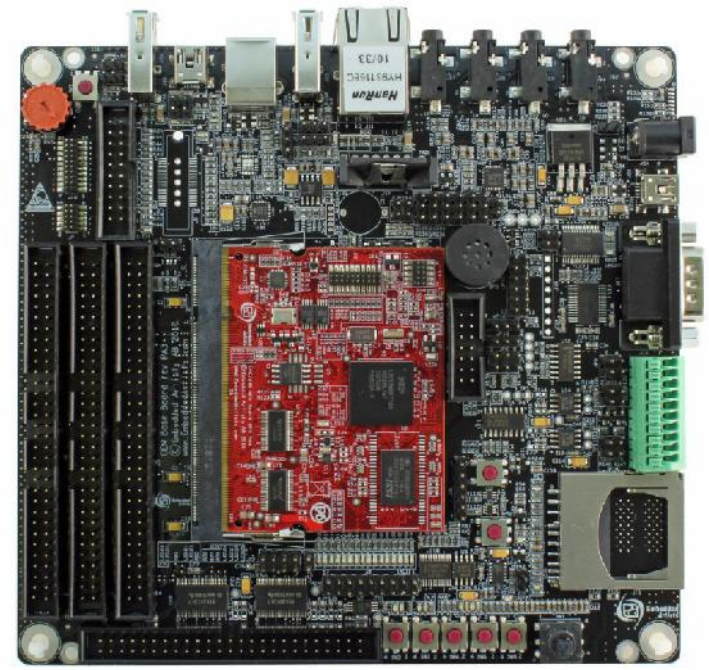
µTrace setup guide for Embedded Artists LPC4088 development board



LAUTERBACH
DEVELOPMENT TOOLS

About this document

This document will explain how to get up and running with the Lauterbach µTrace unit and the Embedded Artists LPC4088 development board. The document partners with a zip file containing the examples and configuration scripts described herein.

Two examples are provided: one runs a simple loop on the Cortex-M3 core in the device; the other runs the FreeRTOS demo program provided as part of the sample code from Embedded Artists.

Pre-Requisites

The Lauterbach TRACE32 software for µTrace has been installed. It is assumed that this has been installed to the default location of **`C:\T32_uTrace`**. It will be referred to as $T32SYS in the rest of this document. No changes were made from the default jumper settings.

TRACE32 Build 48187 or later is required for the LPC4088 device. This can be found by selecting **`About TRACE32...`** from the **`Help`** menu. You will see something like Figure1. The version has been highlighted in this image.

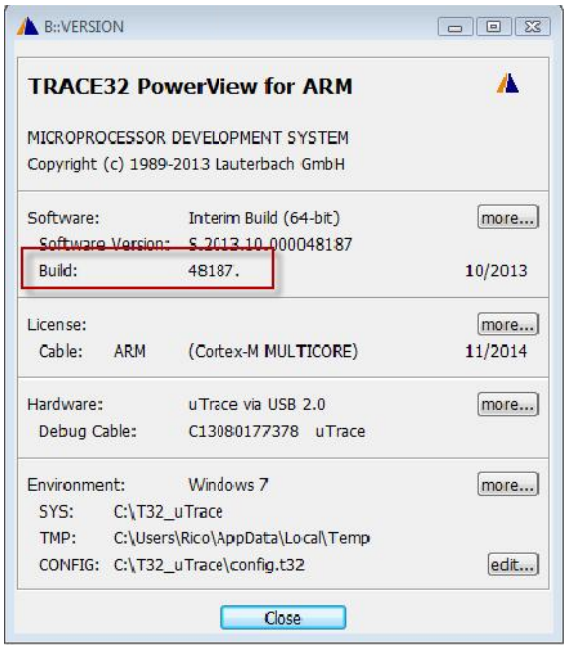


Figure 1: TRACE32 Version

You may also need a FLASH update package to include support for the LPC40xx devices. You will need the **`$T32SYS\demo\arm\flash\lpc40xx.cmm`** file dated Monday 21st October 2013 or later. The date can be found by looking at the end of the comments section (around line 90) of the file.

This version does not yet support the peripheral register view. Only the core Cortex-M3 registers are available.

If your version is older than this or you do not have the FLASH programming scripts please contact your local Lauterbach representative who can arrange an update for you.

Setup Procedures

Unzip the archive **EA_LPC4088.zip** so that it over-writes files in the $T32SYS directory. You may wish to take a backup of this directory beforehand.

Connect the CombiProbe header to Socket A on the µTrace.

Connect the CombiProbe header (using the MIPI34-MIPI20T adapter) to socket J10 on the main board (see Figure 2). Take care to make sure that the connector is correctly aligned. The header on the board is not keyed. Pin 1 is furthest from the red dial.

Figure 2: J10 on main board

In tests here, the board was powered with a 5V external power supply connected to J24, not the mini USB connector at J25.

Power on the µTrace and then power on the LPC4088 board.

Start the TRACE32 software and you should see something like Figure 3.

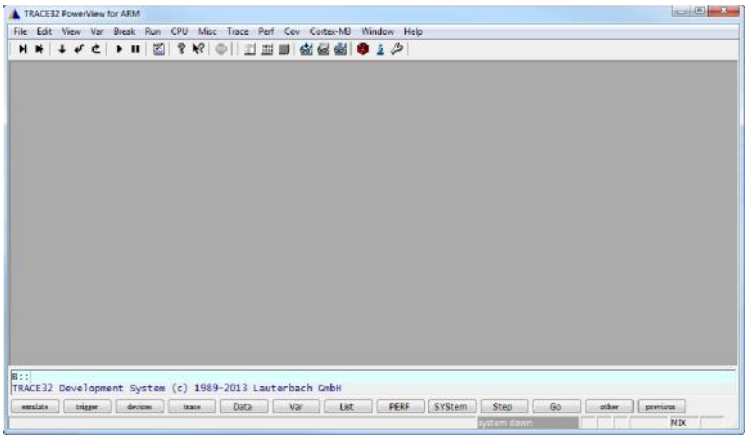


Figure 3: TRACE32

From the **File** menu, select **Run Batchfile...**, browse to $T32SYS and select **startup.cmm**. You should see a window which looks like Figure 4. Detailed use of this script is beyond the scope of this document but more information can be found in the accompanying **Startup Guide.pdf**.

