

arm KEIL

## NXP Kinetis L Series: Cortex-M0+ Lab

### Using the Freedom KL25Z *featuring MTB: Micro Trace Buffer*

Arm Keil MDK 5 Toolkit Fall 2017 V 3.2 Robert Boys bob.boys@arm.com

The latest version of this document is here: [www.keil.com/appnotes/docs/apnt\\_232.asp](http://www.keil.com/appnotes/docs/apnt_232.asp)

### Introduction:

The purpose of this lab is to introduce you to the NXP Kinetis Cortex®-M0+ processor using the Arm® Keil® MDK toolkit featuring the IDE µVision®. We will demonstrate all debugging features available on this processor including Micro Trace Buffer (MTB). At the end of this tutorial, you will be able to confidently work with these processors and Keil MDK. We recommend you obtain the **new Getting Started MDK 5**: from here: [www.keil.com/gsg](http://www.keil.com/gsg).

Keil MDK supports and has examples for most NXP Arm processors. Check the [www.keil.com/NXP](http://www.keil.com/NXP) for the complete list.

**i.MX:** For i.MX support see DS-MDK. [www.keil.com/ds-mdk](http://www.keil.com/ds-mdk) Also: [www.arm.com/ds5](http://www.arm.com/ds5).

Keil MDK-Lite™ is a free evaluation version that limits code size to 32 Kbytes. Nearly all Keil examples will compile within this 32K limit. The addition of a valid license number will turn it into a commercial version.

### Why Use Keil MDK ?

MDK provides these features particularly suited for NXP Cortex-M users:

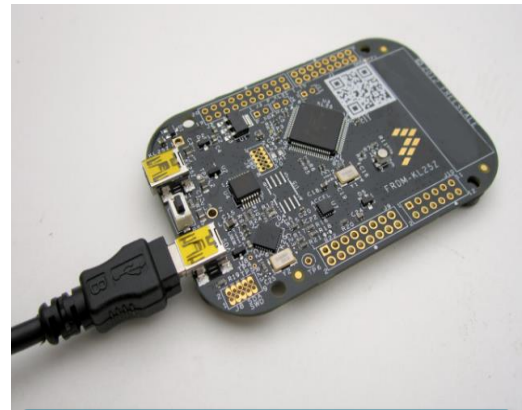
1. µVision IDE with Integrated Debugger, Flash programmer and the Arm® Compiler toolchain. MDK is turn-key "out-of-the-box".
2. Arm Compiler 5 and Arm Compiler 6 (LLVM) are included.
3. **Compiler Safety Certification Kit:** [www.keil.com/safety/](http://www.keil.com/safety/)
4. TÜV certified. SIL3 (IEC 61508) and ASILD (ISO 26262).
5. Dynamic Syntax checking on C/C++ source lines.
6. MISRA C/C++ support using PC-Lint. [www.gimpel.com](http://www.gimpel.com)
7. **Keil Middleware:** Network, USB, Flash File, Graphics and CAN for many NXP processors. Contact Keil Sales for assistance.
8. **Event Recorder for Middleware, RTX and User programs.**
9. RTX is included. RTX has a BSD or Apache 2.0 license with source code. FreeRTOS is now supported. [www.keil.com/RTX](http://www.keil.com/RTX) and [https://github.com/ARM-software/CMSIS\\_5](https://github.com/ARM-software/CMSIS_5)
10. Not on KL25: CoreSight™ Serial Wire Viewer (SWV). ETM instruction trace capability on appropriately equipped NXP processors. Provides Instruction Debugging, Code Coverage and Performance Analysis.
11. Debug Adapters: OpenSDA (CMSIS-DAP or P&E mode), Keil ULINK™2, ULINK-ME, ULINK<sub>pro</sub> and J-Link.
12. Affordable perpetual and term licensing with support. Contact Keil sales for pricing options. [Inside-Sales@arm.com](mailto:Inside-Sales@arm.com)
13. Keil Technical Support is included for one year and is renewable. This helps you get your project completed faster.
14. ULINK<sub>plus</sub> power analysis: [www.keil.com/mdk5/ulink/ulinkplus/](http://www.keil.com/mdk5/ulink/ulinkplus/) Available Nov 2018. Contact Keil sales.
15. Micrium µC/Probe compatible. [www.micrium.com/ucprobe](http://www.micrium.com/ucprobe) Displays/changes variables in graphical formats.

**This document includes details on these features plus more:**

1. Micro Trace Buffer (MTB). Instruction trace. A history where your program has been: the executed instructions.
2. Real-time Read and Write to memory locations for the Watch, Memory and peripheral windows. These are non-intrusive to your program. No CPU cycles are stolen. No instrumentation code is added to your source files.
3. Two Hardware Breakpoints (can be set/unset on-the-fly) and two Watchpoints (also known as Access Breaks).
4. RTX System & Threads window: a kernel awareness program for RTX that updates while your program is running.
5. A DSP example program using Arm CMSIS-DSP libraries.
6. How to create your own µVision projects and an extensive list of document resources available.

### Micro Trace Buffer (MTB):

MDK supports MTB with OpenSDA (in CMSIS-DAP mode), ULINK2/ME or ULINK<sub>pro</sub>. MTB provides instruction trace which is essential for solving program flow and other related problems. How to use MTB is described in this document.



The Freedom KL25Z board connected to run OpenSDA (CMSIS-DAP) and MDK.

## **General Information:**

1. NXP Evaluation Boards & Keil Evaluation Software: 3
2. MDK 5 Keil Software Download and Installation: 3
3. OpenSDA: An NXP Debug Adapter 3
4. Install Keil MDK Software: 3
5. Install Software Packs and Examples: 4

## **Using the OpenSDA CMSIS-DAP Debug Adapter:**

6. Programming the KL25Z Board with OpenSDA (CMSIS-DAP) 5
7. Testing the OpenSDA Installation: 5

## **Blinky Example and Debugging Features:**

8. *Blinky* example using the Freedom KL25Z and OpenSDA: 6
9. Hardware Breakpoints: 6
10. Call Stack & Locals window: 7
11. Watch and Memory windows and how to use them: 8
12. System Viewer (SV): 9
13. Watchpoints: Conditional Breakpoints: 10
14. RTX Kernel Awareness with System and Threads Viewer: 11

## **MTB: Micro Trace Buffer with Blinky:**

15. MTB: Micro Trace Buffer: 12
16. Cod Coverage: 13
17. Exploring the MTB Instruction Trace: 14
18. Trace Buffer Configuration and Control: 15
19. Trace Search: 16
20. Trace Data Wrap Around: 16
21. More MTB Exploration: 17
22. Trace “In the Weeds” Example: 18

## **DSP Sine Example:**

23. DSP Sine using Arm CMSIS-DSP Libraries: 19

## **Creating Your Own MDK Project With and Without RTOS:**

24. Creating Your Own MDK 5 Project from Scratch: 20
25. Adding RTX: 23
26. Adding a Thread: 24
27. Event Recorder 25

## **General Information:**

28. Interesting Bits & Pieces: 26
29. Kinetis KL25 Trace Summary: 27
30. CoreSight Definitions: 28
31. Document Resources: 29
32. Keil Products and Contact Information: 30

## 1) NXP Evaluation Boards & Keil Evaluation Software:

Keil provides board support for Kinetis Cortex-M0+ and Cortex-M4 processors. They include Tower K20, K40, K53, K60, K70 and KL25Z (both Tower and Freedom boards), S32K and many more. See [www.keil.com/NXP](http://www.keil.com/NXP) for the complete list.

This lab was written using a KL25Z Freedom board with OpenSDA in CMSIS-DAP mode. This lab will also work for the TWR-KL28Z72M board. It has a slightly different example program.

For the i.MX series see [www.keil.com/mdk5/ds-mdk/](http://www.keil.com/mdk5/ds-mdk/)

On the last page of this document is an extensive list of resources that will help you successfully create your projects. This list includes application notes, books and labs and tutorials for other NXP boards.

We recommend you obtain the latest Getting Started Guide for MDK5: It is available on [www.keil.com/gsg/](http://www.keil.com/gsg/).

**Arm forums:** <https://developer.arm.com>

**Keil Forums:** [www.keil.com/forum/](http://www.keil.com/forum/)

---

## 2) MDK 5 Keil Software Information: *This document used MDK 5.41*

MDK 5 Core is the heart of the MDK toolchain. This initially will be in the form of MDK Lite which is the evaluation version. The addition of a Keil license will turn it into one of the commercial versions. Contact Keil Sales for more information.

Device and board support are distributed via Software Packs. These Packs are downloaded from the web with the "Pack Installer", the version(s) selected with "Select Software Packs" and your project configured with the "Run Time Environment" (RTE) utilities. These are components of  $\mu$ Vision. You can distribute and install your own Pack for confidentiality.

A Software Pack is an ordinary .zip file with the extension changed to .pack. It contains various header, Flash programming and example files and more. Contents of a Pack is described by a .pdsc file in XML format.

See [www.keil.com/dd2/pack](http://www.keil.com/dd2/pack) for the current list of available Software Packs. More Packs are being added.

**Example Project Files:** This document uses the RTX5\_Blinky example project contained in the S32K Software Pack.

---

**3) OpenSDA:** OpenSDA is NXP's on-board debug adapter used extensively in the Kinetis and other families. It has a P&E and a CMSIS-DAP mode. CMSIS-DAP is an Arm standard. This lab will use the KL25Z board in CMSIS-DAP mode. LPC-Link2 is also CMSIS-DAP compliant. LPC-Link2 has a J-Link mode which is also supported by  $\mu$ Vision.

You are able to incorporate CMSIS-DAP debugger on your own board. See [https://github.com/ARM-software/CMSIS\\_5](https://github.com/ARM-software/CMSIS_5). You do not need an external debugger such as a ULINK2 to do this lab. If you use an external debugger, you must populate SWD J6 connector with a 10 pin CoreSight connector made by Samtec or equivalent FTSH-105-01-L-D-K.

---

## 4) Install Keil MDK Software:

1. Download MDK Core 5.41 or later from the Keil website. <http://www2.keil.com/mdk5/install>
2. Install MDK into the default folder. You can install into any folder, but this lab uses the default C:\Keil\_v5
3. We recommend you use the default folders for this tutorial. We will use C:\00MDK\ for the examples.
4. If you install MDK into a different folder, you will have to adjust for the folder location differences.
5. You do not need an external debug adapter: just the KL25Z board, a USB cable and MDK installed on your PC.
6. You do not need a Keil MDK license for this tutorial. All examples will compile within the 32 K limit.

## 5) Install µVision Software Packs and Examples:

### 1) Start µVision and open Pack Installer:

1. Connect your computer to the internet. This is needed to download the Software Packs.


2. Start µVision by clicking on its desktop icon.

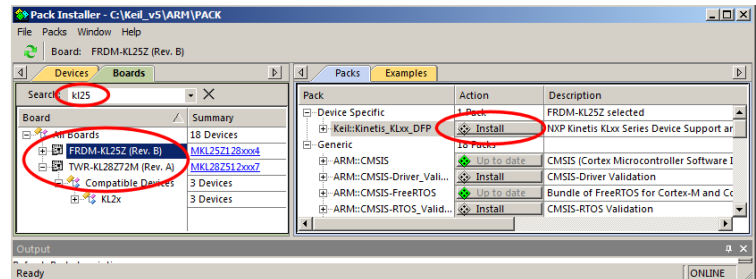
3. Open the Pack Installer by clicking on its icon:

A Pack Installer Welcome screen will open. Read and close it.

4. This window opens up: Select the Packs tab:

5. Note "ONLINE" is displayed at the bottom right. If "OFFLINE" is displayed, connect to the Internet before continuing.

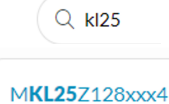
6. If there are no entries shown because you were not connected to the Internet when Pack Installer opened, select Packs/Check for Updates or  to refresh once you have connected to the Internet.



### 2) Install The KL25 Software Pack:

The KL25Z is supported in older Keil Packs which are not available directly through Pack Installer.

1. Go to <https://www.keil.com/dd2/pack/>
2. Type KL25 in the search box and press enter.
3. Click on MKL25Z128xxx4.



4. Click on CMSIS Pack Kinetis\_KLxx\_DFP.
5. Click on Download Recommended Pack.
6. Use Pack Installer program to import that file using the File->Import... commands.
7. On the left pane, select "Boards" and All Boards ->FRDM-KL25Z.

### 3) Install the RTX\_Blinky Example:

The code for steps 3) and 4) is out of date. There is up-to-date code (Blinky and Blinky\_BM) in the class Github repository in PHW/PHW1 (for **ECE 460/560**) or HW/HW0 (for **ECE 461/561**). When you use git to clone or pull updates from the repository, these will be copied to your PC.

### 4) Install the RTX5\_Blinky and DSP5 Examples from Keil.com:

Skip this step, as the PHW1 repository provides the code.

## ***Skip Step 6) unless you are using your personal FRDM-KL25Z.***

### **6) Programming the KL25Z with OpenSDA: an on-board Debug Adapter:**

This document will use OpenSDA as a SWD Debug Adapter. Target connection by  $\mu$ Vision will be via a standard USB cable connected to SDA J7. The on-board Kinetis K20 acts as the debug adapter. Micro Trace Buffer frames can be displayed.

***This Step MUST be done ! at least once...***

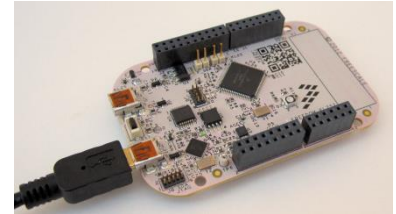
**Program the K20 with the CMSIS-DAP application file CMSIS-DAP.S19:**

#### **1) Locate the file CMSIS-DAP.S19:**

1. CMSIS-DAP.S19 is located in the OpenSDA directory in MDK KL25 projects. Using Windows Explorer navigate to **Blinky\_BM\OpenSDA or Blinky\OpenSDA.** CMSIS-DAP.S19 is located here. The other projects also contain this file. You will copy this file into the Freedom board USB device as described below.

#### **2) Put the Freedom Board into Bootloader: Mode:**

2. Hold RESET button SW1 on the Freedom board down and connect a USB cable to J7 SDA as shown here:
3. When you hear the USB dual-tone, release RESET.
4. The green led D4 will blink about once per second. The Freedom is now ready to be programmed with the CMSIS-DAP application.
5. The Freedom will act as a USB mass storage device called BOOTLOADER connected to your PC. Open this USB device with Windows Explorer.



#### **3) Copy CMSIS-DAP.S19 into the Freedom Board:**

6. Copy and paste or drag and drop CMSIS-DAP.S19 into this Bootloader USB device.

#### **4) Exit Bootloader Mode:**


7. Cycle the power to the Freedom board while **not** holding RESET button down. The green led will blink once and then stay off.
8. The Freedom board is now ready to be used with the  $\mu$ Vision debugger and Flash programmer.

**TIP:** The green led will indicate when  $\mu$ Vision is in Debug mode and connected to the OpenSDA debug port SWD.


Remember, JTAG is not used. The Kinetis Cortex-M0+ has only the SWD port. You can do everything with the SWD port as you can with a JTAG port. SWD is referenced as SW in the  $\mu$ Vision configuration menu.

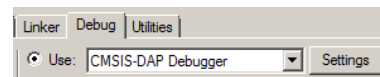
**TIP:** This application will remain in the U6 K20 Flash each time the board power is cycled with RESET off. The next time board is powered with RESET held on, it will be erased. CMSIS-DAP.S19 is the CMSIS DAP application in the Motorola S record format that loads and runs on the K20 OpenSDA processor.

