Training Cortex-M Tracing

Release 09.2024

MANUAL

Training Cortex-M Tracing

Version 05-Oct-2024

History

29-Sep-23	Table of optional trace blocks for member of the Cortex-M family updated.
06-Feb-18	Initial version of the manual.

Cortex-M Trace

This document should be read in conjunction with **"Training Arm CoreSight ETM Tracing"** (training_arm_etm.pdf) which shows how to drive and interrogate the TRACE32 trace subsystems. Many of the views and types of data collected will be similar to those shown in that document. It is assumed that the reader will be familiar with the basic trace concepts, allowing this document to focus on Cortex-M specific features.

Many of the debug components of Cortex-M based designs are optional IP blocks that may or may not have been included in the design of the chip being debugged. Always check the chosen chips' documentation. Trace components may include:

Micro Trace Buffer (MTB)

This allows program flow data to be saved to internal SRAM. The data can be read via the JTAG or Serial Wire Debug (SWD) interface. The amount of SRAM used and the location of the buffer are software configurable. The size of the SRAM buffer limits the amount of program flow trace that can be captured. Trace writes to SRAM take priority over system writes to the AHB-Lite interface, with one or more wait states being inserted into AHB-Lite accesses if a trace write occurs simultaneously. This may affect the run-time performance of the application being traced under high bus load situations. This optional IP block may not be present in all devices.

Embedded Trace Macrocell (ETM)

This unit allows program flow data to be fed to the TPIU. There, it will be formatted for eventual delivery to off-chip trace tools. The amount of trace data that can be captured is equal to the size of the buffer in the external tools or, in the case of streaming, the size of the host PCs hard drive. This feature is an optional IP block and may not be present in all devices.

Instrumentation Trace Macrocell (ITM)

This unit provides three main features: software instrumented trace; integration of trace packets from the DWT into the trace stream; timestamp generation for ITM and DWT trace packets. It is an optional feature and may not be present in all devices.

Data Watchpoint and Trace (DWT)

This optional component provides a number of trace like features: Interrupt trace; Data Trace; ETM Trigger; PC Sampler and Trigger. Each of these will be covered in more detail in a later section of this document. The output from the DWT is fed into the ITM for formatting and inclusion into the trace stream.

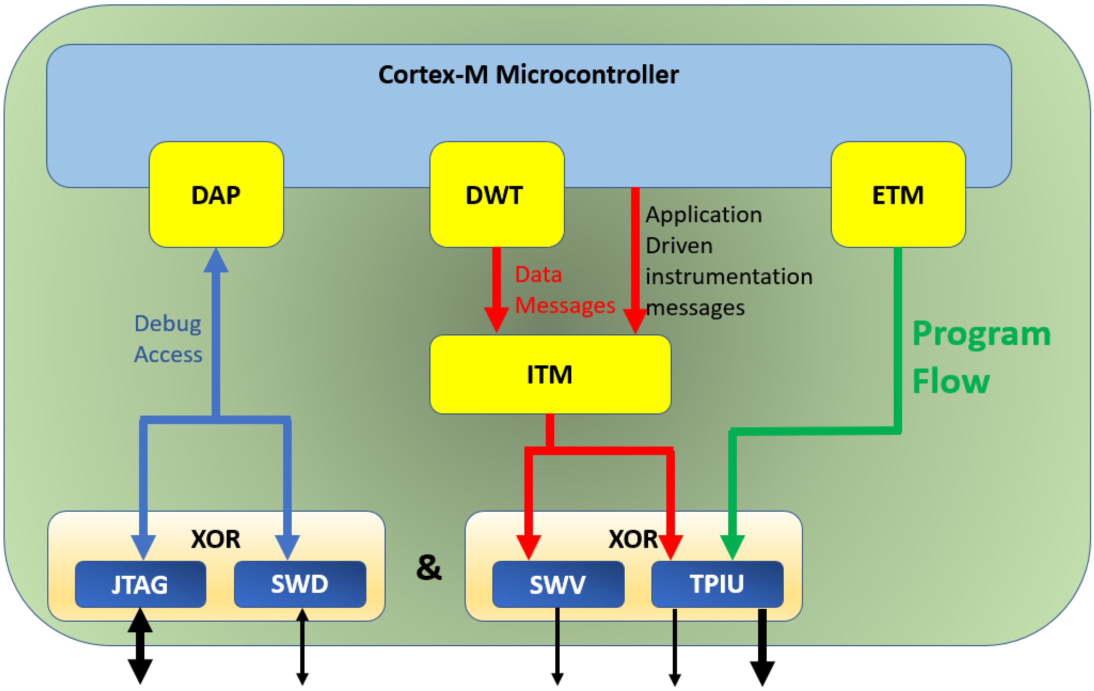
Trace Port Interface Unit (TPIU)

This block is required to route the trace to its final destination: off-chip or on-chip. The TPIU can aggregate trace from multiple sources into a single stream, allowing for tracing of multi-core designs. Two off-chip modes are supported: Serial Wire Viewer (SWV) or Serial Wire Output (SWO); and parallel trace. SWV or SWO is a single bit wide trace port designed for low speed trace – usually from the ITM/DWT – where this option is selected the TPIU will silently ‘drop’ ETM packets and not transmit them. Where parallel trace is selected all packets from all sources will be transmitted. The parallel port can be 1 to 4 bits wide and may be clocked independently from the CPU.

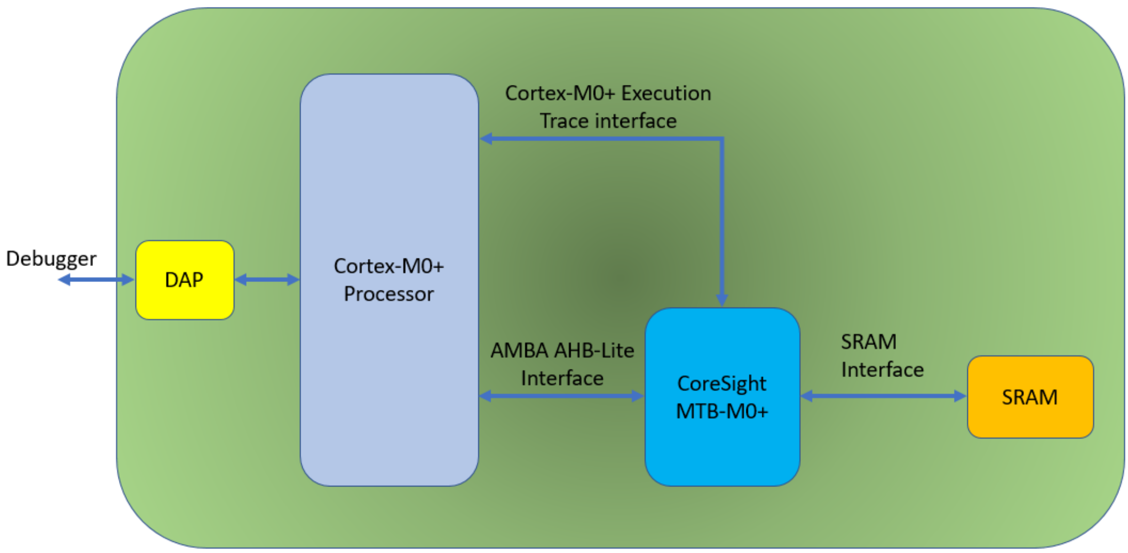
The table below outlines the optional trace blocks that are available for members of the Cortex-M family.

Armv6-M	
Cortex-M0	No trace options.
Cortex-M0+	Optional: MTB Optional: Limited DWT may be included that supports data breakpoints and PCSampler.
Cortex-M1	No trace options.
Armv7-M	
Cortex-M3	Optional: ITM, DWT, ETMv3.
Cortex-M4	Optional: ITM, DWT, ETMv3.
Cortex-M7	Optional: ITM, DWT, ETMv4 (optional with full data trace).
Armv8-M	
Cortex-M23	Optional: DWT. Optional: ETMv3 or MTB. The two are mutually exclusive.
Cortex-M33	Optional: ITM, DWT, ETMv4, MTB. Designs can support both MTB and ETM in the same device.
Cortex-M35P	Optional: ITM, DWT, ETMv4, MTB. Designs can support both MTB and ETM in the same device.
Cortex-M55	Optional: ITM, DWT, ETMv4.
Cortex-M85	Optional: ITM, DWT, ETMv4.
STAR	Optional: ITM, DWT, ETMv4.

A graphical overview of the Cortex-M (>= M23) debug components can be seen in the block diagram below.



This is the block diagram for Cortex-M0+ devices.

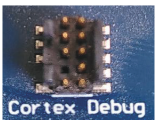


Connectors

Two connectors are commonly used with Cortex-M based systems. The diagram below shows the pinouts and which features are supported by each.

MIPI10-

VREF	1	2	TMS	SWDIO
GND	3	4	TCK	SWCLK
GND	5	6	TDO	SWO
KEY	7	8	TDI	
GND	9	10	nRESET	



Debug Port

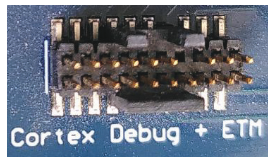
JTAG or SWD
ITM trace via SWV
On-chip trace

Trace Port

4-1 bits wide + 1 pin
Off-chip ITM trace
Off-chip trace (ETM)

MIPI20-T

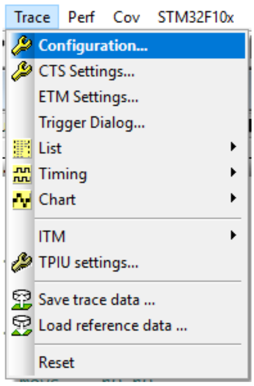
VREF	1	2	TMS	SWDIO
GND	3	4	TCK	SWCLK
GND	5	6	TDO	SWO
KEY	7	8	TDI	
GND	9	10	nRESET	
GND	11	12	TRACECLK	
GND	13	14	TRACE_D[0]	
GND	15	16	TRACE_D[1]	
KEY	17	18	TRACE_D[2]	
GND	19	20	TRACE_D[3]	

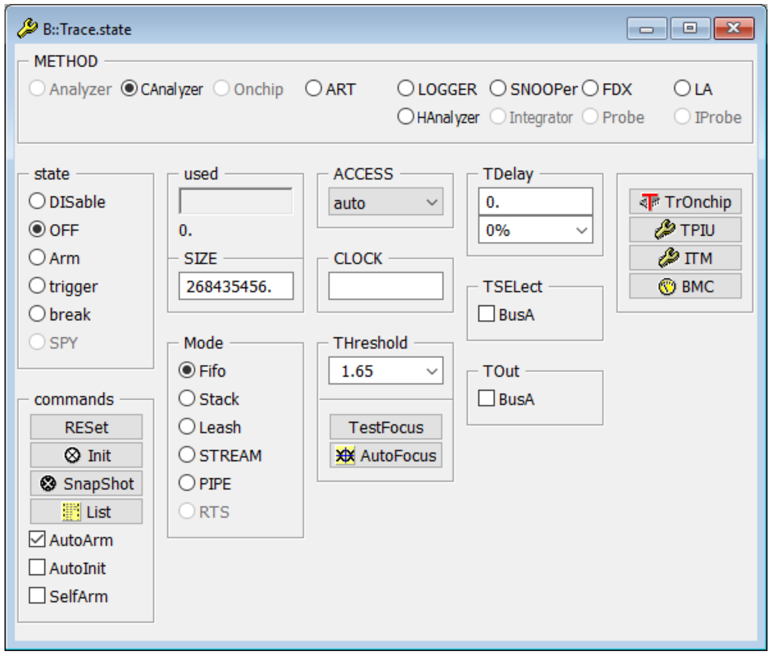


It is imperative to ensure that your development board has the correct connector for the debug features you wish to use. Other connectors are possible.

