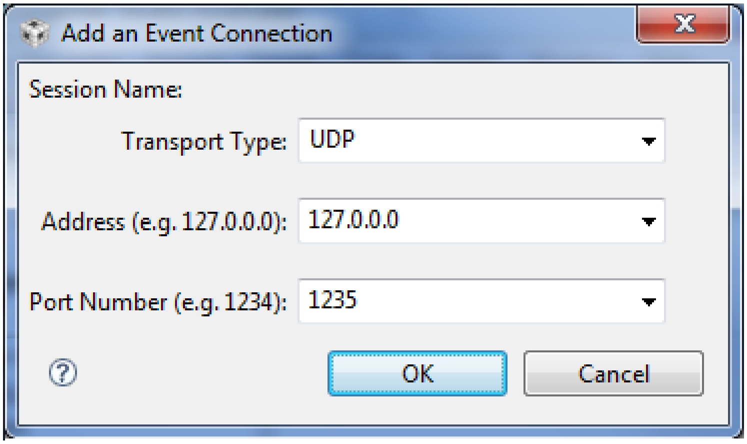
Notice in the figure above that a UIA configuration contains an Event Transport, a Control and Status Transport, and endpoints for each core in the application.

To create a UIA configuration, follow these steps:

1) In CCS, choose the **Tools > Multicore System Analyzer (MCSA) > UIA Config** menu command.

2) Click the **Event Transport** icon in the UIA Config dialog. This lets you specify the transport used for moving logs from the target to the host.

3) In the Add an Event Connection dialog, provide details about the transport you want to use. Different target connections require different transports.



Add an Event Connection

Session Name:

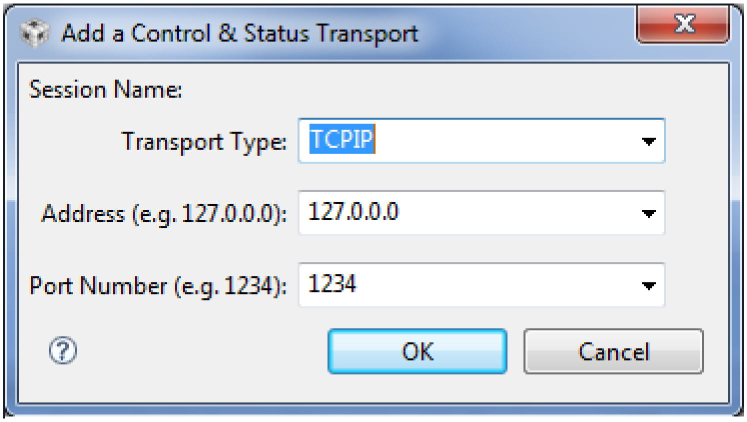
Transport Type: UDP

Address (e.g. 127.0.0.0): 127.0.0.0

Port Number (e.g. 1234): 1235

- The Transport Type options are UDP, TCPIP, JTAG, STM, and FILE. The default is UDP.
- The Address is the IP address of the target or of the master core in the case of a multi-core application.
- The Port is the TCP or UDP port number. The default is 1235.

4) Click the **Control & Status Transport** icon. This transport is used for sending and receiving commands. Different target connections require different transports.



Add a Control & Status Transport

Session Name:

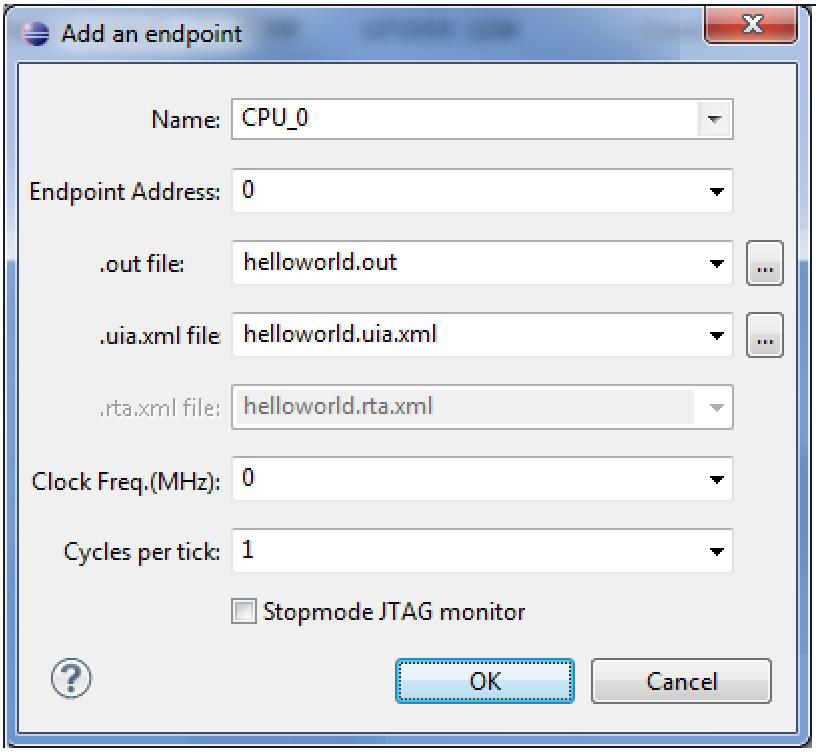
Transport Type: TCPIP

Address (e.g. 127.0.0.0): 127.0.0.0

Port Number (e.g. 1234): 1234

- Either TCP/IP or UDP can be used as the transport type. The default transport type is TCP/IP. If you are using a JTAG event transport, set the control and status transport type to "NONE".
- The default port is 1234.

5) For each core in your application, click the **Endpoint** icon. An "endpoint" is a description of a core and the target application it is running. This provides the host with a list of cores to use in interpreting MCSA log data.



Add an endpoint

Name: CPU_0

Endpoint Address: 0

.out file: helloworld.out

.uia.xml file helloworld.uia.xml

.rta.xml file: helloworld.rta.xml

Clock Freq.(MHz): 0

Cycles per tick: 1

□ Stopmode JTAG monitor

- **Name.** Type the name of the target. If a CCS debug session is running and the target configuration matches that of your target application, you can select a name from the drop-down list. The actual name chosen here is not important, but it is best to use the same names here and the CCS Target Configuration.

- **EndPoint Address.** This is the number of the core starting from 0. For example, use 0 for CPU 0, 1 for CPU1, and so on. These numbers must correspond to the ServiceMgr module’s Core ID, which usually defaults to the index number of the ti.sdo.utils.MultiProc ID from IPC.

- **.out file.** The filename of the compiled and linked target application. Click the **...** button and browse to find the file. The file extension may by .out or may contain the platform, for example, .x64P. When you click OK, the dialog checks to make sure this file exists.

- **.uia.xml file.** Name of the generated MCSA metadata file. This is an XML file that is created if your target application includes any UIA modules. It is typically auto-discovered when you select a .out file and click OK. If the file cannot be found automatically, click **...** and browse to find the file. For example, the file may be stored in the Default\configPkg\package\cfg subdirectory of the project when you build the project.
- **.rta.xml file.** Name of the generated RTA metadata file. This file is created if your target application includes the RTA module. It is typically auto-discovered when you select a .out file and click OK. If the file cannot be found automatically, click **...** and browse to find the file. This file is likely to be stored in the same directory as the .uia.xml file.
- **Clock freq (MHz).** Type the clock speed for this CPU in MHz. If you do not provide the correct value here, the durations reported by MCSA will not be converted to nanoseconds correctly.
- **Cycles per tick.** Type the number of cycles per clock tick for this CPU here. If you do not provide the correct value here, the durations reported by MCSA will not be converted to nanoseconds correctly.
- **Stopmode JTAG monitor.** Check this box if you want records to be transferred via JTAG when the target is halted. That is, check this box if the target application on this endpoint is configured to use any of the following settings for the eventUploadMode parameter of the LoggingSetup module: Upload_SIMULATOR, Upload_PROBEPOINT, or Upload_JTAGSTOPMODE.

6) Once you have created both transports and the endpoints for each core, save the configuration by clicking the **Save** button. Browse for a directory to contain the file and type a filename. The Save As dialog shows that the configuration is saved in a file with an extension of .usmxml. The .usmxml file is used if you want to specify a saved configuration to use in a live session (page 4–3) or when opening a binary file (page 4–15). Behind the scenes, a file with the same name but a .xml extension is also saved, but you can ignore the .xml file.

If you want to edit an item in the configuration, you can double-click on it or right-click and select **Edit the selected item** to open the dialog used to create that item. To delete an item, right-click and select **Delete the selected item**.

To load a UIA configuration, click the 📂 **Open** icon in the toolbar. Browse for and open a configuration file you have saved (with a file extension of .usmxml).

4.4 Opening CSV and Binary Files Containing MCSA Data

The MCSA features can save analysis data in two different file types:

- **CSV files** include UIA configuration information, so you *do not* need a UIA configuration in order to use a CSV file. See Section 4.4.1 for information about opening MCSA data saved to a CSV file.
- **Binary files** do not include UIA configuration information, so you *do* need a UIA configuration in order to see analysis data saved to a binary file. See Section 4.4.2 for information about opening MCSA data saved to a binary file.

A sample CSV data file is provided with MCSA so that you can try the MCSA features immediately.

See Section 4.2 for information about creating binary files and Section for information about creating CSV files.

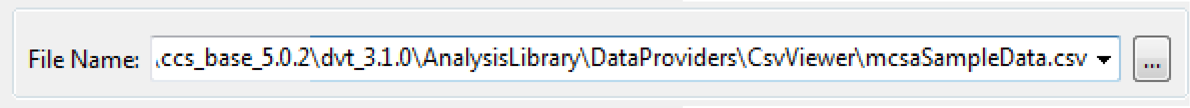
4.4.1 Opening a CSV File with MCSA

You can experiment with the host-side MCSA features using a CSV (comma-separated values) data file that is provided with the DVT installation. This file is a recording of instrumentation data collected in a run-time session using a 6-core EVM6472 application.

Since CSV files contain data that is already decoded, no UIA configuration is needed to open a CSV file containing MCSA data.

To load the provided CSV file, follow these steps:

1) Start Code Composer Studio 5.0. You do not need to open a project or launch a debug session in order to use the analysis features with a pre-recorded CSV file.
2) Choose the **Tools > Multicore System Analyzer (MCSA) > Open CSV File** menu command.
3) In the CSV File Parameters dialog, click the "…" button to the right of the **File Name** field.
4) Browse to the <*ccs_install*>\ccsv5\ccs_base_5.0.x.xx\dvt_3.1.x.xx\AnalysisLibrary\DataProviders\CsvViewer folder, where x.xx is the latest version of CCS and DVT you have installed.
5) Select the mcsaSampleData.csv file and click **Open**.

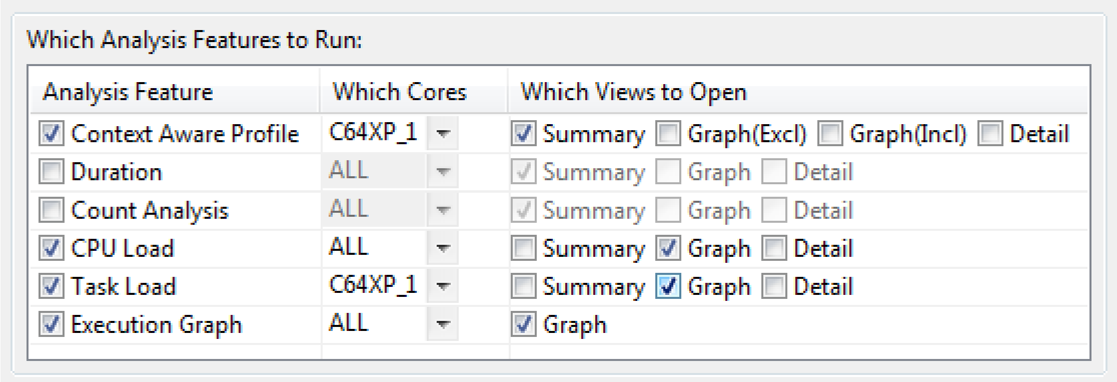


File Name: \ccs_base_5.0.2\dvt_3.1.0\AnalysisLibrary\DataProviders\CsvViewer\mcsaSampleData.csv

6) In the **Analysis Feature** column, choose features you want to use. These features will process events that apply to them when you open the CSV file. (You can run additional analysis features after you open the file.)

7) In the **Which Cores** column, choose whether to display events from ALL cores or a single core. The drop-down list shows the core names for endpoints for any active CCS target configuration for a debugging session. For the Context Aware Profile and Task Load features, a specific core name is required (not ALL), and you can select or type the name.

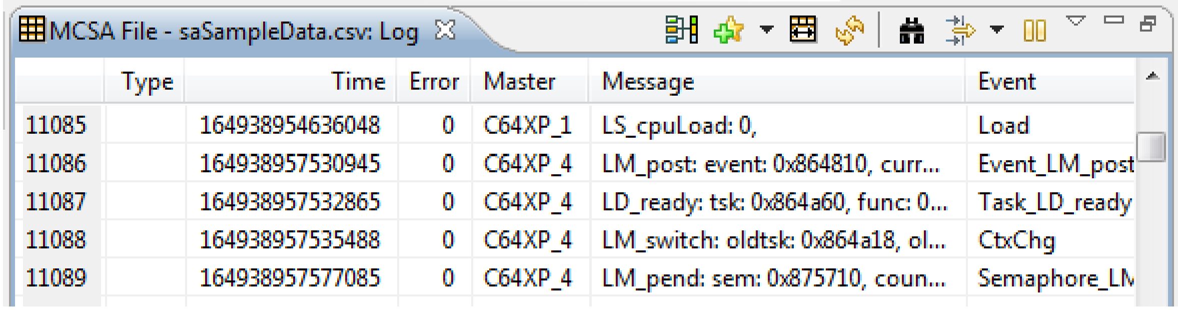
8) In the **Which Views to Open** column, choose the view types you want to open automatically. You can later open more views, but these checkboxes provide an easy way to open a number of them. For this example, check the following boxes:



Which Analysis Features to Run:

Analysis Feature	Which Cores	Which Views to Open
☑ Context Aware Profile	C64XP_1 ▾	☑ Summary ☐ Graph(Excl) ☐ Graph(Incl) ☐ Detail
☐ Duration	ALL ▾	☑ Summary ☐ Graph ☐ Detail
☐ Count Analysis	ALL ▾	☑ Summary ☐ Graph ☐ Detail
☑ CPU Load	ALL ▾	☐ Summary ☑ Graph ☐ Detail
☑ Task Load	C64XP_1 ▾	☐ Summary ☑ Graph ☐ Detail
☑ Execution Graph	ALL ▾	☑ Graph

9) Click **Run**. You will see the MCSA Log view, which displays the events stored in the CSV file.



MCSA File - saSampleData.csv: Log

	Type	Time	Error	Master	Message	Event
11085		164938954636048	0	C64XP_1	LS_cpuLoad: 0,	Load
11086		164938957530945	0	C64XP_4	LM_post: event: 0x864810, curr...	Event_LM_post
11087		164938957532865	0	C64XP_4	LD_ready: tsk: 0x864a60, func: 0...	Task_LD_ready
11088		164938957535488	0	C64XP_4	LM_switch: oldtsk: 0x864a18, ol...	CtxChg
11089		164938957577085	0	C64XP_4	LM_pend: sem: 0x875710, coun...	Semaphore_LM

10) See Section 4.5 for information about how to use the Log view.

After learning to use MCSA features, you can analyze data from your own applications and record your own sessions as CSV files. See page 4–19 for information about creating your own CSV files.

4.4.2 Opening a Binary File with MCSA

Opening a binary file that you saved during a run-time session lets you do later analysis of the results. See Section 4.2 for information about creating binary files.

You can load a binary file that contains MCSA data if you have a UIA configuration that matches the configuration with which the analysis data was saved or if you have a Debugger session open that matches the target configuration for the target used to create the binary file. (In contrast, opening a CSV file containing MCSA data does not require a UIA configuration because the data is already decoded.)

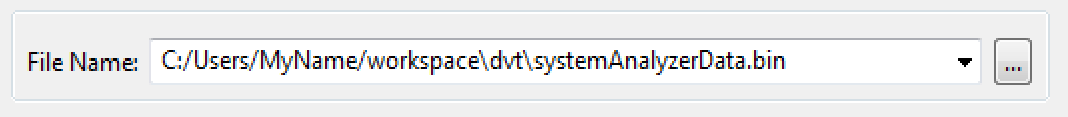
To load a binary file containing MCSA event logs, follow these steps:

1) Start Code Composer Studio 5.0.

2) Create a CCS target configuration and start a debugging session, This enables MCSA to auto-configure your session. (Alternatively, you can create a UIA configuration and save it to a file as described in Section 4.3.)

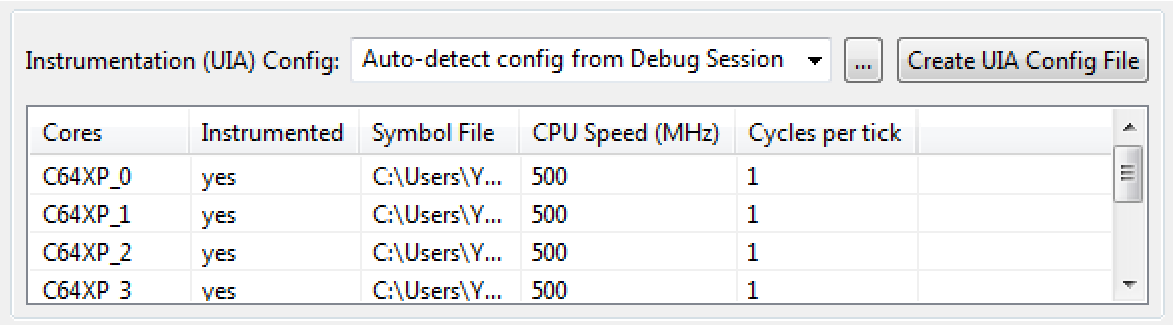
3) Choose the **Tools > Multicore System Analyzer (MCSA) > Open Binary File** menu command.

4) In the first section of the Binary File Parameters dialog, browse for the binary file you want to open. The default file is systemAnalyzerData.bin in your workspace directory.



File Name: C:/Users/MyName/workspace\dvt\systemAnalyzerData.bin

5) The next section of the dialog lets you customize the UIA configuration, which controls how MCSA interprets the cores referenced in the binary file.

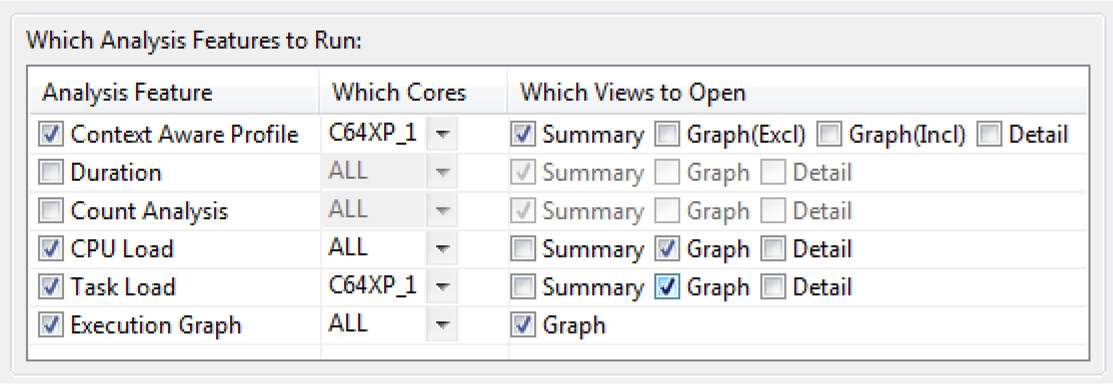


Instrumentation (UIA) Config: Auto-detect config from Debug Session

Cores	Instrumented	Symbol File	CPU Speed (MHz)	Cycles per tick
C64XP_0	yes	C:\Users\Y...	500	1
C64XP_1	yes	C:\Users\Y...	500	1
C64XP_2	yes	C:\Users\Y...	500	1
C64XP_3	yes	C:\Users\Y...	500	1

- **Instrumentation (UIA) Config:** By default, the UIA configuration is automatically generated using the current target configuration for the CCS debug session, the .out file that is currently loaded, and auto-detected IP addresses.
- **Browse:** You can click "..." to browse for a UIA configuration you have saved to a *.usmxml file. The endpoint definitions in the file are used to interpret the binary file and must match the endpoints used when the binary file was saved. (The transport definitions are ignored since the data has already been transported.) See Section 4.3 for more about creating UIA Configuration files.
- **Create UIA Config File:** This button opens a dialog that lets you create and save a configuration.
- **Cores:** The list of cores is shown below the UIA Config field.

6) The last section of the dialog lets you select how to view the data.



Which Analysis Features to Run:

Analysis Feature	Which Cores	Which Views to Open
☑ Context Aware Profile	C64XP_1	☑ Summary ☐ Graph(Excl) ☐ Graph(Incl) ☐ Detail
☐ Duration	ALL	☑ Summary ☐ Graph ☐ Detail
☐ Count Analysis	ALL	☑ Summary ☐ Graph ☐ Detail
☑ CPU Load	ALL	☐ Summary ☑ Graph ☐ Detail
☑ Task Load	C64XP_1	☐ Summary ☑ Graph ☐ Detail
☑ Execution Graph	ALL	☑ Graph

- In the **Analysis Feature** column, choose features you want to use. These features will process events that apply to them when you open the CSV file. (You can run additional analysis features after you open the file.)
- In the **Which Cores** column, choose to display events from ALL cores or a single core. The drop-down list shows core names for endpoints in the selected UIA configuration and in the current CCS target configuration. For the Context Aware Profile and Task Load features, a specific core name is required (not ALL); you can select or type a name.
- In the **Which Views to Open** column, choose views to open automatically. You can later open more views, but these checkboxes provide an easy way to open a number of them.

7) Click **Run** to open the binary file you selected. This opens the MCSA Log view and displays the events stored in the binary file.

8) See Section 4.5 for information about how to use the Log view.

4.5 Using the Log View

The Log view opens automatically when you start a live data collection session or open a binary or CSV file containing MCSA data.

Log View Column Descriptions

The MCSA Log view shows the details of all records. The log contains the following columns:

- **Row Number.** This column indicates only the row number in the current log display. If you filter the records displayed, all the row numbers change.
- **Type.** Displays the event type. For live sessions and binary files, an icon is shown. For CSV files, a numeric value is shown. This field lets you quickly locate or filter certain types of events. A message can have multiple types, such as Error and Analysis; the type with the lower value is shown. For filtering, the type has a value from 0 to 11.

