Application Note

DLT Developer's Guide With Tooling

Ryan Ma

ABSTRACT

The data logger and trace (DLT) peripheral is a key element for non-intrusive data logging with tracing support in application code. This peripheral can be leveraged for both industrial and automotive applications. C29x real-time microcontrollers offer a non-intrusive way to data log critical CPU run time content and provide trace capabilities without additional CPU overhead. This application note focuses on an application use-case described within the introduction, demonstrating each of the features of the DLT along with how to use the SysConfig system configuration tool to set-up and program the DLT. There is additional material on how to setup the DLT visualization tool needed to interpret the data logging and trace contents without the use of external hardware. SysConfig is a tool that exists integrated in Code Composer Studio or as a stand alone program that allows users to generate C header and code files using a graphical user interface (GUI). This application note was done using the F29H859TU8ZEXQL device. However, the content in this application note is applicable to all devices with the DLT peripheral.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

Why is data logging and trace important in real time control systems?

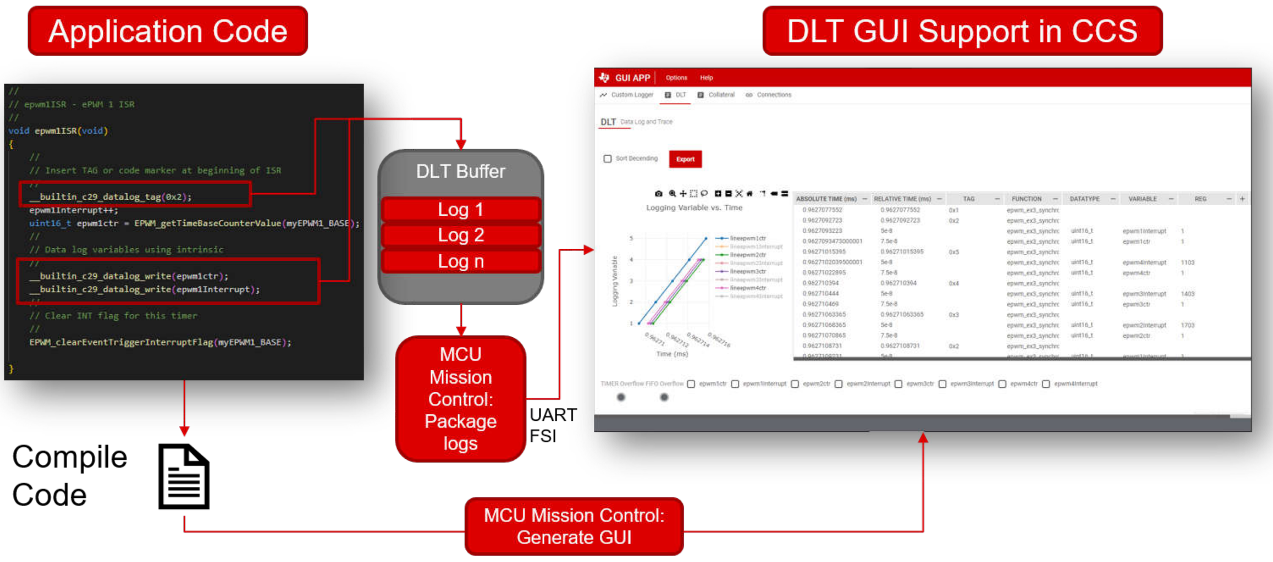
- Debugging application code during and after development or testing
- Profiling application code
- Creating capture logs of a control system for deep analysis
- Tracing application code flow

There are many applications that require different uses of using data log or trace features on a device. The DLT provides a way to log critical run time content, and then export out by JTAG, UART, or FSI. If there is no JTAG connection available, the DLT can still be used if there is a UART or FSI implementation to export the data. The lines of code controlling what is being logged can be kept in the application code without impact on CPU performance.

The DLT provides data logging and code flow execution through dedicated instructions provided in the C29x user guide. When using the DLT for data logging variables or adding code flow markers in the application there is additional information attached to each log. The additional information is dependent on what mode the DLT is capturing the logs. The two modes for the additional information added to each log are time stamping or program counter information. In time stamping mode, the DLT provides information of when variables or code markers are being logged. In program counter mode, the DLT provides information to know where these logs are happening.

There are dedicated instructions used to log information. The instructions have the following names DLTAG and DLREG. DLTAGs are used as the code flow markers. DLREGs are the instructions that allow users to data log variables. With the leverage of multiple instructions running in parallel using the C29x processor these instructions can run in parallel and provide non-intrusive behavior when data logging or adding code flow markers to the application code. The co-processor interface (CPI) looks for these dedicated instructions from the CPU and provides the packet information to the DLT which gets logged to a dedicated memory address region for each CPU. The CPU or DMA can read out the logs from the DLT internal memory and move the logged information as needed.

Here is a high level view from going to application code to visualizing the data being logged by the DLT on a PC.



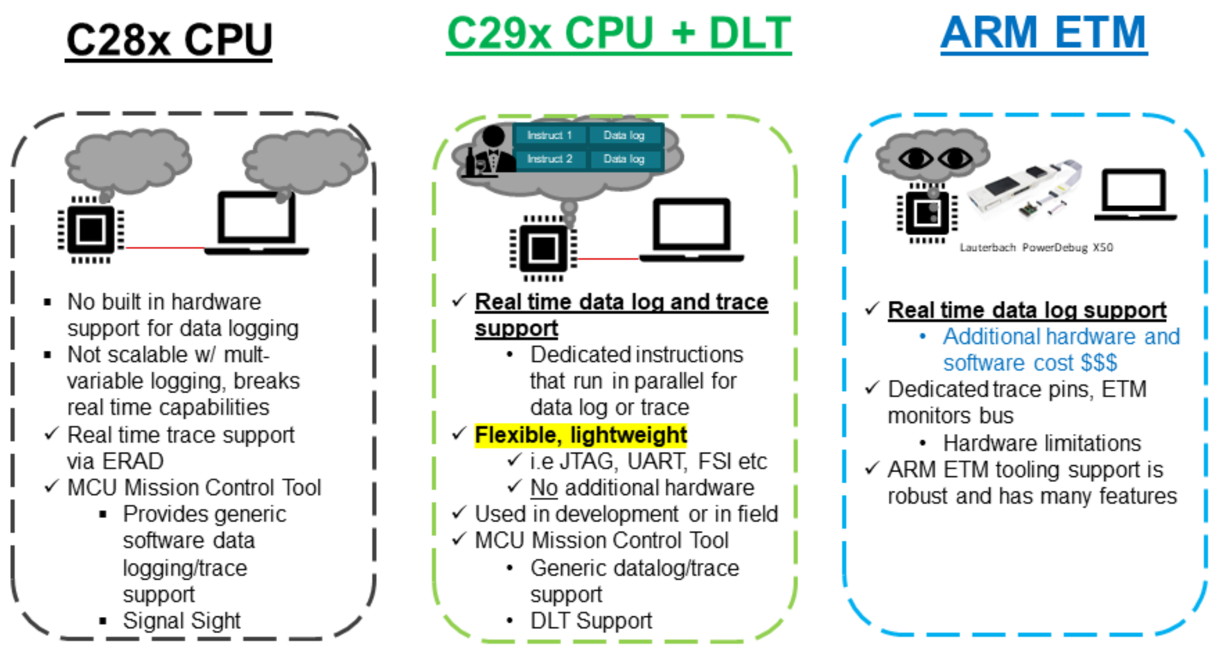
```
//
// epwm1ISR - ePWM 1 ISR
//
void epwm1ISR(void)
{
    //
    // Insert TAG or code marker at beginning of ISR
    //
    __builtin_c29_datalog_tag(0x2);
    epwm1Interrupt++;
    uint16_t epwm1ctr = EPWM_getTimeBaseCounterValue(myEPWM1_BASE);
    //
    // Data log variables using intrinsic
    //
    __builtin_c29_datalog_write(epwm1ctr);
    __builtin_c29_datalog_write(epwm1Interrupt);
    //
    // Clear INT flag for this timer
    //
    EPWM_clearEventTriggerInterruptFlag(myEPWM1_BASE);

}
```

Figure 1-1. DLT Example Code Snippet

This application note explains the steps needed to configure the DLT, data log, and add code markers using the built in compiler intrinsic. The DLT leverages the compiler and SysConfig to provide the fastest way to get started with using this peripheral.

The use-case that is discussed throughout this application report is using the DLT to record a temperature sensor sample and ADC results within ISR.

2 C28x vs C29x vs ARM Logging



C28x CPU

- No built in hardware support for data logging
- Not scalable w/ mult-variable logging, breaks real time capabilities
- ✓ Real time trace support via ERAD
- ✓ MCU Mission Control Tool
  - Provides generic software data logging/trace support
  - Signal Sight

C29x CPU + DLT

- ✓ **Real time data log and trace support**
  - Dedicated instructions that run in parallel for data log or trace
- ✓ **Flexible, lightweight**
  - ✓ i.e JTAG, UART, FSI etc
  - ✓ No additional hardware
- ✓ Used in development or in field
- ✓ MCU Mission Control Tool
  - Generic datalog/trace support
  - DLT Support

ARM ETM

- ✓ **Real time data log support**
  - Additional hardware and software cost $$$
- ✓ Dedicated trace pins, ETM monitors bus
  - Hardware limitations
- ✓ ARM ETM tooling support is robust and has many features

Figure 2-1. C28x vs C29x vs ARM Logging

3 SysConfig

As mentioned in the abstract, this application note utilizes SysConfig to configure the DLT peripheral along with the other required components such as visualizing the DLT packets. The SysConfig GUI allows users to walk through the configurations needed to set up the DLT to be ready to start logging in the application code.

3.1 Start or Stop Logging

The first step to configuring the DLT is setting up the start and stop events to log information. There are two main methods of starting or stopping logged information through the code markers (DLTAGs) or using ERAD events. The third method is security focused and prevents logged information to occur based on the enabled LINK filters. The TAG or ERAD filters control when to start or stop data logging. LINK filters are used to filter out sections of code to data log based on what current LINK has access.