Basic Trace Configuration

The trace system is configured using the **Trace.state** window, which can be accessed via the command line or from the **Trace** menu.





Most of the features in this window are explained under **"ETM Setup"** in Training Arm CoreSight ETM Tracing, page 6 (training_arm_etm.pdf). Here, we will concentrate on those that affect Cortex-M based systems.

The generated trace data can be stored on-chip in a memory buffer or it can be streamed off-chip to a set of trace tools. On-chip trace storage is often referred to as MTB or Embedded Trace Buffer (ETB). Whilst off-chip trace storage is often referred to as ETM. These are historical and largely irrelevant as the same trace data can be routed to either storage medium.

For off-chip trace (ETM, ITM), select **Trace.METHOD CAnalazyer** or **Trace.METHOD Analyzer**. If the tools do not support one of these modes, it will be grayed out.

For on-chip trace (MTB), select **Trace.METHOD Onchip**.

For off-chip trace, the size will be automatically filled based upon the amount of storage for trace data there is in the chip or the tools. This can be over-ridden for off-chip trace if a smaller amount of data is required, simply type the new value into the field or use the command **Trace.SIZE** *<n>*. For on-chip trace (MTB), the size must be specified.

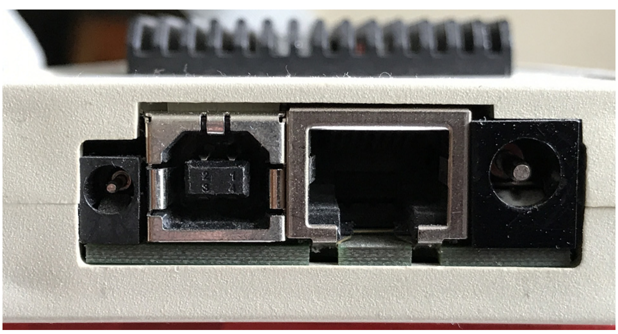
The Mode setting affects the way that the trace buffer is managed.

FIFO	The trace data is stored in the buffer memory in the TRACE32 trace hardware. When the buffer fills, earlier records are over-written with the new data. When the trace sampling stops, the maximum amount of trace history is available.
Stack	The trace data is stored in the buffer memory in the TRACE32 trace hardware. When the buffer is almost full, sampling stops but the target is left running.
Leash	The trace data is stored in the buffer memory in the TRACE32 trace hardware. When the buffer is almost full, trace sampling stops and the target is halted.
STREAM	The buffer memory in the TRACE32 trace hardware is used as a large FIFO and the trace data is streamed to a file on the host system's hard drive. The amount of trace captured is limited by either the size of the hard disk partition, a user specified amount or (2^64)-1 frames.
PIPE	The buffer memory in the TRACE32 trace hardware is used as a large FIFO and the trace data is streamed to an application which reads a pipe or FIFO on the host system. The trace stream must be processed in real-time; no storage is performed. The trace stream can be effectively unbounded.

The **Trace.CLOCK** needs to be set to the core clock frequency.

The **Trace.TDelay** setting is used to position a trigger point somewhere other than the start (STACK of LEASH mode) or end (FIFO mode) of the buffer. It takes either a percentage or a number of records to capture after the trigger condition is reached. The trigger condition is a breakpoint of type TraceTrigger. This does not stop the core but merely triggers the trace system; of course the action of the trigger may also be to stop the core. More information can be found here (**"Trace Buffer Management"**, page 10.

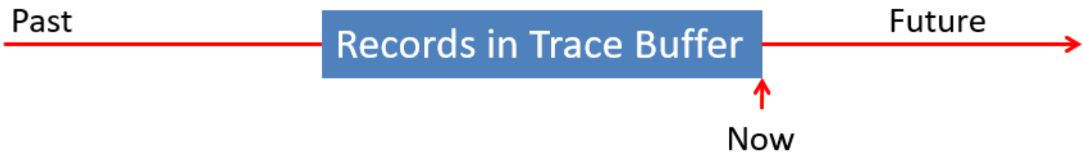
The **Trace.TSELect** and **Trace.TOut** settings allow for an external trigger to be used. This can be the BusA signal which is present on the PODBUS as well as the Trigger pin on the outside the TRACE32 debug hardware. From left to right: Trigger pin, USB connector, (optional) Ethernet connector, power connector.



The Trigger pin is controlled via the Trigger Bus Window (**TrBus**). Selecting **<trace>.TSELect BusA** will use the Trigger pin as the trace trigger event. Selecting **<trace>.TOut BusA** will trigger the pin when the trace system triggers or stops tracing.

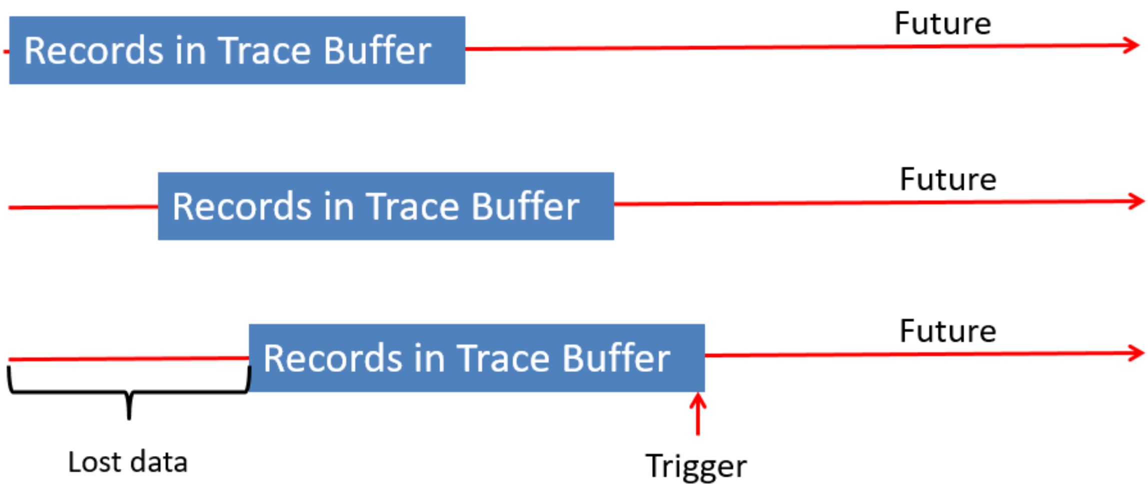
Trace Buffer Management

The trace buffer acts as storage for all of the information generated by the core. When the target stops, the records in the trace buffer allow the user to look back into the past. It is impossible to look back beyond the start of the trace buffer.



If the sampling is halted before the buffer fills a smaller amount of data is collected.

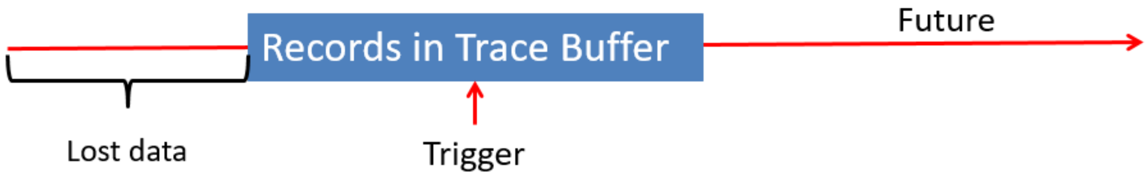
Once the buffer has filled, the time window moves forwards to accommodate new samples. When the trigger occurs, the trace stops sampling. If more than a buffer's worth of data had been generated some of it would have been lost.



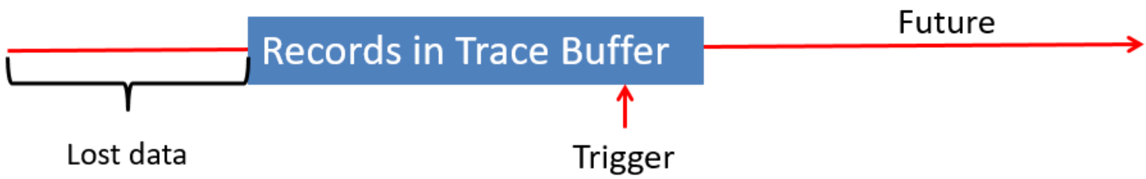
To capture more data:

- Use tools with a larger trace buffer. Eventually, this option will become exhausted.
- Stream the trace data to your local hard drive. More information can be found **"ETM Stream Mode"**, page 17
- Stream the trace data to a local application for real-time processing. The trace data is not stored by TRACE32. More information can be found here: **"Pipe Mode"**, page 74.
- Use the on chip features to filter the generated trace data to view only events of interest. More information can be found here: **"Trace Filtering"**, page 24.

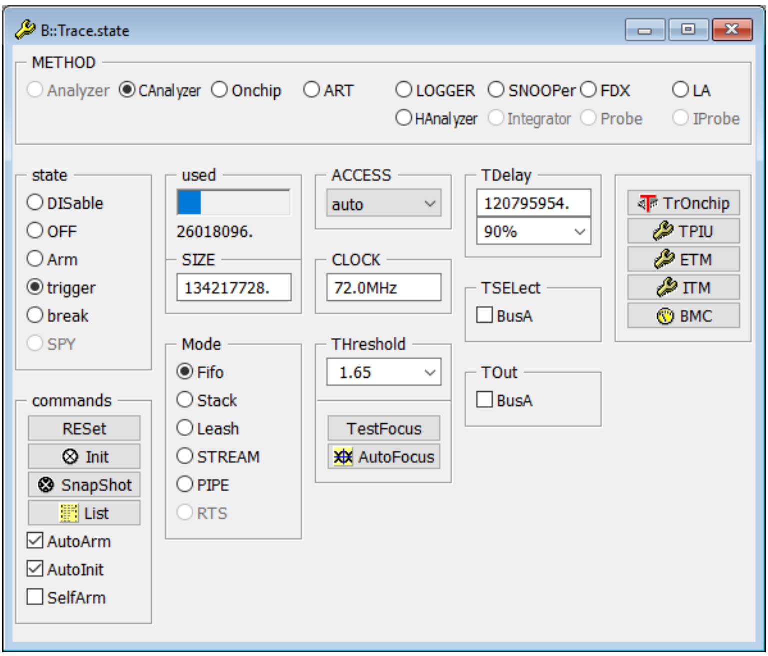
The DWT can generate a Trigger packet which causes the trace tools to trigger on an event. The **Trace.TDelay** option controls how the tools will react. It holds a portion of the trace buffer in reserve to be filled up only after the trigger event occurs. Setting the value to 50% will place the trigger event in the middle of the trace buffer, giving the user an equal amount of trace buffer dedicated to events before the trigger and events after the trigger.



Setting the point to 10% will ‘reserve’ 10% of the buffer to store events that occurred after the trigger.

