

LoggerIdle Uart

From Texas Instruments Wiki

Jump to: [navigation](#), [search](#)

Contents

[hide]

- [1 Introduction](#)
- [2 LoggerIdle](#)
- [3 LoggerIdle Example](#)
 - [3.1 Example Configuration](#)
 - [3.2 Example Transport Function](#)
 - [3.3 Example Task](#)
 - [3.4 Example Exception Hook Function](#)
 - [3.5 Example Output](#)
 - [3.6 Example Source Code](#)
- [4 Using a Newer UIA Product with your CCS Project](#)

Introduction[[edit](#)]

Unified Instrumentation Architecture (UIA) provides target-side APIs and transports for capturing events and uploading them to a host program, such as Code Composer Studio's (CCS) System Analyzer. In some cases, it may not be possible to have CCS connected to the target, but we would still like to see Log data displayed on a host computer. In this article, we will show how to use UIA's LoggerIdle logger to capture Log data from the target and send it to the UART, where it can be viewed on the host computer in a serial port console window. The example we give will run for awhile, logging data to the UART, and then cause a divide-by-zero exception. The exception handler will then dump the remaining Log data in the LoggerIdle buffer to the UART.

The advantage of using LoggerIdle, is that we can format the Log data during idle time instead of during the Log_write call. We could have used xdc.runtime.LoggerSys which formats the Log data, together with xdc.runtime.SysMin. We would then provide an output function for SysMin which writes the data to the UART. However, we would need to call System_flush() to call our output function, and System_flush() disables interrupts. Since the output function should poll on UART writes instead of blocking (since we don't know from what context System_flush() would be called), interrupts could be disabled for a very long time.

LoggerIdle[[edit](#)]

UIA's `LoggerIdle` is a logger that processes log events as they are generated, stores them in a buffer and during idle time, sends a section of the buffer to a user configured transport function.

`LoggerIdle` internally handles the wrapping of the buffer where the Log events are written, making it a "circular" buffer. A write pointer indicates the location in the buffer where the next Log event can be written, and a read pointer indicates the location of the next Log event to be sent to the user's transport function. Log write calls cause the write pointer to advance, and when Log data is passed to the user's transport function in the idle loop, the read pointer advances. If the read pointer catches up the the write pointer, the buffer is 'empty', and if the write pointer catches up the the read pointer, the buffer is full.

The `LoggerIdle` buffer will fill up, if the idle function to output the Log data cannot keep up with the Log writes. `LoggerIdle` has a Boolean configuration parameter that controls the behavior of Log writes when the buffer fills.

- Do not allow further writes to the `LoggerIdle` buffer when it is full.
- Adjust the `LoggerIdle` read pointer until there is space for the event to be written.

There are advantages and disadvantages of each of these options, but one may be preferable over the other, depending on the situation. If we don't allow Log writes when the buffer is full, any Log writes made while the buffer is full will be lost. However, this may not be an issue if the user's transport function runs frequently enough. There is also some overhead in allowing Log writes when the buffer is full. Since Log event sizes vary, there is overhead in determining how much the read pointer must be adjusted to fit the new Log event. In addition, only one event at a time can be output to the user's transport function. The following table summarizes the advantages of each configuration, and when you might want to use it.

Allow writes when full	Advantages	When to use
true	Always get the most recent Log data	Use when you want the most recent Log data, for example to analyze a system crash
false	Less overhead in getting Log data to the transport function	Idle time is sufficient to get the Log data out

For our example, we will enable Log writes when the LoggerIdle buffer is full, since we want to see the most recent Log data up to the exception on the target.

LoggerIdle Example[\[edit\]](#)

We provide an example here for the Stellaris LM4F232 device, that illustrates how to send formatted Log events to the UART, and how to configure a SYS/BIOS exception hook function. Note that the exception handling mechanisms are only available for some targets (e.g., Cortex M3 and M4).

Our example will also route System_printf()'s to the UART. The System_printf() messages will only be seen after a call to System_flush(), an operation that should be used sparingly, as interrupts are disabled during this call.

