Tutorial

Version 1.0.3

Creating a USB Data Logger Application using Middleware and CMSIS

ARM KEIL Microcontroller Tools

The latest version of this document is here: www.keil.com/appnotes/docs/apnt_273.asp

Abstract

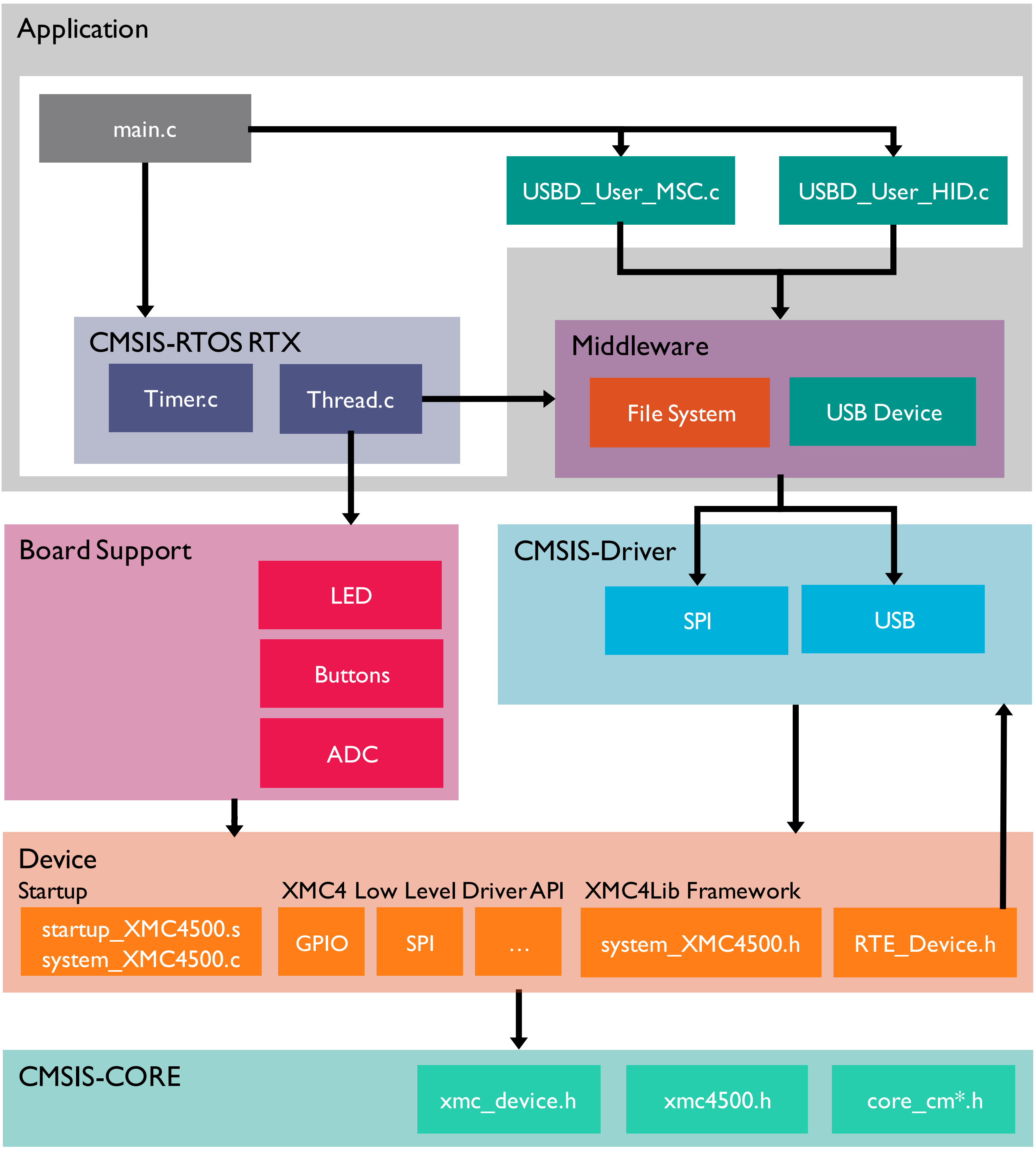
This tutorial shows how to implement a data logger application that collects information from A/D channels and digital I/O ports, recording the data into files inside a micro SD card. When connected to a computer it's enumerated as an USB Composite Device, allowing the access to the files and the transmission of a command to start/stop recording. The tutorial explains the required steps to create the application on an Infineon XMC4500 Relax Kit board but can be easily ported to other underlying hardware as it is using MDK-Professional Middleware and CMSIS, the Cortex Microcontroller Software Interface Standard.

Contents

Introduction

This workshop explains how to create a software framework for a sophisticated microcontroller application using CMSIS and Middleware components. During this workshop a demo application is created that implements the following functions:

- Collect data from A/D channels and digital I/O ports and blink LEDs.
- Create files and write data onto the MicroSD card which is connected to the board through the SPI interface.
- Access the MicroSD Card files from the computer through the USB Mass Storage interface.
- Send commands to the board from the computer through the USB HID interface.



Software Stack

The application is created by using user code templates. These templates are part of software components such as the Middleware, CMSIS-RTOS or the XMC4000 Device Family Pack (DFP).

CMSIS-RTOS RTX is a real-time operating system that is part of MDK and adheres to the CMSIS specification. It is used to control the application.

The **board support** files enable the user to quickly develop code for the hardware that is used here. It provides a simple API to control LEDs, push buttons, A/D converters or other external devices.

Middleware provides stacks for TCP/IP networking, USB communication, graphics, and file access. The Middleware used in this application is part of MDK-Professional and uses several CMSIS-Driver components.

CMSIS-Driver is an API that defines generic peripheral driver interfaces for middleware making it reusable across compliant devices. It connects microcontroller peripherals with middleware that implements for example communication stacks, file systems, or graphic user interfaces. CMSIS-Drivers are available for several microcontroller families and are part of the DFPs. The DFP contains the support for the **device** in terms of startup and system code, a configuration file for the CMSIS-Driver and a device family specific software framework with hardware abstraction layer (HAL) or low level drivers (LLD).

The basis for the software framework is **CMSIS-Core** that implements the basic run-time system for a Cortex-M device and gives the user access to the processor core and the device peripherals. The device header files adhere to the CMSIS-Core standard and help the user to access the underlying hardware.

Prerequisites

To run through the workshop you need to install the following software. Directions are given below:

- MDK-ARM Version 5.14 or later (https://www.keil.com/demo/eval/arm.htm).
- A valid MDK-Professional license.
- Keil::MDK-Middleware 6.3 or higher.
- Infineon::XMC4000_DFP 2.2.0 or later which includes the XMC4500 Relax Kit Board Support Package (BSP). We will download this from the Internet using Pack Installer.
- XMC4500 Relax Kit (http://www.keil.com/boards2/infineon/xmc4500relaxlitekit/).
- Text snippets for copy and paste and completed projects are here: www.keil.com/appnotes/docs/apnt_273.asp

This tutorial assumes you have some experience with the MDK development tool and a basic knowledge of C.

Set up the Workshop Environment

Install MDK:

- Install MDK-ARM Version 5.14 or later. Use the default folder C:\Keil_v5 for the purposes of this tutorial.
- After the initial MDK installation, the Pack Installer utility opens up. Read the Welcome message and close it.

Install the XMC4000 Device Family Pack:

- If Pack Installer is not open, first open µVision®: . Then open Pack Installer by clicking on its icon:
- The bottom right corner should display ONLINE: If it displays OFFLINE, connect your PC to the Internet.
- Locate **Keil::XMC4000_DFP**. Click Install. The installation will commence.
- Once the Pack is installed this will be displayed indicating a successful installation.

