

# STMicroelectronics: Cortex™-M4 Training STM32F407

## Discovery evaluation board using ARM® Keil™ MDK Toolkit



featuring Serial Wire Viewer

Spring 2013 Version 2.0

by Robert Boys, [bob.boys@arm.com](mailto:bob.boys@arm.com)

The latest version of this document is here:  
For a CAN lab on the STM32F4 Discovery:

[www.keil.com/appnotes/docs/apnt\\_230.asp](http://www.keil.com/appnotes/docs/apnt_230.asp)  
[www.keil.com/appnotes/docs/apnt\\_236.asp](http://www.keil.com/appnotes/docs/apnt_236.asp)

### Introduction:

The purpose of this lab is to introduce you to the STMicroelectronics Cortex™-M4 processor using the ARM® Keil™ MDK toolkit featuring the IDE  $\mu$ Vision®. We will use the Serial Wire Viewer (SWV) and the on-board ST-Link V2 Debug Adapter. At the end of this tutorial, you will be able to confidently work with these processors and Keil MDK. See [www.keil.com/st](http://www.keil.com/st).

Keil MDK supports and has examples for most ST ARM processors. Check the Keil Device Database® on [www.keil.com/dd](http://www.keil.com/dd) for the complete list which is also included in MDK: in  $\mu$ Vision, select Project/Select Device for target...

**Linux:** For ST processors running Linux, Android and bare metal are supported by ARM DS-5™. [www.arm.com/ds5](http://www.arm.com/ds5).

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

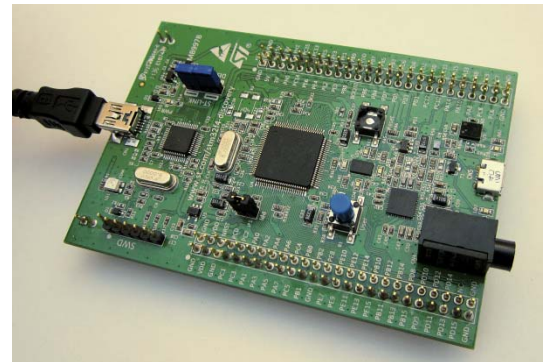
**RTX RTOS:** All variants of MDK contain the full version of RTX with source code. See [www.keil.com/rl-arm/kernel.asp](http://www.keil.com/rl-arm/kernel.asp).

### Why Use Keil MDK ? MDK provides these features particularly suited for Cortex-M users:

1.  $\mu$ Vision IDE with Integrated Debugger, Flash programmer and the ARM® Compiler toolchain. MDK is a turn-key product.
2. A full feature Keil RTOS called RTX is included with MDK. RTX comes with a BSD type license. Source code is provided.
3. Serial Wire Viewer and ETM trace capability is included.
4. RTX Kernel Awareness window. It is updated in real-time.
5. Keil Technical Support is included for one year and is easily renewable. This helps you get your project completed faster.

### This document details these features:

1. Serial Wire Viewer (SWV) and ETM trace. Real-time tracing updated while the program is running.
2. Real-time Read and Write to memory locations for Watch, Memory and RTX Tasks windows. These are non-intrusive to your program. No CPU cycles are stolen. No instrumentation code is added to your source files.
3. Six Hardware Breakpoints (can be set/unset on-the-fly) and four Watchpoints (also known as Access Breaks).
4. RTX Viewer: a kernel awareness program for the Keil RTX RTOS that updates while your program is running.
5. A DSP example program using ARM CMSIS-DSP libraries. [www.arm.com/cmsis](http://www.arm.com/cmsis)



### Serial Wire Viewer (SWV):

**Serial Wire Viewer** (SWV) displays PC Samples, Exceptions (including interrupts), data reads and writes, ITM (printf), CPU counters and a timestamp. This information comes from the ARM CoreSight™ debug module integrated into STM32 CPU. SWV does not steal any CPU cycles and is completely non-intrusive. (except for the ITM Debug printf Viewer).

CoreSight displays memory contents and variable values in real-time and these can be modified on-the-fly.

### Embedded Trace Macrocell (ETM):

ETM records and displays all instructions that were executed. This is very useful for debugging program flow problems such as “going into the weeds” and “how did I get here?”. Keil  $\mu$ Vision uses ETM to provide Code Coverage, Performance Analysis and code execution times. ETM requires a special debug adapter such as a ULINKpro. The Discovery series do not have the ETM connector even though the processor has ETM. Most other ST and Keil boards do have this connector.

### Discovery Board Debug Adapter Connections:

The STM32F407 Discovery board lacks the standard ARM debugger connections. This means it is not easy to connect a ULINK2, ULINKpro or J-Link to these boards. In order to use features like ETM trace, it is easier to obtain a board such as the Keil MCBSTM32 series or a STM32xxx-EVAL board. Versions are available with Cortex-M3 and Cortex-M4 processors. Keil MDK has examples and labs for these boards. This document uses only the on-board ST-LINK. See [www.keil.com/st](http://www.keil.com/st).

## Index:

1. Keil Evaluation Software:	3
2. Keil Software Installation:	3
3. CoreSight Definitions:	3
4. CMSIS: Cortex Microcontroller Software Interface Standard	3
5. Configuring the ST-Link V2:	4
6. <i>Blinky</i> example using the STM32F4 Discovery board:	6
7. Hardware Breakpoints:	6
8. Call Stack & Locals window:	7
9. Watch and Memory windows and how to use them:	8
10. How to view Local Variables in Watch and Memory windows:	9
11. View Variables Graphically with the Logic Analyzer (LA):	10
12. Watchpoints: <i>Conditional Breakpoints</i>	11
13. RTX_Blinky example: Keil RTX RTOS:	12
14. RTX Kernel Awareness using RTX Viewer:	13
15. Logic Analyzer: View variables real-time in a graphical format:	14
16. ITM (Instruction Trace Macrocell):	15
17. Serial Wire Viewer (SWV) and how to use it:	16
1) Data Reads and Writes	16
2) Exceptions and Interrupts	17
3) PC Samples (program counter samples)	18
18. Serial Wire Viewer (SWV) Configuration:	19
19. DSP Sine Example using ARM CMSIS-DSP Libraries	20
20. Creating your own project from scratch:	24
21. ETM Trace and its benefits: <i>for reference</i>	26
22. Serial Wire Viewer summary:	32
23. Useful Documents:	32
24. Keil Products and contact information:	33

## Notes on using this document:

1. The latest version of this document and the necessary example source files are available here:  
[www.keil.com/appnotes/docs/apnt\\_230.asp](http://www.keil.com/appnotes/docs/apnt_230.asp)
2. MDK 4.70 was used in the exercises in this document.
3. Configuring the ST-Link V2 debug adapter starts on page 4.
4. The on-board ST-Link V2 is used by default in this document. All you need install is the USB driver.
5. The original ST-Link (usually called V1) is supported by  $\mu$ Vision but Serial Wire Viewer is not.
6. The first exercise starts on page 6. You can go directly there if using a ST-Link. If you are using a ULINK2, ULINK-ME, ULINK*pro* or J-Link, you will need to configure it appropriately as described on the following pages.
7. The ST-Link V2 interfaces very well with Keil  $\mu$ Vision and its performance is quite good including SWV.

## 1) Keil Evaluation Software:

### Example Programs:

MDK contains many useful ready-to-run examples for boards using ST processors. See C:\Keil\ARM\Boards\ST and \Keil. Many examples are provided to also run in the Keil Simulator. No hardware is needed in these cases.

MDK 4.70 contains two example programs: Blinky and RTX\_Blinky. This Blinky is used in this lab. RTX\_Blinky must be upgraded. The new one blinks all four leds on the Discovery board. A new example, DSP must also be added.

These files can be downloaded from [www.keil.com/appnotes/docs/apnt\\_230.asp](http://www.keil.com/appnotes/docs/apnt_230.asp). The latest version of this document is also available at this location. Put these two directories in file C:\Keil\ARM\Boards\ST\STM32F4-Discovery\ to create \DSP and \RTX\_Blinky directories respectively.

Keil has several labs for various STM32 processors including one using CAN. See [www.keil.com/st](http://www.keil.com/st) for details.

**The directory \RL** consists of middleware examples. Such middleware is a component of MDK Professional. To run these examples a full license is needed. Please contact Keil sales for a temporary license if you want to evaluate Keil middleware and for the list of supported processors.

STMicroelectronics has an entire suite of examples for various STM32 processors using Keil MDK. See [www.st.com](http://www.st.com).

**Keil Sales:** In USA and Canada: [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)

## 2) Keil Software Installation:

This document was written using Keil MDK 4.70 or later which contains  $\mu$ Vision 4. The evaluation copy of MDK (MDK-Lite) is available free on the Keil website. Do not confuse  $\mu$ Vision 4 with MDK 4.0. The number “4” is a coincidence. Nearly all example programs can be compiled within the 32K limit of MDK-Lite: the free evaluation version.

To obtain a copy of MDK go to [www.keil.com/arm](http://www.keil.com/arm) and select the “Download” icon located on the right side.

You can use the evaluation version of MDK-Lite for this lab. A debug adapter such as a ULINK2 is not needed.

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

- **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 Boundary Scan is not possible. SWD is referenced as SW in the  $\mu$ Vision Cortex-M Target Driver Setup. See page 4, 2<sup>nd</sup> picture. The SWJ box must be selected in ULINK2/ME or ULINK $pro$ . SWV must use SWD because of the TDIO conflict described in **SWO** below.
- **SWV:** Serial Wire Viewer: A trace capability providing display of reads, writes, exceptions, PC Samples and printf.  
**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 collect ETM frames and optionally SWV (rather than SWO pin).
- **ETM:** Embedded Trace Macrocell: Provides all the program counter values. Only the ULINK $pro$  provides ETM.

## 4) CMSIS: Cortex Microcontroller Software Interface Standard

ARM CMSIS-DSP libraries are offered for all Cortex-M3 and Cortex-M4 processors.

CMSIS-RTOS provides standard APIs for RTOSs. RTX is a free RTOS available from ARM as part of CMSIS Version 3.0.

STMicroelectronics example software is CMSIS hardware abstraction layer compliant.

See [www.arm.com/cmsis](http://www.arm.com/cmsis) and [forums.arm.com](http://forums.arm.com) for more information. [www.keil.com/st](http://www.keil.com/st)

## 5) Configuring the ST-Link V2:

It is easy to select a USB debugging adapter in  $\mu$ Vision. You must configure the connection to both the target and to Flash programming in two separate windows as described below. They are each selected using the Debug and Utilities tabs.

**Using other Debug Adapters:** This document will use the on-board ST-Link. You can use a ULINK2 or a ULINK $pro$  with suitable adjustments. You would need a suitable adapter to connect a different adapter to the SWD connector on the Discovery board. Some step(s) to turn off the on-board ST-Link adapter might also be necessary to avoid conflicts. It is reported that shorting solder bridge SB10 will hold the ST-Link processor in RESET allowing external adapter operation.

If your debugging sessions are unreliable, please check for additional conflicts or loading on the SWD pins. The SWD connector provides the ability to use the Discovery board as a debug adapter on another board. Its main purpose is not to connect an external tool such as a Keil ULINK2. Some adaptation is required but not difficult to do.

It is possible to use a Segger J-Link with  $\mu$ Vision. Serial Wire Viewer is supported.

The ST-Link is selected as the default debug adapter for the Keil examples for the Discovery board.



Serial Wire Viewer (SWV) is completely supported by ST-LINK Version 2. Firmware V2.16.S0.

### Step 1) Installing the ST-Link USB Drivers: (you need to do this the first time only)

1. Do not have the Discovery board USB port connected to your PC at this time.
2. The USB drivers must be installed manually by executing ST-Link\_V2\_USBdriver.exe. This file is found in C:\Keil\ARM\STLink\USBdriver. Find this file and double click on it. The drivers will install.
3. Plug in the Discovery board to USB CN1. The USB drivers will now finish installing in the normal fashion.

**Super TIP:** The ST-Link V2 firmware update files are located here: C:\Keil\ARM\STLink. This updates the Discovery ST-Link firmware by executing ST-LinkUpgrade.exe. Find this file and double click on it. It will check and report the current firmware version. It is important you are using firmware V2.J16.S0 or later for proper SWV operation.

### Step 2) Select the debug connection to the target: *The following steps are already done by default in the three example programs. These instructions are provided for reference.*

1. Connect your PC to the Discovery board with a USB cable. Start  $\mu$ Vision. It must be in Edit mode (as it is when first started – the alternative to Debug mode) and you have selected a valid project. Blinky will do fine.
2. Select Target Options  or ALT-F7 and select the Debug tab. In the drop-down menu box select ST-Link Debugger as shown here: 

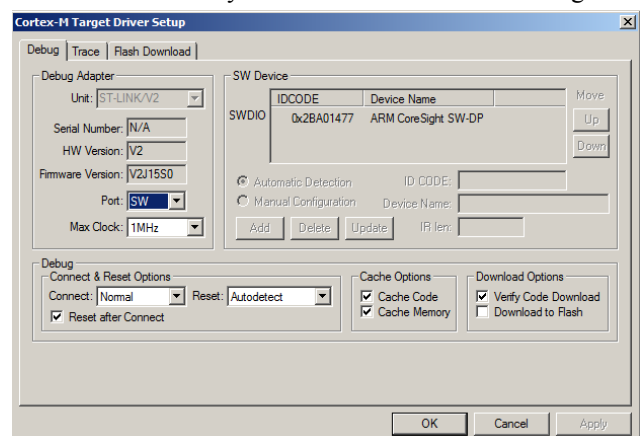
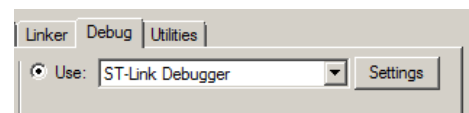
**TIP:** Do NOT select ST-Link (Deprecated Version).

3. Select Settings and the next window below opens up. This is the control panel for the ULINKs and ST-Link. (they are the same).
4. In **Port:** select SW. JTAG is not a valid option for ST-Link and this board. SW is also known as SWD.
5. In the SW Device area: ARM CoreSight SW-DP **MUST** be displayed. This confirms you are connected to the target processor. If there is an error displayed or it is blank this **must** be fixed before you can continue. Check the target power supply. Cycle the power to the board.

**TIP:** To refresh this screen select Port: and change it or click OK once to leave and then click on Settings again.

**TIP:** You can do everything with SW (SWD) as you can with JTAG except for boundary scan.

**Next: configure the Keil Flash programming tool:**



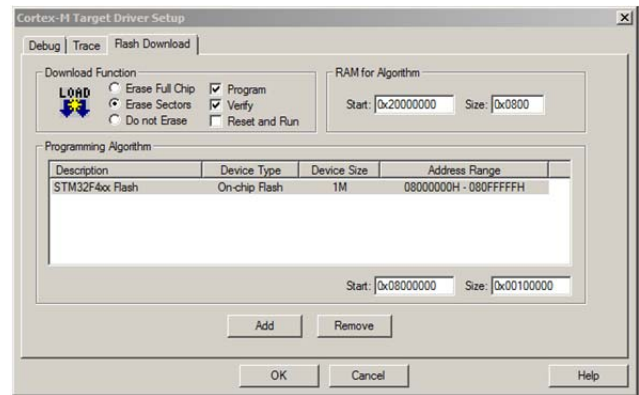
### Step 3) Configure the Keil Flash Programmer:

- Click on OK once and select the Utilities tab.
- Select the ULINK similar to Step 2 above.
- Click Settings to select the programming algorithm if it is not visible or is the wrong one.
- Select STM32F4xx Flash as shown here or the one for your processor:
- Click on OK once.

**TIP:** To program the Flash every time you enter Debug mode, check Update target before Debugging.

- Click on OK to return to the  $\mu$ Vision main screen.
- Select File/Save All.
- You have successfully connected to the STM32 target processor and configured the  $\mu$ Vision Flash programmer.

**TIP:** The Trace tab is where you configure the Serial Wire Viewer (SWV). You will learn to do this later.



### COM led LD1 indication:

LED is blinking RED: the first USB enumeration with the PC is taking place.


LED is RED: communication between the PC and ST-LINK/V2 is established (end of enumeration).  $\mu$ Vision is not connected to ST-Link (i.e. in Debug mode).

LED is GREEN:  $\mu$ Vision is connected in Debug mode and the last communication was successful.

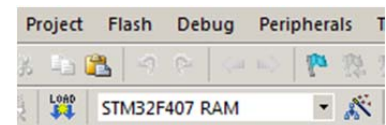
LED is blinking GREEN/RED: data is actively being exchanged between the target and  $\mu$ Vision.