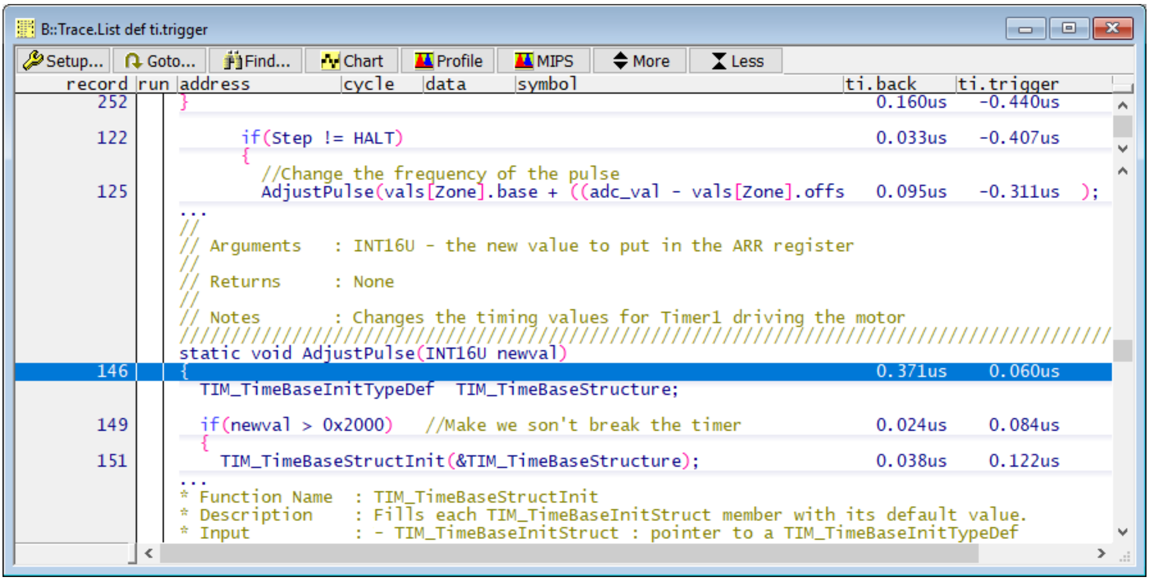


When the trigger event occurs, the trace state changes to “trigger”, as seen in the image below.



When the amount of trace that corresponds to the **Trace.TDelay** setting has been captured the trace will stop sampling and the mode will change to break.

The Trigger point can be located in the **Trace.List** window and adding the Time.TRIGGER column shows all trace events timed relative to the trigger event: events after have a positive time index; events before have a negative time index.



MTB Program Flow Trace

The Micro Trace Buffer (MTB) provides basic program flow trace capabilities for cores with limited resources. It is not designed to compete with ETM or PTM. The trace data is stored in a user configurable area of RAM at runtime. External debug tools can be used to start or stop the trace. The size and location of the RAM storage is configurable in software, allowing the resources to be reclaimed from a development build when they are no longer needed. The MTB can be programmed by the debug tools to cause the processor to enter a halt state when the buffer becomes full.

TRACE32 treats the MTB as an on chip trace buffer. The On Chip family of commands apply and **Trace.METHOD Onchip** should be set.

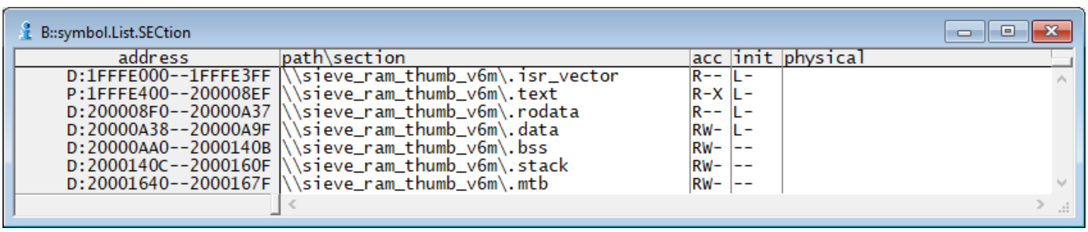
TRACE32 needs to know the base address and size of the buffer. This is done with the commands:

- **Onchip.TBADDRESS** *<address>*
- **Trace.SIZE** *<size>*

The MTB is usually placed in a separate memory region in the linker control file. This makes it easy for TRACE32 to locate the base address with something like this.

```
Trace.TBADDRESS ADDRESS.OFFSET(sYmbol.SECADDRESS(.mtb))
Trace.SIZE 64.
```

A list of all sections can be obtained with the command **sYmbol.List.SECtion** and my look like this.


B::symbol.List.SECtion

address	path\section	acc	init	physical
D:1FFFE000--1FFFE3FF	\\sieve_ram_thumb_v6m\.isr_vector	R--	L-	
P:1FFFE400--200008EF	\\sieve_ram_thumb_v6m\.text	R-X	L-	
D:200008F0--20000A37	\\sieve_ram_thumb_v6m\.rodata	R--	L-	
D:20000A38--20000A9F	\\sieve_ram_thumb_v6m\.data	RW-	L-	
D:20000AA0--2000140B	\\sieve_ram_thumb_v6m\.bss	RW-	--	
D:2000140C--2000160F	\\sieve_ram_thumb_v6m\.stack	RW-	--	
D:20001640--2000167F	\\sieve_ram_thumb_v6m\.mtb	RW-	--	

Here, the base address of the `.mtb` section can be clearly seen.

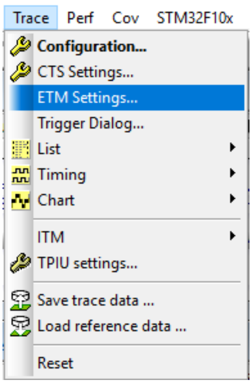
There is no timing information available for the program flow trace; no statistical or analytical operations may be performed.

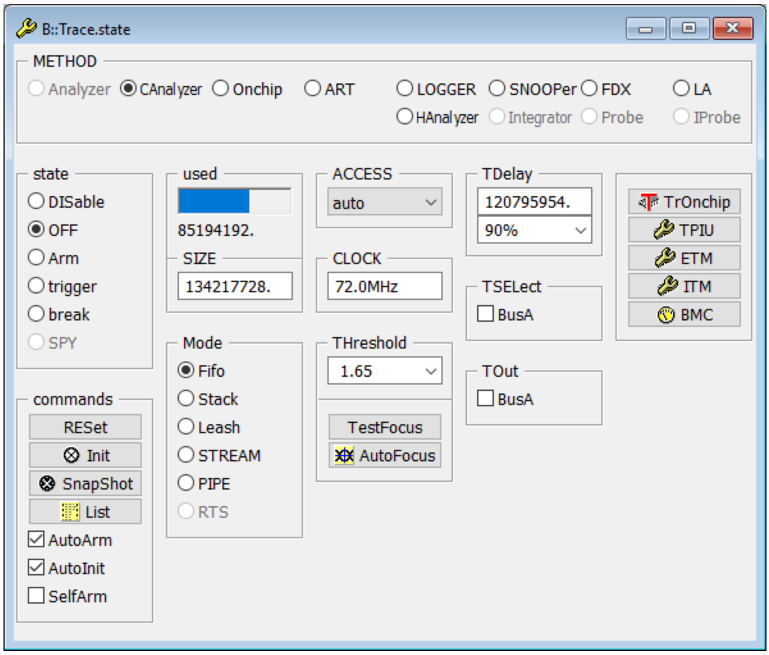
ETM Program Flow Trace

When most people think of trace, what they mean is ETM. This is a program flow trace that is generated by the Cortex-M core. This section should be read in conjunction with .**“Training Arm CoreSight ETM Tracing”** (training_arm_etm.pdf) which provides a more general overview of ETM and the kinds of analysis that can be performed on the captured data.

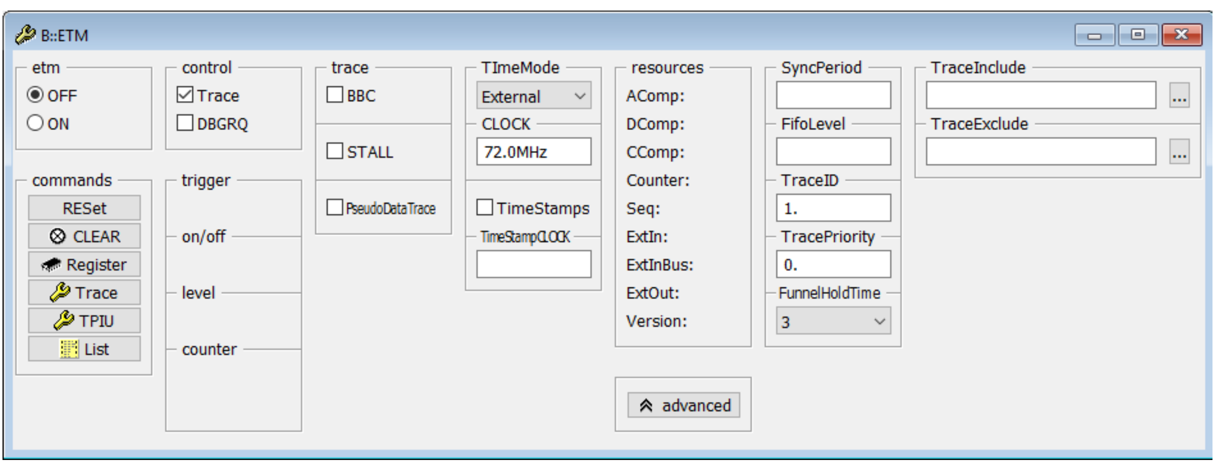
ETM Configuration

ETM trace data requires a fully functional TPIU to be able to pass the data to the final destination (off-chip or on-chip). The ETM features are configured in the ETM window. It can be accessed from the Trace menu, from the Trace Configuration window or by using the command **ETM**.





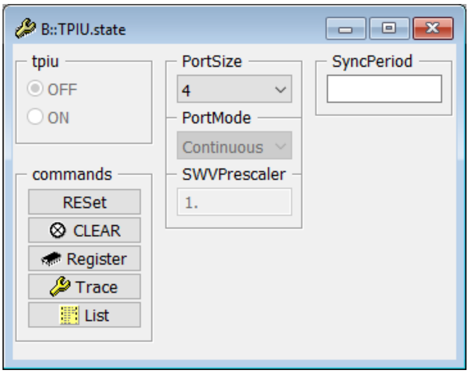
It looks like this.



Next, the TPIU may need to be configured. This can be accessed via:

- Selecting TPIU settings from the Trace menu
- Clicking the TPIU button in the **Trace.state** window
- Using the command **TPIU.state**

The **TPIU.PortSize** value can be changed to 1bit wide, 2bits wide, 4bits wide or Serial Wire Viewer (SWV). The default value of **TPIU.SyncPeriod** is every 1024 packets. This can be changed by the user and configures how frequently synchronisation packets will be emitted by the target.



NOTE:	Some devices lack enough pins to bring out all of the possible signals on the chip. When this occurs, the user will need to ensure that the correct pin muxing is enabled to route the TPIU signals from the chip to the debug header on the board. Check your chipset documentation! Other things to check are: • Your application does not use a set of chip features that prevent the correct multiplexing of the trace port pins. • Any chosen RTOS does not re-multiplex the trace port pins as part of its setup/boot procedures. • Any third party libraries or drivers do not change the trace port pin multiplexing. • Some devices multiplex the trace port pins with GPIO. Ensure the pin direction is correctly set. Also, check the maximum possible speed of the GPIO pins used; the trace will not emit data faster than this.

To capture Program Flow trace using ETM:

1. Set **ETM.ON**. Via command or GUI.
2. Set **Trace.CLOCK**.
3. Set either **Trace.Method Analyzer** or **Trace.Method CAnalyzer**. Unsupported variants will be grayed out.
4. Set **Trace.OFF**.
5. Set **Trace.AutoArm ON**. This will cause the sampling to start and stop in synchronisation with the target.
6. Set **Trace.AutoInit ON**. This will clear the existing buffer before sampling new data.
7. Set the desired **Trace.Mode** (Stack, FIFO or Leash). More information can be found here **“Trace Buffer Management”**, page 10.
8. Start the target running. It can be halted manually or via a breakpoint.

