

# Freescal e CUP Challenge: Cortex-M0+ Tutorial

## Using the Freedom KL25Z and ARM Keil MDK 5.10 Toolkit

featuring MTB: Micro Trace Buffer  
Winter 2014 Version 1.3 by Robert Boys

**ARM KEIL**  
Microcontroller Tools

### Introduction:

The latest version of this document is here: [www.keil.com/appnotes/docs/apnt\\_257.asp](http://www.keil.com/appnotes/docs/apnt_257.asp)

The purpose of this tutorial is to introduce you to the ARM® Keil® MDK toolkit. You will quickly learn how to create, compile, load and run your programs. This document will greatly shorten the learning curve for your development of your entry to the Freescale CUP Challenge. You can use either the Freescale MQX or Keil RTX Real Time OS, or bare-metal.

For community help see [www.keil.com/forum](http://www.keil.com/forum) and <http://community.arm.com/groups/tools/content>.

Obtain the new “Getting Started MDK 5” manual here: [www.keil.com/mdk5](http://www.keil.com/mdk5).

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. See the last page of this document for more labs, appnotes and other information including MQX support in MDK. Also see [www.keil.com/freescale/](http://www.keil.com/freescale/).

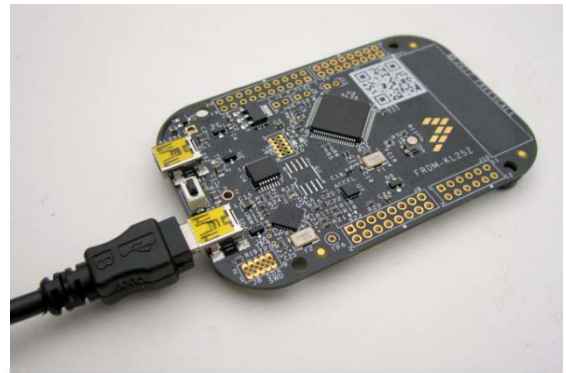
**MDK toolkit:** This document uses MDK 5.10 or later. It is possible to use MDK 4.7x but you will have to adapt this lab. See page 3 details. MDK-Lite is a free evaluation version that limits code size to 32 Kbytes. Nearly all Keil examples will compile in this 32K limit. The addition of a valid license number will turn MDK into a commercial version.

**RTX RTOS:** MDK contains the full version of RTX with Source Code. See [www.keil.com/rl-arm/kernel.asp](http://www.keil.com/rl-arm/kernel.asp).

### Why Use Keil MDK ?

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

1. µVision® IDE with Integrated Debugger, Flash programmer and the ARM® Compiler, assembler and linker toolchain. MDK is a complete turn-key tool solution.
2. A full feature Keil RTOS called RTX is included with MDK. RTX comes with a BSD type license. Source code is provided with MDK and it is CMSIS-RTOS compliant.
3. All applicable ARM debugging technology is supported.
4. OpenSDA is supported. (It is CMSIS-DAP compliant).
5. Kernel Awareness is available for Keil RTX and Freescale MQX. RTX Kernel Awareness is updated in real-time. Many other RTOSs are compatible with MDK.
6. **MQX:** An MQX port for MDK is available including Kernel Awareness windows. See [www2.keil.com/freescale/mqx](http://www2.keil.com/freescale/mqx).
7. **Processor Expert** compatible. For more information see [www.keil.com/appnotes/files/apnt\\_235\\_pe2uv.pdf](http://www.keil.com/appnotes/files/apnt_235_pe2uv.pdf).
8. A lab for the Freedom K20D50M Cortex-M4 is located here: [www.keil.com/appnotes/docs/apnt\\_250.asp](http://www.keil.com/appnotes/docs/apnt_250.asp)



The Freedom KL25Z board connected to run OpenSDA (CMSIS-DAP) with Keil µ Vision:

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

1. Micro Trace Buffer (MTB). A history of the executed instructions showing where your program has been.
2. Real-time display of memory locations and variables in the Watch and Memory 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. Call Stack and Locals window displaying vital information about called functions and their local variables.
5. System and Thread Viewer window: kernel awareness for RTX that updates while your program is running.

### Micro Trace Buffer (MTB):

MDK supports MTB with OpenSDA (CMSIS-DAP), ULINK™2/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 with examples is described in this document.

## General Information:

1. Freescale Evaluation Boards & Keil Evaluation Software: 3
2. MDK 4.73 versus MDK 5.1x Keil Software and OpenSDA: 3
3. Keil Software Download and Installation: 4
4. Software Packs and Example file Install: 5

## Using the OpenSDA CMSIS-DAP Debug Adapter:

5. Programming the KL25Z with OpenSDA: 6
6. Testing the OpenSDA Connection: 6

## Blinky example with various Cortex-M0+ Debug Features:

7. *Blinky* example using the Freedom KL25Z and OpenSDA: 7
8. Hardware Breakpoints: 7
9. Call Stack & Locals window: 8
10. Watch and Memory windows and how to use them: 9
11. How to view Local Variables in Watch and Memory windows: 10
12. System Viewer (SV): 11
13. Watchpoints: Conditional Breakpoints: 12

## MTB Instruction Trace:

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

## RTX: ARM’s RTOS

21. RTX\_Blinky: Keil RTX RTOS example: 19
22. RTX Kernel Awareness using RTX Viewer: 20

## Additional Information:

23. Blinky Project File System Explanation: 21
24. RTX\_Blinky Project File System Explanation: 22
25. Creating your own MDK 5 Blinky project from scratch: 23
26. Creating your own MDK 5 RTX Blinky project from scratch: 26
27. Interesting Bits & Pieces: 27
28. CoreSight Definitions: 28
29. KL25 Cortex-M0+ Trace Summary: 29
30. Document Resources: (see this section for very useful information) 30
31. Keil Contact Information: 30

## Using this document:

1. The latest version of this document and the necessary example source files are available here:  
[www.keil.com/apnotes/docs/apnt\\_257.asp](http://www.keil.com/apnotes/docs/apnt_257.asp)
2. It is very beneficial for you to work through these few examples to get MDK working for you as fast as possible. Be careful and don’t miss any steps. This is the number 1 reason people can’t get MDK to work properly the first time. MDK is very easy to use and is designed to “work out of the box”. You can start writing your programs very quickly.

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

Keil provides board support for Kinetis Cortex-M0+ and Cortex-M4 processors. They include KwikStik, Tower K20, K40, K53, K60, K70 and **KL25Z (both Tower and Freedom boards)**. For Vybrid and the i.MX series see [www.arm.com/ds5](http://www.arm.com/ds5)

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 Freescale boards.

We recommend you obtain the latest Getting Started Guide for MDK5: It is available on [www.keil.com/mdk5/](http://www.keil.com/mdk5/).

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

---

## 2) MDK 4.73 vs MDK 5.10 Keil Software: *This document uses MDK 5.10 or later.*

**MDK 4.7x: Legacy MDK.** Does not contain Software Packs. Projects created with 4.7x are compatible with MDK 5.0 and up. MDK 4.73 is the current official Keil release. It is available at [www.keil.com/demo/eval/armv4.htm](http://www.keil.com/demo/eval/armv4.htm)

**Example Project Files:** MDK 4.73 provides various example programs. They are stored in C:\Keil\ARM\Boards\Freescale\. The directory FRDM-KL25Z is for the Freedom KL25 board and XTWR-KL25Z48M is for the Tower version of the KL25.

**MDK 5.10:** This is the new MDK containing Software Packs. This is a new method of providing software including middleware, CMSIS header and configuration files, RTX and middleware from a webserver. This document uses MDK 5.10. These Packs are downloaded with the Packs Installer and configured with the Run Time Environment (RTE). See page 5.

MDK 4 projects will run on MDK 5.10. You normally have to add MDK v4 Legacy Support: [www2.keil.com/mdk5/legacy](http://www2.keil.com/mdk5/legacy)

**Example Project Files:** This document uses the RTX\_Blinky example project contained in MDK 5.10. The Blinky example is available where this document is stored: [www.keil.com/appnotes/docs/apnt\\_257.asp](http://www.keil.com/appnotes/docs/apnt_257.asp)

To import 4.7x examples into MDK 5.10: [www2.keil.com/mdk5/legacy](http://www2.keil.com/mdk5/legacy). You do not need to install these files for this tutorial.

**MDK 4.7x and 5.10 both have:**

- Complete OpenSDA (CMSIS-DAP) support.
- MTB Trace works with OpenSDA, ULINK2, ULINK-ME and ULINK<sub>pro</sub>.
- Example files for the Freedom and Tower KL25Z boards and many others. MDK 5.10 has a collection of CMSIS-Pack compliant example files including for Kinetis processors. More are to be added.

**RTX:** MDK 4.7x and MDK 5.10 RTX are slightly different. MDK 4.7x is not compliant to CMSIS-RTOS. MDK 5.10 RTX is compliant. The main difference is the different names of functions called. They both work essentially the same.

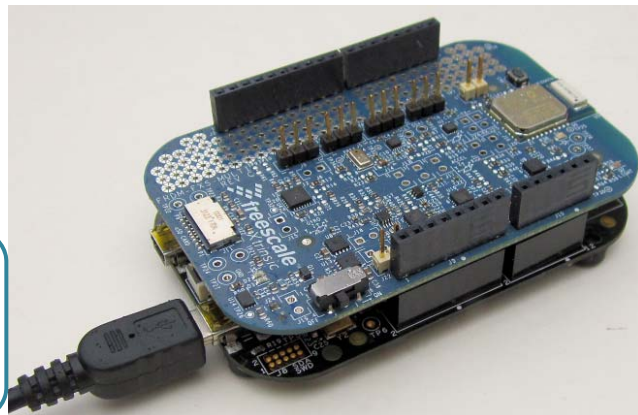
---

**OpenSDA:** OpenSDA is Freescale's name for ARM's CMSIS-DAP. This is an ARM standard that specifies an on-board debug adapter. The Freedom board incorporates CMSIS-DAP. You are able to incorporate CMSIS-DAP debugger on your own board. See [www.arm.com/cmsis](http://www.arm.com/cmsis). You do not need an external debugger such as a ULINK2 to do this lab.

If you construct your own board, in order to debug, you either have to add a CMSIS-DAP processor or connect a ULINK.

---

KL25Z Freedom board connected to a Freescale Xtrinsic 12 axis Sensor FRDM-FXS-MULTI-B board. This particular board version has the Bluetooth option.



### 3) Keil Software Download and Installation:

Download MDK-Core Version 5

1. Download MDK 5.10 or later from the Keil website. [www.keil.com/mdk5/install](http://www.keil.com/mdk5/install)
2. Install MDK into the default directory. You can install into any directory, but this lab uses the default C:\Keil\_v5
3. If you install MDK into a different directory, you will have to adjust for the directory differences.
4. This lab will use C:\MDK\ for the examples.

#### 1) Using an MDK License:

5. **You do not need to install a license to use this tutorial.** Without a license, MDK will run as MDK-Lite, the evaluation version. It will have a compile limit of 32 K which is sufficient for the examples used here. If you install a license, code size will be unlimited and certain other restrictions are lifted.
6. Once a license is installed, MDK-Lite turns in to the full commercial version as specified by your license type.
7. You can install the license at any time. CUP licenses are node-locked. They are tied to a particular computer.



#### 2) Obtaining an MDK License:

As part of its global sponsorship of the Freescale CUP contest, the ARM University Program is pleased to provide a license for the full KEIL® MDK-ARM Professional microcontroller development kit to each participating team.

KEIL MDK-ARM is a complete software development environment for ARM®-based microcontrollers. The tool is specifically designed for ARM®-based microcontroller application development, capable of the most demanding embedded applications.

For more information about KEIL MDK-ARM, please follow this link: [www.keil.com/arm/](http://www.keil.com/arm/)

Instructions on downloading and installing MDK are on the next page.