**No Led: ST-LINK/V2 communication with the target or  $\mu$ Vision has failed. Cycle the board power to restart.**


### Running programs in the internal STM32 RAM:

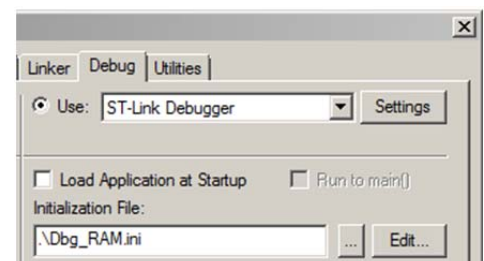
It is possible to run your program in the processor RAM rather than Flash. In this case, the Flash programming tool is not used nor is the Load icon. After successfully compiling the source files, click on Debug icon . An .ini file configures the processor and loads your executable into RAM.

The Discovery Blinky project has a RAM setting. Select STM32F407 RAM as shown here if you want to try this mode.



### Loading and Running your program into RAM:







- Select STM32F407 RAM as shown above.
- Select Target Options  or ALT-F7 and select the Debug tab.
- The ini file is located in the Initialization File: box as shown here:
- Click on Edit... to view its contents.
- Click on the Target tab. Note the RAM at 0x2000\_0000 split between the R/O and R/W memory areas.
- Click on OK to return to the main  $\mu$ Vision window.
- Return to the STM32F407 Flash setting.





## 6) Blinky example program using the ST Discovery board:

We will connect a Keil MDK development system using real target hardware using the built-in ST-Link debug adapter.

1. Start  $\mu$ Vision by clicking on its desktop icon.  Connect your PC to the board with a USB cable to CN1.
2. Select Project/Open Project. Open the file C:\Keil\ARM\Boards\ST\STM32F4-Discovery\Blinky\Blinky.uvproj
3. By default, the ST-Link is selected. If this is the first time you have run  $\mu$ Vision and the Discovery board, you will need to install the USB drivers. See the configuration instructions on page 4 in this case.
4. Compile the source files by clicking on the Rebuild icon.  You can also use the Build icon beside it.
5. Program the STM32 flash by clicking on the Load icon:  Progress will be indicated in the Output Window.
6. Enter Debug mode by clicking on the Debug icon.  Select OK if the Evaluation Mode box appears.  
**Note:** You only need to use the Load icon to download to FLASH and not for RAM operation if it is chosen.
7. Click on the RUN icon.  Note: you stop the program with the STOP icon. 





**The LEDs on the STM32F4 Discovery board will now blink in succession.  
Press the USER button and they will all come on.**

**Now you know how to compile a program, program it into the STM32 processor Flash, run it and stop it !**

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

## 7) Hardware Breakpoints:

The STM32F4 has six 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 block in the left margin on a line in main() in the while loop. Between around lines 80 through 91 will suffice.
2. A red circle will appear and the program will stop.
3. Note the breakpoint is displayed in both the disassembly and source windows as shown below:
4. You can set a breakpoint in either the Disassembly or Source windows as long as there is a gray rectangle indicating the existence of an assembly instruction at that point.
5. Every time you click on the RUN icon  the program will run until the breakpoint is again encountered.
6. You can also click on Single Step (Step In) , Step Over  and Step Out .

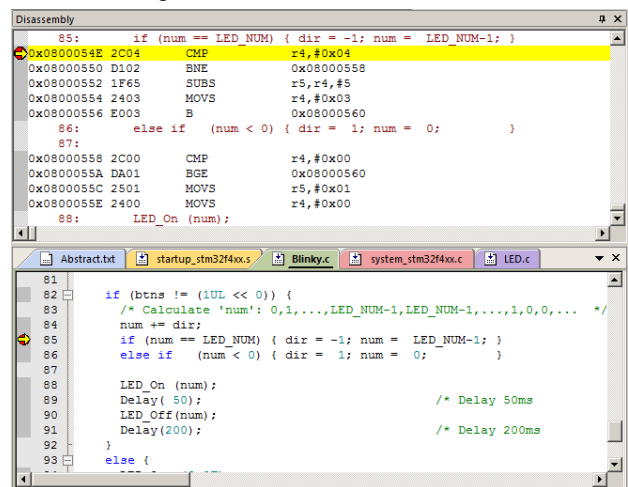
**TIP:** If single step (Step In) doesn't work, click on the Disassembly window to bring it into focus. If needed, click on a disassembly line. This tells  $\mu$ Vision you want to single step at the assembly level rather than at the C source level.

**TIP:** A hardware breakpoint does not execute the instruction it is set to. ARM CoreSight breakpoints are no-skid. These are rather important features.

7. **Remove all the breakpoints when you are done for the next exercise by clicking on them again.**

**TIP:** You can delete the breakpoints by clicking on them or selecting Debug/Breakpoints (or Ctrl-B) and selecting Kill All. Click on Close to return.

**TIP:** You can view the breakpoints set by selecting Debug/Breakpoints or Ctrl-B.



## 8) Call Stack + Locals Window:

### Local Variables:

The Call Stack and Local 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 belonging to 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.

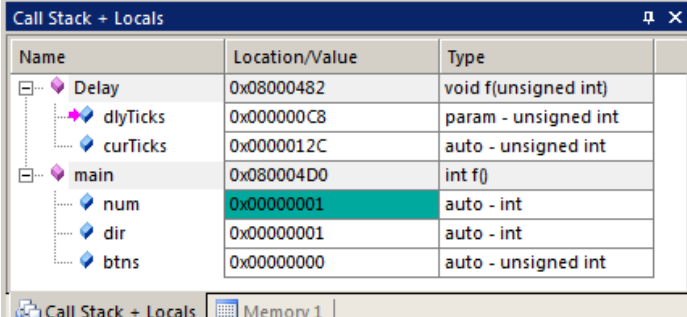
1. Run and Stop Blinky. Click on the Call Stack + Locals tab.
2. Shown is the Call Stack + Locals window.

The contents of the local variables are displayed as well as names of active functions. Each function name will be displayed as it is called from the function before it or from an interrupt or exception.




When a function exits, it is removed from the list.

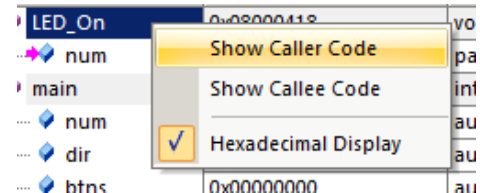
The first called function is at the bottom of this table.

This table is active only when the program is stopped.



Name	Location/Value	Type
Delay	0x08000482	void f(unsigned int)
dlyTicks	0x000000C8	param - unsigned int
curTicks	0x0000012C	auto - unsigned int
main	0x080004D0	int f()
num	0x00000001	auto - int
dir	0x00000001	auto - int
btns	0x00000000	auto - unsigned int

3. Click on the Step In icon or F11: 
4. Note the function different functions displayed as you step through them. If you get trapped in the Delay function, use Step Out  or Ctrl-F11 to exit it faster.
5. Click numerous times on Step In and see other functions.
6. Right click on a function name and try the Show Callee Code and Show Caller Code options as shown here:
7. Click on the StepOut icon  to exit all functions to return to main().



**TIP:** If single step (Step In) doesn't work, click on the Disassembly window to bring it into focus. If needed, click on a disassembly line to step through assembly instructions. If a source window is in focus, you will step through the source lines instead..

**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 or static variables updated in real-time while the program is running. No additions or changes to your code are required. Update while the program is running is not possible with local variables because they are usually stored in a CPU register. They must be converted to global or static variables so they always remain in scope.

If you have a ULINKpro and ETM trace, you can see a record of all the instructions executed. The Disassembly and Source windows show your code in the order it was written. The ETM trace shows it in the order it was executed. ETM provides Code Coverage, Performance Analysis and Execution Profiling.

Changing a local variable to a static or global normally means it is moved from a CPU register to RAM. CoreSight can view RAM but not CPU registers when the program is running.

### 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 what return data is stored on the stack.

**TIP:** You can modify a variable value when the program is stopped.

**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.








Selecting Debug/Kill All Breakpoints deletes Breakpoints but not Watchpoints.

## 9) 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 through the ARM CoreSight debugging technology that is part of Cortex-M processors. It is also possible to “put” or insert values into these memory locations in real-time. It is possible to “drag and drop” variable names into windows or enter them manually.

### Watch window:

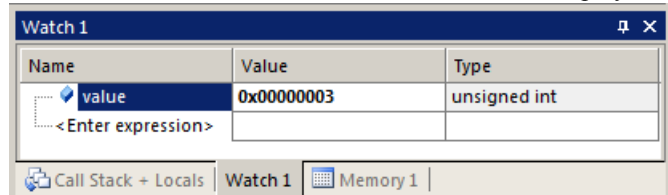
**Add a global variable:** Recall the Watch and Memory windows can’t see local variables unless stopped in their function.

1. Stop the processor  and exit debug mode. 
2. In Blinky.c declare a global variable (I called it value) near line 21 like this: **unsigned int value = 0;**
3. Add the statements `value++;` and `if (value > 0x10) value = 0;` as shown here near line 93:
4. Click on Rebuild.  Click on Load  to program the Flash.
5. Enter Debug mode.  Click on RUN . Recall you can set Watch and Memory windows while the program is running.
6. Open the Watch 1 window by clicking on the Watch 1 tab as shown or select View/Watch Windows/Watch 1.
7. In Blinky.c, right click on the variable value and select Add value to ... and select Watch 1. **value** will be displayed as shown here: 
8. **value** will increment until 0x10 in real-time.

**TIP:** You can also block **value**, click and hold and drag it into Watch 1 or a Memory window.

**TIP:** Make sure View/Periodic Window Update is selected.

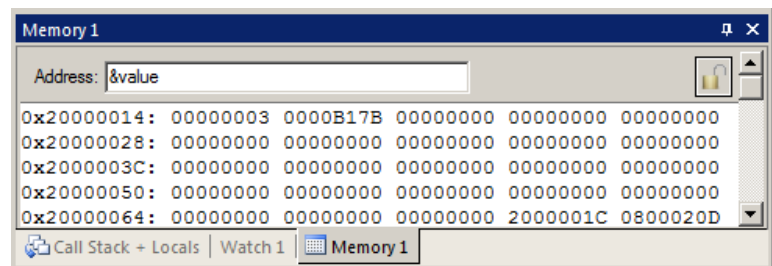
9. You can also enter a variable manually by double-clicking under Name or pressing F2 and using copy and paste or typing the variable.



**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. Drag ‘n Drop **value** into the Memory 1 window or enter it manually. Select View/Memory Windows if necessary.
2. Note the value of **value** is displaying its address in Memory 1 as if it is a pointer. This is useful to see what address a pointer is pointing to but this not what we want to see at this time.
3. Add an ampersand “&” in front of the variable name and press Enter. The physical address is shown (0x2000\_0014).
4. Right click in the memory window and select Unsigned/Int.
5. The data contents of **value** is now displayed as a 32 bit value.
6. Both the Watch and Memory windows are updated in real-time.
7. You can modify value in the Memory window with a right-click with the mouse cursor over the data field and select Modify Memory.



**TIP:** No CPU cycles are used to perform these operations. See the next page “How It Works” for an explanation on how DAP functions.

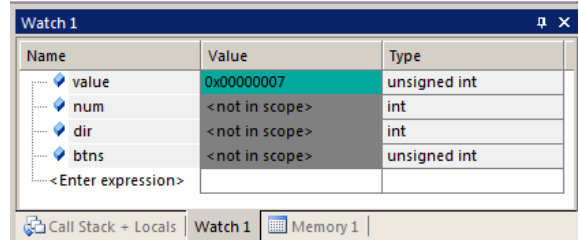
**TIP:** To view variables and their location use the Symbol window. Select View/Symbol Window while in Debug mode.

Serial Wire Viewer does not need to be configured in order for the Memory and Watch windows to operate as shown. This mechanism uses a different feature of CoreSight than SWV. These Read and Write accesses are handled by the Serial Wire Debug (SWD) or JTAG connection via the CoreSight Debug Access Port (DAP), which provides on-the-fly memory accesses.





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

1. Make sure Blinky.c is running. We will use the local variables from main() num, dir and btns.
2. Locate where the three local variables are declared in Blinky.c near line 67, at the start of the main function.
3. Drag and Drop **each variable** into Watch 1 window. Note it says < not in scope > because  $\mu$ Vision cannot access the CPU registers while running which is where value is probably located.
4. Set a breakpoint in the Blinky.c while loop. The problem will stop the program and the current variable values will appear.
5. Remove this breakpoint.
6. Set a breakpoint at `if (btns != (1UL << 0)) {` near line 83.
7. Start the program, hold down the User button and the program will stop. A btns value of 1 will display. Without User pressed, a 0 will be displayed if you click on Run again.



**TIP:** Remember: you can set and unset hardware breakpoints on-the-fly in the Cortex-M4 while the program is running !

8.  $\mu$ Vision is unable to determine the value of these three variables when the program is running because they exist only when main is running. They disappear in functions and handlers outside of main. They are 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.
9. **Remove the breakpoint** and make sure the program is not running . Exit Debug mode. .


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


All you need to do is to make the local variables num, dir and btns global where it is declared in Blinky.c !





1. Move the declarations for num, dir and btns out of main() and to the top of Blinky.c to make them global variables:

```
int main (void) {  
    int32_t num = -1;  
    int32_t dir = 1;  
    uint32_t btns = 0;  
}
```

**TIP:** You can edit files in edit or debug mode. However, you can compile them only in edit mode.

2. Compile the source files by clicking on the Rebuild icon. They will compile with no errors or warnings.
3. To program the Flash, click on the Load icon. . A progress bar will be displayed at the bottom left.

**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.

4. Enter Debug mode. 
5. Click on RUN. 
6. Now the three variables are updated in real-time. Press and release the User button and btns will update to 0 or 1. This is ARM CoreSight technology working.
7. You can read (and write) global, static variables and structures. Anything that stays around in a variable from function to function. This includes reads and writes to peripherals.
8. Stop the CPU and exit debug mode for the next step.  and 

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

### How It Works:

$\mu$ Vision uses ARM CoreSight technology to read or write memory locations without stealing any CPU cycles. This is nearly always non-intrusive and does not impact the program execution timings. Remember the Cortex-M4 is a Harvard architecture. This means it has separate instruction and data buses. While the CPU is fetching instructions at full speed, there is plenty of time for the CoreSight debug module to read or write values without stealing any CPU cycles.




This can be slightly intrusive in the unlikely event the CPU and  $\mu$ Vision reads or writes to the same memory location at exactly the same time. Then the CPU will be stalled for one clock cycle. In practice, this cycle stealing never happens.

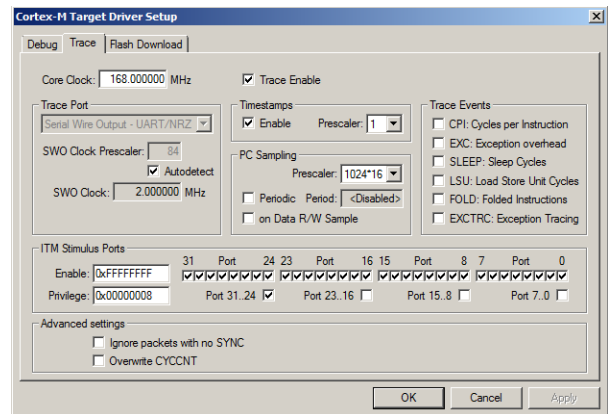
## 11) View Variables Graphically with the Logic Analyzer (LA):

We will display the global variable value you created earlier in the Logic Analyzer. No code stubs in the user code will be used. This uses the Serial Wire Viewer and therefore does not steal CPU cycles.


1. Stop the processor  and exit Debug mode. 

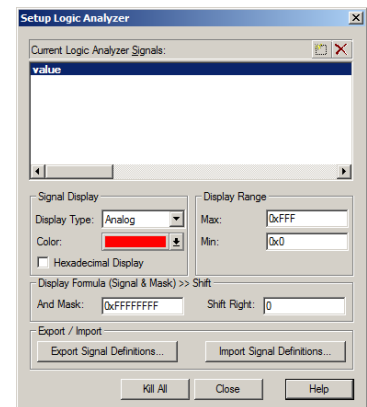
### Configure Serial Wire Viewer (SWV):

2. Select Target Options  or ALT-F7 and select the Debug tab. Select Settings: on the right side of this window. Confirm SW is selected. SW selection is mandatory for SWV. ST-Link uses only SW. Select the Trace tab.
3. In the Trace tab, select Trace Enable. Unselect Periodic and EXCTRC. Set Core Clock: to 168 MHz. Everything else is set as shown here: 
4. Click OK once to return to the Debug tab.
5. In the Initialization File: box: select STM32\_SWO.ini from C:\Keil\ARM\Boards\Keil\MCBSTM32C\Blinky\_Ulp: You can use the Browse icon to locate and enter it. This file configures the STM32 SWV module and default is for SWV. You can also move this file and select it locally.
6. Click OK return to the main menu. Enter debug mode. 




### Configure Logic Analyzer:

1. Open View/Analysis Windows and select Logic Analyzer or select the LA window on the toolbar. 
2. Click on the Blinky.c tab. Right click on **value** and select Add value to... and then select Logic Analyzer. You can also Drag and Drop or enter it manually.
3. Click on the Select box and the LA Setup window appears:
4. With value selected, set Display Range Max: to 0x15 as shown here:
5. Click on Close.




