Keil RTX RTOS the easy way:

by Robert Boys bob.boys@arm.com v 0.4

Introduction:

This note describes the process of operating ARM® Keil™ MDK toolkit featuring µVision®. Getting RTX running is described using the Serial Wire Viewer (SWV) debug technology to view RTX in operation. SWV is available on Cortex-M3/M4 processors and is supported by Keil. This note describes how to get all the components of this important technology working with µVision. **This article needs MDK® 4.11 or later.**

Keil µVision:

µVision is the IDE and is a component of the Keil MDK™ development system.

MDK-ARM™ components include the µVision IDE, ARM Realview® compiler, assembler and RTX™ RTOS.

The Keil ULINK® JTAG/SWD adapter family includes the ULINK2, ULINK-ME and the ULINK*pro*.

RL-ARM™ contains the sources for RTX plus a TCP/IP stack a FLASH file system, USB and CAN drivers.

Why Use Keil MDK ?

MDK provides these features particularly suited for Actel users:

1. µVision IDE with Integrated Debugger, Flash programmer and the RealView ARM compiler.
2. A full feature RTOS is included with MDK: RTX by Keil. No royalty payments are required.
3. Serial Wire Viewer trace capability is included.
4. RTX Kernel Awareness window. It is updated in real-time and uses no system resources.
5. Choice of USB adapters: ULINK2, ULINK-ME, ULINK*pro* and Segger J-Link.
6. Kernel Awareness for Keil RTX, CMX, Quadros and Micrium. All RTOSs will compile with MDK.
7. Keil Technical Support is included for one year. This helps you get your project completed faster.

Serial Wire Viewer:

Serial Wire Viewer (SWV) displays PC Samples, Exceptions (including interrupts), data reads and writes, ITM, CPU counters and a timestamp. This information comes from the ARM CoreSight™ debug module integrated into the Cortex-M3. SWV does not steal any CPU cycles, is non-intrusive and requires no stubs in your source code.

This document details these features:

1. Serial Wire Viewer (SWV).
2. Real-time Read and Write to memory locations for Watch, Memory and RTX Tasks windows.
3. Breakpoints and Watchpoints (Access Breaks).
4. RTX Viewer: a kernel awareness program for the Keil RTOS – RTX.
5. Keil TCP/IP stack – is one component of RL-ARM. A http server example **www.keil.com/download/docs/404.asp**.

Keil Contact Information: www.keil.com

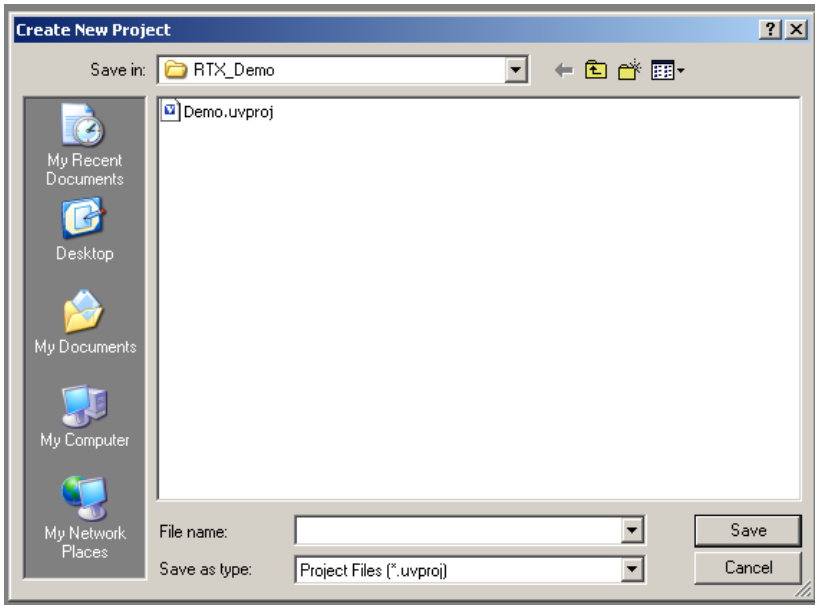
USA: North America:
Keil, An ARM Company
Plano, Texas
800-348-8051 (Toll Free)
sales.us@keil.com
support.us@keil.com

Europe and Asia:
Keil, An ARM Company
Grasbrunn, Germany
+49 89/456040-20
sales.intl@keil.com
support.intl@keil.com

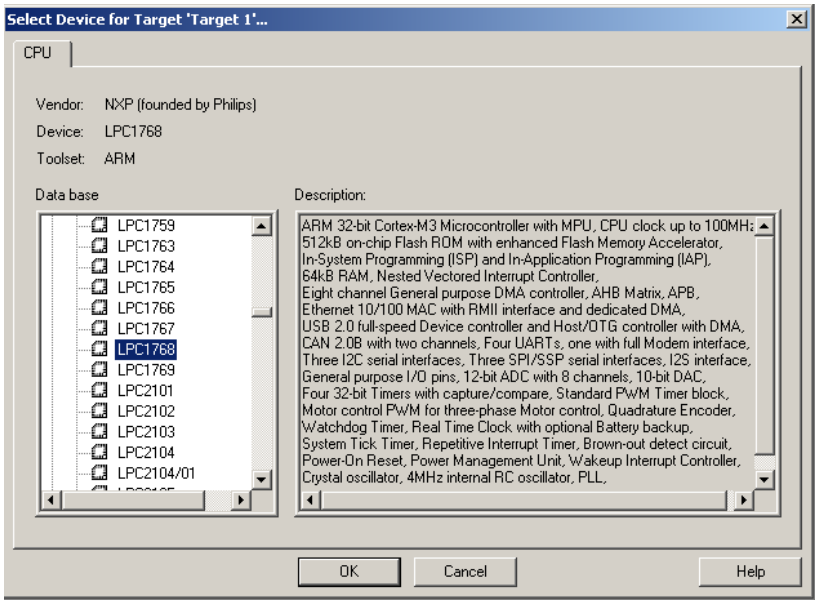
1) Create a new µVision project called RTX_Demo:

This exercise is designed to work with the NXP LPC1768 but will work with any Cortex-M3 processor that is listed in the Keil Device Database®. Unlisted devices can be run by selecting Cortex-M3 under "ARM".

