Getting Started with Red Trace

Red Suite 5

Version 5.1.2

16 April, 2013

Chapter 1. Overview

Red Trace is the collective name for the advanced debug and trace capabilities provided by Code Red Technologies. It uses ARM's CoreSight debug components provided in Cortex-M3/M4 based systems, including the Serial Wire Viewer and instruction trace functionality. Red Trace offers levels of visibility into your debug target never before possible without expensive external trace capture boxes.

Instruction trace is provided in the LPCXpresso IDE and Red Suite. It provides the ability to record a trace of executed instructions on Cortex-M3 and M4 parts that incorporate the on board ETB trace buffer. It works with LPC-Link, and the Red Probe+ debug probes.

The Serial Wire Viewer (SWV) tools provide access to the memory of a running target and facilitate trace without needing to interrupt the target. It also requires only one extra pin on top of the standard Serial Wire Debug (SWD) connection. Use of Red Trace requires connection to the target system using a compatible debug probe. This includes Code Red's Red Probe, Red Probe+, and TI's ICDI, but not LPC Link or the RDB-Link of an RDB1768v2 board.

Note

If you are using Red Suite with an evaluation license, then you will only be able to invoke a single SWV trace run per debug session

.

This guide first discusses the SWV bases features in sections 2 to 7. The instruction trace functionality is described in section 8.

Chapter 2. Red Trace SWV Views

Red Trace presents target information collected from a Cortex-M3/M4 based system in a number of different trace views within the Red Suite IDE

Each trace view is presented within the **Debug Perspective** or the **Develop Perspective** by default.

Trace views that are not required may be closed to simplify the user interface. They may be re-opened using the **Window -> Show View -> Other...** menu item, as per Figure 2.1.

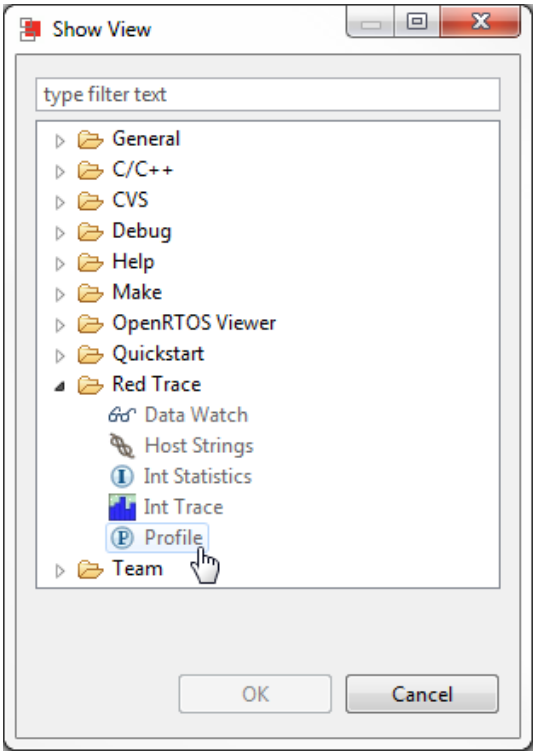


Figure 2.1. Reopening a view

The following trace views are provided:

Profile

- Provides a statistical profile of application activity.

Interrupt Statistics

- Provides counts and timing information for interrupt handlers.

Interrupt Trace

- Provides a time-based graph of interrupts being entered and exited, including nesting.

Data Watch

- Provides the ability to monitor (and update) any memory location in real-time, without stopping the processor

Host Strings

- Provide a very low overhead means of displaying diagnostic messages as your code is running

All views, except for Host Strings, are completely non-intrusive. They do not require any changes to the application, nor any special build options; they function on completely standard applications.

Each trace view provides a set of toolbar buttons that are used to control the collection and presentation of trace information. Starting data collection within one trace view will result in data collection for other views being suspended. The data presentation area of each trace view is enabled only when data collection for the view is active.

Chapter 3. SWV Configuration

3.1. Starting Red Trace

To use Red Trace's SWV features, you must be debugging an application on a Cortex-M3/M4 based MCU, connected via a supported debug probe

You may start Red Trace at any time whilst debugging your program. The program does **not** have to be stopped at a breakpoint. Before the collection of data commences, Red Trace will prompt for settings related to your target processor, as per Figure 3.1.

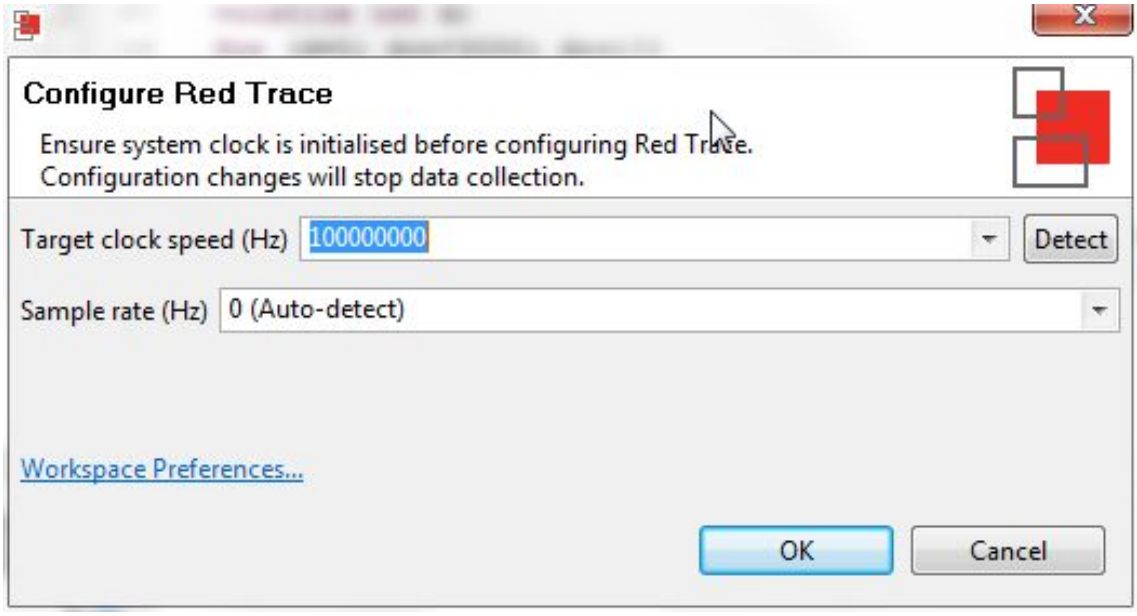


Figure 3.1. Configure Red Trace

3.1.1. Target Clock Speed

The **Detect** button will detect the *current* clock speed of the target processor. Due to the way the Trace data is transferred by the target processor, setting the correct clock speed is essential to determine the correct baud rate for the data transfer. If the clock speed setting does not match the actual clock speed of the processor, data will be lost, and the trace data will be meaningless.

Care needs to be taken when to ensure that when using the **Detect** button, the application has set the clock speed for normal operation, otherwise the **Detect** button will provide an inappropriate value. Clock set up code may be executed by the startup code run before the breakpoint at the start of main() is hit. For example in NXP projects which use CMSIS – Cortex Microcontroller Software Interface Standard – this will be done by the startup code calling the SystemInit() function. However some projects – for instance the TI/Luminary StellarisWare examples – clock setup is typically done within the main() function itself.

Note:
The core frequency must be lower than 80 MHz to use the SWD functionality required by Red Trace on the NXP LPC1850 and LPC4300 targets.

3.1.2. Sample rate

This is the frequency of data collection (sampling) from the target. You can normally leave this set to the default **Auto-detect**, which will provide a sample rate of approximately 50kHz, depending on the target clock speed. Available sample rates are calculated from the clock speed and can be seen in the dropdown.

3.2. Start Trace

Once you are debugging your application, press the **Start Trace** button to begin the collection of trace data and enable the updating (refresh) of the view at regular intervals. Once trace has been started, updates of the view may be paused by pressing the button again (now reading **Stop Trace**). Collection of data will continue while refreshing of the view is paused.

3.3. Refresh

Press the **Refresh** button to start the collection of trace data (if trace had not already been started) and to perform a single update (refresh) of the view. The collection of data will continue. You may switch between continuous refresh (using the **Start Trace** button) and individual refreshes of the view in order to inspect the collected data in more detail.

3.4. Settings

If it becomes necessary to change the system clock speed or the data sampling rate during a debugging session, use the **Settings** button to notify Red Trace. Any changes will stop data collection.

3.5. Reset Trace

Press the **Reset Trace** button to reset any cumulative data presented in the view. The collection of data will continue.

3.6. Save Trace

Press the **Save Trace** button to save the contents of the trace view to a file in CSV format. This can be useful for offline analysis of the data in a spreadsheet, for example.

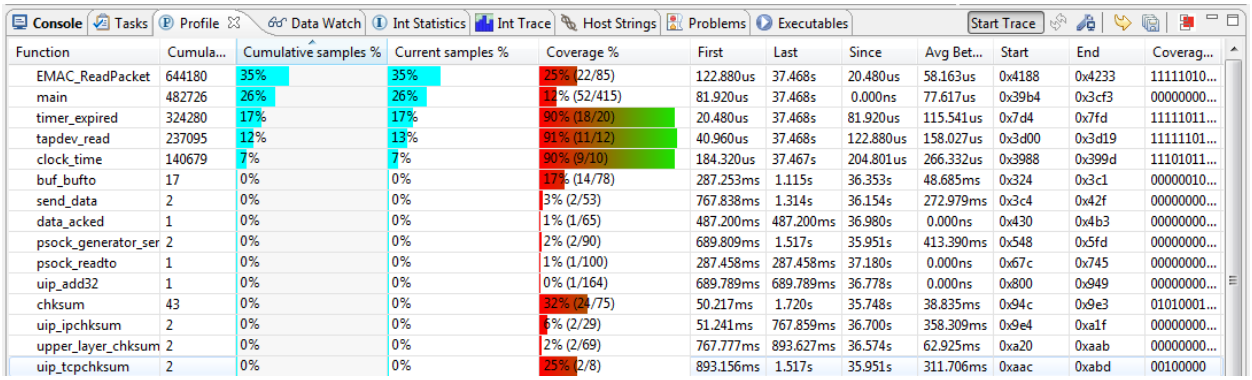
Chapter 4. Profile Trace

4.1. Overview

Profile tracing provides a statistical profile of application activity. This works by sampling the program counter (PC) at the configured sample rate (typically around 50 kHz). It is completely non-intrusive to the application – it does not affect the performance in any way. As profile tracing provides a *statistical* profile of the application, more accurate results can be achieved by profiling for as long as possible. Profile tracing can be useful for identifying application behavior such as code hotspots.

4.2. Profile view

The Profile view gives a profile of the code as it is running, providing a breakdown of time spent in different functions. An example screenshot is shown in Figure 4.1.