Install the Middleware Software Pack:

- Locate **Keil::MDK-Middleware**. Click Update
- Close Pack Installer by selecting File/Exit.

Install your MDK-Professional license.

- In File/License Management, select the 7 day license. This button is only displayed if you are eligible for this offer. It can be used only once.
- You may contact our sales team to request a time-limited license for this workshop: www.keil.com/contact
- For more information and license installation instructions see: www.keil.com/download/license/

Connect the XMC4500 Relax Kit to your PC:

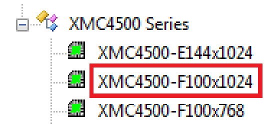
- Use the USB-Micro cable to connect your computer and the XMC4500 Relax Kit board using the port labeled as “X100 Debug”.

Step 1: Create a Project

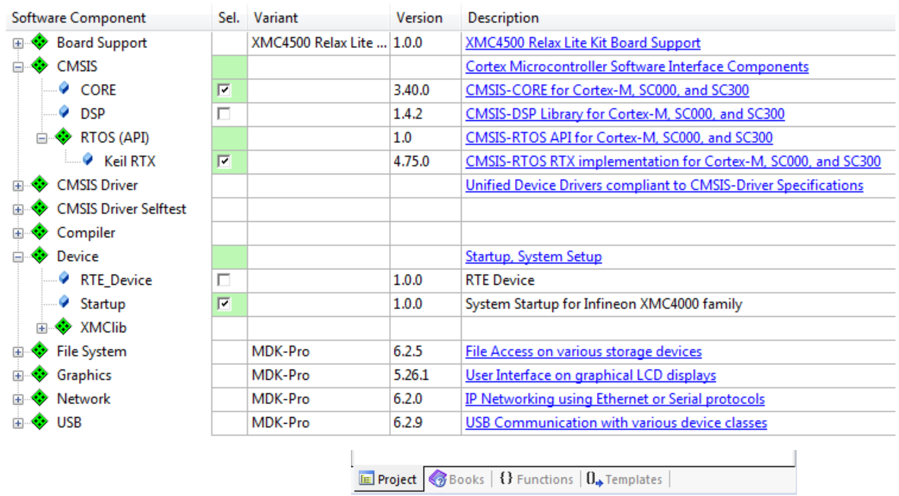
Create a New Project for the Evaluation Board

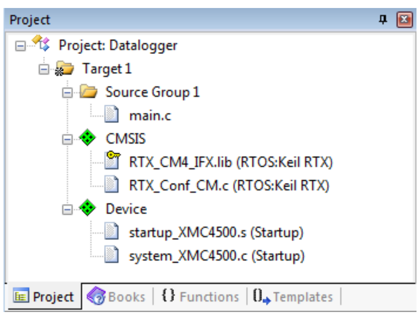
Create a project with initialization files and the main module:

1. In the main µVision menu, select **Project → New µVision Project**. The Create New Project window opens up.
2. Create a suitable folder in the normal fashion and name your project. We will use C:\Datalogger and the project name will be **Datalogger**. When you save the project the project file name will be Datalogger.uvprojx.
   The **Select Device for Target** window opens. Select **XMC4500-F100x1024**:



3. Click on **OK** and the Manage Run-Time Environment window opens:
   Expand the various options as shown and select:
   - CMSIS:Core
   - CMSIS:RTOS (API):Keil RTX
   - Device:Startup
4. Click **OK** to close this window
5. In the Project window expand all the items and have a look at the files µVision has added to your project.



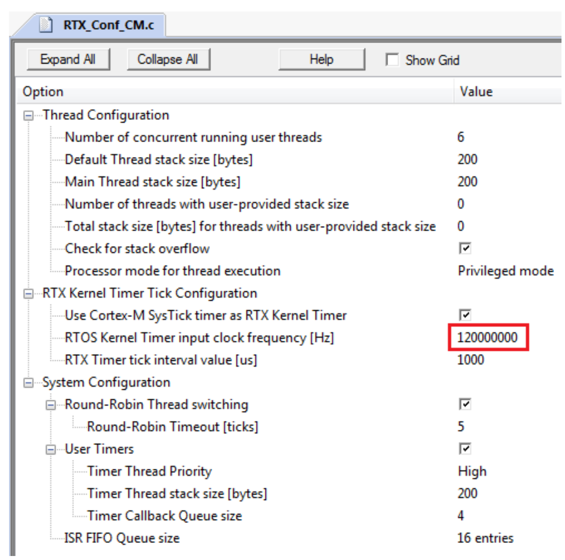


Add the main.c file:

1. Right click on Source Group 1 and select **Add New Item to Group 'Source Group 1'…**
2. In the window that opens, select **User Code Template**. Select **CMSIS-RTOS 'main' function**.
3. Click on **Add**.

Set the RTOS clock frequency:

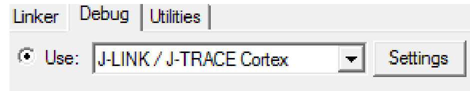
1. In the Project tab under **CMSIS**, double-click on the file **RTX_Conf_CM.c** to open it.
2. In the Configuration Wizard tab, set the **RTOS Kernel Timer input clock frequency [Hz]** to 120000000.
3. Select **File → Save All** or press 
4. Compile the project source files: 
   There will be no errors or warnings displayed in the Build Output window. If you get any errors or warnings, please correct this before moving on to configure the JLINK Debug Adapter.



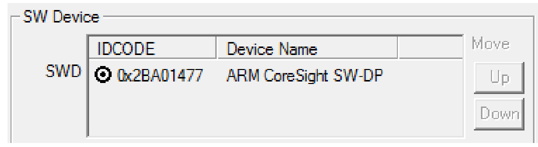
What we have at this point: We have created a new MDK 5 project called Datalogger.uvprojx. We have set the RTOS clock frequency, added the CMSIS environment, a main.c file and compiled the source files to test everything.

Setup the Debug Adapter

Select the J-LINK / J-TRACE Cortex debug adapter:



1. Select Target Options or ALT-F7. Select the **Debug** tab.
2. In the **Use** box select "J-LINK / J-TRACE Cortex".
3. Click on **Settings**. In the **Port** box, select **SW** (for Serial-Wire Debug SWD).
4. In the SW Device box you must see a valid IDCODE and **ARM CoreSight SW-DP**. This indicates that µVision is connected to the J-LINK's debug module.



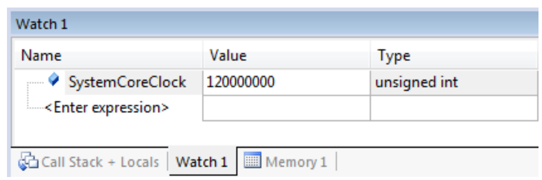
If you see an error or nothing in the SW Device box, you must fix this before you can continue. Make sure the board is connected.

Configure the Serial Wire Viewer (SWV):

1. Select the **Trace** tab. In the **Core Clock** box, enter **120 MHz** and select **Trace Enable**. This sets the speed of the SWO UART signal and debugger timing displays.
2. Unselect **EXCTRC** (Exception Tracing). Leave all other settings at their defaults.