The TAG based filtering has a few options such as the *Start Tag Reference*, *Start Tag Mask*, and the end tag configurations. These can be used to start or stop the DLT from logging. The mask is and'ed with the code marker that is reached in the application code. If the and'ed output of the current tag value and the tag filter start mask equals the start tag reference and the tag based filtering is enabled, then the DLT starts logging. The same is true to stop the DLT from logging using the end tags. The start/end tag reference can be any 16-bit value.

Below is a flow chart diagram that explains what happens if users have both TAG based and ERAD based conditions to start or stop logging.

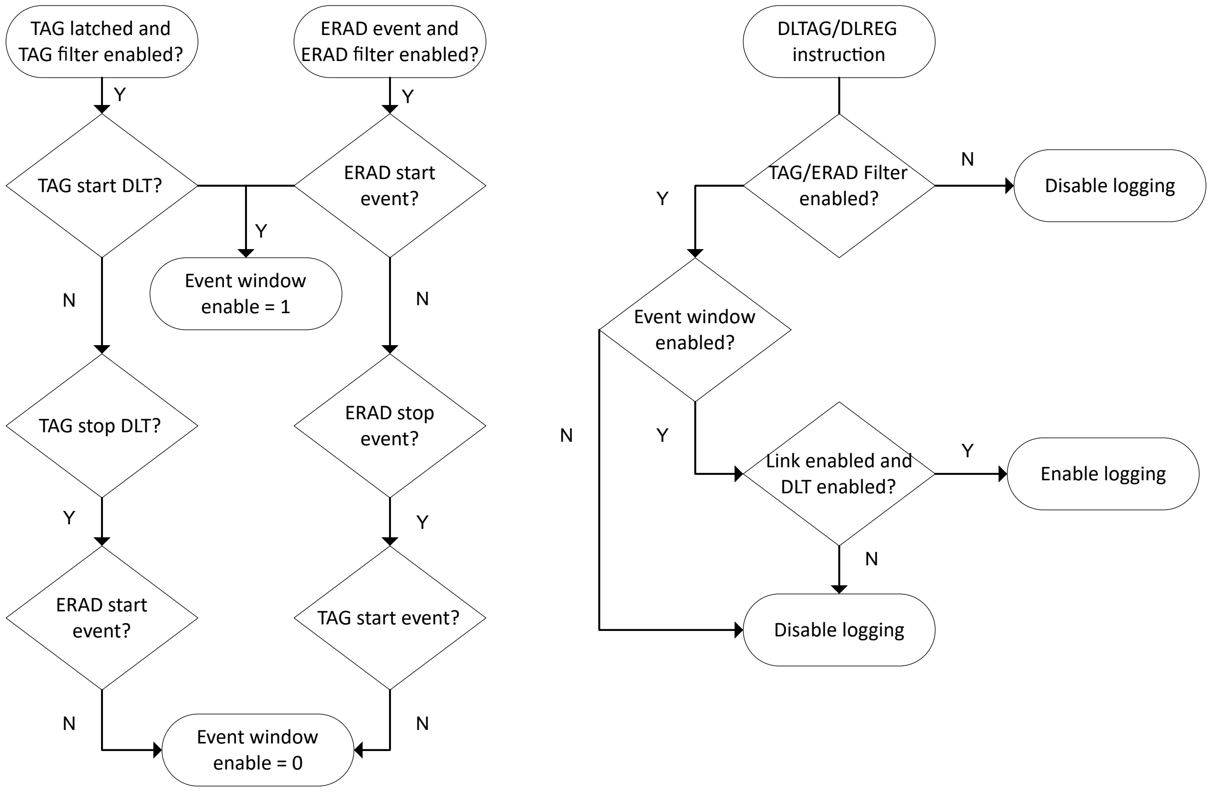


Figure 3-1. Start and Stop Events Decision Chart

Configuring the start or stop end tags can be any 16-bit value. SysConfig offers a helpful GUI to configure these.

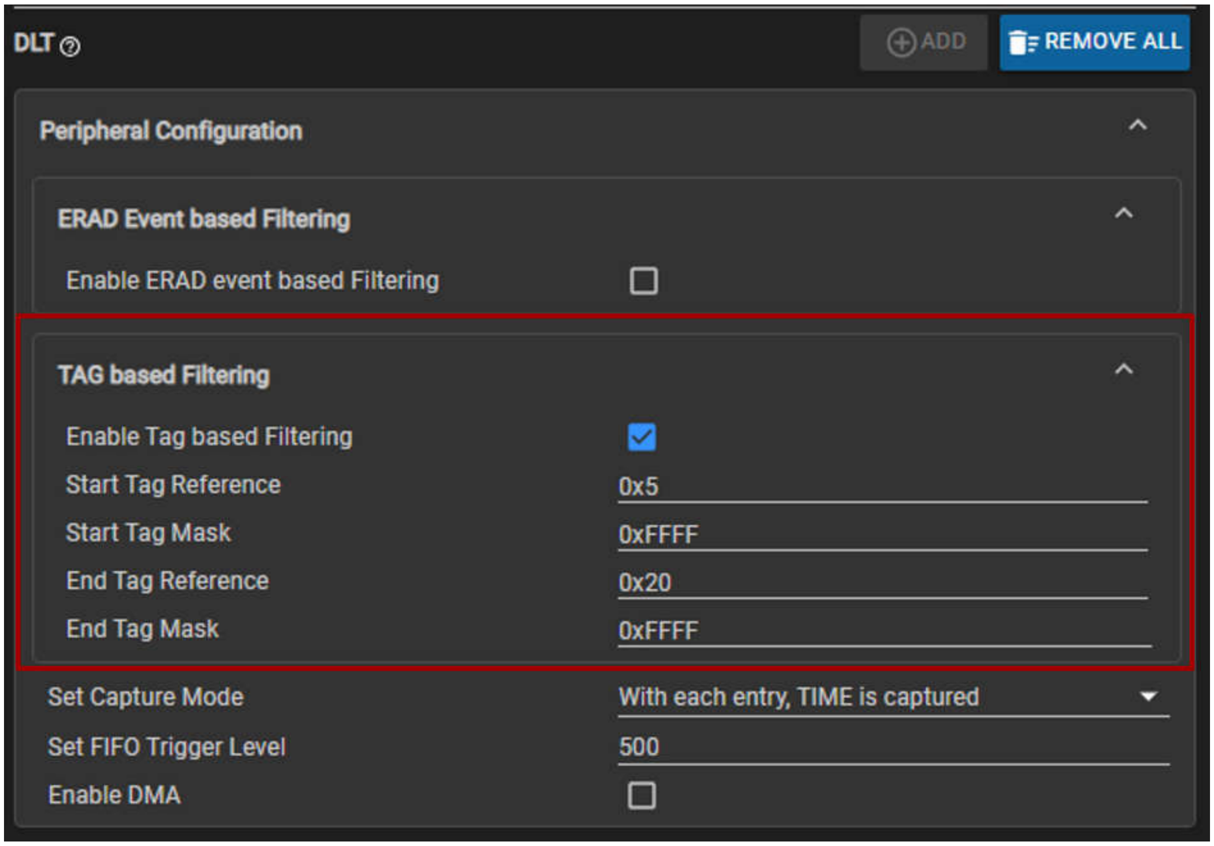

DLT
ADD REMOVE ALL
Peripheral Configuration
ERAD Event based Filtering
Enable ERAD event based Filtering ☐
TAG based Filtering
Enable Tag based Filtering ✓
Start Tag Reference 0x5
Start Tag Mask 0xFFFF
End Tag Reference 0x20
End Tag Mask 0xFFFF
Set Capture Mode With each entry, TIME is captured
Set FIFO Trigger Level 500
Enable DMA ☐

Figure 3-2. DLT Tag Filter SysConfig

The *Enable Tag based Filtering* option must be selected. The configurations outlined to configure the start tag reference value and end tag reference value. The *Start Tag Mask* and *End Tag Mask* get AND'ed with the respective start or end reference values to control when the logs occur. In the above configuration, 0x5 is the start tag reference value being used and the end tag reference value of 0x20. These values are arbitrary and can be configured with any value. The code generated in SysConfig's board.c file is shown below.

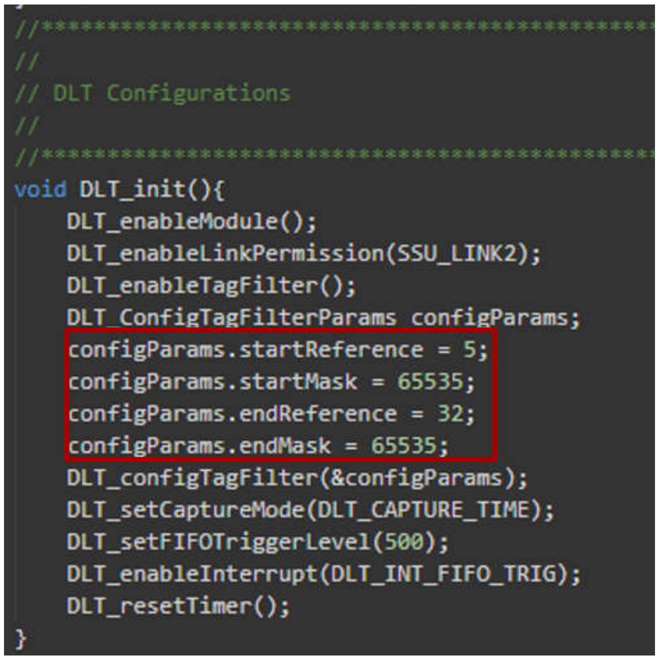

```
//**************************************************
//
// DLT Configurations
//
//**************************************************
void DLT_init(){
    DLT_enableModule();
    DLT_enableLinkPermission(SSU_LINK2);
    DLT_enableTagFilter();
    DLT_ConfigTagFilterParams configParams;
    configParams.startReference = 5;
    configParams.startMask = 65535;
    configParams.endReference = 32;
    configParams.endMask = 65535;
    DLT_configTagFilter(&configParams);
    DLT_setCaptureMode(DLT_CAPTURE_TIME);
    DLT_setFIFOTriggerLevel(500);
    DLT_enableInterrupt(DLT_INT_FIFO_TRIG);
    DLT_resetTimer();
}
```

Figure 3-3. DLT Initialization Snippet

3.2 Capture Modes

There are two independent modes that can be enabled with the DLT. Each capture mode has a specific format of how the additional information with each log appears in the DLT internal memory. This internal memory acts as a FIFO and is used to store the logs of both code markers and variables. Please refer to the “Interpreting DLT logs” section to understand more about how these logs look between both capture modes.