Function	Cumula...	Cumulative samples %	Current samples %	Coverage %	First	Last	Since	Avg Bet...	Start	End	Coverag...
EMAC_ReadPacket	644180	35%	35%	25% (22/85)	122.880us	37.468s	20.480us	58.163us	0x4188	0x4233	11111010...
main	482726	26%	26%	11% (52/415)	81.920us	37.468s	0.000ns	77.617us	0x39b4	0x3cf3	00000000...
timer_expired	324280	17%	17%	90% (18/20)	20.480us	37.468s	81.920us	115.541us	0x7d4	0x7fd	11111011...
tapdev_read	237095	12%	13%	91% (11/12)	40.960us	37.468s	122.880us	158.027us	0x3d00	0x3d19	11111101...
clock_time	140679	7%	7%	90% (9/10)	184.320us	37.467s	204.801us	266.332us	0x3988	0x399d	11101011...
buf_bufto	17	0%	0%	17% (14/78)	287.253ms	1.115s	36.353s	48.685ms	0x324	0x3c1	00000010...
send_data	2	0%	0%	3% (2/53)	767.838ms	1.314s	36.154s	272.979ms	0x3c4	0x42f	00000000...
data_acked	1	0%	0%	1% (1/65)	487.200ms	487.200ms	36.980s	0.000ns	0x430	0x4b3	00000000...
psock_generator_ser	2	0%	0%	2% (2/90)	689.809ms	1.517s	35.951s	413.390ms	0x548	0x5fd	00000000...
psock_readto	1	0%	0%	1% (1/100)	287.458ms	287.458ms	37.180s	0.000ns	0x67c	0x745	00000000...
uip_add32	1	0%	0%	0% (1/164)	689.789ms	689.789ms	36.778s	0.000ns	0x800	0x949	00000000...
chksum	43	0%	0%	32% (24/75)	50.217ms	1.720s	35.748s	38.835ms	0x94c	0x9e3	01010001...
uip_ipchksum	2	0%	0%	6% (2/29)	51.241ms	767.859ms	36.700s	358.309ms	0x9e4	0xa1f	00000000...
upper_layer_chksum	2	0%	0%	2% (2/69)	767.777ms	893.627ms	36.574s	62.925ms	0xa20	0xaab	00000000...
uip_tcpchksum	2	0%	0%	25% (2/8)	893.156ms	1.517s	35.951s	311.706ms	0xaac	0xabd	00100000

Figure 4.1. Profile View

- **Cumulative samples**: This is the total number of PC samples that have occurred while executing the particular function.
- **Cumulative samples (%)**: This is the same as above, but displayed as a percentage of total PC samples collected.
- **Current samples (%):** Number of samples in this function in the last data collection (refresh period)
- **Coverage (%)**: Number of instructions in the function that have been seen to have been executed.
- **Coverage Bitmap**: the coverage bitmap has 1 bit for each half-word in the function. The bit corresponding to the *address* of each sampled PC is set. Most Cortex-M instructions are 16-bits (one half-word) in length. However, there are some instructions that are 32-bits (two half-words). The bit corresponding to the second halfword of a 32-bit instruction will *never* be set.
- **First**: This is the first time (relative to the start time of tracing) that the function was sampled.

- **Last**: This is the last time (relative to the start time of tracing) that the function was sampled.
- **Since**: It is this long since you last saw this function. (current – last)
- **Avg Between**: This is the average time between executions of this function.

Note:

Coverage is calculated statistically — sampling the PC at the specified rate (e.g. 50Khz). It is possible for instructions to be executed but not observed. The longer trace runs for, the more likely a repeatedly executed instruction is to be observed. As the length of the trace increases, the observed coverage will tend towards the true coverage of your code. However, this should not be confused with full code coverage.

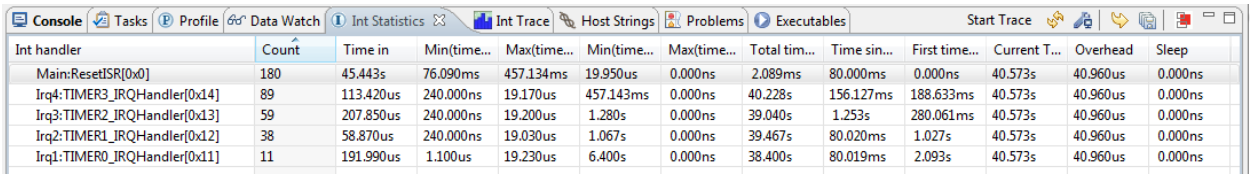
Chapter 5. Interrupts

5.1. Overview

Interrupt tracing provides information on the interrupt performance of your application. This can be used to determine time spent in interrupt handlers and to help optimize their performance.

5.2. Interrupt Statistics view

The Interrupt Statistics view displays counts and timing information for interrupt service handlers. An example screenshot is shown in Figure 5.1.



Int handler	Count	Time in	Min(time...	Max(time...	Min(time...	Max(time...	Total tim...	Time sin...	First time...	Current T...	Overhead	Sleep
Main:ResetISR[0x0]	180	45.443s	76.090ms	457.134ms	19.950us	0.000ns	2.089ms	80.000ms	0.000ns	40.573s	40.960us	0.000ns
Irq4:TIMER3_IRQHandler[0x14]	89	113.420us	240.000ns	19.170us	457.143ms	0.000ns	40.228s	156.127ms	188.633ms	40.573s	40.960us	0.000ns
Irq3:TIMER2_IRQHandler[0x13]	59	207.850us	240.000ns	19.200us	1.280s	0.000ns	39.040s	1.253s	280.061ms	40.573s	40.960us	0.000ns
Irq2:TIMER1_IRQHandler[0x12]	38	58.870us	240.000ns	19.030us	1.067s	0.000ns	39.467s	80.020ms	1.027s	40.573s	40.960us	0.000ns
Irq1:TIMER0_IRQHandler[0x11]	11	191.990us	1.100us	19.230us	6.400s	0.000ns	38.400s	80.019ms	2.093s	40.573s	40.960us	0.000ns

Figure 5.1. Interrupt Statistics View

Information displayed includes:

- **Count**: The number of times the interrupt routine has been entered so far.
- **Time In**: The total time spent in the interrupt routine so far.
- **Min (time in)**: Minimum time spent in the interrupt routine for a single invocation.
- **Max (time in)**: The Maximum time spent in a single invocation of the routine.
- **Min (time between)**: The minimum time between invocations of the interrupt.
- **Max (time between)**: The maximum time between invocations of the interrupt.
- **Total time between**: The total time spent outside of this interrupt routine.
- **Time since last**: The time elapsed since the last time in this interrupt routine.
- **First time run**: The time (relative to the start of trace) that this interrupt routine was first run.
- **Current time**: The elapsed time since trace start.
- **Overhead**: The cumulative overhead time elapsed entering and exiting all handlers
- **Sleep**: Time the processor spent sleeping.

5.3. Interrupt Trace view

The Interrupt Trace view provides a time-based graph of interrupts and exceptions and shows their nesting and when the exception is dismissed. This gives a visual representation of time in interrupt service routines and the transitions between them. An example screenshot is shown in Figure 5.2.

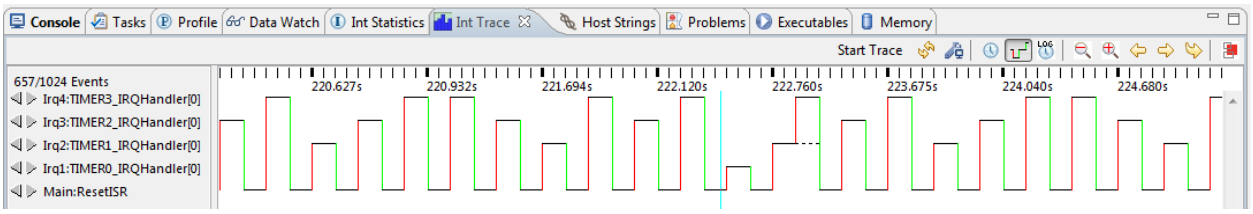


Figure 5.2. Interrupt Trace View

Each interrupt handler (interrupt service routine) is listed in the left hand panel, and horizontally is the time axis. In the waveform, entry into an interrupt routine is indicated with a red vertical line and exit from an interrupt routine is indicated by a green vertical line.

The buttons in the top right corner deserve further explanation and are enlarged in Figure 5.3.

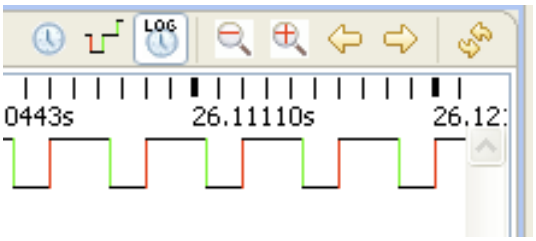


Figure 5.3. Interrupt trace display controls

- Clicking on the clock icon puts the display into a linear time mode.
- The next button puts the display into an event view. In this view time is no longer linear on the x-axis but this can be very useful for showing sparse events that are spread out in time at seemingly unrelated intervals.
- The ‘LOG’ button puts the display into a log-time view which has the effect of compressing time to show more information with less loss of detail.
- ‘+’ and ‘-’ are used to ‘zoom’ the display in the time-axis.
- The left and right buttons are used to scroll the time display.
- The refresh button literally shows a capture of another set of samples in the view.

The panel on the left of the Interrupt Trace view lists the interrupt service routines in the application being debugged, as per Figure 5.4.

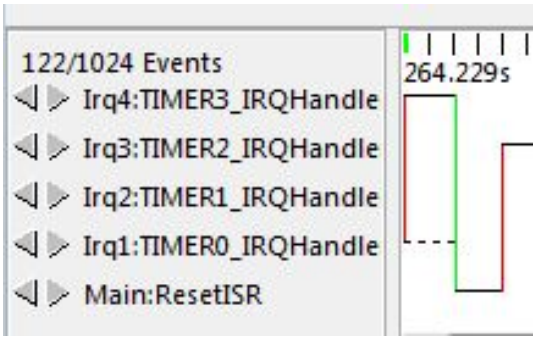


Figure 5.4. Interrupt handlers

The arrows to the left of the names of the service routines allow you to move the centre of the waveform display to the next transition on that event. This is particularly useful with sparse events.

Chapter 6. Data Watch Trace

6.1. Overview

This view provides the ability to monitor (and update) any memory location in real-time, without stopping the processor. In addition up to 4 memory locations can also be traced, allowing all access to be captured. Information that can be collected includes whether data is read or written, the value that is accessed and the PC of the instruction causing the access.

This information can be used to help identify ‘rogue’ memory accesses, monitor and analyze memory accesses, or to profile data accesses.

Real-time memory access is also available, allowing any memory location to be read or written without stopping the processor. This can be useful in real-time applications where stopping the processor is not possible, but you wish to view or modify in-memory parameters. Any number of memory locations may be accessed in this way and modified by simply typing a new value into a cell in the Data Watch Trace view.

Warning:
Data Watch Trace and Instruction Trace (page 19) cannot be used simultaneously as they both require use of the DWT unit.

6.2. Data Watch view

The view is split into 2 sections — with the **item display** on the left and the **trace display** on the right, as per Figure 6.1.

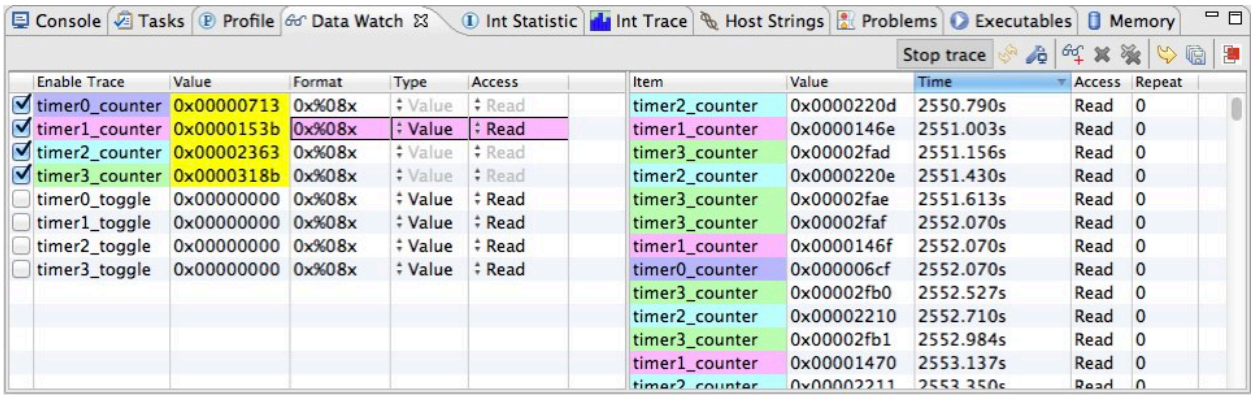


Figure 6.1. Data Watch View

Use the **Add Data Watch Items** button to display a dialog to allow the memory locations that will be presented to be chosen, as per Figure 6.2.