### **7) Testing The OpenSDA Connection: (Optional Exercise)**

1. Start  $\mu$ Vision  if it is not already running. Select Project/Open Project.
2. Select the **Blinky or Blinky\_BM project.**
3. Select "CMSIS-DAP" in the Select Target menu. If you do not have this entry – select anything.

CMSIS-DAP

4. Select Target Options  or ALT-F7 and select the Debug tab:
5. Select CMSIS-DAP Debugger as shown here:



6. Click on Settings: and the window below opens up: If an IDCODE and Device name is displayed, OpenSDA is working. You can continue with the tutorial. Click on OK twice to return to the  $\mu$ Vision main menu.
7. If nothing or an error is displayed in this SW Device box, this **must** be corrected before you can continue.

**TIP:** You can use this test to confirm the operation of any debug adapter selected in the Debug tab as shown above right.

**TIP:** To refresh the SW Device box, in the Port: box select JTAG and then select SW again. You can also exit then re-enter this window. CMSIS-DAP will not work with JTAG selected, only SW. But this is a useful way to refresh the SW setting.



**Screenshot #1  
must show  
IDCODE**

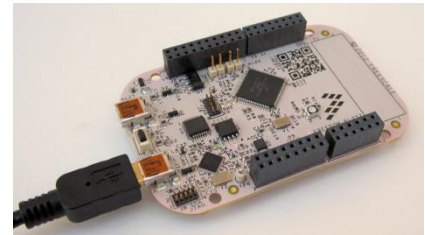


## 8) Blinky example program using the NXP Freedom KL25Z and OpenSDA:

Now we will connect a Keil MDK development system using the Freedom board and OpenSDA in CMSIS-DAP mode. Your board *must* have the application CMSIS-DAP.S19 programmed into the OpenSDA processor before you can continue.

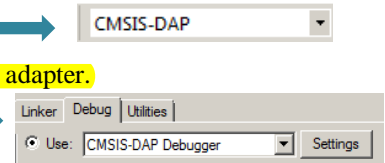
1. Connect a USB cable between your PC and Freedom SDA J7 as shown here:
2. Start  $\mu$ Vision by clicking on its desktop icon.
3. Select Project/Open Project.
4. Open the Blinky file:

PHW1\Blinky\Blinky.uvprojx



### Create Target Options for OpenSDA (in CMSIS-DAP mode):

1. Select Project/Manage/Project Items... or select:
2. In the Project Targets area, select NEW or press your keyboard INSERT key.
3. Enter CMSIS-DAP and press Enter. Click OK to close this window.
4. In the Target Selector menu, select the CMSIS-DAP selection you just made:
5. Select Options for Target or ALT-F7. Click on the Debug tab to select a debug adapter.
6. Select CMSIS-DAP Debugger... as shown here: <an important step>
7. This is where you create and select different target configurations such as to execute a program in RAM or Flash and many other settings.



**TIP:** If you click Settings:, you can test the debug connection as described on the previous page.

8. Click on OK to return to the  $\mu$ Vision main menu. Click OK twice if you clicked Settings:. Select File/Save All

### Compile and RUN the Blinky Project:

1. Compile the source files by clicking on the Rebuild icon. You can also use the Build icon beside it.
2. Enter Debug mode by clicking on the Debug icon. Select OK if the Evaluation Mode box appears.
3. Click on the RUN icon. **Note:** you can stop the program with the STOP icon.

**Screenshot #2**  
with Build Output  
window showing  
compiler version

**The three colour LED D3 on the Freedom board will now blink in sequence.**

**Now you know how to compile a program, program it into the KL25Z processor Flash, run it and stop it !**

**Note:** The board will start Blinky stand-alone. Blinky is now permanently programmed in the Flash until reprogrammed.

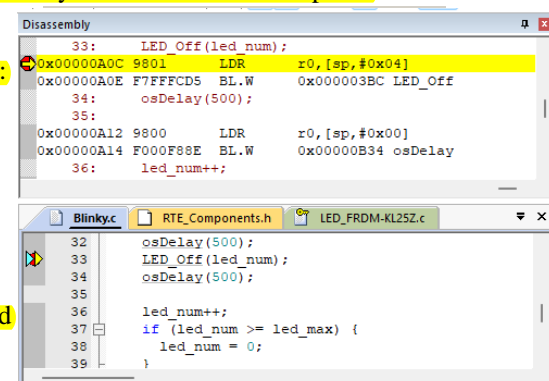
## 9) Hardware Breakpoints:

The KL25Z has two hardware breakpoints that can be set or unset on the fly while the program is running.

1. With Blinky running, in the Blinky.c window, click on a darker grey block on the left on a suitable part of the source code. This means assembly instructions are present at these points. Inside the while loop inside the thread Thread\_LED which itself starts near line 36. You can also click in the Disassembly window to set a breakpoint.
2. A red circle will appear and the program will presently stop.
3. Note the breakpoint is displayed in both the Disassembly and source window:
4. Set a second breakpoint in the while() loop as before.
5. Every time you click on the RUN icon the program will run until the breakpoint is again encountered.
6. Remove the breakpoints by clicking on them.

**TIP:** If you set too many breakpoints,  $\mu$ Vision will warn you.

**TIP:** Arm hardware breakpoints do **not** execute the instruction they are set to and land on. CoreSight hardware breakpoints are no-skid. This is an important feature.







## 10) Call Stack + Locals Window:

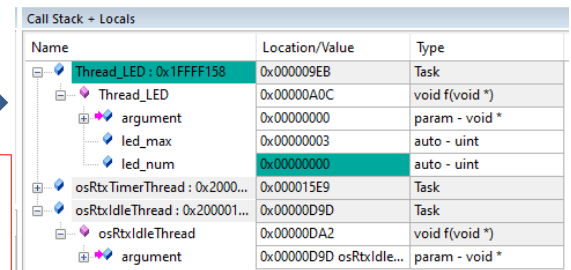
### Local Variables:

The Call Stack and Locals windows are incorporated into one integrated window. Whenever the program is stopped, the Call Stack + Locals window will display call stack contents as well as any local variables located in the active function.

If possible, the values of the local variables will be displayed and if not the message <not in scope> will be displayed. The Call + Stack window presence or visibility can be toggled by selecting View/Call Stack Window in the main µVision window when in Debug mode.



1. Click on RUN .
2. Set a breakpoint in Blinky.c in main() on the `osDelay` statement near line 34. 
3. It will soon stop on this breakpoint.
4. Click on the Call Stack + Locals tab to open it.
5. Shown is this Call Stack + Locals window: It is stopped in `Thread_LED()` .
6. The functions as they were called are displayed. If functions have local variables in scope, they will be displayed.
7. Click on the Step In icon  or F11 a few times:

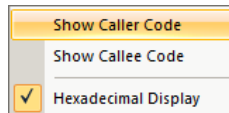
**Screenshot #3**  
with Call Stack  
& Locals



Name	Location/Value	Type
Thread_LED : 0x1FFFF158	0x000009EB	Task
Thread_LED	0x00000A0C	void f(void *)
argument	0x00000000	param - void *
led_max	0x00000003	auto - uint
led_num	0x00000000	auto - uint
osRtxTimerThread : 0x2000...	0x000015E9	Task
osRtxIdleThread : 0x200001...	0x00000D9D	Task
osRtxIdleThread	0x00000DA2	void f(void *)
argument	0x00000D9D osRtxIdle...	param - void *

**TIP:** You can use the Call Stack & Locals window for debugging. Use MTB instruction trace in case the stack is corrupted.

8. You can see functions listed with any variables displayed. 
9. As you step in and out of functions, they are added and then removed.
10. (deleted) A sample Call Stack window is shown below:
11. Note Thread\_LED stays listed. Ordinary functions and interrupt handlers come and go but Threads stay listed.
12. Right click on a function entry and select Show Callee or Caller Code to display the associated code. 
13. When you ready to continue, remove the hardware breakpoint by clicking on its red circle ! You can also type Ctrl-B, select Kill All and then Close.



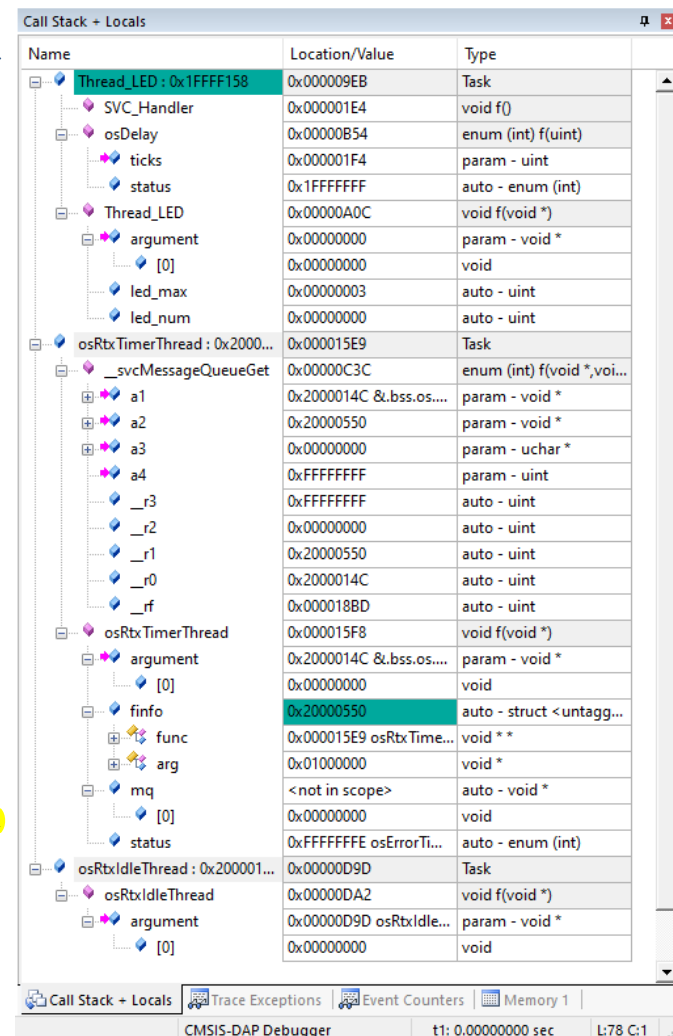
**TIP:** You can modify a variable value in the Call Stack & Locals window when the program is stopped.

**TIP:** You can set two hardware breakpoints with the NXP Cortex-M0+ processor. If you set more than two, or you have two set and the debugger needs one for an operation, µVision will warn you to delete the excessive breakpoints. The Kinetis Cortex-M4 family of processors has 6 hardware breakpoints available.

If Run to Main is selected in the Target Debug tab, at RESET, µVision will set one temporarily at the beginning main(). If you already have two breakpoints set, you will get an error message at runtime.

**Do not forget to remove any hardware breakpoints before continuing.**

**TIP:** To locate the definition of a variable, structure, function or define, right click on it and select Go To Definition...



Name	Location/Value	Type
Thread_LED : 0x1FFFF158	0x000009EB	Task
SVC_Handler	0x000001E4	void f()
osDelay	0x00000B54	enum (int) f(uint)
ticks	0x000001F4	param - uint
status	0x1FFFFFFF	auto - enum (int)
Thread_LED	0x00000A0C	void f(void *)
argument	0x00000000	param - void *
[0]	0x00000000	void
led_max	0x00000003	auto - uint
led_num	0x00000000	auto - uint
osRtxTimerThread : 0x2000...	0x000015E9	Task
_svcMessageQueueGet	0x00000C3C	enum (int) f(void *,voi...
a1	0x2000014C &.bss.os....	param - void *
a2	0x20000550	param - void *
a3	0x00000000	param - uchar *
a4	0xFFFFFFFF	param - uint
_r3	0xFFFFFFFF	auto - uint
_r2	0x00000000	auto - uint
_r1	0x20000550	auto - uint
_r0	0x2000014C	auto - uint
_rf	0x000018BD	auto - uint
osRtxTimerThread	0x000015F8	void f(void *)
argument	0x2000014C &.bss.os....	param - void *
[0]	0x00000000	void
finfo	0x20000550	auto - struct <untagg...
func	0x000015E9 osRtxTime...	void * *
arg	0x01000000	void * *
mq	<not in scope>	auto - void *
[0]	0x00000000	void
status	0xFFFFFFFF osErrorTi...	auto - enum (int)
osRtxIdleThread : 0x200001...	0x00000D9D	Task
osRtxIdleThread	0x00000DA2	void f(void *)
argument	0x00000D9D osRtxIdle...	param - void *
[0]	0x00000000	void



## 11) Watch and Memory Windows and how to use them:

The Watch and Memory windows will display updated variable values in real-time. It does this using the CoreSight DAP debugging technology that is part of Cortex-M processors. It is also possible to “put” or insert values into the Memory window in real-time. It is possible to “drag and drop” variable names into windows or enter them manually. You can also right click on a variable and select Add *varname* to.. and select the appropriate window.

Watch or Memory windows are unable to display local variables. Use global, static, peripheral or a structure: anything that is always in scope. If a local variable is in scope when the program is stopped, its value will be displayed.

### Watch window:

**Add a global variable:** Call Stack, Watch and Memory windows can't see local variables unless stopped in their function.





1. Stop the processor  and exit Debug mode. 
2. Declare a global variable (I called it **counter**) near line **25** in Blinky.c:

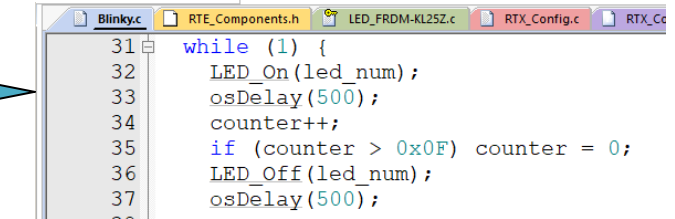
```
unsigned int counter = 0;
```

3. Add the statements near Line **33** just after `osDelay(500);`:

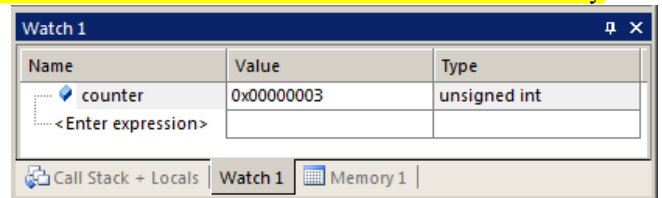
```
counter++;
```

```
if (counter > 0xF) counter = 0;
```

4. Select File/Save All or 
5. Click on Rebuild . There will be no errors. If there are, please fix them.
6. Enter Debug mode.  Click on RUN . You can configure a Watch window while the program is running. You can also do this with a Memory window.
7. Select View/Periodic Window Update if windows update only when the program stops.: ☒ Periodic Window Update
8. In Blinky.c, right click on **counter** and select Add counter to ... and select Watch 1. Watch 1 will automatically open. **counter** will be displayed as shown here:
9. **counter** will update in real time.



```
31 while (1) {
32     LED_On(led_num);
33     osDelay(500);
34     counter++;
35     if (counter > 0x0F) counter = 0;
36     LED_Off(led_num);
37     osDelay(500);
38 }
```



Name	Value	Type
counter	0x00000003	unsigned int
<Enter expression>		

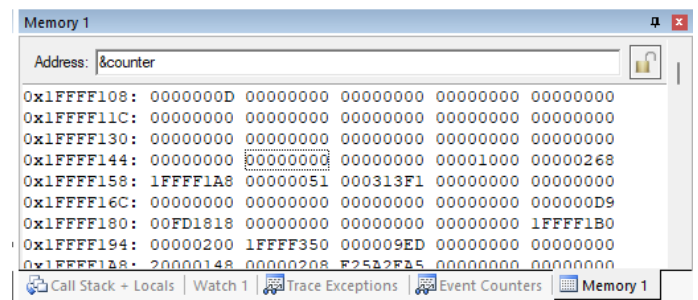
**TIP:** You can modify a variable in a Watch window if it is stopped or changing very slowly.