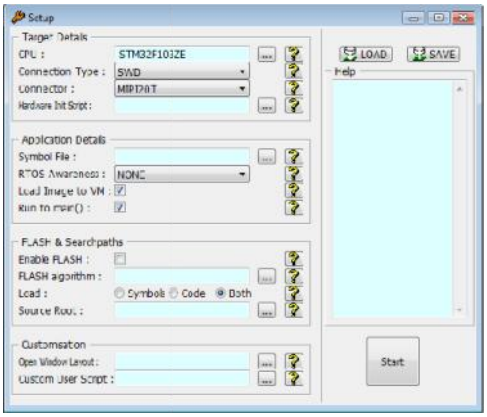


Figure 4: startup.cmm

TRACE32 can be configured to call this script each time you start it up. To do this, edit the **t32.cmm** file in the $T32SYS directory. At the bottom of the file, just before the line that reads "ENDDO" add a line which reads

```
do startup.cmm
```

For now, click the LOAD button and browse for the **basic_demo.t32ini** file which is located in

```
$T32SYS\Eval Boards\Embedded Artists\LPC4088
```

directory. This will populate some of the fields in the startup window. Click the big start button to launch the demo. The target will be initialised, a small application will be downloaded and some basic windows opened so that TRACE32 will now look like Figure 5.

This will give you JTAG control of the target. The connection is via the Serial Wire Debug (SWD) interface to the Cortex-M3.

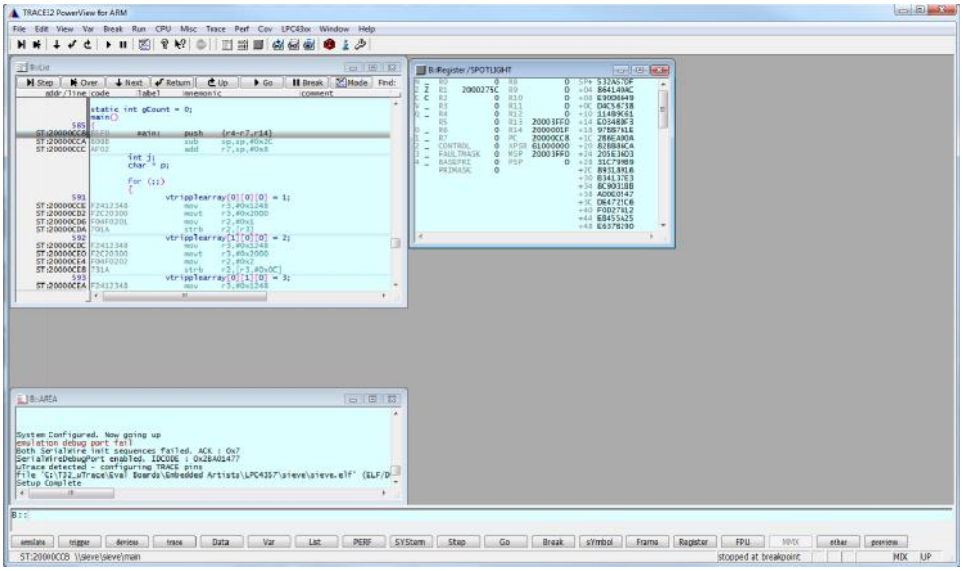


Figure 5: Demo loaded

Trace Setup

No modifications are required to the board to collect ETM trace data. The **hardware_init.cmm** script sets the correct pin multiplexing for the LPC4088 device to enable off-chip trace collection.

Care should be taken whenever the CPU clock exceeds 80MHz as the chip, by default, does not provide sufficient drive strength to the pins to get the trace data off-chip. It is recommended that the user study the **user_init.cmm** script in the **FreeRTOS** example directory for more information .

NXP officially rate the trace pins at 80MHz and suggest users try it for themselves above that. In tests with this board I have achieved 120MHz and others have reported up to 132MHz on other LPC4088 based designs.

FreeRTOS Example

The second example loads the FreeRTOS example program provided by Embedded Artists. The directory contains the executable and the **main.c** source file. For the purposes of the example, this is all that's required. Everything else can be downloaded from the Embedded Artists website, using the serial number that came with your board.

Connect the hardware and start TRACE32 as outlined in the previous example. Run the **startup.cmm** script and load the initialisation file located at:

$T32SYS\Eval Boards\Embedded Artists\LPC4088\freertos_demo.t32ini

Then click the big “start” button. You will eventually be presented with a display which looks like Figure 6. The list of tasks is blank because the scheduler hasn’t been started yet so no tasks have been created.

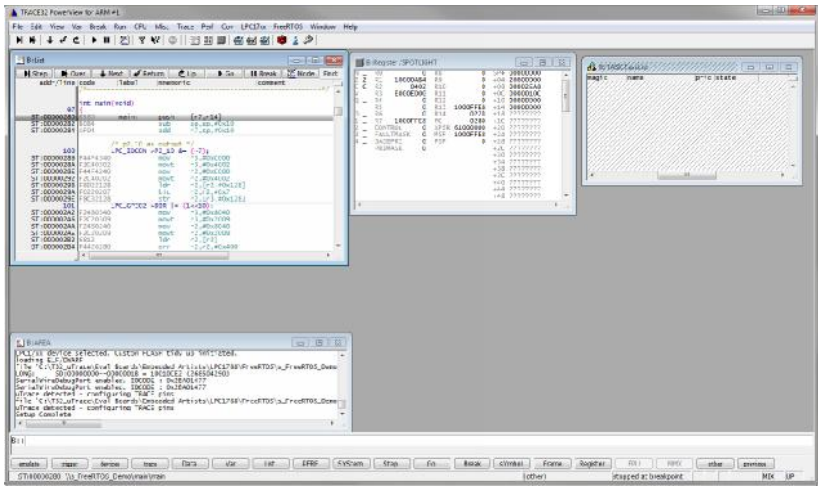


Figure 6: FreeRTOS demo Configuration

Where a window or view is hatched out like this it means that the data that would normally fill it is unavailable. During this example you may see the source view window look like this. This will be because the debugger is trying to display some of the source files that were used to build the example but they just aren’t there. If you click the **Mode** button in the Source View window you will be able to see the dis-assembly listing.