Table 4–1 Event Types

Icon	Value	Type	Comments
	0	Unknown	
⊗	1	Error	
⚠	2	Warning	
i	3	Information	
👓	4	Details	
♻	5	Life Cycle	
📊	6	Analysis	
M1	7	Module 1	Module-specific type
M2	8	Module 2	Module-specific type
M3	9	Module 3	Module-specific type
	10	Emergency	
	11	Critical	

- **Time.** The time the event was logged in nanoseconds. The time in this column has been adjusted and correlated on the host to provide the global time based on a common timeline. The time is converted to nanoseconds using the clock and cycle information provided in the endpoint for each CPU in the UIA configuration.
- **Error.** A 1 in this column indicates that a a data loss error occurred. A 2 indicates that an out-of-sequence error occurred. A 3 indicates

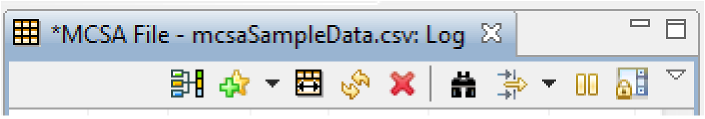
that both types of error occurred. Data loss errors are detected if SeqNo values are missing on a per logger basis. Out-of-sequence errors are determined by comparing Global Time values. Such errors can indicate that either records from a single core or between cores are out-of-sequence. Individual views display a message in the bottom status line if a data loss is detected.

- **Master.** The core on which the event was logged. For example, C64XP_0 and C64XP_1.
- **Message.** A printf-style message that describes the logged event. Values from the Arg0 through Arg8 arguments are plugged into the message as appropriate. In general, messages beginning with "LM" are brief messages, "LD" indicates a message providing details, "LS" messages contain statistics, and "LW" indicates a warning message.
- **Event.** The type of event that occurred. Supported events include the following:
  - Synchronization events (at startup)
  - CtxChg (for context change)
  - Pend, post, and block events from various modules
  - Load, ready, start, and stop events from various modules
  - Set priority and sleep events from the Task module
- **EventClass.** The class of the event that occurred. Supported event classes include:
  - CPU (for Load events, for example from the ti.sysbios.utils.Load module)
  - TSK (for task threads)
  - HWI (for hardware interrupt threads)
  - SWI (for software interrupt threads)
  - FUNC (for some Start and Stop events, for example from the ti.uia.events.UIABenchmark module)
- **Data1.** The main information returned with an event. For many events, Data1 returns the name of the task or the thread type in which the event occurred.
- **Data2.** Further information returned with an event. For load events, for example, Data2 returns the load percentage.
- **SeqNo.** The sequence number with respect to the source logger on the source core. Each logger has its own sequence numbering. This number is used when detecting data loss.

- **Logger.** The name of the logger to which the event was sent. UIA creates several default loggers, and your target code can configure additional loggers.
- **Module.** The module that defines the type of event that occurred. This may be a UIA module, SYS/BIOS module, XDCtools module, or a RTSC module from some other software component. CCS adds an underscore before the module name for certain types of events for internal processing for other views.
- **Domain.** The module that logged the event.
- **Local Time.** The local timestamp on the core where the event was logged. This timestamp typically has a higher resolution than the global timestamp. Local timestamps are not correlated and adjusted to show timing interactions with events on other cores.
- **Arg1 to Arg 8.** Raw arguments passed with the event. Although the number of arguments associated with an event is not limited, typically events are logged with 0, 1, 2, 4, or 8 arguments. If more than 8 arguments are logged, only the first 8 are shown here.

Log View Toolbar Icons and Right-Click Menu Commands

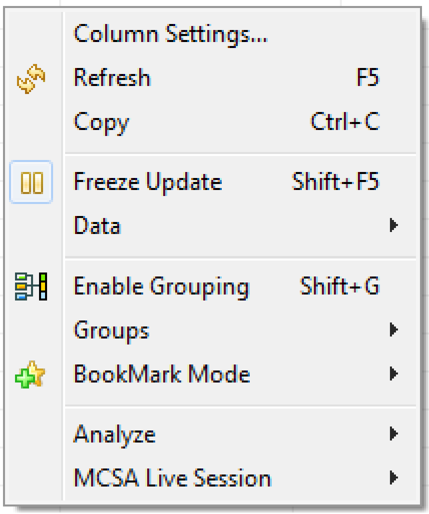
The Log view contains a number of toolbar icons that let you control the view’s behavior.



- Toggle **View With Group** on and off (Shift+G). A "group" synchronizes views of instrumentation data so that scrolling in one view causes similar movement to happen automatically in another. For example, if you group the CPU load graph with the Log view, then click on the CPU Load graph, the Log view displays the closest record to where you clicked in the graph. See Section 3.8.4.
- Click to turn on **Bookmark Mode**. The next record you click in the log will be highlighted in red. Jump to a bookmarked event by using the drop-down list next to the Bookmark Mode icon. Choose **Manage the Bookmarks** from the drop-down list to open a dialog that lets you rename or delete bookmarks. See Section 3.8.3.
- **Auto Fit Columns** sets the column widths in the Log to fit their current contents.
- **Refresh** updates the GUI displays (F5). This button does not collect data from the target.

- **Remove** closes this MCSA session or file and all the associated analysis features and views. If you close the Log view, you are asked if you want to remove this session.
- Open the **Find In** dialog to search this log. See Section 3.8.5.
- **Filter** the log records to match a pattern by using the Set Filter Expression dialog. See Section 3.8.6.
- Select **Row Count** from this drop-down to toggle the column that shows row numbers in the log on and off.

Additional icons described in Section 4.2.1 differ depending on whether you are running a live session or a stored file session and let you control data transfer activity.



You can right-click on the Log view to choose from a menu of options. In addition to toolbar commands, you can use the following additional commands from the right-click menu:

- **Column Settings.** Opens a dialog that hides or displays various columns. You can change the alignment, font, and display format of a column (for example, decimal, binary, or hex).
- **Copy.** Copies the selected row or rows to the clipboard.
- **Enable Auto Scroll.** Allows the log to scroll freely as new data is available. See Section 3.8.8.
- **Data > Export Selected.** Lets you save the selected rows to a CSV (comma-separated value) file. See Section 3.8.7.
- **Data > Export All.** Lets you save all the rows to a CSV file. For example, you might do this so that you can perform statistical analysis on the data values.
- **Groups.** Lets you define and delete groups that contain various types of log messages. See Section 3.8.4.
- **Analyze.** Run one of the analysis features. See Section 4.6.
- **MCSA File or Session.** Lets you pause, resume, and reset a live or binary file session. Lets you stop or reset a CSV file session. See Section 4.2.1, *Managing an MCSA Session*.

4.6 Opening MCSA Features

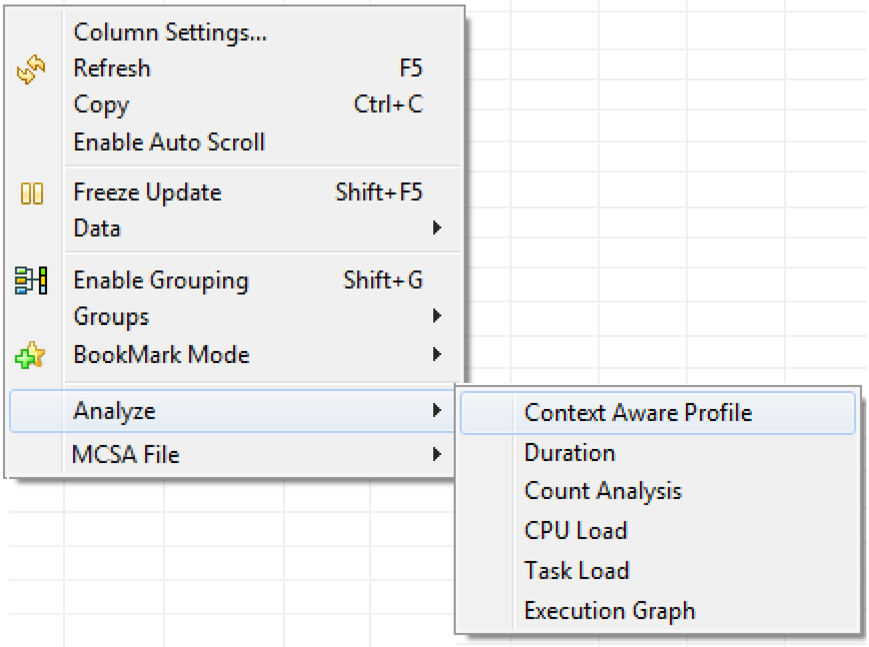
You can open MCSA analysis features in the dialog you use to launch a session. After you start a session, you can open additional analysis features and views. The following analysis features are available:

- **CPU Load.** Shows the SYS/BIOS load data collected for all cores in the system. See Section 4.7.
- **Task Load.** Shows the CPU load data measured within a SYS/BIOS Task thread. See Section 4.8.
- **Execution Graph.** Shows threads on the target(s). See Section 4.9.
- **Count Analysis.** Tracks values on the target. See Section 4.10.
- **Duration.** Calculates the total duration between pairs of execution points on the target. See Section 4.11.
- **Context Aware Profile.** Calculates duration with awareness of interruptions by other threads and functions. See Section 4.12.

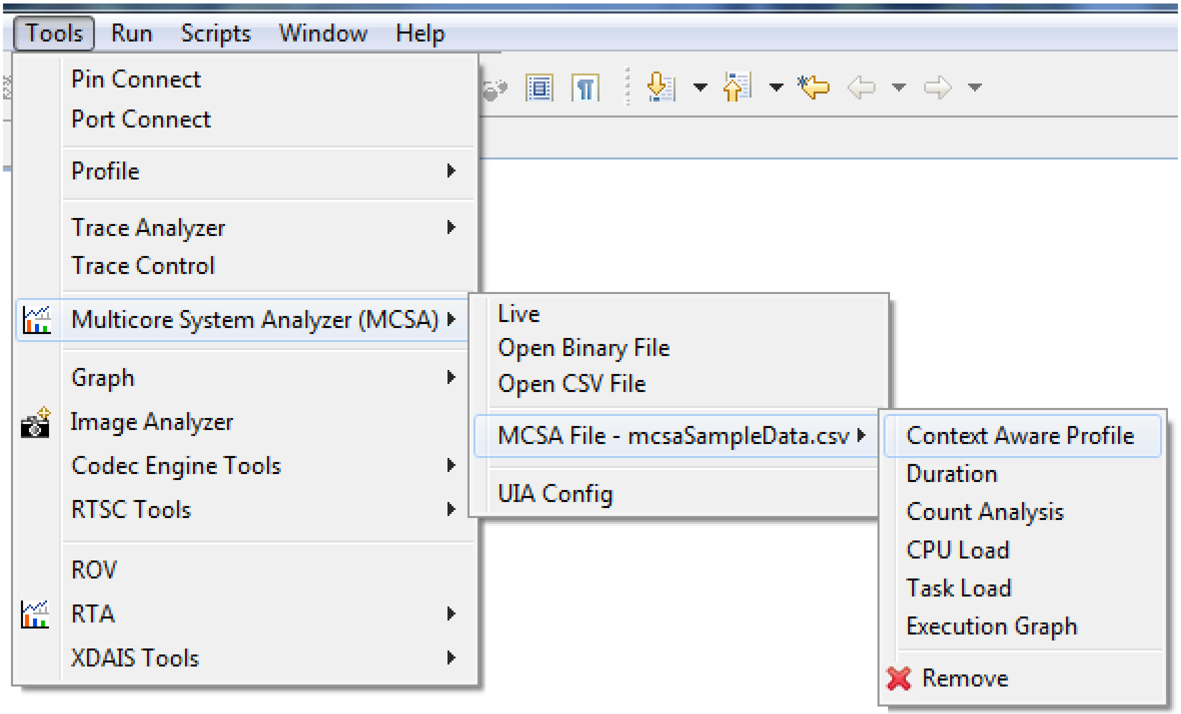
The CPU Load, Task Load, and Execution Graph features display information automatically provided by SYS/BIOS. The Count Analysis, Duration, and Context Aware Profile features display data only if you modify your target code to instrument the required events.

To run an analysis feature, do either of the following:

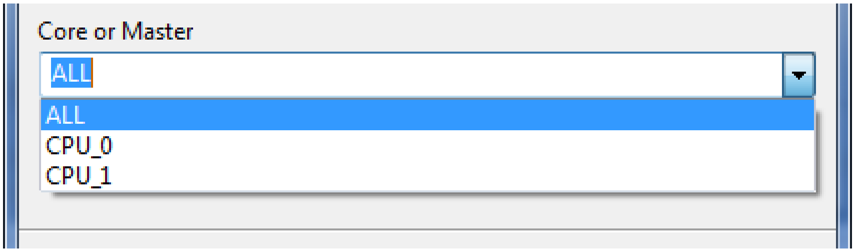
- Right-click on the Log view. From the context menu, choose **Analyze** followed by the analysis feature to run.



- Open the **Tools > Multicore System Analyzer (MCSA)** menu in the main CCS menu bar. Select your MCSA session from the list. Then select the analysis feature to run. The default view for the feature you choose will open.



When you choose to open an analysis feature, you are prompted to choose whether to display events from ALL cores or a single core. You can type the name of a core in this field. The drop-down list shows the core names for endpoints in your currently selected UIA configuration file along with core names for any active CCS target configuration for a debugging session.



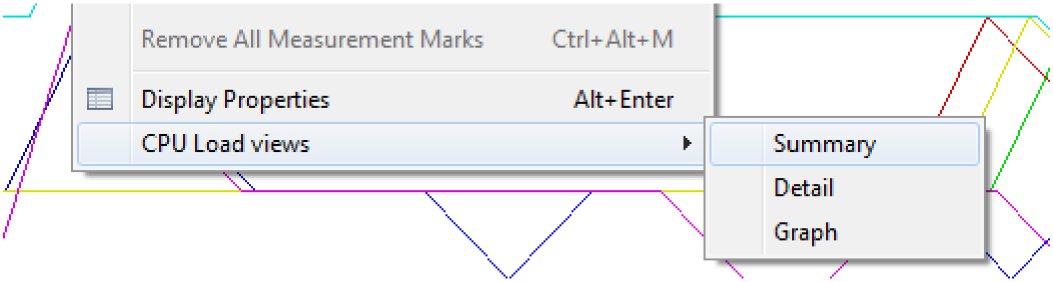
Most analysis features support several views. For example: a summary table, detail table, and a graph. The following table shows the types of views available for each feature:

Table 4–2 Views Available for Various Analysis Features

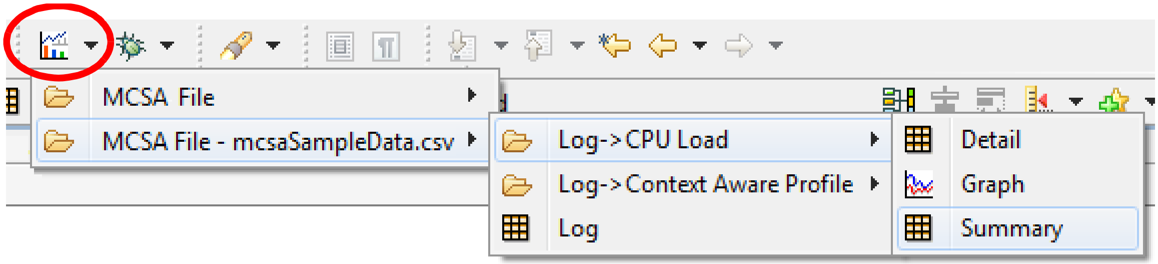
Feature	Summary View	Detail View	Graph View
Log	No	Yes (default)	No
CPU Load	Yes	Yes	Yes (default)
Task Load	Yes	Yes	Yes (default)
Execution Graph	No	No	Yes (default)
Count Analysis	Yes (default)	Yes	Yes
Duration	Yes (default)	Yes	Yes
Context Aware Profile	Yes (default)	Yes	Yes (2 graphs: Exclude and Include)

To open views other that the default view for an analysis feature, do any of the following:

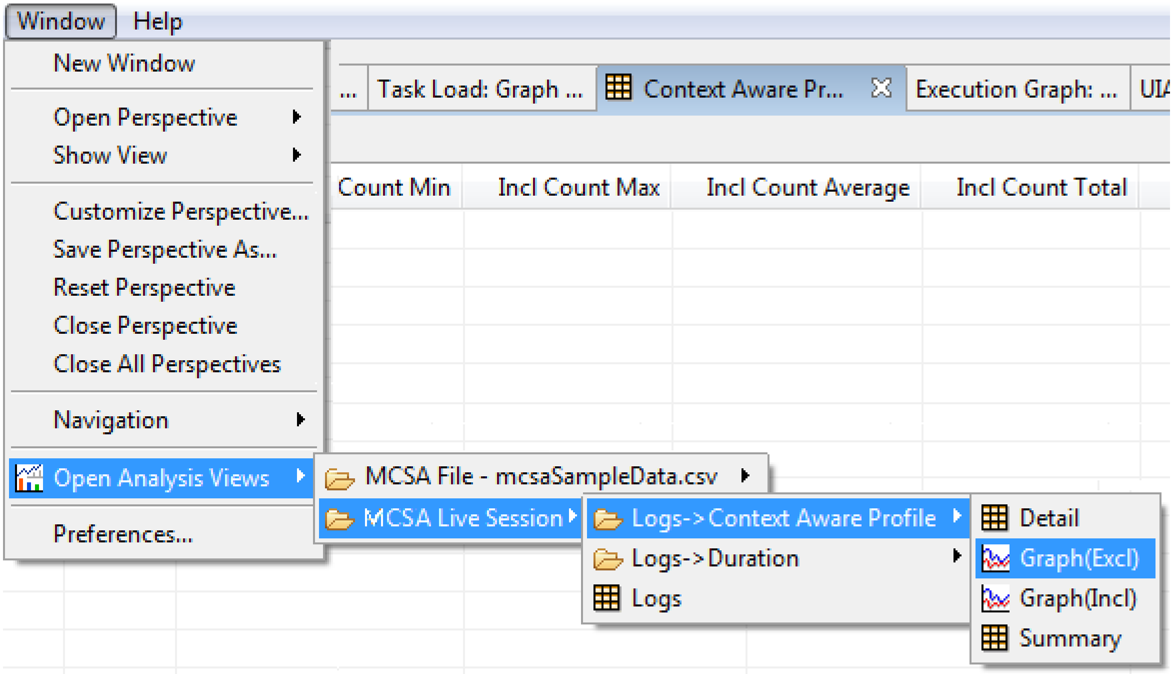
- ❑ Right-click on an analysis view and choose another view for that analysis feature. For example, in the CPU Load graph, you can right-click and choose **CPU Load views > Summary**.


Remove All Measurement Marks Ctrl+Alt+M
Display Properties Alt+Enter
CPU Load views
Summary
Detail
Graph

- ❑ Use the ▦▾ **Analysis View** icon in the main CCS toolbar. Select your MCSA file or session from the drop-down list. Then select the analysis feature you want to use. (Only features that have been run are listed.) Then select the view you want to see.



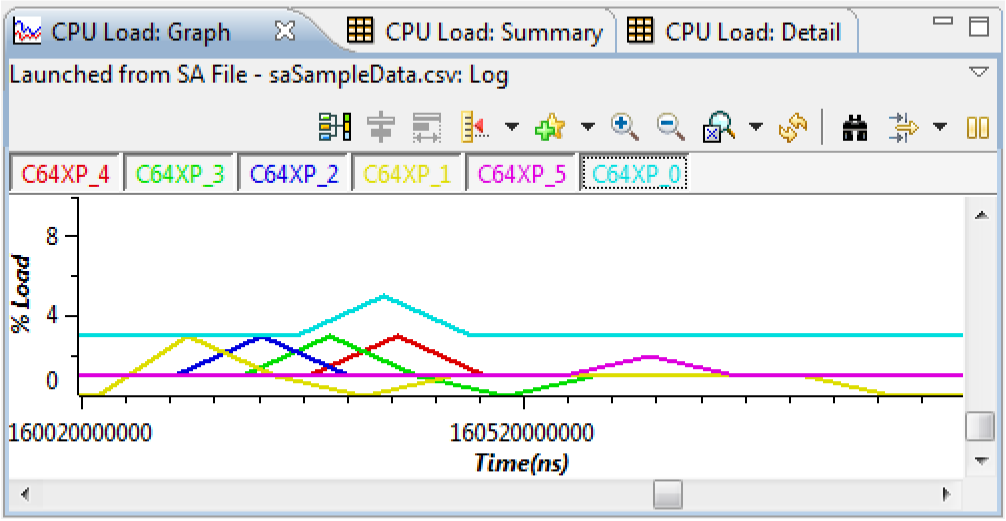
❑ Open the **Window > Open Analysis Views** menu in the CCS menu bar. Select your MCSA file or session from the list. Then select the analysis feature you want to use. (Only features that are currently open are listed.) Then select the view you want to see.



You can synchronize the scrolling and cursor placement in views for the same session by clicking ⊞ **View with Group** icon in the toolbars of the views you want to synchronize.

4.7 Using the CPU Load View with MCSA

The CPU Load feature shows the SYS/BIOS load data collected for all CPUs in the system. The CPU load is the percentage of time a CPU spends running anything other than the SYS/BIOS Idle loop.



To open this feature, choose **Tools > Multicore System Analyzer >** *session_name* **> CPU Load** from the CCS menu bar.

Graph View for CPU Load

The Graph view opens by default. It shows the change in CPU load (as a percentage) with time for each CPU. Clicking on the name of a CPU above the graph highlights the corresponding line in the graph. (If you do not see these buttons, right click on the graph and choose **Legend**.)

Use the toolbar buttons to group (synchronize), measure, zoom, search, and filter the graph. Right-click on the graph to adjust the display properties of the graph.

To open Summary or Detail views for this feature, use the **Analysis View** icon in the main CCS toolbar or right-click on the CPU Load graph and choose from the **CPU Load views** submenu.

❑ The Summary view presents the minimum, maximum, and average CPU load. See Section 4.7.1.

❑ The Detail view presents the raw CPU load data. See Section 4.7.2.

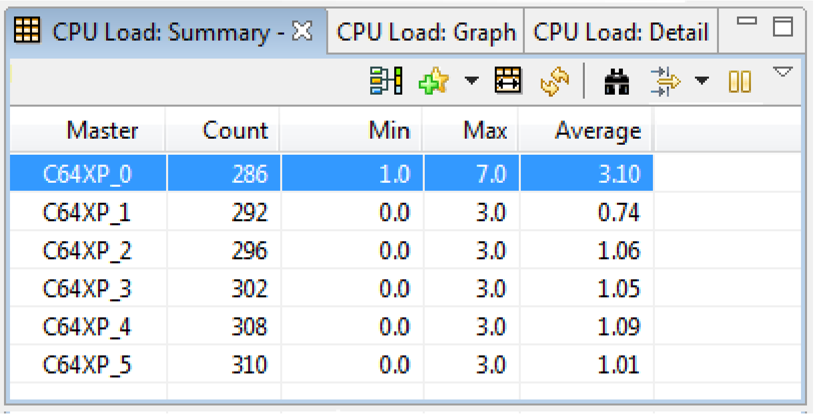
See Also

❑ Section 3.2, *Analyzing System Loading with MCSA*

4.7.1 Summary View for CPU Load

To open the Summary view for the CPU Load feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a CPU Load view and choose **CPU Load views > Summary**.