**TIP:** To Drag 'n Drop into a tab that is not active, pick up the variable and hold it over the tab you want to open; when it opens, move your mouse into the window and release the variable.

**TIP:** Want to know the speed of the CPU ? Enter `SystemCoreClock` into a Watch window.

### Memory window:

1. Right click on **counter** and select Add counter to ... and select the Memory 1 window.
2. Note the value of **counter** is displaying its address in Memory 1 as if it is a pointer. This is useful to see what address a pointer is pointing to but this not what we want to see at this time.
3. Add an ampersand "&" in front of the variable name and press Enter. The physical address is shown: `0x1FFF_F000`.
4. Right click in the Memory window and select Unsigned/Int.
5. The data contents of **counter** is displayed as shown here:
6. Both the Watch and Memory windows are updated periodically in real-time.
7. Right-click with the mouse cursor over the desired data field and select Modify Memory. You can change a memory or variable on-the-fly while the program is still running.



Address	Value
0x1FFF108	0000000D
0x1FFF11C	00000000
0x1FFF130	00000000
0x1FFF144	00000000
0x1FFF158	1FFFF1A8
0x1FFF16C	00000000
0x1FFF180	00FD1818
0x1FFF194	00000200
0x1FFF1A8	20000148

**TIP:** No CPU cycles are used to perform these operations. No code stubs are added to your sources.



## 12) System Viewer (SV):

The System Viewer provides the ability to view registers in the CPU core and in peripherals. In most cases, these Views are updated in real-time while your program is running. These Views are available only while in Debug mode. There are two ways to access these Views: a) View/System Viewer and b) Peripherals/System Viewer. In the Peripheral/Viewer menu, the Core Peripherals are also available: Note the various peripherals available.

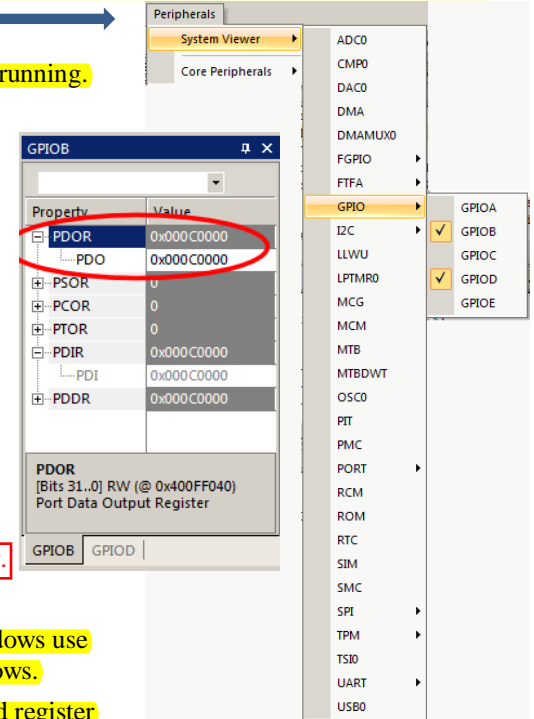
1. Click on RUN . You can open SV windows when the program is running.

### GPIO Port B:


2. Select Peripherals/System Viewer and then GPIO and select GPIOB.
3. This window opens up. Expand PDO:
4. You can now see PDOR and PDO update:
5. You can also open Port D as one LED is connected to this port.

**TIP:** Two LEDs are on Port B and the other (blue) is connected to Port D.

**TIP:** If you click on a register in the properties column, a description about this register will appear at the bottom of the window as shown:

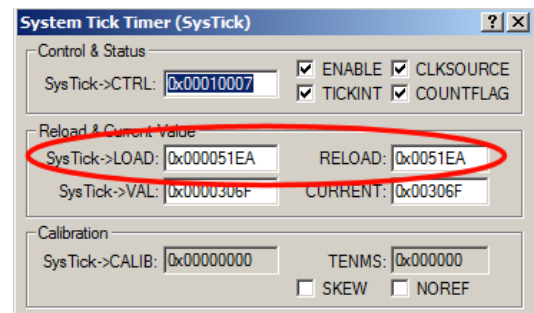


### SysTick Timer: RTX uses the SysTick timer for its switching.

1. Select Peripherals/Core Peripherals and then select System Tick Timer.
2. The SysTick window shown below opens:
3. Note it also updates in real-time while your program runs. These windows use the same CoreSight DAP technology as the Watch and Memory windows.
4. Note the SysTick->Load and RELOAD boxes. This is the timer reload register value. This is set in the RTX configuration file RTX\_Conf\_CM.c.
5. Note that it is set to 0x51EA. The CPU frequency is 20,971,520 MHz. You can check this by putting SystemCoreClock in a Watch window.
6. This is the same value hex value of  $20,971,520/1000-1 = 20,970$  or 0x51EA. This is where this value comes from. Changing the variable passed to this function is how you change how often the SysTick timer creates its interrupt 15.
7. In the RELOAD register in the SysTick window, while the program is running, type in 0x1000 and press Enter !
8. The blinking LEDs will speed up. This will convince you of the power of Arm CoreSight debugging.
9. Replace RELOAD with 0x51EA. You might have to click RESET and then RUN.
10. You can look at other Peripherals contained in the System View windows.
11. When you are done, stop the program  and close all the System Viewer windows that are open.




**TIP:** It is true: you can modify values in the SV while the program is running. This is very useful for making slight timing value changes instead of the usual modify, compile, program, run cycle.

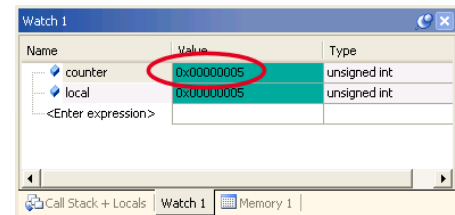
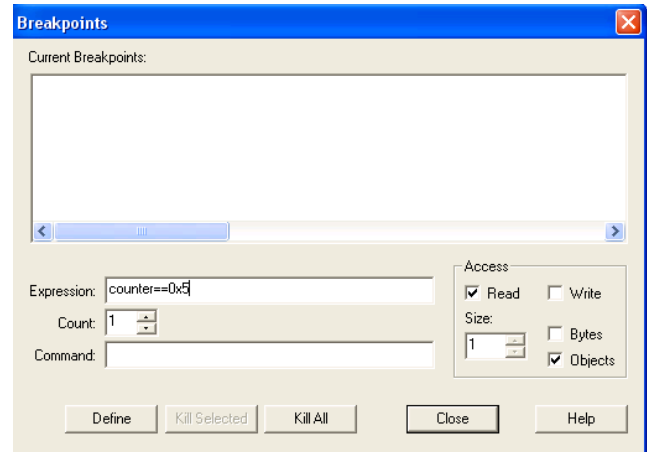
You must make sure a given peripheral register allows and will properly react to such a change. Changing such values indiscriminately is a good way to cause serious and difficult to find problems.



### 13) Watchpoints: Conditional Breakpoints

The KL25 Cortex-M0+ processor has two Watchpoints. Watchpoints can be thought of as conditional breakpoints. Watchpoints are also referred to as Access Breaks in Keil documents. Cortex-M0+ Watchpoints are intrusive with a data test. When the Watchpoint is hit,  $\mu$ Vision must test the memory location. Cortex-M3/M4 Watchpoints are not intrusive for address only equality test. This can be useful for testing to see if a Stack or Heap passed a certain point.

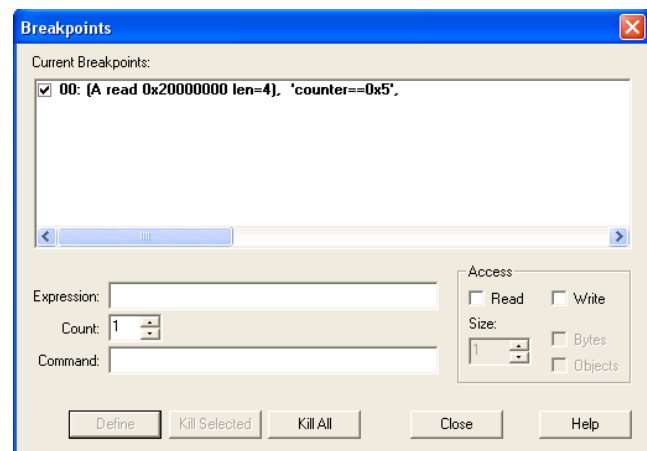
1. Use the same Blinky configuration as the previous page. Stop the program if necessary. Stay in Debug mode.
2. We will use the global variable `counter` you created in Blinky.c to explore Watchpoints.
3. Select Debug in the main  $\mu$ Vision window and then select Breakpoints or press Ctrl-B.
4. Select Access to Read.
5. In the Expression box enter: "`counter == 0x5`" without the quotes. This window will display:
6. Click on Define or press Enter and the expression will be accepted as shown below in the bottom Breakpoints window:
7. Click on Close.
8. Enter the variable `counter` in Watch 1 if it is not already there.
9. Click on RUN, .
10. `counter` might not update in the Watch window depending on how fast the variable is changed. This feature is turned off in  $\mu$ Vision for speed considerations. You will also notice the program slows down. This is because  $\mu$ Vision must test the condition when the write or read occurs to `counter`. Minimize this by selecting only Read or Write Access.
11. When `counter` equals 0x5, the Watchpoint will stop the program. See Watch 1 shown below:
12. Watch expressions you can enter are detailed in the Help button in the Breakpoints window. Triggering on a data read or write is most common. You can leave out the value and trigger on just a Read and/or Write as you select.
13. To repeat this exercise, change `counter` to something other than 0x05 in the Watch window and click on RUN.
14. Stop the CPU, .
15. Select Debug/Breakpoints (or Ctrl-B) and delete the Watchpoint with Kill All and select Close.
16. Exit Debug mode, .



**TIP:** To edit a Watchpoint, double-click on it in the Breakpoints window and its information will be dropped down into the configuration area. Clicking on Define will create another Watchpoint. You should delete the old one by highlighting it and click on Kill Selected or try the next TIP:

**TIP:** The checkbox beside the expression allows you to temporarily unselect or disable a Watchpoint without deleting it.

**TIP:** Raw addresses can be used with a Watchpoint. An example is: `*((unsigned long *)0x20000004)`



## 14) RTX Kernel Awareness using RTX RTOS Watch Window

Users often want to know the number of the current operating task and the status of the other tasks. This information is usually stored in a structure or memory area by the RTOS. Keil provides a Task Aware window for RTX. Keil supports FreeRTOS with a kernel awareness window. See [www.keil.com/pr/article/1280.htm](http://www.keil.com/pr/article/1280.htm).

1. Run **Blinky** by clicking on the Run icon.
2. Open View/Watch Windows and select RTX RTOS.
3. The window below opens up. Note these values are updating in real-time using the same CoreSight DAP read & write technology as used in the Watch and Memory windows.
4. Select View and select Periodic Window Update if these values do not change: ☒ Periodic Window Update
5. You will not have to stop the program to view this data. No CPU cycles are used. Your program runs at full speed. No instrumentation code needs to be inserted into your source. Most of the time the CPU is executing the **osRtxIdleThread**. The processor spends relatively little time in each task. You can change this to suit your needs. There are **two threads plus the idle thread**. You can easily add threads.

RTX RTOS	
Property	Value
System	
Threads	
id: 0x200001D8 "osRtxIdleThread"	osThreadRunning, osPriorityIdle, Stack Used: unknown
id: 0x2000021C "osRtxTimerThread"	osThreadBlocked, osPriorityHigh, Stack Used: 18%
id: 0x1FFFF160 "Thread_LED"	osThreadBlocked, osPriorityNormal, Stack Used: 18%
State	osThreadBlocked
Priority	osPriorityNormal
Attributes	osThreadDetached
Waiting	Delay, Timeout: 146
Stack	Used: 18% [96]
Flags	0x00000000
Message Queues	
id: 0x20000154	Messages: 0, Max: 4

**Demonstrating States: Note: Keil uses the term Threads instead of Tasks for consistency.**

Blinky.c contains one thread that blink the LEDs. Thread\_LED is shown below:

1. The gray areas opposite the line numbers indicate there is valid assembly code located here.
2. Set a breakpoint on one of these in Thread\_LED as shown:
3. (deleted)
4. Click on RUN.
5. When the program stops, this information will be updated in the **RTX RTOS watch window**. The Task running when the program stopped will be indicated with **osThreadRunning**. The window above shows the program stopped and **osRtxIdleThread** running. The states of the other tasks are displayed as well as other useful information.
6. Click on RUN. The program will run until reaching a breakpoint or being stopped.
7. Remove the breakpoints and close the RTX Tasks window. Exit Debug mode.

```
27 void Thread_LED (void *argument) {
28     uint32_t led_max = LED_GetCount();
29     uint32_t led_num = 0;
30
31     while (1) {
32         LED_On(led_num);
33         osDelay(500);
34         counter++;
35         if (counter > 0x0F) counter = 0;
36         LED_Off(led_num);
37         osDelay(500);
38     }
```

### More Information of using RTX:

It is very beneficial to use an RTOS. RTX is a good choice. It is small, efficient and easy to use yet it is full featured. All source code and documentation is provided with RTX. [www.keil.com/RTX](http://www.keil.com/RTX) Getting Started Guide: [www.keil.com/gsg](http://www.keil.com/gsg) RTX has a BSD license and comes with all source code. The new RTX v5 has an Apache 2.0 license and is available inside MDK or on GitHub: [https://github.com/ARM-software/CMSIS\\_5](https://github.com/ARM-software/CMSIS_5) Event Viewer is another RTX viewer. This is not supported in Cortex-M0 because it has no SWV.

## 15) MTB: Micro Trace Buffer:

The Kinetis KL25 processor contains an instruction trace called MTB. The trace buffer is a portion of the KL25 internal RAM which your program must not use. The trace frames are stored here. The size of this buffer is set in the file DBG\_MTB.ini.

Since there is not an unlimited amount of RAM available for MTB, the trace will be small and wraps around. Judicious use of hardware breakpoints and other techniques such as stopping with a Watchpoint will help in this regard. Code Coverage is available with MTB as it is with ETM trace found in other Cortex-M processors.

Instruction trace is very valuable in finding program flow bugs and other issues such as, but not limited to: race conditions and program crashes. Trace is useful to examine the program flow such as when branches occurred (or not) and interrupt calls.

**TIP:** The source code and Disassembly windows show the code as it was written. The MTB Trace displays the order the instructions were executed. The Trace Data window shows the last instruction executed. The Disassembly and source windows point to the next instruction or source line to be executed.