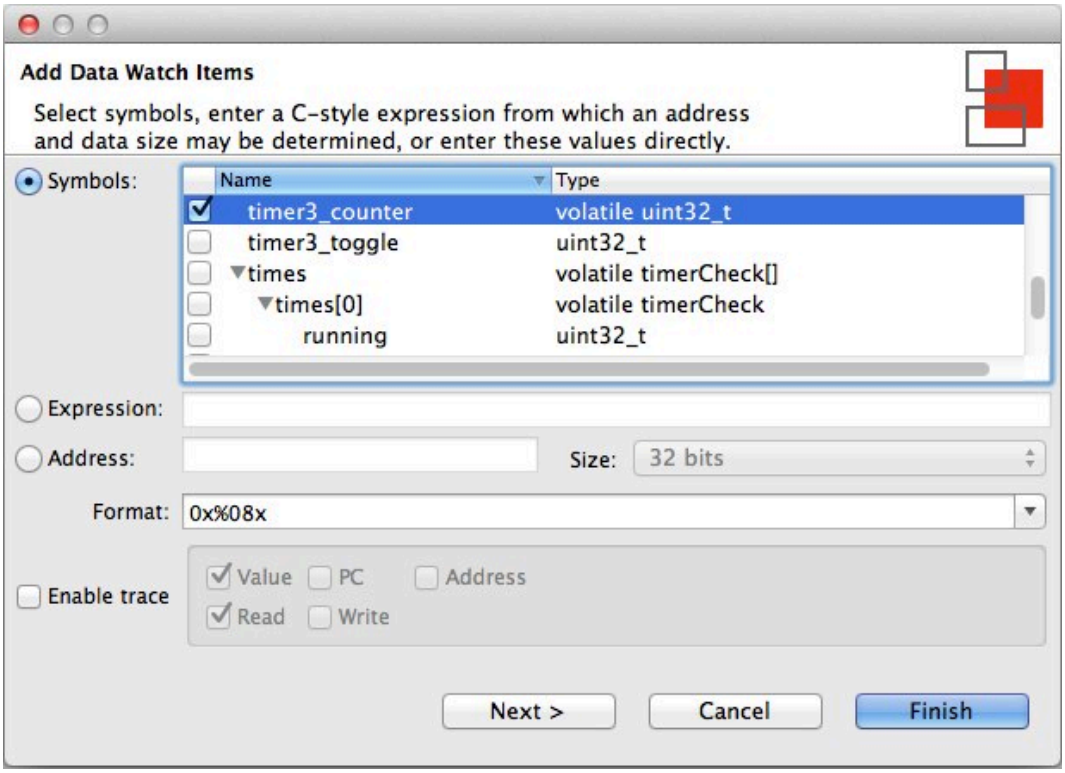


Figure 6.2. Add Data Watch Items

These locations may be specified by selecting global variables from a list; by entering a C expression; or by entering an address and data size directly. Trace may be enabled for each new item. If trace is not enabled, the value of the data item will be read from memory.

Data watch trace works by setting an address into a register on the target chip. This address is calculated at the time that you choose an item to watch in the **Add Data Watch Items** dialog. Thus, while you can use an expression, such as `buffer[bufIndex+4]`, the watched address will not be changed should `bufIndex` subsequently change. This behavior is a limitation of the hardware.

The format drop down box provides several format strings to choose from for displaying an items value. The format string can be customized in this box, as well as in the item display.

With trace enabled, the options for tracing the item's value, the PC of the instruction accessing the variable, or its address can be set. Additionally, the option to trace reads, writes or both can be set when adding a variable. These settings can be subsequently updated in the item display.

Note:

It is not possible to add some kinds of variables when the target is running. Suspending the execution of the target with the ⏸ button before adding these variables will overcome this limitation.

Pressing **Finish** adds the current data watch item to the item display and returns to the data watch view. Pressing **Next** adds the current data watch item, and displays the dialog to allow another item to be added.

6.2.1. Item Display

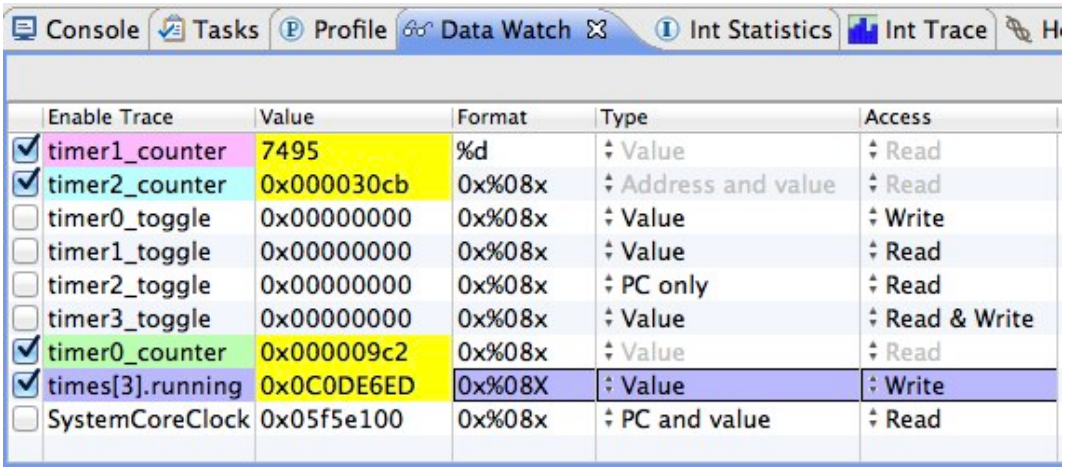


Figure 6.3. Data Watch Item Display

As shown in Figure 6.3, the item display lists the data watch items that have been added. The following information is presented:

- **Enable Trace** - tracing of this item may be enabled or disabled using the checkbox. A maximum of 4 items may be traced at one time. Each traced item is given a color code, so that it may be picked out easily on the trace display (see Figure 6.4).
- **Value** - shows the current value of the item and may be edited to write a new value to the target. If the current value has changed since the last update, then it will be shown highlighted in yellow.
- **Format** - shows the printf-style expression used to format the value and may be edited
- **Type** - shows the trace type, which may be edited while trace is disabled:
  - **Value** - just the trace the value transferred to/from memory
  - **PC only** - just trace the PC of the instruction making the memory access
  - **PC and value** - trace the value and the PC
  - **Address** - trace the address of memory accessed
  - **Address and value** - trace the address and value
- **Access** - shows the access type, which may be edited while trace is disabled:
  - **Write** - just trace writes to the memory location
  - **Read** - just trace reads to the memory location
  - **Read & Write** - trace both reads and writes

The variables in the item display persist between debug sessions. They are saved when the session ends and automatically re-added when the trace is started. If a variable is removed from the code between debug sessions the user is alerted that the variable cannot be restored.

6.2.2. Trace Display

The trace display shows the traced values of the memory locations. Each location being traced is given a particular color code, to allow all access to it to be easily picked out. See Figure 6.4.

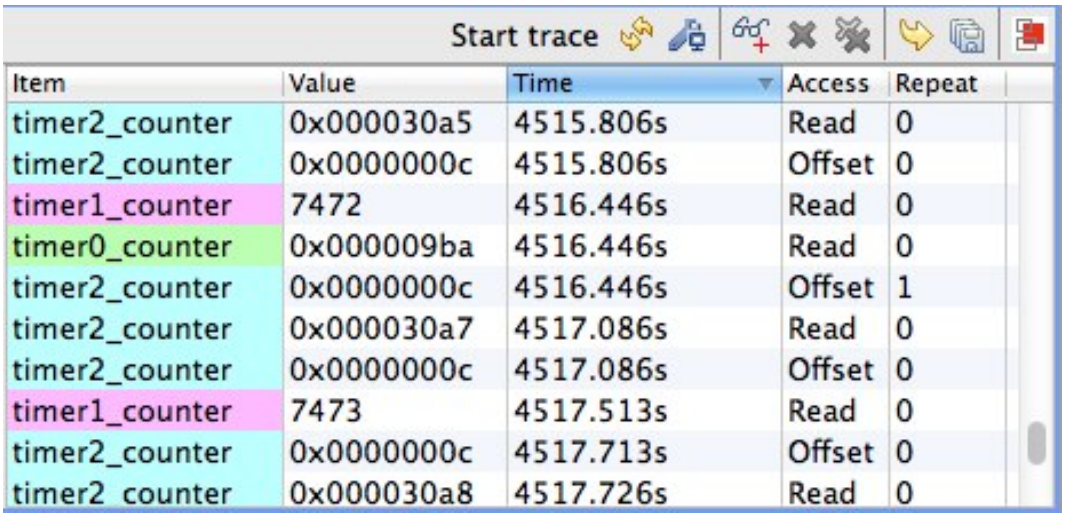


Figure 6.4. Data Watch trace display

Chapter 7. Host Strings (ITM)

7.1. Overview

Host Strings use the Instrumentation Trace Macrocell (ITM) within the Serial Wire Viewer (SWV) functionality provided by Cortex-M3/M4 based systems to display diagnostic messages as your code is running.

The overhead of using Host Strings is much lower than the traditional method of generating diagnostic messages using printf (either through semihosting or retargeted to use a serial port), as the “intelligence” for Host Strings is contained within Red Trace rather than in application code running on the target. The overhead is so low that the instrumentation can remain in your target application code and the generation of the debug messages can simply be globally turned off by default in the ITM trace enable register and then turned on again when the debugger is attached – giving very elegant in-system diagnostic capabilities.

The Host Strings mechanism uses a very low overhead store instruction added to your code or your OS kernel to cause an appropriate diagnostic string to be displayed within the Red Suite IDE. Time stamps are captured along with the instrumentation events.

7.2. Defining Host Strings

Host Strings are defined within a Host Strings file. To create a Host Strings file for a project use the **New File wizard** by right-clicking on the project within the Project Explorer view and selecting **New -> Host Strings File** from the pop-up menu. The Host Strings file must be located immediately under the project folder and must be named `hoststrings.xml`.

Host Strings files are associated with the **Host Strings Editor** and the new file is opened within this editor automatically when the **New File wizard** finishes, as per Figure 7.1.

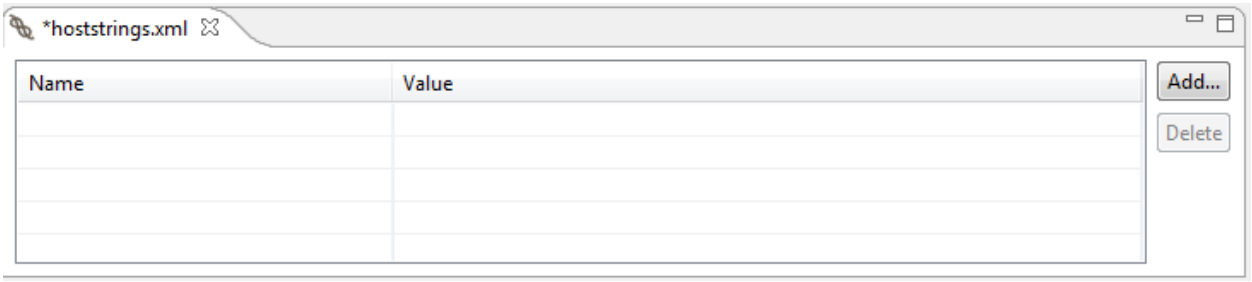


Figure 7.1. Blank Host Strings Editor

Each host string describes a message to be presented within the Host Strings view within Red Suite, plus the formatting information for a single parameter that is sent from the target for presentation within the message. Click on the **Add...** button within the editor to add a new host string, as per Figure 7.2.