For now, let’s get some tasks launched. Select **Set…** from the **Break** menu and enter the text

```
prvQueueReceiveTask\12
```

in the address/expression box. This can be seen in Figure 7. Click the **OK** button and then start the target running. The system will halt when the receive task gets a message and the task listing window will now be populated. It will look like Figure 8.

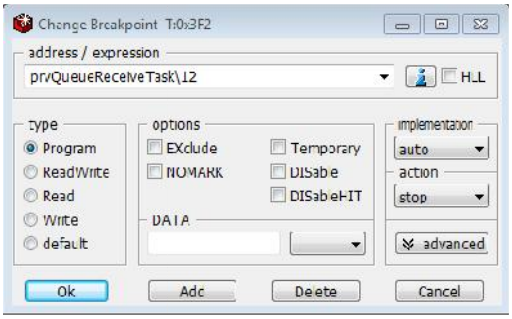


Figure 7: Set a breakpoint

For the breakpoint we could have used an absolute address or a symbol name but we used a combination of function name and source line offset from the start of the function.

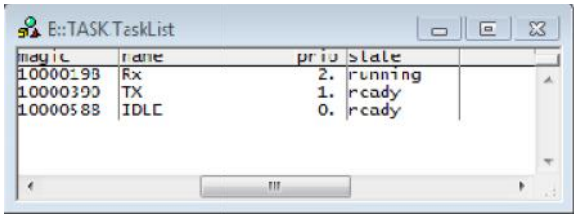


Figure 8: FreeRTOS task list

Tutorials

This section will provide some basic tutorials to help familiarise users with the TRACE32 concept.

Run Control

The target can be controlled via the buttons at the top of the **List** window or using the control buttons on the toolbar. See Figure T1. Users can also right-click on a line of code in any **List** window and select **Go Till** to run to a particular point. If you wish to run to known symbol the **GO** command can be entered on the command line. See Figure T2 for an example that will cause the debugger to run the target until the entry of function **func14** is reached.

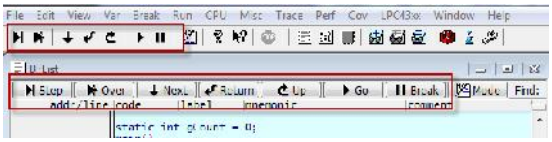


Figure T1: Run Control Buttons

Figure T2: The Go command

Breakpoints

Double-click a source line in any List window to set a default breakpoint. Right-click a line or variable for more control over a breakpoint. For even finer control of breakpoints select **Set...** from the **Break** menu - see Figure T3.

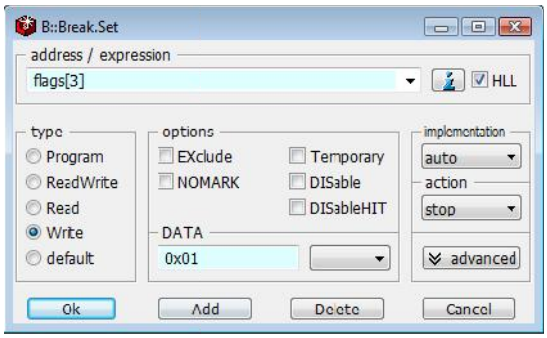


Figure T3: Break.set

Change the settings to match figure T3 and click OK. Start the target running and it will halt at line 681 where the first write of 1 to variable **flags[3]** occurs.

Task aware, conditional and counting breakpoints can all be set. Click the advanced button to access these extra settings. Discussion of these options is beyond the scope of this document but should be reasonably self-explanatory.

The Vector catch unit can be programmed by selecting the **OnChip Trigger...** option from the **Break** menu.

Registers

CPU registers can be viewed by using the **Register** command or by selecting **CPU Registers** from the **CPU** menu. This command can take a **/SPOTLIGHT** option which causes the last four sets of deltas to the window contents to be highlighted. See Figure T4. Double-click a register to change its contents or right-click for indirect views.

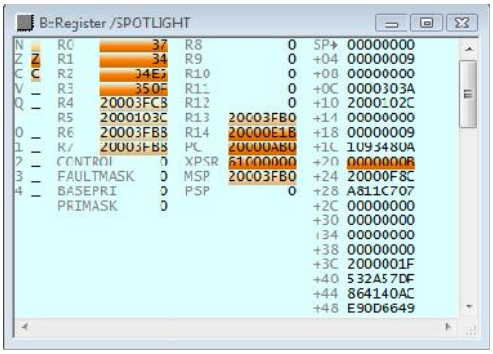


Figure T4: Highlighted Registers

The processor's peripheral control registers can be accessed via a dedicated menu which is dynamically added at runtime once the user has made their CPU selection. From here all of the different sub-systems can be selected. A global view can also be obtained by selecting **Peripherals** from the **CPU** menu. An example can be seen in figure T5. This window can also take the **/SPOTLIGHT** option to highlight any changes to the contents.

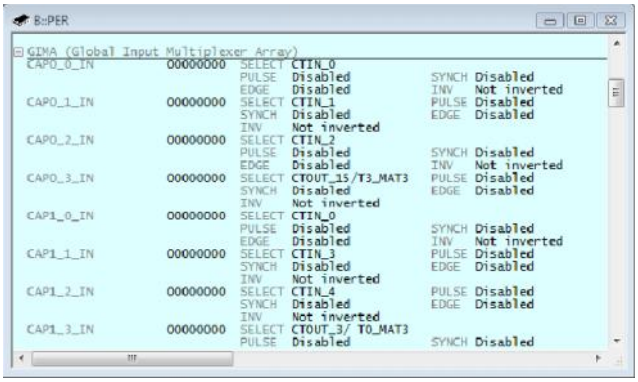


Figure T5: Peripheral Registers

A left-click on any of the registers or bit-fields will cause the address and bits to be displayed in TRACE32 status line.

A right-click on any of the bit-fields will pop up a menu with a list of allowable values.