To request a license for the full KEIL MDK-ARM Professional microcontroller development kit, the Academic in charge of each competing team should submit a donation request by following this link:

[www.arm.com/support/university/educators/embedded/embeddedsystemsmcus-software-tools-for-educators.php](http://www.arm.com/support/university/educators/embedded/embeddedsystemsmcus-software-tools-for-educators.php)

... and clicking on the “Request Donation” button. In the “purpose of donation” field of the email, please mention “Freescale CUP” and your team name. The license type is “Node locked”. KEIL MDK-ARM is supported in Windows OS only.

This license is valid until the worldwide finals of the 2014 Freescale CUP edition (August 31, 2014). After that time, MDK will revert to MDK-Lite, the evaluation version with its 32 K code limit.

For any further questions, please contact the ARM University Program at: [university@arm.com](mailto:university@arm.com)

**For additional information about the ARM University Program:**

**Students:** [www.arm.com/support/university/students](http://www.arm.com/support/university/students)

**Educators:** [www.arm.com/support/university/educators](http://www.arm.com/support/university/educators)

---

#### 3) Installing an MDK License:

A MDK license starts as a 15 character PSN: (Product Serial Number). This is the number you will receive from Keil. µVision generates a 10 character CID (Computer ID) that identifies your computer in its License Management window. You then use the PSN and CID to register on the Keil website.

A 30 character LIC (License ID Code) will then be provided to you to copy and paste into µVision.

The license management section is found in µVision under File/License Management...


See [www.keil.com/download/license](http://www.keil.com/download/license) for detailed instructions on how to install an MDK license in µVision.

---

## 4) µVision Software Packs Download and Install Process:

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


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

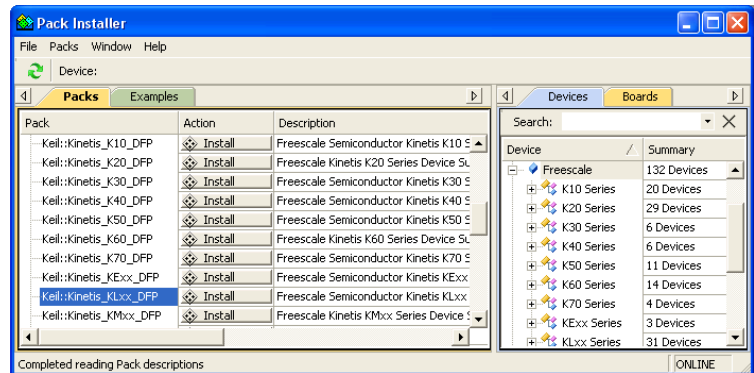
2. Start µVision by clicking on its desktop icon: 

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:

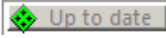
5. Note “ONLINE” is displayed at the bottom right. If “OFFLINE” is displayed, connect to the Internet before continuing.

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:

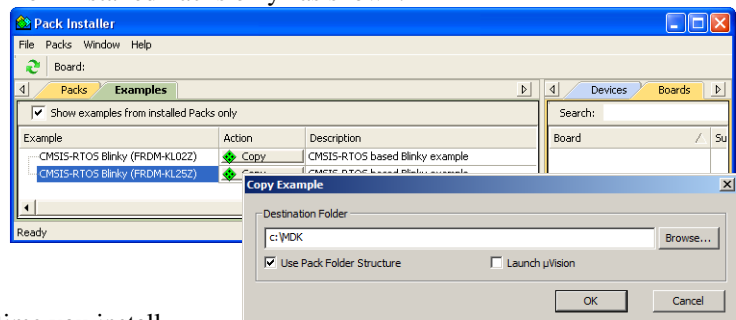
1. Initially, the Software Pack ARM::CMSIS is installed by default. Now, you will install Keil::Kinetis\_KLxx\_DFP. This is for the KL25 Freedom board.
2. Select Keil::Kinetis\_KLxx\_DFP and click on Install. This Software Pack will download and install to C:\Keil\_v5\ARM\Pack\Keil\Kinetis\_KLxx\_DFP\1.0.0 by default. This download can take two to four minutes.

3. Its status is indicated by the “Up to date” icon: 

**TIP:** If you click on the Devices tab, you can select the processor you are using and this will be displayed in the Packs tab.

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

1. Select the Examples tab. Select “Show examples from installed Packs only” as shown:
2. Select CMSIS-RTOS Blinky (FRDM-KL25Z):
3. Select Copy  opposite CMSIS-RTOS Blinky (FRDM-KL25Z): as shown:
4. The Copy Example window opens up: In the Destination Folder box enter C:\MDK. Select Use Pack Folder Structure: Click OK to copy.
5. The RTX\_Blinky example will now copy to C:\MDK\Boards\Freescale\FRDM-KL25Z\.

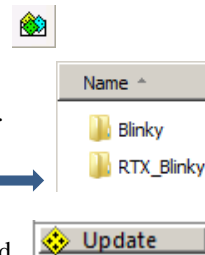


**TIP:** The default directory for copied examples the first time you install MDK is C:\Users\<user>\Documents. For simplicity, we will use the default directory of C:\MDK\ in this tutorial. You can use any directory you prefer.

6. Close the Packs Installer. You can open it any time by clicking on its icon.

### 4) Install the Blinky Example: (bare metal – no RTOS)

1. Obtain the software zip file from [www.keil.com/appnotes/docs/apnt\\_257.asp](http://www.keil.com/appnotes/docs/apnt_257.asp).
2. Unzip this into the directory C:\MDK\Boards\Freescale\FRDM-KL25Z.
3. A Blinky folder will be created along with the RTX\_Blinky folder.



**TIP:** An Update icon means there is an updated Software Pack available for download.

**TIP:** If you look in the directory C:\Keil\_v5\ARM\Pack\Keil\Kinetis\_KLxx\_DFP\1.0.0\Boards\Freescale\FRDM-KL25Z\ you will find another RTX\_Blinky. This is a read-only version. Use only the projects you copied over from the Examples tab. If you ever need to modify a read only file, you must change its permissions in the usual manner.



## 5) 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...***

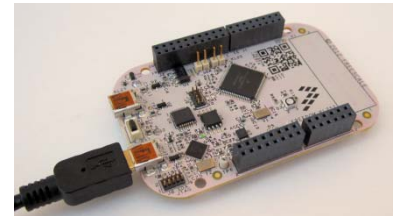
### Program the K20 with the CMSIS-DAP application file CMSIS-DAP.S19:

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

1. CMSIS-DAP.S19 is located in the OpenSDA directory in MDK. Using Windows Explorer navigate to C:\Keil\ARM\Boards\Freescale\FRDM-KL25Z\RTX\_Blinky\OpenSDA. CMSIS-DAP.S19 is located here. 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 the Bootloader USB device.


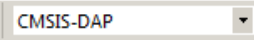


#### 4) Exit Bootloader Mode:

7. Cycle the power to the Freedom board. The green led will blink once and then stay off.
8. The Freedom board is now ready to connect to 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 a 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.

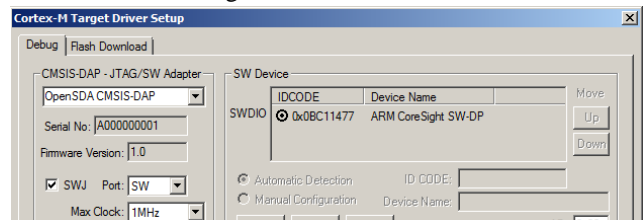
**TIP:** This application will remain in the U6 K20 Flash each time the board power is cycled with RESET off. The next time board is powered with RESET held on, it will be erased. CMSIS-DAP.S19 is the CMSIS application in the Motorola S record format that loads and runs on the K20 OpenSDA processor.

## 6) Testing The OpenSDA Connection: (Optional Exercise)


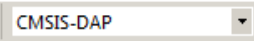


1. Start  $\mu$ Vision  if it is not already running. Select Project/Open Project.
2. Select the Blinky project C:\Keil\ARM\Boards\Freescale\FRDM-KL25Z\Blinky\Blinky.uvprojx.
3. Select "CMSIS-DAP" in the Select Target menu. 
4. Select Target Options  or ALT-F7 and select the Debug tab: 
5. Click on Settings: and the window below opens up: If an ICODE 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.
6. If nothing or an error is displayed in this SW Device box, this **must** be corrected before you can continue.

**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.




**TIP:** To see more detailed information concerning configuring  $\mu$ Vision see [www.keil.com/appnotes/docs/apnt\\_232.asp](http://www.keil.com/appnotes/docs/apnt_232.asp) This is the full version of this tutorial designed for general use. This document leaves out some information that is not needed for the Freescale Freedom CUP Challenge.



## 7) *Blinky* example program using the Freescale Freedom KL25Z and OpenSDA:

1. Now we will connect a Keil MDK development system using the Freedom board and OpenSDA as the debug adapter. OpenSDA is Freescale's implementation of CMSIS-DAP. Your board **must** have the application CMSIS-DAP.S19 programmed into the OpenSDA processor before you can continue. See the previous page.
2. Connect a USB cable between your PC and Freedom USB J7 marked SDA. See page 1 for a photo.
3. Start  $\mu$  Vision by clicking on its desktop icon. 
4. Select Project/Open Project.
5. Open the file: C:\MDK\Boards\Freescale\FRDM-KL25Z\Blinky\Blinky.uvprojx.
6. Select "CMSIS-DAP" in the Select Target menu.   
This is where you create and select different target configurations such as to execute a program in RAM or Flash. This Target selection is pre-configured to use OpenSDA which is ARM CMSIS-DAP compliant.
7. Compile the source files by clicking on the Rebuild icon. . You can also use the Build icon beside it.
8. Program the KL25Z Flash by clicking on the Load icon:  Progress will be indicated in the Output Window.

**TIP:** If you get an error this probably means the Freedom board is not programmed with CMSIS-DAP.S19. Refer to the previous page for instructions. Cycle the power to the board.

9. Enter Debug mode by clicking on the Debug icon.  Select OK if the Evaluation Mode box appears.
10. The green LED D4 will illuminate indicating a successful connection. If not, make sure the board is connected to your PC and the OpenSDA must be programmed into the board. You must fix this before continuing.
11. Click on the RUN icon.  Note: you stop the program with the STOP icon. 

**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 now run Blinky stand-alone. Blinky is now programmed in the Flash until reprogrammed. You can use  $\mu$ Vision to program the Flash with your program. Remove the USB cable, power the board, RESET the board and it will run.

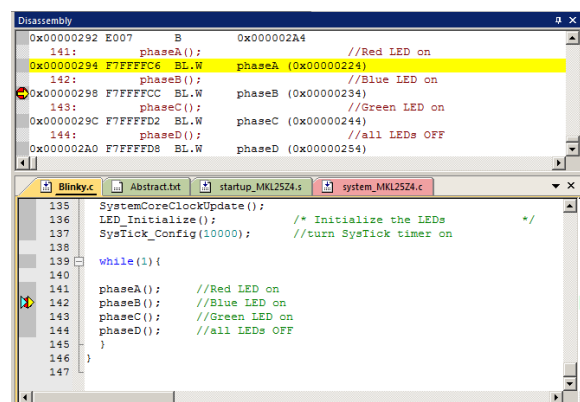
**TIP:** You can use this Blinky project as a template for your bare metal (no RTOS) project. If you want to use an RTOS (recommended), see the RTX\_Blinky example. This is on page 19. Details of the Blinky project are in Section 23 on page 21.

## 8) 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 main() function will work as shown below: 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 windows as shown here:
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.**
7. Clicking in the source window will indicate the appropriate code line in the Disassembly window and vice versa. This is relationship indicated by the cyan arrow and the yellow highlight:

**TIP:** ARM hardware breakpoints do **not** execute the instruction they are set to and land on. CoreSight hardware breakpoints are no-skid. This is a rather important feature for effective debugging.