The Summary view for the CPU Load feature shows the count, minimum, maximum, and average of the reported CPU load measurements for each CPU.



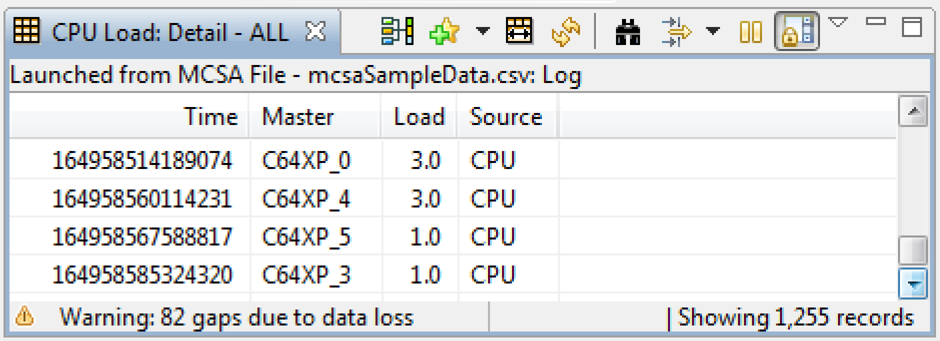
Master	Count	Min	Max	Average
C64XP_0	286	1.0	7.0	3.10
C64XP_1	292	0.0	3.0	0.74
C64XP_2	296	0.0	3.0	1.06
C64XP_3	302	0.0	3.0	1.05
C64XP_4	308	0.0	3.0	1.09
C64XP_5	310	0.0	3.0	1.01

- **Master.** The name of the CPU.
- **Count.** The number of CPU load measurements for this CPU.
- **Min.** The minimum CPU load percentage reported for this CPU.
- **Max.** The maximum CPU load percentage reported for this CPU.
- **Average.** The average CPU load percentage for this CPU.

4.7.2 Detail View for CPU Load

To open the Detail view for the CPU Load feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a CPU Load view and choose **CPU Load views > Detail**.

The Detail view of the CPU Load feature shows records that report the CPU load. The status bar tells how many records are shown and how many gaps occurred.



CPU Load: Detail - ALL

Launched from MCSA File - mcsaSampleData.csv: Log

Time	Master	Load	Source
164958514189074	C64XP_0	3.0	CPU
164958560114231	C64XP_4	3.0	CPU
164958567588817	C64XP_5	1.0	CPU
164958585324320	C64XP_3	1.0	CPU

⚠ Warning: 82 gaps due to data loss | Showing 1,255 records

- **Time.** The time (correlated with other cores) of this load event.
- **Master.** The name of the core on which the load was logged.
- **Load.** The CPU load percentage reported.
- **Source.** The source of the load percentage event.

The columns in this view are also displayed in the Log view (but in a different order). See page 4–17 for column descriptions.

4.7.3 How CPU Load Works with MCSA

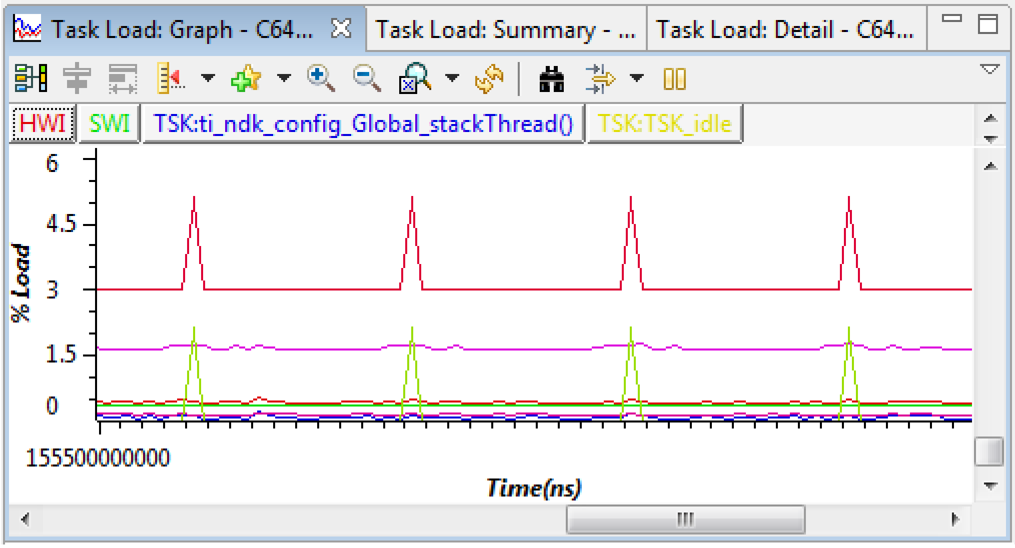
The CPU load is the percentage of time a CPU spends running anything other than the SYS/BIOS Idle loop, which is run by the TSK_idle low-priority Task thread.

The CPU Load feature displays data provided automatically by internal SYS/BIOS calls to functions from the ti.sysbios.utils.Load module. SYS/BIOS threads are pre-instrumented to provide load data using a background thread.

See Section 5.2.1, *Enabling and Disabling Load Logging* for information about how to disable CPU load logging.

4.8 Using the Task Load View with MCSA

The Task Load view shows CPU load data collected on a per-Task and per-thread type basis for the specified CPU. Note that the Task Load feature does not allow you to select all cores; you must select a single core.


Task Load: Graph - C64...
Task Load: Summary - ...
Task Load: Detail - C64...
HWI
SWI
TSK:ti_ndk_config_Global_stackThread()
TSK:TSK_idle
6
4.5
3
1.5
0
% Load
155500000000
Time(ns)

To open this feature, choose **Tools > Multicore System Analyzer >** *session_name* **> Task Load** from the CCS menu bar.

Graph View for Task Load

The Graph view opens by default; it shows the change in load over time on a per-Task basis as a line graph.

Click on the names of Tasks above the graph to highlight those lines in the graph. If you don't see the Task names, right-click on the graph and choose **Legend** from the context menu. If you make the Graph view area wider, more Task names will be shown.

To open other views for the Task Load feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Task Load view and choose from the **Task Load views** submenu.

- The Summary view presents the minimum, maximum, and average load on a per-Task basis. See Section 4.8.1.
- The Detail view presents the raw Task load data. See Section 4.8.2.

Clicking on the name of a thread above the graph highlights the corresponding line in the graph. (If you do not see these buttons, right click on the graph and choose **Legend**.)

Use the toolbar buttons to group (synchronize), measure, zoom, search, and filter the graph. Right-click on the graph to adjust the display properties of the graph.

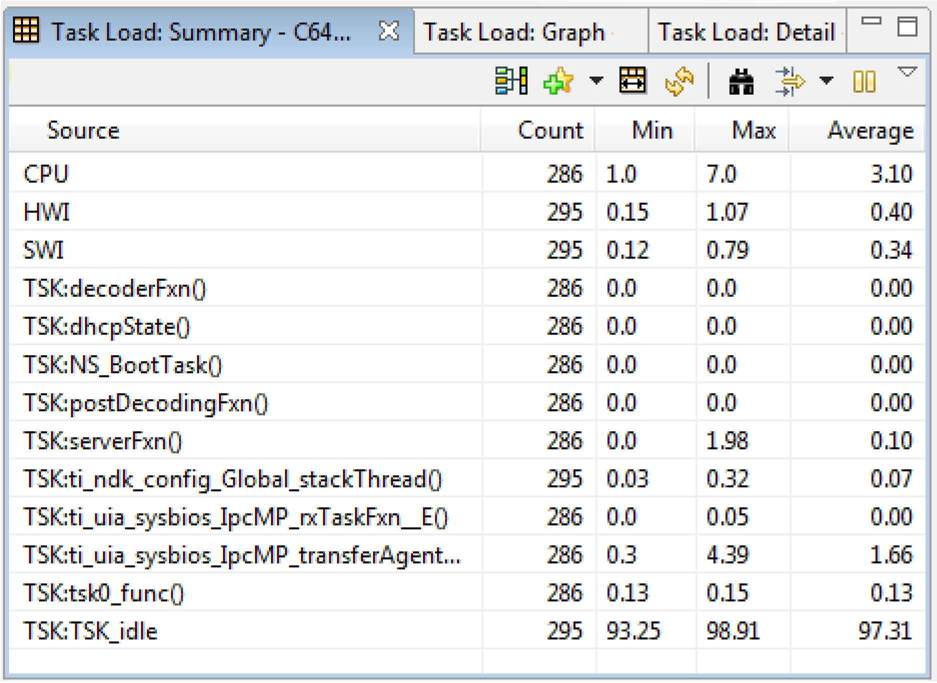
See Also

❑ Section 3.2, *Analyzing System Loading with MCSA*

4.8.1 Summary View for Task Load

To open the Summary view for the Task Load feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Task Load view and choose **Task Load views > Summary**.

The Summary view for the Task Load feature shows the count, minimum, maximum, and average of the reported Task load measurements for each Task.



Source	Count	Min	Max	Average
CPU	286	1.0	7.0	3.10
HWI	295	0.15	1.07	0.40
SWI	295	0.12	0.79	0.34
TSK:decoderFxn()	286	0.0	0.0	0.00
TSK:dhcpState()	286	0.0	0.0	0.00
TSK:NS_BootTask()	286	0.0	0.0	0.00
TSK:postDecodingFxn()	286	0.0	0.0	0.00
TSK:serverFxn()	286	0.0	1.98	0.10
TSK:ti_ndk_config_Global_stackThread()	295	0.03	0.32	0.07
TSK:ti_uia_sysbios_IpcMP_rxTaskFxn__E()	286	0.0	0.05	0.00
TSK:ti_uia_sysbios_IpcMP_transferAgent...	286	0.3	4.39	1.66
TSK:tsk0_func()	286	0.13	0.15	0.13
TSK:TSK_idle	295	93.25	98.91	97.31

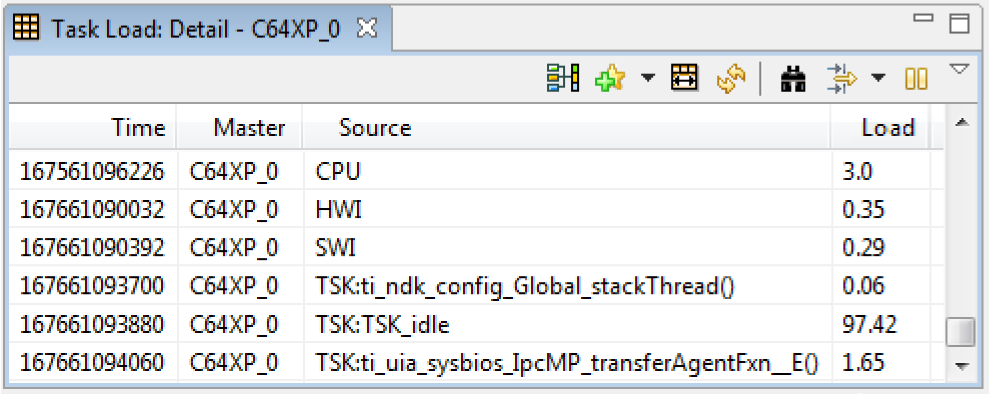
The CPU load is the percentage of time the CPU spent running anything other than the SYS/BIOS Idle loop. The averages for all the sources listed except for the CPU typically add up to approximately 100%. However, that total may be somewhat different if events were dropped, particularly when the load was high.

- **Source.** The name of the task or the thread type.
- **Count.** The number of CPU load measurements reported for this task or thread type.
- **Min.** The minimum CPU load reported for this task or thread type.
- **Max.** The maximum CPU load reported for this task or thread type.
- **Average.** The average CPU load for this task or thread type.

4.8.2 Detail View for Task Load

To open the Detail view for the Task Load feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Task Load view and choose **Task Load views > Detail**.

The Detail view of the Task Load feature shows all records that report the load. These may be for individual Task threads, the Swi module, the Hwi module, or the overall CPU load.



Time	Master	Source	Load
167561096226	C64XP_0	CPU	3.0
167661090032	C64XP_0	HWI	0.35
167661090392	C64XP_0	SWI	0.29
167661093700	C64XP_0	TSK:ti_ndk_config_Global_stackThread()	0.06
167661093880	C64XP_0	TSK:TSK_idle	97.42
167661094060	C64XP_0	TSK:ti_uia_sysbios_IpcMP_transferAgentFxn__E()	1.65

- **Time.** The time (correlated with other cores) of this load event.
- **Master.** The name of the core on which the load was logged.
- **Source.** The name of the task or thread type.
- **Load.** The CPU load percentage reported.

The columns in this view are also displayed in the Log view (but in a different order). See page 4–17 for column descriptions.

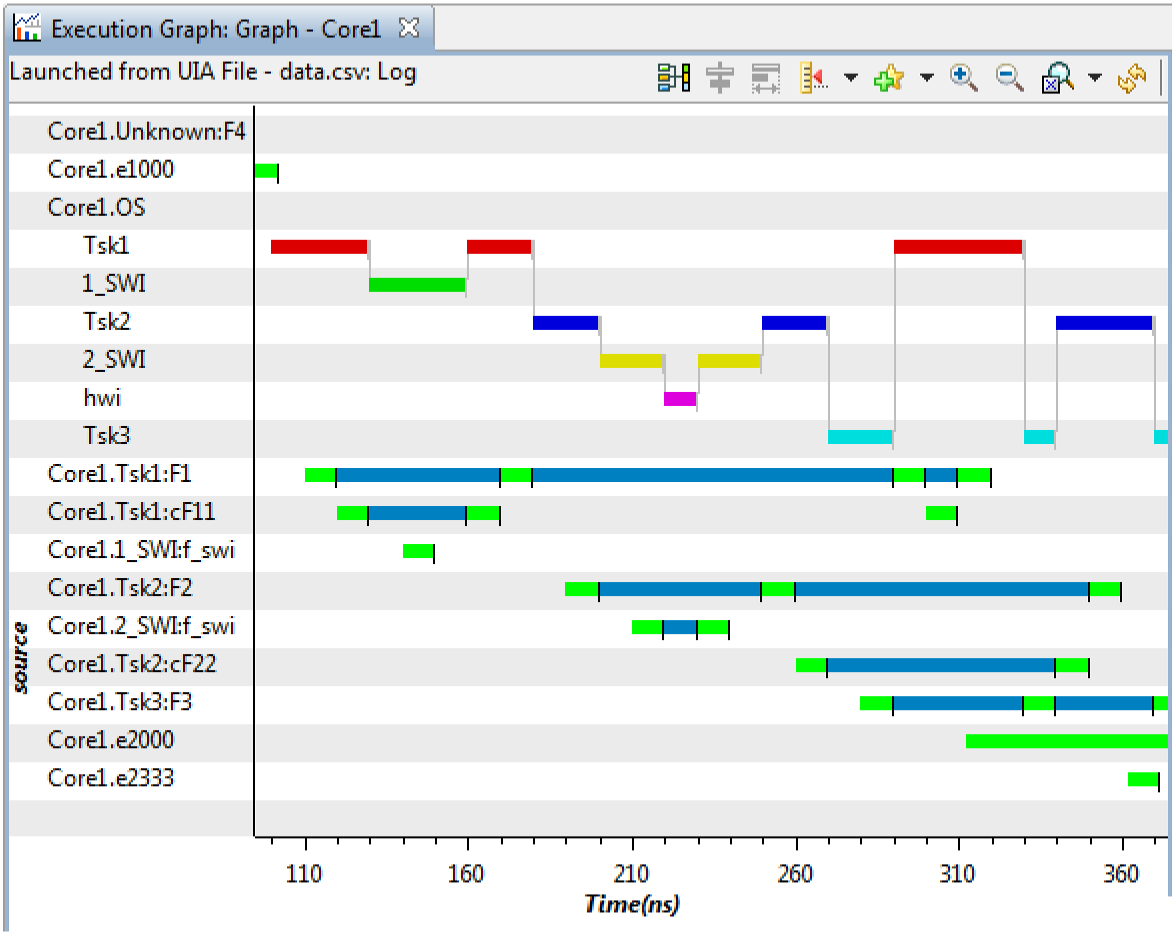
4.8.3 How Task Load Works with MCSA

The Task Load feature displays data provided automatically by internal SYS/BIOS calls to functions from the ti.sysbios.utils.Load module. SYS/BIOS threads are pre-instrumented to provide load data using a background thread.

See Section 5.2.1, *Enabling and Disabling Load Logging* for information about how to disable various types of load logging.

4.9 Using the Execution Graph with MCSA

The Execution Graph shows which thread is running at a given time. You can open this feature by choosing **Tools > Multicore System Analyzer > *session_name* > Execution Graph** from the CCS menu bar.



Sources (cores and threads) are listed in the left column. Click on a source to open an indented list of contexts for that source.

Click on a source named with the format <*core name*>.OS to open the list of threads on that core. A colored line for each item shows when that context is in control.

Source rows with the format <*core name*>.<*thread name*> show context switches in green and running threads in blue. You can click these rows to open them further to show their state. State transitions are indicated by a vertical black line across the colored bar.

Click the **Tree Mode** icon to switch to a hierarchical mode where you can expand and collapse context nodes.

Use the toolbar buttons to group (synchronize), measure, zoom, search, and filter the graph. You will likely need to zoom in a significant amount to see the execution transitions that interest you.

Right-click on the graph to adjust the display properties of the graph.

See Also

❑ Section 3.3, *Analyzing the Execution Sequence with MCSA*

4.9.1 How the Execution Graph Works with MCSA

The Execution Graph uses the same events as the Duration feature and the Context Aware Profile. The Execution Graph displays data about Task, Swi, and Hwi threads provided automatically by internal SYS/BIOS calls. SYS/BIOS threads are pre-instrumented to provide such data using a background thread.

Hwi and Swi can be expanded to list their threads separately only if you enable logging of events for the Hwi and Swi modules. Such logging is turned off by default for performance reasons. See Section 5.2.2, *Enabling and Disabling Event Logging* for how to turn on and off Hwi and Swi event logging.

If a data loss is detected, this graph displays the data loss using a narrow black bar and a message is shown at the bottom of the graph.

If data returned to the host out of sequence, this graph may have unpredictable behavior for state transitions beyond the visible range of the graph.

4.10 Using the Count Analysis Feature with MCSA

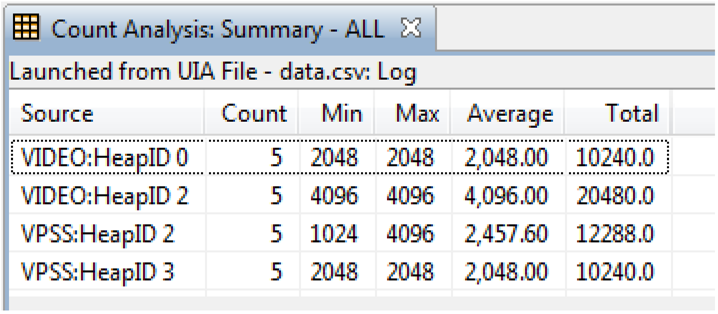
The Count Analysis feature provides statistics and visualization regarding a data value (32-bit signed) logged using a specific target-side UIA event (UIAEvt_intWithKey). For example, you might want to use Count Analysis to analyze how a data value from a peripheral changes over time. Or, you might want to find the maximum and minimum values reached by some variable or the number of times a variable is changed. The analysis is done on groups of log records with matching formatted strings that specify the source.

The Count Analysis feature displays data only if you modify your target code to include UIAEvt_intWithKey events as described in Section 4.10.3.

To open this feature, choose **Tools > Multicore System Analyzer >** *session_name* **> Count Analysis** from the CCS menu bar.

Summary View for Count Analysis

The Summary view is shown when you open the Count Analysis feature. This view shows the count, minimum, maximum, average, and total of the data values reported for each particular source.



Launched from UIA File - data.csv: Log

Source	Count	Min	Max	Average	Total
VIDEO:HeapID 0	5	2048	2048	2,048.00	10240.0
VIDEO:HeapID 2	5	4096	4096	4,096.00	20480.0
VPSS:HeapID 2	5	1024	4096	2,457.60	12288.0
VPSS:HeapID 3	5	2048	2048	2,048.00	10240.0

This view provides only one record for each unique source. The columns shown are as follows:

- **Source.** Statistics are performed on groups determined by combining the core name with a formatted string passed to the Log_writeX() call that created this record.
- **Count.** The number of instances of this source.
- **Min.** The minimum data value for this source.
- **Max.** The maximum data value for this source.
- **Average.** The average data value for this source.

- ❑ **Total.** The total data value for this source.

See page 4–19 for information about using the toolbar icons and right-click menu in the Summary view.

To open other views for the Count Analysis feature, use the **Analysis View** icon in the main CCS toolbar or right-click on the Count Analysis view and choose from the **Count Analysis views** submenu.

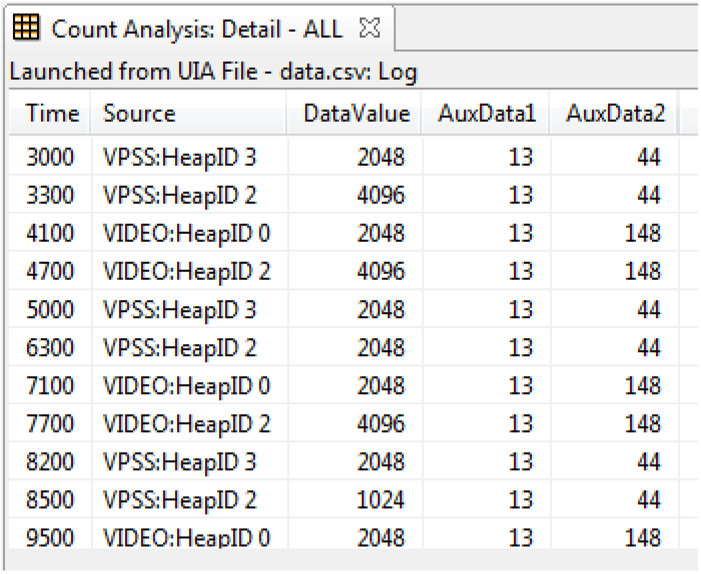
- ❑ The Detail view presents all log records for the UIAEvt_intWithKey event. See Section 4.10.1.
- ❑ The Graph view shows the change in the data value over time. See Section 4.10.2.

See Also

- ❑ Section 3.4, *Performing Count Analysis with MCSA*

4.10.1 Detail View for Count Analysis

To open the Detail view for this feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Count Analysis view and choose **Count Analysis views > Detail**.