Variables

Variables can have their value displayed by left-clicking them in any window. Variables can be dragged to a view or watch window. Local and global variables can be shown by selecting the appropriate options from the **View** or **Var** menus. Right-clicking a variable opens up a menu with a number of options for viewing it. A few are shown in figure T6. Variables can be displayed graphically, in tables, can be cast to other variable types. Memory can be cast to a variable type for display and there are special options for frame buffers, linked lists and waveforms.

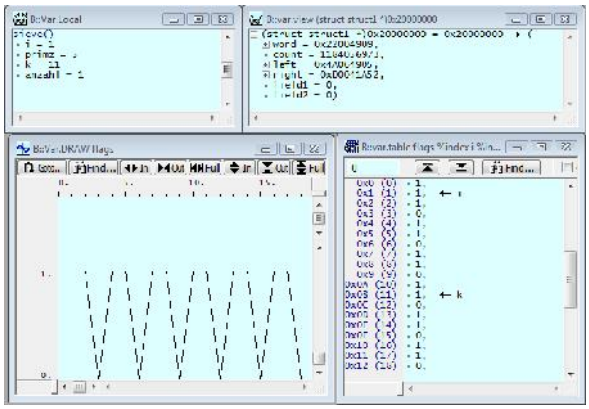


Figure T6: Variable Views

Macros can be created that will take variable values and convert them to real world values, such as volts from an ADC reading.

Try displaying the array **flags[]** in function **sieve** in a number of different ways.

```
go sieve
```

Right-click **flags** and look at the options under **other**.

Where the processor supports it and the debugger has been configured for dualport memory access variable values can be displayed and updated non-intrusively whilst the Cortex-M is executing code.

Memory

Memory can be viewed by selecting Dump... from the View menu. Enter the address to view and set any relevant options. A window will be displayed like figure T7. Memory can be searched for a pattern. Memory can be filled with a pattern or a test pattern. Two ranges can be compared or a CRC can be calculated for a given range. A walking bit test can also be performed over a memory region.

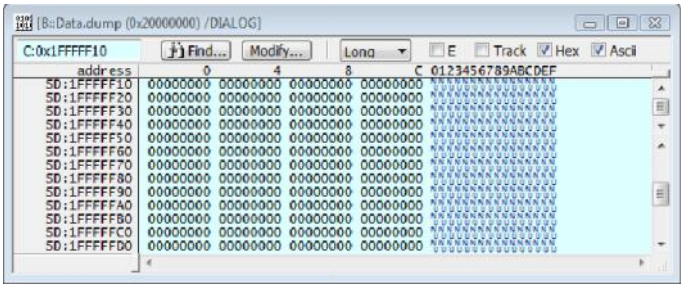


Figure T7: Memory Dump

All memory view windows can have the **/SPOTLIGHT** option added to them allowing the highlight of any changes in contents. Each value has a right-click menu behind it providing access to further options.

Try the following:

```
var.view flags /SPOTLIGHT
data.dump flags /dialog /SPOTLIGHT
```

Change the values in one window and the values in the other will be highlighted.

Performance Analysis

A sample based performance analysis capability is provided. This can be accessed by selecting **Perf Configuration...** from the **Perf** menu. An entire book could be written on this window alone so instead a few examples will be provided to get you started.

This is a sample based metric and may or may not be intrusive depending upon the core chosen. If the DWT in the chosen core supports the PC Snoop mode, the sampling will be made non-intrusively. This can be checked by opening the **Data Watchpoint and Trace** setting from the Peripheral Registers view and checking the availability of PCSAMPLEENA. See Figure T8. PC Snoop is available on all Cortex-M4 cores, all Cortex-M0+ cores, and all Cortex-M3 cores of r2p0 or newer.

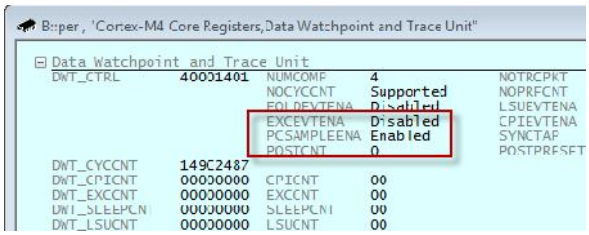


Figure T8: PC Snoop

If this is available non-intrusive metrics can be collected. Set the METHOD in the Perf window to **Snoop**.

If not, the target will need to be halted to read the Program Counter for the samples. Set the METHOD in the Perf window to **StopAndGo**.

To view relative function runtime analysis:

- Set the METHOD as described above
- Set the Mode to PC
- Set the state to OFF
- Click the ListFunc button
- Start the target running

An example is shown in figure T9.

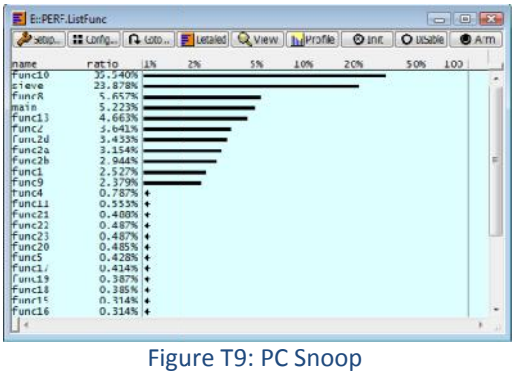


Figure T9: PC Snoop

To view data values:

Set the METHOD as described above
Set the Mode to Memory
Set the State to OFF
Set SnoopAddress to flags
Set SnoopSize to Long
Click the ListDistrib button
Start the target running

An example is shown in figure T10.

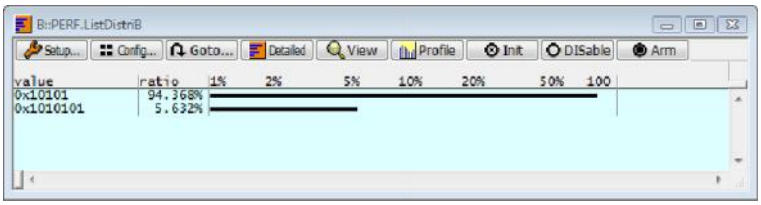


Figure T10: Data Snoop

On-Line Help

If Adobe Acrobat Reader is installed on your PC before you install TRACE32 for µTrace, a help plug-in will be automatically configured. Help can be accessed at any time by pressing the F1 key. Partially type a command and press F1 and after a few seconds wait the appropriate page of the documentation will be opened up in the Acrobat Reader. Click and window and press F1 and help for that window will be displayed. Additional help can be found on the Help menu, including a search capability and a target manual which describes in more detail the debug capabilities of the family of cores you are working with: No. breakpoints, non-intrusive memory access, dealing with watchdogs, etc.