Example Configuration[\[edit\]](#)

LoggerIdle has the following configuration parameters that we are concerned about:

```
<syntaxhighlight lang='javascript'> config SizeT bufferSize = 256; /* LoggerIdle buffer size in MAUS */ config Bool isTimestampEnabled = true; /* Enable/disable logging of 64-bit with event */ config LoggerFxn transportFxn = null; /* User's function for transmitting Log records */ config Bool writeWhenFull = false; /* Enable/disable Log writes when buffer is full */ </syntaxhighlight>
```

We will leave the buffer size as is, keep timestamps enabled, and enable Log writes when the buffer is full. The transport function will be set to loggerIdleSend(), our example function to format the Log data and send it out the UART. The configuration code for LoggerIdle is the following:

```
<syntaxhighlight lang='javascript'> var LoggerIdle =  
xdc.useModule('ti.uia.sysbios.LoggerIdle'); LoggerIdle.bufferSize = 256;  
LoggerIdle.transportFxn = "&loggerIdleSend"; LoggerIdle.isTimestampEnabled = true;  
LoggerIdle.writeWhenFull = true; </syntaxhighlight>
```

Since we want System_printf() messages to go to the UART, we will configure xdc.runtime.System as follows:

```
<syntaxhighlight lang='javascript'> var System = xdc.useModule('xdc.runtime.System');  
  
var SysMin = xdc.useModule('xdc.runtime.SysMin'); SysMin.bufSize = 0x400;  
SysMin.outputFxn = '&log2Uart'; System.SupportProxy = SysMin; </syntaxhighlight>
```

Our SysMin output function, log2Uart(), will also be used by loggerIdleSend() for sending out the formatted Log records.

The configuration code for enabling SYS/BIOS exception handling and plugging our exception hook function is the following:

```
<syntaxhighlight lang='javascript'> var M3Hwi =  
xdc.useModule('ti.sysbios.family.arm.m3.Hwi'); M3Hwi.enableException = true;  
M3Hwi.nvicCCR.UNALIGN_TRP = 1; M3Hwi.nvicCCR.DIV_0_TRP = 1;  
M3Hwi.excHookFunc = '&myExceptionHook'; </syntaxhighlight>
```

Example Transport Function[\[edit\]](#)

The function prototype of the LoggerIdle transport function is:

```
<syntaxhighlight lang='c'>  
Int transportFxn(UChar *buffer, Int size); </syntaxhighlight>
```

The buffer parameter contains the raw Log data and the size is the amount of data in target MAUs. The function should return the number of MAUs actually transferred. Since we don't want to view raw Log data on a COM port console window, our transfer function will format the data.

We use a helper function, log2uart(), which will write *formatted* Log data and System_printf() messages to the UART. Since we are sending data to the UART in the idle loop, we don't want to block, so we use **UART_writePolling()** to write to the UART. UART_writePolling comes with the TI-RTOS drivers library. The log2uart() function code is basically the following (but please refer to the actual attached example for more details and comments):

```
<syntaxhighlight lang='c'> Void log2Uart(Char *buf, Int size) {  
  
    UART_Handle uart;  
    Int nBytes = 0;  
    Int bytesLeft = size;  
    Int bytesToWrite;  
    if (size > 0) {  
        uart = ports[fd].handle; /* A previously opened UART handle */  
        /* Write to the UART 16 bytes at a time, until the transfer is  
finished. */  
        while (bytesLeft > 0) {  
            bytesToWrite = (bytesLeft < 16) ? bytesLeft : 16;  
            nBytes = UART_writePolling(uart, (Char *)buf, bytesToWrite);  
            buf += nBytes;  
            bytesLeft -= nBytes;  
        }  
    }  
}  
</syntaxhighlight>
```

Formatting the Log data before sending it to the UART sounds like a difficult task, but fortunately, we can make use of the xdc.runtime.Log function, Log_doPrint(), and modify it for our needs. Our Log print function, MyLog_doPrint(), replaces calls to System_printf() with System_sprintf(), so that the formatted Log is placed in a buffer.

The buffer is then passed to `log2uart()`. Our transport function, `loggerIdleSend()`, loops through the Log data and fills in a `Log_EventRec` structure to pass to `MyLog_doPrint()`. It uses the `UIA EvnetHdr` macros to determine whether or not the Log record is timestamped, and the length and sequence number of the Log records. This information is used to fill in the `Log_EventRec` structure.