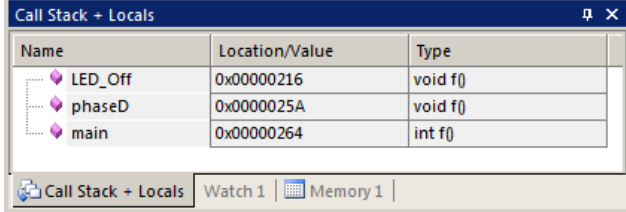

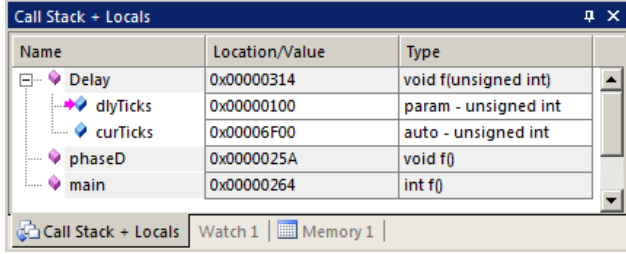



## 9) Call Stack + Locals Window:

### 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.

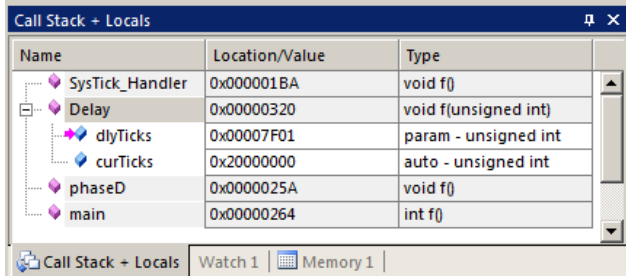
If possible, the values of the local variables will be displayed and if not the message <not in scope> will be displayed. The Call + Stack window presence or visibility can be toggled by selecting View/Call Stack Window in the main µVision window when in Debug mode.


1. Click on RUN .
2. Set a breakpoint in the LED\_Off function near line 94 in Blinky.c. It will soon stop on this breakpoint.
3. Click on the Call Stack + Locals tab if necessary to open it.
4. Shown is this Call Stack + Locals window: 
5. The functions as they were called are displayed. If these functions had local variables, they would be displayed.
6. Click on the Step In icon  or F11 a few times:
7. Continue until the program enters the Delay function. The Call Stack + Locals window will now show Delay with its two local variables and their values: 
8. Click on the StepOut icon  or CTRL-F11 a few times in order to exit all function(s) to return to main().
9. 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:** This is standard “Stop and Go” debugging. ARM CoreSight debugging technology can do much better than this. You can display global and static variables or structures, all updated in real-time in the Watch or Memory windows while the program is running. No additions or changes to your code are required. Variable update while the program is running is not possible with local variables because this type of variable is usually stored in a CPU register.

### Call Stack:

The list of stacked functions is displayed when the program is stopped as you have seen. This is useful when you need to know which functions have been called and are stored on the stack. A breakpoint was set in the SysTick\_Handler function and this event is clearly shown at the top of this window: 

As you click on the StepOut icon  or Ctrl-F11 each function will be removed as it comes off the stack until you are left with only main(). These are added as the program is run and stopped.

**TIP:** You can set two hardware breakpoints with the Freescale 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.

**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...

**TIP:** You can access the Hardware Breakpoint table by clicking on Debug/Breakpoints or Ctrl-B. This is also where Watchpoints (also called Access Points) are configured. You can temporarily disable entries in this table by unchecking them.





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

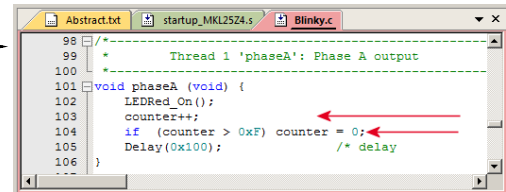
The Watch and Memory windows will display updated variable values in real-time. It does this using the ARM CoreSight debugging technology that is part of Cortex-M processors. You can right click on a variable and select Add *varname* to.. and select the appropriate window or “drag and drop” variable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





### Watch window:

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

1. Stop the program if running by clicking on the STOP icon. . Exit Debug mode. 
2. Declare a global variable (I called it counter) near line 15 in Blinky.c: **unsigned int counter = 0;**
3. Add these statements near Line 103 just after LEDred\_On();:

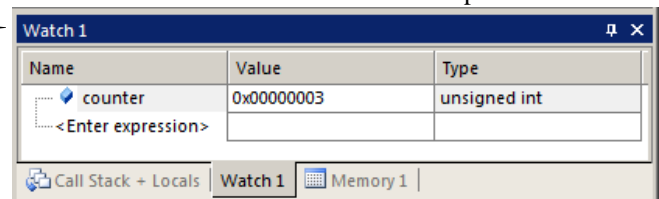
```
counter++;  
if (counter > 0xF) counter = 0;
```



4. Click on Rebuild  and program the Flash with Load .
5. 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.
6. Select View/Periodic Window Update if counter updates only when the program is stopped.

☒ Periodic Window Update

7. In Blinky.c, right click on **counter**, select Add counter to ... and select Watch 1. Watch 1 will open automatically.  
**counter** will be displayed as shown here:
8. **counter** will update in real time.



**TIP:** You can change a value in the Watch window when the program is stopped.

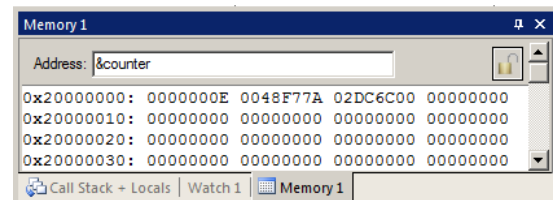
**TIP:** To Drag 'n Drop into a tab that is not active, pick up the variable and hold it over the tab you want to open: when it opens, move your mouse into the window and release the variable.

### Memory window:

1. In Blinky.c, right click on **counter**, select Add counter to ... and select Memory 1.
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. Right click in the memory window and select Unsigned/Int.
4. Add an ampersand "&" in front of the variable name and press Enter. The physical address is shown (0x2000\_0000).
5. The data contents of **counter** is now displayed as shown here:
6. Both the Watch and Memory windows are updated in real-time.

#### How to change a memory value or variable on-the-fly.

7. Right-click with the mouse cursor over the memory field of counter and select Modify Memory. Enter a value and click OK.












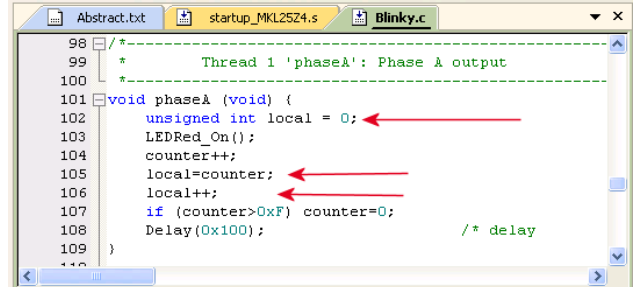
**TIP:** No CPU cycles are used to perform these operations.

**TIP:** To view variables and their location use the Symbol window. Select View/Symbol Window while in Debug mode.

These Read and Write accesses are handled by the Serial Wire Debug (SWD) connection via the CoreSight Debug Access Port (DAP), which provides on-the-fly memory accesses.

## 11) How to view Local Variables in the Watch or Memory windows:

1. Stop the processor  and exit Debug mode. .
2. Create a local variable in the Thread 1 function phaseA near line 102 where you added the variable counter.
3. I called it **local**: `unsigned int local=0;` as shown at line 102 below:
4. Add the lines `local = counter;` and `local++;` as shown at line 105 and 106 below:
5. Compile the source files . Program the Flash .
6. Enter Debug mode.  Click on RUN. .
7. Enter **local** into Watch 1 window by right clicking on it and selecting Add local to.... Watch 1.
8. Note it says <cannot evaluate> or “not in scope”. Stop the program and the value of local will still not display.
9. Start the program .
10. Set a breakpoint in the function phaseA. The program will stop and a value for **local** will now be displayed. The only time a local variable value will be displayed is if the program happens to stop while it is in scope.
11.  $\mu$ Vision is unable to determine the value of **local** when the program is running because it exists only when the function phaseA is running. It disappears in functions and handlers outside of phaseA. **local** is a local or automatic variable and this means it is probably stored in a CPU register which  $\mu$ Vision is unable to access during run time.
12. **Remove the breakpoint** and make sure the program is not running . Exit Debug mode. .



```
98 /*
99 * Thread 1 'phaseA': Phase A output
100 *
101 void phaseA (void) {
102     unsigned int local = 0;
103     LEDRed_On();
104     counter++;
105     local=counter;
106     local++;
107     if (counter>0xF) counter=0;
108     Delay(0x100);
109 }
```






### How to view local variables updated in real-time:

All you need to do is to make **local** static where it is declared in Blinky.c !

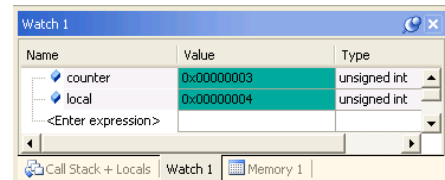
1. In the declaration for **local**, add the **static** keyword like this:

```
void phaseA (void) {
    static unsigned int local = 0;
```

**TIP:** You can also make a variable global or have it as part of a structure so it will update in real-time.

2. Compile the source files by clicking on the Rebuild icon . Select File/Save All or .
3. To program the Flash, click on the Load icon. . Enter Debug mode.  Click on RUN. .
4. Note local is still not updated in real-time. You must first show local to  $\mu$ Vision by stopping the program in phaseA.
5. Set a breakpoint in the function phaseA. The program will stop and a value for local will now be displayed. **Remove the breakpoint.** Click on RUN and **local** will now update in real-time.


**TIP:** You must fully qualify a variable in order for it to update without initially stopping the program while it is in scope. To do this, you can open the View/Symbols window and enter the variable from there. This automatically fully qualifies the variable. In this case, **local** fully qualified is [\\Blinky\\Blinky.c\\phaseA\\local](#). You also can enter this directly into the Watch or Memory windows.






Name	Value	Type
counter	0x00000003	unsigned int
local	0x00000004	unsigned int
<Enter expression>		

6. You can also enter a variable into a Memory window. Remember to prefix it with an &.

**TIP:** Recall you can modify variables or memory locations in the Memory window when the program is running.

7. Stop the CPU  for the next step. Select File/Save All.

**TIP:** View/Periodic Window Update must be selected. Otherwise variables update only when the program is stopped.

**TIP:** To program the Flash automatically when you enter Debug mode select Target Options , select the Utilities tab and select the “Update Target before Debugging” box. This is set by default in the two Blinky projects. This means you can skip using the LOAD icon.  This will be automatically be done when you enter Debug mode. .