Insert a global variable in the Watch window:

1. In the Project tab under **Device**, double-click on **system_XMC4500.c** to open it up.
2. Find the variable SystemCoreClock. It is declared near line 283.
3. Right click on it and select **Add SystemCoreClock to…** and select **Watch 1**. Watch 1 will automatically open if it is not already open and display this variable.
4. In the Watch 1 window, right click on SystemCoreClock in the Name column and unselect **Hexadecimal Display**. SystemCoreClock will now be displayed with the correct frequency of 120 MHz.
   Note: You can add variables to the Watch and Memory windows while your program is running.



What we have at this point: We have selected the debug adapter, enabled the Serial Wire Viewer trace and demonstrated how to display the CPU clock in a Watch window.

Step 2: Add CMSIS-RTOS Thread and Timer

Add and configure CMSIS-RTOS Thread and Timer

Add the Thread.c source file and its initialization:

In the Project window under Target 1, right click **Source Group 1** and select **Add New Item Group 'Source Group 1'**…

1. In the window that opens, select **User Code Template**. Select **CMSIS-RTOS Thread**.
2. Click on **Add**. Note Thread.c is added to the Source Group 1 in the Project window.
3. Click on the main.c tab to bring it in focus in order to edit it.
4. In main.c near line 8, add this line: `extern int Init_Thread(void);`
5. In main.c near line 19 and before `osKernelStart();`, add `Init_Thread();`
6. In Thread.c, replace the content of the while loop, inside the function Thread near line 24, by this code:
   `osSignalWait(0x01,osWaitForever);`
   It will suspend the execution of this thread until the specified signal flag is set.

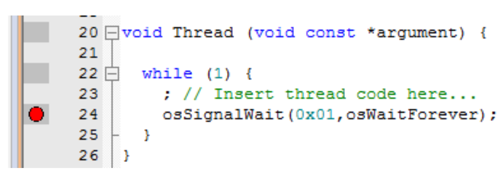
Add the Timer.c source file and its initialization:

1. In the Project window under Target 1, right click **Source Group 1** and select **Add New Item Group 'Source Group 1'**…
2. In the window that opens, select **User Code Template**. Select **CMSIS-RTOS Timer**.
3. Click on **Add**. Note Timer.c is added to the Source Group 1 in the Project window.
4. Click on the main.c tab to bring it in focus in order to edit it.
5. In main.c near line 9, add this line: `extern void Init_Timers(void);`
6. In main.c near line 18 and after `osKernelInitialize();`, add `Init_Timers();`
7. Init_Timers creates two timers: Timer1 (a one-shot) and Timer2 which is a 1 second periodic timer. In this application two periodic timers will be used and then the Timer1 has to be modified. In Timer.c near line 42, change the second parameter of the call `osTimerCreate` from `osTimerOnce` to `osTimerPeriodic`:
   `id1 = osTimerCreate (osTimer(Timer1), osTimerPeriodic, &exec1);`
8. In Timer.c near line 8, add this line: `extern osThreadId tid_Thread;`
   In Timer.c near line 19, inside the function `Timer1_Callback`, add this line:
   `osSignalSet(tid_Thread,0x01);`
   It will set the specified thread's signal flag, allowing its execution periodically.
9. Select **File → Save All** or .
10. Compile the project source files by clicking on the Rebuild icon . There will be no errors or warnings in the Build Output window. If there are any errors or warnings, please correct them before continuing.

Demonstrating the Thread is Working:

1. Program the Flash and enter Debug mode: Click on the RUN icon.

   TIP: To program the Flash manually, select the Load icon:
2. The program is running.
3. In Thread.c, near line 24, set a breakpoint by clicking on the gray box. A red circle will appear. The gray box indicates that assembly language instructions are present and a hardware breakpoint will be legal.
4. The program will soon stop here.



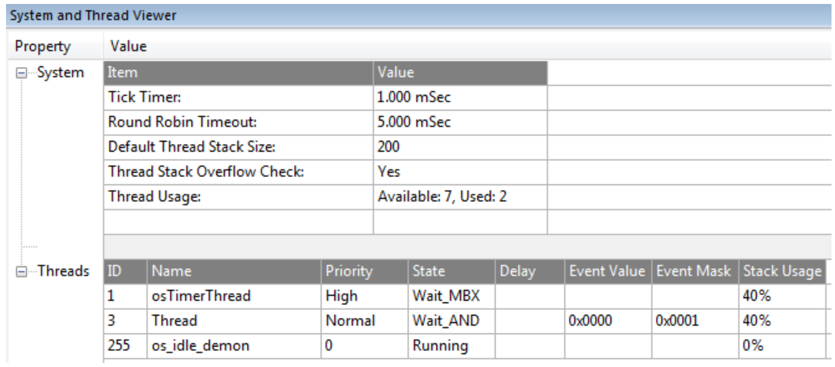
5. Click on RUN and in 100 milliseconds it will stop here again when the thread's execution is resumed.

What we have at this point: We added the RTX Thread and Timer to your project. We enabled a periodic Timer and demonstrated that the thread is running periodically.

RTX Kernel Awareness

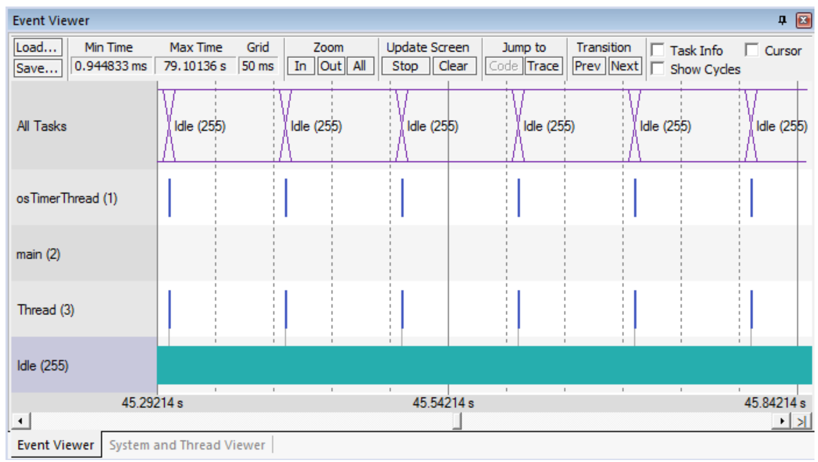
System and Thread Viewer:

1. With the program running, open **Debug → OS Support** and select **System and Thread Viewer**. This window opens up:
   Note: os_idle_demon and osTimerThread threads have been already created.
2. When you click on RUN, the status of these two threads will be updated in real-time until the program stops.
3. Note the various other fields that describe RTX.