Count Analysis: Detail - ALL

Launched from UIA File - data.csv: Log

Time	Source	DataValue	AuxData1	AuxData2
3000	VPSS:HeapID 3	2048	13	44
3300	VPSS:HeapID 2	4096	13	44
4100	VIDEO:HeapID 0	2048	13	148
4700	VIDEO:HeapID 2	4096	13	148
5000	VPSS:HeapID 3	2048	13	44
6300	VPSS:HeapID 2	2048	13	44
7100	VIDEO:HeapID 0	2048	13	148
7700	VIDEO:HeapID 2	4096	13	148
8200	VPSS:HeapID 3	2048	13	44
8500	VPSS:HeapID 2	1024	13	44
9500	VIDEO:HeapID 0	2048	13	148

Each record in the Detail view corresponds to a specific UIAEvt_intWithKey event logged on the target side.

You can export the records from the Count Analysis Detail view to a CSV file that can be used by a spreadsheet. To do this, right-click on the view and choose **Data > Export All**. You might do this in order to perform statistical analysis on the primary and auxiliary data values.

- ❑ **Time.** This column shows the correlated time at which this event occurred.
- ❑ **Source.** This column identifies the group for this event, which was determined by combining the core name with the resulting formatted string from the Log_writeX() call that created this record.
- ❑ **DataValue.** The value used for the analysis.
- ❑ **AuxData1.** These fields are used to pass auxiliary data that may need to be observed. This is the Arg2 field of the input to the AF.
- ❑ **AuxData2.** This is the Arg3 field of the input to the AF.

See page 4–19 for information about using the toolbar icons and right-click menu in the Detail view.

See Also

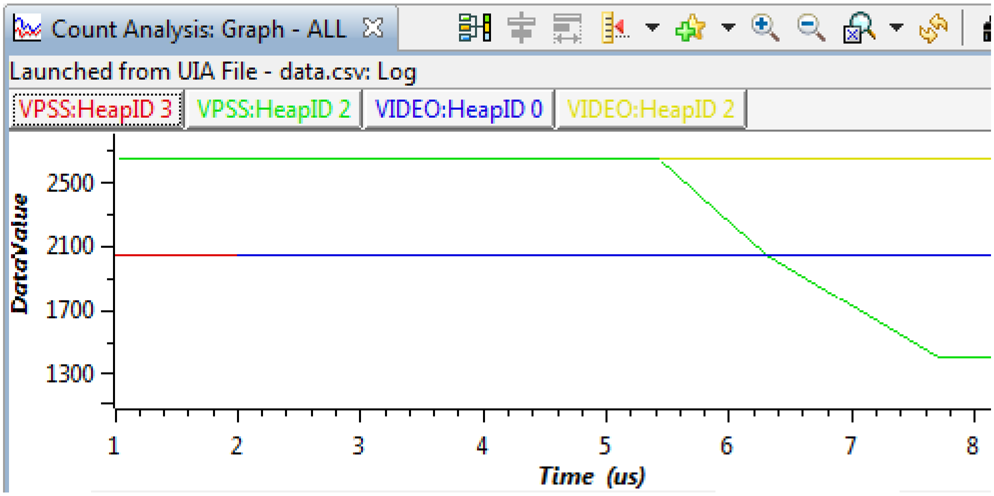
- ❑ Section 3.4, *Performing Count Analysis with MCSA*

4.10.2 Graph View for Count Analysis

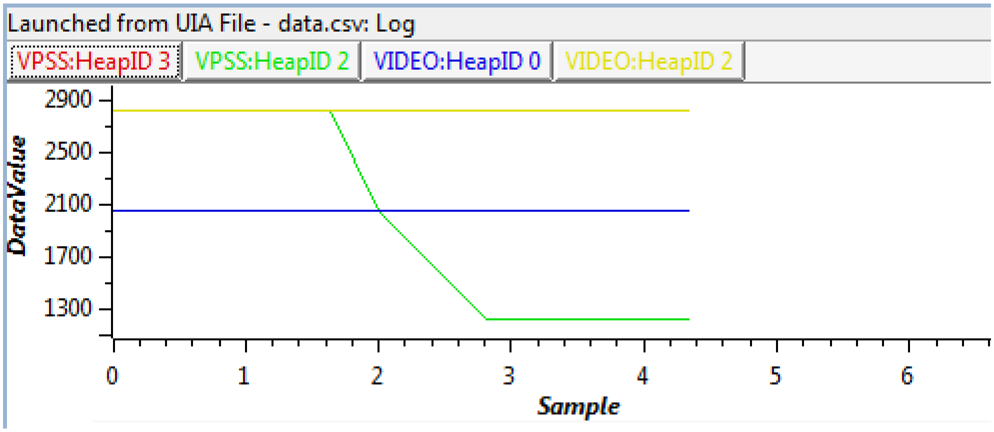
To open the Graph view for the Count Analysis feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Count Analysis view and choose **Count Analysis views > Graph**.

The Graph view shows changes in data values for each unique source. When you open this view, you can choose the core or master whose data values you want to plot (or all cores).

You can also choose whether to plot the data values against time or sample number. By default, data values are plotted vs. time.



In some cases, such as when the data values change at irregular intervals, you might want to plot the data values against the sample number. For example:



Clicking on the name of a measurement above the graph highlights the corresponding line in the graph. (If you do not see these buttons, right click on the graph and choose **Legend**.)

Use the toolbar buttons to group (synchronize), measure, zoom, search, and filter the graph. Right-click on the graph to adjust the display properties of the graph.

4.10.3 How Count Analysis Works with MCSA

Count analysis works with log events that use the UIAEvt_intWithKey event. This event is provided by the ti.uia.events.UIAEvt module. You must add these events to your target code in order to see data in the Count Analysis views.

The following call to Log_write6() logs an event that can be used by the Count Analysis views:

```
Log_write6( UIAEvt_intWithKey, 0x100, 44, 0,
    (IArg)"Component %s Instance=%d", (IArg)"CPU", 1);
```

The parameters for this call are as follows:

1) Use UIAEvt_intWithKey as the first parameter to log an event for Count Analysis.

2) The data value to be listed or plotted for this source. The value will be treated as a 32-bit integer. In the previous example, the data value is 0x100.

3) Additional data to be displayed in the AuxData1 column of the detail view. This value is not plotted in the graph. The value will be treated as a 32-bit integer. If you do not need to pass any auxiliary data here, pass a placeholder value such as 0. In the previous example, the auxData1 value is 44.

4) Additional data to be displayed in the AuxData2 column of the detail view. This value is not plotted in the graph. The value will be treated as a 32-bit integer. If you do not need to pass any auxiliary data here, pass a placeholder value such as 0. In the previous example, the auxData1 value is 0.

5) A string to be used as the source for this record. Statistics are performed on groups of records with matching sources in the Summary view. Groups of records with matching sources are plotted as a data series in the Graph view. This can be a formatted data string such as, "Component %s Instance=%d". Since the values passed after this string are "CPU" and 1, this record would belong to a group of events that shares a formatted data string of "Component CPU Instance=1"

6) Any variables to me used in the formatted data strong for the previous parameter should be added from the sixth parameter on.

4.11 Using the Duration Feature with MCSA

The Duration analysis feature provides information about the time between two points of execution on the target. These points must be instrumented by adding code that passes the `UIABenchmark_start` and `UIABenchmark_stop` events to calls to the Log_write1() function.

The Duration feature displays data only if you modify your target code to include UIABenchmark events as described in Section 4.11.3.

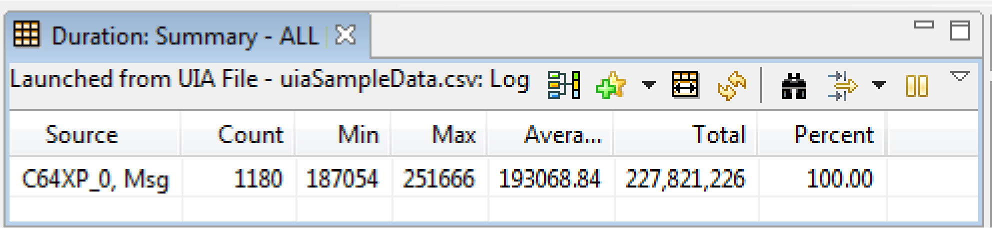
The Duration feature matches start and stop pairs for each "source". A source is identified by combining the core name and the arg1 argument passed to the Log_write1() function when the event argument is `UIABenchmark_start` or `UIABenchmark_stop`. For example, if the target program on CPU_5 makes the following calls, the source identifier will be "`CPU_5, running`".

```
Log_write1(UIABenchmark_start, (xdc_IArg)"running");
...
Log_write1(UIABenchmark_stop, (xdc_IArg)"running");
```

To open the Duration feature, choose **Tools > Multicore System Analyzer >** *session_name* **> Duration** from the main CCS menu bar.

Summary View for Duration Analysis

By default, the Summary view is shown when you open the Duration feature. This view shows the count, minimum, maximum, average, and total time measured between the start and stop times.



Duration: Summary - ALL

Launched from UIA File - uiaSampleData.csv: Log

Source	Count	Min	Max	Avera...	Total	Percent
C64XP_0, Msg	1180	187054	251666	193068.84	227,821,226	100.00

This view provides only one record for each unique source. The columns shown are as follows:

- **Source.** This column shows the identifier that the Duration feature uses to match up Start/Stop pairs.
- **Count.** The number of start/stop pairs that occurred for this source.
- **Min.** The minimum time in nanoseconds between start and stop for this source.
- **Max.** The maximum time in nanoseconds between start and stop for this source.
- **Average.** The average time in nanoseconds between start and stop for this source.
- **Total.** The total time in nanoseconds between all start/stop pairs for this source.
- **Percent.** The percent of the total time for all sources measured that was spent in this source.

See page 4–19 for information about using the toolbar icons and right-click menu in the Summary view.

To open Detail or Graph views for the Duration feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Duration view and choose from the **Duration views** submenu.

- The Detail view presents the raw start and stop times for each start/stop pair that has occurred. See Section 4.11.1.
- The Graph view shows the change in duration over time. See Section 4.11.2.

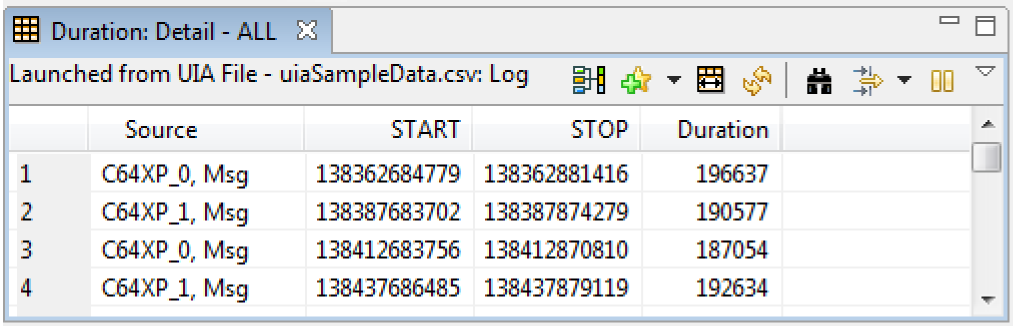
See Also

- Section 3.5, *Benchmarking with MCSA*

4.11.1 Detail View for Duration Analysis

To open the Detail view for the Duration feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Duration view and choose **Duration views > Detail**.

Each record in the Detail view corresponds to a pair of `UIABenchmark_start` or `UIABenchmark_stop` events passed to the Log_write1() function.



Duration: Detail - ALL

Launched from UIA File - uiaSampleData.csv: Log

	Source	START	STOP	Duration
1	C64XP_0, Msg	138362684779	138362881416	196637
2	C64XP_1, Msg	138387683702	138387874279	190577
3	C64XP_0, Msg	138412683756	138412870810	187054
4	C64XP_1, Msg	138437686485	138437879119	192634

There are likely to be multiple records in this view for the same source if the start/stop pairs are in threads that execute multiple times.

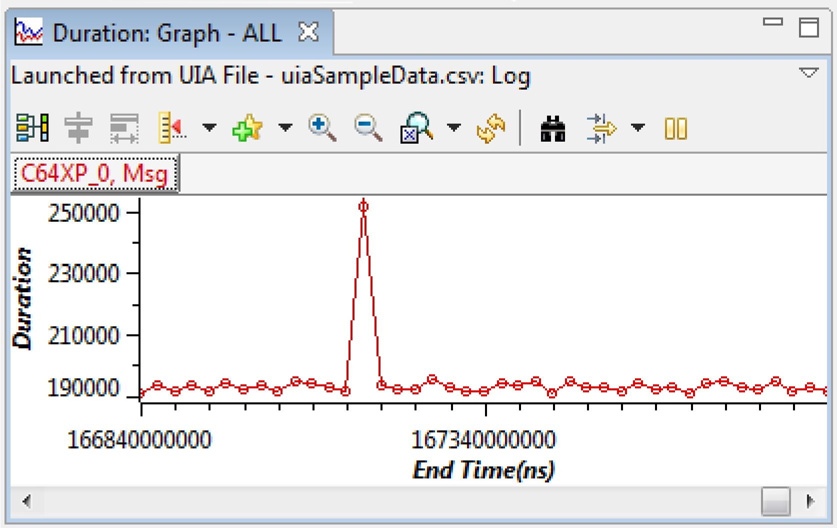
- ❑ **Source.** This column shows the identifier that the Duration feature uses to match up Start/Stop pairs. See Section 4.11 for details.
- ❑ **Start.** A timestamp for when the `UIABenchmark_start` event occurred.
- ❑ **Stop.** A timestamp for when the `UIABenchmark_stop` event occurred.
- ❑ **Duration.** The Stop - Start time.

See page 4–19 for information about using the toolbar icons and right-click menu in the Detail view.

4.11.2 Graph View for Duration Analysis

To open the Graph view for the Duration feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Duration view and choose **Duration views > Graph**.

The Graph view shows the change in duration with time for each unique source.


Duration: Graph - ALL

Launched from UIA File - uiaSampleData.csv: Log

Clicking on the name of a measurement above the graph highlights the corresponding line in the graph. (If you do not see these buttons, right click on the graph and choose **Legend**.)

Use the toolbar buttons to group (synchronize), measure, zoom, search, and filter the graph. Right-click on the graph to adjust the display properties of the graph.

4.11.3 How Duration Analysis Works with MCSA

The Duration feature matches pairs of `UIABenchmark_start` and `UIABenchmark_stop` events (from the ti.uia.events.UIABenchmark module) in target code for a given "source". These events are sent to the host via calls to Log_write1().

A source is identified by combining the core name and the arg1 argument passed to the Log_write1() function when the event argument is `UIABenchmark_start` or `UIABenchmark_stop`. For example, if the target program on CPU_5 makes the following calls, the source identifier will be "`CPU_5, process_1`".

```
#include <xdc/runtime/Log.h>
#include <ti/uia/events/UIABenchmark.h>
...

Log_write1(UIABenchmark_start, (xdc_IArg)"process_1");
...
Log_write1(UIABenchmark_stop, (xdc_IArg)"process_1");
```

The Log_write1() function comes from the XDCtools xdc.runtime.Log module. It is possible to use any of the Log_writeX() functions from Log_write1() to Log_write8(). If you use a Log_writeX() function with additional argument, all other arguments are ignored.

- ❑ The first parameter (`UIABenchmark_start` or `UIABenchmark_stop`) is an event of type Log_Event.
- ❑ The second parameter is a source name string cast as an argument.

See Section 5.4.2, *Enabling Event Output with the Diagnostics Mask* for information about how to enable and disable logging of UIABenchmark events.

The Duration feature handles missing Start or Stop events by ignoring events as needed.

- ❑ If a Start event is followed by another Start event for the same source, the second Start event is ignored and the first Start event is used.
- ❑ If a Stop event is followed by another Stop event for the same source, the second Stop event is ignored.
- ❑ If a Stop event occurs without a matching Start event for the same source, the Stop event is ignored.

Check the Error column in the Log view for a value that indicates a data loss occurred. See page 4–17 for details.

4.12 Using Context Aware Profile with MCSA

The Context Aware Profile feature calculates duration while considering context switches, interruptions, and execution of other functions.

The Context Aware Profile displays data only if you modify your target code to include UIABenchmark events as described in Section 4.12.3.

You can use this feature to see information about "inclusive time" vs. "exclusive time".

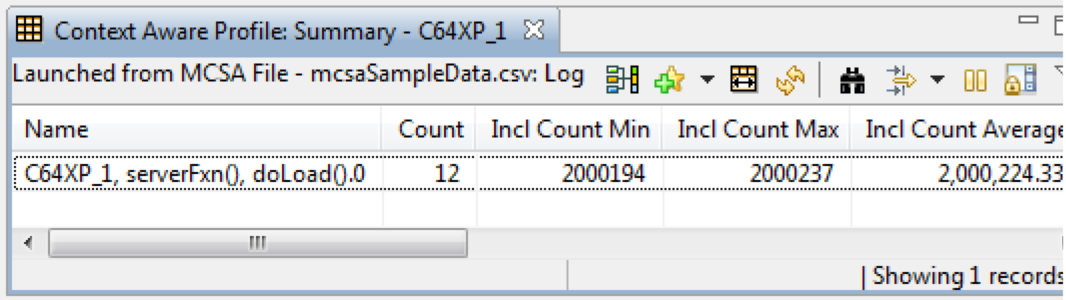
- **Inclusive time** is the entire time between a given pair of start times and stop times.
- **Exclusive time** is the inclusive time minus any time spent running any other thread context. Time spent in called functions and time spent running threads that preempt are yielded to by the thread being measured are not counted in exclusive time.

See Section 4.12.3 for details about how inclusive and exclusive time are calculated.

To open the Context Aware Profile, choose **Tools > Multicore System Analyzer >** *session_name* **> Context Aware Profile** from the CCS menu bar. Note that this feature does not allow you to select all cores; you must select a single core.

Summary View for Context Aware Profile

By default, the Summary view opens, which shows the minimum, maximum, average, and total number of nanoseconds within each thread for the selected core. These statistics are reported both for inclusive and exclusive time.



Context Aware Profile: Summary - C64XP_1

Launched from MCSA File - mcsaSampleData.csv: Log

Name	Count	Incl Count Min	Incl Count Max	Incl Count Average
C64XP_1, serverFxn(), doLoad().0	12	2000194	2000237	2,000,224.33

| Showing 1 records

The summary view shows statistics about each duration context that was measured. The statistics summarize multiple measurements made for each context. The columns in this view are as follows:

- **Name.** The name of the item for this row of statistics.
- **Count.** The number of start/stop pairs that measured this item’s duration.
- **Incl Count Min.** The minimum inclusive time measured.
- **Incl Count Max.** The maximum inclusive time measured.
- **Incl Count Average.** The average inclusive time measured.
- **Incl Count Total.** The total inclusive time measured.
- **Incl Count Percent.** The percent of all the inclusive times reported due to this item.
- **Excl Count Min.** The minimum exclusive time measured.
- **Excl Count Max.** The maximum exclusive time that was measured.
- **Excl Count Average.** The average exclusive time measured.
- **Excl Count Total.** The total exclusive time measured.
- **Excl Count Percent.** The percent of all the exclusive times reported due to this item.

To open Detail or Graph views, use the **Analysis View** icon in the main CCS toolbar or right-click on a Context Aware Profile view and choose from the **Context Aware Profile views** submenu.

- The Detail view presents the raw start and stop times for each start/stop pair measured. See Section 4.12.1.
- The Graph view shows the change in duration over time. See Section 4.12.2.

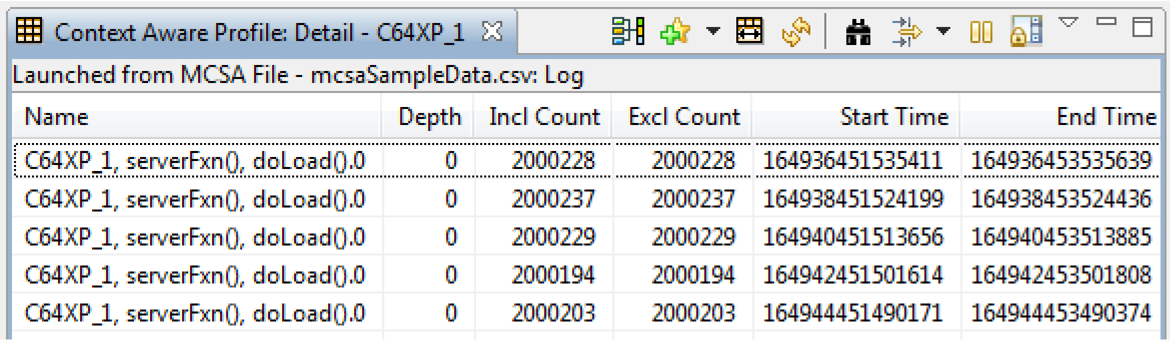
See Also

- Section 3.5, *Benchmarking with MCSA*

4.12.1 Detail View for Context Aware Profile

To open the Detail view for the Context Aware Profile feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Context Aware Profile view and choose **Context Aware Profile views > Detail**.

The detail view shows a record for each start/stop pair of durations recorded.



Context Aware Profile: Detail - C64XP_1

Launched from MCSA File - mcsaSampleData.csv: Log

Name	Depth	Incl Count	Excl Count	Start Time	End Time
C64XP_1, serverFxn(), doLoad().0	0	2000228	2000228	164936451535411	164936453535639
C64XP_1, serverFxn(), doLoad().0	0	2000237	2000237	164938451524199	164938453524436
C64XP_1, serverFxn(), doLoad().0	0	2000229	2000229	164940451513656	164940453513885
C64XP_1, serverFxn(), doLoad().0	0	2000194	2000194	164942451501614	164942453501808
C64XP_1, serverFxn(), doLoad().0	0	2000203	2000203	164944451490171	164944453490374

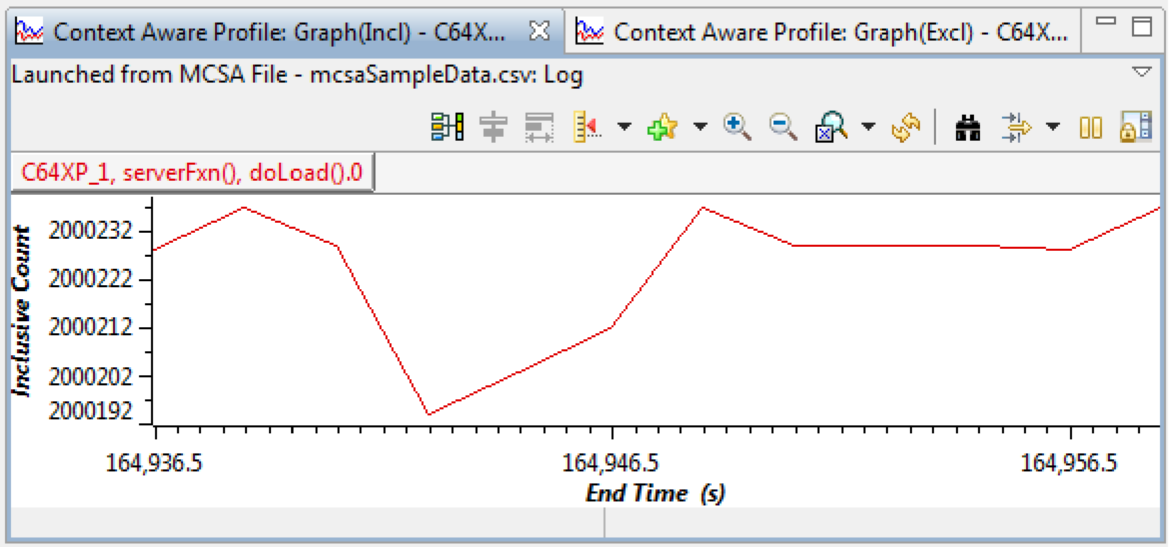
The columns in this view are as follows:

- **Name.** The name of the CPU combined with the function or thread that was measured.
- **Depth.** The number of levels deep for this function context. The top-level function has a depth of 0; functions called by the top-level have a depth of 1, and so on.
- **Incl Count.** The inclusive time for this measurement.
- **Excl Count.** The exclusive time for this measurement.
- **Start Time.** The time in nanoseconds when this measurement was started.
- **End Time.** The time in nanoseconds when this measurement was stopped.