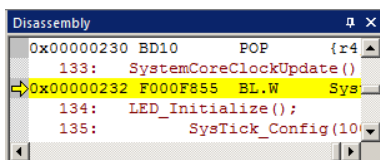
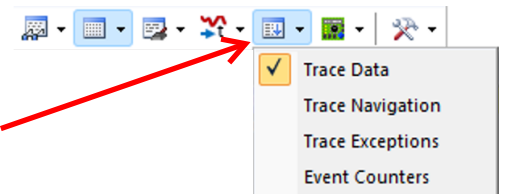
**Trace Buffer:** When the trace data window (buffer) fills up, the oldest frames are over written. How many trace frames are saved and displayed depends on the size of the buffer you selected and the number of program flow changes.

### Open Blinky\_BM to see MTB Operation:

1. In µVision, select Project/Open project.
2. Open the project Blinky from Blinky\_BM\Blinky.uvprojx.
3. (deleted)
4. Blinky\_BM is a simple bare metal (no RTOS) example to make the initial understanding of MTB trace easier.
5. It is pre-configured for MTB with a DBG\_MTB.ini file and OpenSDA in CMSIS-DAP mode.

### Compile, RUN the program and observe MTB Instruction Trace:

1. Compile the source files by clicking on the Rebuild icon.
2. Enter Debug mode. The Flash will be programmed.
3. The program runs to main(). Do not click RUN.
4. Open the Trace Data window by clicking on the small arrow beside this icon:
5. You can also select View/Trace/Trace Data.
6. A window similar to the one below will be visible: Size accordingly. This is a record of the last number of instructions executed by the processor limited by the size of the microcontroller's internal RAM allocated to MTB.
7. Right click on any frame and select Show Functions. The name of the function will be displayed.
8. Note the last instruction executed was a BL.W to main() at frame 199 (in this example).
9. In this example, this BL.W is located at address 0x014E as shown below.
10. You can see the order the instructions were executed.
11. Look in the Disassembly window and you can see the next instruction to be executed is another BL.W. In this case it is the call to the SystemCoreClockUpdate function as shown below left: The yellow arrow is the Program Counter. The BL.W instruction found at the beginning of this function has *not* been executed yet. This is easy to see.






Note: You **will** see slightly different addresses depending on your compiler and other settings.

Trace Data					
Display: Execution					
Index	Address	Opcode	Instruction	Src Code	Function
185	X: 0x0000059A	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
186	X: 0x0000059C	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
187	X: 0x0000059E	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
188	X: 0x000005A0	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
189	X: 0x000005A2	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
190	X: 0x000005A4	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
191	X: 0x000005A6	3D40	SUBS r5,r5,#0x40		
192	X: 0x000005A8	0049	LSLS r1,r1,#1		
193	X: 0x000005AA	468D	MOV sp,r1		
194	X: 0x000005AC	4770	BX lr		
195	X: 0x00000148	4611	MOV r1,r2		
196	X: 0x0000014A	F7FFFFFF	BL.W __rt_lib_init (0x0000013C)		
197	X: 0x0000013C	B51F	PUSH {r0-r4,lr}		
198	X: 0x0000013E	BD1F	POP {r0-r4,pc}		
199	X: 0x0000014E	F000F870	BL.W main (0x00000232)		

**Screenshot #4** at step 11 showing Trace Data window (with last executed instruction (BL.W) highlighted in blue) and Disassembly window (with next instruction to execute (PUSH) highlighted in yellow).


## Tracking between Trace Data and Disassembly and Source Windows:

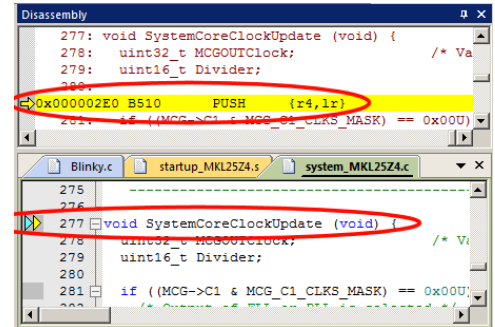
In the example above, the last instruction executed was a BL.W to 0x0232. This put the PC to the start of main().


1. In the Registers window, the Program Counter (PC) (R15) indeed points to 0x0232.
2. Cyan  is a marker from a source window to the Disassembly where it becomes a yellow highlight block.
3. The Yellow arrow  and Yellow triangle  points to the current PC value.
4. Double-click in the Trace Data window on a frame and it is highlighted in the Disassembly and source windows.

**TIP:** The source and Disassembly windows display the code as it was written and the Trace Data window shows the code as it was executed. The Disassembly window displays all assembled instructions. Trace Data shows only executed instructions.

## Single-Stepping:

1. Put the Disassembly window in focus by clicking inside it.
2. The Trace Data window will display the BL.W from the previous page.
3. Click on Step (F11) once. 
4. The BL.W instruction at 0x232 will be executed and added to the bottom of the Trace Data window.
5. Note the C code is listed in the Src code column.  
**Note:** You might see slightly different addresses depending on your compiler and other settings.
6. Examine the Disassembly and Blinky.c windows and see how the trace position is tracked in these windows.
7. The next instruction to be executed is a PUSH instruction at memory 0x02E0 as shown in the Disassembly window. You can confirm the PC is equal to 0x02E0 in the Registers window.
8. Click on Step a number of times to see the effects of the executed instructions. Pay particular attention to branch and stack operations such as POP and PUSH as they are faithfully recorded. Scroll down to see the last frames.
9. You can step through the initialization calls and eventually to where the LEDs are blinked on and off.



**TIP:** If you end up in the Delay() function, click Step Out  or Ctrl-F11 to quickly run to its end and exit. This program spends most of its time in the Delay() function.

Trace Data					
Display: Execution					
Index	Address	Opcode	Instruction	Src Code	Function
192	X: 0x000005A8	0049	LSLS r1,r1,#1		__user_setup_stackheap
193	X: 0x000005AA	468D	MOV sp,r1		__user_setup_stackheap
194	X: 0x000005AC	4770	BX lr		__user_setup_stackheap
195	X: 0x00000148	4611	MOV r1,r2		???
196	X: 0x0000014A	F7FFFFFF	BL.W __rt_lib_init (0x0000013C)		???
197	X: 0x0000013C	B51F	PUSH {r0-r4,lr}		???
198	X: 0x0000013E	BD1F	POP {r0-r4,pc}		???
199	X: 0x0000014E	F000F870	BL.W main (0x00000232)		???
200	X: 0x00000232	F000F855	BL.W SystemCoreClockUpdate (0x00...	SystemCoreClockUpdate();	main

**TIP:** The trace frames can be saved to a file  and the Trace Data window can be cleared. .

## 16) Code Coverage:

Code Coverage states "was this instruction executed". Unexecuted instructions have obviously never been tested. They can cause problems if they unexpectedly execute because of some situation. It is excellent development practice to insure all instructions have been executed and tested. Instruction trace such as MTB, ETB or ETM is an easy way to collect the data.

It is possible to provide Code Coverage from the MTB. It will be limited because of the small trace buffer size.

See [www.keil.com/support/man/docs/uv4/uv4\\_db\\_trace\\_imp\\_codecover.htm%20](http://www.keil.com/support/man/docs/uv4/uv4_db_trace_imp_codecover.htm%20)




## 17) Exploring the MTB Instruction Trace:

**Note:** You might see different addresses as shown here depending on compiler settings. These are using MDK 5.23.

**TIP:** It is possible to debug serious problems by skillfully examining the system stack. But using the trace, as you can see, is much easier and faster. If the stack is destroyed or corrupted by a program crash, the trace will still be available.

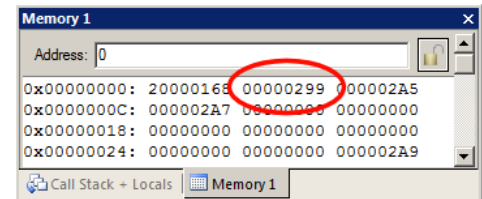
### Examine the Start of the Blinky Program:

1. Scroll to the top of Trace Data. We want to look at the very first instruction executed after RESET.
2. In the window below, Frame Index 0 displays the Reset\_Handler. This is noted in the Function column which shows the functions the instructions belong to. The first occurrence is blocked in orange to help you locate these.
3. The very first instruction executed after RESET is CPSID located at 0x0298. Note the next instruction is at 0x29A and is an LDR instruction.

**TIP:** If you do not see this Reset\_Handler line, it is possible the MTB buffer has been overrun. Restart Blinky. One way is to click RESET  and in the command window entry box ">" type g,main.

Trace Data					
Display: Execution					
Index	Address	Opcode	Instruction	Src Code	Function
0	X: 0x00000298	B672	CPSID I	CPSID I ; Mask interrupts	Reset_Handler
1	X: 0x0000029A	4809	LDR r0,[pc,#36] ; @0x000002C0	LDR R0,=SystemInit	Reset_Handler
2	X: 0x0000029C	4780	BLX r0	BLX R0	Reset_Handler
3	X: 0x000002D8	4942	LDR r1,[pc,#264] ; @0x000003E4	SIM->COPC = (uint32_t)0x00u;	SystemInit
4	X: 0x000002DA	2000	MOVS r0,#0x00		SystemInit
5	X: 0x000002DC	6008	STR r0,[r1,#0x00]		SystemInit
6	X: 0x000002DE	4770	BX lr	}	SystemInit
7	X: 0x0000029E	B662	CPSIE I	CPSIE i ; Unmask interrupts	Reset_Handler
8	X: 0x000002A0	4808	LDR r0,[pc,#32] ; @0x000002C4	LDR R0,=__main	Reset_Handler
9	X: 0x000002A2	4700	BX r0	BX R0	Reset_Handler
10	X: 0x000000C0	F000F802	BLW __scatterload (0x000000C8)		__main
11	X: 0x000000C8	A00C	ADR r0,[pc]+0x34 ; @0x000000FC		__scatterload_rt2
12	X: 0x000000CA	C830	LDM r0!,{r4-r5}		__scatterload_rt2
13	X: 0x000000CC	3808	SUBS r0,r0,#0x08		__scatterload_rt2

4. Open Memory 1 window and enter address 0x0. Right click and select Unsigned Long. This is the beginning of Flash memory. See the Memory window here:
5. In the Memory window, memory 0x00 is the Initial Stack Pointer (0x2000\_0168) which obviously is located in RAM.
6. Location 0x4 is the initial PC and in this case it 0x299. Bit 0 indicates Thumb®2 instruction so subtract 1 and you get 0x298. This is the address of the first instruction in the Trace which is the first instruction executed after RESET. It is CPSID.



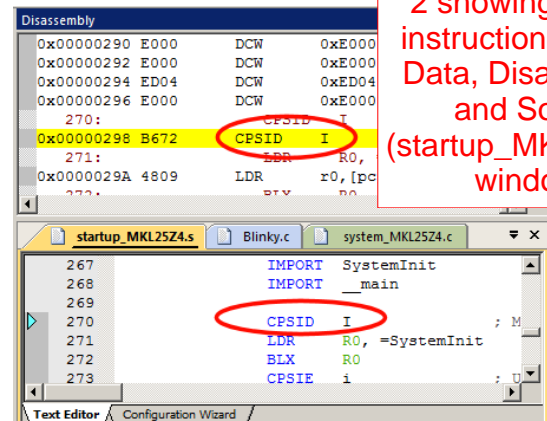
### Show CPSID Instruction in Disassembly and Source window:

It is easy to quickly find the location of an instruction displayed the Trace data window.

1. In the Trace Data window, double-click on the CPSID instruction.
2. This instruction will be displayed in both the Disassembly and startup\_MKL25Z4.s windows as shown here:

**TIP:** If Run to main() is not set in the Target Config window under the Debug tab, no instructions will be executed when Debug mode is entered. The PC will be at the first instruction. You can Step (F11) or RUN from this point and the Trace Data window will update as each instructions are executed.

This is useful to test initialization code after RESET.



## 18) Trace Buffer Configuration and Control:

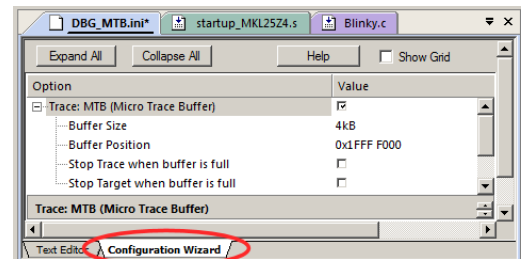
NXP's Cortex-M0+ has two methods of controlling the trace buffer and/or the program. They are:

a) Stop the trace collection when the trace buffer is full and b) Stop the program execution when the trace buffer is full.

### DBG\_MTB.ini file and MTB Configuration:

MTB trace is activated and controlled by the ASCII file DBG\_MTB.ini. It is executed when Debug mode is entered or RESET is clicked. To activate MTB to any project, just add this file as shown here.

1. Exit Debug mode.
2. Select the Target Options icon. Select the Debug tab.
3. Note DBG\_MTB.ini is entered in the Initialization File: box as shown:
4. Select Edit... to open it in µVision along with the other source files.
5. Click on OK to return to the main µVision menu.
6. In the DBG\_MTB.ini window: click the Configuration Wizard tab.
7. An asterisk in the \DBG\_MTB.ini tab indicates this file is not saved. It must be saved by selecting File/Save All or.
8. Click on Expand All. This is where MTB is configured.



### a) Set Buffer Size and Position in RAM:

1. You must allocate the size and location of the internal RAM between your program and the trace buffer.
2. You must not list the RAM used for MTB in the Trace Options windows under the Target tab. This will cause memory access conflicts causing strange problems in your program and the MTB Trace data window.

**Note for a)**

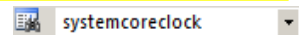
### a) Stop Trace when buffer is full:

1. In DBG\_MTB.ini, Select "Stop Trace when buffer is full". Select File/Save All or.
2. Enter Debug mode. The program runs to the start of main() as before.
3. Click on RUN. After a second or so, click on STOP. Scroll to the end of the Trace Data window.
4. A Trace Gap frame is displayed at Index 2,305. 2,304 instructions are saved in the trace buffer in this case.
5. Scroll to the top of the Trace Data and confirm the first instruction CPSID located at 0x0298 is still recorded.
6. You can use the Search to locate the start of main(). Search for main or SystemCoreClock. It is near Index 200.

The project's DBG\_MTB.ini file allocates 4 kB (0x1FFF\_F000 to 0x1FFF\_FFFF) to the MTB. The Options for Target -> Target tab has been changed to limit the build tools to using the remaining 12 kB of RAM (0x2000\_0000 to 0x2000\_3000). The Linker tab has "Use Memory Layout from Target Dialog" checked.

### b) Stop The Target when buffer is full:

1. Exit Debug mode.
2. In DGB\_MTB.ini, Unselect "Stop Trace when buffer is full". Select "Stop Target when buffer is full".
3. Select File/Save All or. Enter Debug mode. Click on RUN.
4. The program will stop when the trace is full. The trace contains all the executed instructions from the first (0x298) at Index 0 to the last which is SUBS at Index 2,306. You may get slightly different numbers depending on your compiler optimizations. In these two cases, we prevented the trace buffer being overrun. Overrun happens on long runs.



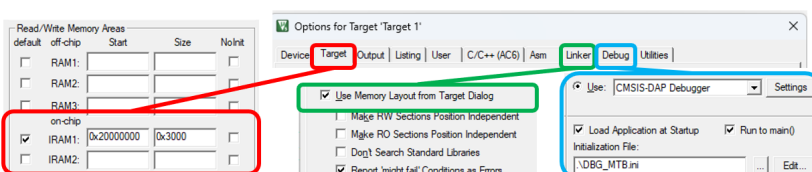
### When Finished:

1. STOP the program. Exit Debug mode.
2. In DGB\_MTB.ini, Unselect "Stop Trace when buffer is full" and unselect "Stop Target when buffer is full".
3. Select File/Save All or.