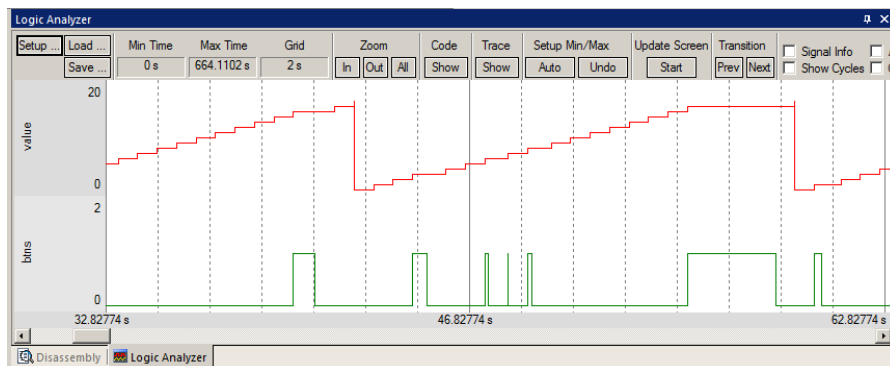
**Run Program: Note:** The LA can be configured while the program is running.

- 1) Click on Run.  Click on Zoom Out until Grid is about 1 second.
- 2) The variable value will increment to 0x10 (decimal 16) and then is set to 0.

**TIP:** If you do not see a waveform, exit and re-enter Debug mode to refresh the LA. You might also have to repower the Discovery board. Confirm the Core Clock: value is correct.

**TIP:** You can show up to 4 variables in the Logic Analyzer. These variables must be global, static or raw addresses such as \*((unsigned long \*)0x20000000).

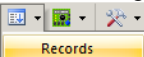
- 3) Enter the static variable btns into the LA and set the Display Range Max: to 0x2. Click on RUN and press the User button and see the voltages below:
- 4) Select Signal Info, Show Cycles, Amplitude and Cursor to see the effects they have.
- 5) Stop the CPU. 



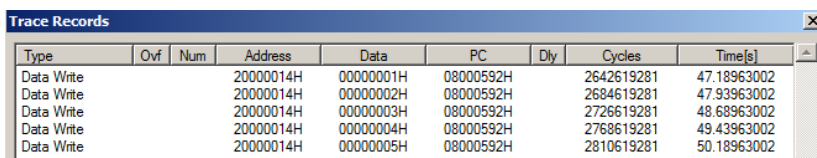
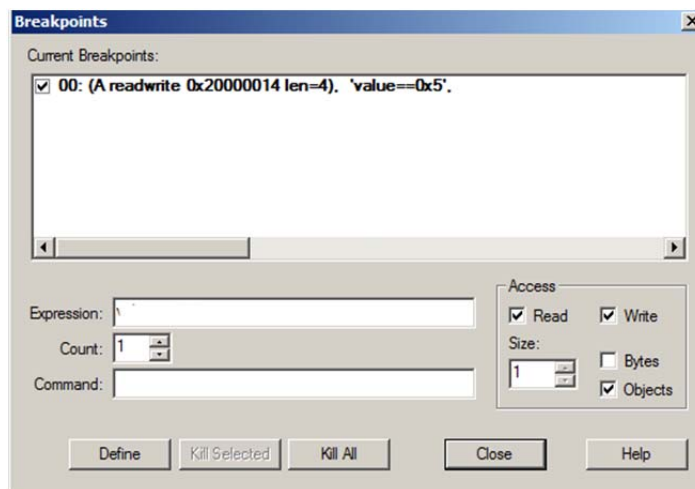
## 12) Watchpoints: Conditional Breakpoints

Recall STM32 processors have 6 hardware breakpoints. These breakpoints can be set on-the-fly without stopping the CPU. The STM32 also have four Watchpoints. Watchpoints can be thought of as conditional breakpoints. The Logic Analyzer uses the same comparators as Watchpoints in its operations and they must be shared. This means in  $\mu$ Vision you must have two variables free in the Logic Analyzer to use Watchpoints. Watchpoints are also referred to as Access Breakpoints.

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 `value` you created in Blinky.c to explore Watchpoints.
3. The SWV Trace does not need to be configured for Watchpoints. However, we will use it in this exercise.
4. The variable `value` should be still entered in the Logic Analyzer from the last exercise on the previous page.
5. Select Debug in the main  $\mu$ Vision window and select Breakpoints or press Ctrl-B.
6. In the Expression box enter: `"value == 0x5"` without the quotes. Select both the Read and Write Access.
7. Click on Define and it will be accepted as shown here: Click on Close.
8. Enter the `value` to the Watch 1 window if it is not already listed.
9. Open Debug/Debug Settings and select the trace tab. Check "on Data R/W sample" and uncheck EXTRC.
10. Click on OK twice. Open the Trace Records

window. 

11. Click on RUN
12. You will see `value` change in the Logic Analyzer as well as in the Watch window.
13. When `value` equals 0x5, the Watchpoint will stop the program.
14. Note the data writes in the Trace Records window shown below. 0x5 is in the last Data column. Plus the address the data written to and the PC of the write instruction. This is with the ST-Link. A ULINK2 will show the same window. A ULINK<sub>pro</sub> or a J-Link (black case) will show a different display.
15. There are other types of expressions you can enter and are detailed in the Help button in the Breakpoints window. Not all are currently implemented in  $\mu$ Vision.
16. To repeat this exercise, click on RUN.
17. When you are finished, stop the program, click on Debug and select Breakpoints (or Ctrl-B) and Kill the Watchpoint.
18. Leave Debug mode.



Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Data Write			20000014H	00000001H	08000592H		2642619281	47.18963002
Data Write			20000014H	00000002H	08000592H		2684619281	47.93963002
Data Write			20000014H	00000003H	08000592H		2726619281	48.68963002
Data Write			20000014H	00000004H	08000592H		2768619281	49.43963002
Data Write			20000014H	00000005H	08000592H		2810619281	50.18963002

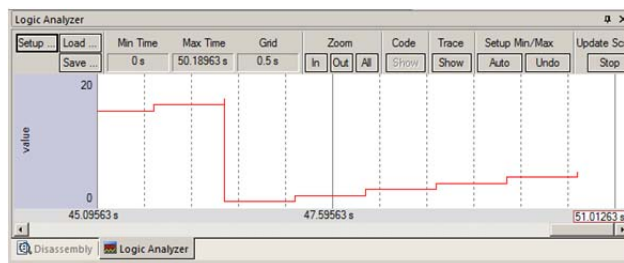
**TIP:** You cannot set Watchpoints on-the-fly while the program is running like you can with hardware breakpoints.

**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 also be entered into the Logic Analyzer. An example is: `*((unsigned long *)0x20000000)`

Shown above right is the Logic Analyzer window displaying the variable `value` trigger point of 0x5.








### 13) RTX\_Blinky Example Program with Keil RTX RTOS: A Stepper Motor example

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.

NOTE: RTX\_Blinky supplied with MDK does not have the correct source files. This example is a two task project that blinks a LED. Supplied with this document is an RTX\_Blinky that has four tasks and lights four LEDs.

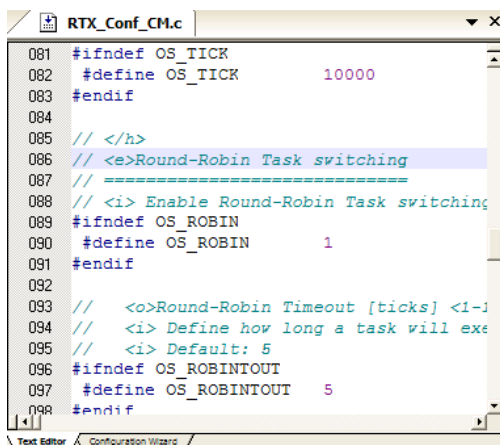
Obtain the source code for RTX\_Blinky\ from [www.keil.com/appnotes/docs/apnt\\_230.asp](http://www.keil.com/appnotes/docs/apnt_230.asp) and replace the contents in the directory C:\Keil\ARM\Boards\ST\STM32F4-Discovery\RTX\_Blinky\ . You can put it somewhere else if you prefer.

1. With  $\mu$ Vision in Edit mode (not in debug mode): Select Project/Open Project.
2. Open the file C:\Keil\ARM\Boards\ST\STM32F4-Discovery\RTX\_Blinky\Blinky.uvproj.
3. This project is pre-configured for the ST-Link V2 debug adapter.
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 four LEDs will blink in succession simulating the signals for a stepper motor.
8. Click on STOP .

We will explore the operation of RTX with the Kernel Awareness windows.

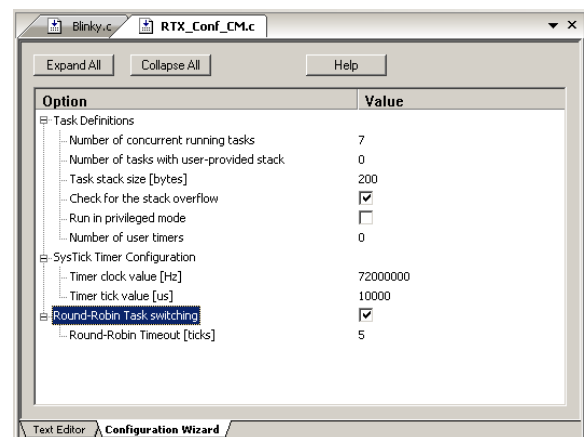
#### The Configuration Wizard for RTX:

1. Click on the RTX\_Conf\_CM.c source file tab as shown below on the left. You can open it with File/Open if needed.
2. Click on the Configuration Wizard tab at the bottom and your view will change to the Configuration Wizard.
3. 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. Changing an attribute in one tab changes it in the other automatically. You should save a modified window.
6. You can create Configuration Wizards in any source file with the scripting language as used in the Text Editor.
7. 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.
8. The  $\mu$ Vision System Viewer windows are created in a similar fashion. Select View/System Viewer.



```
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



## 14) RTX Kernel Awareness using Serial Wire Viewer (SWV):

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. Other RTOS companies also provide awareness plug-ins for  $\mu$ Vision.

1. Run RTX\_Blinky again by clicking on the Run icon.
2. Open Debug/OS Support and select RTX Tasks and System and the window on the right opens up. You might have to grab the window and move it into the center of the screen. These values are updated in real-time using the same read write technology as used in the Watch and Memory windows.

**Important TIP:** View/Periodic Window Update must be selected !

3. Open Debug/OS Support and select Event Viewer. There is probably no data displayed because SWV is not configured.

### RTX Viewer: Configuring Serial Wire Viewer (SWV):

We must activate Serial Wire Viewer to get the Event Viewer working.

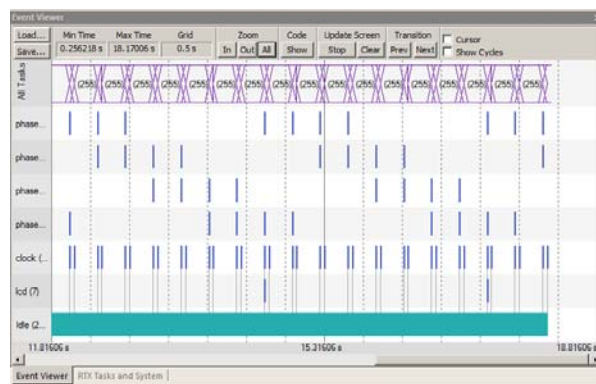
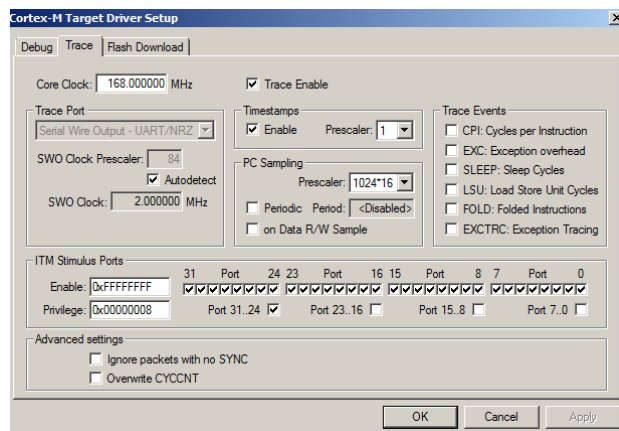
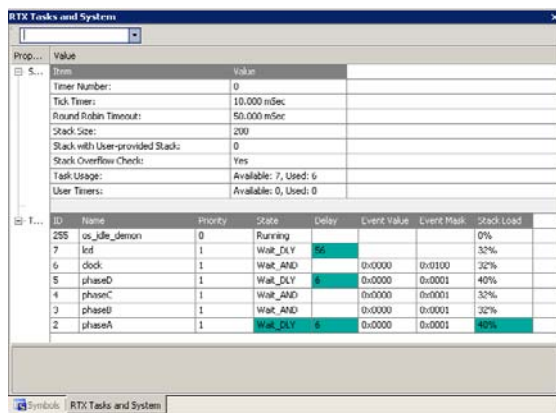
1. Stop the CPU and exit debug mode.
2. Click on the Target Options icon next to the target box.
3. Select the Debug tab. In the box Initialization File: enter `..STM32_SWO.ini` or use the Browse ... button. This file configures the STM32 SWV module and is default is for SWV UART mode. This important entry is shown above:
4. Click the Settings box next to ST-Link Debugger.
5. In the Debug window, make sure Port: is set to SW and not JTAG. SWV works only with SW mode.
6. Click on the Trace tab to open the Trace window.
7. Set Core Clock: to 168 MHz and select Trace Enable.
8. Unselect the Periodic and EXCTRC boxes as shown:
9. ITM Stimulus Port 31 must be checked. This is the method the RTX Viewer gets the kernel awareness information out to be displayed in the Event Viewer. It is slightly intrusive.
10. Click on OK twice to return to  $\mu$ Vision.  
**The Serial Wire Viewer is now configured in  $\mu$ Vision.**
11. Enter Debug mode and click on RUN to start the program.
12. Select "Tasks and System" tab: note the display is updated.
13. Click on the Event Viewer tab.
14. This window displays task events in a graphical format as shown in the RTX Kernel window below. You probably have to change the Range to about 0.2 seconds by clicking on the Zoom ALL and then the + and - icons.

**TIP:** If Event Viewer doesn't work, open up the Trace Records and confirm there are good ITM 31 frames present. Is Core Clock correct? This project is running at 168 MHz.

**Cortex-M3 Alert:**  $\mu$ Vision will update all RTX information in real-time on a target board due to its read/write capabilities as already described. The Event Viewer uses ITM and is slightly intrusive.

The data is updated while the program is running. No instrumentation code needs to be inserted into your source. You will find this feature very useful ! Remember, RTX with source code is included with all versions of MDK.



**TIP:** You can use a ULINK2, ULINK-ME, ULINK $pro$ , ST-Link V2 or J-Link for these RTX Kernel Awareness windows.










## 15) Logic Analyzer Window: View variables real-time in a graphical format:

µVision has a graphical Logic Analyzer window. Up to four variables can be displayed in real-time using the Serial Wire Viewer in the STM32. RTX\_Blinky uses four tasks to create the waveforms. We will graph these four waveforms.

1. Close the RTX Viewer windows. Stop the program  and exit debug mode. .
2. Add 4 global variables `unsigned int phasea` through `unsigned int phased` to Blinky.c as shown here:
3. Add 2 lines to each of the four tasks Task1 through Task4 in Blinky.c as shown below: `phasea=1;` and `phasea=0;`; the first two lines are shown added at lines 084 and 087 (just after LED\_On and LED\_Off function calls). For each of the four tasks, add the corresponding variable assignment statements phasea, phaseb, phasec and phased.

```
28 #define __FI 1
29
30 unsigned int phasea=0;
31 unsigned int phaseb=0;
32 unsigned int phasec=0;
33 unsigned int phased=0;
34
35 /*-----
36 * Function 'sign
```

4. Rebuild the project.  Program the Flash .
5. Enter debug mode .
6. You can run the program at this point. .
7. Open View/Analysis Windows and select Logic Analyzer or select the LA window on the toolbar. .