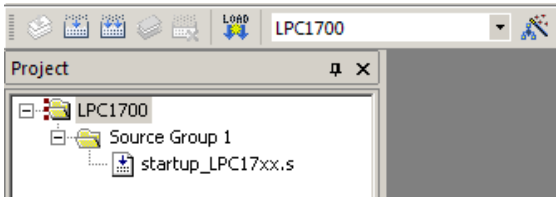
1. With µVision running and not in debug mode, select Project/New µVision Project.
2. In the window that opens up named Create New Project go to the folder C:\Keil\ARM\Boards\Keil\MCB1700.
3. Right click and create a new folder by selecting New/Folder. Name this folder RTX_Demo.
4. Double-click on the newly created folder "RTX_Demo" to enter this folder as is shown below.
5. Name your project "Demo".
6. Click on Save.



7. The "Select Device for Target 1" window shown here opens up.
8. This is the Keil Device Database which lists all the devices Keil currently supports.
9. Locate the NXP directory, open it and select LPC1768 or the Cortex-M3 processor of your choice. Note the device features are displayed
10. Click on OK.



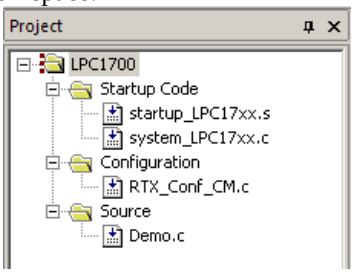
11. A window will open asking if you want to copy the default LPC17xx startup code to your folder and add it to the project. Click on "Yes". This will save you time.
12. In the Project workspace in the upper left hand of µVision, open up the folders by clicking on the "+" beside each folder.
13. We have now created a project called Demo located in the folder RTX_Demo and the target hardware called Target 1 with one source file: startup_LPC17xx.s.
14. Click once (carefully) on the name "Target 1" (or twice if not already highlighted) in the Project Workspace and rename Target 1 to LPC1700. Press Enter or click once on a blank part of the Project workspace to accept this. Note the Target selector also changes. Click on the + to open up the directory structure. You can create many target hardware configurations including a simulator and easily select them. You should se this structure now:



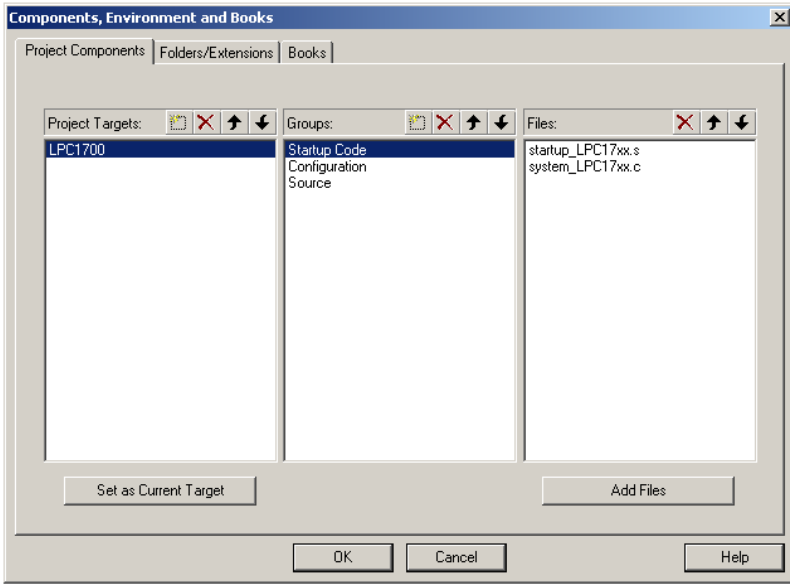
2) Select the project files:

1. Using MS Explore (right click on Windows Start icon), copy RTX_Conf_CM.c from C:\Keil\ARM\Startup to the C:\Keil\ARM\Boards\Keil\MCB1700\RTX_Demo folder
2. Copy system_LPC17xx.c from C:\Keil\ARM\Startup\NXP\LPC17xx to the C:\Keil\ARM\Boards\Keil\MCB1700\RTX_Demo folder.
3. In the Project workspace in the upper left hand of µVision, right-click on "LPC1700" and select "Add Group". Name this new group "Configuration" and press Enter.
4. Make another group and call it Source in the same manner.
5. Click twice on Source Group 1 and rename it Startup Code.

6. Right-click on "Startup Code" and select **Add files to Group "Startup Code…**".
7. Select the file system_LPC17xx.c and click on Add and then Close. This will show up in the Project workspace.
8. Right-click on "Configuration" and select **Add files to Group "Configuration…**".
9. Select the file RTX_Conf_CM.c and click on Add and then Close.
10. Open File/New (or Ctrl-N) and a new blank file will open up in µVision.
11. Open File/Save As and enter Demo.c. Click on Save. This is your empty source file to contain your main function.
12. Right-click on "Source" and select **Add files to Group "Configuration…**".
13. Select the file Demo.c and click on Add and then Close.
14. This concludes setting up the project structure and assigning files to various Groups. You should have this more complex structure in your Projects workspace:



15. Click on Project/Manager and select Components, Environment…. And the window below opens up. This is where you can manage your project structure. You can add and delete Targets, Groups and files as desired. Click on OK to return to the main menu.