## 12) System Viewer (SV):

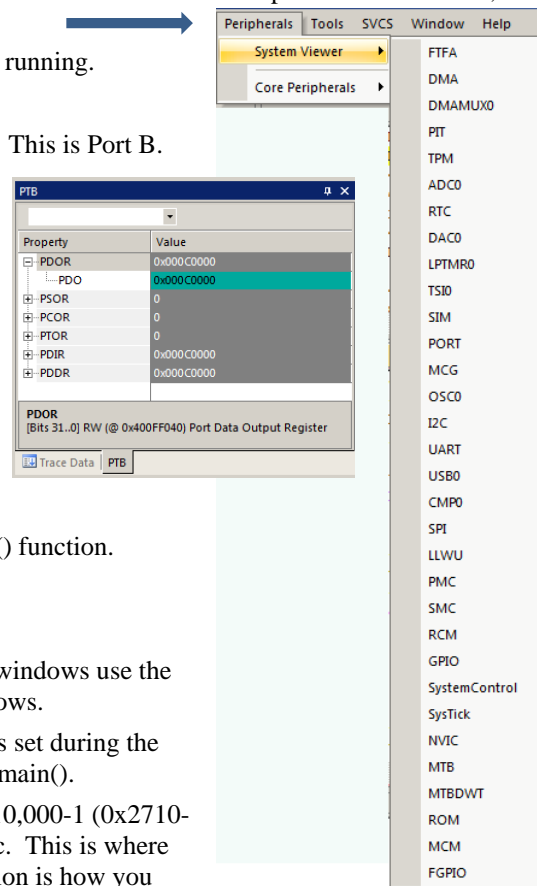
The System Viewer provides the ability to view registers in the CPU core and in peripherals. In most cases, these Views are updated in real-time while your program is running. These Views are available only while in Debug mode. There are two ways to access these Views: **a) View/System Viewer** and **b)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



### GPIO Port B:

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

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

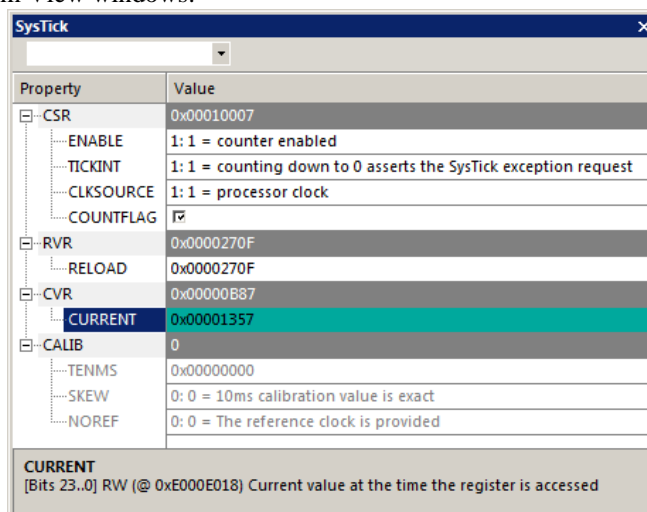


**SysTick Timer:** This program uses the SysTick timer as part of the Delay() function.

1. Select Peripherals/System Viewer and then select SysTick.
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. Expand the RVR register. This is the reload register value. This is set during the SysTick configuration by the SysTick\_Config(10000) function in main().
5. Note that it is set to 0x270F. This is the same value hex value of 10,000-1 (0x2710-1) that is programmed into SysTick\_Config() in main() in Blinky.c. 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.
6. In the RELOAD register in the SysTick window, *while the program is running*, type in 0x5000 and press Enter !
7. The blinking LEDs will slow down. This will convince you of the power of ARM CoreSight debugging.
8. Replace RELOAD with 0x270F. A CPU RESET  will also do this.
9. You can look at other Peripherals contained in the System View windows.
10. 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.

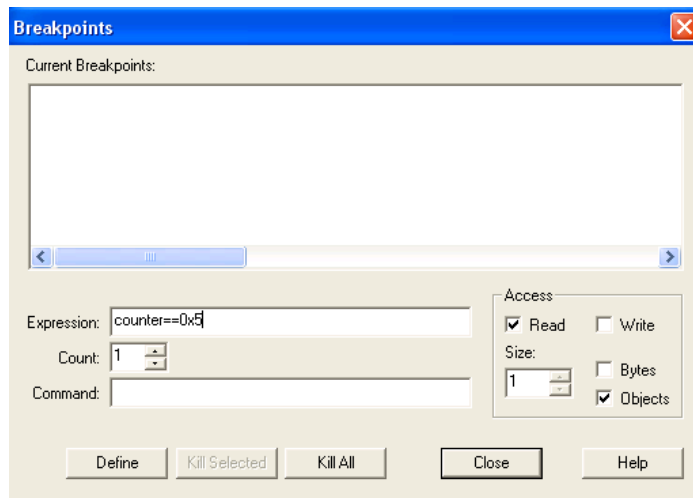
You must make sure a given peripheral register allows and will properly react to such a change. Changing such values indiscriminately is a good way to cause serious and difficult to find problems.






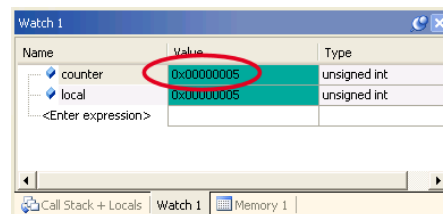
### 13) Watchpoints: Conditional Breakpoints

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 slightly intrusive. When the Watchpoint is hit,  $\mu$ Vision must test the memory location. Cortex-M3/M4 Watchpoints are not intrusive.

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. You can also 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. 

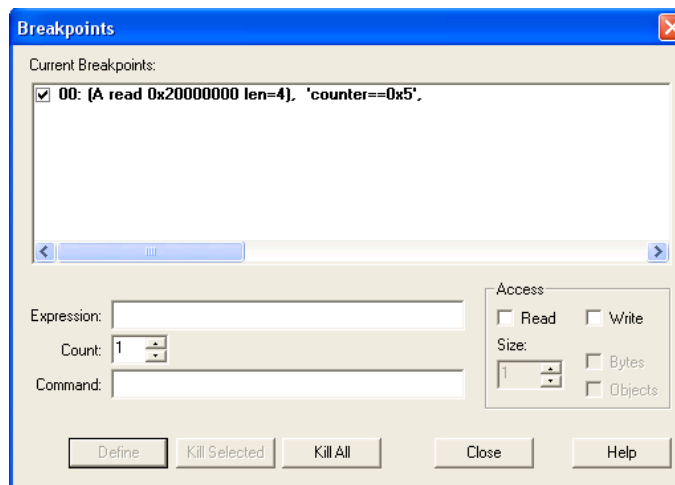


**TIP:** You cannot set Watchpoints on-the-fly while the program is running like you can with hardware breakpoints.

**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)








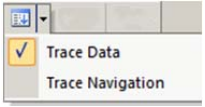
## 14) MTB: Micro Trace Buffer:

The Kinetis KL25 processor contains an instruction trace called MTB. The trace buffer is an area in 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.

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 determine the actual program flow such as when branches occurred (or not) and interrupt calls. Trace displays the order the instructions were executed as opposed as the way they were written and displayed in the source files. 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.




### Basic Instruction Trace Operation:

The project Blinky.uvprojx is pre-configured to use MTB. This exercise demonstrates the use of MTB.


1. Select CMSIS-DAP-MTB in the Target selector: 
2. Compile the source files by clicking on the Rebuild icon .
3. Program the Flash. . Enter Debug mode.  The program runs to main().
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 this 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 a frame belongs to will be displayed.
8. Note the last instruction executed was a BL.W to main() at frame 1,670 (in this example).
9. In the Disassembly window shown below, the next instruction to be executed is a BL.W to SystemCoreClockUpdate() which is the first function call in main().
10. Scroll up near Index 1,280 and you should see some C source lines.

Trace Data					
Display: Execution				in All	
Index	Address	Opcode	Instruction	Src Code	Function
1,658	X: 0x00000314	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
1,659	X: 0x00000316	C5C0	STM r5!,{r6-r7}		__user_setup_stackheap
1,660	X: 0x00000318	3D40	SUBS r5,r5,#0x40		__user_setup_stackheap
1,661	X: 0x0000031A	0049	LSLS r1,r1,#1		__user_setup_stackheap
1,662	X: 0x0000031C	468D	MOV sp,r1		__user_setup_stackheap
1,663	X: 0x0000031E	4770	BX lr		__user_setup_stackheap
1,664	X: 0x0000014C	4611	MOV r1,r2		???
1,665	X: 0x0000014E	F7FFFF5	BL.W __rt_lib_init (0x0000013C)		???
1,666	X: 0x0000013C	B51F	PUSH {r0-r4,lr}		???
1,667	X: 0x0000013E	46C0	MOV r8,r8		???
1,668	X: 0x00000140	46C0	MOV r8,r8		???
1,669	X: 0x00000142	BD1F	POP {r0-r4,pc}		???
1,670	X: 0x00000152	F000F876	BL.W main (0x00000242)		???

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

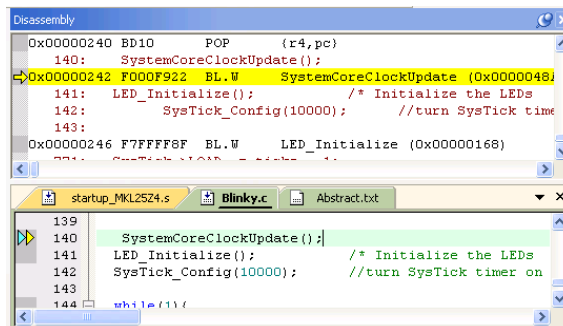
1. In the example above, the last instruction executed was a BL.W to 0x0242. This calls the beginning of main().
2. In the Registers window, the PC (R15) indeed points to 0x0242.
3. Cyan  is a marker from a source window to the Disassembly where it becomes a yellow highlight block.
4. The Yellow arrow  and Yellow triangle  is where the PC is pointing to.
5. Double-click in the Trace Data window on a frame and it is indicated in the Disassembly and source windows.

### Single-Stepping:

1. Put the Disassembly window in focus by clicking inside it.
2. Click on Step (F11). 
3. The BL.W at 0x242 will be executed and added to the bottom of the Trace Data window.

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

1. Examine the Disassembly and Blinky.c windows and see how the trace position is tracked in these windows. The next instruction to be executed is a PUSH at memory 0x048A. You can confirm the PC is equal to 0x048A in the Registers window.
2. Click on RUN and Step 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.



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



## 15) Exploring the MTB Instruction Trace:

MTB provides a record of all instructions executed. No CPU cycles are stolen to accomplish this.

Make sure all breakpoints and Watchpoints are removed. Enter Ctrl-B or Debug/Breakpoints and select Kill All.

**Note:** You might see different addresses as shown here depending on compiler settings. These are using MDK 5.10.

1. Stop the CPU. Click on RESET. Clear the Trace . This makes it easier to see the frames we want.
2. In the Command window, enter “g, main” without the quotes and press Enter. The program will run to main().
3. The Data Trace window as seen on the previous page will display.

**Note these items:**

1. The last instruction executed was a branch: BL.W main which is located at memory 0x0152.
2. Double-click on the POP at 1,669. This is just above the BL.W.
3. Examine this instruction in the Disassembly window. Note there is no clue this POP would result in the next instruction being BL.W. The MTB has faithfully recorded that it was. This is one of the powers of instruction trace.

**TIP:** It is possible to have determined the next instruction after the POP by carefully examining the stack. But using the trace, as you can see, is much easier and faster. If the stack is destroyed by a program crash, the trace will still be available.

**Examine the Start of the Blinky Program:**

4. Scroll to the top of Trace Data. We want to look at the very first instruction executed but it has been overrun.
5. Double-click on one of the instructions close to the top of the Trace Data window.
6. Set a breakpoint at this instruction in the Disassembly window. It will be highlighted in yellow and with a red circle.
7. Exit and re-enter Debug mode. The processor will stop at this breakpoint. Remove this breakpoint.
8. Scroll to the top of the Trace Data window. Note the Function column shows the functions the instructions belong to.
9. The first occurrence is blocked in orange to help you easily locate these.

10. Note the first instruction is at 0x2A8 and is an LDR instruction. This is the very first instruction executed.

11. Open Memory 1 window and enter address 0x0. Right click and select Unsigned Long. See the window below: This is the start of Flash

12. In the window below, 0x00 is the Initial Stack Pointer (0x2000\_0470).

13. 0x4 is the initial PC and is 0x2A9. Bit 0 indicates Thumb<sup>®</sup> 2 instruction so subtract 1 and you get 0x2A8.

This is the address of the first instruction in the Trace which is the first instruction executed after RESET.

14. Click once somewhere in the Disassembly window.

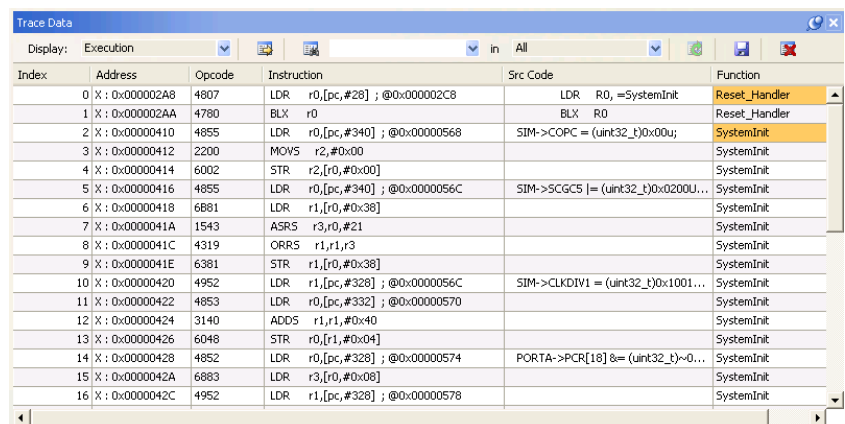
15. Click on Step (F11) and see the Blinky program step by each instruction executed and recorded in the Trace Data window.