```
46 /*-----
47 * Task 1 'phaseA': Phase A out
48 *-----
49 task void phaseA (void) {
50     for (;;) {
51         os_evt_wait_and (0x0001, 0xffff);
52         LED_On (LED_A);
53         phasea=1;
54         signal_func (t_phaseB);
55         LED_Off (LED_A);
56         phasea=0;
57     }
58 }
```

### Enter the Variables into the Logic Analyzer (LA):

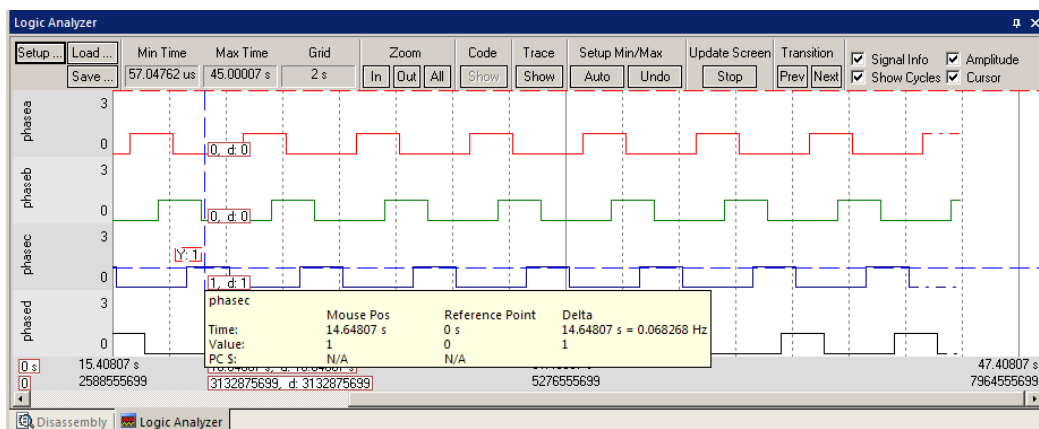
8. Click on the Blinky.c tab. Right click on `phasea`, select Add 'phasea' to... and finally select Logic Analyzer. Phasea will be added to the LA.
9. When it opens, bring the mouse down anywhere into the Logic Analyzer window and release.
10. Repeat for `phaseb`, `phasec` and `phased`. These variables will be listed on the left side of the LA window as shown. Now we have to adjust the scaling.

**TIP:** If you can't get these variables entered into the LA, make sure the Trace Config is set correctly. The Serial Wire Viewer **must** be configured correctly in order to enter variables in the LA.

The Logic Analyzer can display static and global variables, structures and arrays.

It can't see locals: just make them static or global. To see peripheral registers read or write to them and enter them in the LA.

11. Click on the Setup icon and click on each of the four variables and set Max. in the Display Range: to 0x3.
12. Click on Close to go back to the LA window.
13. Using the All, OUT and In buttons set the range to 1 second or so. Move the scrolling bar to the far right if needed.



14. Select Signal Info and Show Cycles. Click to mark a place move the cursor to get timings. Place the cursor on one of the waveforms and get timing and other information as shown in the inserted box labeled phasec:

**TIP:** You can also enter these variables into the Watch and Memory windows to display and change them in real-time.

**TIP:** You can view signals that exist mathematically in a variable and not available for measuring in the outside world.

## 16) ITM (Instrumentation Trace Macrocell) This example uses Serial Wire Viewer.


Recall that we showed you can display information about the RTOS in real-time using the RTX Viewer. This is done through ITM Stimulus Port 31. ITM Port 0 is available for a *printf* type of instrumentation that requires minimal user code. After the write to the ITM port, zero CPU cycles are required to get the data out of the processor and into  $\mu$ Vision for display in its Debug (printf) Viewer window. Note: the global variable `value` from **10) Watch and Memory Windows ...** must be entered and compiled in Blinky.c in order for this exercise to work.

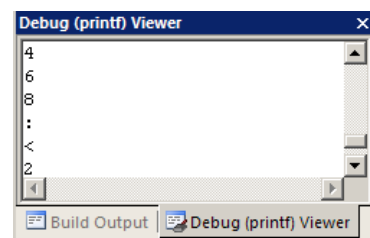
1. Stop the program if it is running and exit Debug mode.
2. Open the project C:\Keil\ARM\Boards\ST\STM32F4-Discovery\Blinky\Blinky.uvproj (do not use RTX\_Blinky).
3. Add this code to Blinky.c. A good place is near line 19, just after the `#include "LED.h"`.

```
#define ITM_Port8(n)    (*((volatile unsigned char *)(0xE0000000+4*n)))
```

4. In the main function in Blinky.c after the second Delay(200); near line 96 enter these lines:

```
ITM_Port8(0) = value + 0x30;    /* displays value in ASCII */
while (ITM_Port8(0) == 0);
ITM_Port8(0) = 0x0D;
while (ITM_Port8(0) == 0);
ITM_Port8(0) = 0x0A;
```

5. Rebuild the source files, program the Flash memory and enter debug mode.
6. Open Select Target Options  or ALT-F7 and select the Debug tab, and then the Trace tab.
7. Configure the Serial Wire Viewer as described on page 10. Use 168 MHz for the Core Clock.
8. Unselect On Data R/W Sample, EXCTRC and PC Sample. (this is to help not overload the SWO port)
9. Select ITM Port 0. ITM Stimulus Port "0" enables the Debug (printf) Viewer.
10. Click OK twice. Enter Debug mode.
11. Click on View/Serial Windows and select Debug (printf) Viewer and click on RUN.
12. In the Debug (printf) Viewer you will see the ASCII of value appear.
13. As value is incremented its ASCII character is displayed.



### Trace Records

1. Open the Trace Records if not already open. Double click on it to clear it.
2. You will see a window such as the one below with ITM and Exception frames.

### What Is This ?

You can see the ITM writes and Data writes (`value` being displayed in the LA).

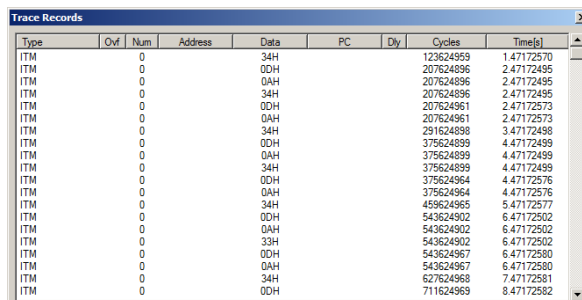
1. ITM 0 frames (Num column) are our ASCII characters from `value` with carriage return (0D) and line feed (0A) as displayed the Data column.
2. All these are timestamped in both CPU cycles and time in seconds.
3. When you are done, stop the processor and exit debug mode.

### ITM Conclusion

The writes to ITM Stimulus Port 0 are intrusive and are usually one cycle. It takes no CPU cycles to get the data out the SAM4 processor and to your PC via the Serial Wire Output pin.

**TIP:** It is important to select as few options in the Trace configuration as possible to avoid overloading the SWO pin. Enter only those features that you really need.

**Super TIP:** `ITM_SendChar` is a useful function you can use to send ITM characters. It is found in the header `core.CM3.h`.



Type	Of	Num	Address	Data	PC	Dly	Cycles	Time[s]
ITM	0	0	34H	123624959			1.47172570	
ITM	0	0	0DH	207624896			2.47172495	
ITM	0	0	0AH	207624896			2.47172495	
ITM	0	0	34H	207624896			2.47172495	
ITM	0	0	0DH	207624961			2.47172573	
ITM	0	0	0AH	207624961			2.47172573	
ITM	0	0	34H	291624898			3.47172498	
ITM	0	0	0DH	375624899			4.47172499	
ITM	0	0	0AH	375624899			4.47172499	
ITM	0	0	34H	375624964			4.47172576	
ITM	0	0	0DH	375624964			4.47172576	
ITM	0	0	0AH	375624964			4.47172576	
ITM	0	0	34H	459624965			5.47172577	
ITM	0	0	0DH	543624902			6.47172502	
ITM	0	0	0AH	543624902			6.47172502	
ITM	0	0	34H	543624902			6.47172502	
ITM	0	0	0DH	543624967			6.47172580	
ITM	0	0	0AH	543624967			6.47172580	
ITM	0	0	34H	627624968			7.47172581	
ITM	0	0	0DH	711624969			8.47172582	


## 17) Serial Wire Viewer (SWV) and how to use it:

**1) Data Reads and Writes:** (Note: Data Writes but not Reads are enabled in the current version of  $\mu$ Vision).

You have already configured Serial Wire Viewer (SWV) on page 13 under **RTX Viewer: Configuring the Serial Wire Viewer:**

Now we will examine some of the features available to you. SWV works with  $\mu$ Vision and ST-Link V2, ULINK2/ME, ULINK $pro$  or a Segger J-Link V6 or higher. SWV is included with MDK and no other equipment must be purchased.

Everything shown here is done without stealing any CPU cycles and is completely non-intrusive. Your program runs at full speed and needs no code stubs or instrumentation software added to your source code. Screens are shown using a ST-Link.

1. Use RTX\_Blinky from the last exercise. Enter Debug mode and Run the program if not already running.
2. Select View/Trace/Records or click on the Trace icon  and select Records.
3. The Trace Records window will open up as shown here:
4. The ITM frames are the data from the RTX Kernel Viewer which uses Port 31 as shown under Num. here:
5. To turn this off, select Debug/Debug Settings and click on the Trace tab. Unselect ITM Stimulus Port 31.

**TIP:** Port 0 is used for Debug `printf` Viewer.

6. Unselect EXCTRC and Periodic.
7. Select On Data R/W Sample.
8. Click on OK twice to return.
9. Click on the RUN icon.
10. Double-click anywhere in the Trace records window to clear it.

11. Only Data Writes will appear now.

**TIP:** You could have right clicked on the Trace Records window to filter the ITM frames out. Unselecting a feature is better as it reduces SWO pin traffic and therefore trace overflows.

**What is happening here ?** 

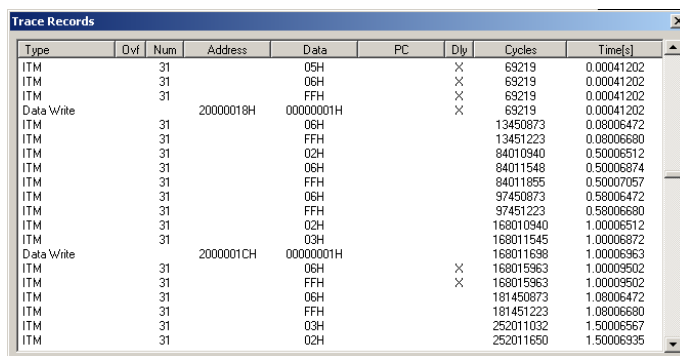
1. When variables are entered in the Logic Analyzer (remember phasea ?), the reads and/or writes will appear in Trace Records.
2. The Address column shows where the variable is.
3. The Data column displays the data values written to phasea.
4. PC is the address of the instruction causing the writes. You activated it by selecting On Data R/W Sample in the Trace configuration window.
5. The Cycles and Time(s) columns are when these events happened.

**TIP:** You can have up to four variables in the Logic Analyzer and subsequently displayed in the Trace Records window.

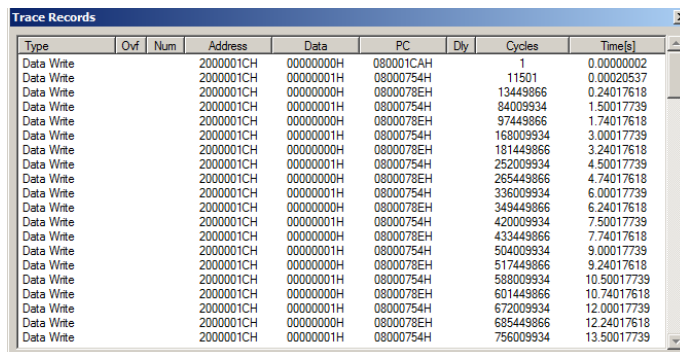
**TIP:** If you select View/Symbol Window you can see where the addresses of the variables are located.

**Note:** You must have Browser Information selected in the Options for Target/Output tab to use the Symbol Browser.

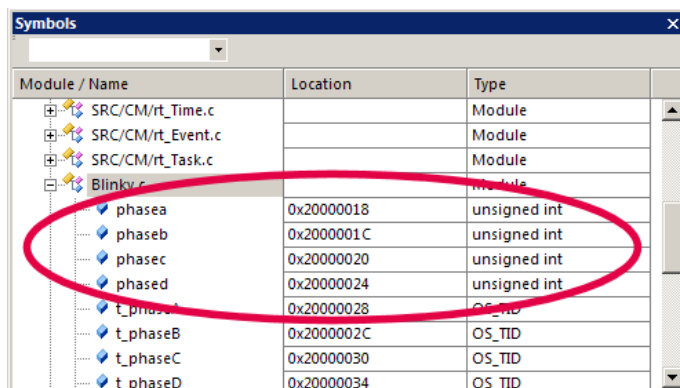
**TIP:** ULINK $pro$  and Segger J-Link adapters display the trace frames in a slightly different style trace window. The J-Link currently does not display Data writes.



Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
ITM		31		09H		X	69219	0.00041202
ITM		31		06H		X	69219	0.00041202
ITM		31		FFH		X	69219	0.00041202
Data Write			20000018H	00000001H		X	69219	0.00041202
ITM		31		06H			13450873	0.08006472
ITM		31		FFH			13451223	0.08006680
ITM		31		02H			84010940	0.50006512
ITM		31		06H			84011548	0.50006874
ITM		31		FFH			84011895	0.50007057
ITM		31		06H			97450873	0.58006472
ITM		31		FFH			97451223	0.58006680
ITM		31		02H			168010940	1.00006512
ITM		31		03H			168011545	1.00006872
Data Write			2000001CH	00000001H			168011638	1.00006963
ITM		31		06H		X	168015363	1.00009502
ITM		31		FFH		X	168015363	1.00009502
ITM		31		06H			181450873	1.08006472
ITM		31		FFH			181451223	1.08006680
ITM		31		03H			252011032	1.50006567
ITM		31		02H			252011650	1.50006935



Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Data Write			2000001CH	00000000H	080001CAH		1	0.00000002
Data Write			2000001CH	00000001H	08000754H		11501	0.00020537
Data Write			2000001CH	00000000H	0800078EH		13449866	0.24017618
Data Write			2000001CH	00000001H	08000754H		84009934	1.50017739
Data Write			2000001CH	00000000H	0800078EH		97449866	1.74017618
Data Write			2000001CH	00000001H	08000754H		16800934	3.00017739
Data Write			2000001CH	00000000H	0800078EH		181449866	3.24017618
Data Write			2000001CH	00000001H	08000754H		25200934	4.50017739
Data Write			2000001CH	00000000H	0800078EH		265449866	4.74017618
Data Write			2000001CH	00000001H	08000754H		33600934	6.00017739
Data Write			2000001CH	00000000H	0800078EH		349449866	6.24017618
Data Write			2000001CH	00000001H	08000754H		42000934	7.50017739
Data Write			2000001CH	00000000H	0800078EH		433449866	7.74017618
Data Write			2000001CH	00000001H	08000754H		50400934	9.00017739
Data Write			2000001CH	00000000H	0800078EH		517449866	9.24017618
Data Write			2000001CH	00000001H	08000754H		58800934	10.50017739
Data Write			2000001CH	00000000H	0800078EH		601449866	10.74017618
Data Write			2000001CH	00000001H	08000754H		67200934	12.00017739
Data Write			2000001CH	00000000H	0800078EH		685449866	12.24017618
Data Write			2000001CH	00000001H	08000754H		75600934	13.50017739



Module / Name	Location	Type
SRC/CM/rt_Time.c		Module
SRC/CM/rt_Event.c		Module
SRC/CM/rt_Task.c		Module
Blinky.c		Module
phasea	0x20000018	unsigned int
phaseb	0x2000001C	unsigned int
phasec	0x20000020	unsigned int
phased	0x20000024	unsigned int
t_phaseA	0x20000028	OS_TID
t_phaseB	0x2000002C	OS_TID
t_phaseC	0x20000030	OS_TID
t_phaseD	0x20000034	OS_TID

## 2) Exceptions and Interrupts:

The STM32 family using the Cortex-M4 processor has many interrupts and it can be difficult to determine when they are being activated and how often. Serial Wire Viewer (SWV) on the STM32 family makes this task easy.

1. Stop the RTX\_Blinky example program. Be in Debug mode. Open Debug/Debug Settings and select the Trace tab.
2. Unselect On Data R/W Sample, PC Sample and ITM Ports 31 and 0. (this is to minimize overloading the SWO port)
3. Select EXCTRC as shown here:
4. Click OK twice.
5. Double click on Trace Records to clear it.

**TIP:** If the SWV trace fails to work properly after this change, exit and re-enter Debug mode.

6. Click RUN to start the program.
7. You will see a window similar to the one below with Exceptions frames displayed.

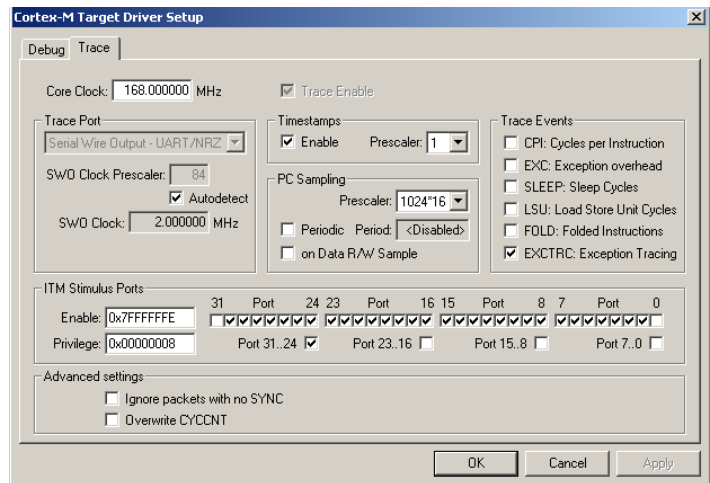
### .What Is Happening ?

1. You can see two exceptions happening.
  - **Entry:** when the exception enters.
  - **Exit:** When it exits or returns.
  - **Return:** When all the exceptions have returned to the main program. This is useful to detect tail-chaining.
2. Num 11 is SVCcall from the RTX calls.
3. Num 15 is the SysTick timer.
4. In my example you can see one data write from the Logic Analyzer.
5. Note everything is timestamped.
6. The “X” in Ovf is an overflow and some data was lost. The “X” in Dly means the timestamps are delayed because too much information is being fed out the SWO pin. Always limit the SWV features to only those you really need.

**TIP:** The SWO pin is one pin on the Cortex-M4 family processors that all SWV information is fed out. There are limitations on how much information we can feed out this one pin. These exceptions are happening at a very fast rate.  $\mu$ Vision easily recovers gracefully from these overflows. Overflows are shown when they happen. Using a ULINKpro helps reduce overruns.

1. Select View/Trace/Exceptions or click on the Trace icon and select Exceptions.
2. The next window opens up and more information about the exceptions is displayed as shown below:
3. Note the number of times these have happened under Count. This is very useful information in case interrupts come too fast or slow.
4. ExtIRQ are the peripheral interrupts.
5. You can clear this trace window by double-clicking on it.
6. All this information is displayed in real-time and without stealing any CPU cycles or stubs in your code !

**TIP:** Num is the exception number: RESET is 1. External interrupts (ExtIRQ), which are normally attached to peripherals, start at Num 16. For example, Num 41 is also known as 41-16 = External IRQ 25. Num 16 = 16 – 16 = ExtIRQ 0.



Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Exception Entry		15					3645281162	65.09430646
Exception Exit		15					3645281343	65.09430970
Exception Return		0					3645281351	65.09430984
Exception Entry		15					3646961162	65.12430646
Exception Exit		15					3646961336	65.12430957
Exception Return		0					3646961344	65.12430971
Exception Entry		15					3648641162	65.15430646
Exception Exit		15					3648641520	65.15431286
Exception Return		0					3648641528	65.15431300
Data Write			2000001CH	00000000H		X	3648643664	65.15435114
Exception Return	X	0				X	3648643664	65.15435114
Exception Entry		15					3650321163	65.18430648
Exception Exit		15					3650321397	65.18431066
Exception Return		0					3650321405	65.18431080
Exception Entry		15					3652001162	65.21430646
Exception Exit		15					3652001407	65.21431084
Exception Return		0					3652001415	65.21431098
Exception Entry		15					3653681162	65.24430646
Exception Exit		15					3653681343	65.24430970
Exception Return		0					3653681351	65.24430984

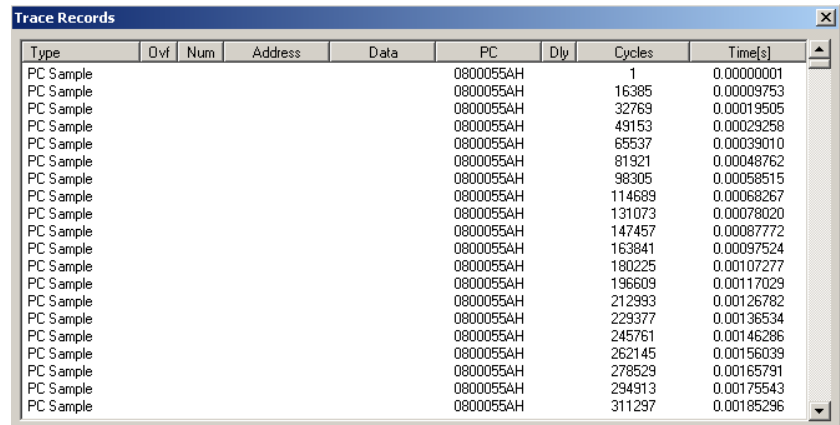
Num	Name	Count	Total Time	Min Time In	Max Time In	Min Time Out	Max Time Out	First Time [s]	Last Time [s]
3	HardFault	0	0 s						
4	MemManage	0	0 s						
5	BusFault	0	0 s						
6	UsageFault	0	0 s						
7		0	0 s						
8		0	0 s						
9		0	0 s						
10		0	0 s						
11	SVCcall	24	8.333 us	8.333 us	8.333 us	60.512 us	16.000 s	0.00005801	16.00012286
12	DbgMon	0	0 s						
13		0	0 s						
14	PendSV	0	0 s						
15	SysTick	1604	2.248 ms	1.250 us	3.315 us	9.997 ms	9.999 ms	0.01006304	16.04006300
16	ExtIRQ 0	0	0 s						
17	ExtIRQ 1	0	0 s						
18	ExtIRQ 2	0	0 s						
19	ExtIRQ 3	0	0 s						

### 3) PC Samples:

Serial Wire Viewer can display a sampling of the program counter.

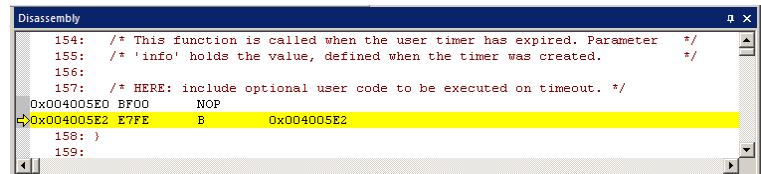
SWV can display at best every 64<sup>th</sup> instruction but usually every 16,384 is more common. It is best to keep this number as high as possible to avoid overloading the Serial Wire Output (SWO) pin. This is easily set in the Trace configuration.

1. Open Debug/Debug Settings and select the Trace tab.
2. Unselect EXCTRC, On Data R/W Sample and select Periodic in the PC Sampling area.
3. Click on OK twice to return to the main screen.
4. Close the Exception Trace window and leave Trace Records open. Double-click to clear.
5. Click on RUN and this window opens:



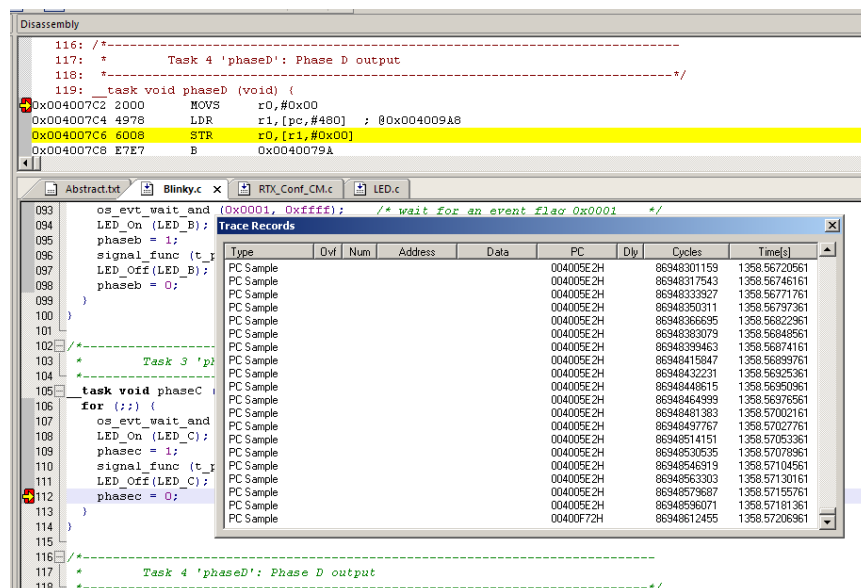
Type	Off	Num	Address	Data	PC	Dly	Cycles	Time[s]
PC Sample					0800055AH		1	0.0000001
PC Sample					0800055AH		16385	0.00009753
PC Sample					0800055AH		32769	0.00019505
PC Sample					0800055AH		49153	0.00029258
PC Sample					0800055AH		65537	0.00039010
PC Sample					0800055AH		81921	0.00048762
PC Sample					0800055AH		98305	0.00058515
PC Sample					0800055AH		114689	0.00068267
PC Sample					0800055AH		131073	0.00078020
PC Sample					0800055AH		147457	0.00087772
PC Sample					0800055AH		163841	0.00097524
PC Sample					0800055AH		180225	0.00107277
PC Sample					0800055AH		196609	0.00117029
PC Sample					0800055AH		212993	0.00126782
PC Sample					0800055AH		229377	0.00136534
PC Sample					0800055AH		245761	0.00146286
PC Sample					0800055AH		262145	0.00156039
PC Sample					0800055AH		278529	0.00165791
PC Sample					0800055AH		294913	0.00175543
PC Sample					0800055AH		311297	0.00185296

6. Most of the PC Samples in the example shown are 0x0040\_05E2 which is a branch to itself in a loop forever routine.  
**Note:** the exact address you get depends on the source code and the compiler settings.
7. Stop the program and the Disassembly window will show this Branch as shown below:
8. Not all the PCs will be captured. Still, PC Samples can give you some idea of where your program is; especially if it is not caught in a tight loop like in this case.