As the code for the transport function and `MyLog_doPrint()` is not listed here, please download the example to see these functions.

Example Task[\[edit\]](#)

The example has one task (other than the idle task), that seemingly loops forever, sleeping and printing messages. The sleep time allows the idle task to run and upload Log data to the UART. After a point, the task causes a divide-by-zero exception, causing the SYS/BIOS exception handler to run and the program to abort. Here is the task code:

```
<syntaxhighlight lang='c'> Void taskFxn(UArg arg0, UArg arg1) {

    Int    sleepDur = 100;
    Int    id = (Int)arg0;
    Int    count = 40;
    while (TRUE) {
        Task_sleep(sleepDur);
        /* Benchmark time to do a Log_print2 */
        t1 = Timestamp_get32();
        Log_print4(Diags_USER1, "Task %d awake, count = %d, %d, %d", id,
count, 0xabcd, 0x1234);
        t2 = Timestamp_get32();
        System_printf("Time to do Log_print2: %d, count: %d\n", t2 - t1,
count);
        /* Decrement count so that eventually we will get an exception
        */
        count--;
        sleepDur = (sleepDur / count) * count;
    }

} </syntaxhighlight>
```

Example Exception Hook Function[\[edit\]](#)

Since Log data is only output during idle time, we won't get the most recent Logs that occurred just before the divide-by-zero exception. To handle this situation, `LoggerIdle` has a flush function which we can call in our exception hook. `LoggerIdle_flush()` will dump whatever Log records remain in the `LoggerIdle` buffer. Since the `LoggerIdle` idle write function may have been in the middle of outputting Logs when it was last preempted, calling `LoggerIdle_flush()` may show duplicate Log records (if `LoggerIdle.writeWhenFull = false`), or a missing record (if `LoggerIdle.writeWhenFull = true`). For that reason, `LoggerIdle_flush()` is not intended for general use. Below is the code for the example exception handler.

```
<syntaxhighlight lang='c'> Void  
myExceptionHook(ti_sysbios_family_arm_m3_Hwi_ExcContext *excp) {  
  
    System_printf("Exception context = 0x%x\n", excp);  
    LoggerIdle_flush();  
  
} </syntaxhighlight>
```

Example Output[\[edit\]](#)

The screen shot below shows the COM port console with the Log and System messages from the example. Note that because we called `LoggerIdle_flush()` in our exception hook function, we get the last Log message before the exception occurred, showing the count value of 1. Also note that System messages are output after the Log messages, even though they were made earlier. This is because System messages will not be displayed until `System_flush()` is called, in this case by the exception handler, after calling our exception hook function.

```
COM4 - PuTTY
#0000000240 [t=0x07f55cfa] ti.sysbios.knl.Task: LM_sleep: tsk: 0x2000461c, func:
0x4391, timeout: 6
#0000000241 [t=0x07f5607a] ti.sysbios.knl.Task: LM_switch: oldtsk: 0x2000461c, o
ldfunc: 0x4391, newtsk: 0x20004668, newfunc: 0x86e9
#0000000242 [t=0x07fc8168] ti.sysbios.knl.Task: LD_ready: tsk: 0x2000461c, func:
0x4391, pri: 2
#0000000243 [t=0x07fc85ce] ti.sysbios.knl.Task: LM_switch: oldtsk: 0x20004668, o
ldfunc: 0x86e9, newtsk: 0x2000461c, newfunc: 0x4391
#0000000244 [t=0x07fc89a8] xdc.runtime.Main: Task 1 awake, count = 1, 43981, 466
0
#0000000245 [t=0x07fcab8c] ti.sysbios.family.arm.m3.Hwi: ERROR: line 942: E_hard
Fault: FORCED
#0000000246 [t=0x07fceafa] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1054: E_usa
geFault: DIVBYZERO
o do Log_print2: 598, count: 5
Time to do Log_print2: 598, count: 4
Time to do Log_print2: 598, count: 3
Time to do Log_print2: 598, count: 2
Time to do Log_print2: 598, count: 1
ti.sysbios.family.arm.m3.Hwi: line 942: E_hardFault: FORCED
ti.sysbios.family.arm.m3.Hwi: line 1054: E_usageFault: DIVBYZERO
Exception occurred in background thread at PC = 0x00004426.
Core 0: Exception occurred in ThreadType_Task.
Task name: {unknown-instance-name}, handle: 0x2000461c.
Task stack base: 0x20000360.
Task stack size: 0x800.
R0 = 0x00000000 R8 = 0xffffffff
R1 = 0x00000006 R9 = 0xffffffff
R2 = 0x20000b04 R10 = 0xffffffff
R3 = 0x00000000 R11 = 0xffffffff
R4 = 0x00006031 R12 = 0x00000090
R5 = 0xffffffff SP(R13) = 0x20000b08
R6 = 0xffffffff LR(R14) = 0x0000441d
R7 = 0xffffffff PC(R15) = 0x00004426
PSR = 0x61000000
ICSR = 0x00415803
MMFSR = 0x00
BFSR = 0x00
UFSR = 0x0200
HFSR = 0x40000000
DFSR = 0x0000000a
MMAR = 0xe000ed34
BFAR = 0xe000ed38
AFSR = 0x00000000
Exception context = 0x20000a30
Terminating execution...
```