4.12.2 Graph Views for Context Aware Profile

To open a Graph view for this feature, use the **Analysis View** icon in the main CCS toolbar or right-click on a Context Aware Profile view and choose **Context Aware Profile views > Graph(Excl)** or **Graph(Incl)**.

The Inclusive and Exclusive graph views for the Context Aware Profile show the change in duration for each context measured as a function of time. For example, you might use this to see if a thread takes longer to perform when the application has been running longer.


Context Aware Profile: Graph(Incl) - C64X...
Context Aware Profile: Graph(Excl) - C64X...
Launched from MCSA File - mcsaSampleData.csv: Log
C64XP_1, serverFxn(), doLoad().0
Inclusive Count
2000232
2000222
2000212
2000202
2000192
164,936.5
164,946.5
164,956.5
End Time (s)

Clicking on the name of a measurement above the graph highlights the corresponding line in the graph. (If you do not see these buttons, right click on the graph and choose **Legend**.)

Use the toolbar buttons to group (synchronize), measure, zoom, search, and filter the graph. Right-click on the graph to adjust the display properties of the graph.

4.12.3 How Context Aware Profiling Works with MCSA

The Context Aware Profile feature matches pairs of start and stop events from the ti.uia.events.UIABenchmark module. These events occur only if you add code to your target application that calls Log_write3() and passes the `UIABenchmark_startInstanceWithAdrs` and `UIABenchmark_stopInstanceWithAdrs` events as parameters.

For example, the following code would produce a start/stop pair that would be used by the Context Aware Profile for the myFunc() function:

```
#include <xdc/runtime/Log.h>
#include <ti/uia/events/UIABenchmark.h>

void myFunc(){
    Log_write3( UIABenchmark_startInstanceWithAdrs,
        (IArg)"Func: id=%x, Fxn=%x", 0, (UArg)&myFunc);
    ...
    Log_write3( UIABenchmark_stopInstanceWithAdrs,
        (IArg)"Func: id=%x, Fxn=%x", 0, (UArg)&myFunc);
    return;
};
```

To profile the entire time spent in the function, your code would use the UIABenchmark_startInstanceWithAdrs event at the beginning of the function and the UIABenchmark_stopInstanceWithAdrs event just prior to any line the could cause the function to return.

In the Log view the EventClass for these events is shown as "FUNC" because a function reference is passed with the event to identify the function that is being profiled.

The Log_write3() function comes from the XDCtools xdc.runtime.Log module. It is possible to use any of the Log_writeX() functions from Log_write3() to Log_write8(). If you use a Log_writeX() function with additional arguments, all other arguments are ignored by the Context Aware Profile feature. The parameters passed to Log_write3() are as follows:

- **evt.** An event (`UIABenchmark_startInstanceWithAdrs` or `UIABenchmark_stopInstanceWithAdrs`) of type Log_Event.
- **arg0.** A message string that must have the following format: `"Func: id=%x, Fxn=%x"`
- **arg1.** Could be used in the future to specify the instance of the function, but the Context Aware Profile currently expects a value of 0.
- **arg2.** A function reference to identify what this start/stop pair is profiling.

See Section 5.4.2, *Enabling Event Output with the Diagnostics Mask* for information about how to enable and disable logging of UIABenchmark events. See Section 5.4.3, *Events Provided by UIA* for more about UIABenchmark events.

The Context Aware Profile also uses context switching information about Task, Swi, and Hwi threads to calculate the inclusive and exclusive time between a start/stop pair. The following table shows whether various types of contexts are included in inclusive and exclusive time. Since the Duration views (page 4–38) are not context-aware, time spent in any context is included in those views.

Table 4–3 Inclusive vs. Exclusive Time

Context or Function Type	**Counted for Inclusive Time**	**Counted for Exclusive Time**	**Counted for Duration**
Time spent in the specified function’s context.	Yes	Yes	Yes
Time spent in functions called from the specified context. For example, you might want to benchmark function A(), which calls functions B() and C().	Yes	No	Yes
Time spent in other Task functions as a result of preemption, yielding, and pend/post actions.	Yes	No	Yes
Time spent in Hwi or Swi thread contexts.	No	No	Yes

The Context Aware Profile feature handles missing Start or Stop events by ignoring events as needed.

- If a Start event is followed by another Start event for the same source, the second Start event is ignored and the first Start event is used.
- If a Stop event is followed by another Stop event for the same source, the second Stop event is ignored.
- If a Stop event occurs without a matching Start event for the same source, the Stop event is ignored.

Check the Error column in the Log view for a value that indicates a data loss occurred. See page 4–17 for details.

Chapter 5

UIA Configuration and Coding on the Target

This chapter describes how to configure and code target applications using UIA modules.

Topic **Page**

5.1 Quickly Enabling UIA Instrumentation

You can begin analyzing data provided by UIA by enabling data collection from pre-instrumented SYS/BIOS threads. Later, you can add target-side code to collect additional data specific to your application.

Once you perform the necessary configuration, you will be able to view UIA data in the Log view, CPU Load, Task Load, and Execution Graph features. Only the Context Aware Profile and Duration features display no data unless you modify your target code by adding benchmarking calls as described in Section 4.11.3, *How Duration Analysis Works with MCSA* and Section 4.12.3, *How Context Aware Profiling Works with MCSA*.

In order to enable data collection from pre-instrumented SYS/BIOS threads and have that data transferred from the target(s) to the host PC running CCS, you must do the following:

Configuration Steps to Perform on All Targets

1) **Remove Legacy Modules.** Remove any statements in your application's configuration file (*.cfg) that include and configure the following modules:

   - ti.sysbios.rta.Agent
   - xdc.runtime.LoggerBuf
   - ti.rtdx.RtdxModule
   - ti.rtdx.driver.RtdxDvr

   If you have logger instances for the xdc.runtime.LoggerBuf, delete those instances.

2) **Use the LoggingSetup Module.** Add the following statement to include UIA's LoggingSetup module in your application's configuration. For example:

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

   Including the LoggingSetup module creates logger instances needed by UIA and assigns those loggers to the modules that need them in order to provide UIA data.

3) If you intend to use some method other than JTAG stop-mode to upload events to the host, set the LoggingSetup.eventUploadMode parameter as described in *Configuring the Event Upload Mode*, page 5-11. For example, you can use a non-JTAG transport such as Ethernet or File, or a JTAG-dependent transport such as simulator or JTAG run-mode. For example:

   ```
   LoggingSetup.eventUploadMode = Upload_NONJTAGTRANSPORT;
   ```

4) **Configure Physical Communication (such as NDK).** You must also configure physical communication between the cores. The application is responsible for configuring and starting the physical communication. For example, this communication may use the NDK. See the target-specific examples provided with UIA (and NDK) for sample code.

Configuration Steps to Perform on Multi-Core Targets Only

1) **Configure the Topology.** If your multi-core application routes data through a single master core, edit your application's configuration file to include UIA's ServiceMgr module and configure MULTICORE as the topology, and identify the master core using the ServiceMgr.masterProcId parameter. For example:

```
var ServiceMgr =
    xdc.useModule('ti.uia.runtime.ServiceMgr');
ServiceMgr.topology = ServiceMgr.Topology_MULTICORE;
ServiceMgr.masterProcId = 3;
```

   The EVMTI816x routes to the ARM, which runs Linux. The EVM6472 routes to the master core. In general, if only one core can access the peripherals, use the MULTICORE topology.

   If each core in your multi-core application sends data directly to CCS on the host, configure the topology as Topology_SINGLECORE (which is the default).

   See Section 5.3.3 for more information about configuring the topology.

2) **Configure IPC.** You must also configure and initialize IPC and any other components needed to enable communication between the cores. For example, you might also need to set up the NDK. See the target-specific examples provided with UIA (and IPC) for sample code. UIA may not be the only user of these resources, so it is left to the application to configure and initialize them.

3) **Configure GlobalTimestampProxy and CpuTimestampProxy.** You must configure the GlobalTimestampProxy parameter in the LogSync module as described in Section 5.3.6. If the frequency of your local CPU will change at run-time, you must also configure the CpuTimestampProxy parameter.

5.1.1 Using XGCONF to Enable UIA Instrumentation

Instead of editing configuration scripts directly, you can use the XGCONF tool within CCS to visually edit an application's configuration. XGCONF shows the RTSC modules—including XDCtools, SYS/BIOS, IPC, and UIA—that are available for configuration.

XGCONF lets you add the use of modules to your application, create instances, and set parameter values. It performs error checking as you work, and so can save you time by preventing you from making configuration errors that would otherwise not be detected until you built the application.

For example, to add UIA instrumentation to a SYS/BIOS application that uses the legacy ti.sysbios.rta.Agent and xdc.runtime.LoggerBuf modules, follow these steps:

1) In CCS, right-click on the project and choose **Show Build Settings**.

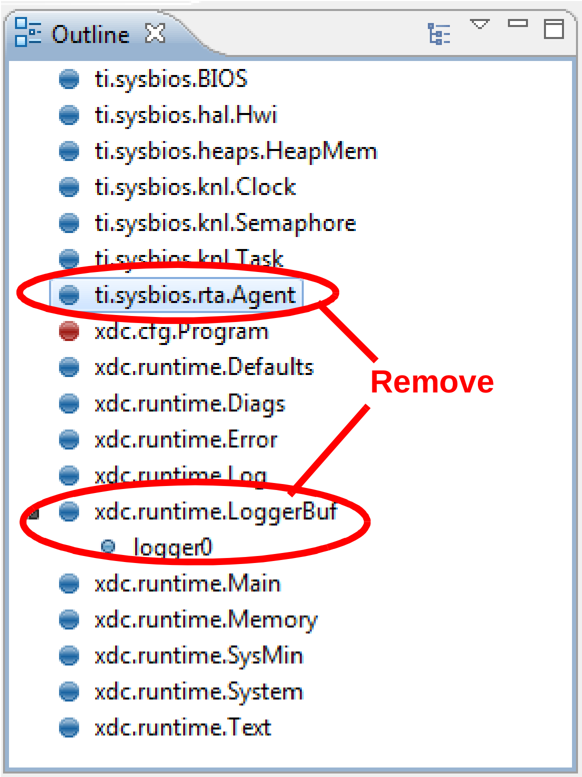
2) In the Properties dialog, choose the **CCS Build** category, then the **RTSC** tab.

3) In the **Products and Repositories** area, check the box next to UIA and select the most recent version. This causes you application to link with the necessary parts of UIA and makes the UIA modules available within XGCONF.

4) Click **OK**.

5) Open the project's configuration file (*.cfg) with XGCONF in CCS. Notice that UIA is now listed in the Available Packages list under Other Repositories. You can expand the UIA package to look at the modules that are available.

6) If you see the ti.sysbios.rta.Agent module listed in the outline, right-click on it and choose **Stop Using Agent**.



Outline

- ti.sysbios.BIOS
- ti.sysbios.hal.Hwi
- ti.sysbios.heaps.HeapMem
- ti.sysbios.knl.Clock
- ti.sysbios.knl.Semaphore
- ti.sysbios.knl.Task
- ti.sysbios.rta.Agent
- xdc.cfg.Program
- xdc.runtime.Defaults
- xdc.runtime.Diags
- xdc.runtime.Error
- xdc.runtime.Log
- xdc.runtime.LoggerBuf
  - logger0
- xdc.runtime.Main
- xdc.runtime.Memory
- xdc.runtime.SysMin
- xdc.runtime.System
- xdc.runtime.Text

7) If a LoggerBuf logger is listed as shown above, select the right-click on the logger instance and choose **Delete**. If you see error messages, select the **Source** tab at the bottom of the center pane. Delete all statements related to the logger instance, and save the file.

8) If the xdc.runtime.LoggerBuf module is listed in the outline, right-click on it and choose **Stop Using LoggerBuf**.

9) If there are any RTDX modules or drivers, remove those from the outline.

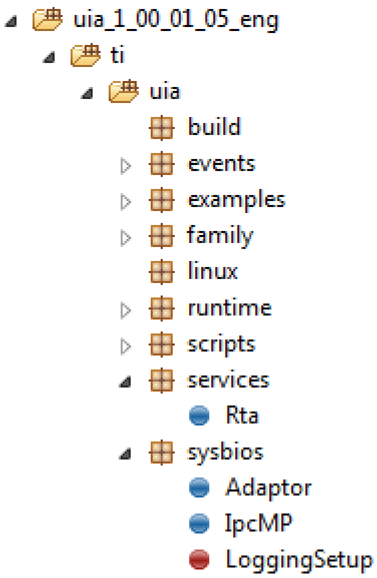
10) Expand the UIA package to find the ti.uia.sysbios.LoggingSetup module and drag it to the Outline.

11) Select the LoggingSetup module in the Outline. Notice that the properties for this module are shown in the center pane. If you see the configuration script instead, click the Properties tab at the bottom of this area.

12) Set a property. For example, you can enable event logging for individual Swi threads by setting sysbiosSwiLogging to false.

13) Set other properties and add other modules as needed.

14) Press Ctrl+S to save your configuration file.



5.2 Configuring SYS/BIOS Logging

You can configure the types of SYS/BIOS events that are logged and sent to MCSA.

- **Load logging** is enabled by default for CPU, Task, Swi, and Hwi threads. As a result, information about loads for those items is available in the CPU Load and Task Load features.
- **Event logging** used to display the Execution Graph is enabled by default only for Task threads. You can enable it for Swi and Hwi threads by configuring the LoggingSetup module.

See Section 5.4.2 for information about configuring other types of logging messages.

5.2.1 Enabling and Disabling Load Logging

By default, all types of SYS/BIOS load logging are enabled as a result of adding the LoggingSetup module to the configuration.

If you want to disable CPU Load logging, you would include the following statement in your target application’s configuration file. However, note that disabling CPU load logging also disables all other load logging.

```
LoggingSetup.loadLogging = false;
```

To disable Task, Swi, or Hwi load logging, you can use the corresponding statement from the following list:

```
var Load = xdc.useModule('ti.sysbios.utils.Load');
Load.taskEnabled = false;
Load.swiEnabled = false;
Load.hwiEnabled = false;
```

Another way to disable load logging is to modify the setting of the Load.common$.diags_USER4 mask, which controls whether load logging is output. For example, the following statements disable all load logging:

```
var Load = xdc.useModule('ti.sysbios.utils.Load');
var Diags = xdc.useModule('xdc.runtime.Diags');
Load.common$.diags_USER4 = Diags.ALWAYS_OFF;
```

The Load.common$.diags_USER4 mask is set to Diags.RUNTIME_ON by the LoggingSetup module unless you have explicitly set it to some other value.

5.2.2 Enabling and Disabling Event Logging

By default, the event logging used to display the Execution Graph is enabled by default only for SYS/BIOS Task threads. As a result, the Execution Graph can be expanded to show individual Task threads, but shows all Swi thread execution as one row, and all Hwi thread execution in another row without showing Swi and Hwi thread names.

Enabling Logging

You can enable event logging for SYS/BIOS Swi and Hwi threads by configuring the LoggingSetup module as follows:

```
LoggingSetup.sysbiosSwiLogging = true;
LoggingSetup.sysbiosHwiLogging = true;
```

Enabling event logging for Swi and Hwi allows you to see the execution status of individual Swi and Hwi threads. Application performance may be impacted if you enable such logging for applications with Swi or Hwi functions that run frequently. In addition, logging many frequent events increases the chance of dropped events.

For Task threads, the events logged are ready, block, switch, yield, sleep, set priority, and exit events. For Swi threads, the events logged are post, begin, and end events. For Hwi threads, the events logged are begin and end events.

The following configuration statements enable logging of all function entry and exit events by your application. This is because your main() function and other user-defined functions (that is, for example, all non-XDCtools, non-SYS/BIOS, non-IPC, and non-UIA modules) inherit their default Diags configuration from the Main module's Diags configuration.

```
Main.common$.diags_ENTRY = Diags.ALWAYS_ON;
Main.common$.diags_EXIT = Diags.ALWAYS_ON;
```

Disabling Logging

To disable Task, Swi, Hwi, or Main event logging, you can use the appropriate statement from the following list:

```
LoggingSetup.sysbiosTaskLogging = false;
LoggingSetup.sysbiosSwiLogging = false;
LoggingSetup.sysbiosHwiLogging = false;
LoggingSetup.mainLogging = false;
```

Another way to disable event logging is to modify the setting of the common$.diags_USER1 and common$.diags_USER2 masks for the appropriate module. This controls whether event logging is output. For example, the following statements disable all event logging:

```
var Task = xdc.useModule('ti.sysbios.knl.Task');
Task.common$.diags_USER1 = Diags.ALWAYS_OFF;
Task.common$.diags_USER2 = Diags.ALWAYS_OFF;

var Swi = xdc.useModule('ti.sysbios.knl.Swi');
Swi.common$.diags_USER1 = Diags.ALWAYS_OFF;
Swi.common$.diags_USER2 = Diags.ALWAYS_OFF;

var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
Hwi.common$.diags_USER1 = Diags.ALWAYS_OFF;
Hwi.common$.diags_USER2 = Diags.ALWAYS_OFF;

Main.common$.diags_USER1 = Diags.ALWAYS_OFF;
Main.common$.diags_USER2 = Diags.ALWAYS_OFF;
```

5.2.3 More About Diags Masks

Since logging is not always desired because of the potential impact on the system performance, you can use the xdc.runtime.Diags module to enable/disable logging both statically and dynamically on a global or per module basis.

By default the ti.uia.sysbios.LoggingSetup module sets the following diagnostics masks to Diags.RUNTIME_ON:

- **diags_USER1 and diags_USER2:** Main, Task, Semaphore, and Event modules. These masks control event logging.
- **diags_USER4:** Main and Load modules. This mask controls load logging.
- **diags_USER3, diags_USER5, and diags_USER6:** Main module.
- **diags_STATUS:** Main module. This mask controls the output of some events provided in the ti.uia.events package.
- **diags_ANALYSIS:** Main module. This mask controls the output of some events provided in the ti.uia.events package.
- **diags_INFO:** Main module. This mask controls the output of some events provided in the ti.uia.events package.

For Swi and Hwi event logging, the diags_USER1 and diags_USER2 masks are set to Diags.RUNTIME_ON only if you have set LoggingSetup.sysbiosSwiLogging or LoggingSetup.sysbiosHwiLogging to true. By default, these are off.

This leaves other masks that are rarely or never used by UIA—diags_ENTRY, diags_EXIT, diags_LIFECYCLE, diags_INTERNAL, diags_ASSERT, diags_USER7, and diags_USER8—at their default values.

The XDCscript portion of the CDOC online reference contains details about which diagnostics masks must be enabled for particular events to be logged.

Note: You should be careful about setting any Defaults.common$ parameters. Such parameter settings are inherited by *all* modules for which the parameter is not explicitly set. This includes all XDCtools, SYS/BIOS, IPC, and UIA modules.

5.2.4 Setting Diags Masks at Runtime

Runtime checking is performed when a diagnostics mask is set to RUNTIME_ON. To improve performance by removing runtime checking, you may want to change the configuration to use Diags.ALWAYS_ON or Diags.ALWAYS_OFF.

If you configure a diagnostics mask to be set to Diags.RUNTIME_ON or Diags.RUNTIME_OFF, your C code can change the setting at runtime by calling the Diags_setMask() function. For example:

```
// turn on USER1 & USER2 events in the Swi module
Diags_setMask("ti.sysbios.knl.Swi+1");
Diags_setMask("ti.sysbios.knl.Swi+2");
...
// turn off USER4 (load) events in the Swi module
Diags_setMask("ti.sysbios.knl.Swi-4");
```

For information about the tradeoffs between Diags.ALWAYS_ON and Diags.RUNTIME_ON, see Section 7.5.2 and its subsections in the *SYS/BIOS User’s Guide* (SPRUEX3). Ignore any mention of the ti.sysbios.rta.Agent module and RTDX; these are replaced by the modules provided with UIA.

See Section 5.4.2 for more about runtime diagnostics configuration.

5.3 Customizing the Configuration of UIA Modules

You can further customize the behavior of UIA modules as described in the subsections that follow:

❑ Section 5.3.1, *Configuring ti.uia.sysbios.LoggingSetup*

❑ Section 5.3.2, *Configuring ti.uia.services.Rta*

❑ Section 5.3.3, *Configuring ti.uia.runtime.ServiceMgr*

❑ Section 5.3.4, *Configuring ti.uia.runtime.LoggerCircBuf*

❑ Section 5.3.5, *Configuring ti.uia.runtime.LoggerSM*

❑ Section 5.3.6, *Configuring ti.uia.runtime.LogSync*

You can further control UIA behavior through the following:

❑ Section 5.3.7, *Configuring IPC*

5.3.1 Configuring ti.uia.sysbios.LoggingSetup

In order to enable UIA instrumentation, your application's configuration file should include the ti.uia.sysbios.LoggingSetup module as follows:

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

See Section 5.2 for how to configure the types of events that are logged and sent to MCSA.

Besides using the LoggingSetup module to configure event logging, you can also configure the loggers that are created as a result of including this module.

Configuring the Event Upload Mode

By default, events are uploaded using the JTAG connection in stop-mode. Events are uploaded over JTAG when the target halts. This mode requires that JTAG connections are supported by the target.