16. Click once somewhere in the source window that is in focus.

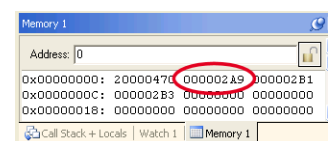
17. Click on Step (F11) and see program step by C source line and recorded in the Trace. Multiple assembly instructions will/might be recorded depending on how many are associated with each particular C source line.

18. Exit Debug mode.

**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 the instructions are executed.



Index	Address	Opcode	Instruction	Src Code	Function
0	X : 0x000002A8	4807	LDR r0,[pc,#28]; @0x000002C8	LDR R0,=SystemInit	Reset_Handler
1	X : 0x000002AA	4780	BLX r0	BLX R0	Reset_Handler
2	X : 0x00000410	4855	LDR r0,[pc,#340]; @0x00000568	SIM->COPC = (uint32_t)0x0000...	SystemInit
3	X : 0x00000412	2200	MOVS r2,#0x00		SystemInit
4	X : 0x00000414	6002	STR r2,[r0,#0x00]		SystemInit
5	X : 0x00000416	4855	LDR r0,[pc,#340]; @0x0000056C	SIM->SCGCS  = (uint32_t)0x0200U...	SystemInit
6	X : 0x00000418	6881	LDR r1,[r0,#0x38]		SystemInit
7	X : 0x0000041A	1543	ASRS r3,r0,#21		SystemInit
8	X : 0x0000041C	4319	ORRS r1,r1,r3		SystemInit
9	X : 0x0000041E	6381	STR r1,[r0,#0x38]		SystemInit
10	X : 0x00000420	4952	LDR r1,[pc,#328]; @0x0000056C	SIM->CLKDIV1 = (uint32_t)0x1001...	SystemInit
11	X : 0x00000422	4853	LDR r0,[pc,#332]; @0x00000570		SystemInit
12	X : 0x00000424	3140	ADDS r1,r1,#0x40		SystemInit
13	X : 0x00000426	6048	STR r0,[r1,#0x04]		SystemInit
14	X : 0x00000428	4852	LDR r0,[pc,#328]; @0x00000574	PORTA->PCR[18] &= (uint32_t)~0...	SystemInit
15	X : 0x0000042A	6883	LDR r3,[r0,#0x08]		SystemInit
16	X : 0x0000042C	4952	LDR r1,[pc,#328]; @0x00000578		SystemInit



Address	Value
0	0x00000470
0x00000000	20000470 000002A9 000002B1
0x0000000C	000002B3 00000000 00000000
0x00000018	00000000 00000000 00000000




## 16) Trace Buffer Control:

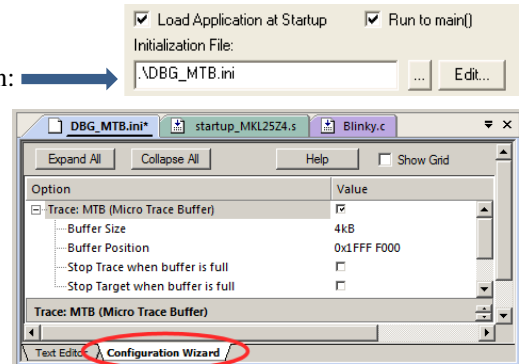
Freescall's Cortex-M0+ has two methods of controlling the trace buffer and/or the program. They are:

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:

The MTB trace is activated and controlled by the ASCII file DBG\_MTB.ini. It is executed when Debug mode is entered. To activate MTB to any project, just add this file as shown here. You must allocate the size of the internal RAM between your program and the trace buffer. Use the file from Blinky and not the RTX\_Blinky project. A RESET function is added.





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







#### 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. Select the Target Options icon . Select the Debug tab.
3. Unselect Run to main(). Click OK. After RESET, the program will not run. It will be at the initial PC address.
4. Enter Debug mode.  Click on RUN . After a second or so, click on STOP .
5. The Trace Data window does not show the BL.W to main(). A Trace Gap frame is displayed: the trace was stopped.
6. Scroll to the top of the Trace Data and confirm the first instruction (LDR) SystemInit at 0x02A8 is recorded.

#### b) Stop 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 (0x2A8) at Index 0 to the last which is a LSLS 0x0458 at Index 1,559. You may get slightly different numbers.

#### When Finished:

1. STOP the program . Exit Debug mode. 
2. Select the Target Options icon . Select the Debug tab. Select Run to main(). Click OK.
3. In DGB\_MTB.ini, Unselect "Stop Trace when buffer is full" and unselect "Stop Target when buffer is full".
4. Select File/Save All or .

#### What are these features useful for ?



The MTB trace will be over written 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.

## 17) 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:

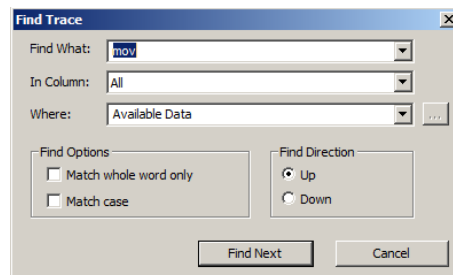
1. Enter Debug mode.  The Trace Data window will display some trace frames.
2. Click in the Trace Search window and enter a term such as push as shown here: 
3. Press Enter and the PUSH instruction or any other occurrence of the word push will be highlighted.
4. Press Enter and each time this will advance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



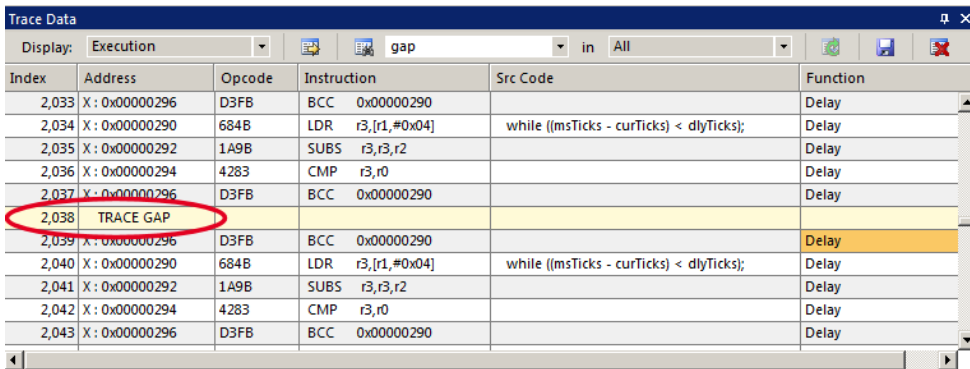
## 18) Trace Data Wrap Around:

The MTB trace buffer is limited in size. In our case, it is set to 4 KB. We have seen so far approximately 1,500 to 2,000 trace frames that can be stored. The actual number depends on the instruction size and other factors. The trace frames coming from CoreSight are highly compressed and  $\mu$ Vision 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/append 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 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 . Note 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 all are displayed.



**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.

The Trace Data window is shown with a search for 'gap' in the 'All' column. The table lists trace frames with indices, addresses, opcodes, instructions, source code, and functions. A 'TRACE GAP' entry 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

## 19) More MTB Exploration:

Now we will look at some more interesting features of the MTB.

1. In the function LEDRed\_On, set a breakpoint near line 67: `FPTB->PSOR = led_mask[LED_GREEN];`
2. Clear the Trace . Click on RUN .
3. The program will run to the breakpoint.
4. You will get a Trace Data window as 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.


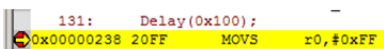
### Interrupt Subroutine Calls:

1. The SysTick timer creates Interrupt 15. This is handled in the SysTick\_Handler in Blinky.c.
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 and will show in source windows.

### Delay Function Interrupted:

1. You can see the Delay function was interrupted by the SysTick interrupt 15 at Index 2,013 and it continued at 2,018.
2. This can be very useful information in locating tricky bugs.

### Continuation of a Function:

1. Located at Index 2,023 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 131 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. If you set a breakpoint on the first instruction in the line Delay():  
4. You will see all executed instructions of phaseD except for the Delay function and the POP at the very end of phaseD at address 0x240.

2,025	X : 0x00000232	B510	PUSH {r4,lr}	void phaseD (void) {	phaseD
2,026	X : 0x00000234	F7FFFD5	BL.W LED_Off (0x000001E2)	LED_Off();	phaseD

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

Trace Data					
Display: Execution					
gap					
in All					
Index	Address	Opcode	Instruction	Src Code	Function
2,008	X : 0x00000282	D3FB	BCC 0x0000027C		Delay
2,009	X : 0x0000027C	684B	LDR r3,[r1,#0x04]	while ((msTicks - curTicks) < dl...	Delay
2,010	X : 0x0000027E	1A9B	SUBS r3,r3,r2		Delay
2,011	X : 0x00000280	4283	CMP r3,r0		Delay
2,012	X : 0x00000282	D3FB	BCC 0x0000027C		Delay
2,013	X : 0x00000196	4840	LDR r0,[pc,#256] ; @0x00000298	msTicks++;	/* in... SysTick_Handler
2,014	X : 0x00000198	6841	LDR r1,[r0,#0x04]		SysTick_Handler
2,015	X : 0x0000019A	1C49	ADDS r1,r1,#1		SysTick_Handler
2,016	X : 0x0000019C	6041	STR r1,[r0,#0x04]		SysTick_Handler
2,017	X : 0x0000019E	4770	BX lr	}	SysTick_Handler
2,018	X : 0x0000027C	684B	LDR r3,[r1,#0x04]	while ((msTicks - curTicks) < dl...	Delay
2,019	X : 0x0000027E	1A9B	SUBS r3,r3,r2		Delay
2,020	X : 0x00000280	4283	CMP r3,r0		Delay
2,021	X : 0x00000282	D3FB	BCC 0x0000027C		Delay
2,022	X : 0x00000284	4770	BX lr	}	Delay
2,023	X : 0x00000240	BD10	POP {r4,pc}	}	phaseD
2,024	X : 0x00000276	E7F6	B 0x00000266	while(1){	main
2,025	X : 0x00000266	F7FFFC4	BL.W phaseA (0x000001F2)	phaseA();	//Re... main
2,026	X : 0x000001F2	B510	PUSH {r4,lr}	void phaseA (void) {	phaseA
2,027	X : 0x000001F4	F7FFFD4	BL.W LEDRed_On (0x000001A0)	LEDRed_On();	phaseA
2,028	X : 0x000001A0	2101	MOV5 r1,#0x01	FPTB->PSOR = led_mask[LED...	LEDRed_On
2,029	X : 0x000001A2	483C	LDR r0,[pc,#240] ; @0x00000294		LEDRed_On
2,030	X : 0x000001A4	04C9	LSLS r1,r1,#19		LEDRed_On
2,031	X : 0x000001A6	6041	STR r1,[r0,#0x04]		LEDRed_On

## 20) 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 section of code in the line: `while ((msTicks - curTicks) < dlyTicks);` in the Blinky.c Delay function.

1. Enter Debug mode. Click on RUN . Click on STOP . Delay should be visible in the Call Stack window.
2. The program will probably be in the Blinky.c line (near 59): `while ((msTicks - curTicks) < dlyTicks);`
3. Note in the Disassembly window near memory location 0x284, the last instruction of the Delay function is BX lr.
4. In the Register window, change R14 (LR) to 0x0 as shown here:
5. 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:
6. This happened when the function Delay tried to return.
7. The Trace Buffer will be mostly full of B instructions.
8. *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.*
9. The PC in Blinky.c will be on the Hard Fault Handler. It is usually around address 0x02B2 as shown above.
10. Set a breakpoint on this B instruction as shown above.
11. Exit and re-enter Debug mode to reset everything.
12. Click on RUN . Then, click on STOP . Clear the trace buffer. This is to make things look clearer.
13. Set R14 (LR) to zero as before.
14. The program will immediately go to the Hard Fault state and the breakpoint will stop execution at the B.
15. The Call Stack window will not show much useful information as the Stack was destroyed on purpose in this case.
16. The Trace Data now shows the last number of instructions executed plus the BX instruction. Usually 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.
17. Note you can see the execution of the SysTick Handler. This type of event is very hard to determine without trace.
18. Click on Step (F11) a few times and the B at the HardFault\_Handler will be executed and displayed as shown below
19. Remove the breakpoint.
20. Exit Debug mode.