Details on how to display the results can be found here **“Displaying the Results”**, page 18 and details on analyzing the trace data can be found here **“Analyzing the Results”**, page 30.

ETM Stream Mode

To capture trace data for extended periods of time may require storing more data than can be held in the buffer in the tools. For example, the µTrace's (MicroTrace) 128Mbyte buffer holds around 20-30seconds of trace of a bare metal demo program running at 66MHz on a Cortex-M3.

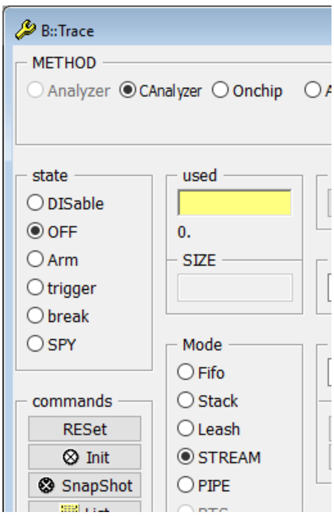
Trace data can be streamed to a local file system for analysis.

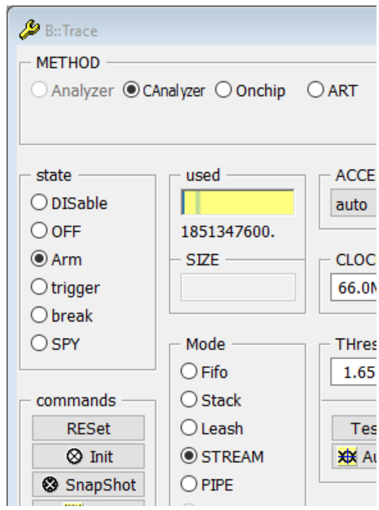
This requires a 64bit host OS and a 64bit version of TRACE32. By default, the stream file is stored to the temporary directory specified in the configuration file (usually ~~/config.t32). This can be found with the command:

```
PRINT OS.PresentTemporaryDirectory()
```

A new stream file can be set with **Trace.STREAMFILE** *<file>*

Set the Trace mode to **STREAM** and the used bar switches from white to yellow as seen in the picture, below left.



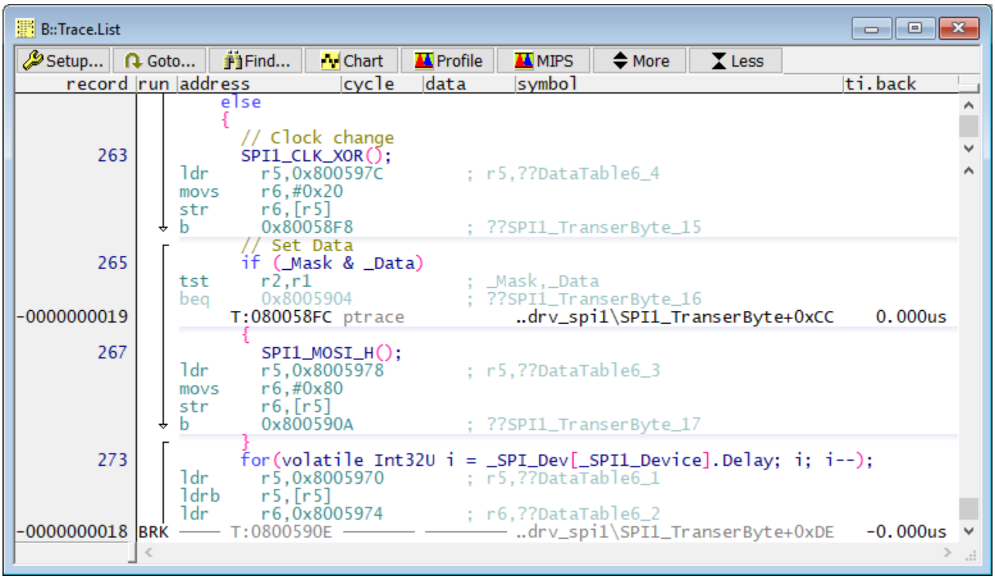


The used bar is now used to show the fill level of the tools' internal buffer. The buffer is now used as a large FIFO to smooth out any peaks in the trace flow before streaming to the host PC. The number underneath is the number of trace frames captured. The image (above, right) shows the state after around 5 and a half minutes of trace capture.

When TRACE32 is closed any stream files are automatically deleted from the host file system. Captured trace data can be saved using **CAnalyzer.STREAMSAVE** *<file>* so that it remains on the local file system after TRACE32 has closed. When TRACE32 is re-started, the saved data can be re-loaded with **CAnalyzer.STREAMLOAD** *<file>*. The data is saved in a raw format and it is not expected that users will be able to interpret this. More information about working with loaded trace files can be found here **"Off-line Analysis"**, page 53.

Displaying the Results

Once the trace data has been captured a list of all events can be obtained by using the **Trace.List** command or by clicking the List button in the Trace configuration window. The results will look like this.

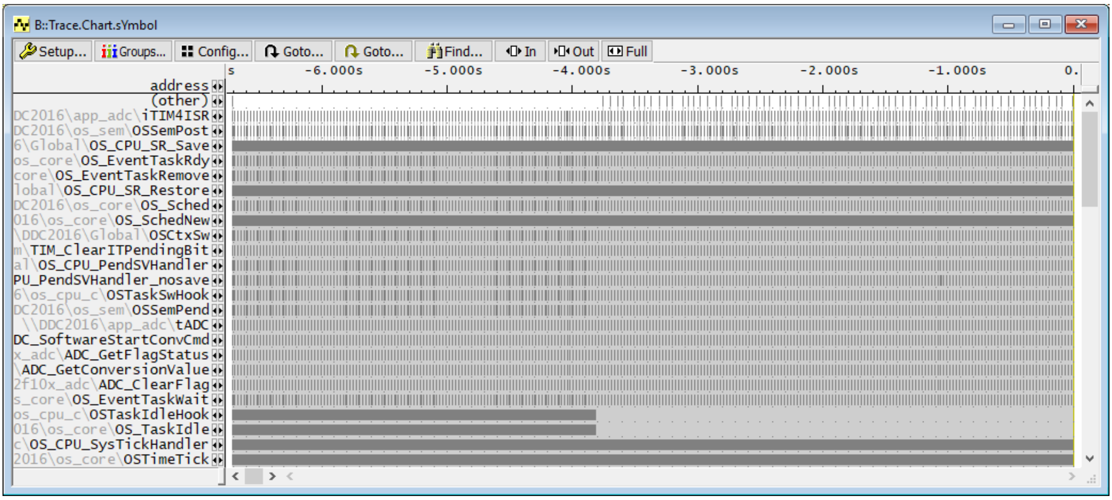


The columns shown are the default values and others can be added. See **Trace.List** for more information.

Along the top of the window is a series of buttons that give access to other features.

Setup	Opens the **Trace** configuration window.
Goto	Opens the **Goto** window which allows the user to jump to points of interest in the trace listing. For more details on filtering **“Trace Filtering”**, page 24 and searching **“Trace Searching”**, page 21.
Find	Opens the search dialog and allows the user to search for events within the trace buffer.

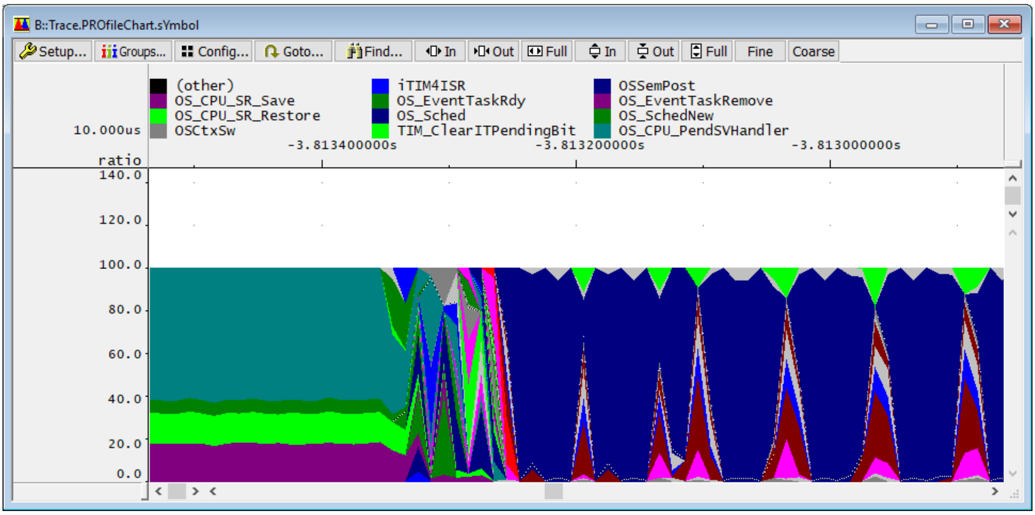
Chart Displays a graphical view of functions on the y-axis and time on the x-axis. Bear in mind that: trace captured in FIFO mode will all have a negative time index; if a Trigger has been used trace captured before the trigger event will have a negative time index.



More information on driving the Chart windows can be found here **“Graphical Navigation”**, page 28.

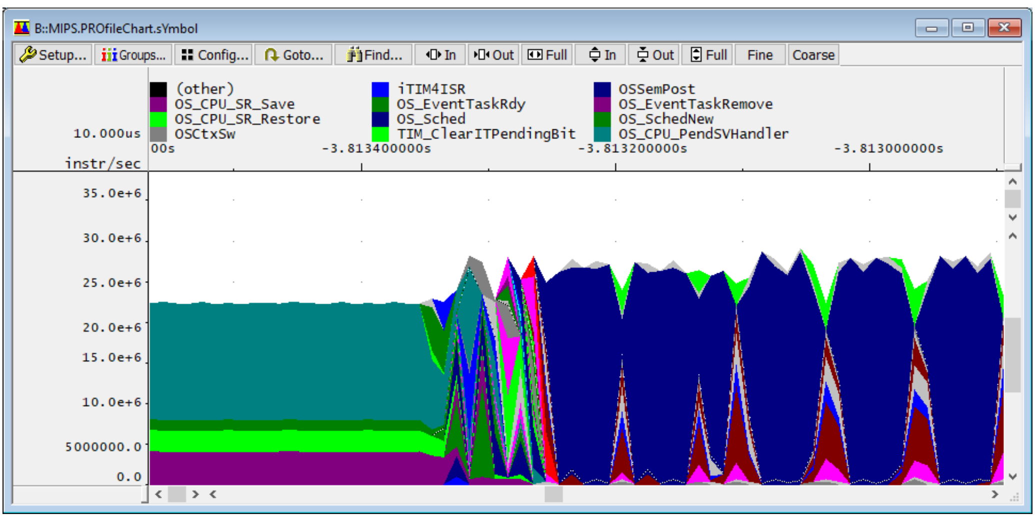
Profile

Shows a graphical representation of cpu usage by function over time. Small time slices are used to calculate the percentage of cpu time used by each function that occurs during the slice. See **"Graphical Navigation"**, page 28 for more information on interacting with this window. Each function is represented by a colored block. Clicking on the colored block opens a pop-up which displays more information about the item.



MIPS

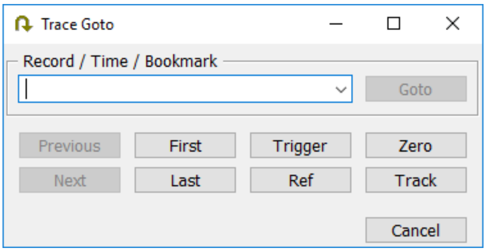
Provides a graphical representation of how many instructions per second (over a given time slice) were used executing each function. **"Graphical Navigation"**, page 28.