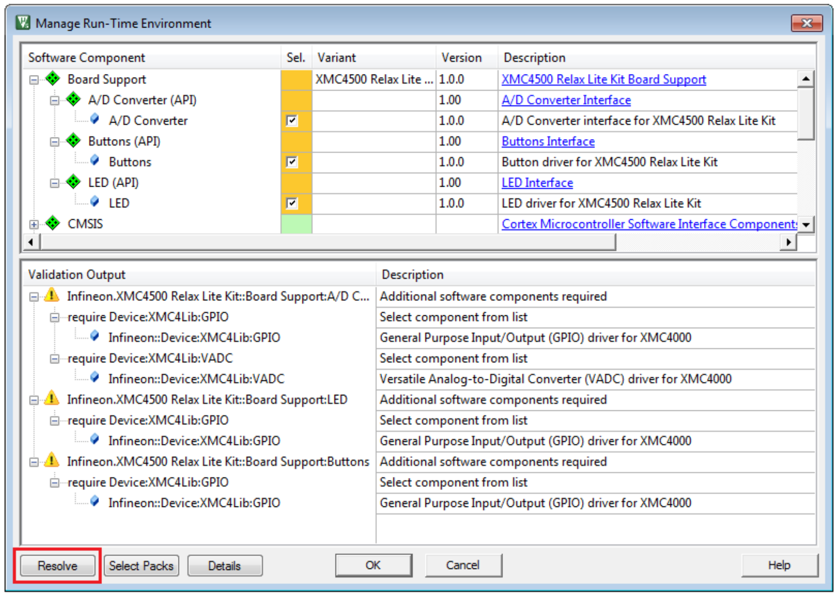


Event Viewer:

1. Open **Debug → OS Support** and select **Event Viewer**. The following window opens. Resize it for convenience. If this window does not display any information, the most likely cause is that the SWV is not enabled or the CPU clock frequency is incorrect. See **Serial Wire Viewer Summary** on the last page for useful SWV hints.
2. Click on RUN. 
3. Using **In**, **Out** and **All** in the **Zoom** field, set the grid for about 50ms.
4. It is easy to see when the threads are running. Note most of the time the **Idle** thread is running.
5. You can tell at a glance the timing of your RTX implementation and if it is behaving as you expect.
6. As you add new tasks, they will be automatically added. The Event Viewer uses the Serial Wire Viewer (SWV).
7. Exit Debug mode: 



Step 3: Add Interfaces to the Board Components



As we want to collect data from board peripherals, as well as blink the LEDs, we need to add the correspondent board support components to the project:

1. Open the Manage Run-Time Environment window and select:
   - Board Support:A/D Converter
   - Board Support:Buttons
   - Board Support:LED
2. Click **Resolve** to add other mandatory middleware components.
3. Click **OK** to close this window.

Initialize board support components:

1. In main.c near line 8, add the include directives:
```
#include "Board_Buttons.h"
#include "Board_LED.h"
#include "Board_ADC.h"
```

2. In main.c near line 22, add the initialization calls:
```
LED_Initialize();
Buttons_Initialize();
ADC_Initialize();
```

Blink the LEDs:

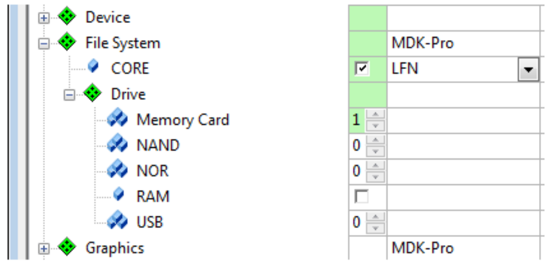
1. In Thread.c, near line 3, add the include directive:
```
#include "Board_LED.h"
```
   Note: An error ✗ might display on this line. Please ignore this for now. Make sure the source lines are typed in exactly as shown to avoid errors. Use your best judgment as to where the source code should be added. Line numbers can change with different versions of the software templates.
2. In Thread.c near line 22, inside the function Thread, declare the following variable:
```
uint8_t led_state = 0x01;
```
3. In Thread.c near line 26, inside the while loop, add the following code:
```
LED_SetOut(led_state);
led_state = ~led_state;
```
4. Select File/Save All or .
5. Compile the project: There will be no errors or warnings in the Build Output window.
6. Program the Flash and enter Debug mode:
7. Click on RUN.
8. The two red LEDs will blink alternately.
9. Exit Debug mode:

What we have at this point: We have selected and initialized board peripheral drivers from the CMSIS-Pack BSP. We have created a simple thread that toggles the LEDs alternately according to a timer event every 100ms.

Step 4: Add File System Support

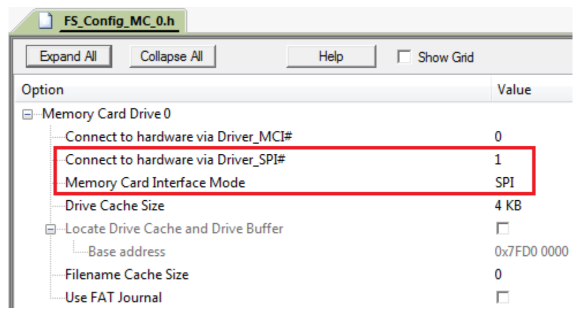
Add the File System component to the project

As we want to create files and save data into the SD card, we need to add support for the File System:



1. Open the Manage Run-Time Environment window and select:
   - File System:CORE:LFN
   - File System: Memory Card
2. Click **Resolve** to add other mandatory middleware components.
3. Click **OK** to close this window.

Connect the File System component to the SPI driver:

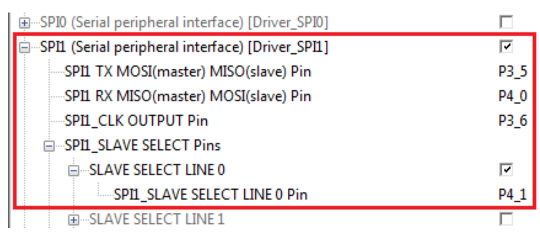


1. In the Project window under the **File System** heading, double click on **FS_Config_MC_0.c** to open it.
2. Click on its **Configuration Wizard** tab and then on **Expand All**.
3. Set **Connect to hardware via Driver_SPI#** to **1**.
4. Set **Memory Card Interface Mode** to SPI.

Initialize the File System:

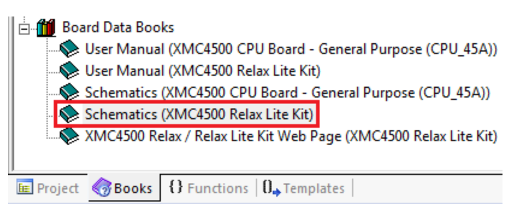
1. In main.c near line 10, add the include directive:
   ```
   #include "rl_fs.h"
   ```
2. In main.c near line 26, add the initialization calls:
   ```
   finit  ("M0:");
   fmount ("M0:");
   ```

Configure the SPI pins for the CMSIS-Driver



1. In the Project window, under the Device header, double click on RTE_Device.h to open it for editing.
2. Click on its Configuration Wizard tab.
3. Enable **SPI1 (Serial peripheral interface)** as shown here:
4. Set the hardware parameters for the SPI1 interface exactly as shown here:
   - **SPI1 TX MOSI (master) MISO (slave) Pin = P3_5**
   - **SPI1 RX MISO (master) MOSI (slave) Pin = P4_0**
   - **SPI1 CLK OUTPUT Pin = P3_6**
   - **SPI1 SLAVE SELECT LINE 0 Pin = P4_1**

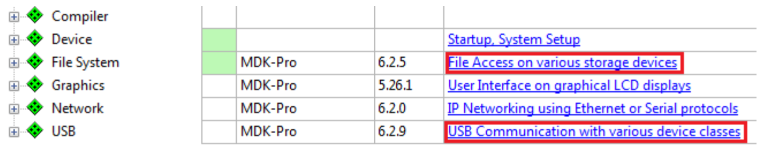
   You can check if the pin assignment is correct in the board schematics, available in µVision under the **Books** tab.