Index	Address	Opcode	Instruction	Src Code	Function
2,023	X : 0x00000280	4283	CHP r3,r0		Delay
2,024	X : 0x00000282	D3FB	BCC 0x0000027C		Delay
2,025	X : 0x00000196	4840	LDR r0,[pc,#256] ; @0x00000298	msTicks++;	/* in... SysTick_Handler
2,026	X : 0x00000198	6841	LDR r1,[r0,#0x04]		SysTick_Handler
2,027	X : 0x0000019A	1C49	ADDS r1,r1,#1		SysTick_Handler
2,028	X : 0x0000019C	6041	STR r1,[r0,#0x04]		SysTick_Handler
2,029	X : 0x0000019E	4770	BX lr		SysTick_Handler
2,030	X : 0x0000027C	6040	LDR r2,[r1,#0x04]	while ((msTicks - curTicks) < dlyTicks);	Delay
2,031	X : 0x0000027E	1A90	SUBS r3,r3,r2		Delay
2,032	X : 0x00000280	4203	CHP r3,r0		Delay
2,033	X : 0x00000282	D3FB	BCC 0x0000027C		Delay
2,034	X : 0x00000284	4770	BX lr		Delay

2,034	X : 0x00000284	4770	BX lr	}	Delay
2,035	X : 0x000002B2	E7FE	B HardFault_Handler (0x000002B2)	B .	HardFault_Handler
2,036	X : 0x000002B2	E7FE	B HardFault_Handler (0x000002B2)	B .	HardFault_Handler
2,037	X : 0x000002B2	E7FE	B HardFault_Handler (0x000002B2)	B .	HardFault_Handler

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

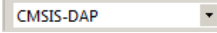





**TIP:** MTB can be used to solve many program flow problems that often require much effort to solve.



## 21) RTX\_Blinky Example Program with Keil RTX RTOS:

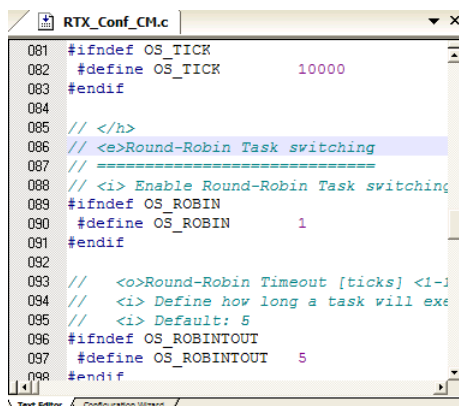
Keil provides RTX, a full feature RTOS. RTX is included as part of Keil MDK including source. It can have up to 255 tasks and no royalty payments are required. This example explores the RTX RTOS project. MDK will work with any RTOS. An RTOS is just a set of C functions that gets compiled with your project. RTX comes with a BSD type license and source code. This tutorial uses the version of RTX that is CMSIS-RTOS compliant. It is included in MDK 5.0 and later.

We highly recommend you get the **NEW! Getting Started MDK 5:** [www.keil.com/mdk5/](http://www.keil.com/mdk5/).

1. With  $\mu$  Vision in Edit mode (not in debug mode): Select Project/Open Project.
2. Open the file C:\MDK\Boards\Freescale\FRDM-KL25Z\RTX\_Blinky\Blinky.uvprojx.
3. Select the debug adapter you are using. This exercise uses CMSIS-DAP. 
4. Compile the source files by clicking on the Rebuild icon. . They will compile with no errors or warnings.
5. To program the Flash manually, click on the Load icon. . A progress bar will be at the bottom left.
6. Enter the Debug mode by clicking on the debug icon  and click on the RUN icon. 
7. The three LEDs will blink in accordance with the four tasks or threads running representing a stepper motor driver. Because there are only three LEDs and four threads, the blue LED comes on twice in the sequence.
8. Click on STOP .

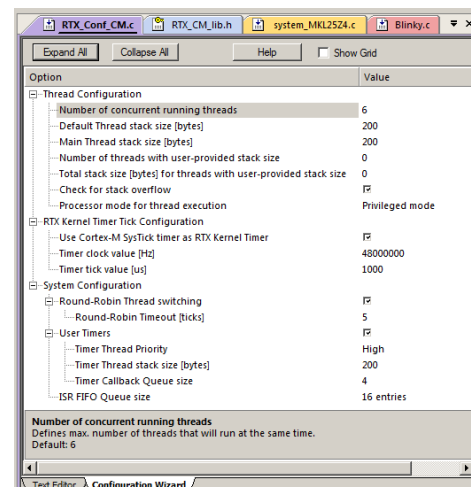
### The Configuration Wizard for RTX:

1. Double-click on RTX\_Conf\_CM.c in the Project window to open it. Note the source code to configure RTX.
2. Click on the Configuration Wizard tab at the bottom and your view will change to the Configuration Wizard.
3. Click on Expand All to open up the individual directories to show the various configuration items available.
4. See how easy it is to modify these settings here as opposed to finding and changing entries in the source code.
5. **Do not change anything at this time !**
6. Changing an attribute in one tab changes it in the other automatically. Save changes with File/Save All.
7. You can create Configuration Wizards in any source file with the scripting language as seen in the Text Editor.
8. This scripting language is shown below in the Text Editor as comments starting such as a `</h>` or `<i>`. See [www.keil.com/support/docs/2735.htm](http://www.keil.com/support/docs/2735.htm) for instructions to add this to your own source code.



```
081 #ifndef OS_TICK
082 #define OS_TICK      10000
083 #endif
084
085 // </h>
086 // <e>Round-Robin Task switching
087 // =====
088 // <i> Enable Round-Robin Task switching
089 #ifndef OS_ROBIN
090 #define OS_ROBIN      1
091 #endif
092
093 // <o>Round-Robin Timeout [ticks] <1-1
094 // <i> Define how long a task will exe
095 // <i> Default: 5
096 #ifndef OS_ROBINTOUT
097 #define OS_ROBINTOUT  5
098 #endif
```

Text Editor: Source Code






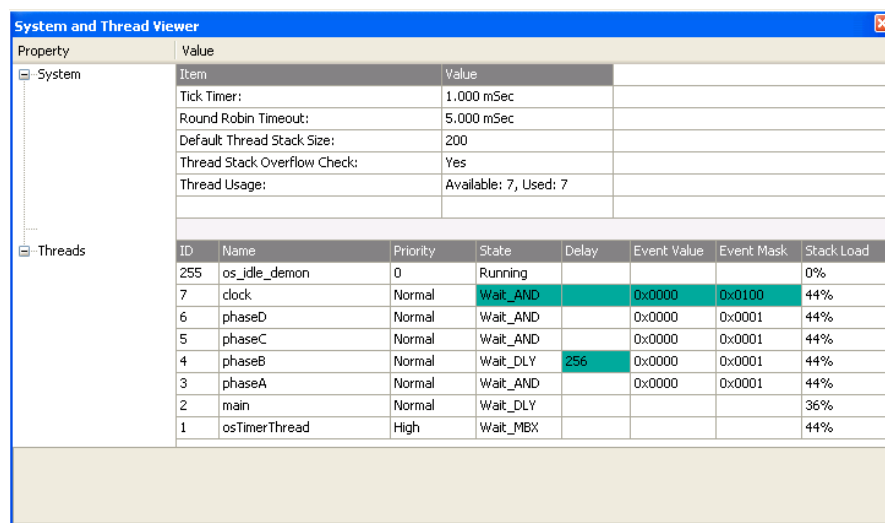
Configuration Wizard

## 22) RTX Kernel Awareness using RTX System and Thread Viewer

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 provides two such windows with the Kinetis Cortex-M4. Other RTOS companies also provide awareness plug-ins for  $\mu$ Vision.

$\mu$ Vision has extensive kernel awareness windows for MQX. See [www2.keil.com/freescale/mqx](http://www2.keil.com/freescale/mqx).

1. Run RTX\_Blinky by clicking on the Run icon. 
2. Open Debug/OS Support and select System and Thread Viewer.  System and Thread Viewer
3. The window below opens up. You have to grab the window and move it into the center of the screen. Note these values are updating in real-time using the same CoreSight 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
1. 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 `os_idle_demon`. The processor spends relatively little time in each task. You can change this to suit your needs.



Property	Value
System	
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: 7

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Running				0%
7	clock	Normal	Wait_AND		0x0000	0x0100	44%
6	phaseD	Normal	Wait_AND		0x0000	0x0001	44%
5	phaseC	Normal	Wait_AND		0x0000	0x0001	44%
4	phaseB	Normal	Wait_DLY	256	0x0000	0x0001	44%
3	phaseA	Normal	Wait_AND		0x0000	0x0001	44%
2	main	Normal	Wait_DLY				36%
1	osTimerThread	High	Wait_MBX				44%


**Demonstrating States:** (note: Tasks and Threads are used interchangeable in Keil MDK)

Blinky.c contains four threads that blink the LEDs. Thread 1 (phaseA) 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 1 as shown: (but not on the `for (;;)` line)
3. Set a breakpoint in one other thread.

4. Click on RUN .

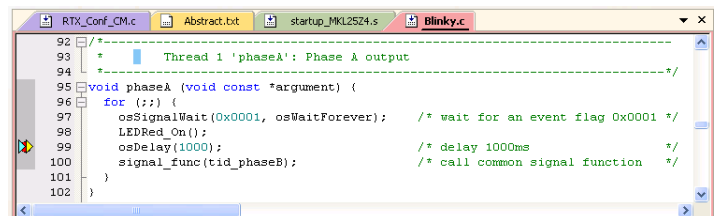
5. When the program stops, this information will be updated in the RTX Tasks window. The Task running when the program stopped will be indicated with a “Running” state. The window above shows the program stopped and phaseA is running. The states of the other tasks are displayed as well as other useful information.

6. Click on RUN . The other thread will show as “Running”. Each time you click RUN, the next thread will run.
7. Remove the breakpoints and close the RTX Tasks window.

### 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.

**This is the end of the exercises.**



## 23) Blinky Project File System Explanation:

You can use the Blinky project as a template for your own projects.

For more information regarding creating your own projects:

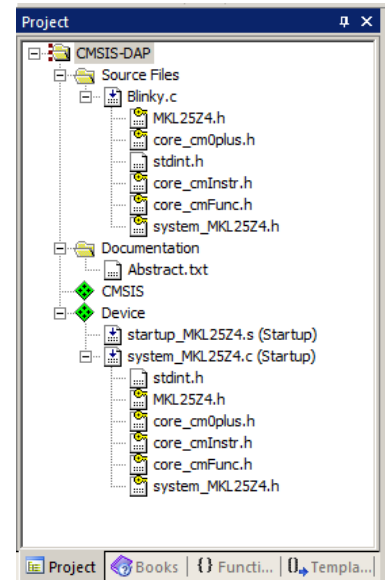
See the **Getting Started: Create Applications with MDK Version 5** available on [www.keil.com/mdk5](http://www.keil.com/mdk5)

The Project window displays the files found in a given project as shown here:

1. If you double click on a file, it will be opened in the source section of  $\mu$ Vision.
2. The Target setting is displayed at the top: It is CMSIS-DAP in this case: This is selected by the Select Target box in the main  $\mu$ Vision menu. You can create your own Target Setting by selecting Project/Manage/Components,...
3. Files are arranged by Groups: such as Source Files, CMSIS and Device etc.