The first mode is time capture mode where each log contains information of when the log was reached. In this mode, the source of the timers is from IPC counter and DLT’s internal counter. Code markers (DLTAGs) use the IPC as the source and data logged variables (DLREGs) use the DLT’s internal counter as the source. For code markers (DLTAGs), the timer value is called TIMER1 and is going to be sourced from the IPC counter. This time is the absolute time of when the IPC starts. For data logging variables (DLREGs), the timer value is called TIMER2 and is going to be sourced from DLT’s internal counter. This timer value is always the time reference between the previous code marker (DLTAG) that was reached. This can also be thought of relative time between the previous code marker.

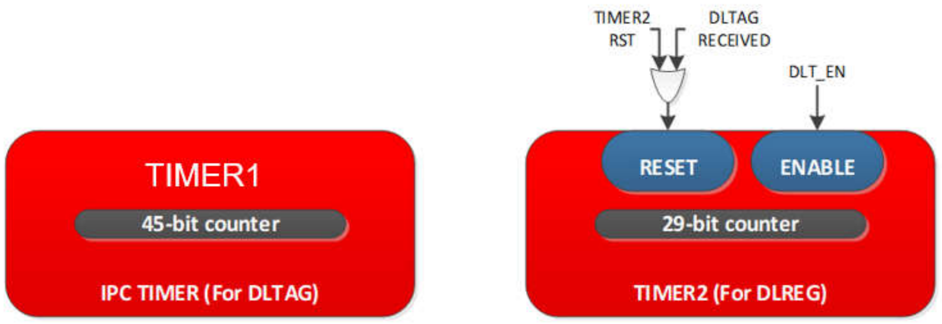
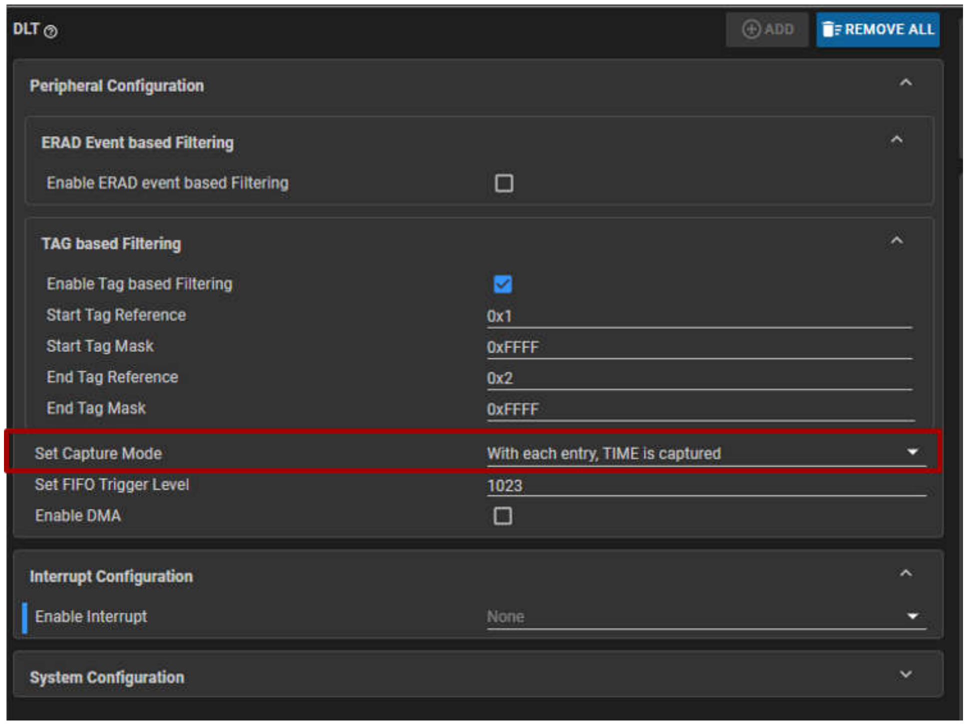


Figure 3-4. TIMER1(DLTAG) vs TIMER2(DLREG)

The second mode is program counter mode where each log contains information of where the log was reached instead of a timer value. SysConfig provides a way to configure the DLT in either mode at initialization.



Peripheral Configuration

ERAD Event based Filtering

Enable ERAD event based Filtering	☐

TAG based Filtering

Enable Tag based Filtering	✓
Start Tag Reference	0x1
Start Tag Mask	0xFFFF
End Tag Reference	0x2
End Tag Mask	0xFFFF
Set Capture Mode	With each entry, TIME is captured
Set FIFO Trigger Level	1023
Enable DMA	☐

Interrupt Configuration

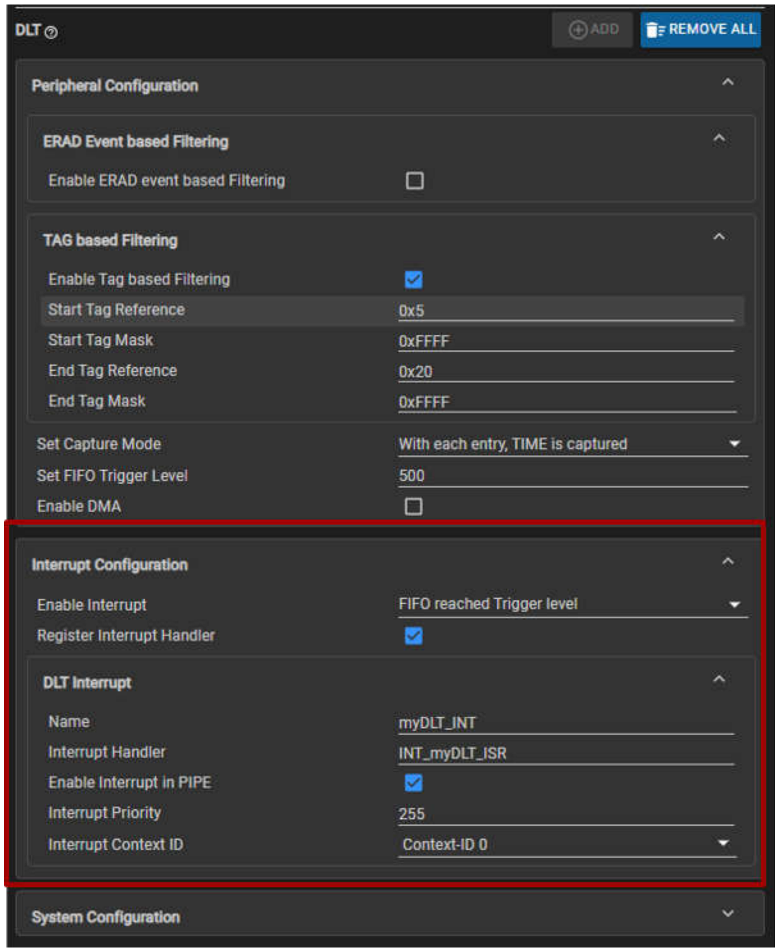
Enable Interrupt	None

System Configuration

Figure 3-5. DLT SysConfig - Capture Mode

3.3 Trigger for transferring Logs

There are a few ways to read data from the DLT buffers and write the logs to another location on the device using the CPU or DMA interface. Once the data has been moved, the logs can then be transferred out using any communication peripheral on the device. The DLT has a way to trigger a CPU interrupt or DMA transfer request based on how many logs are in the buffer. Please refer to the FIFO_CONTROL.WR_CTR_TRIG_LEVEL to know what the max trigger level that can be set. The below configuration generates a DLT interrupt when the FIFO reaches 500 logged elements.



Peripheral Configuration

ERAD Event based Filtering

Enable ERAD event based Filtering	☐

TAG based Filtering

Enable Tag based Filtering	✓
Start Tag Reference	0x5
Start Tag Mask	0xFFFF
End Tag Reference	0x20
End Tag Mask	0xFFFF
Set Capture Mode	With each entry, TIME is captured
Set FIFO Trigger Level	500
Enable DMA	☐

Interrupt Configuration

Enable Interrupt	FIFO reached Trigger level
Register Interrupt Handler	✓

DLT Interrupt

Name	myDLT_INT
Interrupt Handler	INT_myDLT_ISR
Enable Interrupt in PIPE	✓
Interrupt Priority	255
Interrupt Context ID	Context-ID 0

System Configuration

Figure 3-6. DLT SysConfig - Trigger FIFO level and DMA

4 Interpreting DLT Logs

The previous sections have described setting up on how to start and end logs, what different modes there are and how to trigger a transfer of the logs to another location based on the trigger level of the DLT. How do users interpret the logs from the internal memory?

There are two terminologies, code markers and variables being data logged. Each of these have a way to interpret the logs from the nternal memory of the DLT. The code markers or DLTAGs must be interpreted slightly differently than the variables being data logged or DLREGs.

The internal memory where the logs are being stored are at the base address of the DLT_FIFO_REGS. The internal memory works like a FIFO. When reading from the FIFO_BUF_H, the next log is pushed up. Thus the order in reading from the FIFO matters. The way to read from this memory mapped register is to read the FIFO_BUF_L contents first, then read the FIFO_BUF_H contents next.

Depending on the capture mode, interpreting the log is different. When reading from the FIFO, the first item to look for is the LSB of the lower 32 bits. This provides information on whether this is a code marker (DLTAG) or a logged variable (DLREG). The rest of the information can be decoded through the tables described in the technical reference manual. For this application note, the DLT tool is used to interpret DLT information when the mode is set to capture the time values.

5 Compiler Intrinsic

The dedicated instructions from the C29x user guide can be abstracted by using the built-in compiler intrinsic provided in the ti-cgt-c29 compiler. Under the hood of using the built-in compiler intrinsic uses dedicated DLT instructions.