5. Select File/Save All or .

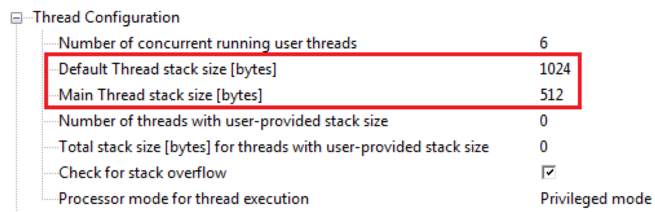
Configure the stack and thread memory resources

The resource requirements of the Middleware components, as the File System and the USB, can be found in the Middleware documentation that is accessible using the link next to the component in the Manage Run-Time Environment window:



Configure Thread Stack sizes:

1. Under the **CMSIS** heading, double click on **RTX_Conf_CM.c** to open it.
2. Click on its **Configuration Wizard** tab and then on **Expand All**.
3. Change **Default Thread stack size [bytes]** to **1024**.
4. Change **Main Thread stack size [bytes]** to **512**.
5. Select File/Save All or .
6. Compile the project: 



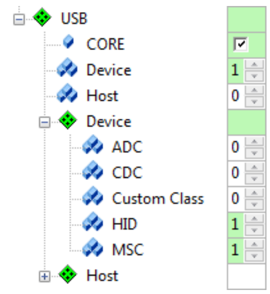
No errors or warnings will be generated as shown in the Build Output window. Please correct any errors or warnings before you continue.

What we have at this point: We have added, configured and initialized the File System component and the CMSIS-Driver for the SPI interface. This configuration allows our application to easily access files in the SD card connected through SPI.

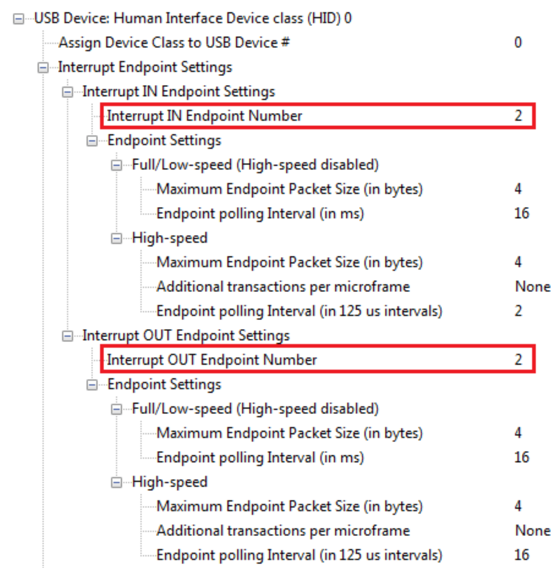
Step 5: Add the USB device classes

In order to access the files in the SD card from a computer, our device has to be recognized as an USB Mass Storage (MSC). We want also to send commands to the device to start and stop the data logger recording, so we need an USB Human Interface Device (HID). Such peripheral device, that supports more than one device class, is called USB Composite Device.

Add the USB components to the project

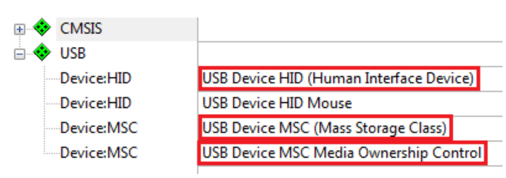


1. Open the Manage Run-Time Environment window and select:
   - CMSIS Driver:USB Device (API)
   - USB:CORE
   - USB:Device = 1
   - USB:Device:HID = 1
   - USB:Device:MSC = 1
2. Click **Resolve** to add other mandatory middleware components.
3. Click **OK** to close this window.



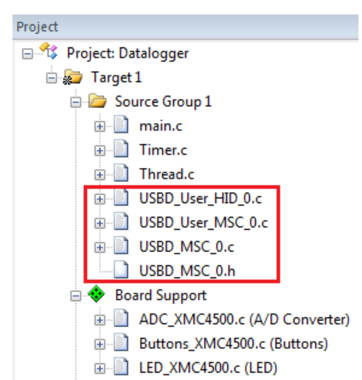
Configure USB components:

1. In the Project window under the **USB** heading, double click on **USBD_Config_0.c** to open it.
2. Click on its **Configuration Wizard** tab and then on **Expand All**.
3. Set **Product ID** to **0x0000**.
4. In the Project window under the **USB** heading, double click on **USBD_Config_HID_0.c** to open it.
5. Click on its **Configuration Wizard** tab and then on **Expand All**.
6. Set **Interrupt IN Endpoint Number** to **2**.
7. Set **Interrupt OUT Endpoint Number** to **2**.



Add USB template files:

1. Right click on Source Group 1 and select **Add New Item to Group 'Source Group 1'…**
2. In the window that opens, select **User Code Template**. Select **USB Device HID (Human Interface Device).**
3. Click on **Add**.
4. Repeat the previous steps to add the **USB Device MSC (Mass Storage Class)** and **USB Device MSC Media Ownership Control** templates.
   Note USBD_User_HID.c, USBD_User_MSC.c, USBD_MSC.c and USBD_MSC.h are added to the Source Group 1 in the Project window.



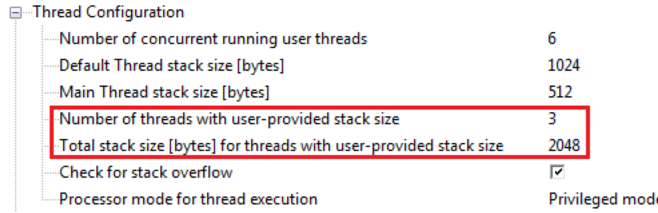
Initialize USB components:

1. In main.c near line 11, add the include directives:
```
#include "rl_usb.h"
#include "USBD_MSC_0.h"
```

2. In main.c near line 30, add the initialization calls:
```
USBD_Initialize (0);
USBD_Connect    (0);
USBD_MSC0_SetMediaOwnerUSB();
```

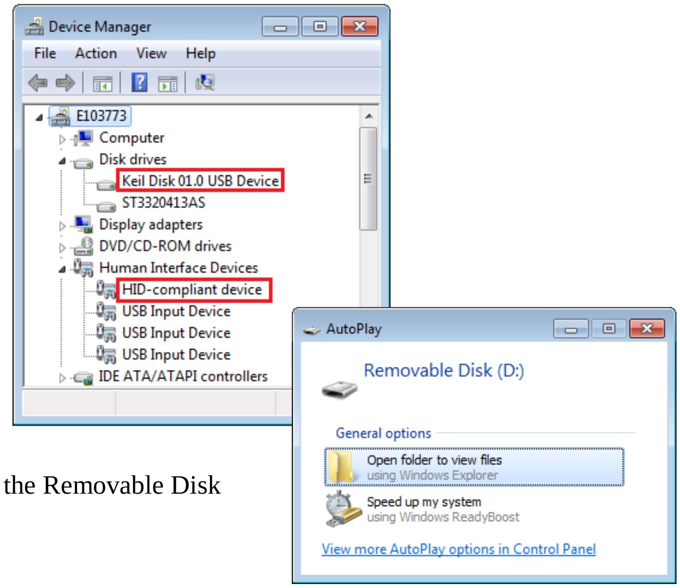
Configure Thread Stack sizes:

1. Under the **CMSIS** heading, double click on **RTX_Conf_CM.c** to open it.
2. Click on its **Configuration Wizard** tab and then on **Expand All**.
3. Change **Number of Threads with user-provided stack size** to **3**.
4. Change **Total stack size [bytes] for threads with user-provided stack size** to **2048**.
5. Select File/Save All or .
6. Compile the project:



Install and test the USB Composite Device:

1. Program the Flash and enter Debug mode:
2. Click on RUN.
3. Insert an SD Card in the slot labelled “X300”.
4. Connect your computer using a USB-Micro cable to the port labelled “X3”.
5. The board will be installed as a USB Composite Device and two new interfaces will appear among the computer devices: a Disk drive and a HID-compliant device.
6. You can access the files stored in the SD card through the Removable Disk drive that is created in your computer.
7. Exit Debug mode:



What we have at this point: We have added the USB component with the MSC and HID interfaces and configured the thread stack size. From the computer we can access the files in the SD Card.