If you want to use Ethernet, probe points, a simulator, or some other method for uploading events, you can specify one of those upload methods by configuring the LoggingSetup.eventUploadMode parameter. For example, you could use the following if you were running a simulator:

```
var LoggingSetup =
    xdc.useModule('ti.uia.sysbios.LoggingSetup');
LoggingSetup.eventUploadMode = Upload_SIMULATOR;
```

The available upload modes are as follows:

Table 5–1 LoggingSetup.eventUploadMode Values

Value	Description
Upload_SIMULATOR	Events are written to LoggerProbePoint loggers and are uploaded from a simulator at the time the event is logged. This mode is not supported on CPUs running multi-process operating systems such as Linux.
Upload_PROBEPOINT	Events are written to LoggerProbePoint loggers and are uploaded at the time an event is logged. The target is briefly halted when an event is uploaded. This mode is not supported on CPUs running multi-process operating systems such as Linux.
Upload_JTAGSTOPMODE	Events are uploaded over JTAG when the target halts. This mode is not supported on CPUs running multi-process operating systems such as Linux. (This is the default mode.)
Upload_JTAGRUNMODE	Events are uploaded directly from the circular buffers via JTAG while the target is running. Note that while events can be uploaded via JTAG, commands cannot be sent to the target via JTAG. This mode is currently supported only on C6x devices.
Upload_NONJTAGTRANSPORT	Events are uploaded over the non-JTAG transport specified by the ServiceMgr.transportType parameter. For example, by Ethernet or File. See Section 5.3.3.

The various event upload modes have different pros and cons. The following table compares the various modes. Comments below the table explain the columns in more detail.

Table 5–2 Comparison of Upload Event Modes

Mode	**Target and Connection Requirements**	**Memory Footprint**	**Performance Impact of Log Records**	**Ease-of-Use**	**Good for Real Hardware or Multi-Core**	**Can Records Be Dropped?**
Simulator	JTAG required	smaller	none	easy	No	No
Probe Point	JTAG required	smaller	none	easy	No	No
JTAG Stop-Mode (default)	JTAG required	smaller	none	easy	No	Can be overwritten
JTAG Run-Mode	JTAG required C6x family only	smaller	slight	easy	Yes	Yes, if CCS cannot keep up
Non-JTAG Transport	no JTAG required; various transports supported	larger	some	more complex	Yes	Yes, if Rta cannot keep up. Also records not seen when target halts.

The only mode that does not require a JTAG connection is the Upload_NONJTAGTRANSPORT mode. So, for targets that do not support JTAG connections, this is the only event upload mode supported.

If you select a non-JTAG mode, you should also select a topology (single-core or multi-core) and a transport type (Ethernet, File, or user-defined). See Section 5.3.3.1, *Configuring the topology* and Section 5.3.3.2, *Configuring the transportType* for details.

The memory footprint is smaller for modes that do not use the ServiceMgr framework.

The non-JTAG modes have a small performance impact on the application because Log records are retrieved from a low-priority Task thread by the ServiceMgr framework.

The modes listed as being easy to use are easy because there are very few decisions to be made. The non-JTAG modes allow you to customize the behavior of the ServiceMgr framework, so there are more choices available to you.

Only JTAG Run-Mode and Non-JTAG mode should be used for applications with real-time constraints running on real (non-simulator) hardware. The other modes do not handle these cases well because halting the target when a record is written might interfere with real-time interactions between cores and the real-time behavior of the application.

For JTAG Stop-Mode, records may be overwritten if the Logger buffer is too small. For JTAG Run-Mode, records may be dropped if the CCS IDE cannot keep up with the number of event logs received. For the non-JTAG modes, records may be dropped if the Rta module cannot keep up with the number of events; see Section 3.6.3, *If MCSA Events are Being Dropped* to troubleshoot this problem. For the Non-JTAG mode, if you halt the target, Log events are not sent to the host; they remain on the target waiting for the application to resume running.

Default Logger Instances

The LoggingSetup module creates logger instances for the following purposes:

❑ **SYSBIOS System Logger.** Receives events related to SYS/BIOS context-switching. For example, pend and post events for the Semaphore and Event modules go to this log. The default size of this logger is 32768 MADUs.

❑ **Load Logger.** Receives load information for the CPU, thread types (Hwi and Swi), and Task functions. The default size of this logger is 1024 MADUs.

❑ **Main Logger.** Receives benchmarking information from pre-instrumented objects and any events you add to the target code. The default size of this logger is 32768 MADUs.

Loggers are responsible for handling events sent via APIs in the xdc.runtime.Log module and the Log extension modules provided by UIA (for example, LogSnapshot and LogSync).

You can use the default configuration provided by LoggingSetup simply by adding the `useModule` statement, which was shown at the beginning of this section.

Configuring Custom Loggers

See the ti.uia.sysbios.LoggingSetup topic in the online help API Reference (CDOC) for more information about this module.

Configuring Logger Buffer Sizes: Logger implementations drop events if the buffer fills up before the events are collected by the Rta module. For this reason, it is a good idea to make sure that the logger is large enough to hold the events that it will receive during each period. Events generally range in size from 8 bytes to 48 bytes. By default, Rta polls loggers every

100 milliseconds. Depending on the period and number of events being logged, you may want to change the size of the loggers. You can change the default sizes of the three loggers created by LoggingSetup as follows:

```
LoggingSetup.loadLoggerSize = 2048;
LoggingSetup.mainLoggerSize = 16384;
LoggingSetup.sysbiosLoggerSize = 16384;
```

Configuring Your Own Loggers: Loggers are implementations of an interface, ILogger, which is defined by XDCtools. By default, the loggers created by LoggingSetup use the LoggerCircBuf implementation provided with UIA.

See Section 5.3.4, *Configuring ti.uia.runtime.LoggerCircBuf*, for an example that configures a custom logger before including the LoggingSetup module. The example customizes the logger size and memory section.

Note: You can only use loggers that inherit from ti.uia.runtime.IUIATransfer with UIA. Currently only the ti.uia.runtime.LoggerCircBuf and ti.uia.runtime.LoggerSMti.uia.runtime.LoggerSM implementations are supported. You cannot use xdc.runtime.LoggerBuf with UIA.

If you have critical events that you want to instrument or want to be able to filter the events in CCS based on the logger to which they were sent, you can create additional loggers to be used by Log calls in your code. See Section 5.3.4, *Configuring ti.uia.runtime.LoggerCircBuf* for details.

5.3.2 Configuring ti.uia.services.Rta

For non-JTAG event upload modes, UIA uses the ti.uia.services.Rta module to provide a real-time analysis service. The Rta module enables a service that collects events from logger instances and sends them to the host.

Your application’s configuration file does not need to include the ti.uia.services.Rta module, because it is automatically included when you set the LoggingSetup.eventUploadMode parameter to UploadMode_NONJTAGTRANSPORT.

Note: You should not include the ti.uia.services.Rta module in your configuration file or set any of its parameters if you are using an eventUploadMode other than UploadMode_NONJTAGTRANSPORT.

By default, the Rta module collects events every 100 milliseconds. You can configure a different interval as in the following example:

```
Rta.periodInMs = 500;
```

You should shorten the period if you are using a simulator. For example:

```
Rta.periodInMs = 5;
```

Setting the periodInMs parameter does not guarantee that the collection will run at this rate. Even if the period has expired, the collection will not occur until the current running Task has yielded and there are no other higher priority Tasks ready.

Setting the period to 0 disables all collection of events.

When you include the Rta module, Rta automatically includes the ti.uia.runtime.ServiceMgr module—the module that actually communicates with the instrumentation host. The ServiceMgr module is described in Section 5.3.3.

A periodInMs parameter is also provided by the ServiceMgr module. When setting the Rta.periodInMs parameter, you should consider the interactions between the settings you use for the SYS/BIOS clock interval (in the ti.sysbios.knl.Clock module), the ServiceMgr.periodInMs parameter, and the Rta.periodInMs parameter.

❑ The SYS/BIOS clock interval should be the shortest interval of the three. By default it is 1 millisecond.

❑ The ServiceMgr.periodInMs parameter should be larger than the SYS/BIOS clock interval, and it should be a whole-number multiple of the SYS/BIOS clock interval. By default it is 100 milliseconds.

❑ The Rta.periodInMs parameter should be equal to or greater than the ServiceMgr.periodInMs parameter, and it should also be a whole-number multiple of ServiceMgr.periodInMs. By default it is 100 milliseconds.

In summary:

SYS/BIOS clock interval < ServiceMgr.periodInMs <= Rta.periodInMs

If periodInMs for ServiceMgr and Rta are too small, your system performance may suffer because of all the context switches. If periodInMs is too large, logger buffers may fill up before the period elapses and you may lose data.

See the ti.uia.services.Rta topic in the online help API Reference (CDOC) for more information about this module.

5.3.3 Configuring ti.uia.runtime.ServiceMgr

The ti.uia.runtime.ServiceMgr module is responsible for sending and receiving packets between the services on the target and the instrumentation host.

When the LoggingSetup module includes the Rta module (because the LoggingSetup.eventUploadMode is UploadMode_NONJTAGTRANSPORT), Rta automatically includes the ti.uia.runtime.ServiceMgr module. If you have a single-core application, you can use the default configuration of the ServiceMgr module.

The ServiceMgr module provides three key configuration parameters in setting up UIA for your device based on your architecture:

- ❑ **topology.** Specifies whether you are using a single-core or multi-core target. See Section 5.3.3.1.
- ❑ **transportType.** Specifies transport to use. See Section 5.3.3.2.
- ❑ **masterProcId.** If this is a multi-core application, specifies which core is routing events to the instrumentation host. See Section 5.3.3.3.

5.3.3.1 Configuring the topology

The default for the ServiceMgr.topology configuration parameter is Topology_SINGLECORE, which means that each core on the device communicates directly with the host.

If you have a multi-core application and the routing of events to CCS is done via a single master core (which then sends the data to CCS), you must include the ServiceMgr module explicitly and configure MULTICORE as the topology. For example:

```
var ServiceMgr =
    xdc.useModule('ti.uia.runtime.ServiceMgr');
ServiceMgr.topology = ServiceMgr.Topology_MULTICORE;
```

The EVMTI816x routes to the ARM, which runs Linux. The EVM6472 routes to the master core. In general, if only one core can access the peripherals, use the MULTICORE topology.

Communication with other cores is routed via the master core, which is specified by the ServiceMgr.masterProcId parameter.

Routing between cores is done via Ipc's MessageQ module. ServiceMgr uses IPC to discover the core configuration and to communicate between the cores. The cores use MessageQ to talk to each other. The masterProcId communicates to CCS. For ’C6472, the master core uses NDK to send and receive TCP/UDP packets to and from CCS.

Note: ServiceMgr expects the application to configure and initialize IPC.

If each core in your multi-core application sends data directly to CCS on the host, configure the topology as Topology_SINGLECORE and do not specify a value for the masterProcId parameter.

5.3.3.2 Configuring the transportType

The ServiceMgr.transportType configuration parameter is used to specify in the type of physical connection to use. For example:

```
ServiceMgr.transportType = ServiceMgr.TransportType_FILE;
```

The following transport options are available:

❑ **TransportType_ETHERNET.** Events and control messages are sent between the host and targets via Ethernet. By default, the NDK is used. The application is responsible for configuring and starting networking stack.

❑ **TransportType_FILE.** Events are sent between the host and targets via File over JTAG. (Note that control messages cannot be sent via this transport.)

❑ **TransportType_USER.** You plan write your own transport functions or use transport functions from some other source and specify them using the ServiceMgr.transportFxns parameter. See Section 5.4.8, *Custom Transport Functions for Use with ServiceMgr* if you plan to use this option.

Not all transport options are supported on all devices.

If you do not specify a transportType, UIA picks an appropriate transport implementation to use based on your device. The defaults are found using the ti.uia.family.Settings module. If the device is unknown to ServiceMgr, TransportType_ETHERNET is used.

Note: The transport type is ignored if you configure events to be uploaded from a simulator, probe points, or JTAG (run-mode or stop-mode) using the LoggingSetup.eventUploadMode parameter (see Section 5.3.1). By default, the eventUploadMode is JTAG stop-mode, and the transportType is ignored.

5.3.3.3 Configuring the masterProcId

If this is a multi-core application, you need to set the ServiceMgr.masterProcId parameter to indicate which core you want to act as the master core for UIA. All packets will be routed through the master core to the instrumentation host.

The core ID numbers correspond the IPC's MultiProc ID values. The ServiceMgr module uses IPC to discover the core configuration and to communicate between cores.

Note: The core chosen as the master must be started first.

For example to have core 3 be the masterProcId on a multicore device:

```
var ServiceMgr =
    xdc.useModule('ti.uia.runtime.ServiceMgr');
ServiceMgr.topology = ServiceMgr.Topology_MULTICORE;
ServiceMgr.masterProcId = 3;
```

5.3.3.4 Configuring Other ServiceMgr Parameters

You can configure how often the ServiceMgr gathers events from logs and transfers them to the host. For example:

```
ServiceMgr.periodInMs = 200;
```

See Section 5.3.2, *Configuring ti.uia.services.Rta* for details on the interactions between the ServiceMgr.periodInMs parameter, the Rta.periodInMs parameter, and the SYS/BIOS clock interval.

UIA makes a distinction between event and message (control) packets.

- ❑ **Event packets** are large in order to hold several event records. For example, if you are using an Ethernet transport, the maximum event packet size is 1472 bytes, which includes the packet header. UIA chooses the size and number of event packets based on the transport and device. In a multicore architecture, you may want to increase the value of the numEventPacketBufs parameter beyond the default of 2 if a lot of logging is done on the non-master cores. This will help reduce the number of events lost.
- ❑ **Message (control) packets** are small and hold only a control message sent from the instrumentation host. The default message packet size is 128 bytes. Note that control messages are only supported via the Ethernet transport. The ServiceMgr.supportControl parameter specifies whether control messages are enabled; it is set automatically as a result of the transport that is used. Control packets occur much less frequently than event packets, so it is rarely

necessary to increase the number of control packet buffers. For those rare cases, you can use the numIncomingCtrlPacketBufs and numOutgoingCtrlPacketBufs parameters to configure the number of message packets.

The ServiceMgr module uses one or two Task threads depending on whether control message handling is enabled. By default, these Tasks have a priority of 1, the lowest level. The receive Task receives control messages from the instrumentation host and forwards them to the transfer agent Task. The transfer agent Task handles all other activities, including period management, event collection, communicating with remote cores, and sending UIA packets to the instrumentation host. The ServiceMgr module provides the following parameters for configuring the priority, stack sizes, and placement of these tasks: rxTaskPriority, rxTaskStackSize, rxTaskStackSection, transferAgentPriority, transferAgentStackSize, and transferAgentStackSection.

See the ti.uia.runtime.ServiceMgr topic in the online help API Reference (CDOC) for more information about this module.

5.3.4 Configuring ti.uia.runtime.LoggerCircBuf

As described in Section 5.3.1, *Configuring ti.uia.sysbios.LoggingSetup*, UIA creates and uses several loggers to contain events.

Loggers are implementations of an interface, ILogger, which is defined by XDCtools. By default, the loggers created by LoggingSetup use the ti.uia.runtime.LoggerCircBuf implementation provided with UIA.

Note: You can only use loggers that inherit from ti.uia.runtime.IUIATransfer with UIA. Currently only the ti.uia.runtime.LoggerCircBuf and ti.uia.runtime.LoggerSMti.uia.runtime.LoggerSM implementations are supported. You cannot use xdc.runtime.LoggerBuf with UIA.

LoggerCircBuf maintains variable-length logger instances that store events in a compressed, non-decoded format in memory. The ti.uia.services.Rta module is responsible for copying the events out of LoggerCircBuf and sending them to CCS on the instrumentation host.

Each core must have its own logger instances. Instances cannot be shared among multiple cores due to the overhead that would be required for multicore synchronization.

You can use the LoggerCircBuf module to configure your own loggers for UIA (instead of using ti.uia.sysbios.LoggingSetup’s defaults). This allows you to configure parameters for the loggers, such as the section that contains the buffer.

For example, the following statements configure a Load logger to be used by LoggingSetup. The size is larger than the default and the logger is stored in a non-default memory section:

```
var loggerCircBufParams = new LoggerCircBuf.Params();

loggerCircBufParams.transferBufSize = 2048;
/* must also place memory section via Program.sectMap */
loggerCircBufParams.bufSection = ’.myLoggerSection’;

var logger = LoggerCircBuf.create(loggerCircBufParams);
logger.instance.name = "Load Logger";

var LoggingSetup =
    xdc.useModule('ti.uia.sysbios.LoggingSetup');
LoggingSetup.loadLogger = logger;
```

You can also create extra LoggerCircBuf instances to handle events from certain modules. Since LoggerCircBuf (and logger implementations in general) can drop events, it is advantageous to put critical events in a dedicated logger instance. For example, the following code creates a logger just for the Swi module.

```
var LoggerCircBuf =
    xdc.useModule('ti.uia.runtime.LoggerCircBuf');
var Swi = xdc.useModule('ti.sysbios.knl.Swi');

/* Give the Swi module its own logger. */
var loggerCircBufParams = new LoggerCircBuf.Params();
loggerCircBufParams.transferBufSize = 65536;
var swiLog = LoggerCircBuf.create(loggerCircBufParams);
swiLog.instance.name = "Swi Logger";
Swi.common$.logger = swiLog;

/* Enable the Swi module to log events */
Swi.common$.diags_USER1 = Diags.RUNTIME_ON;
Swi.common$.diags_USER2 = Diags.RUNTIME_ON;
```

Events generally range in size from 8 bytes (Log_write0 with no timestamp) to 48 bytes (Log_write8 with timestamp). Note that snapshot and memory dump events can be even larger.

See the ti.uia.runtime.LoggerCircBuf topic in the online help API Reference (CDOC) for more information about this module.

5.3.4.1 Configuring a Shared LoggerCircBuf when Multiple Cores Run the Same Image

If you have a single target image that is loaded onto multiple cores, and the LoggerCircBuf loggers are stored in shared memory (for example, external memory), you should set the LoggerCircBufParams.numCores parameter to specify the number of cores running the same image.

The numCores parameter provides a solution to the problem that occurs if the logger's buffer is in shared memory (for example, DDR). Since the image is the same for all the cores, each core attempts to write to the same buffer in the shared memory.

The following example shows how to set the numCores parameter for a logger that is stored in shared memory.

```
var loggerCircBufParams = new LoggerCircBuf.Params();

loggerCircBufParams.transferBufSize = 1024;
loggerCircBufParams.numCores = 4;
/* must also place memory section via Program.sectMap */
loggerCircBufParams.bufSection = ’.sharedMemSection’;

var logger = LoggerCircBuf.create(loggerCircBufParams);
logger.instance.name = "Load Logger";

var LoggingSetup =
    xdc.useModule('ti.uia.sysbios.LoggingSetup');
LoggingSetup.loadLogger = logger;
```

Setting numCores to a value greater than 1 causes LoggerCircBuf to statically allocate additional memory to allow each core to have transferBufSize amount of memory. The amount of memory allocated is the logger’s transferBufSize * numCores.

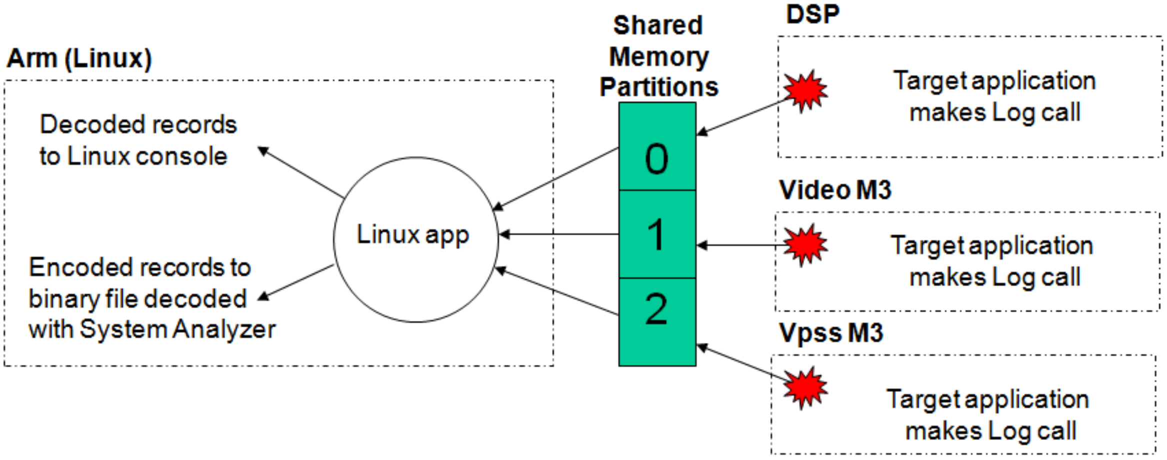
Note: You should set the numCores parameter to a value greater than one *only* if a single image is used on multiple cores of a multi-core device *and* the logger instance's buffer is stored in shared memory. Increasing numCores in other cases will still allow the application to function, but will waste memory.

The default value for numCores is 1, which does not reserve any additional memory for the logger.

5.3.5 Configuring ti.uia.runtime.LoggerSM

The LoggerSM logger implementation stores log records into shared memory. It is intended to be used with a SoC system (such as EVMTI816x) where Linux is running on the host core (such as CortexA8) and SYS/BIOS is running on the targets (for example, M3 and DSP).

When a Log call is made on the target, the record is written into the shared memory. On the Linux host, the records can be read from the shared memory and either displayed to the console or written to a file to be processed by MCSA at a later time.


Arm (Linux)
Decoded records to Linux console
Encoded records to binary file decoded with System Analyzer
Linux app
Shared Memory Partitions
0
1
2
DSP
Target application makes Log call
Video M3
Target application makes Log call
Vpss M3
Target application makes Log call

Each target is assigned its own partition of the shared memory, and writes its log events to that partition only.

For use on Linux, UIA ships a LoggerSM module that can be used to process the records and a command-line application that can make use of the LoggerSM module.

The example that uses LoggerSM on EVMTI816x is located in <*uia_install*>\packages\ti\uia\examples\evmti816x. The tools for use on Linux when the targets are using LoggerSM are located in <*uia_install*>/packages/ti/uia/linux.

Constraints

- ❑ The shared memory must be in a non-cacheable region. LoggerSM does not perform any cache coherency calls. You can place memory in non-cached memory via different mechanisms on different target types. For example, use ti.sysbios.hal.ammu.AMMU for the M3 cores and the ti.sysbios.family.c64p.Cache module for the DSP on the EVMTI816x. See the EVMTI816x examples provided with UIA for details.
- ❑ The shared memory must be aligned on a 4096 boundary.

- ❏ The shared memory must be in a NOLOAD section if multiple cores are using the memory.
- ❏ All cores, including the targets and Linux ARM core, must have the same size memory units, also called Minimum Addressable Data Unit (MADU).
- ❏ Currently the targets and host must all have the same endianness. Removing this restriction is a future enhancement. For example, the EVMTI816x's CortexA8, DSP, and M3 all are little endian.