Clicking the **More** or **Less** buttons will add or remove certain items from the display. In four steps, it moves between showing all CPU cycles, including dummies, to just showing the HLL code.

Trace Searching

The Trace Goto dialog looks like this and can be accessed by clicking the “Goto” button on any trace view window.



It provides a convenient way of jumping to certain points within the **Trace.List** window.

A user entered record number or time index can be entered. If a bookmark has been created this can be located here too. More information about **BookMark**s can be found by following this link.

Jump to the **First** captured trace record.

Jump to the **Trigger** event. More information on the trigger can be found here: **“Trace Buffer Management”**, page 10.

Jump to time index of **zero**. This is the first event in Stack or Leash mode and the last event in FIFO mode. Right-clicking any trace window allows a user created zero point to be defined. This will be used as the zero point in all future calculations. The command **Trace.ZERO** can also be used to set the marker.

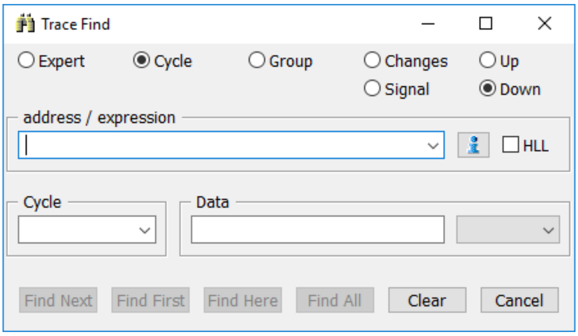
Jump to the **Last** captured trace event.

Jump to a user created reference point. Right-clicking any trace display window will allow a reference point to be manually created. The command **Trace.REF** may also be used to set the reference marker.

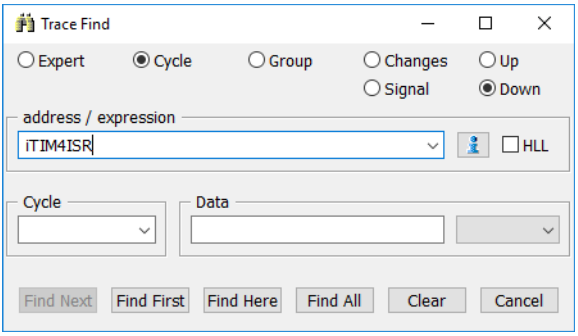
Selecting **Track** will jump to the last position that the user placed a cursor at in any trace window.

Any windows that are opened with the /Track option will also jump to the selected point.

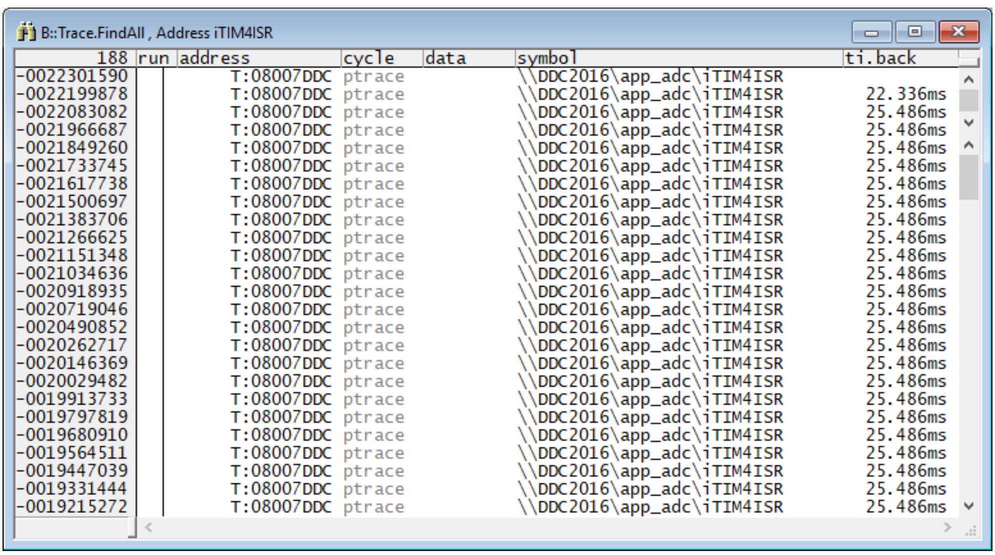
It is possible to search the contents of the trace using the Find window which can be accessed by clicking the Find… button on the **Trace.List** window and looks like this.



For example, to find all occurrences of function `iTIM4ISR`, you would enter:



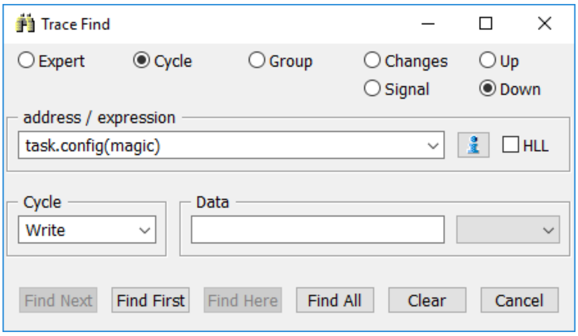
And click the Find All button. A list of all entries to the function `iTIM4ISR` are shown. The **`ti.back`** column shows the time between calls to this function. More information on trace timing can be found here: **"Timing"**, page 45.


B::Trace.FindAll , Address iTIM4ISR

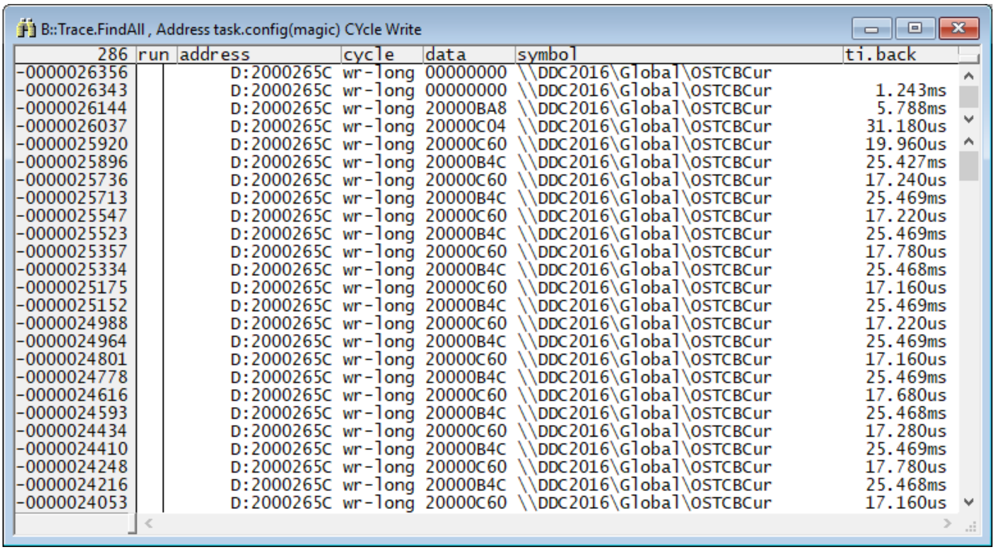
188	run	address	cycle	data	symbol	ti.back
-0022301590		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	
-0022199878		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	22.336ms
-0022083082		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021966687		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021849260		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021733745		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021617738		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021500697		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021383706		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021266625		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021151348		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0021034636		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0020918935		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0020719046		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0020490852		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0020262717		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0020146369		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0020029482		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0019913733		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0019797819		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0019680910		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0019564511		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0019447039		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0019331444		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms
-0019215272		T:08007DDC	ptrace		\\DDC2016\app_adc\iTIM4ISR	25.486ms

Clicking any item in this window will cause the **Trace.List** window to jump to the same point within the trace buffer.

Cortex-M ETM trace is program flow trace only; there are no data trace items. These can be injected via the DWT/ITM. More information can be found here: **"Data Watchpoint and Trace Unit"**, page 55. To search for all task switches (writes to **TASK.CONFIG(magic)**) use:



Then click **Find All**. A list of task switches will be displayed. The values in the **ti.back** column show how long each task or thread was running for before it was switched.



286	run	address	cycle	data	symbol	ti.back
-0000026356		D:2000265C	wr-long	00000000	\\DDC2016\Global\OSTCBCur	
-0000026343		D:2000265C	wr-long	00000000	\\DDC2016\Global\OSTCBCur	1.243ms
-0000026144		D:2000265C	wr-long	20000BA8	\\DDC2016\Global\OSTCBCur	5.788ms
-0000026037		D:2000265C	wr-long	20000C04	\\DDC2016\Global\OSTCBCur	31.180us
-0000025920		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	19.960us
-0000025896		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.427ms
-0000025736		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.240us
-0000025713		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.469ms
-0000025547		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.220us
-0000025523		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.469ms
-0000025357		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.780us
-0000025334		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.468ms
-0000025175		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.160us
-0000025152		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.469ms
-0000024988		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.220us
-0000024964		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.469ms
-0000024801		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.160us
-0000024778		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.469ms
-0000024616		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.680us
-0000024593		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.468ms
-0000024434		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.280us
-0000024410		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.469ms
-0000024248		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.780us
-0000024216		D:2000265C	wr-long	20000B4C	\\DDC2016\Global\OSTCBCur	25.468ms
-0000024053		D:2000265C	wr-long	20000C60	\\DDC2016\Global\OSTCBCur	17.160us

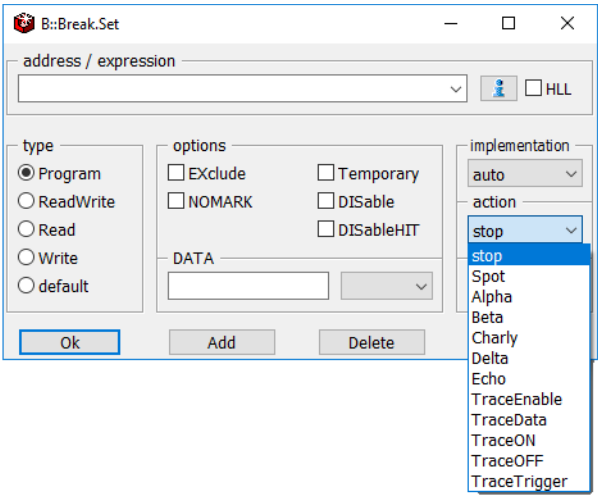
More information on tracing task or thread switches can be found here: **"Task/Thread switch Tracing"**, page 61.

The Find window is a wrapper for the command **Trace.FindAll**. Follow the link for a detailed explanation.

Trace Filtering

Events can be filtered by using the DWT. The DWT filtering takes place on the chip and is non-intrusive to the target's runtime performance. Filtering to capture only those events of interest can relieve pressure on the internal FIFO in the chip and extend the length of time for which data can be captured.

The DWT comparators can be conveniently programmed by using the TRACE32 breakpoint features. Several event filter types are available and can be assigned by using the **Break.Set** command or dialog.



The last five entries on the drop-down menu will affect how trace is generated.

TraceEnable	When a breakpoint of this type is set, a match will generate a trace packet for this event only. All other ETM trace generation is suspended.
TraceData	A match to this event will cause a DWT data trace packet to be emitted.
TraceON	ETM trace will be switched on when this event matches.
TraceOFF	ETM trace will no longer be generated when this event matches.
TraceTrigger	When this event matches, a Trigger packet will be emitted.