For code markers use: __builtin_c29_datalog_tag(TAG);

- This intrinsic expects any 16-bit tag value

For data logging variables use: __builtin_c29_datalog_write(VAR);

- This intrinsic expects any 32-bit variable

Note

Every new function scope must have at least one code marker to denote the new function scope then followed by the variables to be data logged. Only having code markers is acceptable. TI does not recommended to have no code markers in a function scope followed by variables to be data logged.

6 DLT Tool

Reading and interpreting the DLT logs is challenging since there are lots of bits to track and make sure are correct when reconstructing the final packaged logged information.

The DLT Tool has been created to pass this hurdle.

6.1 Visualization

The DLT tool has two communication peripheral designs to export DLT logs out of the device. This application note focuses on a UART-based design to export the DLT FIFO out of a device and visualizes the data on a COM port or through a GUI interface like GUI composer. The other communication peripheral is FSI that can be used with the tool. This requires a FSI to UART-USB bridge device. This application note uses an example found within the F29x MCU SDK under the following folder: C:\ti\c2000\mcu_sdk_f29h85x_x_xx_xx_xx_xxx\examples\driverlib\single_core\transfer\transfer_adc_tempsensor_dlt . The following are detailed steps on how to add the DLT tool to any project.

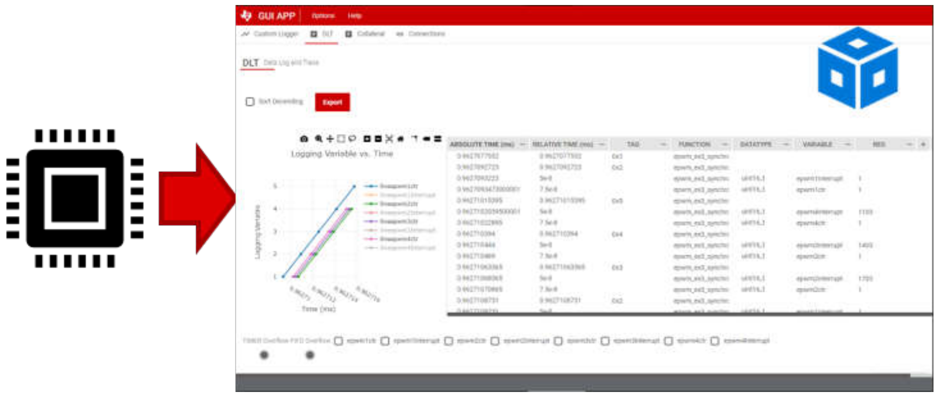


Figure 6-1. DLT

6.2 Walkthrough on Tool

The DLT tool can be found under the MCU Mission Control and Transfer Module within SysConfig.

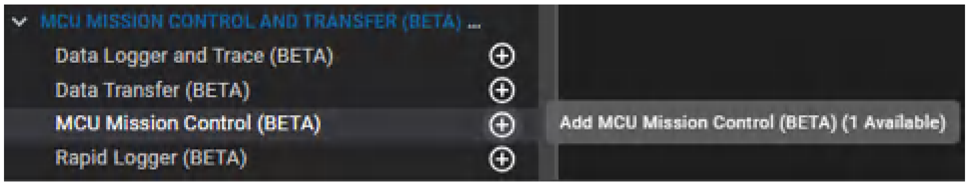
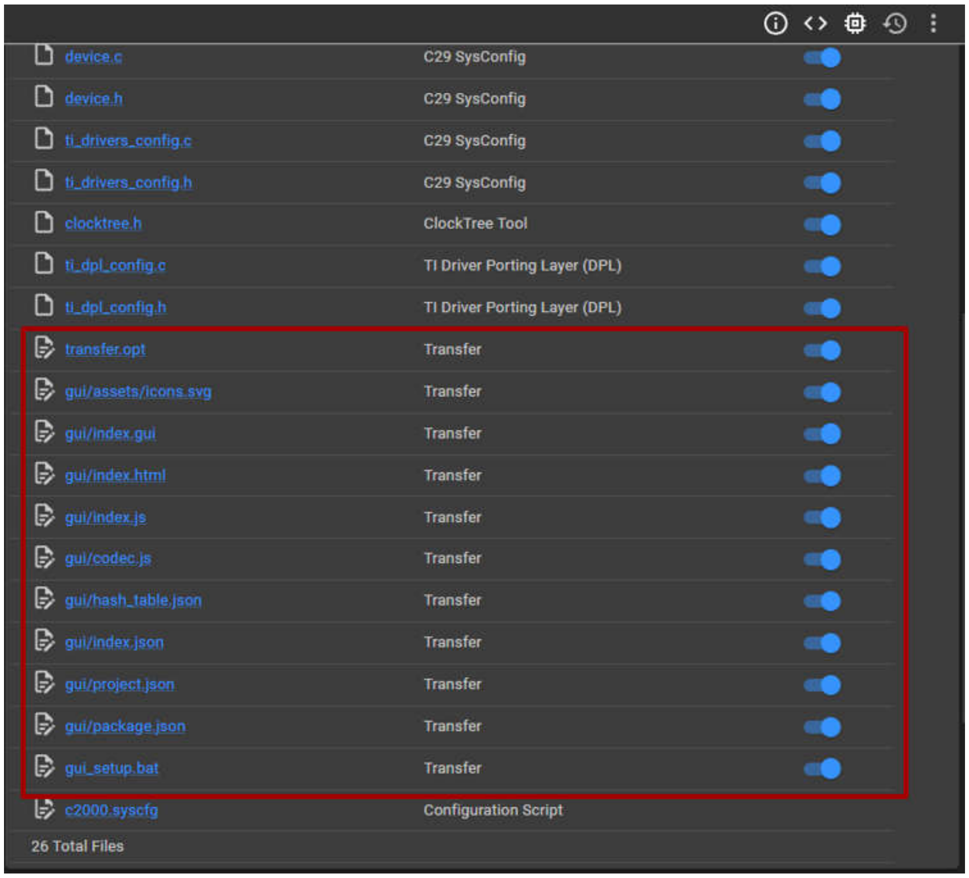

MCU MISSION CONTROL AND TRANSFER (BETA) ...
Data Logger and Trace (BETA)
Data Transfer (BETA)
MCU Mission Control (BETA)
Rapid Logger (BETA)
Add MCU Mission Control (BETA) (1 Available)

Figure 6-2. MCU Mission Control

The first step to adding DLT tooling support is adding the MCU Mission Control module. Once this module is added, there are a few files that get generated to create the final GUI that is shown in CCS. All these files create the front end of the GUI Composer application and the back end required to process incoming data to the GUI.

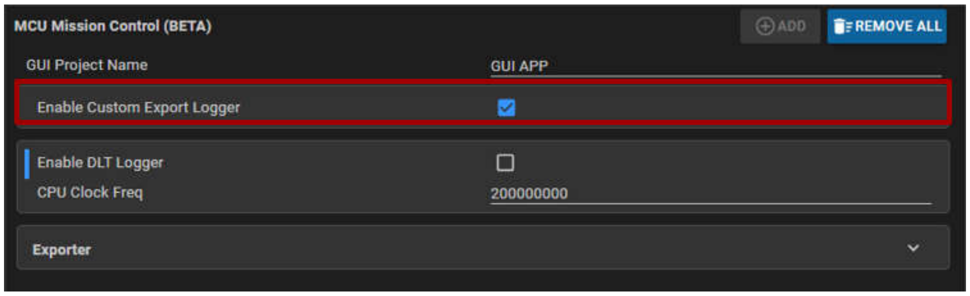


device.c	C29 SysConfig	✓
device.h	C29 SysConfig	✓
ti_drivers_config.c	C29 SysConfig	✓
ti_drivers_config.h	C29 SysConfig	✓
clocktree.h	ClockTree Tool	✓
ti_dpl_config.c	TI Driver Porting Layer (DPL)	✓
ti_dpl_config.h	TI Driver Porting Layer (DPL)	✓
transfer.opt	Transfer	✓
gui/assets/icons.svg	Transfer	✓
gui/index.gui	Transfer	✓
gui/index.html	Transfer	✓
gui/index.js	Transfer	✓
gui/codec.js	Transfer	✓
gui/hash_table.json	Transfer	✓
gui/index.json	Transfer	✓
gui/project.json	Transfer	✓
gui/package.json	Transfer	✓
gui_setup.bat	Transfer	✓
c2000.syscfg	Configuration Script	

26 Total Files

Figure 6-3. MCU Mission Control - Generated Files

This sub module generates all the necessary GUI elements needed for the visualization of the DLT tool. The next item to add is to enable Custom Export Logger.

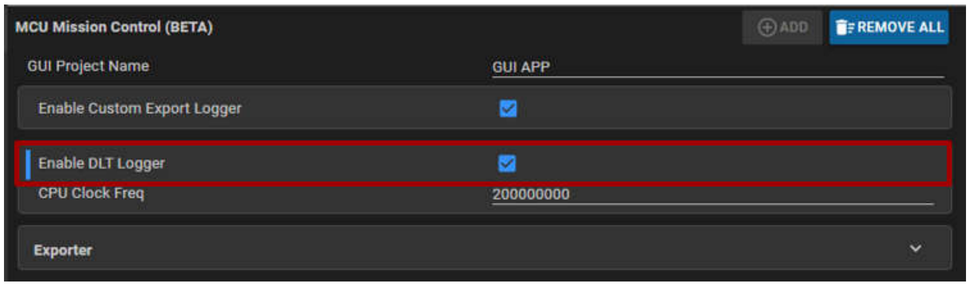


MCU Mission Control (BETA)

GUI Project Name	GUI APP
Enable Custom Export Logger	☑
Enable DLT Logger	☐
CPU Clock Freq	200000000

Figure 6-4. MCU Mission Control - Enable Custom Export Logger

This module assists with defining a package mode that is used to send out the DLT packet to the GUI and what communication peripheral to use. Once enabled, the next option to enable is the *DLT Logger*.