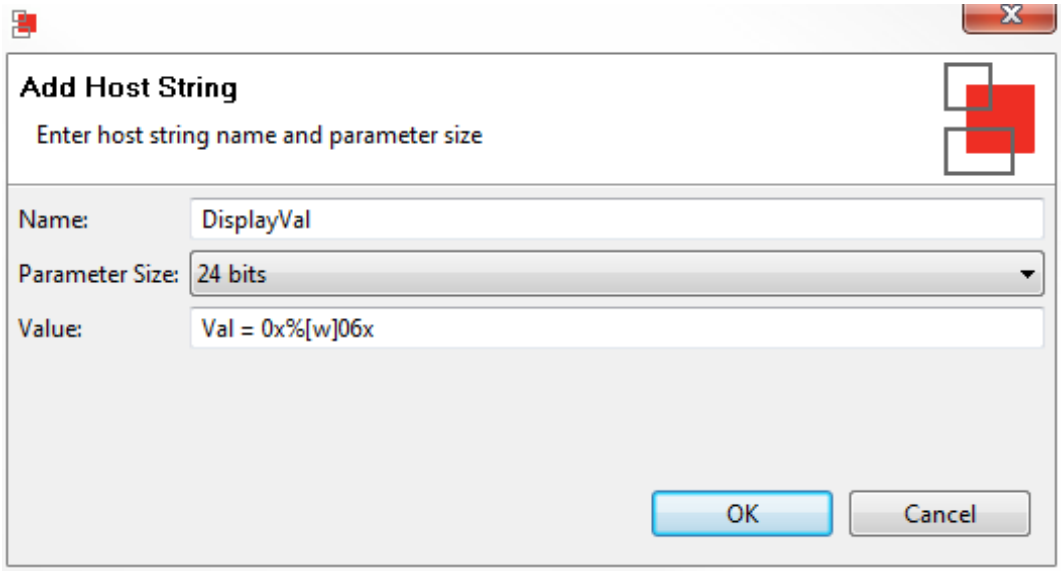


Figure 7.2. Adding a new Host String Name

Name

- The name of the Host Strings Macro that will be invoked by the application code running on the target. Note that this name will be automatically prefixed with `HS_` when the `hoststrings.h` header file is generated by the Host Strings process during project build. Note that the host string name is subject to the naming restrictions for C pre-processor macros.

Parameter Size

- An optional parameter can be passed as part of a call to a Host Strings Macro. This can be of size 8-bit, 16-bit or 24-bit or specified as **None** if no parameter is required. Note that the Host Strings mechanism does not support the passing of full 32-bit quantities as a parameter — such parameters will be truncated to 24 bits by the macro.

Value

- String that will be displayed in the Host Strings view within Red Suite when the Host Strings Macro is executed on the target.
- By default, when you add a new Host String, the value will be set to the name of the Host String, plus the parameter in hexadecimal format. However this value can be edited as required.
- The format of the parameter is `%[inf][width]ctrl`, where
  - `inf` = `b`, `h` or `w` for byte, half, word (truncated to 24-bits).
  - `ctrl` = `u` or `d` (for decimal), `x` or `X` (for hexadecimal), or `b` (for binary).
  - `width` (optional) a number to specify the minimum number of characters to be printed and whether this should be zero-padded (similar to `printf`).

- Host strings providing incorrect formatting information are marked with a warning triangle within the editor.

An example set of Host Strings within the Host Strings Editor are shown in Figure 7.3.

- **DisplayVal** will display the string `"msticks = "`, followed by the parameter in decimal format (as specified by the `%[w]d`).
- **LedsOn** will display the string `"LEDs now on"`.
- **LedsOff** will display the string `"LEDs now off"`.

Tip:
The value of existing host strings may be edited by clicking on the message string within the editor.

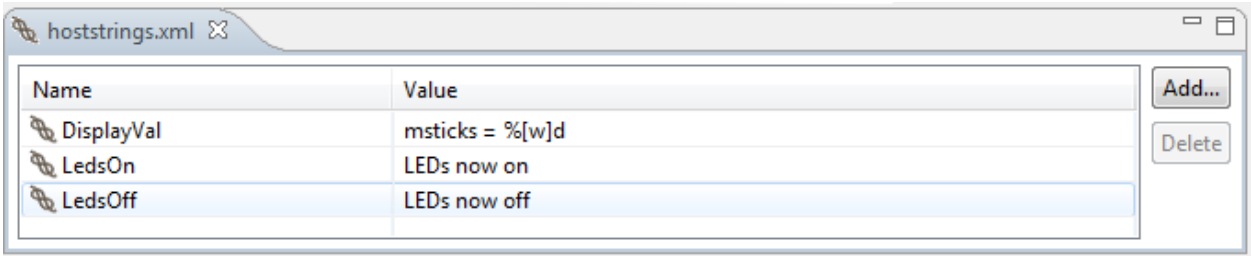


Figure 7.3. Populated Host Strings Editor

7.3. Building the Host Strings macros

When a Host Strings file is created using the **New File wizard**, the management of host strings for the associated project is enabled automatically. A header file named `hoststrings.h` is generated using a Host Strings Builder as part of the project build. This behavior may be enabled or disabled for any project by right-clicking on the project within the Project Explorer view and selecting **Managed Host Strings** from the pop-up menu. The header file contains the host string macros for use within your code

7.4. Instrumenting your code

Project code may be instrumented with Host Strings by referencing the Host Strings header file using :

```
#include "../hoststrings.h"
```

Note that the above assumes that the source code for your project is located in a subdirectory immediately below the root directory of your project (where the Host String files are located). If your project is more complex, then it may be better to add `${ProjDirPath}` to your project include paths using the menu:

Project -> Properties -> C/C++ Build -> Settings -> MCU C Compiler -> Includes

(repeating for both Debug and Release Build variants) and then simply use:

```
#include "hoststrings.h"
```

Host string messages may then be triggered within your code by adding calls to the pre-processor macros defined within this header file. For example:

```
HS_DisplayVal(msticks);
```

7.5. Host Strings view

The Host Strings view presents the message associated with a host string each time the associated host string macro is called within your code. The message includes the formatted value of the parameter passed into the macro.

Figure 7.4 shows example Host Strings output, as generated by running the `RDB1768cmsis2_HostStringsDemo` project, which can be found in the `RDB1768cmsis2.zip` example bundle provided with Red Suite 5.

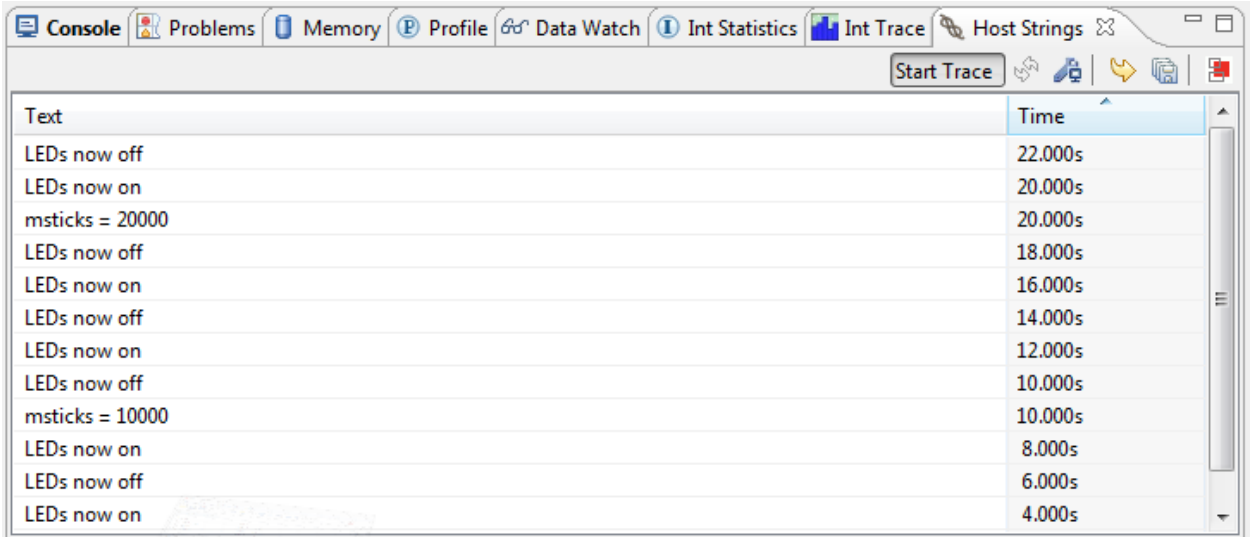


Figure 7.4. Host Strings output

Chapter 8. Instruction Trace

Instruction trace provides the ability to record and review the sequence of instructions executed on a target. Red Suite 5 introduces support for instruction trace via on board trace buffers. Instruction trace makes use of the Embedded Trace Buffer (ETB) on Cortex M3 and M4 parts and the Micro Trace Buffer (MTB) on the Cortex-M0+. The instruction trace which is generated at high speed can be captured in real time and stored in these on-chip buffers, so that they can be downloaded at lower speeds without the need for an expensive debug probe.

Red Suite 5 exposes the powerful Embedded Trace Macrocell (ETM) on Cortex M3 and M4 to focus the generated trace stored on the ETB. Trace can be focused on specific areas of code or triggered by complex events for example. On the Cortex M0+, the MTB provides simple instruction trace using shared SRAM.

This documentation is divided into four parts.

- Getting Started (page 19) - Tutorials to help you learn how to use instruction trace
- Concepts (page 27) - Technical background of instruction tracing
- Reference (page 37) - Details of the interface and operation of instruction trace
- Troubleshooting (page 47) - solutions to common challenges you may face

Note:
Instruction trace is only available on supported targets (page 46) due to hardware requirements and licensing restrictions.

8.1. Getting Started

The following tutorials guide you through the process of using instruction trace.

8.1.1. Configuring the Cortex-M0+ for Instruction Trace

Instruction trace on Cortex-M0+ targets requires that some SRAM need to be reserved. The Micro Trace Buffer (MTB) that provides the Instruction Trace capabilities stores the execution trace to SRAM. It is therefore necessary to reserve some SRAM to ensure that user data is not overwritten by trace data.

In this tutorial you will add an array to your Cortex-M0+ MCU project to reserve the space for the MTB instruction traces. Whenever the MTB is to be used with a project this configuration must be applied. This configuration is **not** required for instruction trace on **Cortex-M3** and **Cortex-M4** MCU parts.

Warning

Enabling the MTB without properly configuring the target's memory usage will result in unpredictable behavior. The MTB can overwrite user code or data which is likely to result in a hard fault.

Earlier versions of Red Suite 5 instructed users to edit the memory configuration to reserve MTB memory. Any older projects that were configured in that way should have their memory configuration reverted to the original memory configuration press **Restore Default** in the **MCU settings** properties page Project -> Properties -> C/C++ Build -> MCU settings .

Instruction trace makes it easy for you to reserve memory for the MTB. The following steps describe the process:

1. start a debug session to the target

2. Display the **Instruction Trace config** view by clicking Window -> Show View -> Instruction Trace

3. Pause execution of the target and press the refresh button in the **Instruction Trace config**

   - Instruction Trace will then search for valid reserved memory.

   - Assuming that the project is not configured for MTB use, the **Instruction Trace config** view will show an instruction screen.

4. Choose the desired buffer size from the instruction screen.

5. Copy the generated code from the **Instruction Trace config** and paste it into the source code file containing the main function.

6. Terminate your target

7. Rebuild and relaunch the debug session

The generated code includes a header file that places an array called `__mtb_buffer__` of the requested size into memory at the required alignment. The MTB will then be configured to use this space as its buffer allowing it to record execution trace without overwriting user code or data. The array `__mtb_buffer__` should not be used within your code since the MTB will overwrite any data entered into it.

To remove the MTB buffer, simply remove the code that you pasted into your project.

8.1.2. Trace the most recently executed instructions

This tutorial will get you started using the Instruction Trace capabilities of Red Trace. You will configure the target's trace buffer as a circular buffer, let your program run on your target, suspend it and download the list of executed instructions.

Note:
Instruction Trace depends on optional components in the target. These components may of may not have been implemented by the chip vendor. See the Instruction Trace Overview (page 27) for more information.

First, you will debug code on your target. You can import an example project or use one of your own.

Step 0: Configure memory if using a Cortex-M0+ parts If you are using a Cortex-M0+ part with an MTB you must first configure the memory usage of your code to avoid conflicts. See the Configuring the Cortex-M0+ for instruction trace (page 19) for instructions on how to do this.

Step 1: Start the target and show the Instruction Trace view

1. Build and execute your code on the target.
2. Suspend the execution of your target by selecting **Run -> Suspend**
3. Display the **Instruction Trace** view by clicking Window -> Show View -> Instruction Trace

Step 2: Configure the trace buffer

1. Press the **Record Trace Continuously** button in **Instruction Trace**
2. Resume execution of your target by selecting **Run -> Resume**

The trace buffer should now be configured as a circular buffer and your target should be running your code. Once the trace buffer is filled up, older trace data is overwritten by the newer trace data. Configuring the trace buffer as a circular buffer ensures that the most recently executed code is always stored in the buffer.

If the **Record Trace Continuously** button was grayed out, or you encountered error when trying to set check out the troubleshooting guide (page 47) .