2) Configure µVision for RTX:

Select the Keil Simulator:

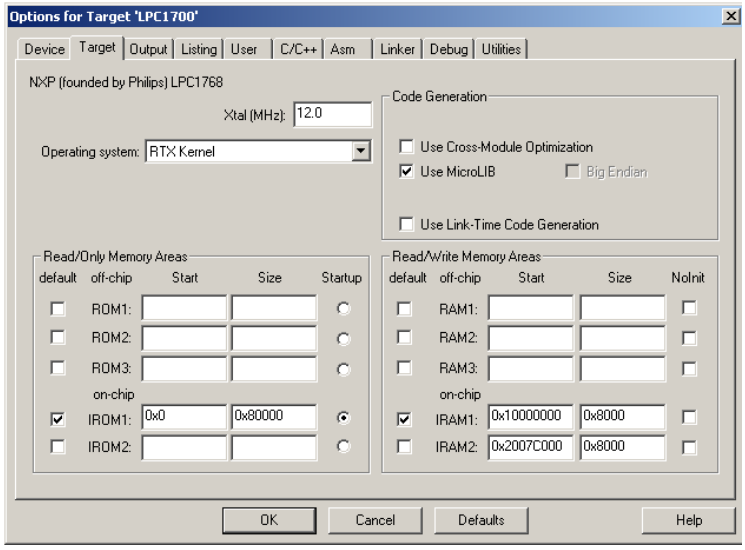
1. Select Options For Target  and select the Debug tab. This is where you can select the Keil Simulator or a debug adapter to connect to a real target. We will start out by using the simulator so no target hardware is required.
2. Select the simulator by checking "Use Simulator".
3. Do not close this window yet.

Tell µVision we are using RTX:

1. Click on the Target tab. In the Operating System drop down box, select RTX as shown below.
2. Select Use MicroLib. This selects a compiler mode that will result in much smaller code size.

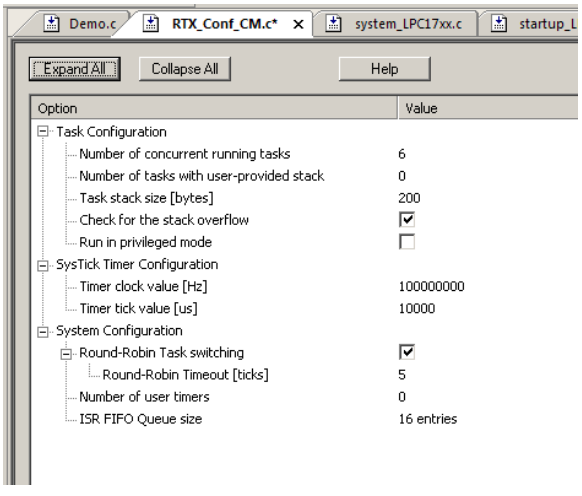
Note: The memory areas below are where you can set up RAM and ROM for a real target processor. µVision will use this information to create a scatter file for you if "Use Memory layout from target Dialog" in the Linker tab is selected.

3. All other configuration items in the Options for Target window can be left at their default.
4. Click on OK to close the Options for Target window.
5. Select File/Save All.



µVision configuration is now complete:

µVision is now completely configured for RTX operation. All we need is to tell RTX how fast the CPU is running (for timing information) and to put some source code into Demo.c.



Tell RTX how fast the CPU is running:

1. Open File/Open and select the file RTX_Conf_CM.c.
2. Click the Configuration Wizard tab at the bottom.
3. Click on Expand All and this window will open up:
4. Change SYSTICK Timer Configuration Timer clock value to 100000000 (8 zeroes). Timer tick should be 10000.
5. Note the other items you can change here.

Tip: If you double-click on a filename in the Project workgroup window, you will place that file in the µVision main window and open it. After a file is in the main window, you can select it by clicking on its tab.

Tip: Any changes you make here will be reflected in the code in the Text Editor tab and vice versa. You must rebuild your files in order for any changes to have effect.

3) Creating the Demo.c Source file to create the RTX program:

A) Description of the program:

1. We will create a simple three task program using RTX.
2. Task1 and Task2 will toggle two respective global variables.
3. The init task will be used to create task1 and task2 and then it will self delete and disappear.
4. Task1 and Task2 will run sequentially in Round Robin mode forever. Tasks are created as standard C functions. The __Task keyword tells the compiler to not create an entry and exit code for the tasks. RTX will do this.

B) Entering the source code: (Note: comments are optional) Remember, a***ll C source code is case sensitive !***

1. **Open the file Demo.c** you created. Either by clicking on its tab or by double-clicking on it in the Project window.
2. **Add these lines:**

```
#include <RTL.h>                    /* RTX header file */
#include <LPC17xx.H>              /* LPC17xx definitions (or your own part) */
```

3. **Create two global variables:**

```
unsigned int counta = 0;          /* counta and countb used in RTX demo */
unsigned int countb = 0;
```

4. **Create your Task 1:**

```
__task void task1 (void) { /* __task is a RTX keyword. */
for (;;) {                        /* Infinite loop – runs while task1 runs. */
      counta++;                   /* Increment global variable counta. */
      }
}
```

5. **Create your Task 2:**

```
__task void task2 (void) {
for (;;) {
      countb++;
      }
}
```

6. **Create the Init Task:** This task's job is to create the other two tasks that will actually run. We will call it init.

```
__task void init (void) {
      os_tsk_create (task1, 1);        /* Creates task1 with priority 1 (default)*/
      os_tsk_create (task2, 1);        /* Creates task2 with priority 1 (default)*/
      os_task_delete_self ();          /* Goes away and task1 starts */
}
```

7. **Create the main() function:**

```
      int main (void)  {
      SystemInit();                    /* initialize the Coretx-M3 processor */
      os_sys_init(init);               /* Start the init task */
}
```

8. **Select File/Save All. This saves all your source files and the project for complete recall later.**

All you need to do now is to compile the files, enter debug mode and run it !