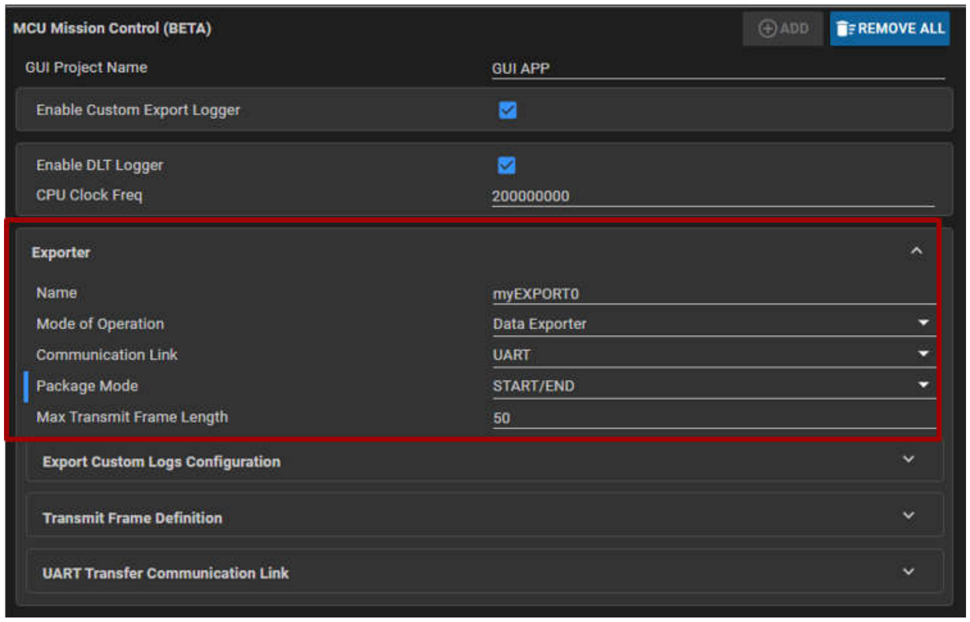


MCU Mission Control (BETA)

GUI Project Name	GUI APP
Enable Custom Export Logger	☑
Enable DLT Logger	☑
CPU Clock Freq	200000000

Figure 6-5. MCU Mission Control - Enable DLT Logger

Once *Enable Custom Export Logger* is enabled, there are a few configurable options that can be selected. The options can be viewable under the *Exporter* tab. The tab looks like Figure 6-6.



GUI Project Name	GUI APP
Enable Custom Export Logger	✓
Enable DLT Logger	✓
CPU Clock Freq	200000000
Exporter	
Name	myEXPORT0
Mode of Operation	Data Exporter
Communication Link	UART
Package Mode	START/END
Max Transmit Frame Length	50

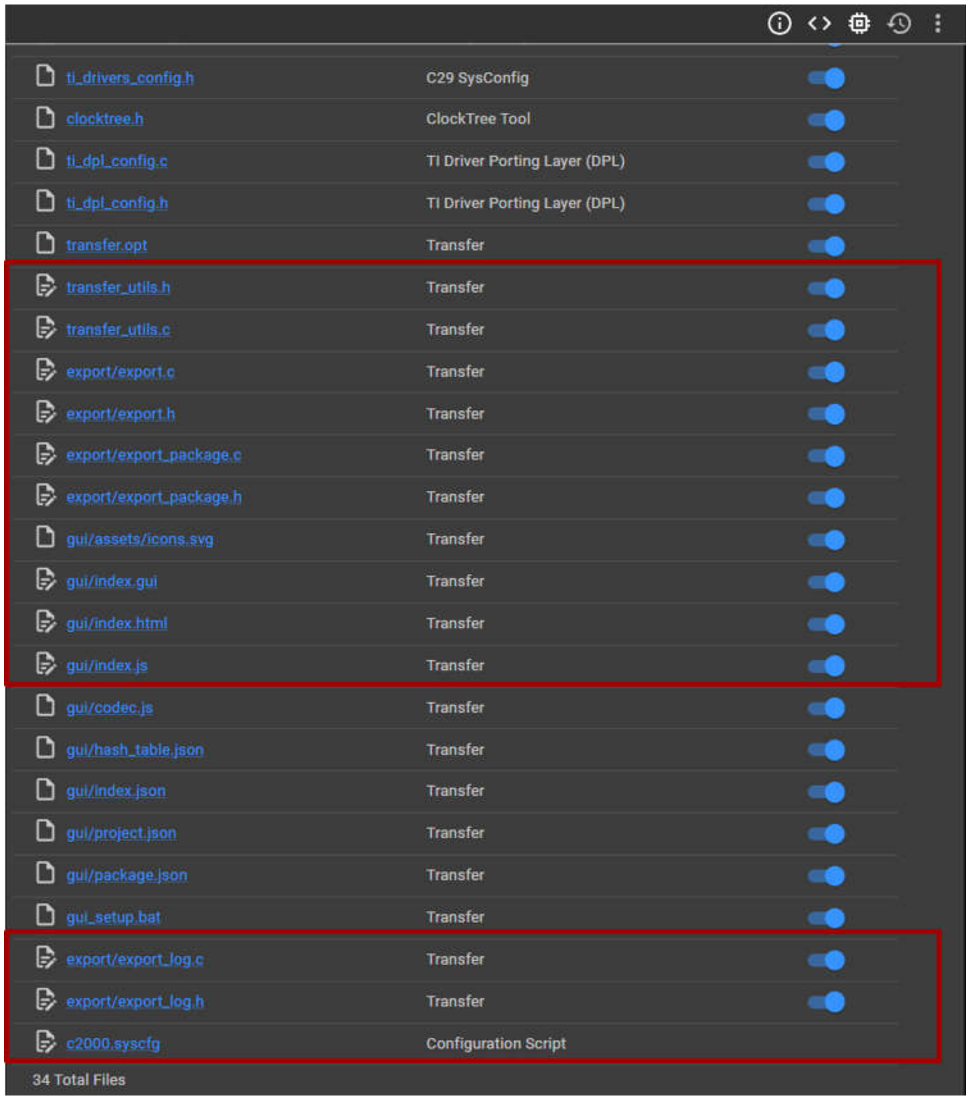
Export Custom Logs Configuration

Transmit Frame Definition

UART Transfer Communication Link

Figure 6-6. MCU Mission Control - Exporter Module

Change the *Package Mode* to be *START/END*. SysConfig generates a software layer to send out data through simple APIs that can be found under the following generated files.

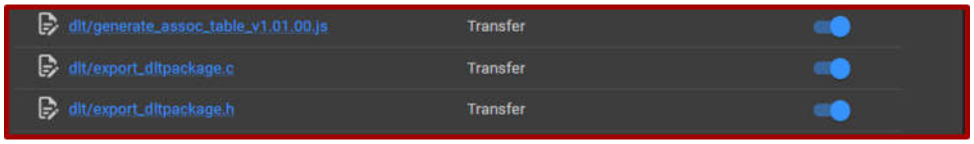


ti_drivers_config.h	C29 SysConfig	
clocktree.h	ClockTree Tool	
ti_dpl_config.c	TI Driver Porting Layer (DPL)	
ti_dpl_config.h	TI Driver Porting Layer (DPL)	
transfer.opt	Transfer	
transfer_utils.h	Transfer	
transfer_utils.c	Transfer	
export/export.c	Transfer	
export/export.h	Transfer	
export/export_package.c	Transfer	
export/export_package.h	Transfer	
gui/assets/icons.svg	Transfer	
gui/index.gui	Transfer	
gui/index.html	Transfer	
gui/index.js	Transfer	
gui/codec.js	Transfer	
gui/hash_table.json	Transfer	
gui/index.json	Transfer	
gui/project.json	Transfer	
gui/package.json	Transfer	
gui_setup.bat	Transfer	
export/export_log.c	Transfer	
export/export_log.h	Transfer	
c2000.syscfg	Configuration Script	

34 Total Files

Figure 6-7. MCU Mission Control - Exporter Module Code Generation

The files being generated are used to package the data in a specified format before exporting the data out of the device. Under export and export_package.c, there are various functions that can be used to export data out of the device. In this application note, the focus is on the DLT support API being generated out of SysConfig. Below are the main files used to export a DLT log and visualize through the GUI.

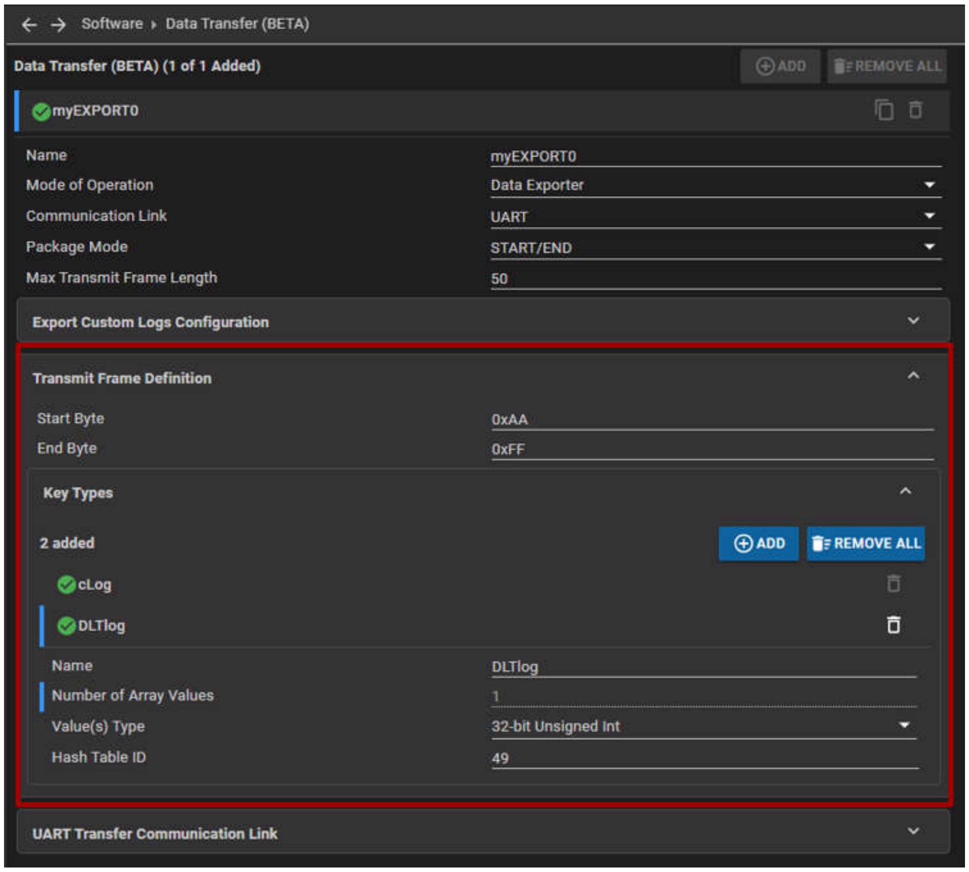


dlt/generate_assoc_table_v1.01.00.js	Transfer	
dlt/export_dltpackage.c	Transfer	
dlt/export_dltpackage.h	Transfer	

Figure 6-8. MCU Mission Control - DLT Logger Code Generation

Inside of dlt/export_dltpackage.h, there is a high level API that can be used to export a uint32_t array of size 2 to the GUI.