### What are these features useful for ?

The MTB trace will be overwritten as your program runs. It is possible, even probable, that the trace frames you want to examine will disappear. Using one of these two features can help you keep instructions in your field of interest.

**TIP:** Set a breakpoint on an exception vector (i.e. the Hard Fault if you end up here). If a fault occurs in your program, this will stop the program and also the trace collection. Otherwise the trace buffer will be full with only the Hard Fault instruction.





Copyright © 2017 Arm Ltd. All rights reserved  
www.keil.com

## 19) Trace Search:


With all the trace frames collected it can be difficult to find the frames you want. The Trace Data window includes two useful search tools.

### Pull-Down Menu:

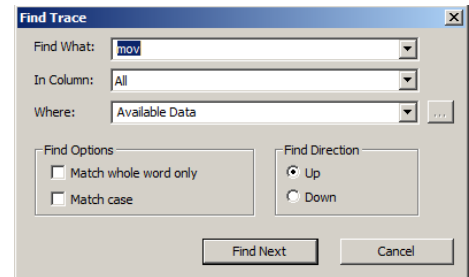
1. Enter Debug mode.  The Trace Data window will display some trace frames.
2. Click in the Trace Search window and enter a term such as push as shown here: 
3. Press Enter and the PUSH instruction or any other occurrence of the word push will be highlighted.
4. Press Enter and each time this will advance to the next occurrence and be highlighted.
5. F3 will advance to the next highest occurrence and Shift-F3 advances to the preceding one.

**TIP:** If “The text as specified below was not found:” is displayed, try clicking on any trace frame to bring them into focus.

### Find Trace:

6. Click on the Find a trace record icon:  This window opens up: You can enter search terms in the usual manner.






**TIP:** You can also select the Find Trace window by clicking on one of the trace frames and press Ctrl-F. Make sure the Find Trace window opens.



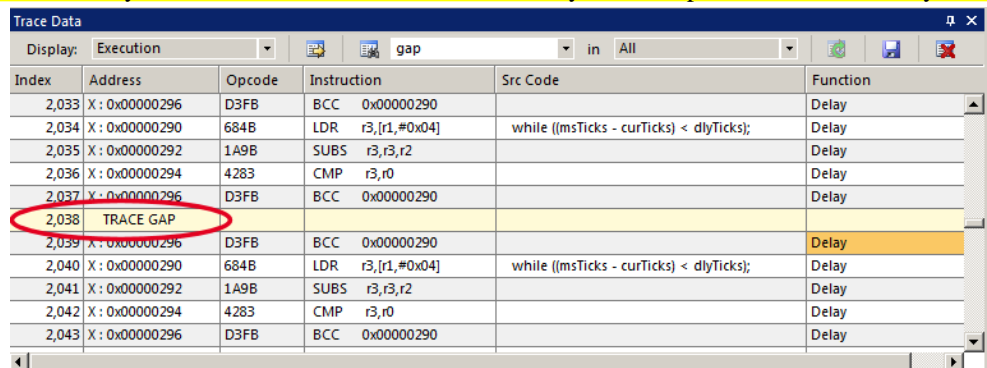
## 20) Trace Data Wrap Around:

The MTB trace buffer is limited in size. In our case, it is set to 4 KB. We have seen so far approximately 2,000 trace frames that can be stored. The actual number depends on the instruction size and other factors such as number of branches. The trace frames coming from CoreSight are highly compressed and µVision reconstructs the trace data as you see it displayed.

As your program runs, old trace frames are over written and discarded by new ones. When you stop the program, any trace frames present are saved/appended to a file. The next run of trace frames is collected and when the program is stopped again, they are appended to the older frames and they are then all displayed in the Trace Data window. We will demonstrate this:

1. Clear the Trace Data window.  Click on RUN . After a second or so, click on STOP .
2. There will be approximately 2,000 trace frames displayed in the Trace Data window. Remember this number.
3. Click on RUN . After a second or so, click on STOP . Compare this Index to your last Index number.
4. Now there will be more trace frames: more than there is processor internal RAM to store them...maybe 4,000.
5. Search the trace buffer for “gap”: without the quotes using a method described above.
6. After the first “end of trace” (in my case it was 2,037) there will be a Trace Gap note as shown here:
7. Repeat a few runs and see that the number of trace frames increases. Search for gap and note this appending method.
8. Each new set of trace frames is appended to the older set of preceding frames and are all displayed in the Trace Data.




**TIP:** You must always remember that a Trace Gap represents an undeterminable number of instructions that are not recorded and hence lost. You are not able to assume any executed instructions are linked over any Trace Gap. This is because they were overwritten during a long execution run.

The Trace Data window is shown with a search for 'gap'. The table lists trace frames with Index, Address, Opcode, Instruction, Src Code, and Function. A 'TRACE GAP' is highlighted at index 2,038.

Index	Address	Opcode	Instruction	Src Code	Function
2,033	X: 0x00000296	D3FB	BCC 0x00000290		Delay
2,034	X: 0x00000290	684B	LDR r3,[r1,#0x04]	while ((msTicks - curTicks) < dlyTicks);	Delay
2,035	X: 0x00000292	1A9B	SUBS r3,r3,r2		Delay
2,036	X: 0x00000294	4283	CMP r3,r0		Delay
2,037	X: 0x00000296	D3FB	BCC 0x00000290		Delay
2,038			TRACE GAP		
2,039	X: 0x00000296	D3FB	BCC 0x00000290		Delay
2,040	X: 0x00000290	684B	LDR r3,[r1,#0x04]	while ((msTicks - curTicks) < dlyTicks);	Delay
2,041	X: 0x00000292	1A9B	SUBS r3,r3,r2		Delay
2,042	X: 0x00000294	4283	CMP r3,r0		Delay
2,043	X: 0x00000296	D3FB	BCC 0x00000290		Delay

## 21) More MTB Exploration: Note:

Note: Your numbers might be different than the ones shown:

1. Exit and enter Debug mode to provide a clean start for this demonstration.
2. In the function LEDRed\_On, set a breakpoint near line 67: `FPTD->PSOR = led_mask[LED_GREEN];`
3. Clear the Trace . Click on RUN .
4. The program will run to the breakpoint.
5. Click on RUN . This to go past the initializations.
6. You will get a Trace Data window similar to the one below:

```
65 void LEDRed_On (void) {
66     FPTB->PSOR = led_mask[LED_BLUE];
67     FPTD->PSOR = led_mask[LED_GREEN];
68     FPTB->PCOR = led_mask[LED_RED];
69 }
```

### Function Column:

1. The executed instructions are displayed. The functions they are located in are noted by the Function column.
2. The beginning of a function run is highlighted in orange.


### Interrupt Subroutine Calls:

1. The SysTick timer creates Interrupt 15. The SysTick\_Handler in Blinky.c near line 48 to 50 increments msTicks.
2. Note the instructions executed a result of this interrupt are displayed at Index 2,013 through 2,017.
3. You can see these instructions reside from 0x196 to 0x19E. Double-click on a line to show it in the source windows.

### Delay Function Interrupted:

1. You can see the Delay function was interrupted by the SysTick interrupt 15 at Index 2,257 and it continued at 2,262.
2. This can be very useful information in locating tricky bugs.

### Continuation of a Function:

1. Located at Index 2,267 is the POP instruction that is the end of the phaseD function. Double-click this line to highlight it in the Disassembly window. Where is the rest of this function in the trace buffer ?
2. If you search for phaseD, you will not find any other occurrence. It was not recorded. What happened is the Delay function, which is called by phaseD near line 127 in Blinky.c, swamps the trace buffer. The other instructions in phaseD were over written. You can confirm this by looking backwards in the trace until the next Trace Gap. If you need to record phaseD, you will have to set a breakpoint at a thoughtful spot in your program.
3. Set a breakpoint on the first instruction in the line Delay():  127 Delay (0x500);
4. Run the program. It will stop at the breakpoint.
5. You will see all executed instructions of phaseD (there are not many) except for the Delay function and the POP at the

4,302	X: 0x00000262	F7FFFDE	BL.W	phaseD (0x00000222)	phaseD();	//all LEDs OFF	main	very
4,303	X: 0x00000222	B510	PUSH	{r4,lr}	void phaseD (void) {		phaseD	end

of phaseD at address 0x230. This is because these are after a long run of the Delay() function.




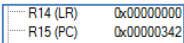


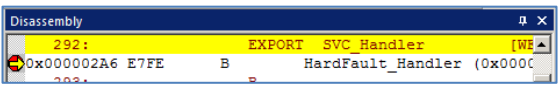






6. Remove all breakpoints. Click on them or you can enter Ctrl-B and select Kill All and then Close.

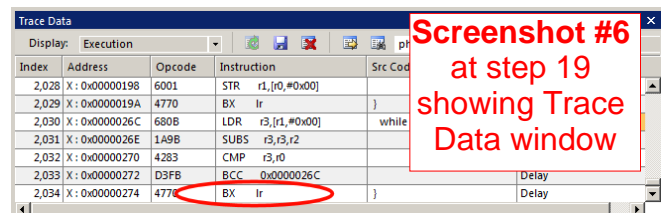
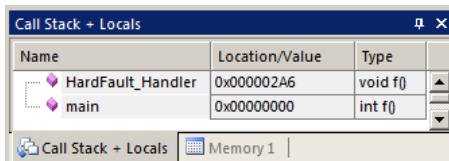
Trace Data						
Display: Execution						
Index	Address	Opcode	Instruction	Src Code	Function	
2,255	X: 0x00000270	4283	CMP r3,r0		Delay	
2,256	X: 0x00000272	D3FB	BCC 0x0000026C		Delay	
2,257	X: 0x00000192	483D	LDR r0,[pc,#244]; @0x00000288	msTicks++; /* increment counter ne...	SysTick_Handler	
2,258	X: 0x00000194	6801	LDR r1,[r0,#0x00]		SysTick_Handler	
2,259	X: 0x00000196	1C49	ADDs r1,r1,#1		SysTick_Handler	
2,260	X: 0x00000198	6001	STR r1,[r0,#0x00]		SysTick_Handler	
2,261	X: 0x0000019A	4770	BX lr		SysTick_Handler	
2,262	X: 0x0000026C	680B	LDR r3,[r1,#0x00]	while ((msTicks - curTicks) < dlyTicks);	Delay	
2,263	X: 0x0000026E	1A9B	SUBS r3,r3,r2		Delay	
2,264	X: 0x00000270	4283	CMP r3,r0		Delay	
2,265	X: 0x00000272	D3FB	BCC 0x0000026C		Delay	
2,266	X: 0x00000274	4770	BX lr		Delay	
2,267	X: 0x00000230	BD10	POP {r4,pc}		phaseD	
2,268	X: 0x00000266	E7F6	B 0x00000256	while(1){	main	
2,269	X: 0x00000256	F7FFFCC	BL.W phaseA (0x000001F2)	phaseA(); //Red LED on	main	
2,270	X: 0x000001F2	B510	PUSH {r4,lr}	void phaseA (void) {	phaseA	
2,271	X: 0x000001F4	F7FFFD2	BL.W LEDRed_On (0x0000019C)	LEDRed_On();	phaseA	
2,272	X: 0x0000019C	4939	LDR r1,[pc,#228]; @0x00000284	FPTD->PSOR = led_mask[LED_BLUE]; /* Blue LE...	LEDRed_On	
2,273	X: 0x0000019E	2002	MOVS r0,#0x02		LEDRed_On	
2,274	X: 0x000001A0	3180	ADDs r1,r1,#0x80		LEDRed_On	
2,275	X: 0x000001A2	6048	STR r0,[r1,#0x04]		LEDRed_On	

## 22) Trace “In the Weeds” Example

Perhaps the most useful use of trace is to show how your program got to an unexpected place. This can happen when normal program flow is disturbed by some error and it goes “into the weeds”. Finding these errors can be extremely challenging. A record of all instructions executed prior to this usually catastrophic error is recorded to the limits of the trace buffer size.

A good way to crash your program is to make register LR = 0. When a BX lr is executed, the program will surely crash. There is a suitable section of code in the Blinky.c Delay function: `while ((msTicks - curTicks) < dlyTicks);`

1. Enter Debug mode.  Click on RUN . Click on STOP .
2. Delay will probably be visible in the Call Stack window.
3. The program will probably be in the Blinky.c line (near 59): `while ((msTicks - curTicks) < dlyTicks);`
4. If not, try the RUN and Stop sequence again.
5. Note in the Disassembly window near memory location 0x284, the last instruction of the Delay function is BX lr.
6. In the Register window, change R14 (LR) to 0x0 as shown here: .
7. Click on RUN . The LEDs will not blink indicating a problem. Click on STOP . The program will be at the Hard Fault vector as shown here: .
8. This happened when the function Delay tried to return.
9. The Trace Buffer will be mostly full of B instructions.
10. We want to stop the program when a hard fault occurs. Otherwise, a HardFault\_handler which is by default a branch to itself, will fill up the trace buffer and overwrite the trace frames that can really help us.
11. The PC in Blinky.c will be on the Hard Fault Handler. It is usually around address 0x02A6 as shown above.
12. Set a breakpoint on this B instruction as shown above.
13. Click RESET.  Click on RUN . Then, click on STOP . The program will be on the while in step 3.
14. Clear the trace buffer.  This is to make things look clearer.
15. Set R14 (LR) in the Registers window to zero as before.
16. Click on RUN .
17. The program will immediately go to the Hard Fault state and the breakpoint will stop execution at the Branch.
18. The Call Stack window (shown below left) will not show much useful information as the Stack was destroyed.
19. The Trace Data now shows the last number of instructions executed plus the BX instruction. Nearly always the last one listed is the one that caused the crash. But not always. Clearly, in this case, you can see the sequence of instructions that caused the fault.
20. Note you can see the execution of the SysTick Handler. This type of event is very hard to detect without trace.
21. Click on Step (F11) a few times and the B at the HardFault\_Handler will be executed and displayed as shown below
22. Remove the breakpoint.
23. Exit Debug mode. 



Index	Address	Opcode	Instruction	Src Code	Function
2,032	X: 0x00000270	4283	CMP r3,r0		Delay
2,033	X: 0x00000272	D3FB	BCC 0x0000026C		Delay
2,034	X: 0x00000274	4770	BX lr		Delay
2,035	X: 0x000002A6	E7FE	B HardFault_Handler (...)	B	HardFault_Handler
2,036	X: 0x000002A6	E7FE	B HardFault_Handler (...)	B	HardFault_Handler

**TIP:** Remember, a CoreSight hardware breakpoint does not execute the instruction it is set to.

**TIP:** MTB can be used to solve many program flow problems that often require much effort to solve.







## 23) DSP SINE example using Arm CMSIS-DSP Libraries:

Arm CMSIS-DSP libraries are offered for Cortex-M processors. DSP libraries are provided in MDK and are selected with the MRTE utility. This example incorporates Keil RTX RTOS

This example creates a sine wave to which noise is added, and then the noise is filtered out leaving the original sine wave.

To obtain this example file, go to [www.keil.com/appnotes/docs/apnt\\_232.asp](http://www.keil.com/appnotes/docs/apnt_232.asp) and copy the DSP folder into C:\00MDK\Boards\NXP\FRDM-KL25Z. A \DSP directory will be created. You should have already done this on page 4.