There is an issue with Adobe Reader 10 which causes the right book to be opened but not the correct page to be displayed.

You can also contact your local Lauterbach representative if you have any questions about the operation of the µTrace unit or the TRACE32 software interface. A list can be found at:

http://www.lauterbach.com/tsupport.html

Trace Examples

All of the previous tutorials have been using the JTAG or SWD interface. The next batch will look at using the off-chip trace or ETM. The example scripts configure the trace port and pins but the board still needs to be modified as described on page 4 so that the trace signals are available for the µTrace to capture.

Basic Trace Collection

Select Configuration from the Trace menu. You should get a window like that in figure T11. Make sure that the METHOD is set to CAnalyzer and the state is set to OFF. Ensure that the AutoArm box is ticked. This allows tracing to start and stop as the target CPU starts and stops.

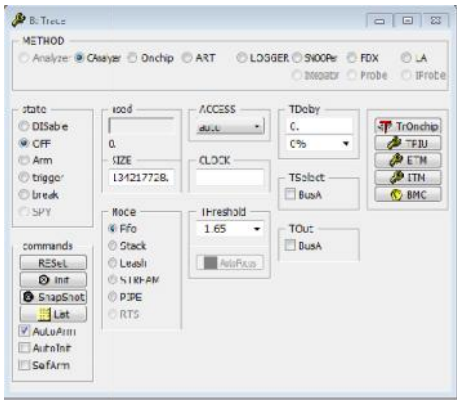


Figure T11: Trace Configuration

Start the target and let it run for a few seconds before stopping it again. There should be a blue bar in the **used** box to indicate the number of trace records captured. This should number in the tens or hundreds of thousand for a few seconds of run time. If it is less than a hundred or so you may need to check the resistor positioning as there appears to be no meaningful trace data.

Once you have some trace data captured, click the List button and see the program flow information. The window will look like figure T12. Click the **More** or **Less** buttons to filter the amount of information displayed in the window.

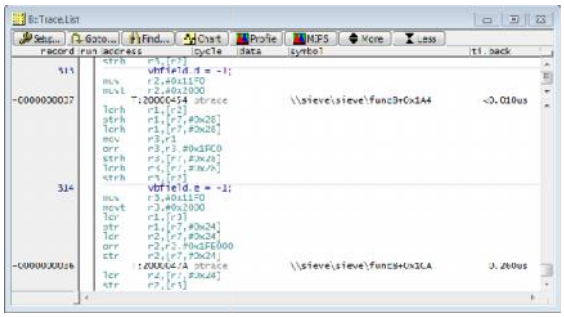


Figure T12: Program flow trace

The trace data can be searched. Click the **Find** button and enter the text “sieve” into the address/expression box. Then click the **Find All** button. This will show a window that looks like figure T13 with all occurrences of calls to the function **sieve** in the trace buffer. Clicking on any of these will cause the trace listing window to jump to that point in the buffer so you can see the program flow around that event.

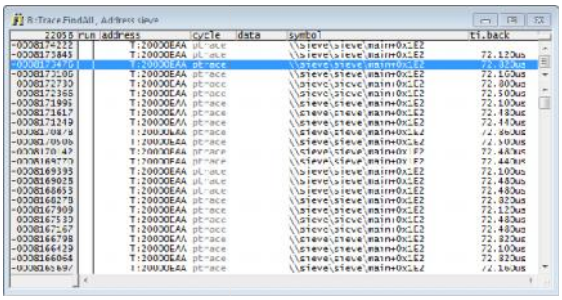


Figure T13: Search Results

The **ti.back** column in the search results window shows the time between function calls. It should average out at around 72.5us.

By default there is no data trace on the Cortex-M so data reads and writes will not be traced and cannot be searched for. However, if the DWT on your device supports it you can use a data breakpoint (up to four of them are allowed for in the Cortex-M specification but the actual number is core specific) to cause a data trace event to be injected into the trace stream. Care should be taken when doing this as data trace packets cannot be as easily compressed as the program flow trace packets and you may get an internal trace FIFO overflow and some data will be lost.

In the trace list window, click the Chart button to see a view of functions against time, similar to that in Figure T14. The example shown here has been zoomed in to show individual functions against the timeline on the horizontal axis.

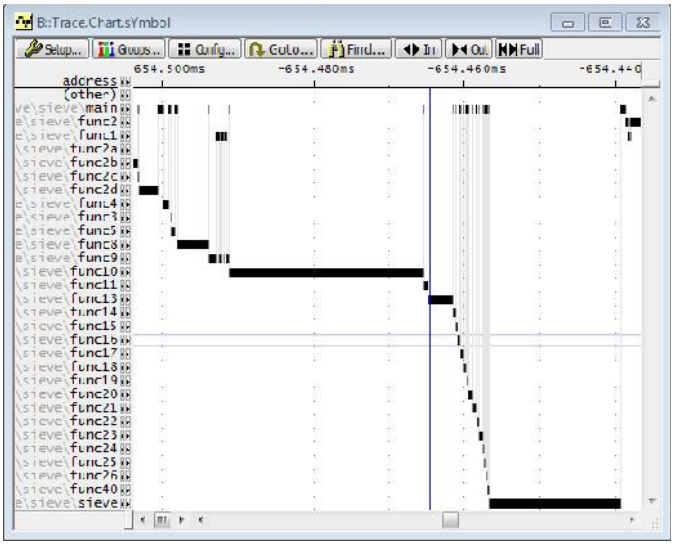


Figure T14: Trace Chart

The zoom can be controlled in a number of ways:

- Click the **In** and **Out** buttons
- Click on the chart and use the mouse scroll wheel to zoom in and out
- Click and drag to select an area of the chart and then left click within it to zoom it to the full size of the window
- Double-click on the chart but do not release the second mouse click. Move the mouse up and down to zoom in and out around the point clicked on.