Under the *Exporter* module, open the Transmit Frame Definition and add a new key type called *DLTlog* where the value type is 32-bit unsigned int. The hash table ID must be *49* for GUI composer to know what packet is being received on the GUI side.



Data Transfer (BETA) (1 of 1 Added)

myEXPORT0

Name	myEXPORT0
Mode of Operation	Data Exporter
Communication Link	UART
Package Mode	START/END
Max Transmit Frame Length	50

Export Custom Logs Configuration

Transmit Frame Definition

Start Byte	0xAA
End Byte	0xFF

Key Types

2 added

cLog

DLTlog

Name	DLTlog
Number of Array Values	1
Value(s) Type	32-bit Unsigned Int
Hash Table ID	49

UART Transfer Communication Link

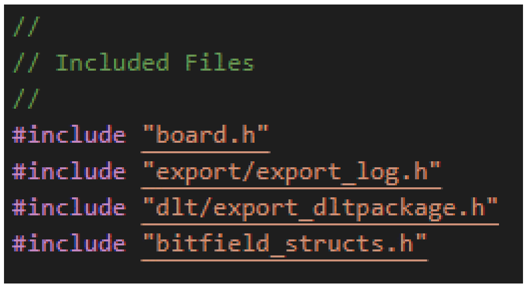
Figure 6-9. MCU Mission Control - Transmit Frame Definition

For the correct pinmux to use the UART on the controlSOM, select GPIO42 for UART_TX and GPIO43 for UART_RX.

Once the MCU Mission Control is added and enable customer export logger is selected. Under the *Enable DLT Logger* checkbox, the CPU Clock Freq must be added that is being used by the device. The clock frequency is used to determine the timer values that is associated with each log.

When enabling the *Enable DLT Logger* checkbox, the *dlt/generate_assoc_table_v1.01.00.js* javascript file gets generated and is used to parse source code information through the compiler tool's Abstract Syntax Tree (AST) dump. This javascript file is used to find the DLT intrinsics that are being used and to be able to display them on the GUI.

The header files must be included to add to application project to use the generated SysConfig support.

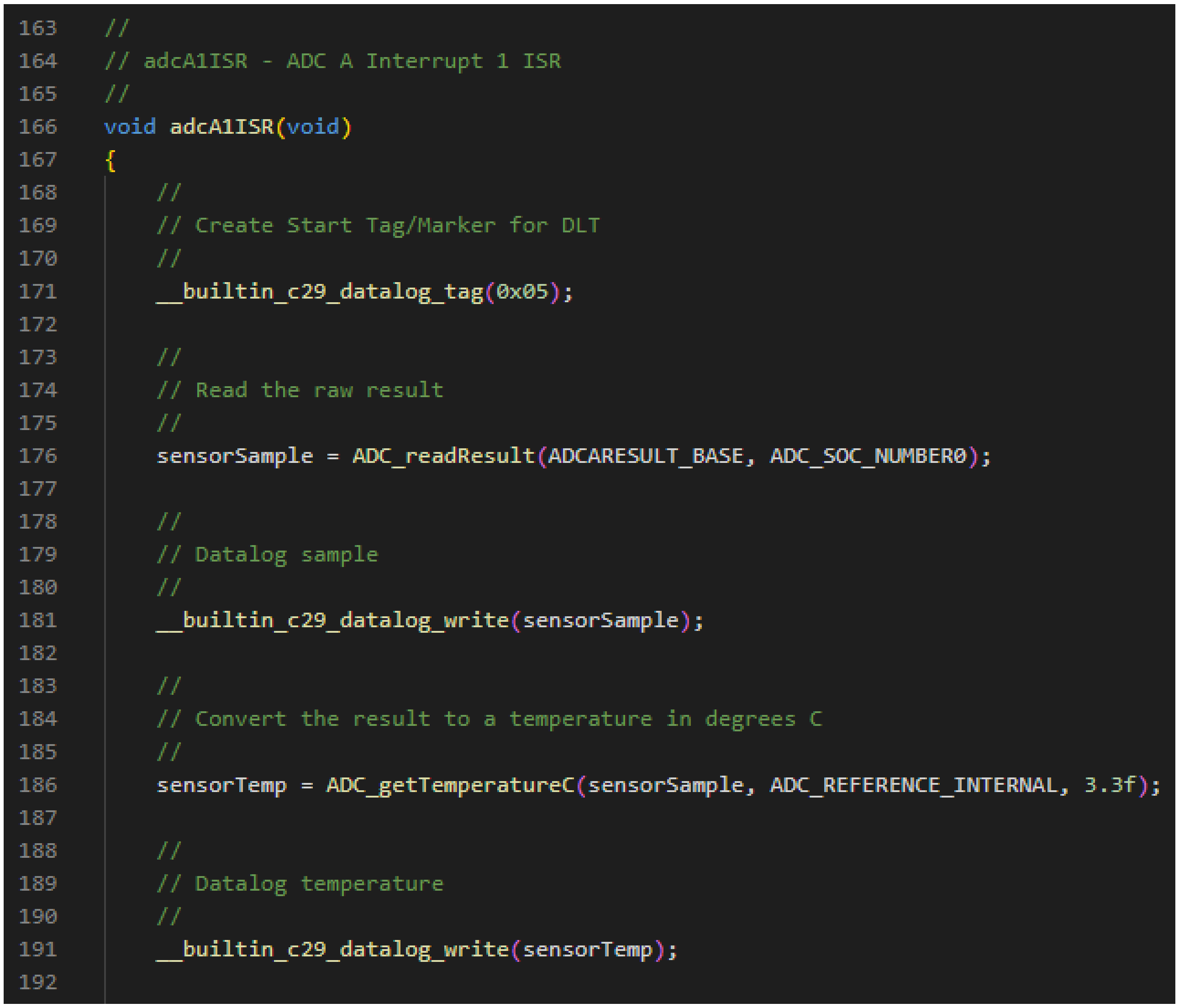


```
//
// Included Files
//
#include "board.h"
#include "export/export_log.h"
#include "dlt/export_dltpackage.h"
#include "bitfield_structs.h"
```

Figure 6-10. MCU Mission Control - Necessary Includes

6.3 Add Logs to Application

There are very specific variables to data log in the example. A couple of the variables to log are sensorSample and sensorTemp. The variables that can be passed into the intrinsic can also be an element of an array. For example, if an array of size 30 is being used. The intrinsic can be passed with array[29] to data log the 30th element. Here is the code snippet to add to the reference design to log the variables. Adhering to the guidelines, at least one tag is placed in every new function scope. Within the reference design under the psfbpcmc.c file, add the following instrinsics.



```
//
// adcA1ISR - ADC A Interrupt 1 ISR
//
void adcA1ISR(void)
{
    //
    // Create Start Tag/Marker for DLT
    //
    __builtin_c29_datalog_tag(0x05);

    //
    // Read the raw result
    //
    sensorSample = ADC_readResult(ADCARESULT_BASE, ADC_SOC_NUMBER0);

    //
    // Datalog sample
    //
    __builtin_c29_datalog_write(sensorSample);

    //
    // Convert the result to a temperature in degrees C
    //
    sensorTemp = ADC_getTemperatureC(sensorSample, ADC_REFERENCE_INTERNAL, 3.3f);

    //
    // Datalog temperature
    //
    __builtin_c29_datalog_write(sensorTemp);

```

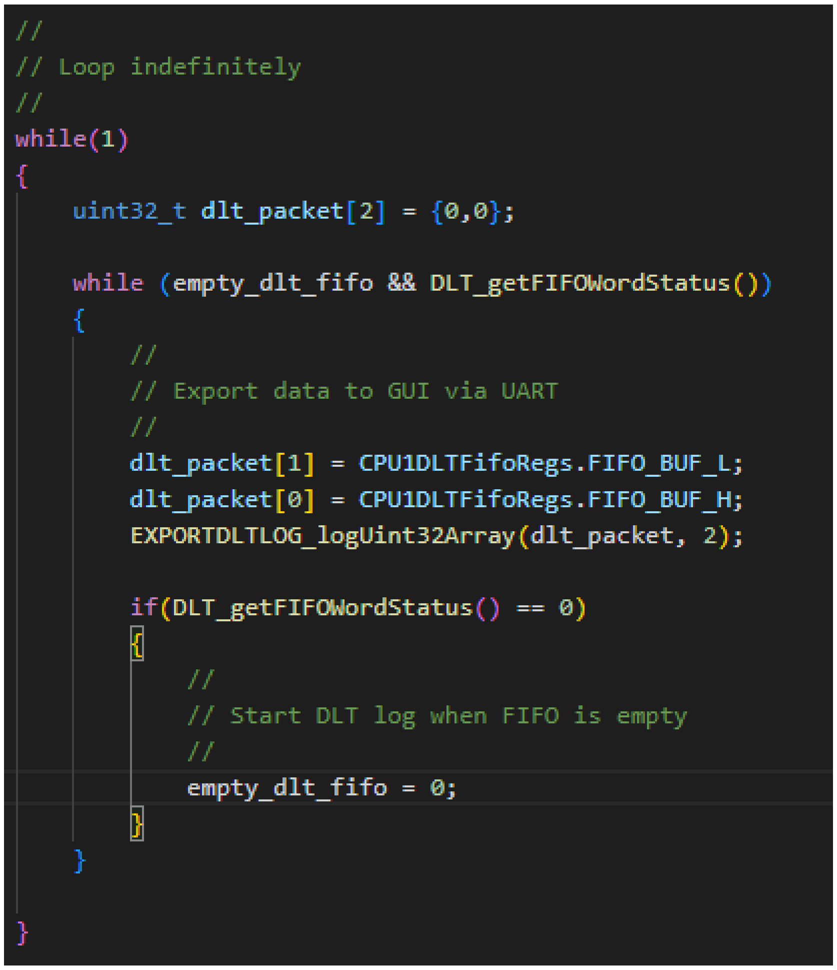
Figure 6-11. DLT Code Marker and Data Log Intrinsic - PSFB_updateSensedValues()

As shown in Figure 6-11, the logging starts to the DLT log buffer since the value passed into the code marker intrinsic is 0x05 and this value matches the start reference value configured in DLT initialization.