Configuring the Targets

The following example configuration script causes a target to use the LoggerSM module to place UIA events in shared memory:

```
var LoggerSM = xdc.useModule('ti.uia.runtime.LoggerSM');
var LoggingSetup =
    xdc.useModule('ti.uia.sysbios.LoggingSetup');
var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');

LoggerSM.sharedMemorySize = 0x20000;
LoggerSM.numPartitions = 3;
LoggerSM.partitionId = MultiProc.id;
LoggerSM.bufSection = ".loggerSM";
var logger = LoggerSM.create();

LoggingSetup.loadLogger = logger;
LoggingSetup.mainLogger = logger;
LoggingSetup.sysbiosLogger = logger;
```

Alternately, since only one LoggerSM instance is used for all logging, you can create the logger instance and use it in all cases where logging occurs as follows:

```
Defaults.common$.logger = LoggerSM.create();
```

The parameters you can set include:

- ❏ **sharedMemorySize** specifies the total size of the shared memory for all targets. You must set this parameter to the same value on all targets. The default size is 0x20000 MADUs. For example, on the EVMTI816x, if the sharedMemorySize is 0x3000, each target—DSP, videoM3 and vpssM3—would get 0x1000 MADUs of shared memory for log records.
- ❏ **numPartitions** specifies the number of cores that can use the shared memory. The memory will be divided into this number of equal partitions. You must set this parameter to the same value on all targets. The default is 3 partitions. If the sharedMemorySize is not evenly divisible by 3, the extra memory is unused at the end of the shared memory.

❏ **partitionID** determines which partition this target uses. This value must be different on all targets. For example, in the EVMTI816x examples, the DSP gets partition 0, videoM3 gets 1, and vpssM3 gets 2. This corresponds with the IPC Multicore IDs. You can set this parameter at runtime using the LoggerSM_setPartitionId() API, which must be called before module startup occurs. For example, you could call this function using the xdc.runtime.Startup.firstFxns array.

❏ **decode** specifies whether the target decodes the record before writing it to shared memory. The default is true, which means the target decodes the record into an ASCII string and writes the string into the shared memory. The Linux tool extracts the string and prints it to the Linux console. This approach is expensive from a performance standpoint. The benefit is that it is easy to manage and view on the host.

  If you set decode to false, encoded records are written to the shared memory. The Linux tool writes the encoded records to a single binary file (see page 5–27) that can be decoded by MCSA. This approach makes Log module calls much faster on the target. Note that different cores can have different decode values.

❏ **overwrite** determines what happens if the shared memory partition fills up. By default, any new records are discarded. This mode allows you to read the records while the target is running. If you set this parameter to true, old records are overwritten by new records. In this mode, records can only be read on Linux when the targets are halted (or crashed), because both the target and host must update a read pointer.

❏ **bufSection** specifies the section in which to place the logger’s buffer. See the next section for details.

Placing the Shared Memory

The **bufSection** parameter tells LoggerSM where to place the buffer it creates. Each core’s bufSection must be placed at the same address. For example, the EVMTI816x examples all place the ".loggerSM" section at 0x8f000000 with the following configuration statements:

```
LoggerSM.bufSection = ".loggerSM";
...
Program.sectMap[".loggerSM"] = new Program.SectionSpec();
Program.sectMap[".loggerSM"].loadAddress = 0x8f000000;
  // or loadSegment = "LOGGERSM";
Program.sectMap[".loggerSM"].type = "NOLOAD";
```

Note that the "NOLOAD" configuration is required. Without this, as each core is loaded, it would wipe out the previous core's initialization of its partition of the shared memory.

The LoggerSM module requires that all targets sharing the memory have the same base address. To confirm all targets have the same address, look at the address of the ti_uia_runtime_LoggerSM_sharedBuffer__A symbol in all the targets' mapfiles. The address must be the same on all targets. This physical address must also be used in the Linux LoggerSM and loggerSMDump tools.

There are several ways to place the shared memory for the .loggerSM section. Here are two ways.

❑ The EVMTI816x LoggerSM examples use a custom platform file (in ti/uia/examples/platforms) where an explicit memory segment is created as follows:

```
["DDR_SR0", {name: "DDR_SR0", base: 0x8E000000,
   len: 0x01000000, space: "code/data",access: "RWX"}],
["DDR_VPSS", {name: "DDR_VPSS", base: 0x8F800000,
   len: 0x00800000, space: "code/data",access: "RWX"}],
["LOGGERSM", {name: "LOGGERSM", base: 0x8F000000,
   len: 0x00020000, space: "data",access: "RWX"}],
```

❑ You can create a custom memory map in the config.bld file as follows:

```
/* For UIA logging to linux terminal */
memory[24] = ["LOGGERSM", {name: "LOGGERSM",
    base: 0x8F000000, len: 0x00020000, space: "data"}];
```

Using the Linux LoggerSM Module

The non-XDC LoggerSM module knows how to read the shared memory contents and process them. If the records are decoded, it displays them to the Linux console. If the records are encoded, they are written (along with the UIA events headers) into a binary file. This module is provided in <*uia_install*>/packages/ti/uia/linux.

The two main APIs in this module are LoggerSM_run() and LoggerSM_setName().

❑ **LoggerSM_run()** processes logs in all the partitions. The syntax is as follows:

```
int LoggerSM_run(  unsigned int physBaseAddr,
                   size_t sharedMemorySize,
                   unsigned int numPartitions,
                   unsigned int partitionMask,
                   char *filename)
```

  - **physBaseAddr** specifies the physical address of the shared memory. The address used here must match the address configured on all the targets.
  - **sharedMemorySize** specifies the total size of the shared memory. This size must match the targets' sharedMemorySize parameter.
  - **numPartitions** specifies the number of partitions in the shared memory. This must match the targets' numPartitions parameter.
  - **partitionMask** is a bitmask to determine which partitions to process. For example, if numPartitions is 3, but you only want to process partitions 1 and 2, set the partitionMask to 0x6 (110b).
  - **filename** specifies a filename to use if encoded records are found. If this is NULL, the default name is loggerSM.bin. Encoded records from all targets that send encoded records are placed in the same file. Since a UIA Packet header is also included, MCSA can determine which records go with which core.

  This function returns LoggerSM_ERROR if any parameters are invalid; otherwise, this function never returns.

❑ **LoggerSM_setName()** associates a name to a partition ID. Calling this function for each target before you call LoggerSM_run() allows the decoded output to include the name instead of just the partition ID. The syntax is as follow:

```
int LoggerSM_setName(  unsigned int partitionId,
                       char *name);
```

  This function returns LoggerSM_SUCCESS if it is successful or LoggerSM_ERROR if any parameters are invalid.

See the source code in LoggerSM.c and LoggerSM.h for more APIs.

Using the Linux loggerSMDump Tool

UIA also provides the loggerSMDump.c file, which shows how to use the Linux LoggerSM module with the EVMTI816x board to send decoded records to the console and encoded records to a binary file. This example is provided in the *<uia_install>*/packages/ti/uia/examples/evmti816x directory. The directory also includes a makefile to build the tool. the loggerSMDump.c file calls both LoggerSM_setName() and LoggerSM_run().

The command-line syntax is:

```
loggerSMDump.out <addr> <core_name> [<filename>]
```

To terminate the tool, press Ctrl+C.

- **addr.** The physical address of the shared memory in Hex. The shared memory physical address must be 4 KB aligned.
- **core_name.** The name of the cores that are processed. Valid names are: "dsp", "video", "vpss", "m3" or "all". "m3" processes both video and vpss. "all" processes all three targets.
- **filename.** If target sends encoded records, specify the name of the file to store the encoded records. They can be decoded by MCSA. This parameter is optional. If no filename is specified and encoded events are found, the default file name is loggerSM.bin.

Here are some command-line examples:

```
./loggerSMDump.out 0x8f000000 video myBinaryFile

./loggerSMDump.out 0x8f000000 m3 myBinaryFile

./loggerSMDump.out 0x8f000000 all
```

This example shows output from loggerSMDump. In this case, the video M3’s records were encoded, so they went into the binary file instead.

```
N:VPSS  P:2 #:00113 T:00000000|21f447cd S:Start:
N:VPSS  P:2 #:00114 T:00000000|21f637d3 S:Stop:
N:VPSS  P:2 #:00115 T:00000000|21f69b15 S:count = 35
N:DSP   P:0 #:00249 T:00000000|3ce48c2f S:Stop:
N:DSP   P:0 #:00250 T:00000000|3ce5f28d S:count = 80
N:DSP   P:0 #:00251 T:00000000|3d8689eb S:Start:
N:DSP   P:0 #:00252 T:00000000|3da59483 S:Stop:
N:DSP   P:0 #:00253 T:00000000|3da6facf S:count = 81
N:VPSS  P:2 #:00116 T:00000000|22d92a23 S:Start:
N:VPSS  P:2 #:00117 T:00000000|22db1689 S:Stop:
N:VPSS  P:2 #:00118 T:00000000|22db7797 S:count = 36
```

Use the following legend to parse the output:

- N: name of the partition owner
- P: partition Id
- T: timestamp [high 32 bits | low 32 bits]
- S: decoded string

5.3.6 Configuring ti.uia.runtime.LogSync

Events that are logged by different CPUs are typically timestamped using a timer that is local to that CPU in order to avoid the overhead of going off-chip to read a shared timer.

In order to correlate one CPU’s events with those logged by another CPU, it is necessary to log "sync point events" that have, as parameters, both the local CPU’s timestamp value and a "global timestamp" that was read from a global shared timer. Any CPUs that log sync point events with global timestamps read from the same global shared timer can be correlated with each other and displayed against a common timeline for analysis and comparison.

The ti.uia.runtime.LogSync module is provided in order to support this type of event correlation. It provides sync events that are used to correlate timestamp values. The Rta module handles all of the sync point event logging that is required in order to support the event correlation.

In general, you will need to configure the LogSync module, but will not need to call the module’s APIs from your application. For information about LogSync module APIs, see Section 5.4.5.

Setting the Global Timestamp Module Proxy

If you have a multi-core application, your application must configure the GlobalTimestampProxy parameter on a target-specific basis to provide a timestamp server.

This parameter defaults correctly for the C6472 and TCI6616 platforms. However, for EVMTI816x, it defaults to null, which prevents any multi-core event correlation from being performed. In general, you can use a timestamp module that implements the IUIATimestampProvider interface for your target. You should configure the GlobalTimestampProxy as follows:

```
var LogSync = xdc.useModule('ti.uia.runtime.LogSync');
var GlobalTimestampTimer =
xdc.useModule('ti.uia.family.c64p.TimestampC6472Timer');

LogSync.GlobalTimestampProxy = GlobalTimestampTimer;
```

- ❏ **C6472.** Use the ti.uia.family.c64p.TimestampC6472Timer module as the proxy. When you use this proxy, the default value for the maxCpuClockFreq is 700 MHz.
- ❏ **TCI6616 (simulator).** The ti.uia.family.c66.TimestampC66XGlobal module should be the proxy.
- ❏ **EVMTI816x.** The ti.uia.family.dm.TimestampDM816XTimer module should be the proxy. This setting is not auto-configured.

- **Other.** If no module that implements the IUIATimestampProvider interface for your target is available, you can use, for example, a timer provided by SYS/BIOS as the global timer source for event correlation "sync point" timestamps. The following statements configure such as proxy:

  ```
  var LogSync = xdc.useModule('ti.uia.runtime.LogSync');
  var BiosTimer =
   xdc.useModule('ti.sysbios.family.c64p.TimestampProvider');
  LogSync.GlobalTimestampProxy = BiosTimer;
  ```

  If you are using a global timer that does not implement the IUIATimestampProvider interface, you must also configure the maxGlobalClockFreq parameter. If the maxGlobalClockFreq parameter is not configured, you see a warning message at build time that says UIA Event correlation is disabled. You must configure both the maxGlobalClockFreq.lo and maxGlobalClockFreq.hi parameters, which set the lower and upper 32 bits of the frequency, respectively.

  ```
  LogSync.maxGlobalClockFreq.lo = 700000000; //low 32b
  LogSync.maxGlobalClockFreq.hi = 0;         //upper 32b
  ```

  If the CPU timestamp clock frequency is not 700 MHz, you must also configure the lo and hi parameters. For example:

  ```
  LogSync.maxCpuClockFreq.lo = 1000000000;   //low 32b
  LogSync.maxCpuClockFreq.hi = 0;            //upper 32b
  ```

Setting the Local Timestamp Module Proxy

If the frequency of the local CPU may change at run-time, you also need to configure the CpuTimestampProxy parameter of the LogSync module. The timestamp proxies provided for this purpose are:

- ti.uia.family.c64p.TimestampC64XLocal
- ti.uia.family.c66.TimestampC66XLocal

Configuring the CpuTimestampProxy with a local timestamp module allows applications that change the CPU frequency to report this information to MCSA so that event timestamps can be adjusted to accommodate the change in frequency.

The following configuration script shows how to configure the C66x Local Timestamp module for use as the CpuTimestampProxy:

```
var TimestampC66Local =
    xdc.useModule('ti.uia.family.c66.TimestampC66Local');
TimestampC66Local.maxTimerClockFreq = {lo:1200000000,hi:0};
var LogSync = xdc.useModule('ti.uia.runtime.LogSync');
LogSync.CpuTimestampProxy = TimestampC66Local;
```

Injecting Sync Points into Hardware Trace Streams

Correlation with hardware trace (for example, the C64x+ CPU Trace) can be enabled by injecting references to the sync point events into the hardware trace stream. The LogSync.injectIntoTraceFxn parameter allows you to inject sync point information into hardware trace streams. You can specify a pointer to a function that handles the ISA-specific details of injecting information into the trace stream.

For C64x+ Full-GEM (Generalized Embedded Megamodule) devices, which support CPU trace and Advance Event Triggering (AET), use the address of the GemTraceSync_injectIntoTrace() function provided by the ti.uia.family.c64p.GemTraceSync module. For information about GEM, see the *TMS320C64x+ DSP Megamodule Reference Guide* (SPRU871).

This example for the TMS320C6472 platform shows configuration statements with multicore event correlation for the CPUs enabled. More examples are provided in the CDOC for the LogSync module.

```
// Including Rta causes Log records to be collected and sent
// to the instrumentation host. The Rta module logs sync
// point events when it receives the start or stop
// command, and prior to sending up a new event packet if
// LogSync_isSyncPointEventRequired() returns true.
var Rta = xdc.useModule('ti.uia.services.Rta');

// By default, sync point events are logged to a dedicated
// LoggerCircBuf buffer named 'SyncLog' assigned to the
// LogSync module. A dedicated event logger buffer helps
// ensure that sufficient timing information is captured to
// enable accurate multicore event correlation. Configure
// LogSync.defaultSyncLoggerSize to set a buffer size.
var LogSync = xdc.useModule('ti.uia.runtime.LogSync');

// For C64x+ and C66x devices that provide CPU trace hardware
// capabilities, the following line enables injection of
// correlation information into the GEM CPU trace, enabling
// correlation of software events with the CPU trace events.
var GemTraceSync =
   xdc.useModule('ti.uia.family.c64p.GemTraceSync');

// Configure a shared timer to act as a global time reference
// to enable multicore correlation. The TimestampC6472Timer
// module implements the IUIATimestampProvider interface, so
// assigning this timer to LogSync.GlobalTimestampProxy
// configures LogSync's global clock params automatically.
var TimestampC6472Timer =
   xdc.useModule('ti.uia.family.c64p.TimestampC6472Timer');
LogSync.GlobalTimestampProxy = TimestampC6472Timer;
```

Setting CPU-Related Parameters

LogSync provides a number of CPU-related parameters—for example, CpuTimestampProxy, cpuTimestampCyclesPerTick, maxCpuClockFreq, and canCpuCyclesPerTickBeChanged. These parameters generally default to the correct values unless you are porting to a non-standard target. For example, the CpuTimestampProxy defaults to the same proxy used by the xdc.runtime.Timestamp module provided with XDCtools.

Setting the globalTimestampCpuCyclesPerTick parameter is optional. It is used to convert global timestamp tick counts into CPU cycle counts for devices where there is a fixed relationship between the global timer frequency and the CPU clock. For example:

```
LogSync.globalTimestampCpuCyclesPerTick = 6;
```

5.3.7 Configuring IPC

When ServiceMgr.topology is ServiceMgr.Topology_MULTICORE, the underlying UIA code uses IPC (or more specifically its MessageQ and SharedRegion modules) to move data between cores. See the IPC documentation for details on how to configure communication between cores for your multi-core application.

The following IPC resources are used by UIA:

- ❑ Uses up to 4 message queues on each processor.
- ❑ Uses the SharedRegion heap when allocating messages during initialization. Which SharedRegion heap is determined by the IpcMP.sharedRegionId parameter. The default is SharedRegion 0.
- ❑ Determines the SharedRegion allocation size by the size and number of packets. This is calculated using ServiceMgr parameters as follows:

  ```
  maxCtrlPacketSize *
  (numIncomingCtrlPacketBufs + numOutgoingCtrlPacketBufs)
  + maxEventPacketSize * numEventPacketBufs
  ```

See Section 6.1, *IPC and SysLink Usage* for further information.

Note that, depending on the cache settings, these sizes might be rounded up to a cache boundary.

5.4 Target-Side Coding with UIA APIs

By default, SYS/BIOS provides instrumentation data to be sent to CCS on the host PC if you have configured UIA. It is not necessary to do any additional target-side coding once UIA is enabled.

If you want to add additional instrumentation, you can do by adding C code to your target application as described in the sections that follow.

In general, UIA provides a number of new events that can be using with the existing Log module provided as part of XDCtools. Section 5.4.1 describes how to log events, Section 5.4.2 describes how to enable event logging, and Section 5.4.3 provides an overview of the events provided by UIA.

Section 5.4.4 describes the LogSnapshot module APIs, which allow you to log memory values, register values, and stack contents.

Section 5.4.5 describes ways to configure and customize the synchronization between timestamps on multiple targets.

Section 5.4.6 describes the APIs provided by the LogCtxChg module, which allows you to log context change events.

Section 5.4.7 describes how to use the Rta module APIs to control the behavior of loggers at runtime.

Section 5.4.8 explains how to create and integrate a custom transport.

5.4.1 Logging Events with Log_write() Functions

The Log_writeX() functions—Log_write0() through Log_write8()—provided by the xdc.runtime.Log module expect an event as the first argument. This argument is of type Log_Event.

The ti.uia.events package contains a number of modules that define additional events that can be passed to the Log_writeX() functions. For example, this code uses events defined by the UIABenchmark module to determine the time between two calls:

```
#include <ti/uia/events/UIABenchmark.h>
...
Log_write1(UIABenchmark_start, (xdc_IArg)"start A");
...
Log_write1(UIABenchmark_stop, (xdc_IArg)"stop A");
```

In order to use such events with Log_writeX() functions, you must enable the correct bit in the appropriate module's diagnostics mask as shown in Section 5.4.2 and choose an event to use as described in Section 5.4.3.

5.4.2 Enabling Event Output with the Diagnostics Mask

For an overview of how to enable SYS/BIOS load and event logging, see Section 5.2. This section discusses how to enable logging of events provided by the ti.uia.events module.

Whether events are sent to the host or not is determined by the particular bit in the diagnostics mask of the module in whose context the Log_writeX() call executes. For example, UIABenchmark events are controlled by the ANALYSIS bit in the diagnostics mask.

For example, suppose you placed calls that pass UIABenchmark events to Log_write1() in a Swi function to surround some activity you want to benchmark.

```
Log_write1(UIABenchmark_start, (xdc_IArg)"start A");
...
Log_write1(UIABenchmark_stop, (xdc_IArg)"stop A");
```

If the ANALYSIS bit in the diagnostics mask were off for the Swi module, no messages would be generated by these Log_write1() calls.

By default, the LoggingSetup module sets the ANALYSIS bit to on only for the Main module, which affects logging calls that occur during your main() function and other functions that run outside the context of a SYS/BIOS thread. However, LoggingSetup does not set the ANALYSIS bit for the Swi module.

To cause these benchmark events to be output, your configuration file should contain statements like the following to turn the ANALYSIS bit for the Swi module on in all cases:

```
var Swi = xdc.useModule('ti.sysbios.knl.Swi');
var Diags = xdc.useModule('xdc.runtime.Diags');
var UIABenchmark =
    xdc.useModule('ti.uia.events.UIABenchmark');

Swi.common$.diags_ANALYSIS = Diags.ALWAYS_ON;
```

Alternately, you could enable output of UIABenchmark events within the Swi context by setting the ANALYSIS bit to RUNTIME_OFF and then turning the bit on and off in your runtime code. For example, your configuration file might contain the following statements:

```
var Swi = xdc.useModule('ti.sysbios.knl.Swi');
var Diags = xdc.useModule('xdc.runtime.Diags');
var UIABenchmark =
    xdc.useModule('ti.uia.events.UIABenchmark');

Swi.common$.diags_ANALYSIS = Diags.RUNTIME_OFF;
```

Then, your C code could contain the following to turn ANALYSIS logging on and off. (See the online documentation for the Diags_setMask()Diags_setMask() function for details about its control string argument.)

```
// turn on logging of ANALYSIS events in the Swi module
Diags_setMask("ti.sysbios.knl.Swi+Z");
...
// turn off logging of ANALYSIS events in the Swi module
Diags_setMask("ti.sysbios.knl.Swi-Z");
```

5.4.3 Events Provided by UIA

The ti.uia.events package contains a number of modules that define additional events that can be passed to the Log_writeX() functions. Section 5.4.2 uses the UIABenchmark_start and UIABenchmark_stop events from the ti.uia.events.UIABenchmark module as an example.

To use an event described in this section, you must do the following:

❏ Include the appropriate UIA module in your .cfg configuration file. For example:

```
var UIABenchmark =
    xdc.useModule('ti.uia.events.UIARBenchmark');
```

❏ Include the appropriate header files in your C source file. For example:

```
#include <xdc/runtime/Log.h>
#include <ti/uia/events/UIABenchmark.h>
```

❏ Use the event in your C source file. For example:

```
Log_write2(UIABenchmark_start, (xdc_IArg)"Msg %d",
           msgId);
```

The following UIA modules provide events you can use with the Log_writeX() functions:

Table 5–3 Log_Event Types Defined by UIA Modules

Module	Events	Diagnostics Control Bit	Comments
UIABenchmark	Start and stop events. Versions of start and stop let you identify the instance of the function being benchmarked using a numeric ID, an address, or a string.	diags_ANALYSIS	UIABenchmark supports context-awareness. That is, it reports time elapsed exclusive of time spent in other threads that preempt or otherwise take control from the thread being benchmarked.
UIAErr	Numerous error events used to identify common errors in a consistent way.	diags_STATUS (ALWAYS_ON by default)	These events have an EventLevel of EMERGENCY, CRITICAL, or ERROR. Special formatting specifiers let you send the file and line at which an error occurred.
UIAEvt	Events with detail, info, and warning priority levels.	diags_STATUS or diags_INFO depending on level	An event code or string can be with each event type.
UIAMessage	Events for msgReceived, msgSent, replyReceived, and replySent.	diags_INFO	Used UIA and other tools and services to report the number of messages sent and received between tasks and CPUs.
UIAStatistic	Reports bytes processed, CPU load, words processed, and free bytes.	diags_ANALYSIS	Special formatting specifiers let you send the file and line at which the statistic was recorded.