This is a working minimal RTX project. This is Round Robin and each task will run for 50 msec before it switches to the next task that is ready and has a same or higher priority. The switching time can be changed in the file RTX_Conf_CM.c as Round Robin Timeout (ticks). You can see that it is really easy to get RTX working.

So, let's get RTX running and see how it works.

3) Compiling Demo.c and the other source files:

1. Click on the Rebuild icon  to compile the sources. Must compile and link with no errors or warnings.
2. Enter Debug mode by clicking on its icon. 
3. Start the program by clicking on the RUN icon. 
4. Note: you can stop the program with the STOP icon.  Leave it running.

Tip: You are currently using the simulator therefore you do not need to use the Load icon  to program any memory. This is also true if you are running in RAM in a real target and have configured this properly. You enter debug mode directly.

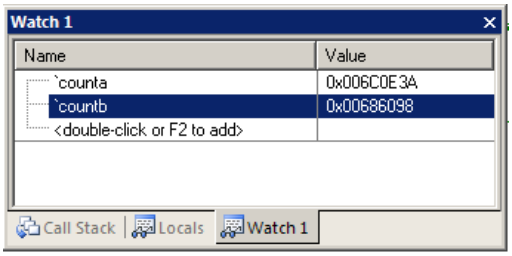
If you want to run the program in Flash and have configured it properly, you will need to use the Load icon  or have the Update Target before Debugging selected in the Flash programmer configuration window before entering debug mode.

4) µVision Features useful to view and control RTX operation:

A) Watch window:

We will use the watch window to monitor the two global variables we created: counta and countb.

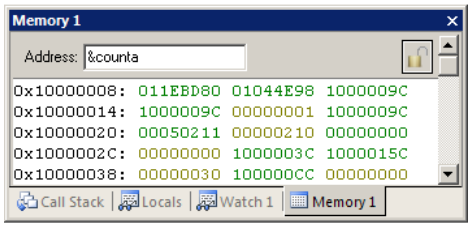
1. Open a Watch window if it is not already open: open View/Watch Window and select Watch 1.
2. Locate counta in Demo.c and block it using the mouse. Click on counta and drag it into Watch 1 and release.
3. Similarly, drag countb into Watch 1.
4. The values of counta and countb will alternatively increment as each of its respective task runs as shown here:
5. Double-click on counta or countb while it is incrementing. Enter 0 and press the Enter key. You can modify Watch or Memory window values on the fly without stealing any CPU cycles. This works with the simulator or on a real Cortex-M target processor.



Note: We are using the simulator here. On my computer the tasks change state every 5 or 6 seconds. This helps us view RTX in slow motion. On a real target RTX will run much faster.

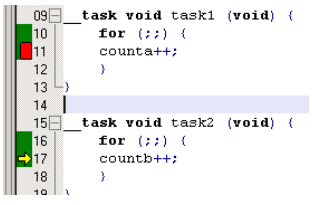
B) Memory window:

1. Open Memory 1 (if not already open) by selecting View/Memory windows and select Memory 1.
2. Drag counta into this Memory window (or enter it manually).
3. Note the value in counta is used to point to a physical memory address. This is useful working with pointers.
4. Add and ampersand "&" in front of counta and now you will see the contents of the variable counta.
5. Right click in the Memory window and select Unsigned and then Long.
6. Now counta is displayed as a 32 bit number and so is countb since it is adjacent in memory to counta.



C) Hardware Breakpoints: (there are usually 6 on a Cortex-M3)

1. Set a hardware breakpoint by double-clicking next to the line incrementing counta in Demo.c. This will create a red box as shown here and presently the program will stop here once Task1 starts to run. The yellow arrow is the Program Counter contents.
2. Click on RUN several times and note counta increments each time.
3. Remove this breakpoint and set one on countb++. Click on RUN and note the program will stop once Task2 runs.
4. Remove this breakpoint and click on RUN.

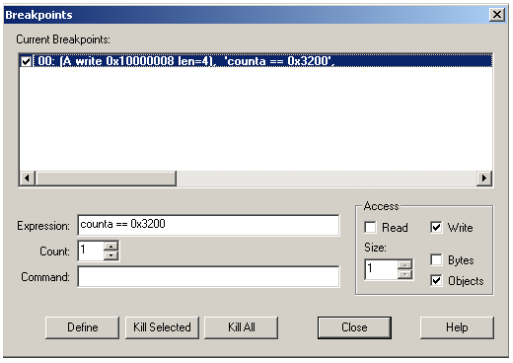


TIP: You can add variables to the Watch and Memory windows while the program is running. You can also set breakpoints "on the fly" as well as many other µVision features. This is also true when using a real target processor.

D) Watchpoints: (also called Access Breaks)

It is possible to stop the program when a variable equals a user specified value. This is very useful during debugging.

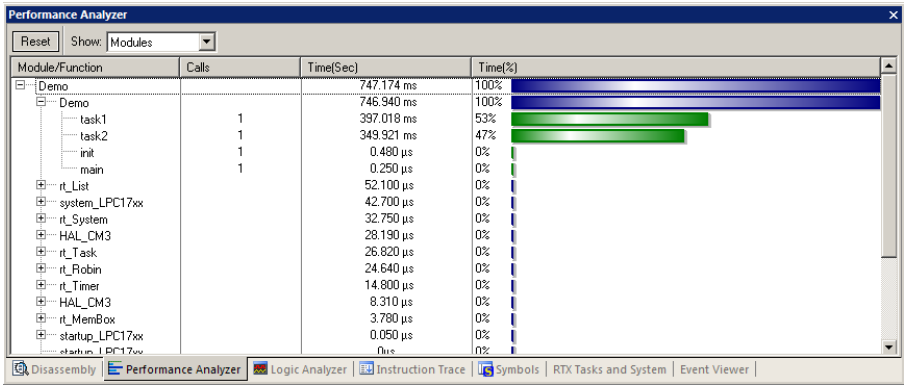
1. Stop the program by clicking on the STOP icon.
2. Open Debug/Breakpoints or press Ctrl-B.
3. Enter "counta == 0x3200", and select "Write" as shown here. Click on Define to move the Watchpoint to the upper area. Click Close.
4. Set counta = 0x0 in the watch window. Click on RUN.
5. Presently the processor will stop when count equals 0x3200.
6. Open the Breakpoint window and select Kill All to remove the Watchpoint. Select Close.
7. Click on RUN for the next example.