```
154: /* This function is called when the user timer has expired. Parameter */
155: /* 'info' holds the value, defined when the timer was created. */
156:
157: /* HERE: include optional user code to be executed on timeout. */
0x004005E0 BFD0 NOP
0x004005E2 E7FE B 0x004005E2
158:
159:
```

9. Set a breakpoint in one of the tasks.
10. Run the program and when the breakpoint is hit, you might see another address at the bottom of the Trace Records window. See the screen below:
11. Scroll to the bottom of the Trace Records window and you might (probably not) see the correct PC value displayed. Usually, it will be a different PC depending on when the sampling took place.
12. To see all the instructions executed, you can use the ETM instruction trace with a ULINK<sup>pro</sup>.
13. Remove the breakpoint.
14. Stop the program.
15. Leave Debug mode.





The screenshot shows the IDE with the Disassembly window open, displaying assembly code for a task. Below it, the Trace Records window is open, showing a list of PC samples. The PC values in the trace records are mostly 0x004005E2, which corresponds to the branch instruction in the disassembly window.

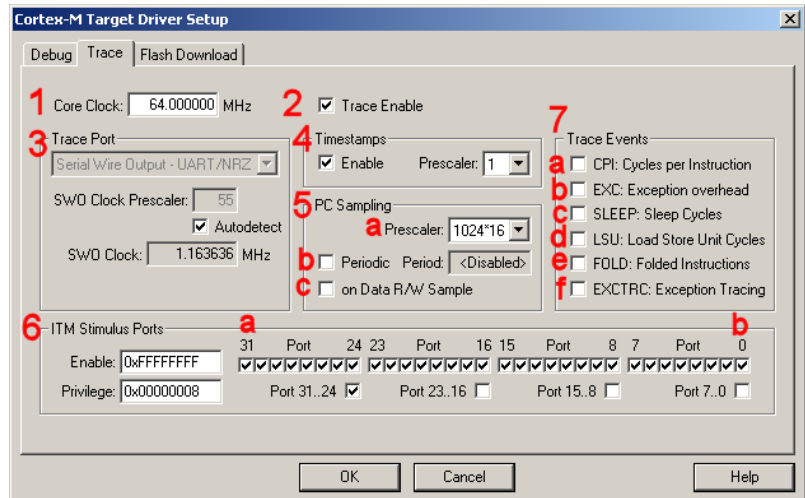


## 18) Serial Wire Viewer (SWV) Configuration window: (for reference)

The essential place to configure the trace is in the Trace tab as shown below. You cannot set SWV globally for  $\mu$ Vision. You must configure SWV for every project and additionally for every target settings within a project you want to use SWV. This configuration information will be saved in the project. There are two ways to access this menu:

- A. **In Edit mode:** Select Target Options  or ALT-F7 and select the Debug tab. Select Settings: on the right side of this window and then the Trace tab. Edit mode is selected by default when you start  $\mu$ Vision.
- B. **In Debug mode:** Select Debug/Debug Settings and then select the Trace tab. Debug mode is selected with .

- 1) **Core Clock:** The CPU clock speed for SWV. The CPU speed can be found in your startup code or in Abstract.txt. It is usually called SYSCLK or Main Clock. This *must* be set correctly for all adapters except ULINKpro.
- 2) **Trace Enable:** Enables SWV and ITM. It can only be changed in Edit mode. This does not affect the Watch and Memory window display updates.
- 3) **Trace Port:** This is preset for ST-Link.
- 4) **Timestamps:** Enables timestamps and selects the Prescaler. 1 is the default.
- 5) **PC Sampling:** Samples the program counter.



- a. **Prescaler** 1024\*16 (the default) means every 16,384<sup>th</sup> PC is displayed. The rest are not collected.
- b. **Periodic:** Enables PC Sampling.
- c. **On Data R/W Sample:** Displays the address of the instruction that caused a data read or write of a variable listed in the Logic Analyzer. This is not connected with PC Sampling but rather with data tracing.
- 6) **ITM Stimulus Ports:** Enables the thirty-two 32 bit registers used to output data in a *printf* type statement to  $\mu$ Vision. Port 31 (a) is used for the Keil RTX Viewer which is a real-time kernel awareness window. Port 0 (b) is used for the Debug (printf) Viewer. The rest are currently unused in  $\mu$ Vision.
  - **Enable:** Displays a 32 bit hex number indicating which ports are enabled.
  - **Privilege:** Privilege is used by an RTOS to specify which ITM ports can be used by a user program.
- 7) **Trace Events:** Enables various CPU counters. All except EXCTRC are 8 bit counters. Each counter is cumulative and an event is created when this counter overflows every 256 cycles. These values are displayed in the Counter window. The event created when a counter wraps around is displayed in the Instruction Trace window.
  - a. **CPI: Cycles per Instruction:** The cumulative number of extra cycles used by each instruction beyond the first, one including any instruction fetch stalls.
  - b. **Fold:** Cumulative number of folded instructions. These results from a predicted branch instruction where unused instructions are removed (flushed) from the pipeline giving a zero cycle execution time.
  - c. **Sleep:** Cumulative number of cycles the CPU is in sleep mode. Uses FCLK for timing.
  - d. **EXC:** Cumulative cycles CPU spent in exception overhead not including total time spent processing the exception code. Includes stack operations and returns.
  - e. **LSU:** Cumulative number of cycles spent in load/store operations beyond the first cycle.
  - f. **EXCTRC:** Exception Trace. This is different than the other items in this section. This enables the display of exceptions in the Instruction Trace and Exception windows. It is not a counter. This is a very useful feature to display exception events and is often used in debugging.

**TIP:** Counters will increment while single stepping. This can provide some very useful information. You can read these counters with your program as they are memory mapped.






## 19) DSP SINE example using ARM CMSIS-DSP Libraries:

ARM CMSIS-DSP libraries are offered for ARM Cortex-M3 and Cortex-M4 processors. DSP libraries are provided in MDK in C:\Keil\ARM\CMSIS. README.txt describes the location of various CMSIS components. See [www.arm.com/cmsis](http://www.arm.com/cmsis) and [forums.arm.com](http://forums.arm.com) for more information. CMSIS is an acronym for Cortex Microcontroller Software Interface Standard.

This example creates a sine wave with noise added, and then the noise is filtered out. The waveform in each step is displayed in the Logic Analyzer using Serial Wire Viewer.

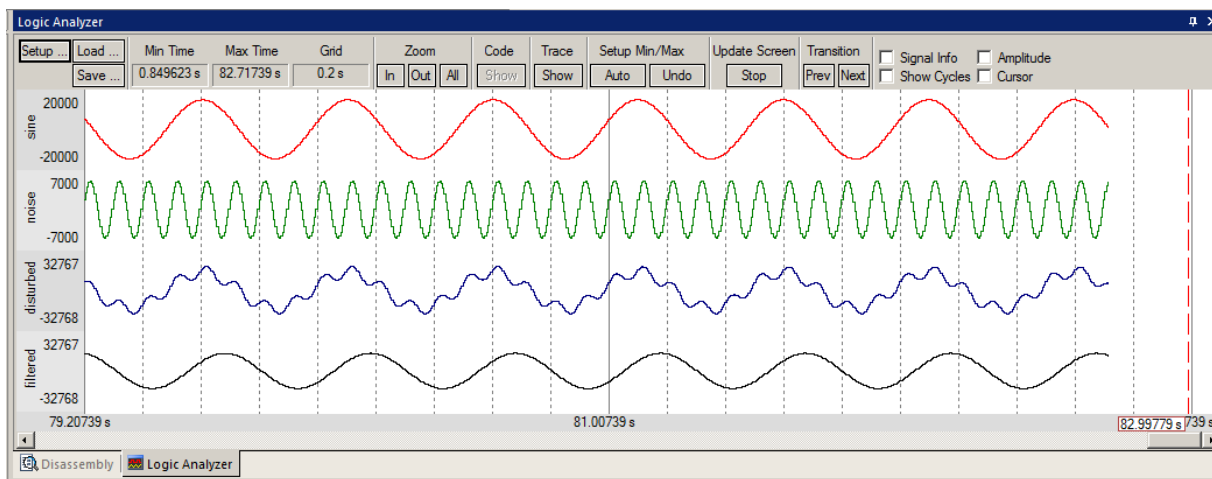
This example incorporates Keil RTX RTOS. RTX is available free with a BSD type license. RTX source code is provided.

To obtain this example file, go to [www.keil.com/appnotes/docs/apnt\\_230.asp](http://www.keil.com/appnotes/docs/apnt_230.asp) Extract DSP to ...\\STM32F4-Discovery\.

1. Open the project file sine: C:\Keil\ARM\Boards\ST\STM32F4-Discovery\DSP\sine.uvproj
2. Build the files.  There will be no errors or warnings.
3. Program the STM32 flash by clicking on the Load icon:  Progress will be indicated in the Output Window.
4. Enter Debug mode by clicking on the Debug icon.  Select OK if the Evaluation Mode box appears.
5. Click on the RUN icon.  Open the Logic Analyzer window. 
6. Four waveforms will be displayed in the Logic Analyzer using the Serial Wire Viewer as shown below. Adjust Zoom Out for an appropriate display. Displayed are 4 global variables: sine, noise, disturbed and filtered.

**TIP:** If one variable shows no waveform, disable the ITM Stimulus Port 31 in the Trace Config window.

7. The project provided has Serial Wire Viewer configured and the Logic Analyzer loaded with the four variables.



8. Open the Trace Records window and the Data Writes to the four variables are listed as shown here:
9. Leave the program running.
10. Close the Trace Records window.

**TIP:** The ULINK<sub>pro</sub> trace display is different and the program must be stopped to update it.

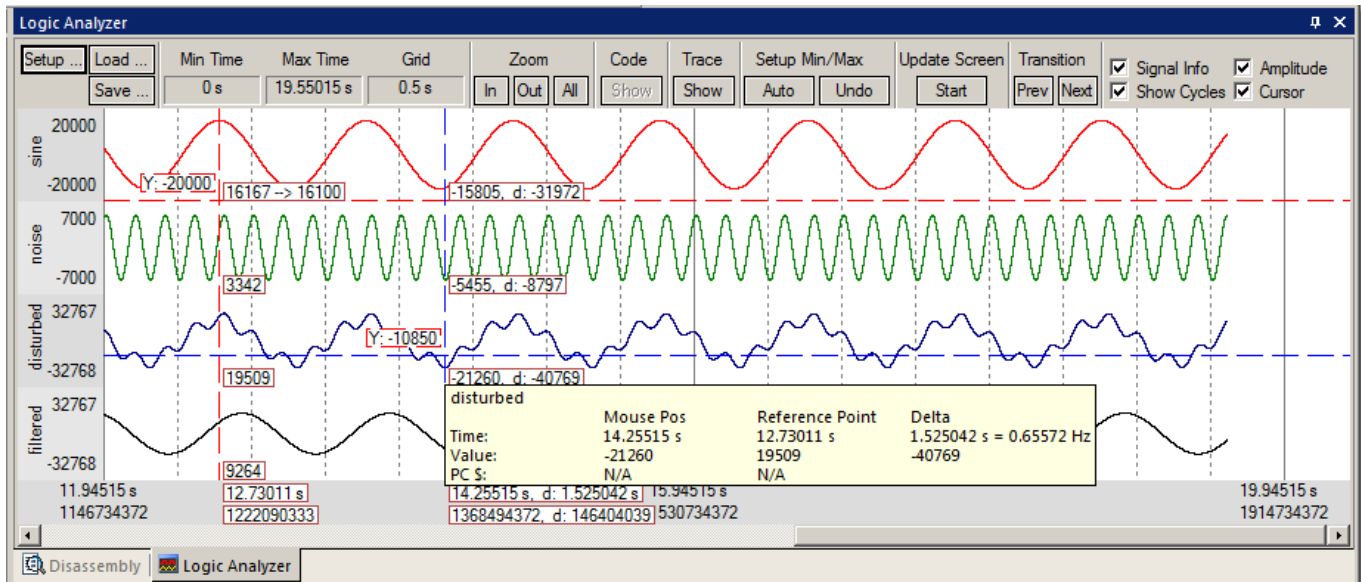
The Watch 1 window will display the four variables updating in real time as shown below:

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

Type	Of	Num	Address	Data	PC	Dly	Cycles	Time[s]
Data Write			20000000H	2C2DH	00400252H		9741265867	101.47151945
Data Write			20000002H	F081H	00400280H	X	9741274850	101.47161302
Data Write	X		20000006H	F326H	004002CEH	X	9741274850	101.47161302
Data Write			20000000H	2EF4H	00400252H		9741745582	101.47651648
Data Write			20000002H	F5CBH	00400280H	X	9741754550	101.47660990
Data Write	X		20000006H	F6FFH	004002CEH	X	9741754550	101.47660990
Data Write			20000000H	318CH	00400252H		9742225305	101.48151359
Data Write			20000002H	FC15H	00400280H	X	9742234332	101.48160762
Data Write	X		20000006H	FAE2H	004002CEH	X	9742234332	101.48160762
Data Write			20000000H	33F1H	00400252H		9742705028	101.48651071
Data Write			20000002H	02CDH	00400280H	X	9742714032	101.48660450
Data Write	X		20000006H	FECBH	004002CEH	X	9742714032	101.48660450
Data Write			20000000H	3621H	00400252H		9743184839	101.49150874
Data Write			20000002H	0927H	00400280H	X	9743193814	101.49160223
Data Write	X		20000006H	02B7H	004002CEH	X	9743193814	101.49160223
Data Write			20000000H	381AH	00400252H		9743664482	101.49650502
Data Write			20000002H	0EA9H	00400280H	X	9743673432	101.49659825
Data Write	X		20000006H	06A2H	004002CEH	X	9743673432	101.49659825
Data Write			20000000H	39DAH	00400252H		9744144197	101.50150205
Data Write			20000002H	12BAH	00400280H	X	9744153214	101.50159598

### Signal Timings in Logic Analyzer (LA):

1. In the LA window, select Signal Info, Show Cycles, Amplitude and Cursor.
2. Click on STOP in the Update Screen box. You could also stop the program but leave it running in this case.
3. Click somewhere in the LA to set a reference cursor line.
4. Note as you move the cursor various timing information is displayed as shown below:



### RTX Tasks and System:

5. Click on Start in the Update Screen box to resume the collection of data.
6. Open Debug/OS Support and select RTX Tasks and System. A window similar to below opens up. You probably have to click on its header and drag it into the middle of the screen.
7. Note this window does not update: nearly all the processor time is spent in the idle daemon: it shows it is Running. The processor spends relatively little time in other tasks. You will see this illustrated clearly on the next page.
8. Set a breakpoint in one of the tasks in DirtyFilter.c by clicking in the left margin on a grey area.
9. Click on Run and the program will stop here and the Task window will be updated accordingly. Here, I set a breakpoint in the noise\_gen task:
10. Clearly you can see that noise\_gen was running when the breakpoint was activated.
11. Remove the breakpoint.

**TIP:** You can set hardware breakpoints while the program is running.

**TIP:** Recall this window uses the CoreSight DAP read and write technology to update this window. Serial Wire Viewer is not used and is not required to be activated for this window to display and be updated.

The Event Viewer does use SWV and this is demonstrated on the next page.

The screenshot shows the RTX Tasks and System window. It contains two main sections: System properties and a list of tasks.

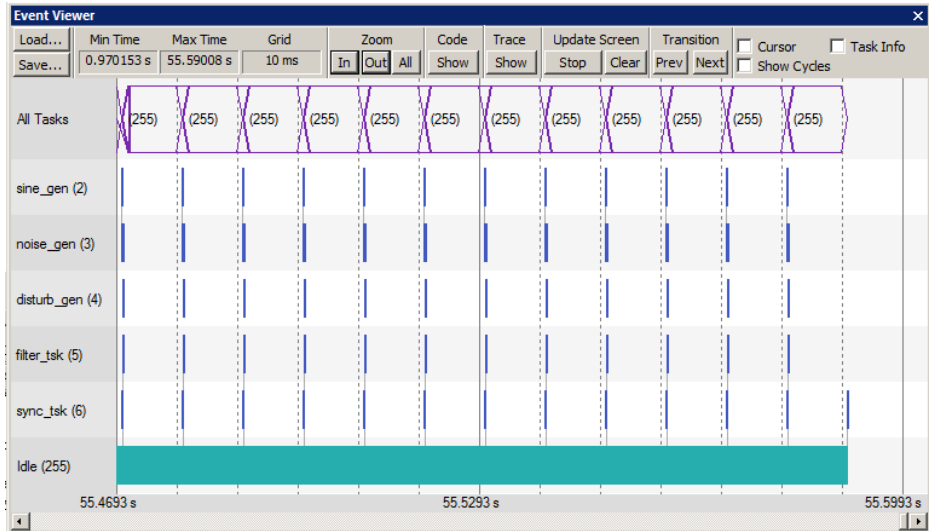
Property	Value
Item	Value
Timer Number:	0
Tick Timer:	10.000 mSec
Round Robin Timeout:	
Stack Size:	200
Tasks with User-provided Stack:	0
Stack Overflow Check:	Yes