### Adding Files:

1. To add an existing source file to a Group, right click on a Group name (i.e. Source Files) and select: **Add Existing Files to Group 'Source Files'...** Select an existing source file. This file will be added to the Group.
4. To create and add a new file to a Group, right click on a Group name and select: **Add New Item to Group 'Source Files'...** The window below right opens up. Select the type of file you want created. Enter a name for it as shown (I chose DSP) and select Add. A blank file will be created and added to the project. This shown below on the bottom left:



### File Description by Groups:

#### Source Files:

5. **Blinky.c:** The file containing the main() function. You can change the name of this file.

#### Documentation:

6. **Abstract.txt:** a description of the program detailing its attributes.

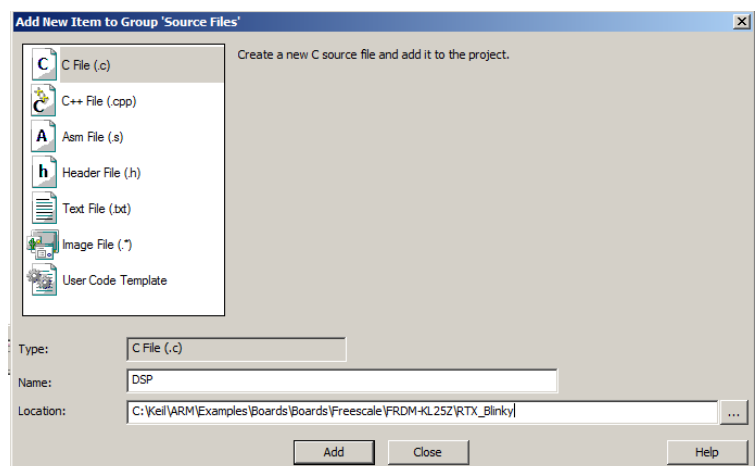
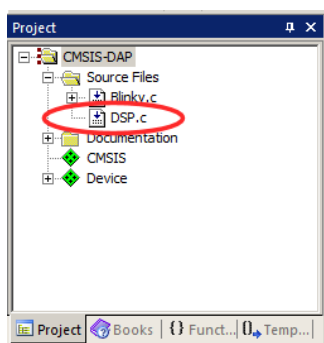
#### CMSIS:

**Empty.** RTX and other files can go here.

**Device:** *These are CMSIS files and are compliant with CMSIS 2.0.*

7. **startup\_MKL25Z4.s:** Creates initial SP and PC and exception vector table plus more.
8. **system\_MKL25Z4.c:** Configures PLL and clock systems.

Header files used by the source files are not described. To view them, just double click on them and they will be opened in the source window.



**TIP:** The icons with the yellow key means this file is read-only. If you need to modify such a file, you will need to find it and modify its permissions. It might be in such a directory: C:\Keil\_v5\ARM\Pack\Keil\Kinetis\_KLxx\_DFP\1.0.0\Device

## 24) RTX\_Blinky Project File System Explanation:

You can use the RTX\_Blinky project as a template for your own projects.


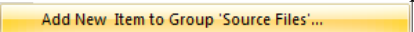
For more information regarding RTX:

See the **Getting Started: Create Applications with MDK Version 5** available on [www.keil.com/mdk5](http://www.keil.com/mdk5)

The Project window displays the files found in a given project as shown here:

9. If you double click on a file, it will be opened in the source section of  $\mu$ Vision.
10. The Target setting is displayed at the top: It is CMSIS-DAP in this case: This is selected by the Select Target box in the main  $\mu$ Vision menu. You can create your own Target Setting by selecting Project/Manage/Components,...
11. Files are arranged by Groups: such as Source Files, CMSIS and Device etc.

### Adding Files:

2. To add an existing source file to a Group, right click on a Group name (i.e. Source Files) and select:  Select an existing source file. This file will be added to the Group.
12. To create and add a new file to a Group, right click on a Group name and select: . The window below right opens up. Select the type of file you want created. Enter a name for it as shown (I chose DSP) and select Add. A blank file will be created and added to the project. This shown below on the bottom left:

### File Description by Groups:

#### Source Files:

13. **Blinky.c**: The file containing the main() function.

#### Documentation:

14. **Abstract.txt**: a description of the program detailing its attributes.

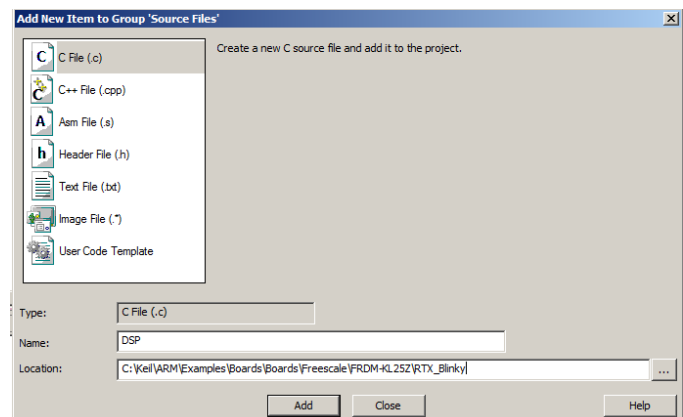
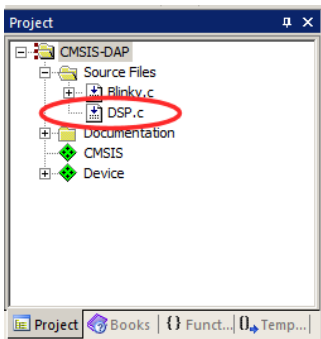
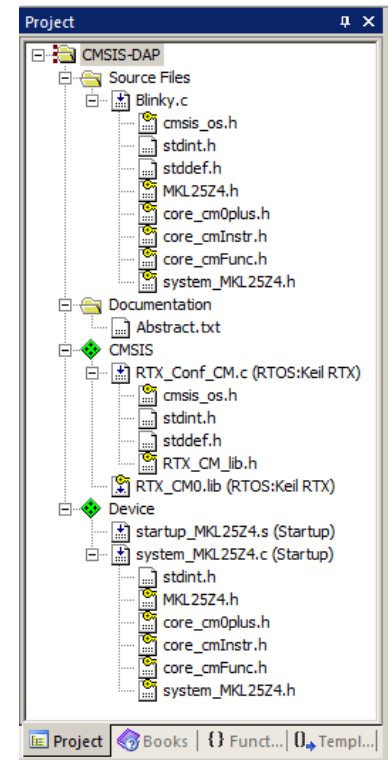
**CMSIS:** *These are CMSIS files and are compliant with CMSIS 2.0.*


15. **RTX\_Conf\_CM.c**: Contains configuration items for RTX.
16. **cmsis\_os.h**: the header file needed for RTX RTOS.

#### Device:

17. **startup\_MKL25Z4.s**: Creates initial SP and PC and exception vector table plus more.
18. **system\_MKL25Z4.c**: Configures PLL and clock systems.

Header files used by the source files are not described here. To view them, just double click on them and they will be opened in the source window.



**TIP:** The icons with the yellow key  means this file is read-only. If you need to modify such a file, you will need to find it and modify its permissions.

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

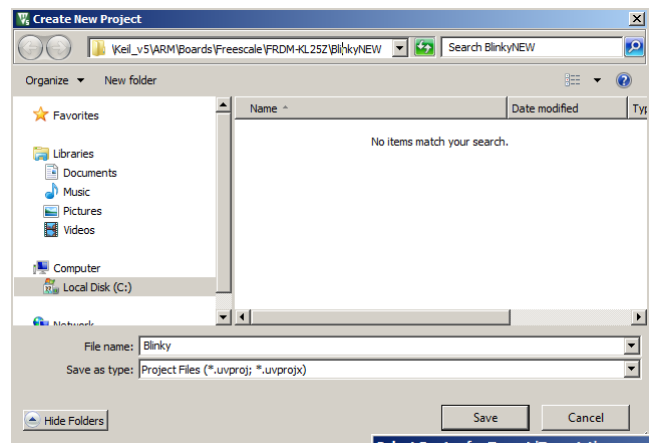
All examples provided by Keil are pre-configured. All you have to do is compile them. You can use them as a starting point for your own projects. However, we will start this example project from the beginning to illustrate how easy this process is. Once you have the new project configured; you can build, 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.

### 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 5.
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:\MDK\Boards\Freescale\FRDM-KL25Z\
3. Right click in this window and select New and create a new folder. I called it BlinkyNEW.
4. Double click on BlinkyNew to open it or highlight it and select Open.
5. In the File name: box, enter Blinky. Click on Save.
6. This creates the project Blinky.uvproj.
7. As soon as you click on Save, the next window opens:




### Select the Device you are using:

1. Expand Freescale Semiconductor and then KLxx Series, then KL2x and then select MKL25Z128xxx4 as shown here:

**TIP:** Chip icons in colour are from the Software Packs. Any grey icons are from MDK 4.7x.

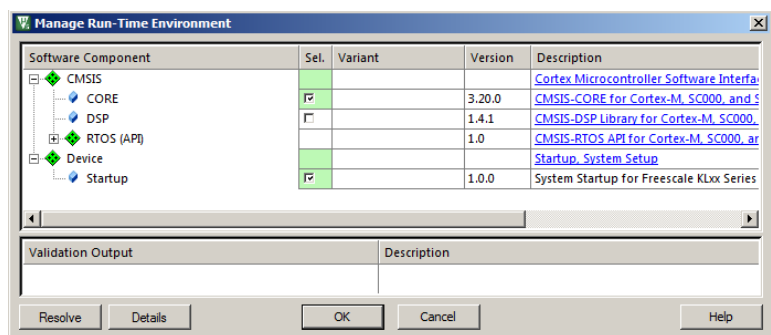
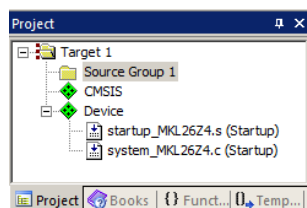
2. Click OK and the Manage Run Time window shown below bottom right opens.

### Select the CMSIS components 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.

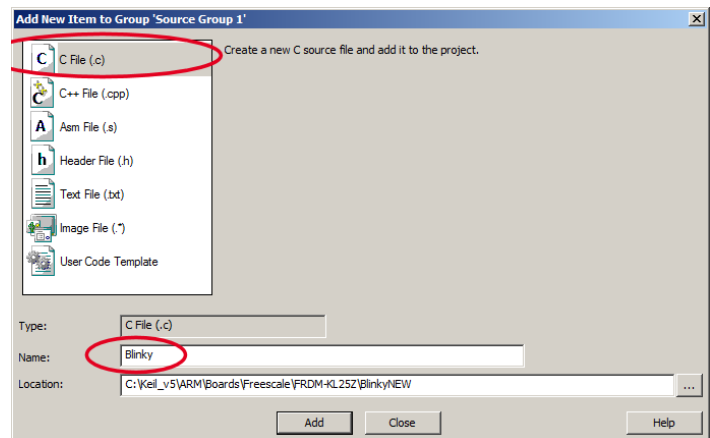






### Create a blank C Source File:

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 New Item to Group 'Source Files'...



### Add Some Code to Blinky.c:

9. In the blank Blinky.c, add the C code below:
10. Click on File/Save All or 
11. Build the files.  There will be no errors or warnings if all was entered correctly.