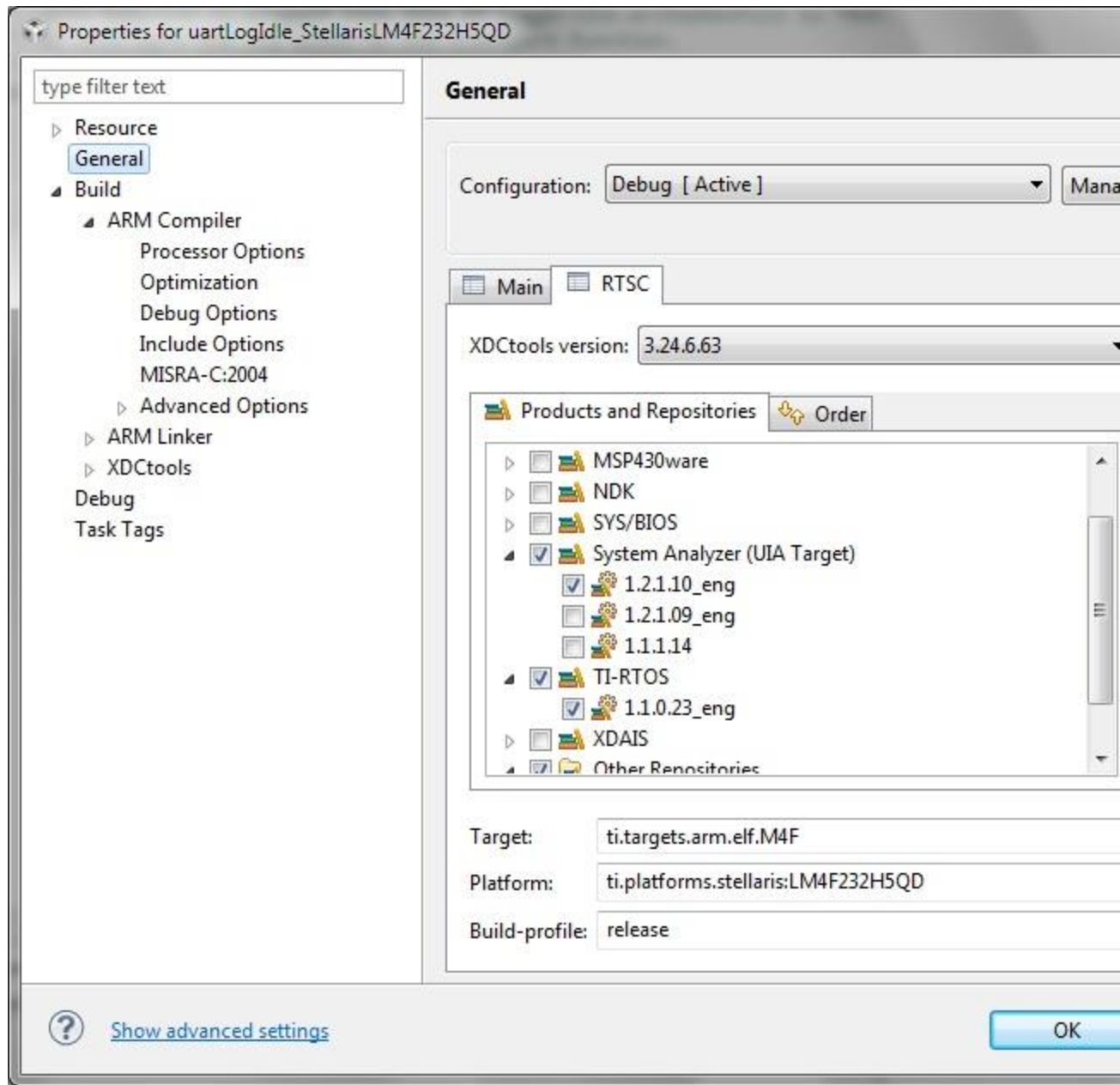
Example Source Code[\[edit\]](#)