E) Performance Analyzer:

The Performance Analyzer (PA) tells you where your program is spending its time. This can alert you to "time hogs". PA is available only with the Keil Simulator or the ULINK*pro* adapter with an ETM trace equipped processor such as LPC1768.

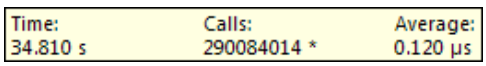
1. Select View/Analysis Windows and select Performance Analyzer.
2. Open Demo + and then the next Demo + to see the window displayed here:
3. If you select Functions in the Show: box, Task1 and Task2 are treated as functions.
4. With Show: set to Modules, stop the processor and click on the RESET icon.
5. Click on RUN and watch the changes as various modules are run. Note the various statistics gathered.

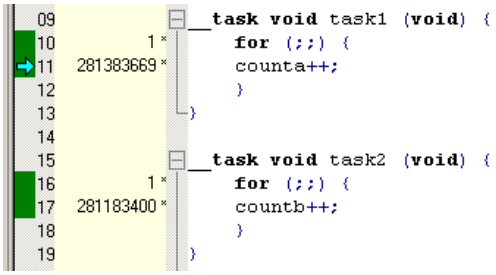


F) Execution Profiling (EP):

Execution Profiling displays how many times a function has been called or the total time spent in the function. EP works with the simulator or with the ULINK*pro* with an ETM trace equipped processor such as LPC1768 or many STM32 parts.

1. Select Debug and select Execution profiling and either Show Times or Show Calls. An extra column opens in the source and disassembly window as shown to the right:
2. Hold the cursor over a time or a call and a display window appears showing both calls and times as shown here:





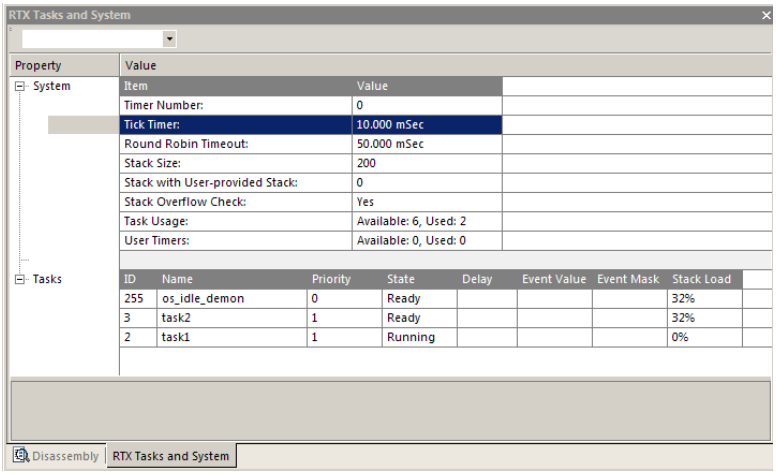
TIP: You can group times or calls by blocking the source text of interest.

Right click on this block and select Outlining. Various options are provided in this menu to configure this feature.

G) RTX Tasks and System window:

This is one of two RTX kernel awareness windows available. The information displayed is obtained from the Cortex-M3 DAP (Debug Access Port) which reads and writes memory locations through the JTAG or SWD ports nearly always non-intrusively. The Watch and Memory windows use this same ARM CoreSight technology.

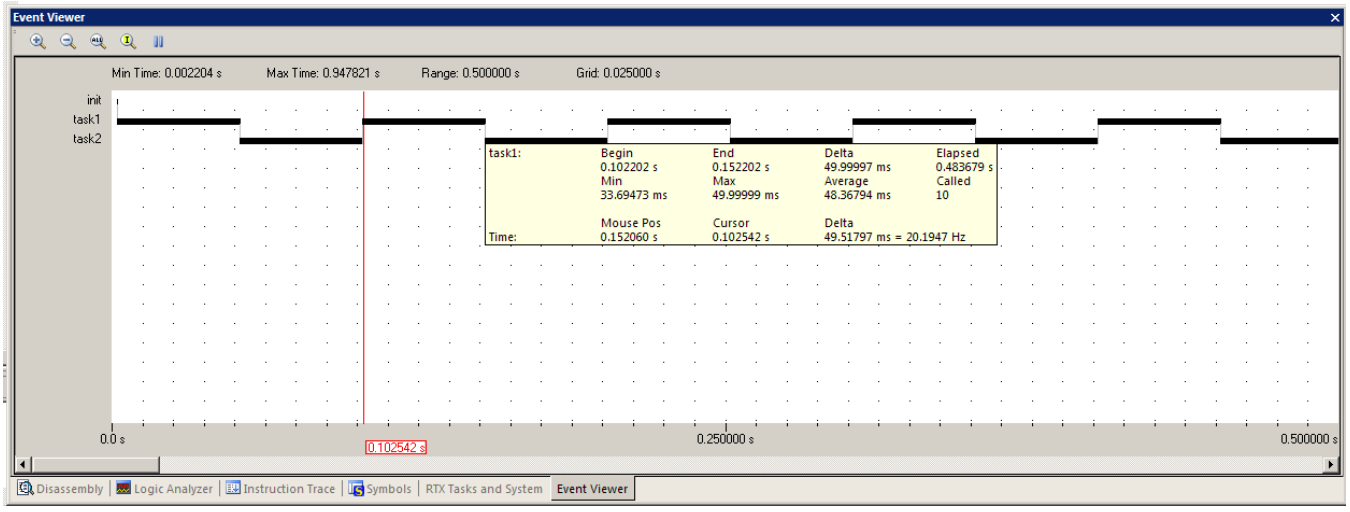
1. Open Debug/OS Support and select RTX Tasks and System. This window opens:
2. As RTX runs this will be updated.
3. Note the State of the tasks changing.
4. The information located in the Value area is derived from the file RTX_Conf_CM.c. Check the screen on page 4 and see similar items listed.