Step 3: Download the content of the buffer

1. Suspend the execution of your target by selecting **Run -> Suspend**
2. Press **Download trace** in the **Instruction Trace** view.

There may be a short delay as the trace is downloaded from target and decompressed. Once the trace has been decompressed, it is displayed as a list of executed instructions in the **Instruction Trace** view.

Step 4: Review the captured trace

In this step we will explore the captured trace by stepping through it in the instruction view and linking the currently selected instruction to the source code which generated it as well as seeing it in context in the disassembly view.

1. Toggle the **Link to source** button  so that it is selected

2. Toggle the **link to disassembly**  so that it is selected too

3. Select a row in the **Instruction Trace** view

4. Use the up and down cursor keys to scroll through the rows in the **Instruction Trace** view.

As a row becomes selected the source code corresponding to the instruction in that row should be highlighted in the source code editor. The disassembly view should also update with the current instruction selected. There can be a slight lag in the disassembly view as the instructions are downloaded from the target and disassembled.

Step 5: Highlight the captured instructions

In this step we will turn on the profile view. In the source code editor the instruction which were traced will be highlighted. This highlighting can be useful for seeing code coverage. In the disassembly view each instruction is labeled with the number of times each it was executed.

- Toggle the **Profile information** button 

See Toggle profile information (page 40) for more information.

8.1.3. Stop trace when a variable is set

In this tutorial you will configure instruction trace to stop when the value of a variable is set to a specific value.

This tracing could be useful for figuring out why a variable is being set to a garbage value. Suppose we have a variable that is mysteriously being set to `0xff` when we expect it to always be between `0x0` and `0xA` for example.

The set up depends on which trace buffer your target implements. See one of the following sections for the detailed instructions for your target.

- Cortex M3 or M4 using ETB (page 23)

- Cortex M0+ using MTB (page 25)
  - Note: only supported by Freescale Kinetis M0+ part, not for NXP LPC8xx parts.

Cortex M3 or M4 using ETB

To trace the instructions that resulted in the write of the unexpected value we are going to have trace continuously enabled with the ETB acting as a circular buffer. Next we will set up a trigger (page 33) to stop trace being written to the ETB after the value `0xff` gets written to the variable we are interested in. The trigger event requires two DWT comparators. One of the comparators watches for any write to the address of the variable and the other watches for the value `0xff` being written to any address. The position of the trigger in the trace is set to capture a small amount of data after the trigger and then to stop putting data into the ETB.

Step 1: Start the target and show the Instruction Trace config view

1. Build and execute your code on the target.
2. Suspend the execution of your target by selecting **Run -> Suspend**
3. Display the **Instruction Trace Config** view by clicking Window -> Show View -> Instruction Trace Config
4. Press the refresh button in the **Instruction Trace config** view

Step 2: Find the address of the variable

1. Display the **Disassembly** view by clicking Window -> Show View -> Disassembly
2. Enter the variable name into the location search box and hit enter.
3. Copy the address of the variable to clipboard.

Note:
We are assuming that the variable is a global variable.

Step 3: Enable trace

In this step we configure trace to be generated unconditionally in the **Instruction Trace config** view.

1. In the **Trace enable** section:
   a. Select the **Simple** tab
   b. Select **enable**

Step 4: Enable stalling

To make sure that no packets get lost if the ETM becomes overwhelmed we enable stalling. Setting the **stall level** to 14 bytes mean that the processor will stall when there are only 14 bytes left in the formatting buffer. This setting ensures that we do not miss any data, however, it comes at the cost of pausing the CPU when the ETM cannot keep up with it. See stalling (page 30) in the Concepts section for more information.

1. Check the **Stall processor** check box

2. Drag the slider to set the **Stall level** to 14 bytes

Step 5: Configure watchpoint comparator

In the first watchpoint comparator choose **data write** in the comparator drop-down box and then enter the address of the variable that you obtained earlier. This event resource will be true whenever there is a write to that address, regardless of the value written.

Step 6: Configure the value written

Select **Data Value Write** from the drop-down, note that data comparators are not implemented in all watch point comparators. Enter the value we want to match, `0xff`, in the text box. Next we link the data comparator to the comparator we configured in **step 5** by selecting 1 in both the **link 0** and **link 1** fields. Select the **Data size** to **word**.

The event resource **Comparator 2** will now be true when **Comparator 1** is true and the word `0x000000ff` is written.

Step 7: Configure the trigger condition

In the **trigger condition** section select the **One Input** tab. Set the resource to be **Watchpoint Comparator 2** and ensure that the **Invert resource** option is **not** checked. These setting ensure that a trigger is asserted when the **Watchpoint Comparator 2** is true — i.e. when `0xff` is written to our focal variable.

Step 8: Prevent trace from being recorded after the trigger

Slide the **Trigger position** slider over to the right, so that only 56 words are written to the buffer after the trigger fires. This setting will provide some context for the trigger and allows up to 4040 words of trace to be stored from before the trigger, which will help us see how the target ended up writing `0xff` to our variable.

The configured view should look like Figure 8.3.

Step 9: Configure and resume the target

Now press the green check button to apply the configuration to the ETM and ETB. Resume the target after the configuration has been applied. Once the target resumes, the buffer will start filling with

instructions. Once the buffer is filled the newest instructions will overwrite the oldest. When the value `0xff` is written to the focal variable, the **trigger counter** will start to count down on every word written to the buffer. Once the **trigger counter** reaches zero, no further trace will be recorded, preserving earlier trace.

Step 10: Pause target and download the buffer

After some time view the captured trace by pausing the target and downloading the content of the buffer:

1. Suspend the execution of your target by selecting **Run -> Suspend**
2. Display the **Instruction Trace** view by clicking Window -> Show View -> Instruction Trace
3. Download the content of the ETB by pressing in the **Instruction Trace** view.
4. Check that the trigger event occurred and was captured by the trace by clicking on

Cortex M0+ using MTB

To trace the instructions that resulted in the write of the unexpected value we are going to enable continuous trace with the Micro Trace Buffer (MTB) acting as a circular buffer. Next we will configure the MTB_DWT to disable the MTB after the value `0xff` gets written to the variable we are interested in. The MTB_DWT is a vendor specific module available on some Freescale parts which allows two DWT comparators to start and stop MTB trace collection. One comparator watches for any write to the address of the variable and the other watches for the value `0xff` being written to any address.

Note: NXP LPC8xx does not support this functionality.

Step 0: Configure the target memory usage

The MTB shares memory with your program's data in the target. It is therefore necessary to ensure that the MTB does not overwrite your program's data. See the Configuring the Cortex-M0+ for instruction trace (page 19) for detailed instruction on setting it up.

Warning

Using the MTB without properly configuring the target's memory usage will result in unpredictable behavior.

Step 1: Start the target and show the Instruction Trace Config view

1. Build and execute your code on the target.
2. Suspend the execution of your target by selecting **Run -> Suspend**
3. Display the **Instruction Trace Config** view by clicking Window -> Show View -> Instruction Trace Config

4. Press the refresh button in the **Instruction Trace config** view

Step 2: Find the address of the variable

1. Display the **Disassembly** view by clicking Window -> Show View -> Disassembly
2. Enter the variable name into the location search box and hit enter.
3. Copy the address of the variable to clipboard.

Note:
We are assuming that the variable is a global variable.

Step 3: Enable trace

In this step we enable MTB tracing and allow it to be disabled when it receives a stop signal.

1. Switch back to the **Instruction Trace Config** view.
2. In the **Settings** section check the box marked **Enable MTB**
3. Also check the box marked **Disable on Stop signal**

Step 4: Watermark

To use the MTB as a circular buffer we do not use any watermarking functionality. Ensure that all the check boxes in the watermark section are cleared. The position of the watermark slider should have no effect. To find out more about watermarking see Watermarking (page 29) in the Concepts section.

Step 5: Configure the first watchpoint comparators

In this step we will configure a watchpoint to be true when both of the following condition are true: the value `0xff` is written to any location in memory, and the second comparator is also true.

1. Select **Data Value Write watchpoint** from the drop-down for the first comparator, note that this option is only implemented in the first watch point comparators.
2. Enter the value to match, `0xff`, in the text box.
3. In the **Data size** choose **word** so that we match a complete 32-bit word
4. Check the **Link to second comparator** box
5. Select the radio button in the **Stop** column for this comparator so that it send a stop signal when it matches.

Step 6: Configure the second watchpoint comparator

In this step we will configure the second comparator to be true whenever any value is written to the specified address. The first comparators will only return true it both comparators match – that is if the value `0xff` is written to memory and the the location of the write matches the address of the variable of interest.

1. Select **Data Write watchpoint** from the second watchpoint's comparator drop-down.
2. Enter the address of the variable obtained from the disassembly view.
3. Select the radio button in the **Start** column for this comparator so that it does **not** send a stop signal on every match

Note:
Since the **Enable Start signal** was not selected in the setting options the second comparator sending a start signal will have no effect. The MTB will only react to the stop signal which will be sent by the first comparator when both comparators match.

Step 7: Configure and resume the target

Now press the green check button  to apply the configuration to the MTB. Resume the target. Once the target resumes, the buffer will start filling with instructions. Once the buffer is filled the newest instructions will overwrite the oldest until `0xff` is written to the specified variable.

Step 10: Pause target and download the buffer

After some time you can pause the target again and retrieve the content of the buffer and view the captured trace.

1. Suspend the execution of your target by selecting **Run -> Suspend**
2. Display the **Instruction Trace** view by clicking  Window -> Show View -> Instruction Trace
3. Download the content of the MTB by pressing the download trace button  in the **Instruction Trace** view.

8.2. Concepts

8.2.1. Instruction Trace Overview

8.2.2. MTB Concepts

The CoreSight Micro Trace Buffer for the M0+ (MTB) provides simple execution trace capabilities to the Cortex-M0+ processor. It is an optional component which may or may not be provided by a given silicon vendor.

Red Trace supports the MTB on Freescale's Kinetis Cortex-M0+ targets and NXP's LPC8xx targets. Freescale augments the MTB with a simplified data watchpoint and trace unit (DWT) which facilitates the enabling and disabling of the MTB by LPC DWT instruction or data comparators.

The MTB captures instruction trace by detecting non-sequentially executed instructions and recording where the program counter (PC) originated and where it branched to. Given the code image and a this information about non-sequential instructions, the instruction trace component of Red Trace is able to reconstruct the executed code.

The huge number of instructions executed per second on a target generate large volumes of trace data. Since the MTB only has access to a relatively small amount of memory, it gets filled very quickly. To obtain useful trace a developer can configure the MTB to only focus on a small area of code, to act as a circular buffer or download the content of the buffer when it fills up. Additionally, all three of these techniques may be combined.

The following sections contain detailed information about the MTB's operation and use.

- Enabling the MTB (page 28) - options for controlling the MTB
- MTB memory configuration (page 29) - how the MTB uses memory
- MTB Watermarking (page 29) - reacting to the buffer filling up
- MTB Auto-resume (page 29) - combining multiple trace buffer captures
- MTB Data Watchpoint Trace (page 29) - using comparators with the MTB

Warning

The MTB does not have its own dedicated memory. The memory map used by the target must be configured so that some RAM is reserved for the MTB. The MTB must then be configured to use that reserved space as described in the Configuring the Cortex-M0+ for instruction trace (page 19) in the Getting Started section.

Enabling the MTB

The Micro Trace Buffer (MTB) can be enabled by pressing the **Continuous Recording** button  or checking **enable MTB** in the **Instruction Trace Config** view. Trace will only be recorded by the MTB when it is enabled.

The MTB can also be enabled and disabled by two external signals `TSTART` and `TSTOP`. On Freescale parts these signals are connected to the MTB_DWT block to allow comparator based control of the MTB. See MTB Data Watchpoint Trace (page 29) for more information.

If `TSTART` and `TSTOP` are asserted at the same time `TSTART` takes priority.

MTB memory configuration

Both the size and the position in memory of the MTB's buffer are user configurable. This flexibility allows the developer to balance the trade off between the amount of memory required by their code and the length of instruction trace that the MTB is able to capture before getting overwritten or needing to be drained.