Code Coverage

With program flow trace available it is easy to get code coverage information. Select **Add Tracebuffer** from the **Cov** menu. This will add the contents of the current trace buffer to the existing code coverage database. Like this multiple test runs can be aggregated. Now select **List functions** from the **Cov** menu. You should see a window like that in Figure T15.

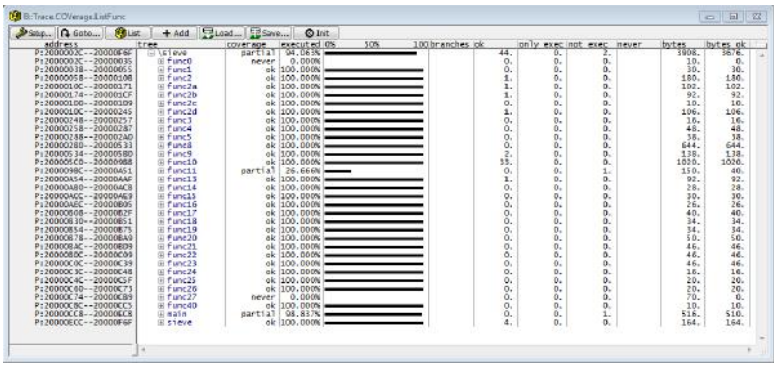


Figure T15: Function level code coverage

Any of the functions can be expanded by clicking on the "+" icon to see individual sources lines. Expand **func11**, which has only partial coverage, to see which lines haven't been covered. Double-click on line 438—438 to see more detail about that line which only has partial coverage. You should see a modified source view window similar to Figure T16. If you toggle the Mode button to switch to High Level Language (HLL) view you will see that only the default case in the switch statement has ever been executed.

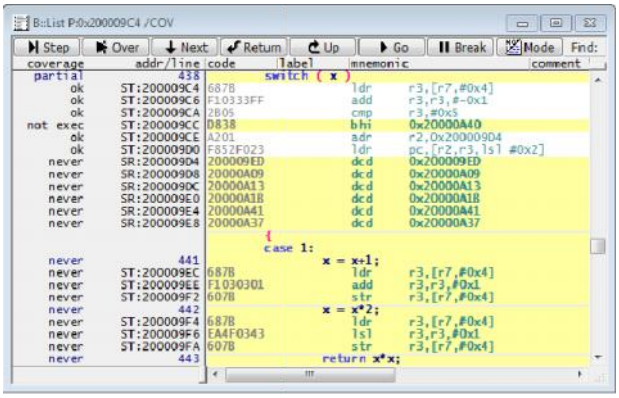


Figure T16: Low level code coverage

Performance Analysis

Collect some trace data and then from the **Perf** menu select **Function Runtime** and then **Show Detailed Tree**. You will get something similar to Figure T17. Figure T17 has had some of the irrelevant columns removed from the display so that it more easily fits this page.

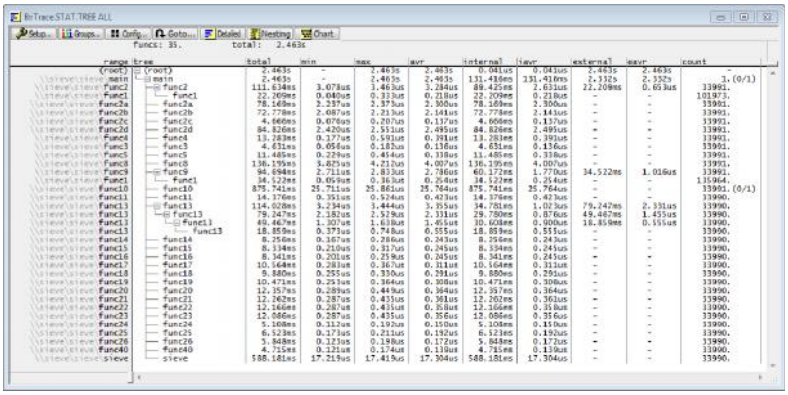


Figure T17: Detailed Performance Measurements

For the sample period, this view shows the minimum, maximum and mean time spent in each function. It also shows time consumed by any sub-functions and the number of times each function was called.

Each function has a right-click menu on it to provide more detailed analysis of call trees, runtimes and distance between calls to a function. Try some of the options and see what you can learn about this code. For example, right-click **func1** and select **Linkage** or **Parents** from the menu.

Trace Based Debugging

Collect some trace and then select **CTS Settings** from the **Trace** menu. You will see a window like that shown in Figure T18. Change the state to ON. This will take anything from a few seconds to several minutes to process, depending upon how much trace data you have sampled and how fast your connection to the µTrace unit is.

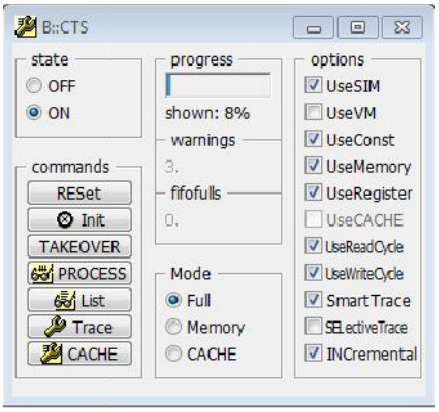


Figure T18: CTS Settings

When it has finished processing, the buttons in any **List** windows will become yellow and some new buttons will be added:

- Step back over
- Step back into
- Go back to Entry
-

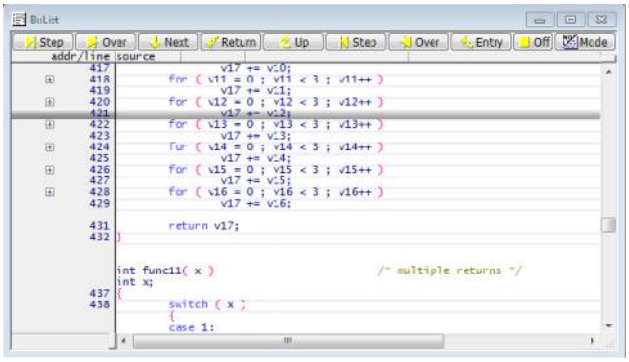


Figure T19: New Run Control Buttons

See Figure T19 for this. This allows users to step and run backwards and forwards through a reconstruction of the system context at any point during the period that was sampled into the trace buffer. You can inspect the contents of memory, registers and variables as far as they can be reconstructed. Where something cannot be reconstructed it's value will be replaced with '????'.