H) RTX Event Viewer

Event Viewer provides a graphical representation of how long and when individual tasks run. Event Viewer runs in the simulator and when on a target processor it uses Serial Wire Viewer (SWV). SWV must be properly configured.

1. Select Debug/OS Support and select Event Viewer. No configuration is need for the simulator.
2. Click on the All icon and then use + and – to get a display similar to that below. You might need to wait a few seconds to pass through a few task state changes. Event Viewer updates automatically while the program is running.
3. Note the init task ran once which is what we would expect
4. Hold the mouse the mouse over a task bar and the displayed task information is shown. In this case task1 is running and lasts for about 50 msec. This is result of the Timer .
5. Click somewhere to create the red cursor and additional information concerning elapsed times is displayed in Time:
6. If you change the value of the Round Robin Timeout value (5 is default) in RTX_Conf_CM.c and rebuild your project and run it again, these times will be different. This is an interesting experiment to try. Afterwards, make sure you return the timeout value back to 5. The Round Robin Timeout is equal to the number stated times the tick value found in RTX_Conf_CM.c. The tick value in Cortex-M3 processors is usually 10 msec by default.



5) Explanation of Demo.c Source Code:

At this point you have a minimal RTX program running. RTX is capable of many more features and we will start to evaluate them. First, here is a description of what Demo.c is doing:

1. Program starts running at the main() function.
2. The Cortex-M3 system is initialized.
3. Task init is then called.
4. init creates task1 and task2 and then deletes itself. init will not run again except after a system RESET.
5. init starts the first task which is task1.
6. task1 runs forever. We have included no code to tell it to pass control to task2.
7. After 50 msec, the Round Robin Timeout forces task1 to end and task2 to start.
8. After another 50 msec, task2 is forced to stop and task1 runs again. This continues forever.

Tasks of equal priority will be run in Round Robin when they enter the ready state. Higher priority tasks will preempt (or interrupt) a running task. Lower priority tasks will not run.

It might be more intuitive if Round Robin Timeout was called Round Robin Task Switching Time. There are other ways to switch tasks, but first, let's talk a bit about the Idle Demon (or perhaps more correctly Idle Daemon).

6) Idle Demon

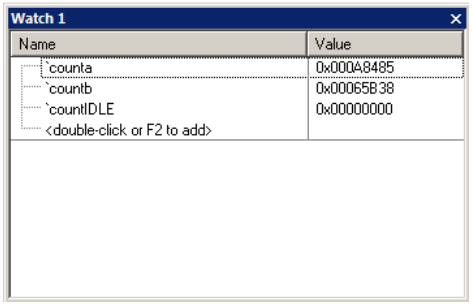
If no tasks are ready to run then RTX will execute its idle demon. This is located in the file RTX_Conf_CM.c. You can insert user code to run during this idle time. We will put a variable into the idle demon so we can detect when it is being executed. The idle demon itself is created automatically by the RTX kernel.

1. Stop  the program if it is running.
2. Near the top of file RTX_Conf_CM.c insert this global variable: unsigned int countIDLE = 0;
   Just after the #include <RTL.h> line is a good place.
3. In the idle demon code (which starts near line 141 in RTX_Conf_CM.c) enter: countIDLE++;
4. Your code will look like this:

```
__task void os_idle_demon (void) {
  /* The idle demon is a system task, running when no other task is ready */
  /* to run. The 'os_xxx' function calls are not allowed from this task.  */

  for (;;) {
  /* HERE: include optional user code to be executed when no task runs.*/
  countIDLE++;
  }
}
```

5. Exit debug mode  and rebuild  the files. Re-enter debug mode.
6. Drag countIDLE into the Watch 1 window or enter it manually.
7. Click on RUN  . The Watch 1 window will update but countIDLE will not increment. This means the Idle Demon is never executed. This window is shown here:
8. This variable will only increment while the Idle Demon is running and serves as a good test.
9. The Idle Demon is reserved priority 0 (the lowest).



TIP: You can edit the source files while in either Debug or Edit mode, but you must compile them while in Edit mode only.

7) Switching Tasks:

We have already seen how we can switch the tasks Round Robin based on time only. RTX has some other methods to switch tasks.

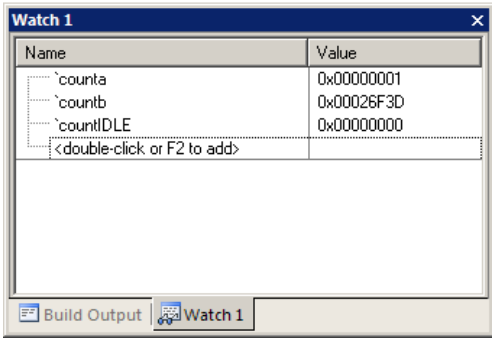
A) Cooperative Task Switching: os_tsk_pass();

When the Pass command is used the next ready task of the same priority is run.