Since the MTB uses the same SRAM used by global variables, the heap and stack, care must be taken to ensure that the target is configured so that the MTB's memory and the memory used by your code do not overlap. For more information see Configuring the Cortex-M0+ for instruction trace (page 19) in the Getting Started Guide.

MTB Watermarking

The MTB watermarking functionality allows the MTB to respond to the buffer filling to a given level by stopping further trace generation or halting the execution of the target. The watermark level and the actions to perform can be set in the **Instruction Trace Config** view (page 41). The defined action is performed when the Watermark level matches the MTB's write pointer value. The halt action can be augmented with the **auto-resume** behavior.

MTB Auto-resume

Red Trace provides the option to automatically download the content of the buffer and resume execution when the target is paused by the watermarking mechanism. This **auto-resume** functionality allows extended trace runs to be performed without being constrained by the size of the on chip buffer.

There is a significant performance cost associated with using **auto-resume** since the time taken to pause the target, download the buffer content and resume it again is much greater than the time the MTB takes to fill the buffer.

The data is not decompressed during the auto-resume cycles and so it is still necessary to press **Download trace buffer**  in the **instruction trace** toolbar to view the captured trace.

Tip:

To suspend the target once **auto-resume** is set press the **Cancel** button in the **downloading trace** progress dialog box. If the **downloading trace** progress dialog box is not displayed long enough to click **Cancel** use the **Stop auto-resume** button  in the **instruction trace view** toolbar. This disables **auto-resume** so that the target will remain halted the next time the MTB suspends it.

MTB Data Watchpoint Trace

Freescale provide an additional component to augment the MTB. The MTB_DWT unit provides two comparators that can be used to enable and disable the MTB. This unit is only available on Freescale parts.

The comparators support matching on the following:

- instruction address
- data access at an address
- data access value

When a comparator matches it can either send a `TSTART` signal or a `TSTOP` signal. When the MTB is configured to respond to these signals they can enable and disable the MTB. See the Stop Trace when variable is set (page 25) for an example in the Getting Started section.

See Data Watchpoint and Trace (page 35) for more information about configuring the comparators.

Warning:
The MTB_DWT can only match instructions at word aligned addresses.

MTB Downloading Trace

To obtain the instruction trace from the MTB once it has captured trace into it buffer, the content of the buffer needs to be downloaded by Red Trace and decompressed. There must be an active debug session for the trace to be downloaded. It can be downloaded using the Download trace buffer button  in the Instruction Trace view.

The trace recorded by the MTB is compressed. The code image is required to decompress the trace. This image must be identical to the image running on the target when the trace was captured. It is possible to download and decompress a trace from a previous session if the same code image is running on the target. However, if the trace buffer contains old trace data, and a different code image is downloaded to the target, downloading the buffer may result in invalid trace being displayed.

8.2.3. Embedded Trace Macrocell

The Embedded Trace Macrocell (ETM) provides real-time tracing of instructions and data. By combining the ETM with an ETB, some features of this powerful debugging tool are accessible using a low cost debug probe.

This section describes some of the key components and features of the ETM.

Stalling

The Embedded Trace Macrocell (page 30) (ETM) uses as relatively small FIFO buffer to store formatted packets. These packets are then copied to the embedded trace buffer (page 32) (ETB). This FIFO buffer can overflow when the rate of packets being written into it exceed the rate at which they can be copied out to the ETB.

If stalling is supported on the target the ETM can be set to stall the processor when an FIFO overflow is imminent. The stall level sets the threshold number of free bytes in the FIFO at which the processor should be stalled. The stall does not take effect instantly and so the level should be set such that there is space for further packets to be added to the buffer after the stall level is reached. Overflows can still occur even with stalling enabled.

The maximum value is the total number of bytes in the FIFO buffer. Setting the stall level to this will stall the processor whenever anything is entered into the FIFO. For example if a stall level of 14 bytes was chosen, the processor would be instructed to stall any time there were fewer than 14 bytes left in the in the buffer. If the target had a 24 byte FIFO buffer, this level would allow a 10 byte safety buffer for packets generated between the stall level being detected and the processor actually stalling.

Stalling is enabled from the **Instruction Trace Config** view (page 44). Check the **Stall processor** box and select a stall level with the slider.

ETM events

Events are boolean combinations of two event resources (page 31). These events can be used to control when tracing is enabled or to decide when the trigger will occur for example. The available event resources are dependent on the chip vendor's implementation.

Events can be specified in the **Instruction Trace Config** view (page 44). For details on how to build these events see ETM event configuration (page 45) .

Event Resources

Event resources are used by the ETM (page 30) to control the ETM's operation. They can be combined to form ETM events (page 31).

Different sets of event resources are implemented by different silicon vendors. The following event resources are supported by Red Trace:

- Watchpoint comparators (page 31) — match on addresses and data values
- Counters (page 32) — match when they reach zero
- Trace start/stop unit (page 32) — combine multiple input
- External (page 32) — match on external signals
- Hard wired (page 32) — always match

Watchpoint comparator event resource

Watchpoint comparators are event resources that facilitate the matching of addresses for the PC and the data access as well as matching data values. They are implemented by the Data Watchpoint and Trace (page 35) (DWT) component.

Counter event resource

A target may support a single **reduced counter** on counter 1. The reduced counter decrements on every cycle. This event resource is true when the counter equals zero, it then reloads and continues counting down. The reload value can be set in the ETM **Instruction Trace config** view.

Trace start/stop unit event resource

The trace start / stop block is an event resource that can combine all of the Watchpoint comparators. For example you could have trace start when the target enters a function call and stop when it exits that function, or use it to generate complex trigger events.

When a start comparator becomes true, the start / stop block asserts its output to high and stays high until a stop comparator becomes true. The start / stop block logic is reset to low when the ETM is reset.

Note that checking both start and stop for the same comparator will be treated as just checking start.

To use this method

- Set the trace enable to use the start stop block
- Find the entry and exit addresses for the focal function (in the disassembly view)
- Create instruction match conditions in the DWT comparators for those address
- Check the start box for the entry address and stop boxes for the exit address.

External event resource

External input may be connected to the ETM by the silicon vendor. Please see their documentation for more information about these vendor configurable elements.

Hard wired event resource

This resource will always be observed to be true. It can be useful for enabling something constantly when used with an **A** function, or to disable something when used with a **NOT A** function.

8.2.4. Embedded Trace Buffer

The Embedded Trace Buffer (ETB) makes it possible to capture the data that is being generated at high speed in real-time and to download that data at lower speeds without the need for an expensive debug probe. Red Trace Instruction trace on Cortex-M3 and Cortex-M4 targets is facilitated by the ETB in conjunction with the Embedded Trace Macrocell (page 30) (ETM).

The ETB and ETM are optional components in the Cortex-M3 and Cortex-M4 targets. Their implementation is vendor specific. When they are implemented the vendors may implement different

subsets of the ETM's and ETB's features. Instruction trace will automatically detect which features are implemented for your target; however, note that not all of the features listed in this guide may be available on your target.

The ETM compresses the sequence of executed instruction into packets. The ETB is an on chip buffer that stores these packets. This tool downloads the stored packets from the ETB and decompresses them back into a stream of instructions.

This feature is useful for finding out how your target reached a specific state. It allows you to visualize the flow of instructions stored in the buffer for example.

There are multiple ways to use the ETB. The simplest is continuous recording, where the ETB is treated as a circular buffer, overwriting the oldest information when it is full. There are also more advanced options that allow the trace to focused on code that is of interest to make the most of the ETBs memory. This focusing is achieved by stopping tracing after some trigger or by excluding regions of code. These modes of operation are described in this document.

Triggers

The trigger mechanism of using instruction trace works by constantly recording trace to the ETB until some fixed period after a specified event. This method allows the program flow up to, around or after some event to be investigated.

There are two components that need to be configured to use triggers. These are the trigger condition and the trigger counter. The trigger condition is an event (a Boolean combination of event resources). When the trigger condition event becomes true, the trigger counter counts down every time a word is written to the ETB. When the trigger counter reaches zero no further packets are written to the ETB.

To use the ETB in this way the Trace Enable event is set to be always on (hard wired). The ETB is constantly being filled, overflowing and looping round, overwriting old data. The trigger counter is set using the trigger position slider. The counter's value is set to the **words after trigger** value. You can think of the position of the slider as being the position of the trigger in the resulting trace that will be captured by the ETB.

There are three main ways of using the trigger: trace after – when that the data from the trigger onwards is the interesting information; trace about — the data either side of the trigger is of interest; and trace before — data before the trigger is the key information.

Trace after

When the slider is at the far left, the **words before trigger** will be zero and the **words after trigger** will be equal to the number of words which can be stored in the ETB. This corresponds to the situation where the region of interest occurs after the trigger condition. Once the ETB receives the trigger packet the

trigger counter, which was equal to **words after trigger** will count down with every subsequent word written to the ETB, until it reaches zero. The trigger packet will be early in the trace and the instruction trace will include all instructions from when the trigger event occurred, until the ETB buffer filled up.

Note that in order to facilitate decoding of the trace the ETM periodically emits synchronization information. On some systems the frequency that these are generated can be set. Otherwise, by default, they are generated every `0x400` cycles. It is therefore necessary to make sure that you allow some trace data to be collected before your specific area of interest when using the trigger mechanism so that this synchronization information is included.

Trace around

As the trigger position slider is moved to the right, the **words after trigger** value decreases. This means that the data will stop getting written to the ETB sooner. Since the Trace Enable event was set to true, there will be older packets, from before the trigger event, still stored in the ETB. The resulting instruction trace will include some instruction from before the trigger and some from after the trigger. Use the slider to balance the amount of trace before and after the trigger.

Trace before

With the trigger position slider towards the far right, the instruction trace will focus on the trace before the trigger event. Note that only trace captured after the instruction trace has been configured will be captured. For example pausing the target and setting a trigger condition for the next instruction, with the trace position set to the far right would not be able to include instruction trace from before the trace was configured, but rather it would stop after **words after trigger** number of words was written to the ETB, leaving most of the ETB unused.

Note that setting the trigger position all the way to the right, such that **words after trigger** is zero will disable the trigger mechanism.

Timestamps

When the **timestamps** check-box is selected and implemented, a time stamp packet will be put into the packet stream. The interval between packets may be configurable if the target allows it, or may be generated at a fixed rate.

If supported, there is a Timestamp event. Timestamps are generated when the event fires. A counter resource could be used to periodically enter timestamps into the trace stream for example.

Note:
ARM recommends against using the **always true** condition as it is likely to insert a large number of packets into the trace stream and make the FIFO buffer overflow.

Note that a time of zero in the time stamp indicates that time stamping is not fully supported

Debug Request

The ETB can initiate a debug request when a trigger condition is met. This setting causes the target to be suspended when a trigger packet is created.

Note

There may be a lag of several instructions between the trigger condition being met and the target being suspended

Output all branches

One of the techniques used by the ETM to compress the trace is to output information only about indirect branches. Indirect branches occur when the PC (program counter) jumps to an address that cannot be inferred directly from the source code.

The ETM provides the option to output packets for all branch instructions — both indirect and direct. Checking this option will output a branch packet for every branch encountered. These branch packets enable the reconstruction of the program flow without requiring access to the memory image of the code being executed.

This option is not usually required as the Instruction trace tool is able to reconstruct the program flow using just the indirect branches and the memory image of the executed code. This option dramatically increases the number of packets that are output and can result in FIFO overflows, resulting in data loss or reduced performance if stalling (page 30) is enabled. It can also make synchronization harder (e.g. in triggered traces) as you can end up with fewer I-Sync packets in the ETB.

8.2.5. Data Watchpoint and Trace

The data watchpoint and trace (DWT) unit is an optional debug component. Instruction trace uses its watchpoint to control trace generation. The Red Trace **Data Watch** view uses it to monitor memory locations in real-time, without stopping the processor.

Warning:

Instruction Trace and Data Watch Trace (page 11) cannot be used simultaneously as they both require use of the DWT unit.

Instruction address Comparator

Use the program counter (PC) value to set a watchpoint comparator resource true when the PC matches a certain instruction. Choose **Instruction** from the comparator drop-down and enter the PC to match in the match value field. Use the Disassembly view to find the address of the instruction that you are interested in. When the PC is equal to the entered match value, the watchpoint comparator will be true; otherwise it is false.