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



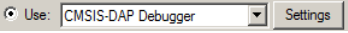
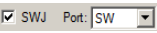

/*-----
  MAIN function
  *-----*/
int main (void) {

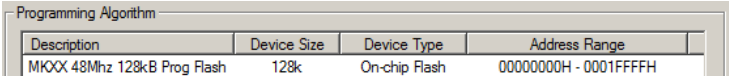
    while(1) {
        counter++;
        if (counter > 0x0F) counter = 0;
    }
}
```

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

Add Existing Files to Group 'Source Files'...


### Configure the Target CMSIS-DAP: Please complete these instructions carefully to prevent unusual problems...

1. Select the Target Options icon . Select the **Target** tab.
2. Enter 8 in Xtal (MHz). This is used for timing calculations. Select Use MicroLIB to optimize for smaller code size.
3. Select the **Output** tab. Click on Select Folder for Objects...: 
4. In the Browse for Folder window that opens: right click and create a new folder called Flash.
5. Double click on Flash to enter this folder and click OK. Compilation files will now be stored in this Flash folder.
6. Click on the **Listings** tab. Click on Select Folder for Objects...: Double click on Flash and click OK to close.
7. Click on the **Debug** tab. Select CMSIS-DAP Debugger in the Use: box: 
8. Select Settings: icon beside Use: CMSIS-DAP.
9. 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.
10. Click on OK **once** to go back to the Target Configuration window.
11. 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: 
12. Click on OK twice to return to the main menu.







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

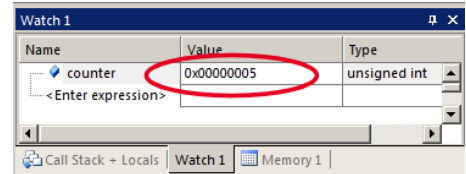
13. Click on File/Save All or 

14. Build the files.  There will be no errors or warnings if all was entered correctly. If there are, please fix them !

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




## Running Your Program:

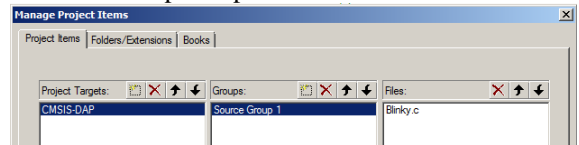
1. Program the KL25 Flash by clicking on the Load icon:  Progress will be indicated in the Output Window.
2. Enter Debug mode by clicking on the Debug icon: .
3. Click on the RUN icon.  Note: you stop the program with the STOP icon. 
4. Right click on counter in Blinky.c and select Add counter to ... and select Watch 1.
5. counter should be updating as shown here:
6. 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.
7. You should now be able to add your own source code to create a meaningful project.




**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.

**Creating a New Target Setting:** This Target Configuration will activate the MTB Instruction Trace feature:

1. Stop the program.  Remove any breakpoints (Ctrl-B, Kill All, Close). Exit Debug mode. 
2. Select Project/Manage/Components, Environment, Books... 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.





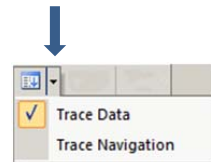
**What this means:** We now have two targets: CMSIS-DAP and CMSIS-DAP MTB. Each one points to its own Target Options configuration. At this point they are the same. We will modify CMSIS-DAP MTB to activate the MTB trace.

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:\MDK\Boards\Freescale\FRDM-KL25Z\Blinky5 and insert DBG\_MTB.ini as shown here:
9. Click OK to exit this window. MTB is now activated with the defaults.



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

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

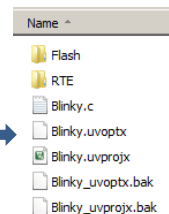


**What this means:** A collection of Target Options is saved in the Target Setting CMSIS-DAP. Another one, with the MTB activated, is saved with CMSIS-DAP MTB. You can modify any Target Options window and it will be saved.

**Cleaning up your Project:** (you only need to do this once: this is not a critical step)

We modified the folder where the output and listings files are stored. This was in Steps 3 through 7 on the preceding page. If you did a Build before this was done, there will be files in your project root directory. Now we want them only in .\Flash.



1. Exit µVision. Otherwise, you can't delete files that it still has open.
2. Open Microsoft Explorer and navigate to:  
C:\MDK\Boards\Freescale\FRDM-KL25Z\BlinkyNEW\.
3. Delete all files and folders except these: (you can delete Flash – a Build will recreate it.)
4. You can also leave any backup or µVision files that identify your computer to retain your settings.
5. Restart µVision. Having all compilation files stored in the .\Flash folder makes it cleaner.



## 26) Creating your own RTX MDK 5 project from scratch:

The MDK Software Packs makes it easy to configure an RTX project. There are two versions of RTX: The first comes with MDK 4.7x and earlier. The second comes with MDK 5.10 and later. This second one is CMSIS-RTOS compliant.

Configuring RTX is easy in MDK 5.10 and later. 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. In Blinky.c, at the top, add this line: `#include "cmsis_os.h"`

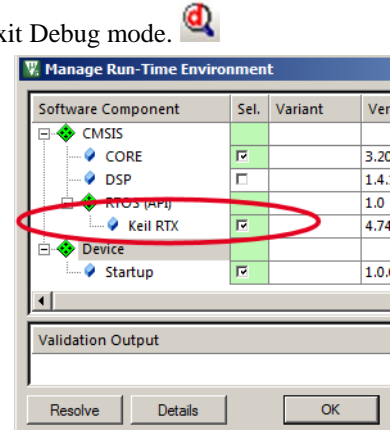
4. Open the Manage Run-Time Environment window: 

5. Expand all the elements as shown here: 


6. Select Keil RTX as shown and click OK.

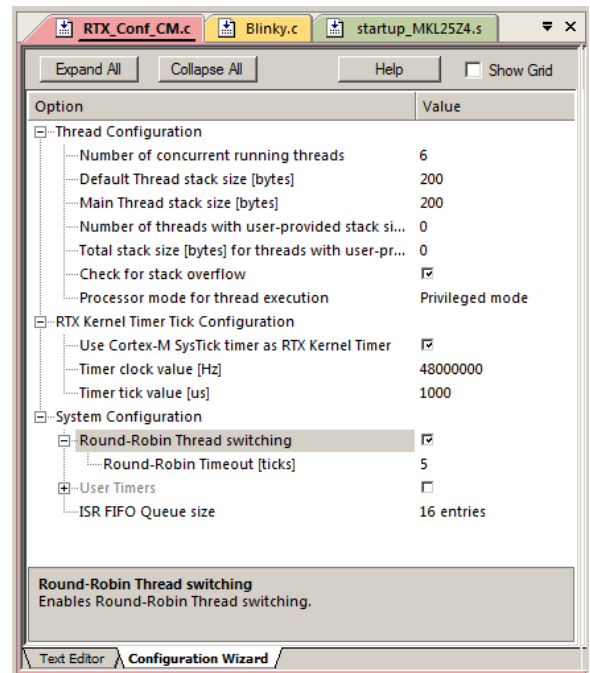
7. Appropriate RTX files will be added to your project. See the Project window.

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 48000000 as shown: (48 MHz)
6. Unselect User Timers. Use defaults for the other settings.



### Build and Run Your RTX Program:

1. Build the files.  Program the Flash: .
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: the main thread is the only one running. 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 DSP5 and RTX\_Blinky examples to use as templates and hints.
3. If you copy Blinky.c from the RTX\_Blinky project, it will blink the LEDs as it has the RTX code incorporated into it.
4. **Getting Started MDK 5:** Obtain this useful book here: [www.keil.com/mdk5/](http://www.keil.com/mdk5/). It has very useful information on implementing RTX.

This completes the exercise of creating your own RTX project from scratch.

System and Thread Viewer							
Property	Value						
System	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: 6, Used: 1					
Threads	ID	Name	Priority	State	Delay	Event Value	Event Mask
	255	os_idle_demon	0	Ready			32%
	1	main	Normal	Running			0%







## 27) Some Interesting Bits & Pieces:

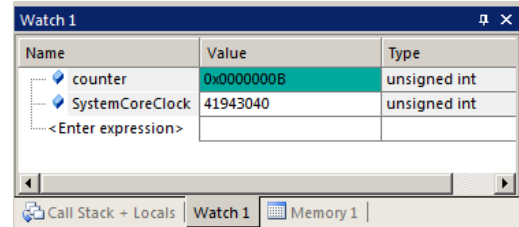
### Processor Clock Speed:

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

**SystemCoreClock:** `system_MKL25Z4.c` contains a global variable `SystemCoreClock` near line 96. 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 lines 78, 83 or 88. Which one is used depends on `#define CLOCK_SETUP 0` near line 58. By setting `CLOCK_SETUP` to 0, 1 or 2 changes the clock frequency to 41.94, 48 or 100 MHz respectively. This is easily seen in lines 74 through 89. This is not actually measuring the CPU speed, rather just indicating which of the choices is selected.

### 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. In `system_MKL25Z4.c`, change the `#define CLOCK_SETUP 0` to 1. This is near line 58.
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.



## 28) 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 ULINK*pro* is needed.

- **JTAG:** Provides access to the CoreSight debugging module located on the Cortex processor. It uses 4 to 5 pins.
  - **SWD:** Serial Wire Debug is a two pin alternative to JTAG and has about the same capabilities except for no Boundary Scan. SWD is referenced as SW in the  $\mu$ Vision Cortex-M Target Driver Setup.
  - The SWJ box must be selected. KL25 processors use SWD exclusively. There is no JTAG on the KL25.
  - **SWV:** Serial Wire Viewer: A trace capability providing display of reads, writes, exceptions, PC Samples and printf. SWV must use SWD because of the TDIO conflict described in **SWO** below.
  - **DAP:** Debug Access Port. 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 and RTOS kernel awareness windows in real-time 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 in your sources. You do not need to configure or activate DAP.  $\mu$ Vision does this automatically when you select the function.
  - **SWO:** Serial Wire Output: SWV frames usually come out this one pin output. It shares the JTAG signal TDIO.
  - **Trace Port:** A 4 bit port that ULINK*pro* uses to output ETM frames and optionally SWV (rather than the 1 bit port SWO pin). ULINK*pro* normally uses the Trace Port.
  - **ETM:** Embedded Trace Macrocell: Lists the executed instructions. Cortex-M4 with a ULINK*pro* provide ETM.
  - **MTB:** Micro Trace Buffer. A portion of the device internal RAM is used for an instruction trace buffer. Only on Cortex-M0+ processors. Kinetis Cortex-M4 processors provide ETM trace instead.
  - **Hardware Breakpoints:** The Kinetis Cortex-M0+ has 2 breakpoints. The Kinetis Cortex-M4 has 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.
  - **WatchPoints:** Both the Freescale Cortex-M0+ and Cortex-M4 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.
-



## 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.

### Serial Wire Viewer displays in various ways: : (Cortex-M0+ does not have SWV. Kinetis Cortex-M4 does)

- PC Samples.
- Data reads and writes. (Cortex-M0/M0+ does has this feature)
- Exception and interrupt events.
- CPU counters.
- Timestamps for these.

### 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. MTB or ETM trace is best for this.
- Communication protocol and timing issues. System timing problems.

### 30) Document Resources:

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

#### 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 processors. A good list of books on ARM processors is found at [www.arm.com/university](http://www.arm.com/university) by selecting “Teaching Resources”. You can also select ARM Related Books but make sure to also select the “Books suited for Academia” tab to see the full selection.
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:

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. Export Processor Expert Projects to  $\mu$ Vision™ [www.keil.com/appnotes/docs/apnt\\_235.asp](http://www.keil.com/appnotes/docs/apnt_235.asp)
21. Using MQX with Keil  $\mu$ Vision Kernel Awareness [www.keil.com/freescale/mqx.asp](http://www.keil.com/freescale/mqx.asp)
22. Keil Videos: [www.keil.com/videos](http://www.keil.com/videos)

#### Keil Tutorials for Freescale Boards:

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

23. KL25Z Freedom (the full version of this document) [www.keil.com/appnotes/docs/apnt\\_232.asp](http://www.keil.com/appnotes/docs/apnt_232.asp)
24. K20D50M Freedom Board [www.keil.com/appnotes/docs/apnt\\_243.asp](http://www.keil.com/appnotes/docs/apnt_243.asp)
25. Kinetis K60N512 Tower [www.keil.com/appnotes/docs/apnt\\_239.asp](http://www.keil.com/appnotes/docs/apnt_239.asp)
26. Kinetis K60D100M Tower [www.keil.com/appnotes/docs/apnt\\_249.asp](http://www.keil.com/appnotes/docs/apnt_249.asp)

### 31) Keil Contact Information:

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

**Keil Sales:** In North and South America: [sales.us@keil.com](mailto:sales.us@keil.com) or 800-348-8051. Outside the US: [sales.intl@keil.com](mailto:sales.intl@keil.com)

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

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

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

**ARM Accredited Engineer Program:** [www.arm.com/aae](http://www.arm.com/aae)

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