1. Stop the program and exit Debug mode if necessary.
2. Open the project file sine: C:\00MDK\Boards\NXP\FRDM-KL25Z\DSP\sine.uvprojx.
3. Select CMSIS-DAP in the Target Selector: 
4. Build the files.  There will be no errors or warnings.
5. Enter Debug mode by clicking on the Debug icon.  Select OK if the Evaluation Mode box appears.
6. Click on the RUN icon. 
7. Open Watch 1 by selecting View/Watch/Watch 1 if necessary.
8. Four global variables will be displayed in Watch 1 as shown here:  
If these variables are changing the program is likely working properly.

Watch 1		
Name	Value	Type
sine	0xCF0E	short
noise	0x0800	short
disturbed	0xD336	short
filtered	0xC34F	short
<Enter expression>		

### Other Things You can do:

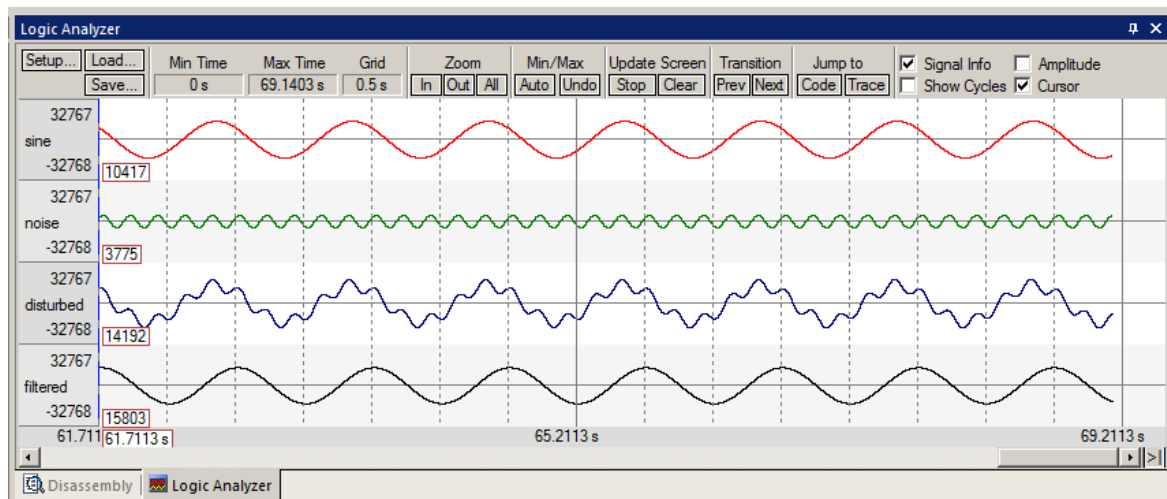
1. **MTB:** Select CMSIS-DAP MTB and you can see the MTB trace working.
2. **System and Threads Viewer:** Select Debug/OS Support/ Threads and System Viewer.

### Serial Wire Viewer (SWV) and ETM Instruction Trace: (only for reference)

The Kinetis Cortex-M4 processors have Serial Wire Viewer (SWV). The four waveforms of the global variables shown above will be displayed in the Logic Analyzer as shown below. The Kinetis KL25Z does not have SWV so this screen is shown for reference only.

If you use a Kinetis Cortex-M4 and with any Keil ULINK or a J-Link, you can use this Logic Analyzer windows plus many other Serial Wire Viewer (SWV) features. Many Kinetis Cortex-M4 processors also have ETM trace.

See [www.keil.com/appnotes/docs/apnt\\_243.asp](http://www.keil.com/appnotes/docs/apnt_243.asp)



## 24) Creating your own MDK 5 project from scratch:

All examples provided by Keil are pre-configured. All you have to do is compile them. You can use them as a starting point for your own projects. However, we will start this example project from the beginning to illustrate how easy this process is. Once you have your new project configured; you can build, load and run a bare Blinky example. It will have an empty main() function so it does not do much. However, the processor startup sequences are present and you can easily add your own source code and/or files. You can use this process to create any new project, including one using an RTOS.

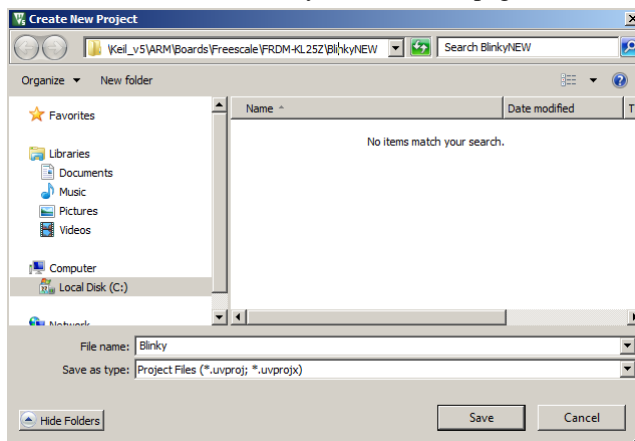
**MCUXpresso SDK Builder:** A file containing files and examples is available for this board on <https://mcuxpresso.nxp.com/>

### Install the Kinetis Software Pack for your processor:

1. Start  $\mu$ Vision and leave in Edit mode. Do not be in Debug mode.
2. **Pack Installer:** The Pack for the KL25Z processor must be installed. This has already been done on page 4.
3. You do not need to copy any examples over.

### Create a new Directory and a New Project:


1. In the main  $\mu$ Vision menu, click on Project/New  $\mu$ Vision Project...
2. In the window that opens, shown below, go to the folder C:\00MDK\Boards\NXP\FRDM-KL25Z\
3. Right click in this window and select New and create a new folder. I called it BlinkyNEW.
4. Highlight BlinkyNew and select Open.
5. In the File name: box, enter Blinky. Click on Save.
6. This creates the project Blinky.uvproj. (is MDK 4)
7. As soon as you click on Save, the next window opens:

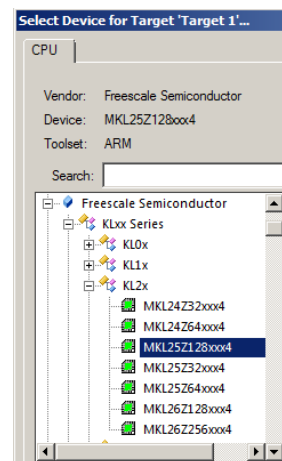


### Select the Device you are using:

1. Expand NXP then KLxx Series, then KL2x and then select MKL25Z128xxx4 as shown: Make sure you go to the lowest level processor or this will not work.
2. Click OK and the Manage Run Time window shown below bottom right opens.

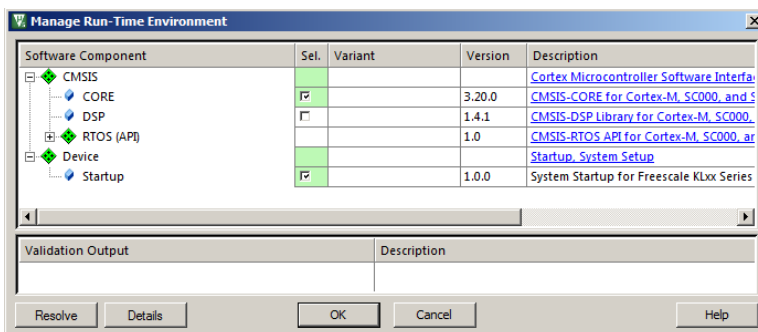
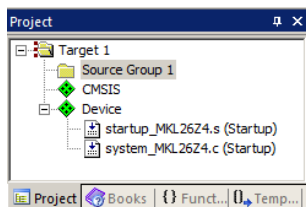
### Select the CMSIS (System and Startup) files you want:

1. Expand all the items and select CORE and Startup as shown below. They will be highlighted in Green indicating there are no other files needed. Click OK.
2. Click on File/Save All or select the Save All icon: 
3. The project Blinky.uvproj. will now be changed to Blinky.uvprojx.
4. You now have a new project list as shown on the bottom left below: The appropriate CMSIS files you selected have been automatically entered and configured.
5. Note the Target Selector says Target 1. Highlight Target 1 in the Project window.
6. Click once on it and change its name to CMSIS-DAP and press Enter. The Target selector name will also change.

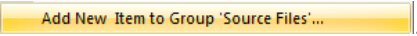



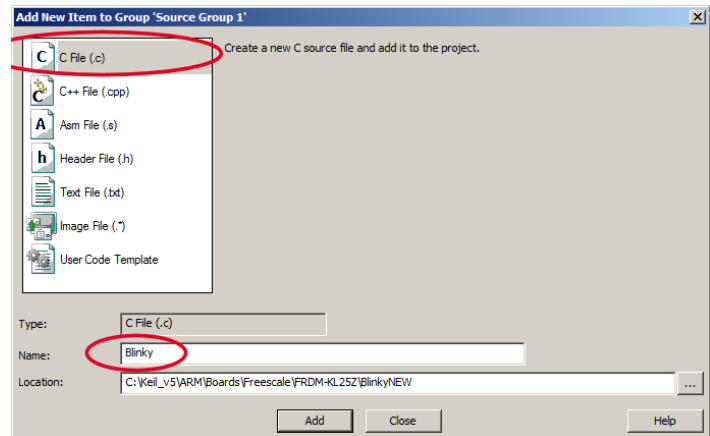
### What has happened to this point:

You have created a blank  $\mu$ Vision project using MDK 5 Software Packs. All you need to do now is add your own source files.





## Create a blank C Source File:

1. Right click on Source Group 1 in the Project window and select .
2. This window opens up:
3. Highlight the upper left icon: C file (.c):
4. In the Name: field, enter Blinky.
5. Click on Add to close this window.
6. Click on File/Save All or .
7. Expand Source Group 1 in the Project window and Blinky.c will now display.
8. It will also open in the Source window.



## Add Some Code to Blinky.c:


1. In the blank Blinky.c, right click and select Insert '# include file'. Select # include <MKL25Z4.h>
2. Add the rest of the code as shown in the box below:
3. Click on File/Save All or .
4. Build the files.  There will be no errors or warnings if all was entered correctly.

Step 3.5: Open "Options for Target" (Alt-F7). Target tab should be selected already. In Code Generation section, for ARM Compiler select "Use default compiler"


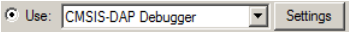
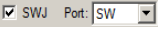
```
#include <MKL25Z4.h>
unsigned int counter = 0;

/*-----
  MAIN function
  -----*/
int main (void) {
    SystemCoreClockUpdate();
    while(1) {
        counter++;
        if (counter > 0x0F) counter = 0;
    }
}
```




**TIP:** You can also add existing source files:



## Configure the Target CMSIS-DAP Debug Adapter:

1. Select the Target Options icon . Select the **Target** tab.
2. Select Use MicroLIB to optimize for smaller code size.
3. Click on the **Debug** tab. Select CMSIS-DAP Debugger in the Use: box: .
4. Select Settings: icon beside Use: CMSIS-DAP.
5. Select SWJ and SW as shown here:  A JTAG selection here will return an RDDI error. If your KL25Z is connected to your PC, you should now see a valid IDCODE and Device Name in the SW Device box.




## Confirm Flash Programming Algorithm is Configured:

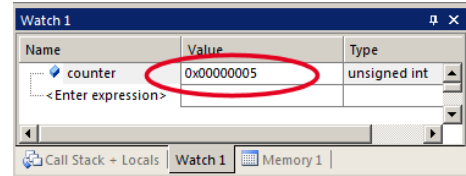
1. Click on OK *once* to go back to the Target Configuration window.
2. Click on the **Utilities** tab. Select Settings and confirm the correct Flash algorithm is present: Shown is the correct one for the Freedom KL25Z board: 
3. Click on OK twice to return to the main menu.
4. Click on File/Save All or .
5. Build the files.  There will be no errors or warnings if all was entered correctly. If there are, please fix them !

Programming Algorithm			
Description	Device Size	Device Type	Address Range
MKXX 48Mhz 128kB Prog Flash	128k	On-chip Flash	00000000H - 0001FFFFH

**The Next Step ?** Let us run your program and see what happens ! Please turn the page....





## Running Your Program:

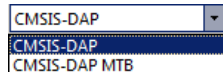
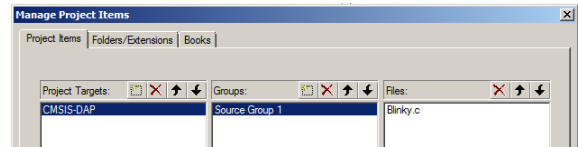
1. Enter Debug mode by clicking on the Debug icon . The Flash will be programmed.
2. Click on the RUN icon . **Note:** you stop the program with the STOP icon .
3. Right click on counter in Blinky.c and select Add counter to ... and select Watch 1.
4. counter will be updating as shown here:
5. You can also set a breakpoint in Blinky.c and the program should stop at this point if it is running properly. If you do this, remove the breakpoint.
6. You should now be able to add your own source code to create a meaningful project. Samples are provided on the NXP SDK Builder found on <https://mcuxpresso.nxp.com/>




**TIP:** The Watch 1 is updated periodically, not when a variable value changes. Since Blinky is running very fast without any time delays inserted, the values in Watch 1 will appear to jump and skip sequential values you know must exist.

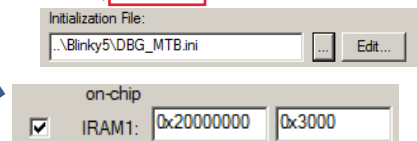
## Configuring MTB Instruction Trace:

1. Stop the program.  Remove any breakpoints (Ctrl-B, Kill All, Close). Exit Debug mode. 
2. Select Project/Manage/Project Items... or select: ... and this window opens up:
3. Select the Insert icon  or press Insert key on the PC.
4. Enter CMSIS-DAP MTB and press Enter. Click OK to exit.
5. Open the Target Selector and you will now find CMSIS-DAP MTB visible:
6. Select CMSIS-DAP MTB.






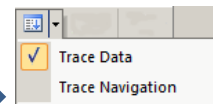
**What this means:** We now have two targets: CMSIS-DAP and CMSIS-DAP MTB. Each one points to its own Target Options configuration. At this point they are the same. We will modify CMSIS-DAP MTB to activate the MTB trace.

7. Select the Target Options icon . Select the **Debug** tab. Note the Initialization File: box is empty.
8. Using the Browse icon, go to the directory C:\00MDK\Boards\NXP\FRDM-KL25Z\Blinky and insert DBG\_MTB.ini as shown here:
9. Select the Target tab. In IRAM1, change RAM to these values:
10. Click OK to exit this window. MTB is now activated with the defaults.



**TIP:** In real life you are probably better to copy this .ini file into your own project.

11. Click on File/Save All or  to save all your work so far.
12. Enter Debug mode.  The program will Run to main() automatically.
13. Open the Trace Data window: View/Trace/Trace Data or the small arrow beside the icon: 
14. The Trace Data window will be full of trace frames. Right click inside it and select Show Functions.





**What this means:** A collection of Target Options is saved in the Target Setting CMSIS-DAP. Another one, with the MTB activated, is saved with CMSIS-DAP MTB. You can modify any other Target Options window and it will be saved. You can select them at will. You usually need a rebuild after changing targets.

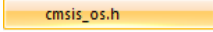
## 25) Adding **RTX** to your Project:

The MDK Software Packs makes it easy to configure an RTX project. This example uses RTX 4.

Configuring RTX is easy. These steps use the same configuration as in the preceding Blinky example.