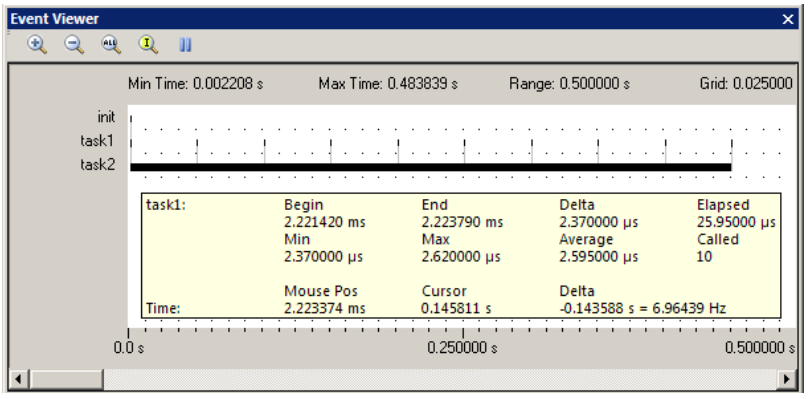
1. Stop the program if it is running.
2. Add the line `os_tsk_pass();` just after the `counta++;` line in Task1:

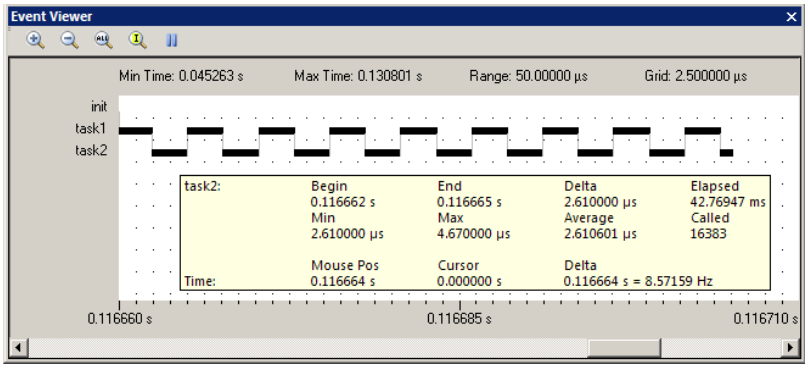
```
__task void task1 (void) {
        for (;;) {
        counta++;
        os_tsk_pass();
        }
}
```

3. Exit debug mode and rebuild the files. Re-enter debug mode.
4. Click on RUN.
5. Note the variable counta increments by one and passes control to Task2 which increments until the timeout occurs.
6. Then Task2 times out and control passes back to Task1 and the process repeats forever.
7. The Event Viewer will also show the different timings. As shown below, Task1 now runs for a mere 2.37 usec.
8. If you add `os_tsk_pass();` to Task2 in the same way, task2 will run for only one cycle and passes control to Task1 and the cycle repeats.



9. The bottom Event Viewer shown below illustrates counta and countb incremented only once and then control is passed to the next ready task. This continues Round Robin forever.
10. The variables in the Watch window also increment equally now.
11. Delete the line `os_tsk_pass();` for the next exercise.





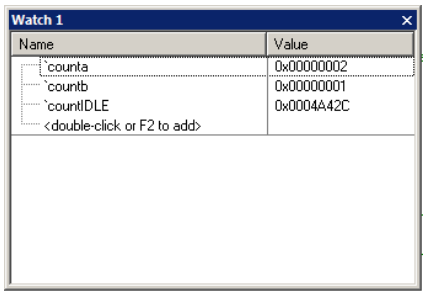
B) Delay: os_dly_wait (ticks);

The Delay function pauses the calling task by the amount of ticks passed as the argument. Control will switch to the next task ready else passes to the idle demon. After the specified number of ticks has expired, the calling task will be placed in the ready state. The delay does not use up processing time with a loop.

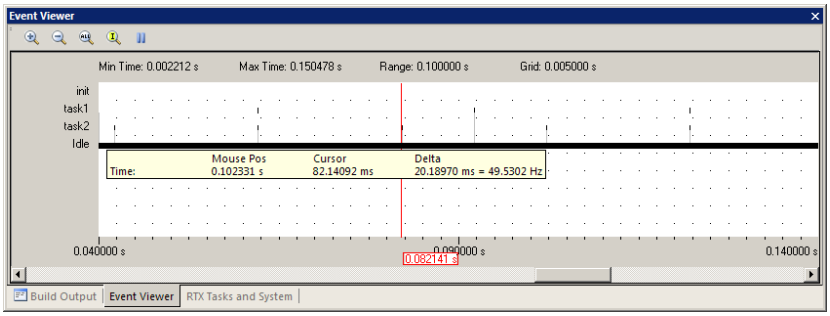
1. Stop the program if it is running.
1. Add the line os_dly_wait (3); just before the line `counta++;` in Task1:
2. Add the line os_dly_wait (2); just before the line `countb++;` in Task2:
3. Note: make sure you have removed `os_tsk_pass();` from the previous exercise.
4. Exit debug mode and rebuild the files. Re-enter debug mode.
5. Click on RUN.