6.4 Export DLT Log

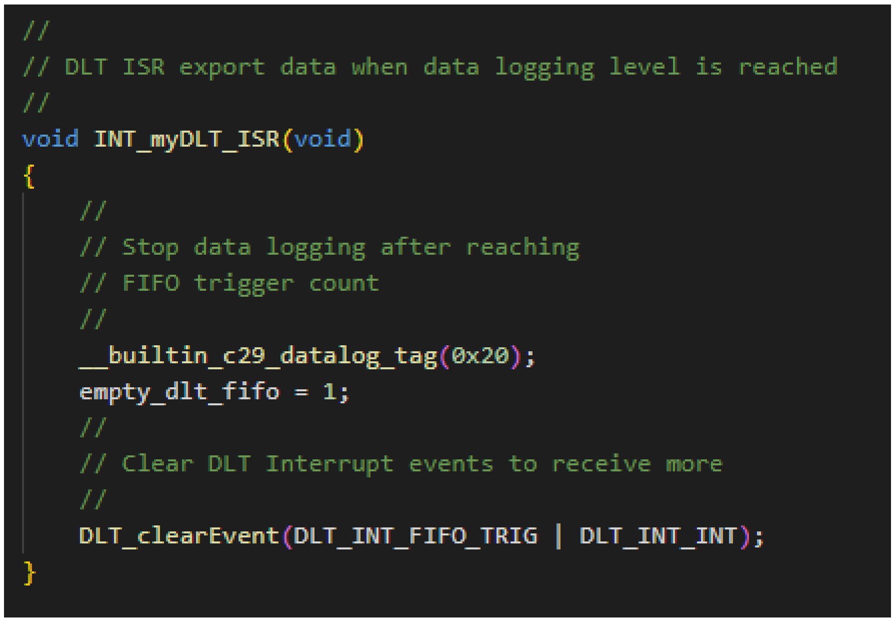
After inserting all the compiler intrinsic, exporting out of the device can be easily done by reading from the FIFO and calling the DLT export package software API. Inside of a background loop the following code snippet can be used to send out the DLT package.

The below empties out the content from the DLT log buffer and writes to an API to export the data out of the device. The DLT interrupt content is used to set the empty_dlt_fifo flag when the FIFO reaches 500 elements.



```c
//
// Loop indefinitely
//
while(1)
{
    uint32_t dlt_packet[2] = {0,0};

    while (empty_dlt_fifo && DLT_getFIFOWordStatus())
    {
        //
        // Export data to GUI via UART
        //
        dlt_packet[1] = CPU1DLTFifoRegs.FIFO_BUF_L;
        dlt_packet[0] = CPU1DLTFifoRegs.FIFO_BUF_H;
        EXPORTDLTLOG_logUint32Array(dlt_packet, 2);

        if(DLT_getFIFOWordStatus() == 0)
        {
            //
            // Start DLT log when FIFO is empty
            //
            empty_dlt_fifo = 0;
        }
    }

}
```

Figure 6-12. DLT Code Marker and Data Log Intrinsic - psfb_main.c

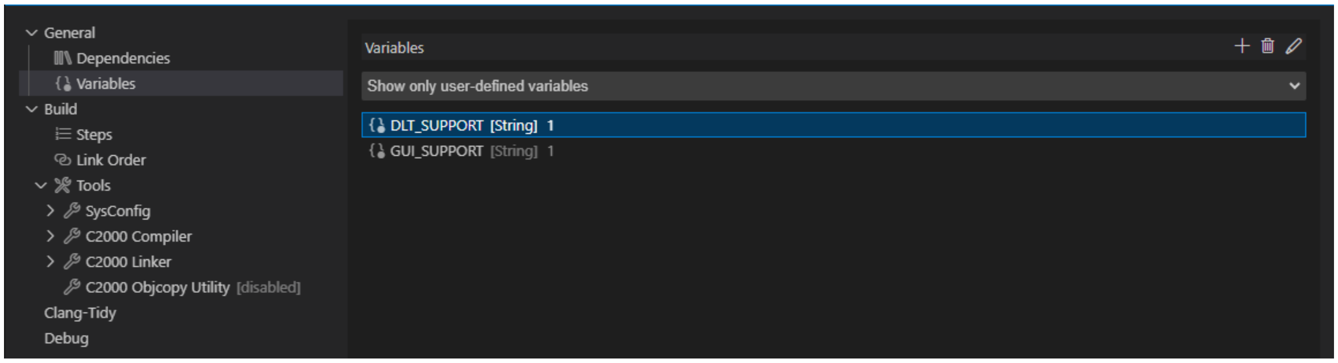


```c
//
// DLT ISR export data when data logging level is reached
//
void INT_myDLT_ISR(void)
{
    //
    // Stop data logging after reaching
    // FIFO trigger count
    //
    __builtin_c29_datalog_tag(0x20);
    empty_dlt_fifo = 1;
    //
    // Clear DLT Interrupt events to receive more
    //
    DLT_clearEvent(DLT_INT_FIFO_TRIG | DLT_INT_INT);
}
```

Figure 6-13. DLT Interrupt Content

6.5 CCS Theia

The final step in all this is going to be adding the post build step to generate our GUI. To enable the GUI support, go to Project Properties -> General -> Variables and add two variables called DLT_SUPPORT and GUI_SUPPORT. Set the values to 1.



DLT_SUPPORT [String] 1
GUI_SUPPORT [String] 1

Figure 6-14. User Defined Variables

This step is required to generate the GUI, please copy and paste the following post build steps.

if ${DLT_SUPPORT} == 1 ${NODE_TOOL} ${BuildDirectory}\syscfg\dlt\generate_assoc_table_v1.01.00.js "${CG_TOOL_ROOT}/bin/c29clang.exe -Xclang -ast-dump=json" "-c -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source/driverlib" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source/bitfields" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/examples/device_support/include" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source/rtlibs/dcl" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source/rtlibs/dsp/fpu/fpu32/fft" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source/rtlibs/iqmath" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source/kernel/freertos/Source/include" -I"${COM_TI_MCU_SDK_F29H85X_INSTALL_DIR}/source/kernel/freertos/Source/portable/CCS/C2000_C29x" -I"${PROJECT_ROOT}/" -I"${CG_TOOL_INCLUDE_PATH}" -I"${PROJECT_ROOT}/CPU1_RAM/syscfg" -I"${CG_TOOL_INCLUDE_PATH}" -DDEBUG -g"

if ${GUI_SUPPORT} == 1 ${BuildDirectory}\syscfg\gui_setup.bat

Note

This is for CPU1_RAM configuration, if using for FLASH, update CPU1_RAM to be the CPU1_FLASH folder in the first post build command.

After adding these post build steps, go ahead and build the project. Once the project finishes building to view the GUI, go to View -> Reload Window. Once the window finishes reloading, go to View -> Plug-Ins -> gui_app. The name that appears depends on the name provided in the *Gui Project Name*, under the MCU Mission Control module.

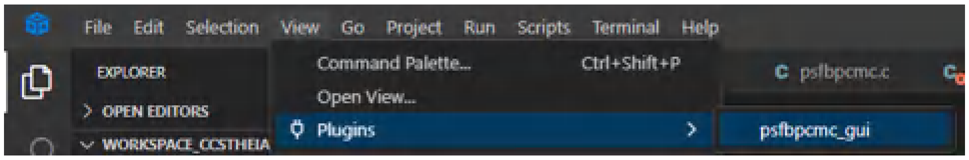
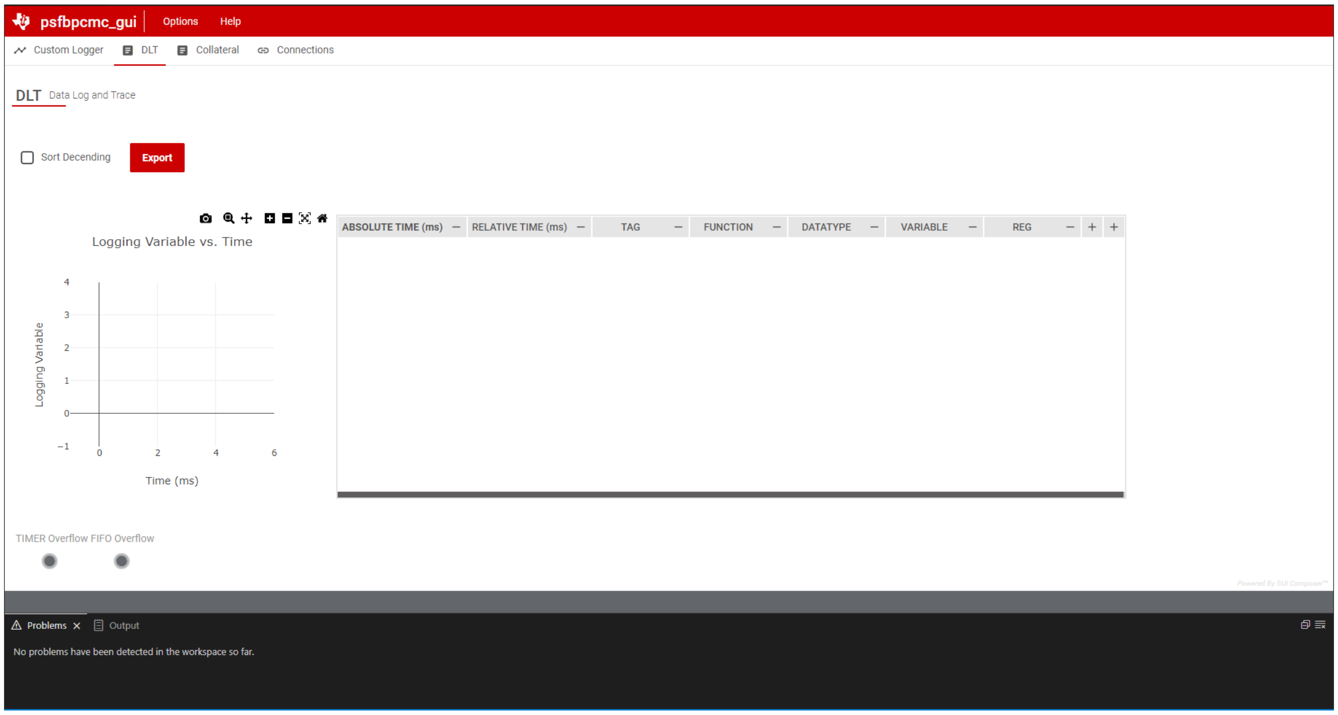