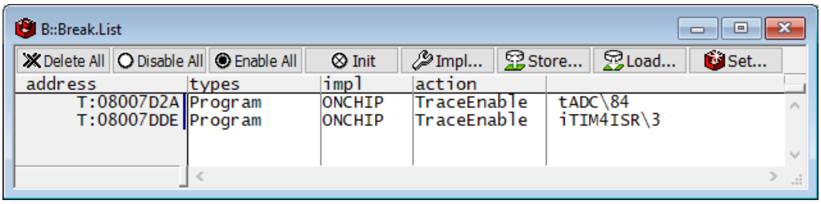
By using a matching pair of TraceON and TraceOFF markers it is possible to restrict the trace to only an area of interest and nothing else. This will extend the length of time for which meaningful data can be captured at the expense of not capturing all events.

The Trigger packet causes the trace tools to trigger on an event. The **Trace.TDelay** option in the Trace configuration window controls how the tools will react. It holds a portion of the trace buffer in reserve to be filled up only after the trigger event occurs. Setting the value to 50% will place the trigger event in the middle of the trace buffer, giving the user an equal amount of trace buffer dedicated to events before the trigger and events after the trigger.

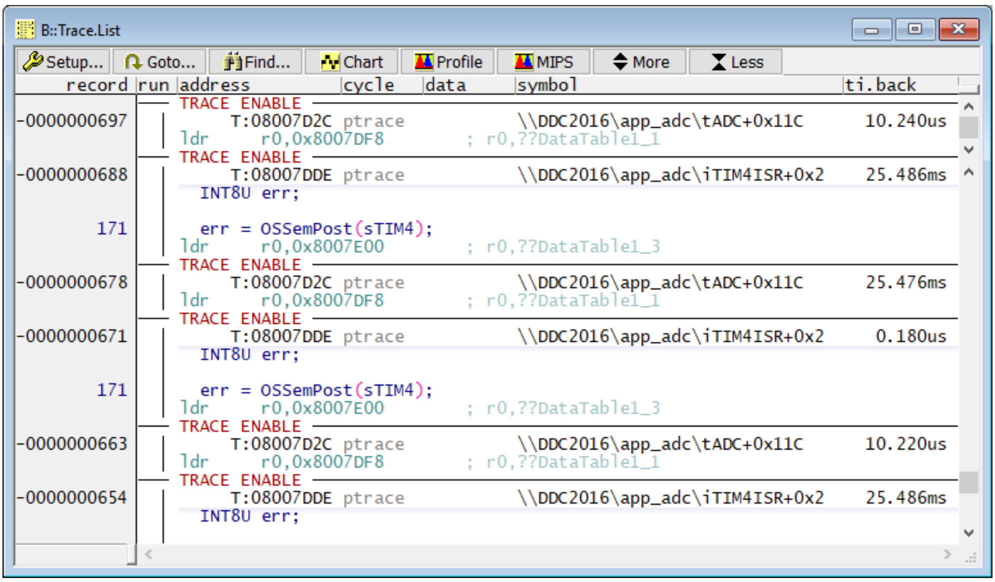
For example, the time taken between an Interrupt Service Routine executing and the task designed to respond to that interrupt being scheduled.

When using Filters like this it is recommended to switch the timing mode to be cycle accurate (**ETM.TImeMode** **CycleAccurate**. Some early Cortex-M cores do not support a cycle accurate timing model. In these cases, set **Trace.PortFilter** **ON**.

Set a breakpoint on the ISR and the wake-up part of the task of type **/TraceEnable**. The list in the picture below shows an example of marking two events.



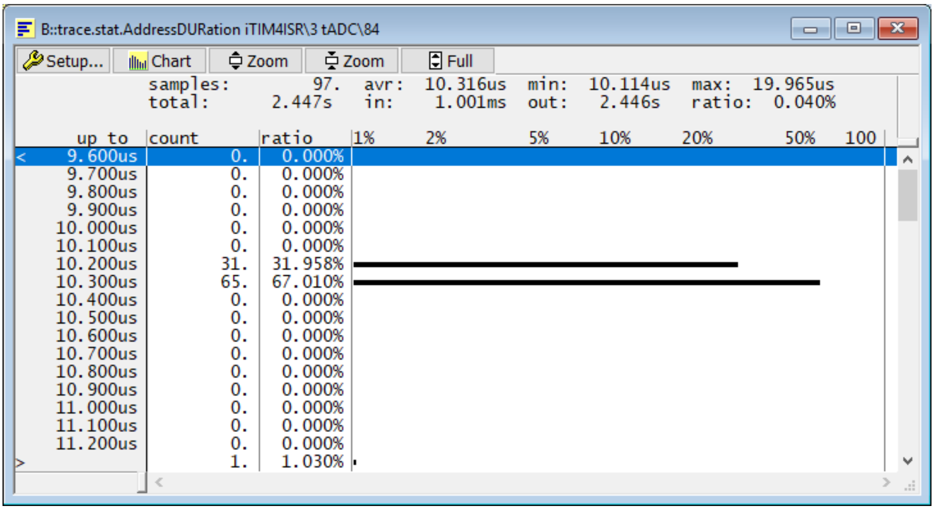
Listing the contents of the trace buffer with **Trace.List** shows only the filtered events.



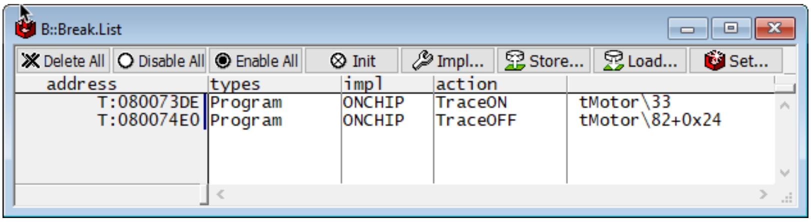
To show a histogram of distances between two points use the command **Trace.STATistic.AddressDURation**. For example:

To measure the time between the last instruction of the ISR and the first instruction of the task that wakes to deal with the ISR. All of the results are between 10.150 us and 10.350 us.

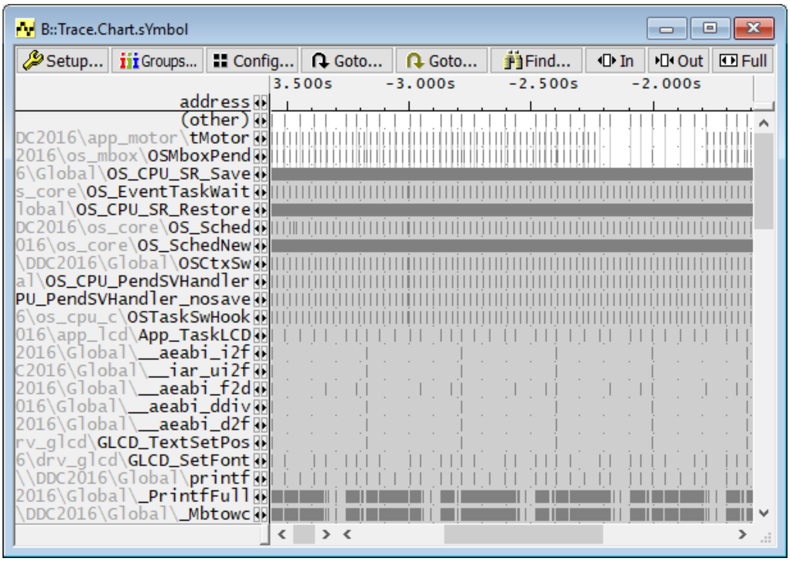
Trace.STATistic.AddressDURation iTIM4ISR\3 tADC\84



A pair of breakpoints can be used: a breakpoint of type **/TRACEON** to start tracing at a particular event; and a breakpoint of type **/TRACEOFF** to stop tracing at this event. For example: To trace the main loop of a task (`tMotor`), set a pair of breakpoints to mark the on and off positions.



Run the target to collect the trace data and then view the results. As we can see from the image below, more has been captured than just the task we were interested in.



By marking a start and stop point, all code executed between these two points will be sampled. This includes interrupts, sub-functions, task switches, etc.

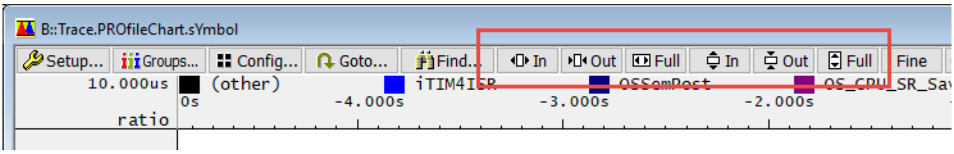
If the application is running as a small bare metal loop then this approach works very well. If the application has an RTOS or scheduler then more data may be collected than anticipated.

Setting a ranged breakpoint does not help as the DWT will generate a trace event for each branch within the range. This will result in a lot of FIFO overflows. To view only the code within the task of interest it is better to use a group (see **“Trace and Groups”**, page 40) and filter on that afterwards. If the amount of trace is too large to fit in the tools’ internal buffer, consider using Stream mode: **“ETM Stream Mode”**, page 17.

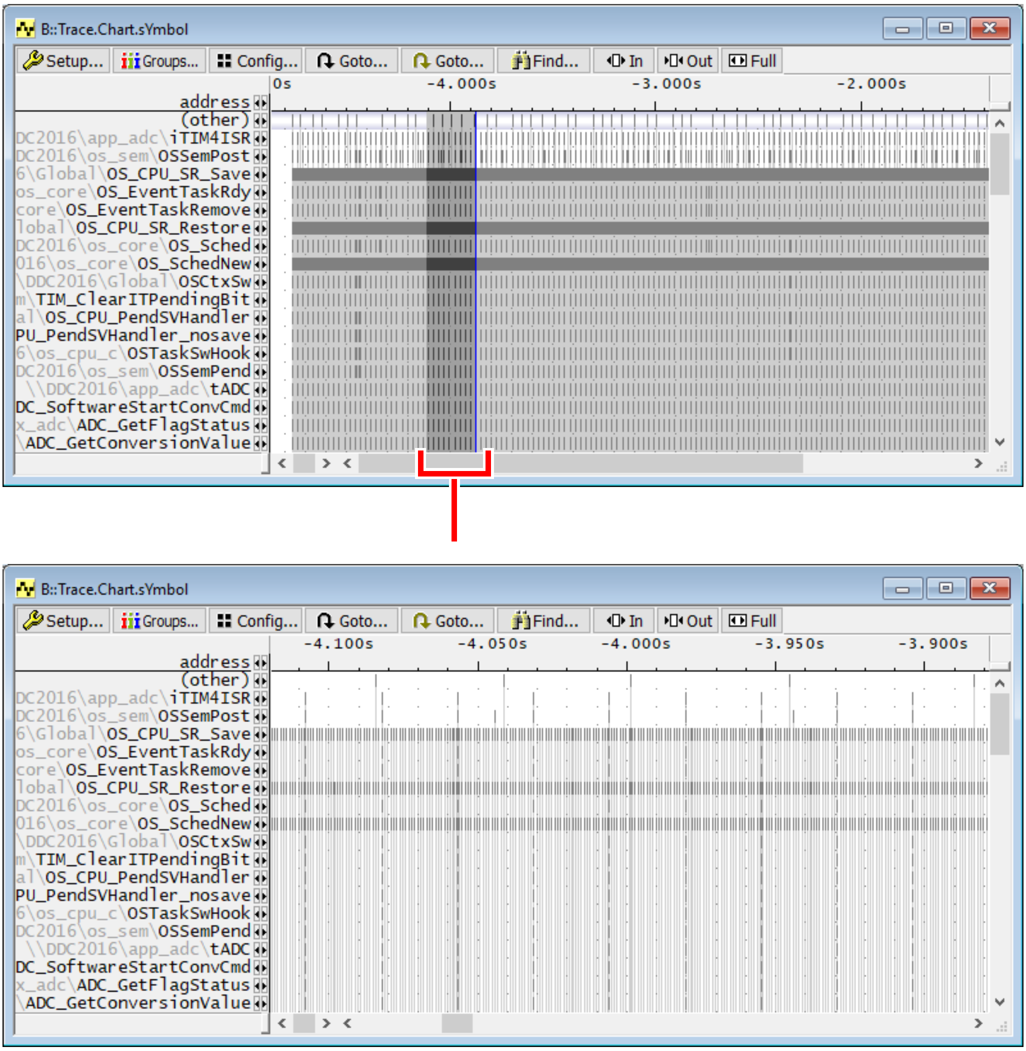
Graphical Navigation

All graphical views of the trace data have some common navigation features.