Additional entries in the **List** window's right-click menu will be added:

- Go Till
- Go Back Till

Clicking the **List** button in the CTS window will open a different view of the trace data, showing function nesting and function and instruction runtimes. Each of the functions can be expanded. Where data values can be reconstructed the change in variable will be shown at each step. An example can be seen in Figure T20.

A chart view can be constructed from this window by clicking the **Chart** button.

Right-clicking any line of code in the **CTS.List** window or right-clicking anywhere in the **CTS.Chart** window will cause a menu to popup. Select Set CTS and all windows will be updated to reflect the state of the target as reconstructed at that point in history. Using this allows users to quickly zone in the actual cause of a bug rather than just trapping on the subsequent error caused by the bug.

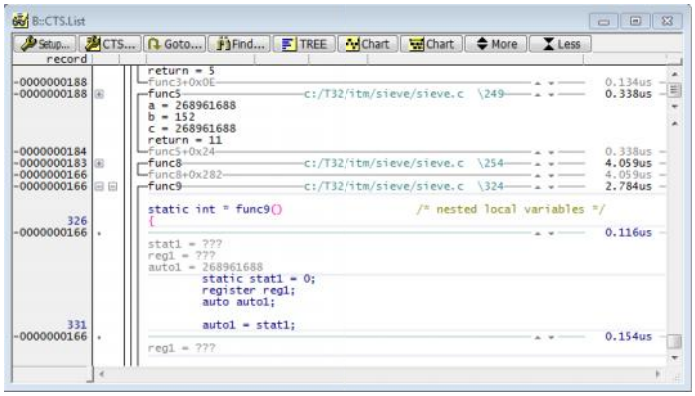


Figure T20: CTS.List view

Before 'normal' debugging can be resumed the CTS mode must be exited. The can be done by setting the state to OFF in the CTS window or by entering the command **CTS.OFF**.

RTOS Aware Examples

General

When a Kernel awareness package has been loaded a system specific menu is added to the debugger. This contains new menu options which allow Kernel or system objects to be displayed. This could be a list of task, semaphores, events or message queues. It could be stack usage or special symbol loading operations. Generally, whatever makes sense for the target RTOS; not all RTOSes support all features.

More detail on your chosen RTOS and the interaction with the TRACE32 software can be found in the **$T32SYS\pdf** directory. Look for the **rtos_<your_chosen_rtos>.pdf** file. For example: **rtos_freertos.pdf**.

Task Listing

Select **Display Tasks** from the **FreeRTOS** menu. This will show a list of all tasks on the system and a summary of their state. An example can be seen in Figure T21. There is always a right-click menu on the number in the magic column for each task. Double-clicking the magic number will open a window showing the task control block.

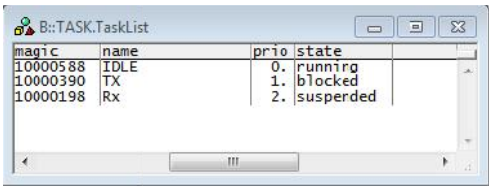


Figure T21: FreeRTOS Task List

Message Queues

Selecting **Display Queues** from the **FreeRTOS** menu will prompt you to browse for a message queue variable. Once you have selected this a window will open showing the status of the queue and any tasks blocked on it. An example can be seen in Figure T22. Again, there is a right-click menu on the "magic" number for each queue.

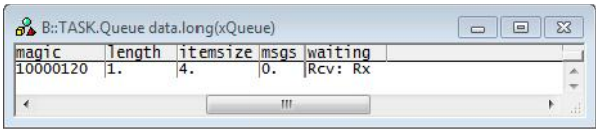


Figure T22: FreeRTOS Message Queue

Display Stack Usage

The Kernel Awareness can provide details of stack usage for each task. Select:

FreeRTOS->Stack Coverage->List Stacks

for a display similar to Figure T23.

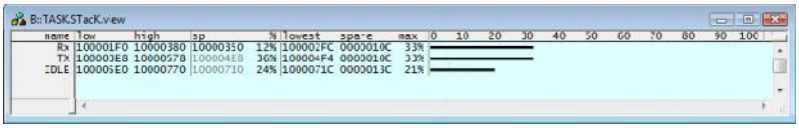


Figure T23: FreeRTOS Stack Information

This shows the stack frame for each task, the current stack pointer and (if the RTOS supports it) the maximum stack usage by that task.

Task Stack Frames

The current Stack Frame or call stack can be shown by selecting **Stackframe** from the **View** menu. When a Kernel Awareness is loaded the task dropdown becomes activated and allows users to display the call stack for any running task in the system. Simply select the required task. To return to the current task, select the blank line at the top of the list. An example can be seen in Figure T24.

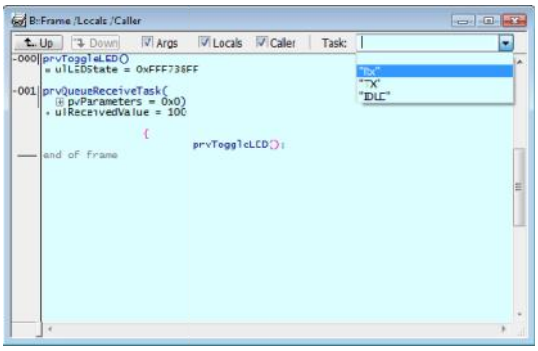


Figure T24: FreeRTOS Task Call Stack

Task aware breakpoints

Task aware breakpoints can be set. Open the **Break.Set** window and expand it to full size by clicking the Advanced button. For the address/expression fill in the variable **ulReceivedValue**. Change the type to "Write". Finally, in the task drop down select task "**Rx**". You should have something like Figure T25. Click the "OK" button to set the breakpoint.