Step 6: Implement the Data Logger Functions

Implement a real time clock

In order to print timestamps associated to the collected data, we will implement a clock with our previously created CMSIS-RTOS Timer.

1. Double click on **Timer.c** to open it for editing.
2. Near line 3, add the include directive:
```
#include <stdio.h>
```
3. Near line 11, add the definition:
```
struct clock {
  unsigned char  hour;
  unsigned char  min;
  unsigned char  sec;
  unsigned short msec;
};
static struct clock time;
```
4. Near line 41, inside the Timer2_Callback, insert the code:
```
if (++time.msec == 100) {
  time.msec = 0;
  if (++time.sec == 60) {
    time.sec = 0;
    if (++time.min == 60) {
      time.min = 0;
      if (++time.hour == 24) {
        time.hour = 0;
      }
    }
  }
}
```
5. Near line 78, change the Timer2 interval from 1000ms to 10ms:
```
status = osTimerStart (id2, 10);
```
6. In the end of the file near line 85, append this function:
```
void PrintTimeStamp(FILE *f) {
  fprintf (f,"%02d:%02d:%02d.%02d",
    time.hour,
    time.min,
    time.sec,
    time.msec);
}
```

Add the user code to the thread

We will modify our previously created CMSIS-RTOS Thread to implement the data logger functionalities: collect data from A/D converter and sample the state of the pushbuttons. Every 100ms the measurement from the A/D converter is stored into a file, together with a timestamp. The same happens when the state of a pushbutton changes.

To allow file access we add the following application code in the module Thread.c:

1. Double click on **Thread.c** to open it for editing.

2. Near line 4, add the include directives:
```
#include "Board_Buttons.h"
#include "Board_ADC.h"
#include "rl_fs.h"
#include "rl_usb.h"
#include "USBD_MSC_0.h"
#include <stdio.h>
```

3. Near line 18, add the code:
```
# extern void PrintTimeStamp(FILE *f);

typedef enum {
  DEV_IDLE,
  DEV_START_RECORD,
  DEV_STOP_RECORD,
  DEV_RECORDING
} DeviceState;
DeviceState gState = DEV_IDLE;

void SetRecording(bool bMode) {
  if (bMode) {
    if (gState==DEV_IDLE) gState = DEV_START_RECORD;
  } else {
    if (gState==DEV_RECORDING) gState = DEV_STOP_RECORD;
  }
}

void LogButton(uint8_t button, uint8_t state) {
FILE *f;
const char ButName[][7] = {"1", "2"};
const char ButState[][9] = {"Released", "Pressed"};
  f = fopen("ButtonsLog.txt","a");
  if (f != NULL) {
    PrintTimeStamp(f);
    fprintf(f," - Button %s %s\n",ButName[button],ButState[state]);
    fclose (f);
  }
}

void LogADC(uint16_t val) {
FILE *f;
  f = fopen("AdcLog.txt","a");
  if (f != NULL) {
    PrintTimeStamp(f);
    fprintf (f," - 0x%04X - %4.2fV\n", val,
     (float) (val * 3.3 / (1 << ADC_GetResolution ())));
    fclose (f);
  }
}
```

4. Near line 66, replace the content of the function Thread with the code:
```
void Thread (void const *argument) {
  uint8_t led_state = 0x01;
  uint8_t but_current;
  uint8_t but_last;
  uint8_t but_changed;
  uint8_t but_num;
```

```
  uint16_t adc_val;

  while (1) {

    switch (gState) {

      case DEV_IDLE:
        break;

      case DEV_START_RECORD:
        // hide logical unit
        USBD_MSC0_SetMediaOwnerFS();
        finit("M0:");
        fmount("M0:");
        ADC_StartConversion();
        gState = DEV_RECORDING;
        break;

      case DEV_STOP_RECORD:
        LED_SetOut(0);
        // show logical unit
        USBD_MSC0_SetMediaOwnerUSB();
        gState = DEV_IDLE;
        break;

      case DEV_RECORDING:
        // Buttons sampling
        but_current = (Buttons_GetState());
        but_changed = but_current ^ but_last;
        but_last = but_current;
        but_num = 0;
        while(but_changed)
        {
          if (but_changed&1) {
            LogButton(but_num,but_current&1);
          }
          but_num++;
          but_changed>>=1;
          but_current>>=1;
        }

        // ADC sampling
        if (ADC_ConversionDone() == 0) {
          adc_val = ADC_GetValue();
          LogADC(adc_val);
        }
        ADC_StartConversion();

        // Toggle leds
        LED_SetOut(led_state);
        led_state = ~led_state;
        break;
    }

    osSignalWait(0x01,osWaitForever);
  }
}
```

Add the USB command handling

As previously stated, the USB HID Client will be used to start/stop recording.
The HID Client running on a PC sends an HID output report that is handled by our application.

Modify USBD_User_HID_0.c:

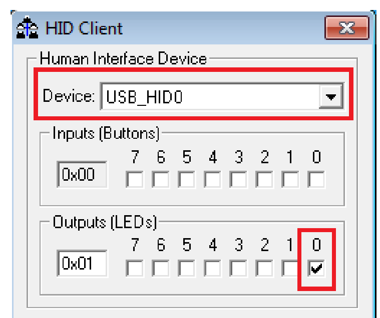
1. Double click on **USBD_User_HID_0.c** to open it for editing.
2. Near line 43, add the declaration:
   `extern void SetRecording(bool bMode);`
3. Near line 119, add the call:
   `SetRecording((*buf)&0x01);`

Note this call switches the state machine executed by our thread.

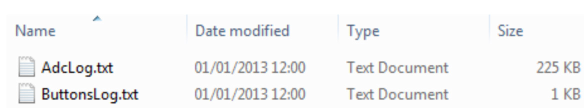
8. Select File/Save All or .
9. Compile the project:

Using the Data Logger

1. Program the Flash and enter Debug mode:
2. Click on RUN:
3. In order to send the command to start recording, run the HID Client software that is located at the following path:
   C:\Keil_v5\ARM\Utilities\HID_Client\Release\HIDClient.exe
4. In the **HID Client** window, select the **Device USB_HID0**.
5. Check the checkbox **"0"** under the **Outputs (LEDs)** tab. This action sends an HID output report to the board and the data logger starts recording.
   The Removable Disk is hidden from the computer because at this moment the data logger application has exclusive access to the file system. The LEDs blink alternately.
   The A/D Converter measures the input pin P14.0 and the values are recorded every 100ms.
   If you press buttons 1 and 2 in the board, the press and release events are recorded too.



6. Uncheck the checkbox **"0"** under the **Outputs (LEDs)** tab.
   The data logger stops recording and the LEDs turn off.
   In the computer the Removable Disk pops up again and the files can be accessed.
   The data are stored in text files: AdcLog.txt and ButtonsLog.txt.