Task Usage:	Available: 7, Used: 5
User Timers:	Available: 0, Used: 0

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Ready				32%
6	sync_tsk	1	Wait_DLY	1			32%
5	filter_tsk	1	Wait_AND		0x0000	0x0001	32%
4	disturb_gen	1	Wait_AND		0x0000	0x0001	32%
3	noise_gen	1	Running		0x0000	0x0001	0%
2	sine_gen	1	Wait_AND		0x0000	0x0001	32%

### Event Viewer:

1. Stop the program. Click on Setup... in the Logic Analyzer. Select Kill All to remove all variables. This is necessary because the SWO pin will likely be overloaded when the Event Viewer is opened up. Inaccuracies might occur. If you like – you can leave the LA loaded with the four variables to see what the Event Viewer will look like.
2. Select Debug/Debug Settings.
3. Click on the Trace tab.
4. Enable ITM Stimulus Port 31. Event Viewer uses this to collect its information.
5. Click OK twice.
6. Click on RUN.
7. Open Debug/OS Support and select Event Viewer. The window here opens up:
8. Note there is no Task 1 listed. Task 1 is main\_tsk and is found in DirtyFilter.c near line 208. It runs some RTX initialization code at the beginning and then deletes itself with `os_tsk_delete_self()`; found near line 195.



**Important TIP:** If the SWV trace fails to work properly after this change, exit and re-enter Debug mode.

**TIP:** If Event Viewer is blank or erratic, or the LA variables are not displaying or blank; this is likely because the Serial Wire Output pin is overloaded and dropping trace frames. Solutions are to delete some or all of the variables in the Logic Analyzer to free up some bandwidth.

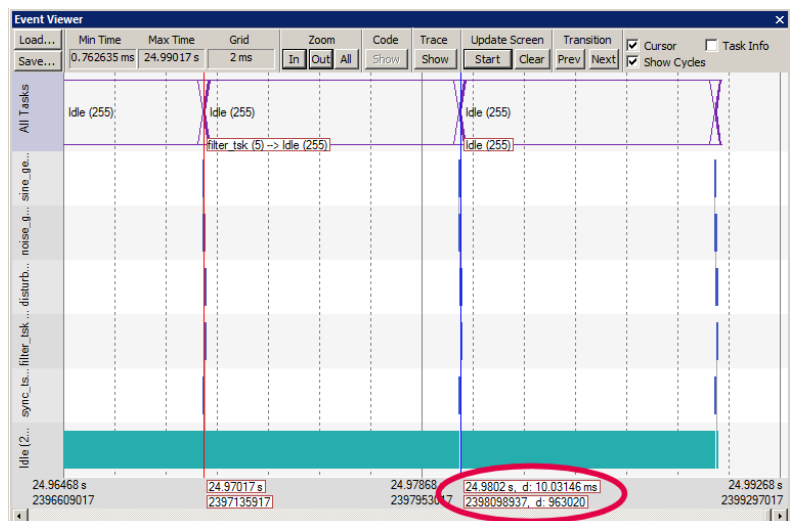
ULINK<sub>pro</sub> is much better with SWO bandwidth issues. These have been able to display both the Event and LA windows. ULINK<sub>pro</sub> uses the faster Manchester format than the slower UART mode that ST-Link, ULINK2 and J-Link uses.

ULINK<sub>pro</sub> can also use the 4 bit Trace Port for faster operation for SWV. The Trace Port is mandatory for ETM trace.

- Note on the Y axis each of the 5 running tasks plus the idle daemon. Each bar is an active task and shows you what task is running, when and for how long.
- Click Stop in the Update Screen box.
- Click on Zoom In so three or four tasks are displayed.
- Select Cursor. Position the cursor over one set of bars and click once. A red line is set here:
- Move your cursor to the right over the next set and total time and difference are displayed.
- Note, since you enabled Show Cycles, the total cycles and difference is also shown.

The 10 msec shown is the SysTick timer value. This value is set in `RTX_Conf_CM.c`. The next page describes how to change this.

**TIP:** ITM Port 31 enables sending the Event Viewer frames out the SWO port. Disabling this can save bandwidth on the SWO port if you are not using the Event Viewer.

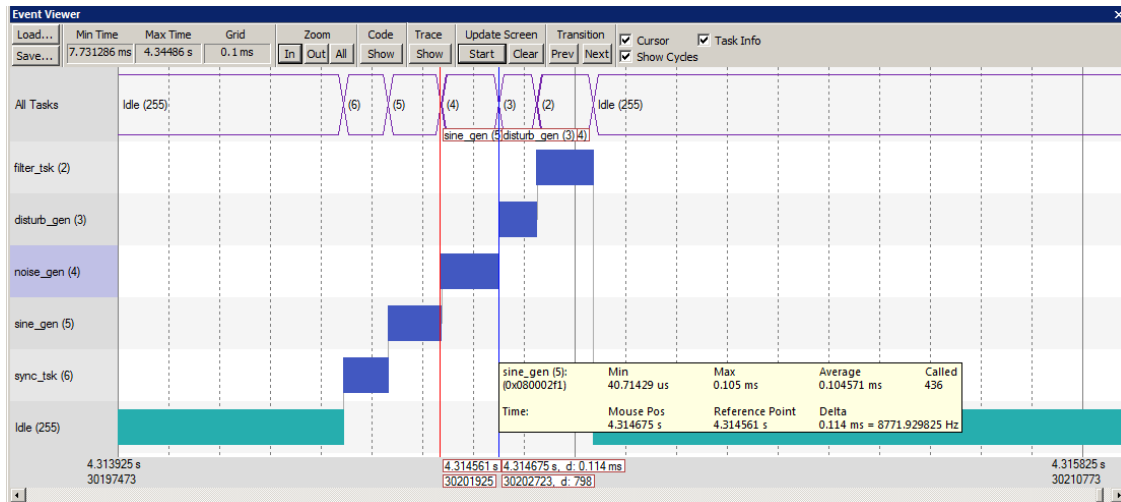


### Event Viewer Timing:



1. Click on Zoom In until one set of tasks is visible as shown below:
2. Enable Task Info (as well as Cursor and Show Cycles from the previous exercise).
3. Note one entire sequence is shown. This screen is taken with ST-Link with LA cleared of variables.
4. Click on a task to set the cursor and move it to the end. The time difference is noted. The Task Info box will appear.

**TIP:** If the Event Viewer does not display correctly, the display of the variables in the Logic Analyzer window might be overloading the SWO pin. In this case, stop the program and delete all variables (Kill All) and click on Run.



The Event Viewer can give you a good idea if your RTOS is configured correctly and running in the right sequence.

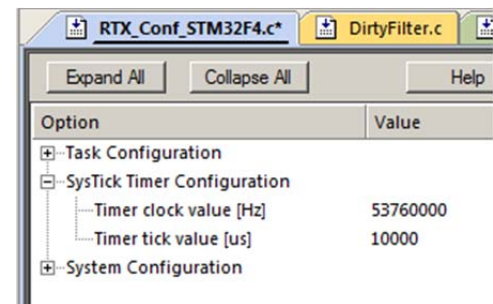


### SysTick Timer Changing:

1. Stop the processor  and exit debug mode .
2. Open the file RTX\_Conf\_CM.c from the Project window. You can also select File/Open in C:\Keil\ARM\Boards\ST\STM32F4-Discovery\DSP.
3. Select the Configuration Wizard tab at the bottom of the window. See page 12 for an explanation on how the Wizard works.
4. This window opens up. Expand SysTick Timer Configuration.
5. Note the Timer tick value is 10,000 usec or 10 msec.
6. Change this value to 20,000.

**TIP:** The 5,376,000 is the CPU speed. The Discovery board has a 8 MHz crystal. This program was designed for 168 MHz with a 25 MHz crystal. Therefore it runs 8/25 slower than designed for. The PLL is configured in CMSIS file system\_stm32f4xx.c and is easily modified.

7. Rebuild the source files and program the Flash.
8. Enter debug mode  and click on RUN .
9. When you check the timing of the tasks in the Event Viewer window as you did on the previous page, they will now be spaced at 20 msec.



**TIP:** The SysTick is a dedicated timer on Cortex-M processors that is used to switch tasks in an RTOS. It does this by generating an exception 15. You can view these exceptions in the Trace Records window by enabling EXCTRC in the Trace Configuration window.

1. Set the SysTick timer back to 10,000. You will need to recompile the source files and reprogram the Flash.

***This ends the exercises. Thank you !***

Next is how to make a new project from scratch (almost scratch) and Keil product and contact information.



## 20) Creating your own project from scratch: Using the Blinky source files:

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. We will use the existing source code files so you will not have to type them in. Once you have the new project configured; you can build, load and run a bare Blinky example. It has 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. This is for a STM32 processor and can be used for a Cortex-M4.

### Create a new project called Mytest:

1. With  $\mu$ Vision running and not in debug mode, select Project/New  $\mu$ Vision Project...
2. In the window Create New Project that opens, go to the folder C:\Keil\ARM\Boards\ST\STM32F4-Discovery.

### Create a new folder and name your project:

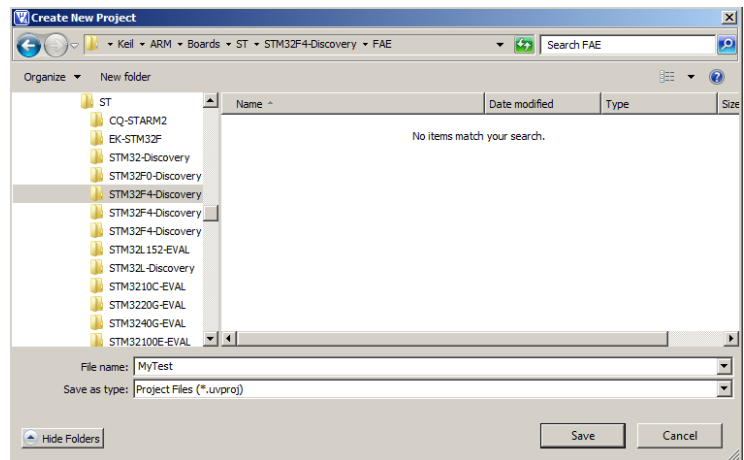
3. Right click inside this window and create a new folder by selecting New/Folder. I named this new folder FAE.
4. Double-click on the newly created folder “FAE” to enter this folder. It will be empty.
5. Name your project in the File name: box. I called mine Mytest. You can choose your own name but you will have to keep track of it. This window is shown here:
6. Click on Save.

### Select your processor:

7. The Select Device for “Target 1” opens up as shown at the bottom of this page:
  8. This is the Keil Device Database<sup>®</sup> which lists all the devices Keil supports. You can create your own if desired for processors not released yet.
  9. Locate the ST directory, open it and select a STM32F415ZG (or the device you are using). Note the device features are displayed.
  10. Select OK.
- $\mu$ Vision will configure itself to this device.

### Select the startup file:

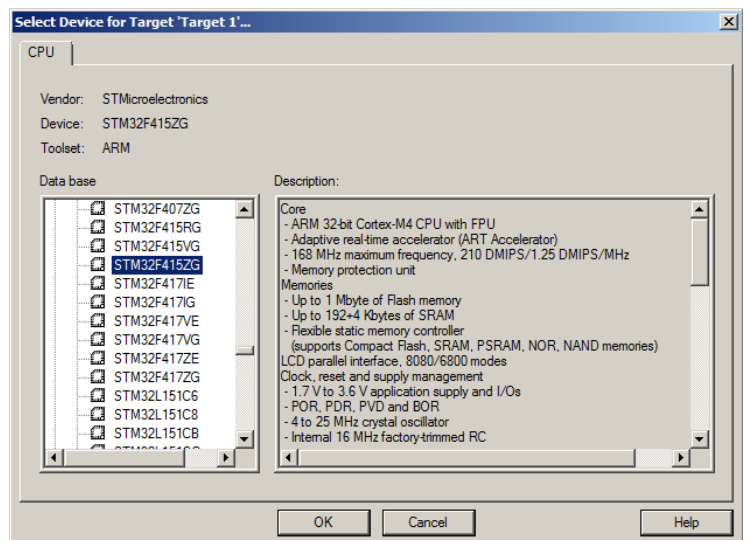
11. A window opens up asking if you want to insert the default STM32 startup file to your project. Click on “Yes”. This will save you some time.
12. In the Project Workspace in the upper left hand of  $\mu$ Vision, expand the folders Target 1 and Source Group 1 by clicking on the “+” beside each folder.
13. We have now created a project called Mytest with the target hardware called Target 1 with one source assembly file startup\_stm32f4xx.s and using the STM32F4 processor you chose.



**TIP:** You can create more target hardware configurations and easily select them. This can include multiple Options settings, simulation and RAM operations. See Projects/Manage/Components

### Rename the Project names for convenience:

14. Click once on the name “Target 1” (or twice if not already highlighted) in the Project Workspace and rename Target 1 to something else. I chose STM32F4 Flash. Press Enter to accept this change. Note the Target selector in the main  $\mu$ Vision window also changes to STM32F4 Flash. This is your first target.
15. Similarly, change Source Group 1 to Startup. This will add some consistency to your project with the Keil examples. You can name these or organize them differently to suit yourself.
16. Select File/Save All.



*Continued on the next page...*


### Select the source files and debug adapter:

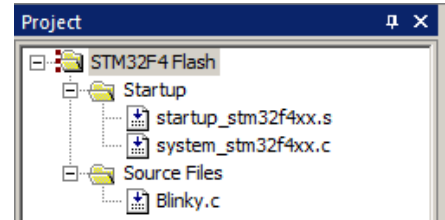
1. Using MS Explore (right click on Windows Start icon), copy blinky.c and system\_stm32f4xx.c from C:\Keil\ARM\Boards\ST\STM32F4-Discovery\Blinky to the ..\STM32F4-Discovery\FAE folder you created. Copy these files from an example project that utilizes a similar STM32 processor as you are using.

### Source Files:


2. In the Project Workspace in the upper left hand of  $\mu$ Vision, right-click on “STM32F4 Flash” and select “Add Group”. Name this new group “Source Files” and press Enter. You can name it anything. There are no restrictions from Keil.
3. Right-click on “Source Files” and select **Add files to Group “Source Files”**.
4. Select the file Blinky.c and click on Add (once!) and then Close.

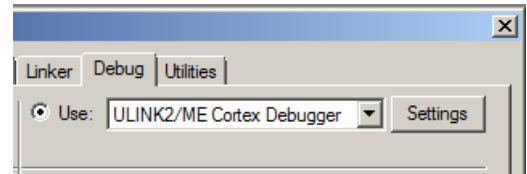
### System File:

5. Right-click on “Startup” and select **Add files to Group “Source Files”**.
6. Select system\_stm32f4xx.c and click on Add (once!) and then Close.
7. Your Project window will look similar to the one shown here: 



### Select your Debug Adapter:

8. By default the simulator is selected when you create a new  $\mu$ Vision project. You probably need to change this to a Debug adapter such as ST-Link, ULINK2 or ULINKpro.
9. Select Target Options  or ALT-F7 and select the Debug tab. Select ULINK/ME Cortex Debugger as shown below: If you are using another adapter such as ST-Link, J-Link or ULINKpro, select the appropriate adapter from the pull-down list.
10. Select JTAG/SWD hardware debugging (as opposed to selecting the Simulator) by checking the circle just to the left of the word “Use:” as shown in the window to the right:
11. Select Run to Main (unless you do not want this)
12. Select the Utilities tab and select the appropriate debug adapter and the proper Flash algorithm for your processor.
13. Click on the Target tab and select MicroLIB for smaller programs. See [www.keil.com/appnotes/files/apnt202.pdf](http://www.keil.com/appnotes/files/apnt202.pdf) for details.
14. Click on OK.



### Modify Blinky.c

15. Double-click the file Blinky.c in the Project window to open it in the editing window or click on its tab if open.
16. Delete everything in Blinky.c except the main () function to provide a basic platform to start with:






```
include <stdio.h>
#include "STM32F4xx.h"

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

    while (1); {                loop forever
    }
}
```

17. Select File/Save All

### Compile and run the program:

18. Compile the source files by clicking on the Rebuild icon. . You can also use the Build icon beside it.
19. Program the STM32 flash by clicking on the Load icon: . Progress will be indicated in the Output Window.
20. Enter Debug mode by clicking on the Debug icon. 
21. Click on the RUN icon.  Note: you stop the program with the STOP icon. 
22. The program will run but since while (1) is empty – it does not do much. It consists of a NOP and Branch to itself. You can set a breakpoint on any assembly or source lines that have a darker grey box signifying assembly code.
23. You are able to add your own source code to create a meaningful project.

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







You can also configure a new RTX project from scratch using the RTX\_Blinky project.

## ETM Trace Examples: *For reference only...*

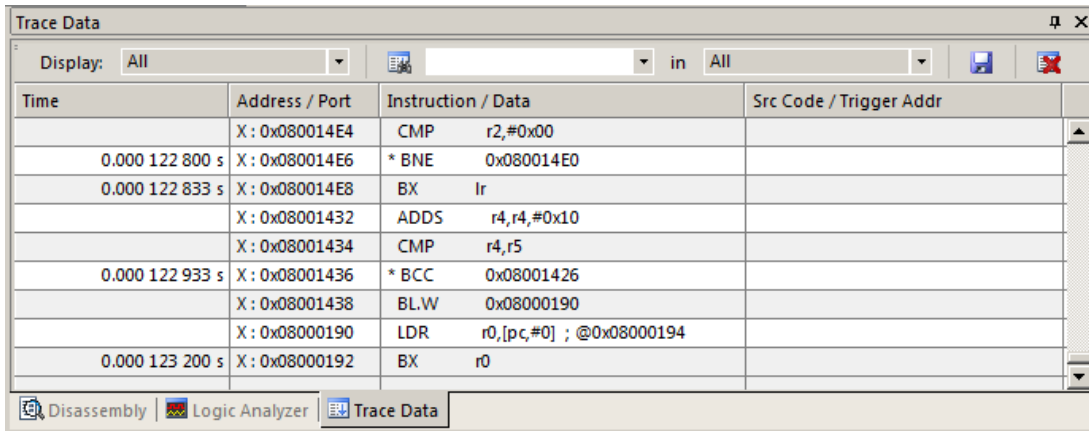
These examples were run on the STM3240G-EVAL evaluation board. These are applicable for the Keil MCBSTM32F400 board. These examples are included for reference. A ULINK<sub>pro</sub> debug adapter is required for ETM operation.

ETM provides serious debugging power as shown on the next few pages. It is worth the small added cost.


Most STM32 processors are ETM equipped.

1. Connect the ULINK<sub>pro</sub> to the STM3240G board using the 20 pin CN13 Trace connector.
2. Start  $\mu$ Vision by clicking on its desktop icon. 
3. Select Project/Open Project. Open C:\Keil\ARM\Boards\ST\STM3240G-EVAL\Blinky\_Ulp\Blinky.uvproj.
4. Select TracePort Instruction Trace in the Target Options box as shown here: 
5. Compile the source files by clicking on the Rebuild icon.  You can also use the Build icon beside it.
6. Program the STM32 flash by clicking on the Load icon:  Progress will be indicated in the Output Window.
7. Enter Debug mode by clicking on the Debug icon.  Select OK if the Evaluation Mode box appears.
8. DO NOT CLICK ON RUN YET !!!
9. Open the Data Trace window by clicking on the small arrow beside the Trace Windows icon. 

10. Examine the Instruction Trace window as shown below: This is a complete record of all the program flow since RESET until  $\mu$ Vision halted the program at the start of main() since Run To main is selected in  $\mu$ Vision.



Time	Address / Port	Instruction / Data	Src Code / Trigger Addr
	X : 0x080014E4	CMP r2,#0x00	
0.000 122 800 s	X : 0x080014E6	* BNE 0x080014E0	
0.000 122 833 s	X : 0x080014E8	BX lr	
	X : 0x08001432	ADDS r4,r4,#0x10	
	X : 0x08001434	CMP r4,r5	
0.000 122 933 s	X : 0x08001436	* BCC 0x08001426	
	X : 0x08001438	BLW 0x08000190	
	X : 0x08000190	LDR r0,[pc,#0] ; @0x08000194	
0.000 123 200 s	X : 0x08000192	BX r0	



11. In this case, 123 200 s shows the last instruction to be executed. (BX r0). In the Register window the PC will display the value of the next instruction to be executed (0x0800\_0192 in my case). Click on Single Step once. 
12. The instruction PUSH will display: | 0x080011DA | PUSH (r3,lr) | int main(void) { /\* Main Program \*/ |
13. Scroll to the top of the Instruction Trace window to frame # 1. This is the first instruction executed after RESET.

**A STM3240G-EVAL board connected to a ULINK<sub>pro</sub> using the special CoreSight 20 pin ETM connector:**










nearly

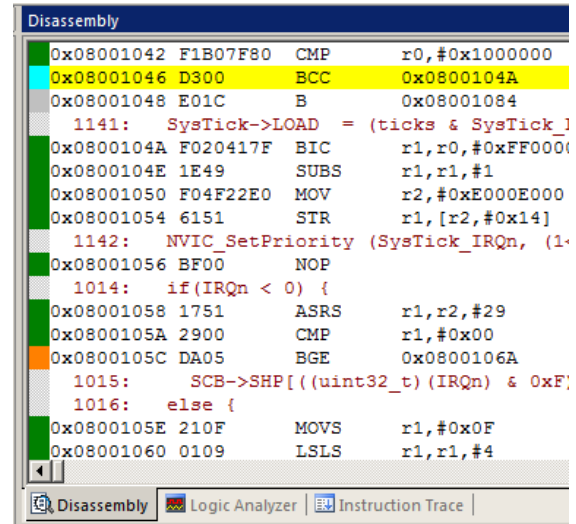
### 3) Code Coverage: For reference only...

14. Click on the RUN icon.  After a second or so stop the program with the STOP icon. 
15. Examine the Disassembly and Blinky.c windows. Scroll and notice different color blocks in the left margin:
16. This is Code Coverage provided by ETM trace. This indicates if an instruction has been executed or not.

Colour blocks indicate which assembly instructions have been executed.

-  1. Green: this assembly instruction was executed.
-  2. Gray: this assembly instruction was not executed.
-  3. Orange: a Branch is always not taken.
-  4. Cyan: a Branch is always taken.
-  5. Light Gray: there is no assembly instruction at this point.
-  6. RED: Breakpoint is set here.
-  7. Next instruction to be executed.

In the window on the right you can easily see examples of each type of Code Coverage block and if they were executed or not and if branches were taken (or not).