1. Using the same example from the preceding pages, Stop the program  and Exit Debug mode. 

2. Select CMSIS-DAP: 

3. At the top of Blinky.c, right click and select Insert '# include file'. Select # include <cmsis\_os.h> 

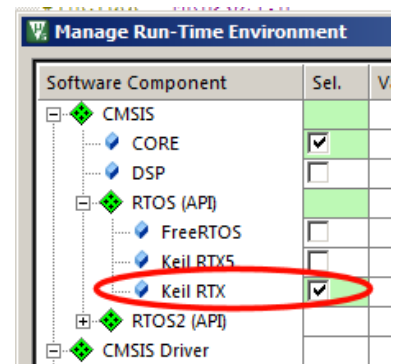
4. Open the Manage Run-Time Environment window: 

5. Expand all the elements as shown here: 


6. Select Keil RTX as shown and click OK.

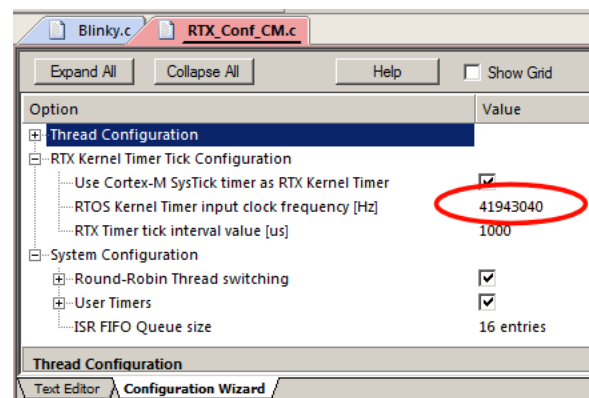
7. Appropriate RTX files will be added to your project. See the Project window.

8. Click on File/Save All or 






### Configure RTX:

1. In the Project window, expand the CMSIS group.
2. Double click on RTX\_Conf\_CM.c to open it.
3. Select the Configuration Wizard tab: Select Expand All.
4. The window is displayed here: 
5. Set Timer clock value: to 41943040 as shown: (41.9 MHz)
6. Use defaults for the other settings.

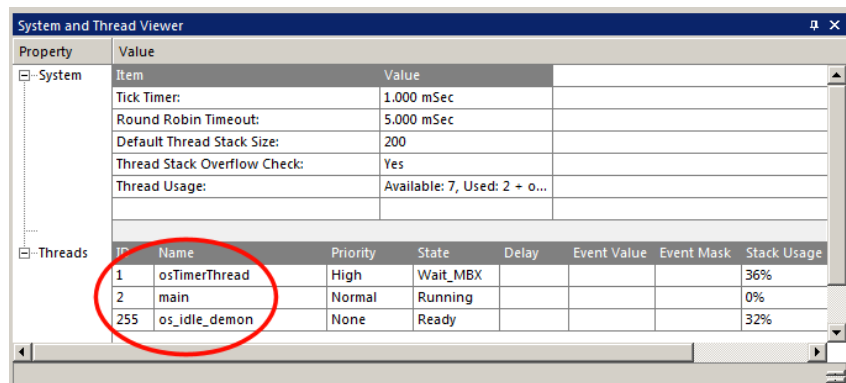


### Build and Run Your RTX Program:

1. Build the files.  No errors or warning.
2. Enter Debug mode:  Click on the RUN icon. 
3. Select Debug/OS Support/System and Thread Viewer. The window below opens up.
4. You can see two threads plus the idle demon. As you add more threads to create a real RTX program, these will automatically be added to this window.

### What you have to do now:



1. You must add the RTX framework into your code and create your threads to make this into a real RTX project configured to your needs.
2. See the DSP and RTX\_Blinky examples to use as templates and hints.
3. **Getting Started MDK 5:** Obtain this useful book here: [www.keil.com/mdk5/](http://www.keil.com/mdk5/). It has useful information on implementing RTX.





## 26) Adding a Thread to your RTX\_Blinky:

We will create and activate a thread. We will add another global variable counter2 to give it something to do.

1. Stop the program  and Exit Debug mode. 
2. In Blinky.c, add this line near line 4 before the main() function:

```
4 unsigned int counter2 = 0;
```

### Create the Thread job1:

3. Add this code to be the thread **job1** before main():

**TIP:** osDelay(1000) delays the program by 1000 clock ticks to slow it down so we can see the values of counter and counter2 increment by 1.

```
6 void job1 (void const *argument) {
7     for (;;) {
8         counter2++;
9         if (counter2 > 0x0F) counter2=0;
10        osDelay(1000);
11    }
12 }
```

### Add osDelay to main():

4. Add this line just after the if statement near line 22:



```
22 osDelay(1000);
```

### Define and Create the Thread:



5. Define job1 near line 14 just before main():
6. Create the thread job1 near line 20 just before the while(1) loop in main():

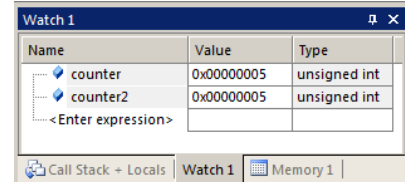
```
14 osThreadDef(job1, osPriorityNormal, 1, 0);
```

```
20 osThreadCreate(osThread(job1), NULL);
```

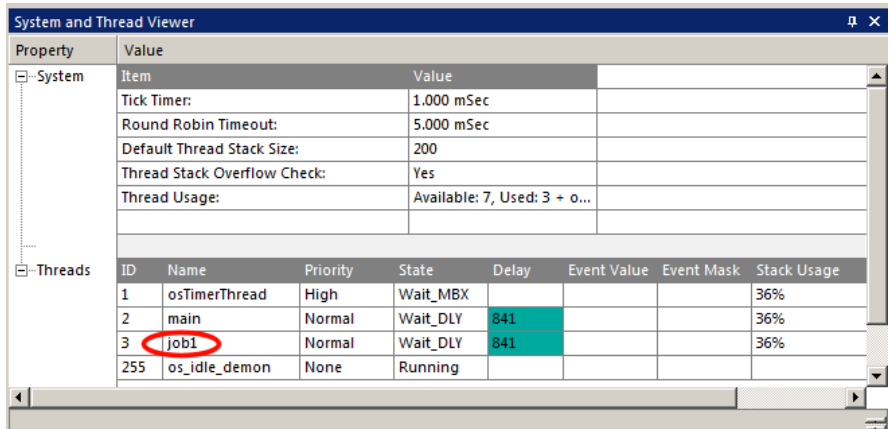
7. Click on File/Save All or 
8. Build the files.  There will be no errors or warnings. If there are, please fix them before continuing.

### Run the Program and configure Watch 1 and see job1 running:

1. Enter Debug mode:  Click on the RUN icon. 
2. Right click on counter2 in Blinky.c and select Add counter2 to ... and select Watch 1.
3. Both counter and counter2 will increment but slower than before: The two osDelay(1000) function calls each slow the program down by 1000 msec. This makes it easier to watch these two global variables increment. OsDelay() is a function provided by RTX.
4. Open the System and Thread Viewer by selecting Debug/OS Support.
5. Note that job1 has now been added as a thread as shown below:
6. Note os\_idle\_demon is always labelled as Running. This is because the program spends most of its time here.
7. Set a breakpoint in job1 and the program will stop there and job1 is displayed as "Running" in the Viewer.
8. Set another breakpoint in the while(1) loop in main() and each time you click RUN, the program will change threads.
9. There are many attributes of RTX you can add. See the RTX documentation and the MDK 5 Getting Started Guide.
10. You can easily add more threads as needed.
11. RTX has many more features that you can utilize. RTX is a full feature vary capable RTOS.



Name	Value	Type
counter	0x00000005	unsigned int
counter2	0x00000005	unsigned int
<Enter expression>		



Property		Value
Item		Value
Tick Timer:		1.000 mSec
Round Robin Timeout:		5.000 mSec
Default Thread Stack Size:		200
Thread Stack Overflow Check:		Yes
Thread Usage:		Available: 7, Used: 3 + o...




  

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				36%
2	main	Normal	Wait_DLY	841			36%
3	job1	Normal	Wait_DLY	841			36%
255	os_idle_demon	None	Running				





## 27) Event Recorder:

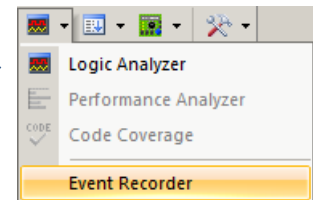
Event Recorder is a new  $\mu$ Vision feature. Code annotations can be inserted into your code to send out messages to  $\mu$ Vision and be displayed as shown below. Keil Middleware and RTX5 have these annotations already inserted. You can add Event Recorder annotations to your own source code. SWV is not used. CoreSight DAP Reads and Writes are used.

Documentation for Event Recorder is found here: [www.keil.com/pack/doc/compiler/EventRecorder/html/](http://www.keil.com/pack/doc/compiler/EventRecorder/html/)



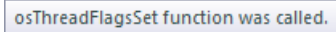


Event Recorder				
Enable Recorder: <input checked="" type="checkbox"/>    Mark: <input type="text"/> All Operations Stopped				
Event	Time (sec)	Component	Event Property	Value
0	0.00000000		Init Event	Restart Count=0x00000001
1	0.00000010	RTX Kernel	KernelInitialize	
2	0.00000020	RTX Kernel	KernelInitializeCompleted	
3	0.00000030	RTX Thread	ThreadNew	func=job1, argument=0x00000000, attr=0x00000000
4	0.00000040	RTX Memory	MemoryAlloc	mem=0x20000000, size=80, type=1, block=0x20000010
5	0.00000050	RTX Memory	MemoryAlloc	mem=0x20000000, size=208, type=0, block=0x20000060
6	0.00000060	RTX Thread	ThreadCreated	thread_id=0x20000010
7	0.00000070	RTX Thread	ThreadNew	func=job2, argument=0x00000000, attr=0x00000000
8	0.00000080	RTX Memory	MemoryAlloc	mem=0x20000000, size=80, type=1, block=0x20000130
9	0.00000090	RTX Memory	MemoryAlloc	mem=0x20000000, size=208, type=0, block=0x20000180
10	0.00000100	RTX Thread	ThreadCreated	thread_id=0x20000130
11	0.00000110	RTX Kernel	KernelStart	
12	0.00000120	RTX Thread	ThreadCreated	thread_id=0x200012B4
13	0.00000130	RTX Thread	ThreadSwitch	thread_id=0x20000010
14	0.00000140	RTX Kernel	KernelStarted	




### Demonstrating Event Recorder with RTX5\_Blinky:

1. Open Blinky.uvprojx in C:\00MDK\Boards\NXP\FRDM-KL25Z\RTX5\_Blinky\
2. Click on Rebuild. . Enter Debug mode.  Click on RUN .
3. Open Event Recorder by selecting View/Analysis/Event Recorder or 
4. Since Event Recorder is activated in RTX v5, the window above will display.
5. Various RTX events will display as they happen. You can do this for your own code.



### Event Recorder Features:

1. Stop and start Event Recorder while the program is running: Enable Recorder: ☒
2. Clear the window when the program is stopped: 
3. Stop the program. 
4. In the Mark: box, enter ThreadSwitch and these frames will be highlighted as shown here: This is useful to find events that do not occur frequently.
5. If you click on a frame in the Event Property column, you will be taken to Help for this event.
6. Hover your mouse over an event in the Value column and a hint will display such as this one:  

7. Stop the program.  Close any Event Recorder and Threads and Event (RTX RTOS) windows.
8. Exit Debug mode.  **This is the end of the exercises.**

Event Recorder				
Enable Recorder: <input checked="" type="checkbox"/>    Mark: ThreadSwitch All Operations				
Event	Time (sec)	Component	Event Property	Value
51	0.00000510	RTX Thread	ThreadDelay	ticks=200
52	0.00000520	RTX Thread	ThreadBlocked	thread_id=0x20000130, timeout=200
53	0.00000530	RTX Thread	ThreadSwitch	thread_id=0x200012B4
54	0.00000540	RTX Thread	ThreadDelayCompleted	
55	0.00000550	RTX Thread	ThreadUnblocked	thread_id=0x20000010, ret_val=osOK
56	0.00000560	RTX Thread	ThreadDelayCompleted	
57	0.00000570	RTX Thread	ThreadUnblocked	thread_id=0x20000130, ret_val=osOK
58	0.00000580	RTX Thread	ThreadSwitch	thread_id=0x20000010
59	0.00000590	RTX Thread	ThreadDelay	ticks=200

## 28) Some Interesting Bits & Pieces:

### Processor Clock Speed:

The clock speed is determined in the file `system_MKL25Z4.c`. This is where the processor PLL and other clock attributes are configured.

### SystemCoreClock:

The file `system_MKL25Z4.c` contains a global variable `SystemCoreClock` near line 103. You can view this in the Watch window to determine the processor clock frequency. Its value is determined from `DEFAULT_SYSTEM_CLOCK` which is defined in `system_MKL25Z4.h` like this: `#define DEFAULT_SYSTEM_CLOCK`

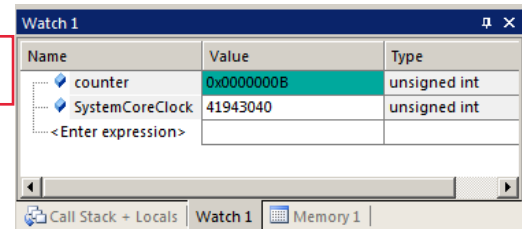
Which one is used depends on **what CLOCK\_SETUP is.** By setting `CLOCK_SETUP` to 0, 1 or 2 changes the clock frequency to 41.94, 48 or 8 MHz respectively. This is easily seen in lines 81 through 95. This is not actually measuring the CPU speed, rather just indicating which of the choices is selected. This means that a bug in the CMSIS System file could give erroneous results.

### Configure SystemCoreClockUpdate:

1. Near the top of `main()` in `Blinky.c`, add this line: `SystemCoreClockUpdate();`
2. This will initialize the clocks in `system_MKL25Z4.c`.

### Display SystemCoreClock:

1. With  $\mu$ Vision in Debug mode and running, enter `SystemCoreClock` into Watch 1.
2. Right click on the Value field for `SystemCoreClock` and unselect Display Hexadecimal to get a base 10 value.
3. Note the clock is about 41.9 MHz. We thought it was 48 MHz !
4. There are two ways to define the value of `CLOCK_SETUP`: in `system_MKL25Z4.h` or in the compiler command line options.



5. Stop the program. Enter Debug mode.
6. Build the files. Program the Flash.
7. Enter Debug mode. Click on the RUN icon.
8. `SystemCoreClock` will now display 48 MHz.

Method 1: Add `#define CLOCK_SETUP 1` to `system_MKL25Z4.h` before the `#ifdef CLOCK_SETUP`.  
Method 2: In Project->Options for Target -> C/C++/Preprocessor Symbols "Define" box, enter `CLOCK_SETUP=1` and click OK.