Figure 6-15. CCS Theia Plugins

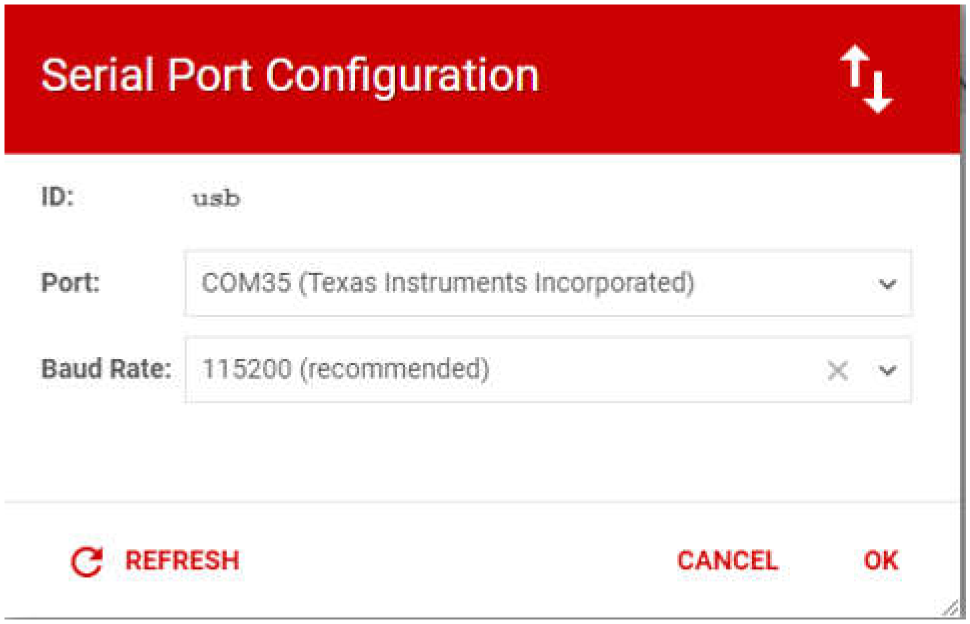
The GUI a tab called DLT, where all DLT content goes to. The table and graph shows case all tag markers and variables that were applied to the source code. If more intrinsics are added in the source code, the project needs to be rebuilt and the GUI needs to be reopened.



No problems have been detected in the workspace so far.

Figure 6-16. Base GUI in CCS Theia

Select the correct COM Port by going to *Options → Serial Port Settings...*.

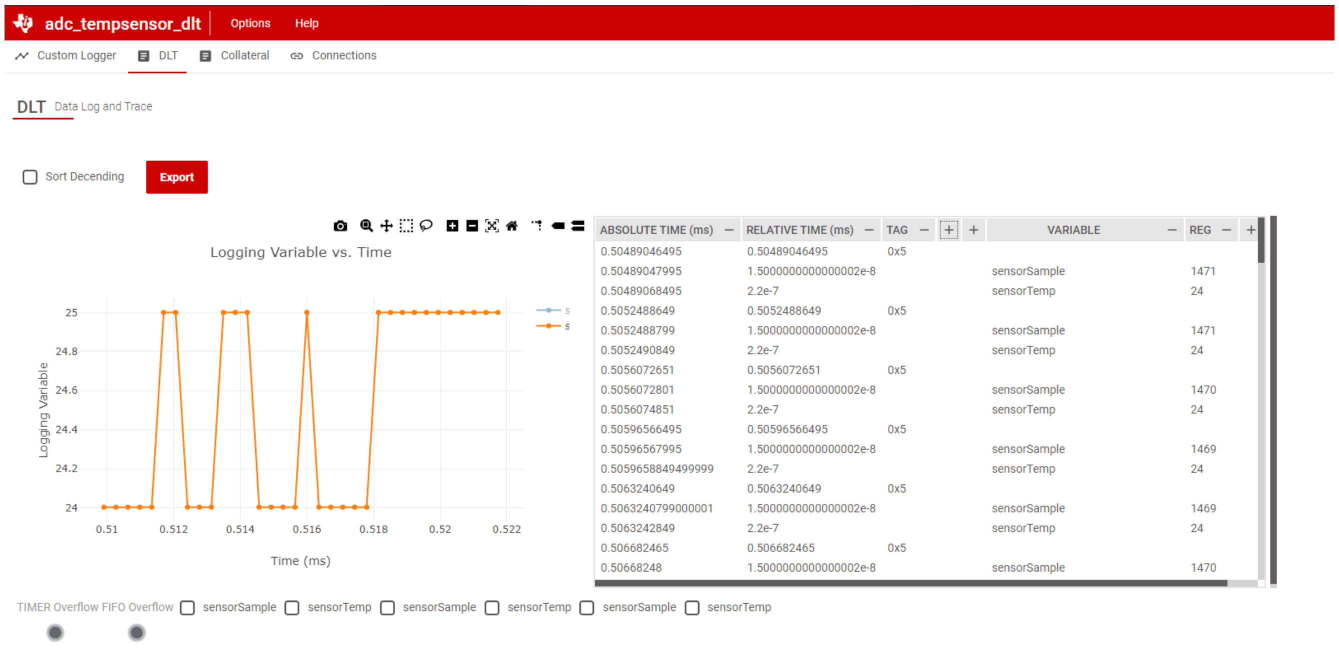


ID: usb
Port: COM35 (Texas Instruments Incorporated)
Baud Rate: 115200 (recommended)

Figure 6-17. Serial Port Configuration

The correct COM port must be selected for the GUI to function properly.

Connect to the device and load the program to the device and the graph fills with data.



ABSOLUTE TIME (ms)	RELATIVE TIME (ms)	TAG	VARIABLE	REG
0.50489046495	0.50489046495	0x5		
0.50489047995	1.5000000000000002e-8		sensorSample	1471
0.50489068495	2.2e-7		sensorTemp	24
0.5052488649	0.5052488649	0x5		
0.5052488799	1.5000000000000002e-8		sensorSample	1471
0.5052490849	2.2e-7		sensorTemp	24
0.5056072651	0.5056072651	0x5		
0.5056072801	1.5000000000000002e-8		sensorSample	1470
0.5056074851	2.2e-7		sensorTemp	24
0.50596566495	0.50596566495	0x5		
0.50596567995	1.5000000000000002e-8		sensorSample	1469
0.5059658849499999	2.2e-7		sensorTemp	24
0.5063240649	0.5063240649	0x5		
0.5063240799000001	1.5000000000000002e-8		sensorSample	1469
0.5063242849	2.2e-7		sensorTemp	24
0.506682465	0.506682465	0x5		
0.50668248	1.5000000000000002e-8		sensorSample	1470

Figure 6-18. Final Output GUI Composer

The columns that are shown are the translated DLT package logs. Absolute time is the IPC timer value converted to milliseconds. Relative time is always relative to a previous code marker or DLTAG. The *TAG* column is the value that was passed into the __builtin_c29_datalog_tag compiler intrinsic. This serves as a way to trace the application code. The function column informs what function the variable being data log is sourced from. The *datatype* column refers to the data type of the variable from the source code. The *variable* column refers to the variable being data logged. The *Reg* column is the value of the variable at the time the log was reached in the application code. There are additional columns that are minimized as the columns are related to overflow and whether the log was a code marker or variable log.

Table 6-1. DLT GUI

Absolute Time	Relative Time	TAG	Function	Datatype	Variable	Reg	Timer 1 OVF	Timer 2 OVF	Code marker or variable log?
Absolute time of IPC timer	Relative time between code markers or DLTAGs	TAG value used in __builtin_c29_datalog_tag	Source file information and function name used in source code where DLT is used	Data type of variable (i.e. float, int, uint etc)	Variable name used in application code passed into __builtin_c29_datalog_write	Register value of the variable being logged	If OVF is 1, then an overflow occurred	If OVF is 1, then an overflow occurred	0 - Variable Log 1- Code Marker

7 Summary

This application note highlights all of the major features of the DLT and how the DLT can be utilized in an application. All of the required configurations for both the DLT and visualizing the DLT log contents were shown. For more specific guidance on registers on the DLT module, see the device-specific technical reference manual.

8 References

- Texas Instruments, F29H85x and F29P58x Real-Time Microcontrollers, technical reference manual
- Texas Instruments, C29 DLT Peripheral Overview Video, webpage
- Texas Instruments, C29x Academy, webpage
- Texas Instruments, C29x SDK, tool page