See the online reference documentation (CDOC) for the modules in the ti.uia.events package for more details and examples that use these events.

The online reference documentation for the event modules in the ti.uia.events package contains default message formats for each event in the XDCscript configuration (red) section of each topic. A number of the message formats for these events contain the special formatting specifiers %$S and %$F.

❑ **%$S —** Handles a string argument passed to the event that can, in turn, contain additional formatting specifiers to be interpreted recursively. Note that you cannot use the $S formatting specifier in strings passed as a parameter. For example, the message format for the UIAErr_fatalWithStr event includes the 0x%x format specifier for an integer error code and the %$S format specifier for a string that may in turn contain format specifiers. This example uses that event:

```
#include <xdc/runtime/Log.h>
#include <ti/uia/events/UIAErr.h>
...
Int myFatalErrorCode = 0xDEADC0DE;
String myFatalErrorStr = "Fatal error when i=%d";
Int i;
...
Log_write3(UIAErr_fatalWithStr, myFatalErrorCode,
          (IArg)myFatalErrorStr, i);
```

- ❑ **%$F** — Places the file name and line number at which the event occurred in the message. The call to Log_writeX() for an event that includes the %$F specifier in its formatting string should pass the filename (using the __FILE__ constant string) and the line number (using __LINE__). For example:

```
#include <xdc/runtime/Log.h>
#include <ti/uia/events/UIAErr.h>
...
Log_write2(UIAErr_divisionByZero, (IArg)__FILE__,
    __LINE__);
```

  The resulting message might be:

```
"ERROR: Division by zero at demo.c line 1234."
```

The ti.uia.services.Rta module defines Log_Events that are use internally when that module sends events to the host. You should not use these events in application code.

5.4.4 LogSnapshot APIs for Logging State Information

You can use snapshot events to log dynamic target state information. This lets you capture the execution context of the application at a particular moment in time.

You call functions from the ti.uia.runtime.LogSnapshot module in order to use a snapshot event. The diags_ANALYSIS bit in the module’s diagnostics mask must be on in order for snapshot events to be logged. (This bit is on by default.)

The LogSnapshot module provides the following functions:

- ❑ **LogSnapshot_getSnapshotId()** Returns a unique ID used to group a set of snapshot events together.

- **LogSnapshot_writeMemoryBlock()** Generates a LogSnapshot event for a block of memory. The output is the contents of the memory block along with information about the memory block. See Example 1 that follows.
- **LogSnapshot_writeNameOfReference()** This function lets you associate a string name with the handle for a dynamically-created instance. A common use would be to log a name for a dynamically-created Task thread. The host-side UIA features can then use this name when displaying data about the Task's execution. You might want to call this API in the create hook function for SYS/BIOS Tasks. See Example 2 that follows.
- **LogSnapshot_writeString()** Generates a LogSnapshot event for a string on the heap. Normally, when you log a string using one of the Log APIs, what is actually logged is the address of a constant string that was generated at compile time. You can use this API to log a string that is created at run-time. This API logs the value of the memory location that contains the contents of the string, not just the address of the string. See Example 3 that follows.

Example 1: Logging a Snapshot to Display the Contents of Some Memory

For example, the following C code logs a snapshot event to capture a block of memory:

```
#include <ti/uia/runtime/LogSnapshot.h>
...
UInt32* pIntArray = (UInt32 *)malloc(sizeof(UInt32) * 200);
...
LogSnapshot_writeMemoryBlock(0, "pIntArray",
      (UInt32)pIntArray, 200);
...
```

The following will be displayed for this event in the Message column of the Log view, where *`pIntArray`* is the full, unformatted contents of the array. Note that depending on the length of the memory block you specify, the output may be quite long.

```
Memory Snapshot at demo.c line 1234
[ID=0,adrs=0x80002000,len=200 MAUs] pIntArray
```

Example 2: Logging a Name for a Dynamically-Created Task

The following C code logs a Task name:

```
#include <ti/uia/runtime/LogSnapshot.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
...
// Task create hook function that logs the task name.
// Note: Task name is not required when creating a BIOS task.
// However, providing a name makes using the host-side
// analysis features easier to use.

Void tskCreateHook(Task_Handle hTask, Error_Block *eb) {
    String name;
    name = Task_Handle_name(hTask);
    LogSnapshot_writeNameOfReference(hTask,
        "Task_create: name=%s", name, strlen(name)+1);
}
```

The following text would be displayed for this event in the Message column of the Log view if a handle to the "10msThread" were passed to this tskCreateHook function.

```
nameOfReference at demo.c line 1234 [refID=0x80002000]
     Task_create: name=10msThread.
```

Example 3: Logging the Contents of a Dynamically-Created String

The following C code logs the contents of the string stored in `name`.

```
#include <ti/uia/runtime/LogSnapshot.h>
...
Void myFunc(String name){
    ...
    LogSnapshot_writeString(0, name, (UInt32)name,
        strlen(name));
}
```

The following text will be displayed for this event, where `ValueOfParm` is the value of the string rather than the address:

```
String Snapshot at demo.c line 1234 [snapshotID=0]
    Processing name=ValueOfParm
```

5.4.5 LogSync APIs for Multi-Core Timestamps

You can use sync events from the LogSync module to correlate timestamp values in a multicore application. The host uses the difference between the timestamp on the target and the global timestamp to correlate the sequence of events across multiple targets.

The ti.uia.services.Rta module automatically uses the LogSync module to send sync point events if the target has been suspended or halted since the last time an event packet was sent to the host. It also sends sync point events when you start or reset the Log view on the host.

Since the Rta module handles synchronization automatically, in most cases you would only call functions from this module in your application if you are using your own NDK stack and/or are writing a custom service to take the place of the Rta module.

For information about configuring the LogSync module, see Section 5.3.6.

5.4.6 LogCtxChg APIs for Logging Context Switches

The ti.uia.runtime.LogCtxChg module provides a number of functions that log context-switching events. In most cases, you would only use the LogCtxChg APIs if you are instrumenting an OS other than SYS/BIOS.

SYS/BIOS automatically logs events for Task switches and Swi and Hwi start and stop events. You only need to make sure the correct diagnostics settings are enabled to see these events in the UIA analysis features.

In addition to functions that log Task, Swi, and Hwi events (which is done automatically by SYS/BIOS), the LogCtxChg module also provides functions to log changes in application, channel, frame, and user-defined contexts. You might use these APIs in a SYS/BIOS application if you need to keep track of contexts other than the normal threading contexts. For example, the following C code logs a context change event that identifies a newly loaded application ID:

```
#include <ti/uia/runtime/LogCtxChg.h>
...
Void loadApplication(Int newAppId){
    ...
    LogCtxChg_app("New AppID=0x%x",newAppId);
}
```

This event prints the Log call site (%$F) and a format string (%$S) that is formatted with any additional arguments. The following text is an example of what could be displayed for the event:

```
"AppID Ctx Change at Line 123 in appLoader.c
    [Prev. AppID = 0x1234] New AppID=0x1235"
```

5.4.7 Rta Module APIs for Controlling Loggers

The ti.uia.services.Rta module provides a number of APIs you can use to control loggers and the transmission of data by Rta. When you use these Rta APIs, you should be aware that the MCSA features on the instrumentation host may also be sending similar requests to the target in response to user activity within CCS.

Note: The Rta module is available only if you have set LoggingSetup.eventUploadMode to UploadMode_NONJTAGTRANSPORT.

Rta provides several runtime APIs that let you control whether loggers are enabled. These APIs are:

- **Rta_disableAllLogs()** disables all loggers serviced by Rta. All Log records are discarded by a logger when it is disabled.
- **Rta_enableAllLogs()** enables all loggers that are currently disabled.
- **Rta_resetAllLogs()** empties the contents of all loggers serviced by Rta. This function does not change the state of the loggers.

Rta provides runtime APIs to control the transmission of data. These APIs are:

- **Rta_startTx()** tells Rta to begin reading the logs and sending the records to the host.
- **Rta_stopTx()** tells Rta to stop reading the logs and sending them to the host.
- **Rta_snapshotAllLogs()** allows the application to delay reading the logs for the specified waitPeriod. The reset parameter tells Rta whether it should reset all the logs.

Note some transports might require the host to send a "connect" request. For example the TransportNdk needs to obtain the IP address of the host before it can send events.

See the ti.uia.services.Rta topic in the online help for more information about these APIs.

5.4.8 Custom Transport Functions for Use with ServiceMgr

The ti.uia.runtime.Transport module defines function prototypes for the transport functions that can be plugged into the ServiceMgr. UIA ships several implementations of this interface in the <*uia_install*>\packages\ti\uia\sysbios directory.

The transport implementations do not have to be XDC modules. They are simply files containing a set of standard C functions. For an example, see <*uia_install*>\packages\ti\uia\sysbios\TransportNdk.c. Only one transport set can be used on a target. The functions need to be configured at build time via the ti.uia.runtime.ServiceMgr.transportFxns parameter. The ServiceMgr module plugs the transportFxns automatically if the ti.uia.runtime.ServiceMgr.transportType is set to TransportType_USER.

For example, if you or someone else creates a transport called RapidIO, that transport can be plugged in by setting the transportType parameter to ti.uia.runtime.ServiceMgr.TransportType_USER and then plugging the transportFxns manually. It must also set up the following parameters as directed by the developer of the new transport:

- **ServiceMgr.supportControl.** Set to true if the transport supports receiving messages from the host. For example TransportFile does not.
- **ServiceMgr.maxEventPacketSize.** Specify the maximum size of an outgoing event packet. For example TransportNdk uses 1472, which is the EMAC size minus the headers.
- **ServiceMgr.maxCtrlPacketSize.** Specify the maximum size of the control message packets. This can be zero if supportControl is false.

These three parameters are undefined by default, so they must be set if you are using TransportType_USER.

The following example shows a configuration script that plugs a transport called "TransportXYZ" into the ServiceMgr module:

```
var ServiceMgr =
    xdc.useModule('ti.uia.runtime.ServiceMgr');
ServiceMgr.transportType = ServiceMgr.TransportType_USER;
var xyzTransport = {
    initFxn:  '&TransportXYZ_init',
    startFxn: '&TransportXYZ_start',
    recvFxn:  '&TransportXYZ_recv',
    sendFxn:  '&TransportXYZ_send',
    stopFxn:  '&TransportXYZ_stop',
    exitFxn:  '&TransportXYZ_exit',
};
ServiceMgr.transportFxns = xyzTransport;

ServiceMgr.supportControl     = true;
ServiceMgr.maxEventPacketSize = 1024
ServiceMgr.maxCtrlPacketSize  = 1024;
```

The following list describes the transport functions. Note that all of these functions are called by the ServiceMgr module. An application should not call these functions directly. The function call syntax is shown in case you are writing your own transport functions.

- **initFxn()** is called during module startup, which occurs before main() runs. Minimal actions can take place at this point, since interrupts are not yet enabled and the state of the application is just starting up. Generally only internal initialization is performed by this function. This function must have the following call syntax:

  ```
  Void (*initFxn)();
  ```

- **startFxn()** is called once or twice (depending on whether control messages are supported) after the SYS/BIOS Task threads have started to run.

  - This function is called with the UIAPacket_HdrType_EventPkt argument before any events are sent. This allows the transport to initialize anything needed for event transmission. The function returns a handle to a transport-specific structure (or NULL if this is not needed). This handle is passed to the sendFxn() and stopFxn().

  - If the transport supports control messages from a host, startFxn() is also called with the UIAPacket_HdrType_Msg argument. This allows the transport to initialize anything needed for message transmission (both sending and receiving). Again, the transport can return a transport-specific structure. This structure can be different from the one that was returned when the UIAPacket_HdrType_EventPkt argument was passed to startFxn().

This function must have the following call syntax:

```
Ptr (*startFxn)(UIAPacket_HdrType);
```

- ❏ **recvFxn()** is called to receive incoming messages from the host. The handle returned by the startFxn() is passed to the recvFxn(). Also passed in is a buffer and its size. The buffer is passed in as a double pointer. This allows the transport to double-buffer. For example, the recvFxn() can return a different buffer than the one it was passed. This potentially reduces extra copies of the data. The recvFxn() can be a blocking call. The recvFxn() returns the actual number of bytes that are placed into the buffer. If the transport does not support control messages, this function can simply return zero. This function must have the following call syntax:

```
SizeT (*recvFxn)(Ptr, UIAPacket_Hdr**, SizeT);
```

- ❏ **sendFxn()** is called to send either events or messages. If sendFxn() is called to transmit an event, the first parameter is the handle returned from the startFxn(UIAPacket_HdrType_EventPkt) call. Similarly, if a message is being sent, the first parameter is the handle returned from the startFxn(UIAPacket_HdrType_Msg) call. The size of the packet is maintained in the UIAPacket_Hdr. The sendFxn() can be a blocking call. This function returns true or false to indicate whether the send was successful or not. Again, a double pointer is used to allow the transport to return a different buffer to allow double-buffering. This function must have the following call syntax:

```
Bool (*sendFxn)(Ptr, UIAPacket_Hdr**);
```

- ❏ **stopFxn()** is the counterpart to the startFxn() function. The stopFxn() is called the same number of times as the startFxn(). The calls will pass the handles returned by the startFxn(). This function must have the following call syntax:

```
Void (*stopFxn)(Ptr);
```

- ❏ **exitFxn()** is the counterpart to the initFxn() function. This function must have the following call syntax:

```
Void (*exitFxn)(Void);
```

Transports are allowed to have additional functions that can be directly called by the application. For example in the provided TransportFile transport, there is a TransportFile_setFile() function. The downside to adding extended functions is that subsequent ports to a different transport will require changes to the application code.

Chapter 6

Advanced Topics for MCSA

This chapter provides additional information about using MCSA components.

Topic **Page**

6.1 IPC and SysLink Usage

This section lists the IPC and SysLink modules used on various types of cores by UIA. It describes how they are used.

DSP/Video/VPSS

On a DSP, Video, or VPSS core, the ti.uia.runtime.ServiceMgr module uses the following IPC modules.

- ❑ **MessageQ.** The ServiceMgr proxy creates four message queues. Two are used to maintaining "free" messages. These are using instead of allocating messages from a heap.
- ❑ **SharedRegion.** The ServiceMgr proxy allocates memory from the SharedRegion at startup time. It places these buffers onto the "free" message queues. Which SharedRegion to use is configurable.
- ❑ **MultiProc.** The ServiceMgr proxy uses this module to identify the cores.
- ❑ **Ipc.** The ServiceMgr proxy uses the userFxn hook to make sure Ipc is started before UIA tries to use it.

See Section 5.3.7, *Configuring IPC* for more information.

Note: The examples provided with UIA do not use IPC explicitly. They were designed this way to make porting easier.

Linux ServiceMgr Module

In a core running Linux, the Linux ServiceMgr module uses IPC modules as follows:

- ❑ **MessageQ.** ServiceMgr creates two message queues. One is used to maintain "free" messages. These message queues are used instead of allocating messages from a heap.
- ❑ **SharedRegion.** ServiceMgr allocates memory from SharedRegion at startup time. It places these buffers onto the "free" message queue. Which SharedRegion to use is configurable.
- ❑ **MultiProc.** ServiceMgr uses this module to identify the cores.

Linux Application

A Linux application running on a Linux master core uses the following SysLink modules:

- ❑ **SysLink.** Used for setup and destroy calls.
- ❑ **ProcMgrApp.** Used to load the other cores.

6.2 Linux Support for UIA Packet Routing

UIA currently supports the routing of UIA packets on Linux. By default, events are sent out from the Linux core over Ethernet. They are not written to a file by default.

Support for logging on Linux is currently available if you are using the LoggerSM shared memory logger implementation on the EVMTI816x platform. See Section 5.3.5, *Configuring ti.uia.runtime.LoggerSM*.

To use this routing capability, the ti\uia\linux\bin\servicemgr.a library must be linked into your Linux application. The Linux application must call the ServiceMgr_start() API after Ipc has been initialized. If must also call the ServiceMgr_stop() API before Ipc is shutdown. For example code, see the <*uia_install*>packages\ti\uia\examples\evmti816x\uiaDemo.c file.

Since the ServiceMgr on Linux is not an XDC module, you configure it using the ServiceMgr_setConfig() API. There is also a corresponding ServiceMgr_getConfig() API. Both functions are passed the following configuration structure, which is specified in the ServiceMgr.h file:

```
typedef struct ServiceMgr_Config {
    Int maxCtrlPacketSize;
    Int numIncomingCtrlPacketBufs;
    Int sharedRegionId;
    Char fileName[128];
} ServiceMgr_Config;
```

Refer to the ServiceMgr.h file for details about the parameters.

If you do not call ServiceMgr_setconfig(), no usable defaults are supplied.

The uiaDemo.c Linux application example basically performs the following steps:

```
main()
{
    SysLink_setup()
    ProcMgrApp_startup() //on all cores specified on cmdline

    ServiceMgr_start()

    Osal_printf("\nOpen DVT and start getting events." \
                "[Press Enter to shutdown the demo.]\n");
    ch = getchar();

    ServiceMgr_stop()
    ProcMgrApp_shutdown() //on all cores specified on cmdline
    SysLink_destroy()
}
```

6.3 Rebuilding Sample Projects from the Command Line

In addition to building the sample projects in CCS, you can also build them from the command line using the "xdc" command provided by XDCtools. To do so, follow these steps:

1) Copy the complete directory for the example you want to use to a working directory. The examples are provided in the <*uia_install*>\packages\ti\uia\examples directory.

2) Make sure that your XDCPATH environment variable is set correctly. For example, to build the evm6472 or simTCI6616 examples, the following XDCPATH is needed. (Specify the actual locations of your IPC, SYS/BIOS, and UIA installations instead of using # for version numbers.)

```
XDCPATH=
C:/Program Files/Texas Instruments/ipc_1_##_##_##/packages;
C:/Program Files/Texas Instruments/bios_6_##_##_##/packages;
C:/Program Files/Texas Instruments/uia_1_##_##_##/packages;
%NDK_INSTALL_DIR%/packages;
C:/Program Files/Texas Instruments/pdk_c6616_1_#_#_#/packages
```

where

```
NDK_INSTALL_DIR=
C:/Program Files/Texas Instruments/ndk_2_##_##_##
```

3) Change directory to your working directory.

4) Run the "xdc" command as follows:

```
%xdc
```

6.4 Rebuilding Target-Side UIA Modules

Pre-built UIA libraries are supplied, but you can rebuild individual modules using the "xdc" command provided by XDCtools. To rebuild a module, follow these steps:

1) Copy and rename the supplied default config.bld file using the following command:

```
%cp <uia_install>/uia_1_##_##_##/etc/config.bld.default
<uia_install>/uia_1_##_##_#/packages/config.bld
```

2) Edit the config.bld file.

3) Set the rootDir in config.bld to point to your installation of the CodeGen tools.

4) In the Build.target array, comment out any build targets you do not want to use. Then save the file.

5) Make sure that your XDCPATH environment variable is set correctly. For example, to build the evm6472 or simTCI6616 examples, the following XDCPATH is needed. (Specify the actual locations of your IPC, SYS/BIOS, and UIA installations instead of using # for version numbers.)

```
XDCPATH=
C:/Program Files/Texas Instruments/ipc_1_##_##_##/packages;
C:/Program Files/Texas Instruments/bios_6_##_##_##/packages;
C:/Program Files/Texas Instruments/uia_1_##_##_##/packages;
%NDK_INSTALL_DIR%/packages;
C:/Program Files/Texas Instruments/pdk_c6616_1_#_#_#/packages
```

where

```
NDK_INSTALL_DIR=
C:/Program Files/Texas Instruments/ndk_2_##_##_##
```

6) Change directory to the desired directory. For example,

```
%cd <uia_install>/packages/ti/uia/runtime
```

7) Run the "xdc" command as follows:

```
%xdc
```

6.5 Benchmarks

Benchmark testing was performed for a SYS/BIOS application with UIA load and event logging on the EVMTI816x and the EVM6472.

6.5.1 UIA Benchmarks for EVM6472

The benchmark test loads each core to 50% with application code. The ti.sysbios.utils.Load module records both Task and CPU loads. All communication to the instrumentation host is done by core 0 via the NDK.

Each UDP holds about 60 Log event records. For the test, both L1D and L1P caches were enabled. Code was placed in SL2, and data in LL2. The program was built for whole_program_debug.

The application calls Log_print() twice every millisecond. The CPU and Task load information is collected every 500ms. Rta send events every 100ms.

Table 6–1 Benchmarking Results for EVM6472, Tests 1 and 2

Only Core 0 sending events		**All cores sending events (via core 0)**	
# UDP packets:	36/sec	# UDP packets:	214/sec
Avg Packet Size:	1250 bytes	Avg Packet Size :	1250 bytes
App load (core 0) :	50.1%	App load (core 0):	50.0 %
Idle (core 0):	49.8%	Idle (core 0):	49.2%
UIA (core 0):	>.1%	UIA (core 0):	.5%
Hwi/Swi/Ndk/misc	.1%	Hwi/Swi/Ndk/misc	.3%
		Results for other cores:	
		Application load:	50.1%
		Idle:	49.8%
		UIA (core n):	>.1%
		Hwi/Swi/misc	.1%

6.5.2 UIA Benchmarks for EVMTI816x

The simpleTask.c EVMTI816x example, which is provided in the <*uia_install*>\packages\ti\uia\examples\evmti816x directory, has a Task thread on each core (DSP, Video, VPSS) that runs an algorithm to flip the bits on a large buffer every millisecond. The Task is released by a Clock function that posts a semaphore. The example logs a UIABenchmark_start/UIABenchmark_stop event before and after running the algorithm.

For the first test, load events were logged every 500 ms, and UIABenchmark events were performed around the algorithm. The number of UDP packets/second was 126, and the number of events/second was around 8000. The CPU loads obtained from these events and "top" on Linux are provided in the following table.

Table 6–2 Benchmarking Results for EVMTI816x, Test 1

	Hwi	**Swi**	**App Task**	**UIA Task**	**Idle**		
DSP	.1	.12	.3	.1	99.39		
Video	.32	.33	3.62	.22	95.31		
Vpss	.31	.36	3.6	.22	95.52		
	usr	**sys**	**nic**	**io**	**idle**	**irq**	**sirq**
Arm	0	0	0	0	98	0	0

For the second test, event logging for Hwi, Swi, and Task threads was added. Loads were again logged every 500 ms, and UIABenchmark events were performed. The number of UDP packets/second was 1055, and the number of events/second was around 51,000.

Table 6–3 Benchmarking Results for EVMTI816x, Test 2

	Hwi	**Swi**	**App Task**	**UIA Task**	**Idle**		
DSP	.36	.38	.33	.8	98.25		
Video	1.05	1.09	3.89	1.58	92.43		
Vpss	.97	1.1	3.89	1.52	92.52		
	usr	sys	nic	io	idle	irq	sirq
Arm	0	9	0	0	88	0	1

Index