```
Disassembly
0x08001042 F1B07F80 CMP      r0,#0x1000000
0x08001046 D300    BCC      0x0800104A
0x08001048 E01C    B        0x08001084
1141: SysTick->LOAD = (ticks & SysTick_
0x0800104A F020417F BIC      r1,r0,#0xFF0000
0x0800104E 1E49    SUBS     r1,r1,#1
0x08001050 F04F22E0 MOV      r2,#0xE000E000
0x08001054 6151    STR      r1,[r2,#0x14]
1142: NVIC_SetPriority (SysTick_IRQn, (1
0x08001056 BF00    NOP
1014: if (IRQn < 0) {
0x08001058 1751    ASRS      r1,r2,#29
0x0800105A 2900    CMP      r1,#0x00
0x0800105C DA05    BGE      0x0800106A
1015: SCB->SHP[ ((uint32_t) (IRQn) & 0xF
1016: else {
0x0800105E 210F    MOVS      r1,#0x0F
0x08001060 0109    LSLS     r1,r1,#4
```

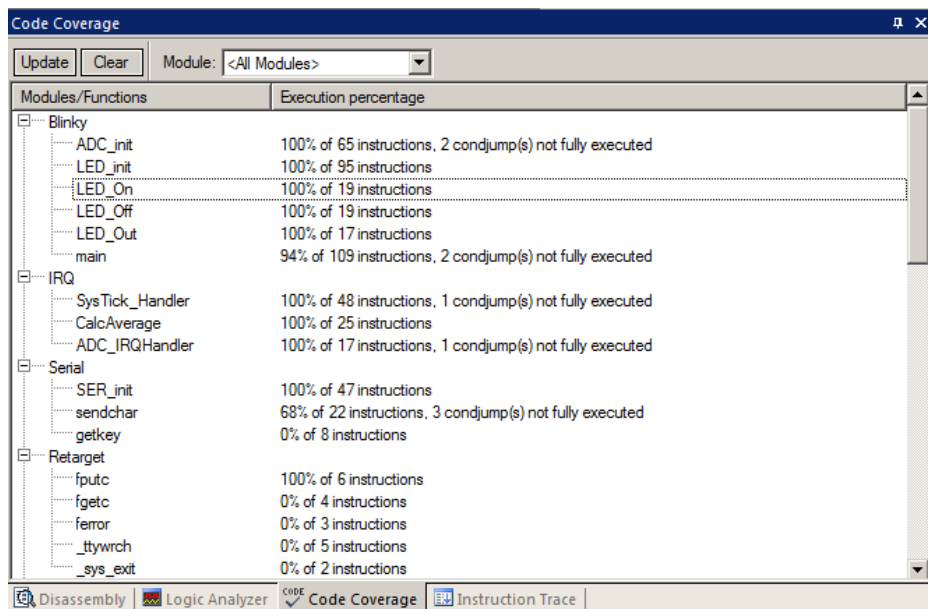
Why was the branch BCC always taken resulting in 0x0800\_1048 never being executed ? Or why the branch BGE at 0x800\_105C was never taken ? You should devise tests to execute these instructions so you can test them.

Code Coverage tells what assembly instructions were executed. It is important to ensure all assembly code produced by the compiler is executed and tested. You do not want a bug or an unplanned circumstance to cause a sequence of untested instructions to be executed. The result could be catastrophic as unexecuted instructions cannot be tested. Some agencies such as the US FDA require Code Coverage for certification.

Good programming practice requires that these unexecuted instructions be identified and tested.

Code Coverage is captured by the ETM. Code Coverage is also available in the Keil Simulator.


A Code Coverage window is available as shown below. This window is available in View/Analysis/Code Coverage. Note your display may look different due to different compiler options.

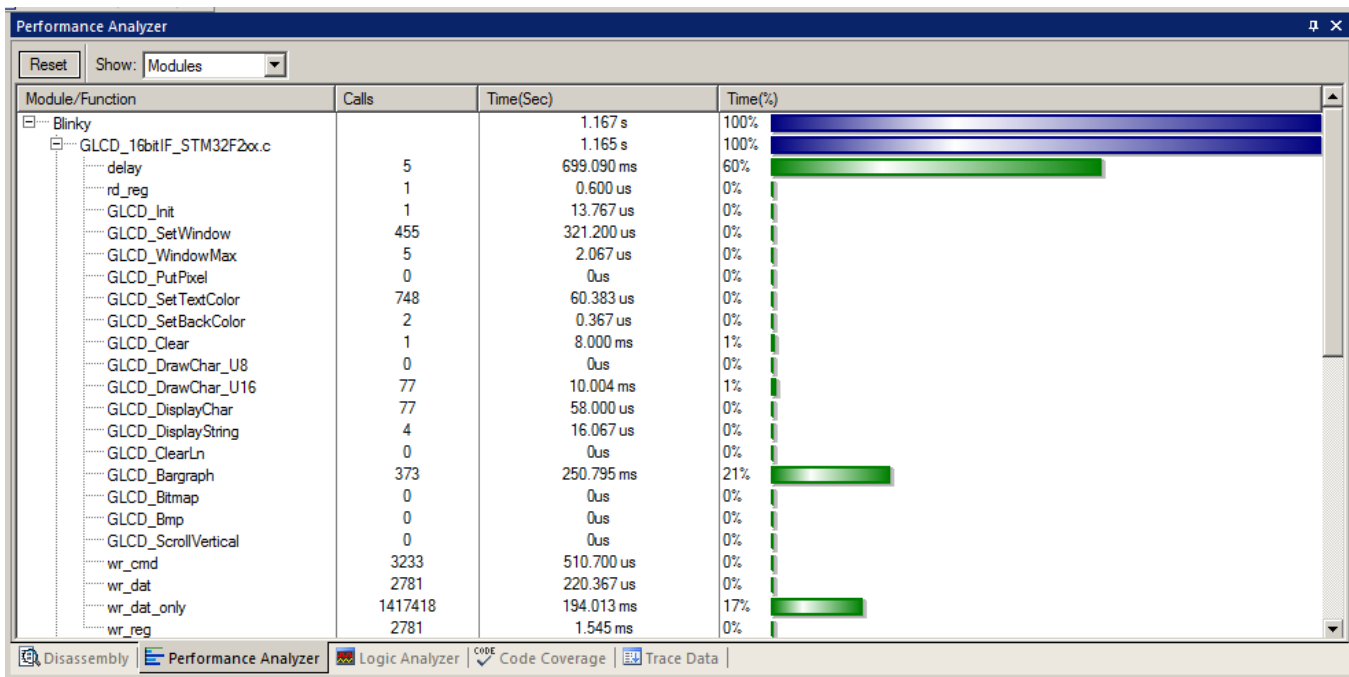



#### 4) Performance Analysis (PA): *For reference only...*

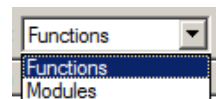
Performance Analysis tells you how much time was spent in each function. The data can be provided by either the SWV PC Samples or the ETM. If provided by the SWV, the results will be statistical and more accuracy is improved with longer runs. Small loops could be entirely missed. ETM provides complete Performance Analysis. Keil provides only ETM PA.


Keil provides Performance Analysis with the  $\mu$ Vision simulator or with ETM and the ULINK $pro$ . SWV PA is not offered. The number of total calls made as well as the total time spent in each function is displayed. A graphical display is generated for a quick reference. If you are optimizing for speed, work first on those functions taking the longest time to execute.

1. Use the same setup as used with Code Coverage.
2. Select View/Analysis Windows/Performance Analysis. A window similar to the one below will open up.
3. Exit Debug mode and immediately re-enter it.  This clears the PA window and resets the STM32 and reruns it to main() as before. Or select the Reset icon in the PA window to clear it. Run the program for a short time.
4. Expand some of the module names as shown below.
5. Note the execution information that has been collected in this initial short run. Both times and number of calls is displayed.
6. We can tell that most of the time at this point in the program has been spent in the GLCD routines.



7. Click on the RUN icon. 
8. Note the display changes in real-time while the program Blinky is running. There is no need to stop the processor to collect the information. No code stubs are needed in your source files.
9. Select Functions from the pull down box as shown here and notice the difference.
10. Exit and re-enter Debug mode again and click on RUN. Note the different data set displayed.
11. When you are done, exit Debug mode.



**TIP:** You can also click on the RESET icon  but the processor will stay at the initial PC and will not run to main(). You can type **g, main** in the Command window to accomplish this.

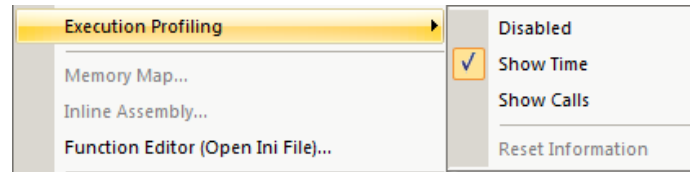
When you click on the RESET icon, the Initialization File .ini will no longer be in effect and this can cause SWV and/or ETM to stop working. Exiting and re-entering Debug mode executes the .ini script again.



## 5) Execution Profiling: *For reference only...*

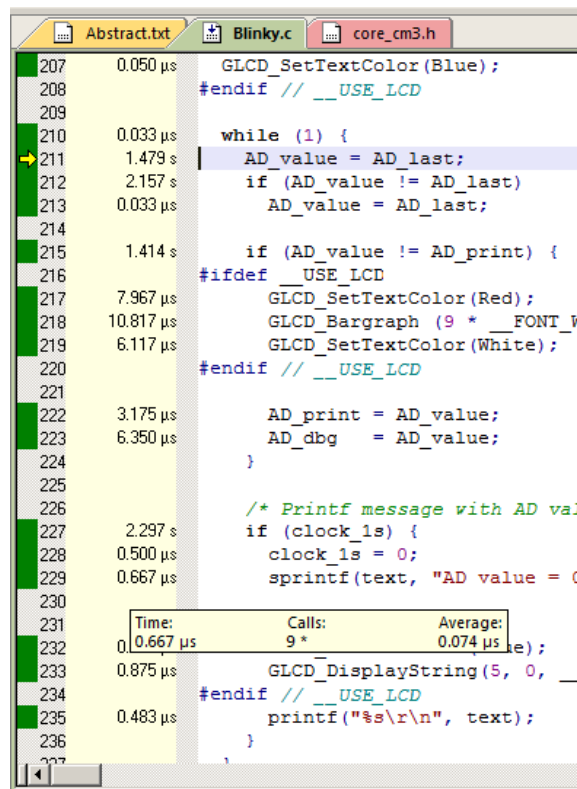
Execution Profiling is used to display how much time a C source line took to execute and how many times it was called. This information is provided by the ETM trace. It is possible to group source lines (called collapse) to get combined times and number of calls. This is called Outlining. The  $\mu$ Vision simulator also provides Execution Profiling.

1. Enter Debug mode.
2. Select Debug/Execution Profiling/Show Time.
3. In the left margin of the disassembly and C source windows will display various time values.
4. Click on RUN.
5. The times will start to fill up as shown below right:
6. Click inside the yellow margin of Blinky.c to refresh it.
7. This is done in real-time and without stealing CPU cycles.
8. Hover the cursor over a time and ands more information appears as in the yellow box here:



Time:	Calls:	Average:
19.599 s	139910257 *	0.140 $\mu$ s

9. Recall you can also select Show Calls and this information rather than the execution times will be displayed in the margin.



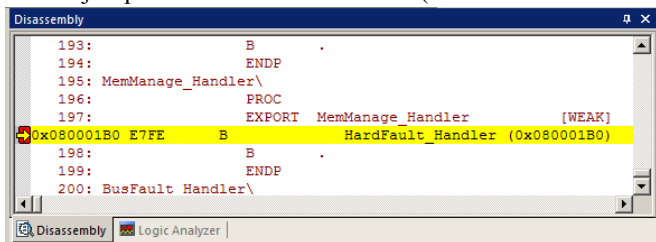
## 6) In-the-Weeds Example: *For reference only...*


Some of the hardest problems to solve are those when a crash has occurred and you have no clue what caused this. You only know that it happened and the stack is corrupted or provides no useful clues. Modern programs tend to be asynchronous with interrupts and RTOS task switching plus unexpected and spurious events. Having a recording of the program flow is useful especially when a problem occurs and the consequences are not immediately visible. Another problem is detecting race conditions and determining how to fix them. ETM trace handles these problems and others easily and is not hard to use.

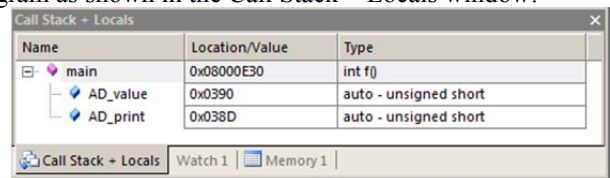
If a Hard Fault occurs, the CPU will end up at the address specified in the Hard Fault vector located at 0x00 000C. This address points to the Hard Fault handler. This is usually a branch to itself and this Branch instruction will run forever. The trace buffer will save millions of the same branch instructions. This is not useful. We need to stop the CPU at this point.

This exception vector is found in the file startup\_stm32f4xx.s. If we set a breakpoint by double-clicking on the Hard Fault handler and run the program: at the next Hard Fault event the CPU will jump to the Hard Fault handler (in this case located at 0x0800 01B0 as shown to the right) and stop.

The CPU and also the trace collection will stop. The trace buffer will be visible and extremely useful to investigate and determine the cause of the crash.



1. Open the Blinky\_Ulp example, rebuild, program the Flash and enter Debug mode. Open the Data Trace window.
2. Locate the Hard fault vector near line 207 in the disassembly window or in startup\_stm32f4xx.s.
3. Set a breakpoint at this point. A red block will appear as shown above.
4. Run the Blinky example for a few seconds and click on STOP.
5. Click on the Step\_Out icon  to go back to the main() program as shown in the Call Stack + Locals window:
6. In the Disassembly window, scroll down until you find a POP instruction. I found one at 0x0800 1256 as shown below in the third window:
7. Right click on the POP instruction (or at the MOV at 0x0800 124E as shown below) and select Set Program Counter. This will be the next instruction executed.
8. Click on RUN and immediately the program will stop on the Hard Fault exception branch instruction.
9. Examine the Data Trace window and you find this POP plus everything else that was previously executed. In the bottom screen are the 4 MOV instructions plus the offending POP.
10. Note the Branch at the Hard Fault does not show in the trace window because a hardware breakpoint does execute the instruction it is set to therefore it is not recorded in the trace buffer.



```
0x08001248 F1A40401 SUB    r4, r4, #0x01
0x0800124C DCFD      BGT    r0, r9
0x0800124E 4648      MOV    r0, r9
0x08001250 4631      MOV    r1, r6
0x08001252 462A      MOV    r2, r5
0x08001254 4643      MOV    r3, r8
0x08001256 E8BD9FF0 POP    {r4-r12, pc}
0x0800125A 0000      MOVN  r0, r0
      _scatterload:
```

Trace Data			
Display:	All	in	All
Time	Address / Port	Instruction / Data	Src Code / Trigger Addr
	X: 0x08000DD4	LDRB r1, [r0, #0x00]	
	X: 0x08000DD6	CBZ r1, 0x08000DE0	
1.215 414 190 s	X: 0x08000DE0	MOV r0, #0xFFFFFFFF	return -1; /* Conv. in pro...
1.215 414 210 s	X: 0x08000DE4	BX lr	
	X: 0x0800124E	MOV r0, r9	
	X: 0x08001250	MOV r1, r6	
	X: 0x08001252	MOV r2, r5	
	X: 0x08001254	MOV r3, r8	
1.215 420 300 s	X: 0x08001256	POP {r4-r12, pc}	

The frames above the POP are a record of all previous instructions executed and tells you the complete program flow.

## 21) Serial Wire Viewer and ETM Trace Summary:

### Serial Wire Viewer 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 SWV by definition.

### Serial Wire Viewer displays in various ways:

- PC Samples.
- Data reads and writes.
- Exception and interrupt events.
- CPU counters.
- Timestamps.

### ETM Trace 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).
- Plus - don't have to stop the program. Crucial to some projects.

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

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

For information on Instruction Trace (ETM) please visit [www.keil.com/st](http://www.keil.com/st) for other labs discussing this.

## 22) Useful Documents:

1. **The Definitive Guide to the ARM Cortex-M3** by Joseph Yiu. (he also has one for the Cortex-M0) Search the web.
2. **MDK-ARM Compiler Optimizations: Appnote 202:** [www.keil.com/appnotes/files/apnt202.pdf](http://www.keil.com/appnotes/files/apnt202.pdf)
3. **Lazy Stacking and Context Switching Appnote 298:** [www.arm.com](http://www.arm.com) and search for Lazy Stacking.
4. **A list of resources is located at:** <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.
5. **ARM Infocenter:** <http://infocenter.arm.com> [www.arm.com/cmsis](http://www.arm.com/cmsis) [forums.arm.com](http://forums.arm.com)

## 23) Keil Products and Contact Information:

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

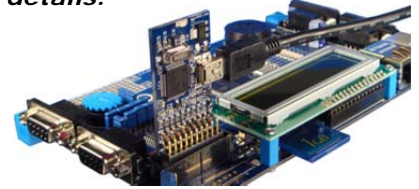
- MDK-Lite™ (Evaluation version) \$0
- **NEW !!** MDK-ARM-CM™ (for Cortex-M series processors only – unlimited code limit) - \$3,200
- MDK-Standard™ (unlimited compile and debug code and data size) - \$4,895
- MDK-Professional™ (Includes Flash File, TCP/IP, CAN and USB driver libraries) \$9,500

*For special promotional pricing and offers, please contact Keil Sales for details.*

### USB-JTAG adapter (for Flash programming too)

- ULINK2 - \$395 (ULINK2 and ME - SWV only – no ETM)
- ULINK-ME – sold only with a board by Keil or OEM.
- ULINKpro - \$1,250 – Cortex-Mx SWV & ETM trace.

*MDK also supports ST-Link and Segger J-Link Debug adapters.*



The Keil RTX RTOS is now provided under a Berkeley BSD type license. This makes it free.

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

Keil provides free DSP libraries for the Cortex-M3 and Cortex-M4.

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

All products are available from stock.

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

Call Keil Sales for special university pricing. Go to [www.arm.com](http://www.arm.com) and search for university to view various programs and resources.

MDK supports STM32 Cortex-M3 and Cortex-M4 processors. Keil supports many other ST processors including 8051, ARM7, ARM9™ and ST10 processors. See the Keil Device Database® on [www.keil.com/dd](http://www.keil.com/dd) for the complete list of STMicroelectronics support. This information is also included in MDK.

**Note: USA prices.** Contact [sales.intl@keil.com](mailto:sales.intl@keil.com) for pricing in other countries.

Prices are for reference only and are subject to change without notice.

For the entire Keil catalog see [www.keil.com](http://www.keil.com) or contact Keil or your local distributor.

For Linux, Android and bare metal (no OS) support on ST processors such as SPEAr, please see DS-5 [www.arm.com/ds5](http://www.arm.com/ds5).



### For more information:

**Keil Sales** In USA: [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)

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

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

For the latest version of this document and for more STMicroelectronics specific information, go to [www.keil.com/st](http://www.keil.com/st)

CMSIS Version 3: [www.arm.com/cmsis](http://www.arm.com/cmsis) or [forums.arm.com](http://forums.arm.com)