Name	Date modified	Type	Size
AdcLog.txt	01/01/2013 12:00	Text Document	225 KB
ButtonsLog.txt	01/01/2013 12:00	Text Document	1 KB

```
00:06:17.80 - 0x0009 - 0.01V
00:06:17.90 - 0x01ED - 0.40V
00:06:18.00 - 0x0511 - 1.04V
00:06:18.10 - 0x0839 - 1.70V
00:06:18.20 - 0x09EC - 2.05V
00:06:18.30 - 0x0C8D - 2.59V
00:06:18.40 - 0x0F5B - 3.17V
00:06:18.50 - 0x0FFF - 3.30V
00:06:18.60 - 0x0FFF - 3.30V
```

```
00:06:20.30 - Button 1 Pressed
00:06:20.50 - Button 1 Released
00:06:21.20 - Button 2 Pressed
00:06:21.40 - Button 2 Released
```

7. Exit Debug mode:

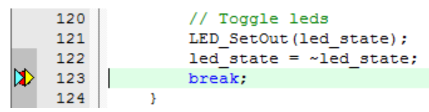
What we have at this point: The data logger application is complete. We can use the HID Client to start/stop data recording and we can read the logged data in the Removable Disk that automatically pops up.

Step 7: Improve Data Recording Operations

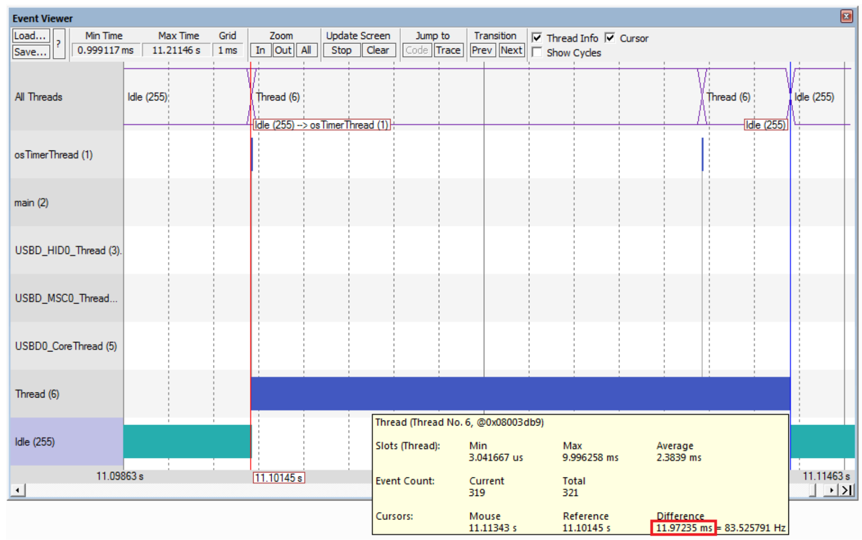
Analyzing the source code of the **Thread.c** file, where the data logger functionalities are implemented, we can see that files are being opened and closed every time an event is triggered. It's a robust implementation that reduces the risk of data loss since the **fclose** function effectively writes the file stream to the media and flushes the associated buffers.
However such implementation needs relatively high CPU resources as we can see in the Event Viewer.

Viewing RTX Activity with the Event Viewer:

1. Enter in Debug mode:  and RUN: 
2. Select the **Event Viewer** tab or if not already open: Select **Debug → OS Support → Event Viewer**.
3. Adjust the column width so the entire Thread names are visible as shown below. Data will be visible if the Serial Wire Viewer (SWV) is configured properly.



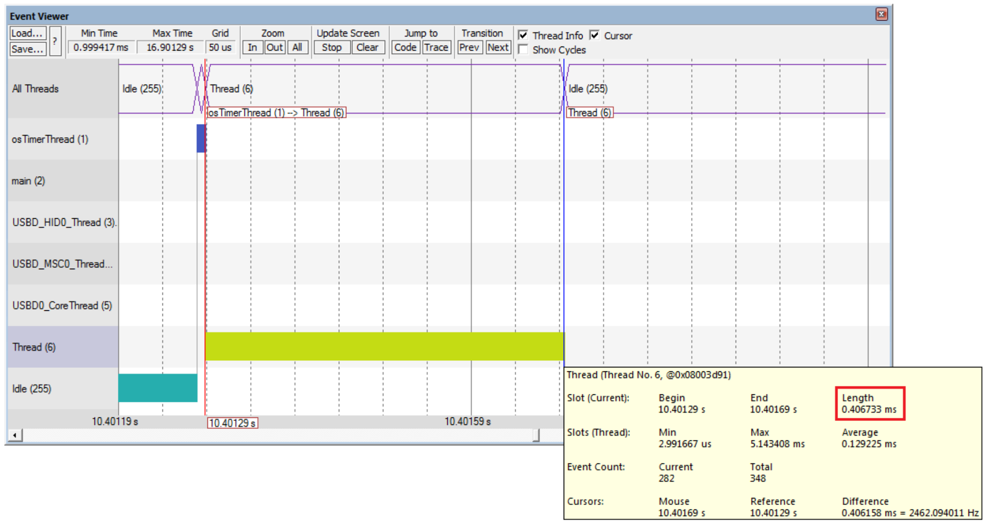
4. Start the data logger recording as described in the previous chapter.
5. Near line 123, at the end of the **DEV_RECORDING** case, set a breakpoint by clicking on the gray area:
6. Select **Stop** in the **Update Screen** box.
7. Set the grid using **Zoom In** and **Out**.
8. Scroll back and forth in time and you can see when the other threads were active.
9. Enable the **Cursor** and **Task Info** boxes to measure timings of these events.
   Note the Threads visible: The **Thread (6)** data shows the activity of this thread.
   In this example the thread takes about ~12ms for each file recording operation.
10. Clear the breakpoint by clicking on it.
11. Exit Debug mode: 



We can have a less resource-hungry implementation by calling the **fopen** and **fclose** functions only when respectively starting and stopping the data recording. Follow the steps:

1. Double click on **Thread.c** to open it for editing.
2. Near line 19, add the declarations:
```
FILE *f_but;
FILE *f_adc;
```
3. Near line 39, remove the local declaration:
```
FILE *f;
```
4. Near line 41, remove the call:
```
f = fopen("ButtonsLog.txt","a");
```
5. In the following line, replace **f** with **f_but**:
```
if (f_but != NULL) {
```
6. Near line 44, remove the call:
```
fclose (f);
```
7. Near line 48, remove the local declaration and the call:
```
FILE *f;
f = fopen("ButtonsLog.txt","a");
```

8. In the following line, replace **f** with **f_adc**:
   `if (f_adc != NULL) {`
9. Near line 51, remove the call:
   `fclose (f);`
10. Near line 82, after **fmount**, add the following calls:
    ```
    f_but = fopen("ButtonsLog.txt","a");
    f_adc = fopen("AdcLog.txt","a");
    ```
11. Near line 89, after **case DEV_STOP_RECORD:**, add the following calls:
    ```
    fclose (f_but);
    fclose (f_adc);
    ```
12. Select File/Save All or .
13. Compile the project:
14. Program the Flash and enter Debug mode:
15. Click on RUN.
16. Start the data recording and check the Thread optimization in the Event Viewer as previously described: now in this example it takes only about 0,4ms.
17. Exit Debug mode:



Serial Wire Viewer Summary

Serial Wire Viewer (SWV) is a 1 bit data trace. It is output on the SWO pin which is shared with the JTAG TDO pin. This means you cannot use JTAG and SWV together. Instead, use Serial Wire Debug (SWD or SW) which is a two pin alternative to JTAG and has about the same capabilities. SWD is selected inside the µVision IDE amd is easy to use.