The example code can be downloaded here: [File:LoggerIdle UART example StellarisLM4F232H5QD.zip](#)

Using a Newer UIA Product with your CCS Project[\[edit\]](#)

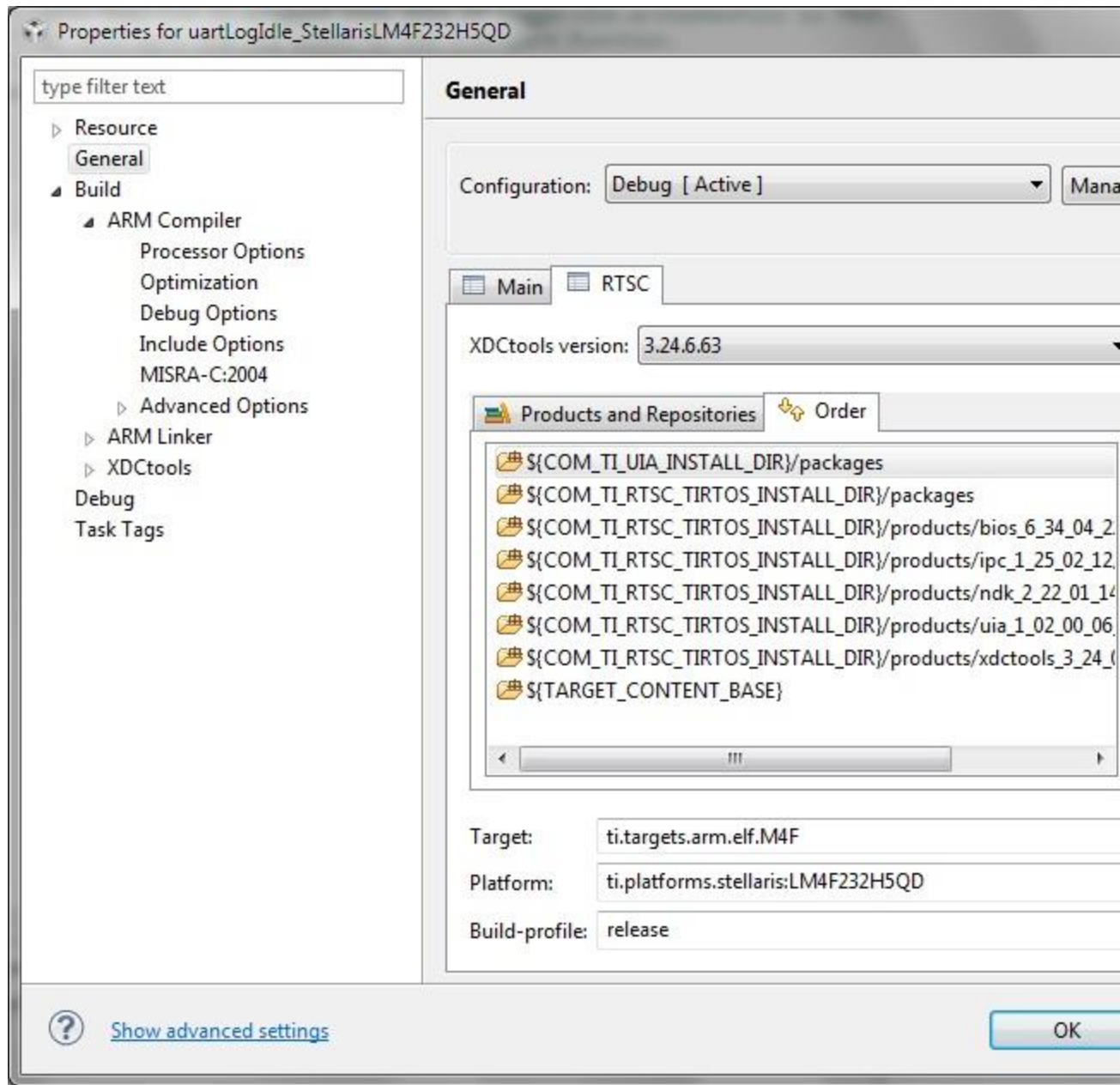
This example requires UIA version 1.02.01.11 or higher. If you are using a TI-RTOS product that includes UIA 1.02.01.11 or higher, you do not need to perform the steps described in this section. If your TI-RTOS product contains a UIA version that is older than 1.02.01.11, you will need to install a newer version of UIA. There are also a couple of steps you need to do to get your CCS project to build using the newly installed UIA product instead of with the one included in TI-RTOS:

- First, in the project settings, you need to enable UIA. See the screen shot below.



- Second, you need to change the search order of the repositories, so that the newer UIA is found first. Click

on the "Order" tab, select the UIA package, and click the "UP" button until UIA moves to the top. See the screen shot below.



{{

1. switchcategory:MultiCore=

- For technical support on MultiCore devices, please post your questions in the [C6000 MultiCore Forum](#)
- For questions related to the BIOS MultiCore SDK

Keystone=

- For technical support on MultiCore devices, please post your questions in the [C6000](#)

C200
techn
supp
the C
pleas
your
ques
on T
C200
Foru
Plea
only


(MCSDK), please use the [BIOS Forum](#)

Please post only comments related to the article **LoggerIdle Uart** here.

[MultiCore Forum](#)

- For questions related to the BIOS MultiCore SDK (MCSDK), please use the [BIOS Forum](#)

Please post only comments related to the article **LoggerIdle Uart** here.



[Amplifiers & Linear](#)
[Audio](#)
[Broadband](#)
[RF/IF &](#)
[Digital](#)
[Radio](#)
[Clocks &](#)
[Timers](#)
[Data](#)
[Converters](#)

[DLP & MEMS](#)
[High-Reliability](#)
[Interface](#)
[Logic](#)
[Power](#)
[Management](#)

[Processors](#)

- [ARM Processors](#)
- [Digital Signal Processors \(DSP\)](#)
- [Microcontrollers \(MCU\)](#)
- [OMAP Applications Processors](#)

[Switches & Multiplexers](#)
[Temperature](#)
[Sensors & Control ICs](#)
[Wireless](#)
[Connectivity](#)

Retrieved from "https://processors.wiki.ti.com/index.php?title=LoggerIdle_Uart&oldid=182528"

Navigation menu

Personal tools

- [Log in](#)
- [Request account](#)

Namespaces

- [Page](#)
- [Discussion](#)



Variants

Views

- [Read](#)
- [View source](#)
- [View history](#)



More

Search



Navigation

- [Main Page](#)
- [All pages](#)
- [All categories](#)
- [Recent changes](#)
- [Random page](#)
- [Help](#)

Toolbox

- [What links here](#)
- [Related changes](#)
- [Special pages](#)
- [Printable version](#)
- [Permanent link](#)
- [Page information](#)

- This page was last edited on 28 July 2014, at 06:34.
- Content is available under [Creative Commons Attribution-ShareAlike](#) unless otherwise noted.

- [Privacy policy](#)
- [About Texas Instruments Wiki](#)

- [Disclaimers](#)
- [Terms of Use](#)