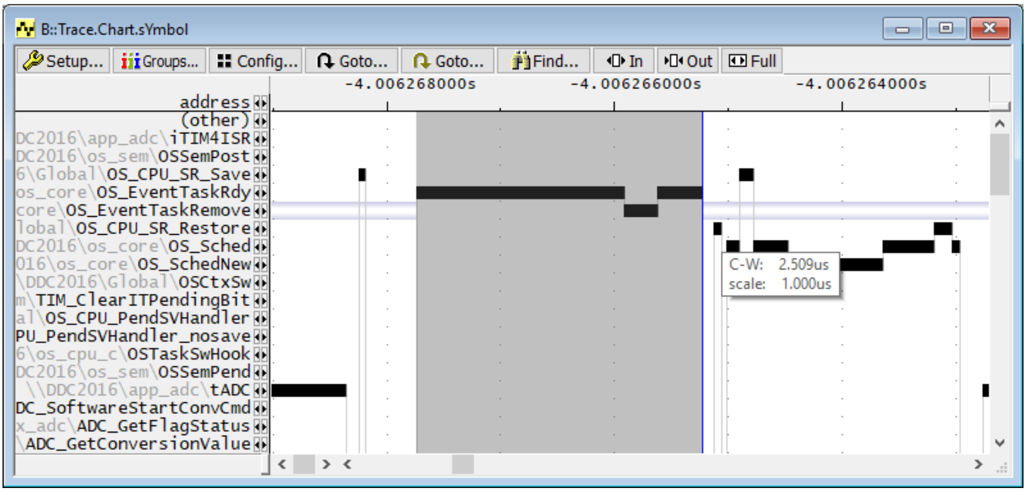
Zoom/pan using the In/Out/Full buttons. Some can zoom in two dimensions, others in only one.



Click and drag to select an area. Click within the selected area to zoom that to full screen.



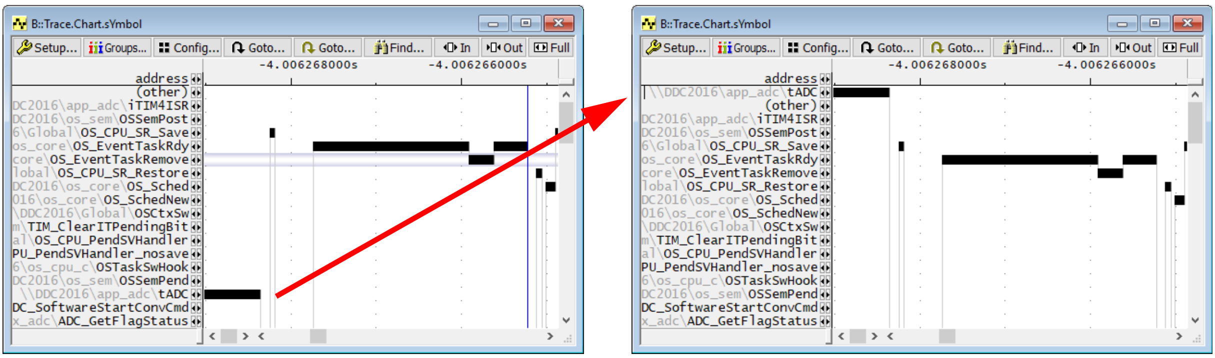
Click and drag to time a region.



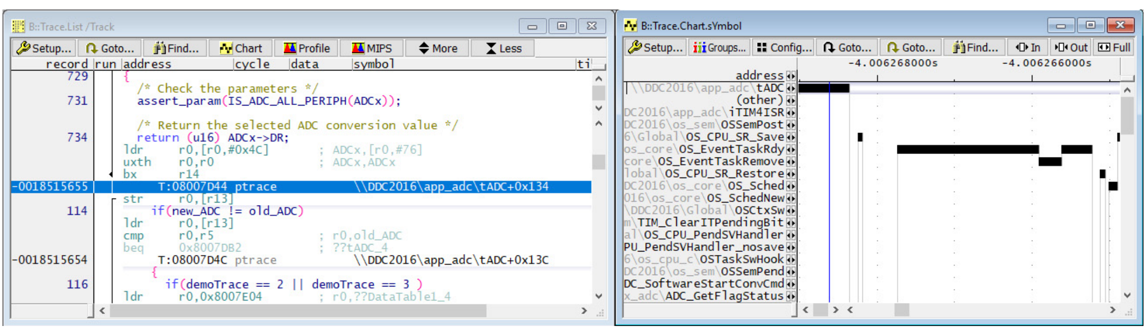
Click and use the mouse scroll wheel: scroll up to zoom in and scroll down to zoom out.

Double-click but don't release the second mouse button press. Whilst holding the button, move the mouse up to zoom in and down to zoom out. Move left and right to scroll through the window along the x-axis.

The **Trace.Chart.sYmbol** window can be ordered along the y-axis. Drag the row to the required position.

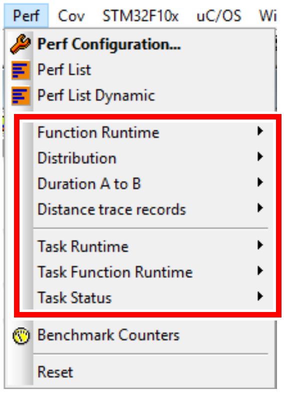


Adding the **/Track** option to any window showing trace data allows it to snap to a cursor placed in any other trace display window.



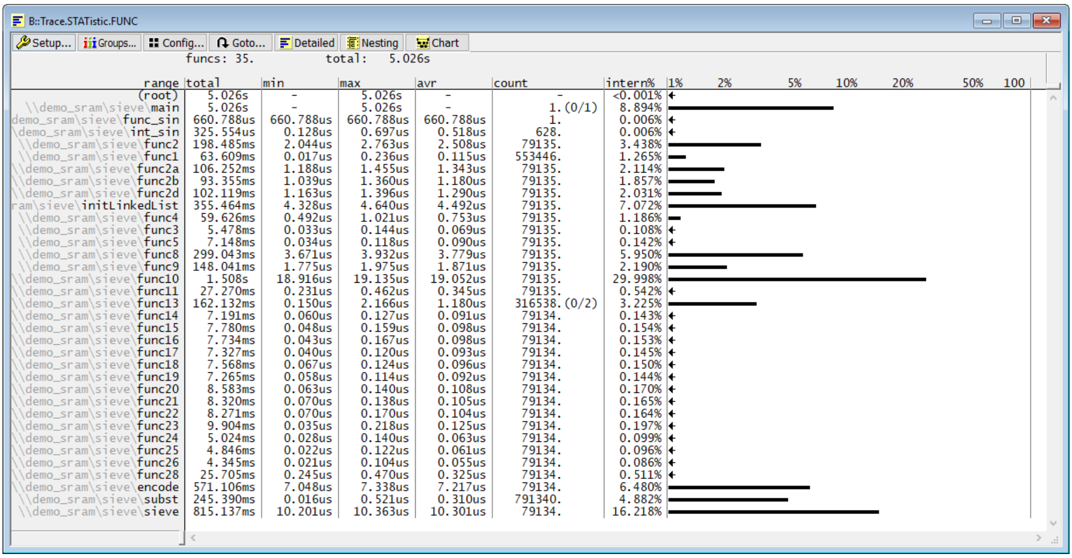
Analyzing the Results

The collected trace data can be analyzed to show a lot of performance information about the code running on the target. This is handled by the **PERFormance** commands and more information can be found by following the link. The features can also be accessed from the **Perf** menu; the highlighted features will be discussed in this manual.



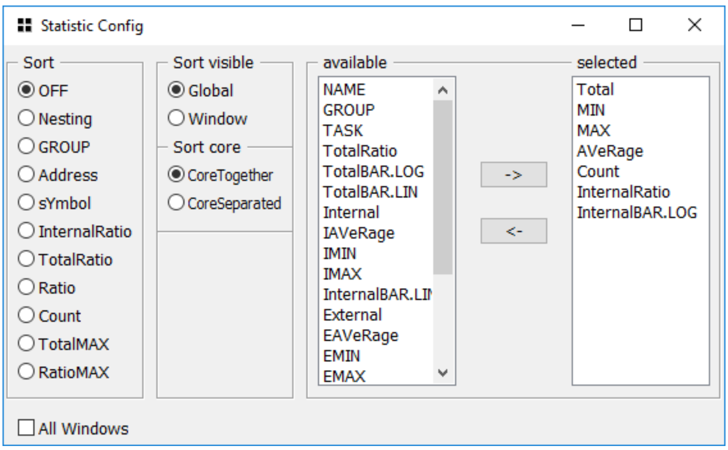
These items work with collected trace data.

The items in this menu deal with how long each function and/or sub-function takes to execute and how many times it is called. A basic display looks like this:

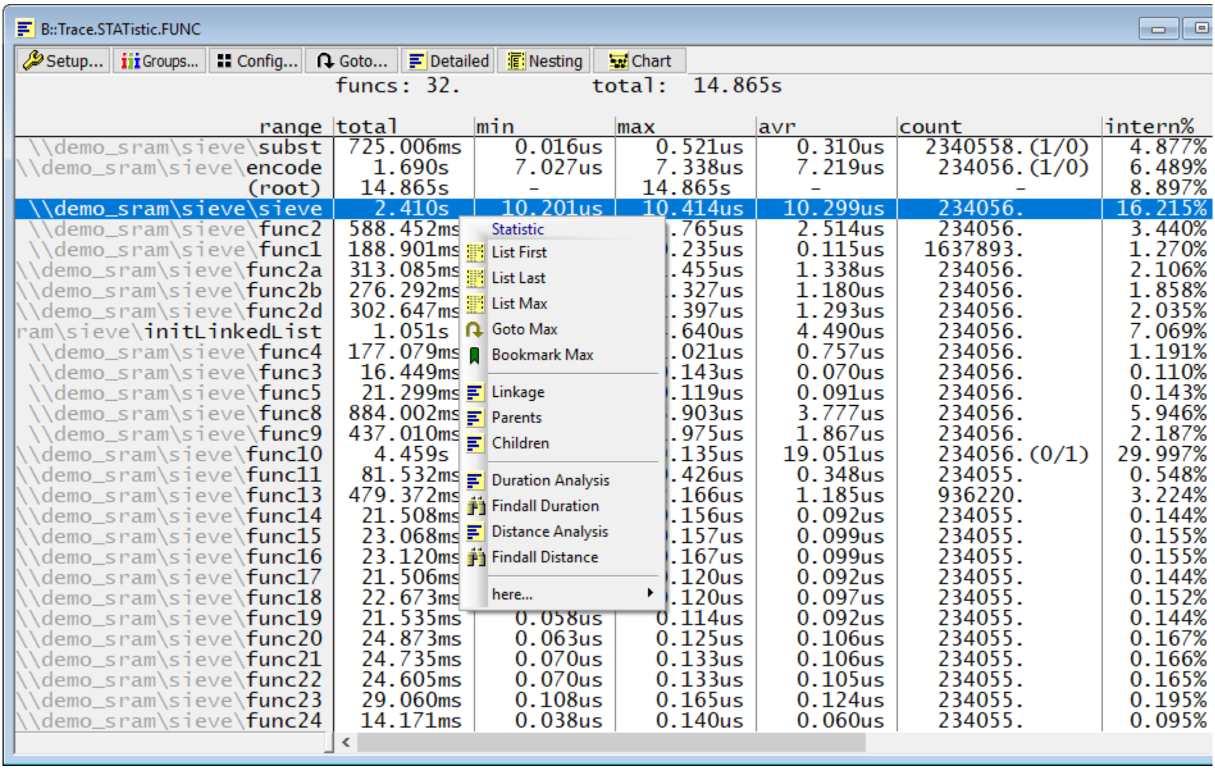


This shows a list of functions that were sampled during the trace collection period and runtime statistics (including min, max and mean) for each.