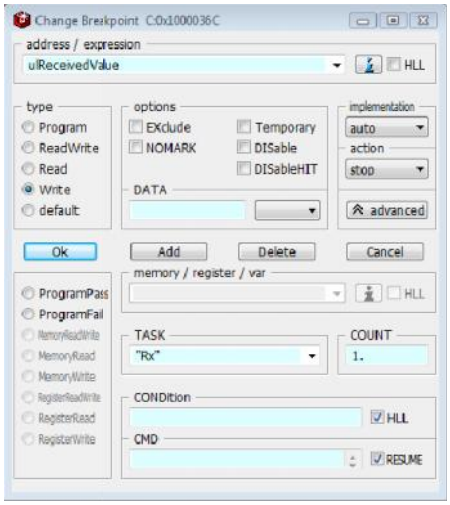


Figure T25: Task Aware Breakpoint

From the **Break** menu select the **List** option to display a list of all current breakpoints.

Start the target running and it will eventually halt in a block of assembly code at the label "**lastbytes:**". In the **Break.List** window the breakpoint which caused the halt will be highlighted in yellow. An example of this can be seen in Figure T26.

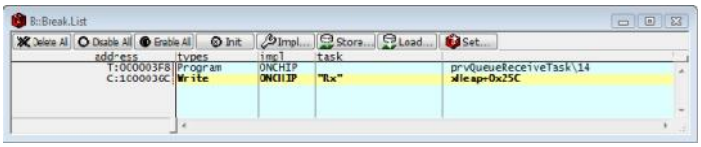


Figure T26: Active Breakpoint

Right-click the breakpoint and select **Change...** from the pop-up menu. In the resulting **Break.Set** window (which should have the breakpoint details already filled in) change the task from "Rx" to "TX". Click the "OK" button to change the breakpoint and start the target running again. This time it will not halt as the writes to **ulReceivedValue** occur outside of the context of task "Rx".

Task aware tracing

Click the new icon ()that has been added to the toolbar. The target will run for 6 seconds whilst it collects some trace data. Once the data has been collected four new windows will be opened showing task switch data. The results will look like Figure T27.

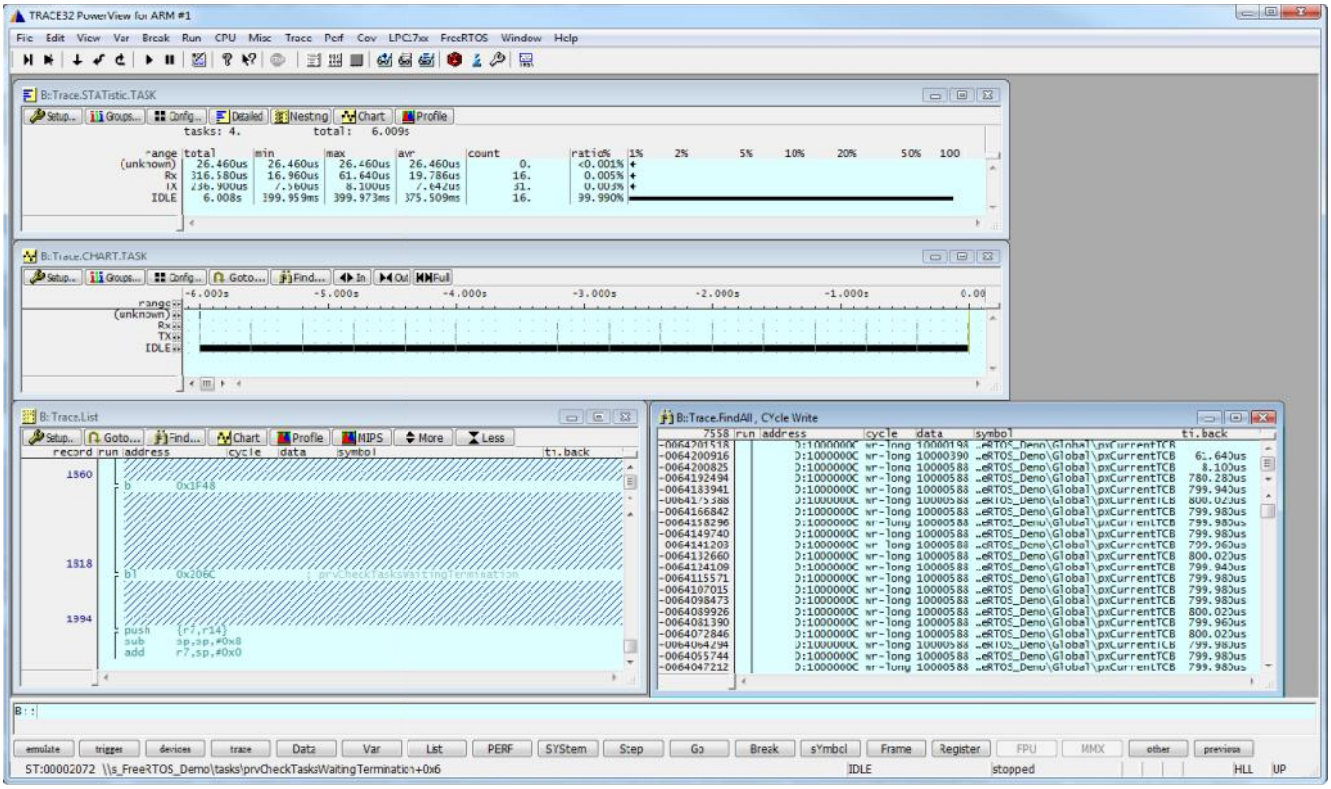


Figure T27: Task Runtime Statistics

Clicking any Windows and pressing F1 will open the documentation to give you more information about the window.

The top window shows detailed runtime statistics of all tasks in the system. Additional columns can be added but shown initially are total time spent in, minimum, maximum and mean times for each task.

The second window shows task switches over time. This behaves as the Trace chart windows described earlier for panning and zooming.

The bottom right window lists all task switches and the time between each (approx. 800us in this example). Left-clicking any of these will cause the bottom left window to jump to that point in the trace buffer so you can see the circumstances around that event.

To return to the previous display, right-click any of the grey background and select **P000** from the pop-up menu.