Setting a mask enables a range of addresses to be matched by a single comparator. Set the **Mask size** to be the number of low order bytes to be masked. The range generated by the mask is displayed next to the **Mask size** box. The comparator event resource will be true whenever the PC is within the range defined by the mask.

Note:

Instruction addresses must be half-word aligned

Warning

Instruction address comparators should not be applied to match a `NOP` or an `IT` instruction, as the result is unpredictable.

Data access address Comparators

Data access address Comparators are event resources that watch for reads or writes to specific addresses in memory. As with the instruction comparator, the address is entered as the match value. There are three different data access comparators:

- Data R/W — true when a value is read from or written to the matched address
- Data Read — only true if a value is read from the matched address.
- Data Write — only true if a value is written to the matched address.

Like instruction address comparators, data access address comparators can operate on a range of addresses.

Note:

These comparators do not consider the value being written or read — they only consider the address that is being read from or written to.

Data Value comparator

Data value comparators are triggered when a specified value is written or read, regardless of the address of the access. This comparator is typically implemented on one of the Watchpoint comparators on Cortex chips. There are three types of this comparator:

- Data Value R/W — true when a value is read or written that is equal to the **Match Value**
- Data Value Read — only true if a value is read that is equal to the **Match Value**
- Data Value Write — only true if a value is written that is equal to the Match Value

The size of the value to be match must be configured as either a **word**, **half word** or **byte** in the **Data size** drop-down. Only the lowest order bits up to the request size will be matched. For example, if the **Data size** is set to **byte**, only the lowest order byte of the match value will be used in the comparisons.

Typically, you might want to match only a specific value written to a specific variable. Data value comparators provide this facility by linking to up to two data access address comparators:

- Access to any address
  - set **link 0** and **link 1** to the data value comparator number
- Access to one address specified in another DWT comparator
  - Set both **link 0** and **link 1** to the address match comparator number
- Access to either of two addresses specified in two separate DWT comparators
  - Set **link0** to the first address comparator id
  - Set **link1** to the second address comparator id

Tip:
When there are only two DWT comparators the option to link the two comparators is given as a check-box.

Cycle Count

If supported by the chip vendor, the first comparator can implement comparison to the **Cycle Counter**. The **Cycle Counter** is a 32-bit counter which increments on every cycle and overflows silently. This event resource is true when the cycle counter is equal to the match value.

To use this feature, choose **Cycle Counter** from the comparator drop-down and enter the match value into the **Match Value** field.

8.3. Reference

8.3.1. Instruction trace view

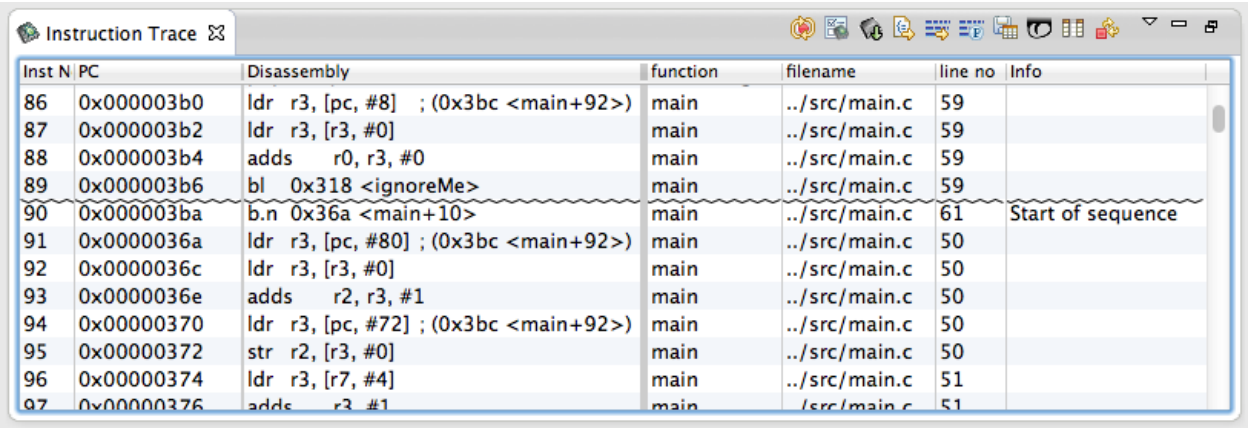
From the **Instruction Trace view** you can configure trace for your target, and download and view the captured trace. Open the **Instruction Trace view** by clicking Window -> Show View -> Instruction Trace.

It should look like Figure 8.1.

For trace generated by the ETM, the color of the text in the instruction list provides information about the traced instruction. Grayed-out text indicates that the instruction did not pass it condition. A red background indicates a break in the trace due an ETM FIFO buffer overflow. Instructions may be missing between red highlighted instruction and the proceeding entry in the trace view. If the **Stall** option is available it can be used to help ensure this does not occur in subsequent traces.

A break in the trace may occur due to trace becoming disabled and then renabled (for example to exclude the tracing of a delay function). Breaks in the trace are indicated by a line drawn across the row.

A green background highlights the trigger packet that is generated after the trigger condition is met. Press the **trigger button** to jump to this instruction in the instruction list.



Inst N	PC	Disassembly	function	filename	line no	Info
86	0x000003b0	ldr r3, [pc, #8] ; (0x3bc <main+92>)	main	../src/main.c	59	
87	0x000003b2	ldr r3, [r3, #0]	main	../src/main.c	59	
88	0x000003b4	adds r0, r3, #0	main	../src/main.c	59	
89	0x000003b6	bl 0x318 <ignoreMe>	main	../src/main.c	59	
90	0x000003ba	b.n 0x36a <main+10>	main	../src/main.c	61	Start of sequence
91	0x0000036a	ldr r3, [pc, #80] ; (0x3bc <main+92>)	main	../src/main.c	50	
92	0x0000036c	ldr r3, [r3, #0]	main	../src/main.c	50	
93	0x0000036e	adds r2, r3, #1	main	../src/main.c	50	
94	0x00000370	ldr r3, [pc, #72] ; (0x3bc <main+92>)	main	../src/main.c	50	
95	0x00000372	str r2, [r3, #0]	main	../src/main.c	50	
96	0x00000374	ldr r3, [r7, #4]	main	../src/main.c	51	
97	0x00000376	adds r3, #1	main	/src/main.c	51	

Figure 8.1. The instruction trace view

8.3.2. Instruction Trace view Toolbar buttons

There are several buttons in the toolbar of the **Instruction Trace View** that allow you to use Instruction Trace with your target.

- Record trace continuously
- Show Instruction Trace config view
- Download trace buffer from target
- Link to source
- Link to disassembly
- Toggle profile information
- Save trace to csv
- Jump to trigger
- Stop auto resume
- Select columns to display

Record trace continuously

The Record trace continuously button configures the trace buffer as a circular buffer. Once the trace buffer is filled up, older trace data is overwritten by newer trace data.

This mode of operation ensures that when the target is paused, the buffer will contain the most recently executed instructions.

Note: The target must be connected, with your code downloaded and execution suspended, before you can configure trace.

Show Instruction Trace config view

Press the Show Instruction Trace config view button to display the **Instruction Trace config** view. This view will provide you with access to all of the trace buffer's configuration settings.

The **Instruction Trace config** view's contents depend on the features supported by your target. See the following sections for more information on your target.

- Cortex M0+ MTB (page 41)
- Cortex M3 ETB (page 44)
- Cortex M4 ETB (page 44)

Note: The target must be connected, with your code downloaded and execution suspended, before you can configure trace.

Download trace buffer

Press the Download trace buffer button to download the content of the trace buffer from the target. The data will be downloaded and decompressed. The list of executed instructions will be entered into the Instruction View table.

Notes

- The target must be suspended in order to perform this action
- The content of the trace buffer can persist across resets on some targets. The buffer is decompressed using the current code image. If the code has changed since the data was entered in the buffer, the decompresser's output will be garbage.
- If no instructions are listed after downloading, check your configuration to make sure that instruction trace started.

Link to source

The Link to source toggle button enables the linking of the currently-selected instruction in the Instruction Trace View to the corresponding line of source code in the source code viewer. The line of source code that generated the selected instruction will be highlighted in the source code viewer.

Tip:
You can use the the cursor keys within the Instruction Trace View to scroll through the executed instructions.

Link to disassembly

The Link to disassembly toggle button enables the linking of the currently selected instruction in the Instruction Trace View to the corresponding instruction in the disassembly viewer. The selected instruction will be highlighted in the disassembly viewer.

Note: The target must be suspended to allow the disassembly view to display the code on the target.

Tip:
You can use the the cursor keys within the Instruction Trace View to scroll through the executed instructions.

Toggle profile information

The Toggle profile button enables and disables the display of profile information corresponding to the currently downloaded instruction trace.

When the display of profile information is enabled, a column appears in the disassembly view that shows the count of each executed instruction that was captured in the trace buffer.

Lines of source code whose instructions were recorded in the trace buffer are highlighted in the source view.

Tip:
You can customize the display of the highlighting by editing the **Profile info in source view** item in the preferences panel under General -> Editors -> Text Editors -> Annotations

Save trace

The Save trace button saves the content of the **Instruction Trace** View to a csv file. These files can be large and may take a few seconds to save.

Jump to trigger

Trace downloaded from an ETB (the embedded trace buffer on the Cortex M3 and M4) may contain a trigger packet. If the trace stream contain such a packet, the **jump to trigger** button will show it in the **instruction trace** view.

Stop auto-resume

When using the MTB auto-resume feature (page 29), the user may not have time to press the suspend button or the **Cancel** button in the download trace progress dialog box as the target is being rapidly suspended and restarted. Pressing the **Stop auto-resume** button will turn off the auto-resume feature, so that the target will suspend the next time the MTB reaches its watermark level without resuming.

Select columns

You can choose the columns shown in the instruction list using the select columns action. The available columns are listed below.

Rearrange the column ordering in the table by dragging the header of the columns.

Table 8.1. Instruction view column descriptions

Column	Description
Inst No	The index of the instruction in the trace
Time	The timestamp associated with an instruction
PC	The address of the instruction
Disassembly	The disassembled instruction
C	The condition code for the instruction. E for instructions that passed their condition and were executed, N for instructions which were not executed.
opCode	The op code for the instruction.
Arguments	The arguments for the op code
Offset	Offset of the instruction within the function
Function	The C function name which the instruction belongs to
Filename	The C source file that the instruction is associated with
Line no	The line number in the C source file that the instruction is associated with

8.3.3. Instruction Trace Config view for the MTB

Instruction trace with the MTB can be fully configured using the **Instruction Trace Config** view. Open the **Instruction Trace view** by clicking Window -> Show View -> Instruction Trace or by clicking on the **Instruction Trace Config** button in the **Instruction Trace** view. Once the target is connected, refreshing the view with the refresh button will display the options for the target.

Changes made to the MTB configuration are only applied when the **Apply** button is pressed.