1. The Core Clock: is the CPU frequency and must be set accurately. In this tutorial, 120 MHz is used. If you see ITM frames in the Trace Records window of number other than 0 or 31, or no frames at all, the clock frequency is probably wrong.
2. SWV is configured in the Cortex-M Target Setup in the Trace tab. **In Edit mode:** Select Target Options  or ALT-F7 and select the Debug tab. Select Settings: Then select the Trace tab. **In Debug mode:** Select Debug/Debug Settings.. and then select the Trace tab.
3. If SWV stops working, you can get it working by exiting and re-entering Debug mode. In rare cases, you might also have to cycle the board power.
4. SWV outputs its data over a 1 bit SWO pin. Overloading can be common depending on how much information you have selected to be displayed. Reduce the information to only that which you really need helps as does limiting the activity of variables. Using a ULINK*pro* on boards equipped with a 20 CoreSight ETM connector enables the SWV information to be output on the 4 bit ETM trace port.
5. For more information on XMC4500 Relax Kit see: http://www.keil.com/boards2/infineon/xmc4500relaxlitekit

---

Watch, Memory windows and Serial Wire Viewer can display:

- Global and Static variables. Raw addresses: i.e. *((unsigned long *)0x20000004)
- Structures.
- Peripheral registers – just read or write to them.
- Can’t see local variables. (just make them global or static).
- Cannot see DMA transfers – DMA bypasses CPU and CoreSight and CPU by definition.
- You might have to fully qualify your variables or copy them from the Symbol window.

Serial Wire Viewer (SWV) displays in various ways:

- PC Samples.
- A printf facility that does not use a UART.
- Data reads. Graphical format display in the Logic Analyzer: Up to 4 variables can be graphed.
- Exception and interrupt events.
- All these are Timestamped.
- CPU counters.

Instruction Trace (ETM):

- ETM Trace records where the program has been. Assembly instructions are all recorded.
- Assembly is linked to C source when available (this is up to your program).
- A recorded history of the program execution *in the order it happened.*
- Provides Performance Analysis and Code Coverage. Higher SWV performance.
- ETM needs a Keil ULINK*pro* to provide the connection to the 4 bit Trace Port.

Document Resources

Books

- ***NEW!*** **Getting Started MDK 5:** www.keil.com/mdk5/.
- A good list of books on ARM processors is found at: www.arm.com/support/resources/arm-books/index.php
- µVision contains a window titled **Books**. Many documents including data sheets are located there.
- A list of resources is located at: www.arm.com/products/processors/cortex-m/index.php (Resources tab).
- The Definitive Guide to the ARM Cortex-M0/M0+ by Joseph Yiu. Search the web for retailers.
- The Definitive Guide to the ARM Cortex-M3/M4 by Joseph Yiu. Search the web for retailers.
- Embedded Systems: Introduction to Arm Cortex-M Microcontrollers (3 volumes) by Jonathan Valvano.
- MOOC: Massive Open Online Class: University of Texas: http://users.ece.utexas.edu/~valvano/

Application Notes

1. Overview of application notes: www.keil.com/appnotes
2. ***NEW!*** Keil MDK for Functional Safety Applications: www.keil.com/safety
3. Using DAVE with µVision: www.keil.com/appnotes/files/apnt_258.pdf
1. Using Cortex-M3 and Cortex-M4 Fault Exceptions www.keil.com/appnotes/files/apnt209.pdf
2. CAN Primer using NXP LPC1700: www.keil.com/appnotes/files/apnt_247.pdf
3. CAN Primer using the STM32F Discovery Kit www.keil.com/appnotes/docs/apnt_236.asp
4. Segger emWin GUIBuilder with µVision™ www.keil.com/appnotes/files/apnt_234.pdf
5. Porting an mbed project to Keil MDK™ www.keil.com/appnotes/docs/apnt_207.asp
6. MDK-ARM™ Compiler Optimizations www.keil.com/appnotes/docs/apnt_202.asp
7. Using µVision with CodeSourcery GNU www.keil.com/appnotes/docs/apnt_199.asp
8. RTX CMSIS-RTOS in MDK 5 http://www.keil.com/pack/doc/cmsis_rtx/index.html
9. Lazy Stacking on the Cortex-M4 www.arm.com and search for DAI0298A
10. Sending ITM printf to external Windows applications: www.keil.com/appnotes/docs/apnt_240.asp
11. Barrier Instructions http://infocenter.arm.com/help/topic/com.arm.doc.dai0321a/index.html
12. Cortex Debug Connectors: http://www.keil.com/support/man/docs/ulinkpro/ulinkpro_cs_connectors.htm

Useful ARM Websites

1. Keil Forum: www.keil.com/forum
2. ARM Connected Community Forum: http://cc.arm.com/groups/tools
3. ARM University Program**:** www.arm.com/university
4. ARM Accredited Engineer Program: www.arm.com/aae
5. mbed™: http://mbed.org
6. CMSIS standard: www.arm.com/cmsis
7. CMSIS documentation: www.keil.com/cmsis

For comments or corrections on this document please email feedback@keil.com

Keil Products and Contact Information

Keil Microcontroller Development Kit (MDK-ARM™)

- MDK-Lite (Evaluation version) - $0
- MDK-ARM-CM™ (for Cortex-M series processors only – unlimited code limit)
- MDK-Standard (unlimited compile and debug code and data size Cortex-M, ARM7 and ARM9)
- MDK-Professional (Includes Flash File, TCP/IP, CAN and USB driver libraries and Graphic User Interface (GUI)
- **NEW!** ARM Compiler Qualification Kit: for Safety Certification Applications

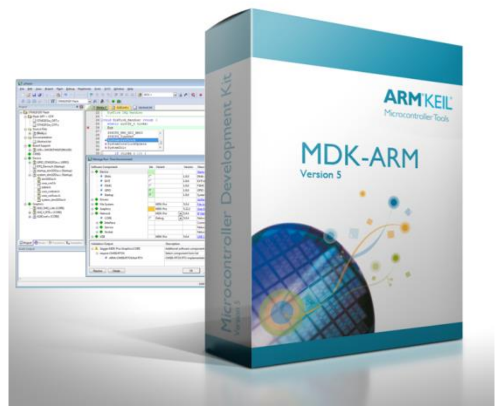
USB-JTAG adapter (for Flash programming too)

- ULINK2 - (ULINK2 and ME - SWV only – no ETM)
- ULINK-ME – sold only with a board by Keil or OEM.
- ULINK*pro* – Faster operation and Flash programming, Cortex-Mx SWV & ETM trace.
- **NEW!** ULINK*pro* D – Faster operation and Flash programming, Cortex-Mx SWV, no ETM trace.

For special promotional or quantity pricing and offers, please contact Keil Sales.

Contact sales.us@keil.com 800-348-8051 for USA prices.

Contact sales.intl@keil.com +49 89/456040-20 for pricing in other countries.



CMSIS-RTOS RTX is now provided under a BSD license. This makes it free.

All versions, including MDK-Lite, include CMSIS-RTOS RTX *with source code!*

Keil includes free DSP libraries for the Cortex-M0, M0+, M3, M4 and M7.

Call Keil Sales for details on current pricing, specials and quantity discounts. Sales can also provide advice about the various tools options available to you. They will help you find various labs and appnotes that are useful.

All products are available from stock.

All products include Technical Support for 1 year. This is easily renewed.

Call Keil Sales for special university pricing. Go to www.arm.com/university to view various programs and resources.

Keil supports many other Infineon processors including 8051 and C166 series processors. See the Keil Device Database® on www.keil.com/dd for the complete list of Infineon support. This information is also included in MDK.

For more information:

Keil Sales In USA: sales.us@keil.com or 800-348-8051. Outside the US: sales.intl@keil.com or +49 89/456040-20

Keil Technical Support in USA: support.us@keil.com or 800-348-8051. Outside the US: support.intl@keil.com.