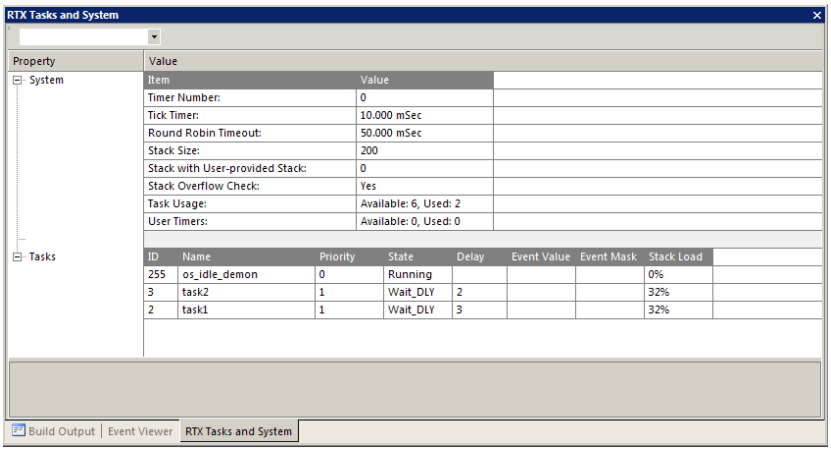
What is Happening:

1. In the Watch window note that counta and countb increment but much more slowly than before. Note that RTX has automatically created the Idle Demon task. Most of the CPU time is spent in the idle demon.time. You can use the Performace Analyzer to confirm this.
2. Task1 is paused for 2 ticks (20 msec) and is put in the Wait_Dly state. This is shown as Wait_Dly state box in the Tasks and System window.
3. If Ready, Task2 runs and countb is incremented once, if not Ready, Idle demon runs.
4. Task2 is paused for 3 ticks (30 msec). This is shown as Dly_Wait in the Tasks and System window.
5. If Ready, Task1 runs and countb is incremented once, if not, the Idle demon runs.



1. Open the Event Viewer and using the All, + and – icons adjust the range to get a window similar to this one:
2. In the Event Viewer, place a cursor on a task2 tick and hold the cursor on the next task2 event. The time Delta will be displayed: in this case 20 msec (2 * 10 msec).
3. Repeat on Task1 and note the time delta is 30 msec. This confirms the operation of os_dly_wait for times of 2 and 3 ticks.
4. Run the program while viewing these windows to gain an understanding of how the tasks are switched.





TIP: A task with an os_dly_wait can also wait for an event to be passed to it.

5. When done, remove the two instances of `os_dly_wait` for the next exercise.

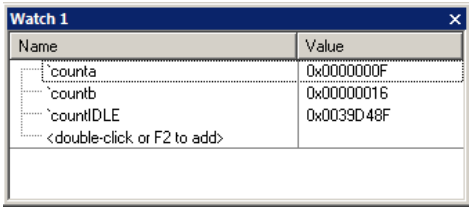
C) Periodic Task Execution: os_itv_set (ticks) and os_itv_wait ();

This function determines a periodic wake-up interval with os_itv_set. Then we put the task to sleep with os_itv_wait.

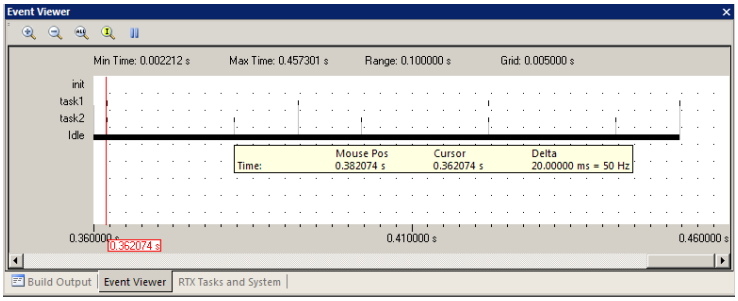
2. Stop the program if it is running.
3. Add the line os_itv_set (3); just before the line `for (;;) {` in Task1:
4. Add the line os_itv_set (2); just before the line `for (;;) {` in Task2:
5. Add the line os_itv_wait (); just before the line `counta++;` in Task1:
6. Add the line os_itv_wait (); just before the line `countb++;` in Task2:
6.
7. Note: make sure you have removed `os_tsk_pass(); and os_dly_wait` from the previous exercises.
8. Exit debug mode and rebuild the files. Re-enter debug mode.
9. Click on RUN .

What is Happening:

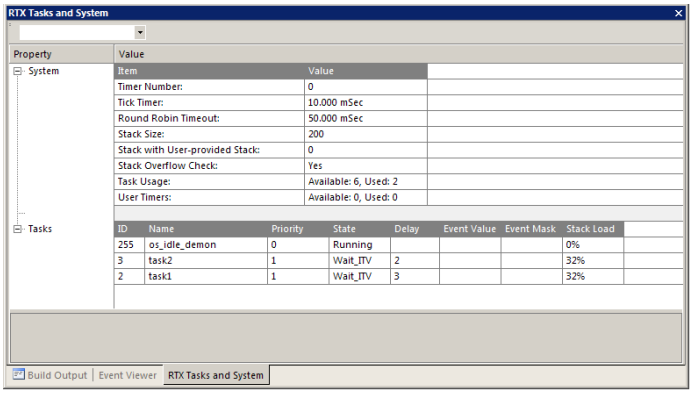
1. In the Watch window note that counta and countb increment but much more slowly than before. Task1 runs fewer times (counta increments slower than countb) than Task2.
2. Note that RTX has automatically created the Idle Demon task. Most of the CPU time is spent in the idle demon.time
3. Task1 is put to sleep for 3 ticks (30 msec) and is put in the Wait_ITV state as shown in the state box in the Tasks and System window.
4. If Ready, Task2 runs and countb is incremented once, if not Ready, Idle demon runs.
5. Task2 is put to sleep for 2 ticks (20 msec). This is shown as Wait_ITV in the Tasks and System window.
6. If Ready, Task1 runs and countb is incremented once, if not, the Idle demon runs.



7. Open the Event Viewer and using the All, + and – icons adjust the range to get a window similar to this one:
8. In the Event Viewer, place a cursor on a task2 tick and hold the cursor on the next task2 event. The time Delta will be displayed: in this case 20 msec (2 * 10 msec).
9. Repeat on Task1 and note the time delta is 30 msec. This confirms the operation of os_itv_set for times of 2 and 3 ticks respectively.



10. Run the program while viewing these windows to gain an understanding of how the tasks are switched.
11. Remove the 4 instances of functions os_itv_set and os_itv_wait for the next execise.



…to be continued…