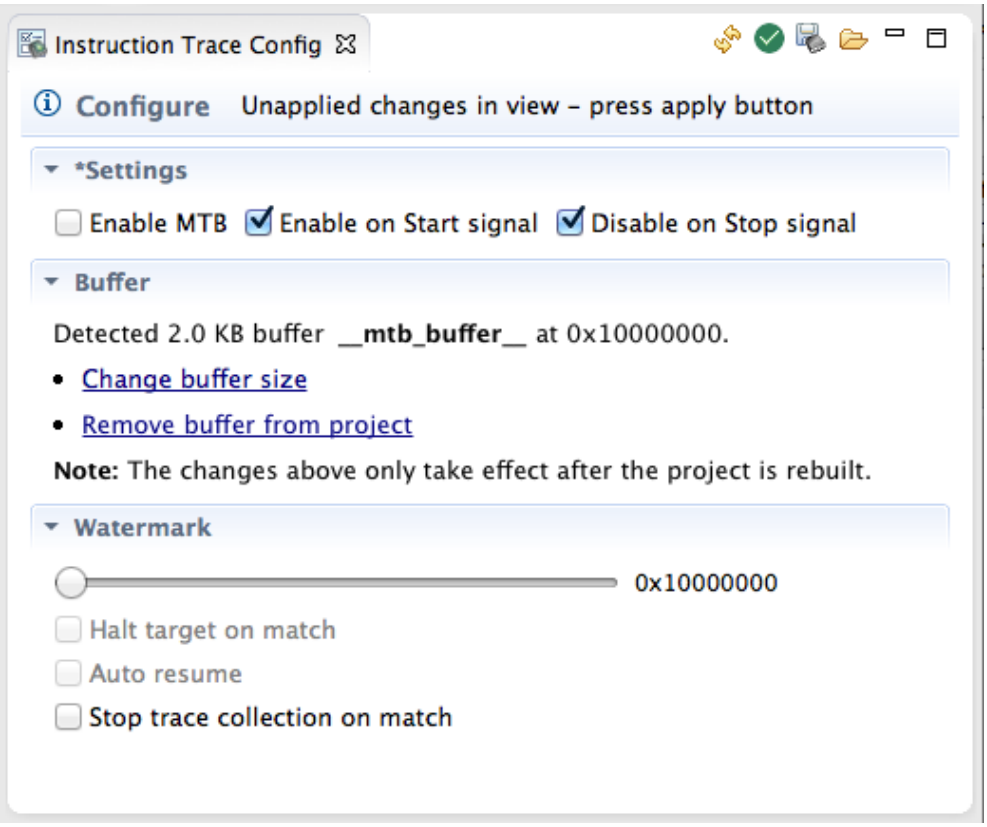


Figure 8.2. Instruction Trace config view for MTB instruction trace

Configuring the buffer

If the target does not have a buffer allocated for the MTB the view will display instruction on how to create the buffer when it is refreshed.

Enabling

The three check-boxes in the top section of the view control whether the MTB is enabled or not. The first check-box **Enable MTB** can be used to directly enable or disable the MTB. The second two check-boxes control whether or not the MTB is affected by start and stop signals being sent by the MTB_DWT or other external sources.

See *enabling the MTB* (page 28) in the Concepts section for more information. See *stop trace when a variable is set* (page 25) in the getting started guide for an example.

Buffer

The buffer section of the MTB configuration view displays information about where in memory the MTB data is stored. It displays the size of the buffer and provides instructions on how to change or remove the buffer.

See *MTB memory configuration* (page 29) for more information and *Configuring the Cortex-M0+ for instruction trace* (page 19) for an example.

Watermark

The watermark section of the MTB advanced configuration view allows you to configure an action to be performed when the buffer fills to a certain level. The slider configures the watermark level at which the action is triggered. The address next to the slider indicates the absolute address of the watermark.

Selecting **Halt target on match** suspends the target when the buffer fills to the specified watermark level. When the target is halted the content of the buffer is automatically drained and stored. The buffer is reset so that it can be refilled once the target is resumed. The trace is not decompressed or displayed in the **Instruction Trace** view until the **Download buffer** button is pressed. This behavior allows multiple trace runs to be concatenated.

Selecting **Auto resume** allows the target to be automatically restarted once the buffer has been downloaded. It only has an effect if **Halt target on match** is also selected. This auto-resume feature allow the trace capture not to be limited to the size of the MTB's buffer allowing code coverage to measured. The frequent interruptions have a large impact on target performance.

Note

You can suspend an auto-resuming target by pressing the **cancel** button in the buffer download progress dialog box. If the MTB buffer is sufficiently small, then the progress dialog box may not be displayed long enough for the user to select **cancel**. In that case you should press the **Stop auto-resume** button . Both of these methods will turn off the auto-resume feature and the target will suspend without restarting the next time the watermark matches.

Selecting **stop trace collection on match** allows the MTB to stop recording trace once it has been filled once, without interrupting the execution of the target. This feature could be useful when used in conjunction with a DWT comparator that starts trace on a certain condition.

Watchpoint comparators

Freescale targets allow the MTB to make use of a restricted DWT – note that this feature is not available on targets from other venders. The MTB can use the two MTB_DWT watchpoint comparators to conditionally enable and disable trace. See *Data Watchpoint and Trace* (page 35) for information on how to configure these conditions. The signal a matched comparator sends to the MTB depends on the radio button selected. If the radio button in the **start** column is selected then a `TSTART` signal is sent to the MTB. If the **Enable on Start signal** box is checked, then the `TSTART` signal will enable the MTB. If the radio button in the **stop** column is selected a `TSTOP` signal is sent to the MTB. If the **Disable on Stop signal** box is checked then the `TSTOP` signal will disable the MTB.

Viewing the state of the MTB

The state of the target is read each time the refresh button in the **Instruction Trace Config** view. For example the, the **Enable MTB** box will show whether or not the MTB is currently enabled. This information can be useful for confirming that `TSTART` and `TSTOP` signals are affecting the MTB as expected when using DWT comparators.

Pressing the **Apply** button will update the MTB's configuration — even if no settings are changed by the user. This action will have the effect of clearing the content of the MTB's buffer. That is, if the MTB contains trace that has not been downloaded and then the user applies the configuration, the content of the buffer will be lost.

8.3.4. Instruction Trace Config view for the ETB

Instruction trace with the ETB and ETM can be fully configured using the **Instruction Trace Config** view. Open the **Instruction Trace view** by clicking Window -> Show View -> Instruction Trace or by clicking on the **Instruction Trace Config** button in the **Instruction Trace** view. Once the target is connected, refreshing the view with the refresh button will display the options for the target.

Changes made to the MTB configuration are only applied when the **Apply** button is pressed. A section title will have a '*' appended to it if it contains unapplied changes.

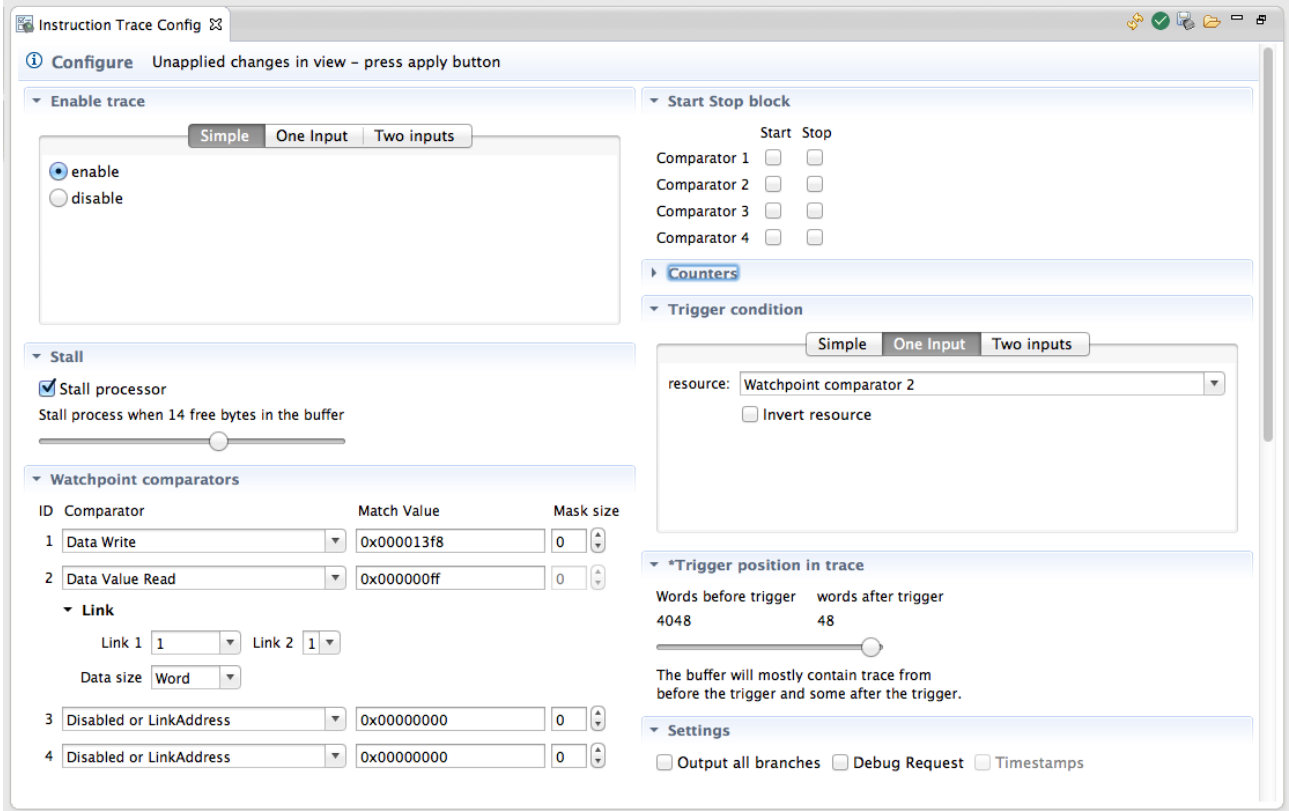


Figure 8.3. The advanced configuration dialog for the ETB

ETM Event configuration

An ETM event (page 31)is a boolean combination of up to two event resource inputs(page 31). The **Instruction Trace Config** view provides an easy way to build these events by allowing the user to choose the complexity of the event. These are used for trace enablement and the trigger condition for example.

- The simplest option is a binary enabled/disabled choice. This is accessed by selecting the **Simple** tab. Select **enable** for the Event to always be true or **disable** for the event to always be false. See Figure 8.4.

- To use a single event resource select the **One Input** tab. An event resource can be chosen from the drop-down and the event will be true when the event resource is true. Checking the **Invert resouce** box will cause the Event to be true when the event resource is false, and visa-versa. See Figure 8.5.

- To combine two event resources select the **Two Inputs** tab. The events can be chosen from the drop-downs and the logical combination operation selected. As with the **One Input** tab the resources can be inverted. See Figure 8.6.

The configuration on the visible tab is used when the **Apply** button is clicked.

Figure 8.4. Simple ETM event configuration

Figure 8.5. One Input ETM event configuration

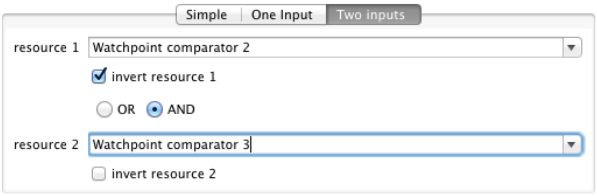


Figure 8.6. Two Inputs ETM event configuration

8.3.5. Supported targets

Instruction Trace is not supported on all targets. It only works on targets which have the necessary hardware. Additionally not all vendors are supported at this time.

NXP

Instruction trace is supported on the following targets from NXP:

- LPC8xx (MTB)
- LPC18xx (ETM / ETB)
- LPC43xx (ETM / ETB)

Freescale

Instruction trace is supported on the following targets from Freescale:

- Kinetis L Series (MTB)
- Kinetis K Series (ETM / ETB)

8.4. Troubleshooting

This section of the help provides solutions to common problems that you may encounter while using Instruction Trace.

8.4.1. General

Instruction Trace claims target not supported when it should be

Instruction Trace caches some information about the target in the Launch configuration to reduce setup time. In some cases this information may become corrupted. Deleting the Launch configuration will force Instruction Trace to refresh this information.

You can also double check that your target is in *Supported Targets* (page 46)

8.4.2. MTB

Target crashes when MTB is enabled

The MTB may be overwriting code or data on the target. Check that the MTB's memory configuration is compatible with the target's memory configuration.

Target keeps resuming itself and I cannot stop it

Try to press the **Stop auto-resume** button  in the **instruction trace view** toolbar. This button disables **auto-resume** so that the target will remain halted the next time the MTB suspends it.

See *Auto-resume* (page 29) in the Concepts section for more information.

MTB_DWT not sending signal to MTB on instruction match

The MTB_DWT instruction match comparators only match word aligned instructions. Check if the instruction you are trying to match is word aligned (i.e. its address is divisible by 4).

8.4.3. ETB

Trigger Packet missing from trace even though trigger occurred

It may be that the trigger packet was lost due to FIFO overflow and was not written to the ETB. To make sure that it actually was triggered, look at the trigger counter (the number of words to write after the trigger condition). The trigger counter only decrements after the trigger has been activated. If it has not decremented, check your trigger condition.

If the trigger counter has decremented and you see no packet, try enabling stalling if it is implemented.