### Single-Stepping:

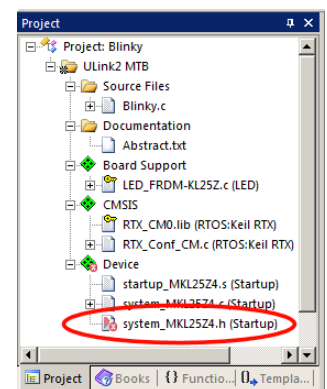
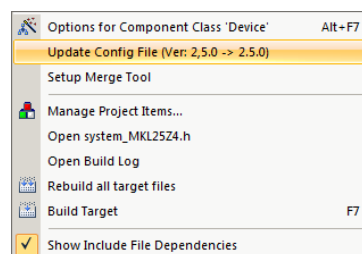
1. With `Blinky.c` or any C source file in focus (`Blinky.c` tab is underlined), the program counter jumps a C line at a time. The yellow arrow indicates the next C line to be executed.
2. When the Disassembly window is in focus indicated by the top bar turning dark gray or a darker colour. Clicking Step Into now jumps the program counter one assembly instruction at a time.
- 3.

### Updating CMSIS Files:

Sometimes a file will be older than the one provided by the latest Software Pack for your processor. In this example, in the project window, you can see the small red marks indicating this file is not the latest.

Both the file and its header (Device in this case) are marked.

1. Right click on the file name and select Update Config File as shown below:
2. The latest version will be inserted in your project and the red marks will be gone.
3. Click on File/Save All or



## 29) Kinetis KL25 Cortex-M0+ Trace Summary:

### Watch and Memory windows can see:

- Global variables.
- Static variables.
- Structures.
- Peripheral registers – just read or write to them.
- Can't see local variables. (just make them global or static).
- Can't see DMA transfers – DMA bypasses CPU and CoreSight and CPU by definition.

### Instruction Trace (MTB) is good for:

- Trace adds significant power to debugging efforts. Tells where the program has been.
- A recorded history of the program execution *in the order it happened*.
- Trace can often find nasty problems very quickly.
- Weeks or months can be replaced by minutes.
- Especially where the bug occurs a long time before the consequences are seen.
- Or where the state of the system disappears with a change in scope(s).

### These are the types of problems that can be found with a quality trace:


- Pointer problems.
- Illegal instructions and data aborts (such as misaligned writes).
- Code overwrites – writes to Flash, unexpected writes to peripheral registers (SFRs), a corrupted stack. *How did I get here ?*
- Out of bounds data. Uninitialized variables and arrays.
- Stack overflows. What causes the stack to grow bigger than it should ?
- Runaway programs: your program has gone off into the weeds and you need to know what instruction caused this. Is very tough to find these problems without a trace. ETM trace is best for this.
- Communication protocol and timing issues. System timing problems.

### 30) CoreSight Definitions: It is useful to have a basic understanding of these terms:

**Note:** The KL25Z Cortex-M0+ options are highlighted in red below: Kinetis Cortex-M4 processors have all features except MTB. To use SWV, any Keil ULINK or Segger J-link debug adapter is needed and to use ETM trace, a ULINKpro is needed.

1. **JTAG:** Provides access to the CoreSight debugging module located on the Cortex processor. It uses 4 to 5 pins.
2. **SWD:** Serial Wire Debug is a two pin alternative to JTAG and has about the same capabilities except Boundary Scan is not possible. SWD is referenced as SW in the  $\mu$ Vision Cortex-M Target Driver Setup. The SWJ box must be selected in ULINK2/ME or ULINKpro. Serial Wire Viewer (SWV) must use SWD because the JTAG signal TDO shares the same pin as SWO. The SWV data normally comes out the SWO pin or Trace Port.
3. JTAG and SWD are functionally equivalent. The signals and protocols are not directly compatible.
4. **DAP:** Debug Access Port. This is a component of the Arm CoreSight debugging module that is accessed via the JTAG or SWD port. One of the features of the DAP are the memory read and write accesses which provide on-the-fly memory accesses without the need for processor core intervention.  $\mu$ Vision uses the DAP to update Memory, Watch, Peripheral and RTOS kernel awareness windows while the processor is running. You can also modify variable values on the fly. No CPU cycles are used, the program can be running and no code stubs are needed. You do not need to configure or activate DAP.  $\mu$ Vision configures DAP when you select a function that uses it. Do not confuse this with CMSIS\_DAP which is an Arm on-board debug adapter standard.
5. **SWV:** Serial Wire Viewer: A trace capability providing display of reads, writes, exceptions, PC Samples and printf.
6. **SWO:** Serial Wire Output: SWV frames usually come out this one pin output. It shares the JTAG signal TDO.
7. **Trace Port:** A 4 bit port that ULINKpro uses to collect ETM frames and optionally SWV (rather than SWO pin).
8. **ITM:** Instrumentation Trace Macrocell: As used by  $\mu$ Vision, ITM is thirty-two 32 bit memory addresses (Port 0 through 31) that when written to, will be output on either the SWO or Trace Port. This is useful for printf type operations.  $\mu$ Vision uses Port 0 for printf and Port 31 for the RTOS Event Viewer. The data can be saved to a file.
9. **ETM:** Embedded Trace Macrocell: Displays all the executed instructions. The ULINKpro provides ETM. ETM requires a special 20 pin CoreSight connector. ETM also provides Code Coverage and Performance Analysis. ETM is output on the Trace Port or accessible in the ETB (ETB has no Code Coverage or Performance Analysis).
10. **ETB:** Embedded Trace Buffer: A small amount of internal RAM used as an ETM trace buffer. This trace does not need a specialized debug adapter such as a ULINKpro. ETB runs as fast as the processor and is especially useful for very fast Cortex-A processors. Not all processors have ETB. See your specific datasheet.
11. **MTB:** Micro Trace Buffer. A portion of the device internal user RAM is used for an instruction trace buffer. Only on Cortex-M0+ processors. Cortex-M3/M4 and Cortex-M7 processors provide ETM trace instead.
12. **Hardware Breakpoints:** The Cortex-M0+ has 2 breakpoints. The Cortex-M3, M4 and M7 usually have 6. These can be set/unset on-the-fly without stopping the processor. They are no skid: they do not execute the instruction they are set on when a match occurs. The CPU is halted before the instruction is executed.
13. **Watchpoints:** Both the Cortex-M0, M0+, Cortex-M3, Cortex-M4 and Cortex-M7 can have 2 Watchpoints. These are conditional breakpoints. They stop the program when a specified value is read and/or written to a specified address or variable. There also referred to as Access Breaks in Keil documentation.

#### Read-Only Source Files:

Some source files in the Project window will have a yellow key on them:  This means they are read-only. This is to help unintentional changes to these files. This can cause difficult to solve problems. These files normally need no modification.  $\mu$ Vision icon meanings are found here: [www.keil.com/support/man/docs/uv4/uv4\\_ca\\_filegrp\\_att.htm](http://www.keil.com/support/man/docs/uv4/uv4_ca_filegrp_att.htm)

If you need to modify one, you can use Windows Explorer to modify its permission.

1. In the Projects window, double click on the file to open it in the Sources window.
2. Right click on its source tab and select Open Containing folder.
3. Explorer will open with the file selected.
4. Right click on the file and select Properties.
5. Unselect Read-only and click OK. You are now able to change the file in the  $\mu$ Vision editor.
6. It is a good idea to make the file read-only when you are finished modifications.



### 31) Document Resources:

See [www.keil.com/NXP](http://www.keil.com/NXP)

#### Books:

1. **NEW!** Getting Started MDK 5: Obtain this free book here: [www.keil.com/mdk5/](http://www.keil.com/mdk5/).
2. There is a good selection of books available on ARM: [www.arm.com/support/resources/arm-books/index.php](http://www.arm.com/support/resources/arm-books/index.php)
3.  $\mu$ Vision contains a window titled Books. Many documents including data sheets are located there.
4. Keil manuals and documents: [www.keil.com/arm/man/arm.htm](http://www.keil.com/arm/man/arm.htm) **Videos:** [www.keil.com/videos](http://www.keil.com/videos)
5. A list of resources is located at: [www.arm.com/products/processors/cortex-m/index.php](http://www.arm.com/products/processors/cortex-m/index.php)  
Click on the Resources tab. Or search for “Cortex-M3” on [www.arm.com](http://www.arm.com) and click on the Resources tab.
6. The Definitive Guide to the Arm Cortex-M0/M0+ by Joseph Yiu. Search the web.
7. The Definitive Guide to the Arm Cortex-M3/M4 by Joseph Yiu. Search the web.
8. Embedded Systems: Introduction to Arm Cortex-M Microcontrollers (3 volumes) by Jonathan Valvano

#### Application Notes: [www.keil.com/appnotes](http://www.keil.com/appnotes)

9. Using Cortex-M3 and Cortex-M4 Fault Exceptions [www.keil.com/appnotes/files/apnt209.pdf](http://www.keil.com/appnotes/files/apnt209.pdf)
10. Segger emWin GUIBuilder with  $\mu$ Vision™ [www.keil.com/appnotes/files/apnt\\_234.pdf](http://www.keil.com/appnotes/files/apnt_234.pdf)
11. Porting mbed Project to Keil MDK™ [www.keil.com/appnotes/docs/apnt\\_207.asp](http://www.keil.com/appnotes/docs/apnt_207.asp)
12. MDK-ARM™ Compiler Optimizations [www.keil.com/appnotes/docs/apnt\\_202.asp](http://www.keil.com/appnotes/docs/apnt_202.asp)
13. Using  $\mu$ Vision with CodeSourcery GNU [www.keil.com/appnotes/docs/apnt\\_199.asp](http://www.keil.com/appnotes/docs/apnt_199.asp)
14. RTX CMSIS-RTOS Download [www.keil.com/demo/eval/rtx.htm](http://www.keil.com/demo/eval/rtx.htm)
15. Barrier Instructions <http://infocenter.arm.com/help/topic/com.arm.doc.dai0321a/index.html>
16. Lazy Stacking on the Cortex-M4: [www.arm.com](http://www.arm.com) and search for DAI0298A
17. Cortex Debug Connectors: [www.arm.com](http://www.arm.com) and search for cortex\_debug\_connectors.pdf
18. Sending ITM printf to external Windows applications: [http://www.keil.com/appnotes/docs/apnt\\_240.asp](http://www.keil.com/appnotes/docs/apnt_240.asp)
19. FlexMemory configuration using MDK [www.keil.com/appnotes/files/apnt220.pdf](http://www.keil.com/appnotes/files/apnt220.pdf)
20. **NEW!** Migrating Cortex-M3/M4 to Cortex-M7 processors: [www.keil.com/appnotes/docs/apnt\\_270.asp](http://www.keil.com/appnotes/docs/apnt_270.asp)
21. **NEW!** ARMv8-M Architecture Technical Overview [www.keil.com/appnotes/files/apnt\\_291.pdf](http://www.keil.com/appnotes/files/apnt_291.pdf)
22. **NEW!** Determining Cortex-M CPU Frequency using SWV [www.keil.com/appnotes/docs/apnt\\_297.asp](http://www.keil.com/appnotes/docs/apnt_297.asp)

#### Keil Tutorials for NXP Boards:

[www.keil.com/NXP](http://www.keil.com/NXP)

1. KL25Z Freedom [www.keil.com/appnotes/docs/apnt\\_232.asp](http://www.keil.com/appnotes/docs/apnt_232.asp)
2. K20D50M Freedom Board [www.keil.com/appnotes/docs/apnt\\_243.asp](http://www.keil.com/appnotes/docs/apnt_243.asp)
3. Kinetis K60N512 Tower [www.keil.com/appnotes/docs/apnt\\_239.asp](http://www.keil.com/appnotes/docs/apnt_239.asp)
4. Kinetis K60D100M Tower [www.keil.com/appnotes/docs/apnt\\_249.asp](http://www.keil.com/appnotes/docs/apnt_249.asp)
5. Kinetis FRDM-K64F Freedom [www.keil.com/appnotes/docs/apnt\\_287.asp](http://www.keil.com/appnotes/docs/apnt_287.asp)
6. Kinetis K64F120M Tower [www.keil.com/appnotes/docs/apnt\\_288.asp](http://www.keil.com/appnotes/docs/apnt_288.asp)
7. NXP S32K Cortex-M4: [www.keil.com/appnotes/docs/apnt\\_299.asp](http://www.keil.com/appnotes/docs/apnt_299.asp)

**Forums:** [www.keil.com/forum](http://www.keil.com/forum) <http://community.arm.com/groups/tools/content> <https://developer.arm.com/>

**Arm University program:** [www.arm.com/university](http://www.arm.com/university). Email: [university@arm.com](mailto:university@arm.com)

**mbed:** <http://mbed.org>

For comments or corrections on this document please email [bob.boys@arm.com](mailto:bob.boys@arm.com).

For more information on the Arm CMSIS standard: [www.keil.com/cmsis](http://www.keil.com/cmsis).

## 32) Keil Products and Contact Information: See [www.keil.com/NXP](http://www.keil.com/NXP)

### Keil Microcontroller Development Kit (MDK-ARM™)

- **MDK-Lite™** (Evaluation version) 32K Code and Data Limit - \$0
- **New MDK-ARM-Essential™** For all Cortex-M series processors – unlimited code limit
- **New MDK-Plus™** MiddleWare Level 1. ARM7™, ARM9™, Cortex-M, SecureCore®.
- **New MDK-Professional™** MiddleWare Level 2. For details: [www.keil.com/mdk5/version520](http://www.keil.com/mdk5/version520).

For the latest MDK details see: [www.keil.com/mdk5/selector/](http://www.keil.com/mdk5/selector/)

Keil Middleware includes Network, USB, Graphics and File System. [www.keil.com/mdk5/middleware/](http://www.keil.com/mdk5/middleware/)

### USB-JTAG/SWD Debug Adapters All ULINK products support MTB.

- ULINK2 - (ULINK2 and ME - SWV only – no ETM)
- **New ULINKplus-** Cortex-Mx High performance SWV & power measurement.
- ULINKpro - Cortex-Mx SWV & ETM instruction trace. Code Coverage and Performance Analysis.
- ULINKpro D - Cortex-Mx SWV no ETM trace ULINKpro works with Arm DS-5.

You can use OpenSDA on the Kinetis boards. For Serial Wire Viewer (SWV), a ULINK2, ULINK-ME or a J-Link is needed. For ETM support, a ULINKpro is needed. OS-JTAG or OpenSDA do not support SWV or ETM. KL25 does not have SWV.

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

Keil RTX RTOS is now provided under a Berkeley BSD or Apache 2.0 license. This makes it free.

All versions, including MDK-Lite, includes Keil RTX RTOS *with source code* !

**New** MDK now supports FreeRTOS.

Keil provides free DSP libraries for the Cortex-M processors.

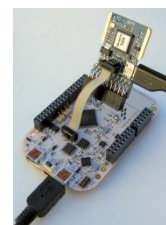
Sales can provide advice about the various tools options available to you. They will help you find various labs and appnotes that are useful.

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](http://www.arm.com/university)

Keil supports many other NXP processors including ARM9™, Cortex-Rand Cortex-A processors. See [www.keil.com/NXP](http://www.keil.com/NXP) for the complete list.



### For more information:

**Sales In Americas:** [sales.us@keil.com](mailto:sales.us@keil.com) or 800-348-8051. **Europe/Asia:** [sales.intl@keil.com](mailto:sales.intl@keil.com) +49 89/456040-20

**Keil Technical Support** in USA: [support.us@keil.com](mailto:support.us@keil.com) or 800-348-8051. Outside the US: [support.intl@keil.com](mailto:support.intl@keil.com).

**Global Inside Sales Contact Point:** [Inside-Sales@arm.com](mailto:Inside-Sales@arm.com) **Arm Keil World Distributors:** [www.keil.com/distis](http://www.keil.com/distis)

**Forums:** [www.keil.com/forum](http://www.keil.com/forum) <http://community.arm.com/groups/tools/content> <https://developer.arm.com/>