Many other columns can be added and the list can be sorted in a number of ways. Clicking the “Config” button gives access to this dialog, which looks like this.

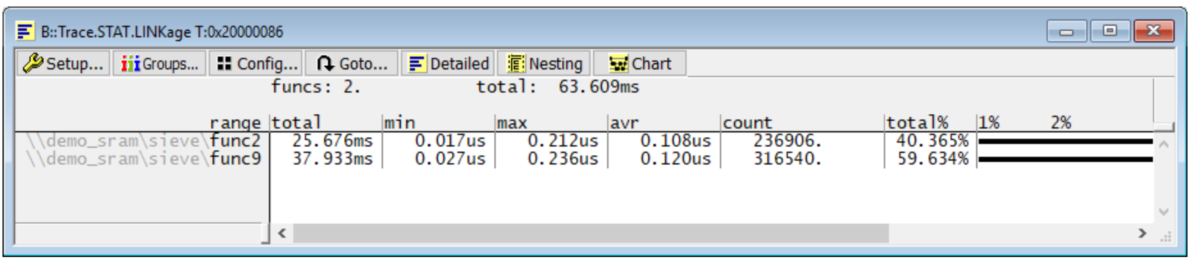


Each line in the list has a right-click menu associated with it. This provides access to more detailed analyses.

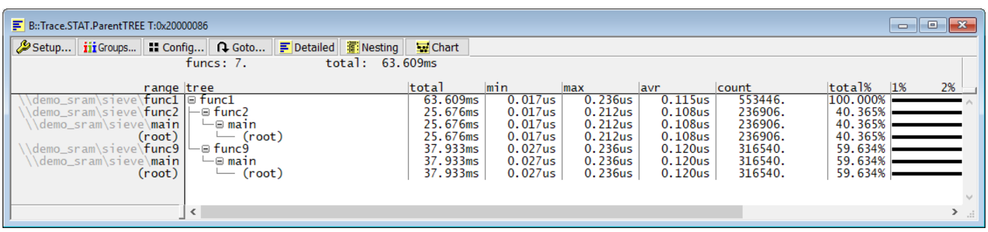


The first section jumps to the first, last or maximum entry in the **Trace.List** window.

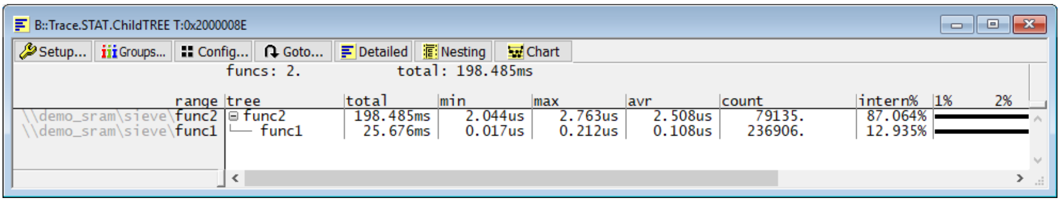
Linkage shows an analysis of all places that this function was called from along with runtime information. This example shows all the places in the application where `func1()` is called from and for each it displays the runtime measurements. It is a convenient method to access **Trace.STATistic.LINKage**.



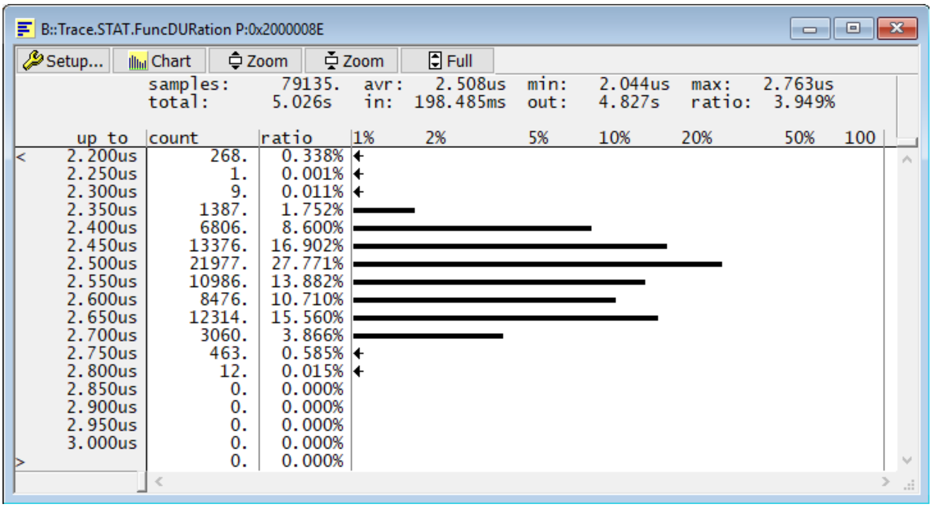
Parents shows the call tree back to the entry point or root of the application, again with performance information for each function in the tree. This is a convenient way to access **Trace.STATistic.ParentTREE**.



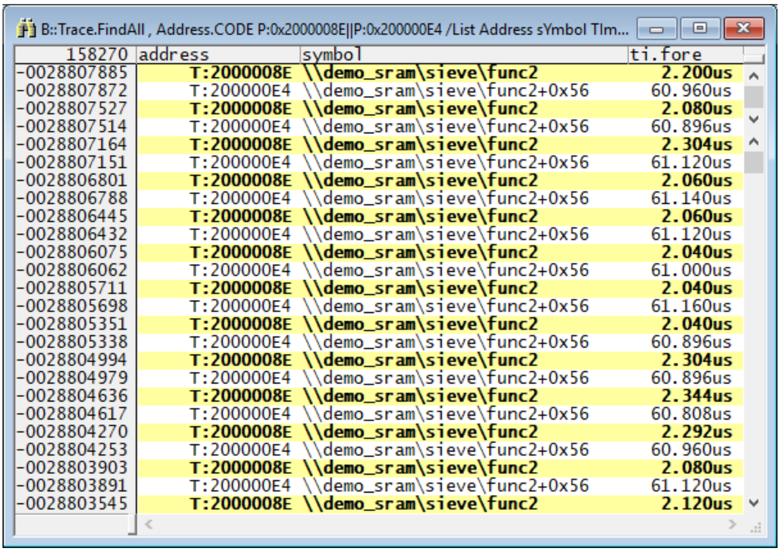
Children shows the call tree starting at the selected function and traversing downwards through all of the sub-functions with performance information for each node. This menu item provides an easy way to access **Trace.STATistic.ChildTREE**.



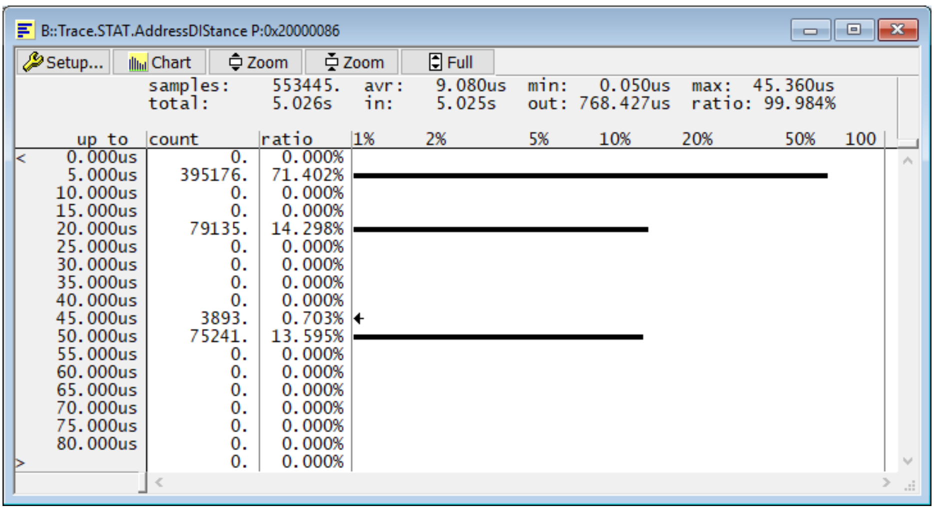
Duration shows a histogram of runtimes for the selected function. TRACE32 allocates 16 appropriate bucket sizes ad assigns the runtime values to each of these. These can be over-ridden by the user on the command line by using **Trace.STATistic.FuncDURation**. The zoom buttons and scroll bar can be used to navigate or display more or less details for a specific range.



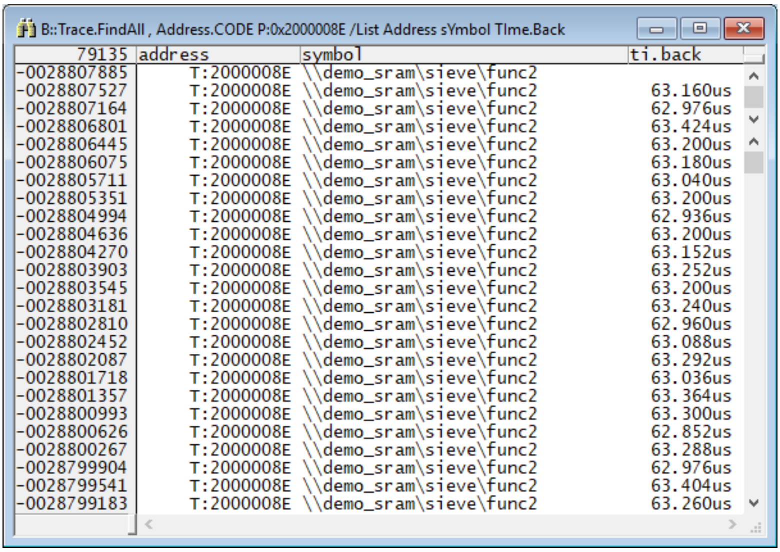
Findall Duration provides a convenient way of automatically searching for all entry and exit points for the selected function. The yellow lines are function entries and the white lines are the corresponding exit. The value in the ti.fore column shows the time for each event, so the top line shows that func2 took 2.20 µs to execute and it was another 60.960µs before it was called again.



Distance Analysis shows the amount of time that elapsed between one call to a function and the next call to that function. This is a convenient way of accessing the **Trace.STATistic.AddressDIStance** command.



Findall Distance is an easy way to search for all entry points of the selected function. The ti.fore column shows the time between calls to that function.



B::Trace.FindAll , Address.CODE P:0x2000008E /List Address sYmbol TIme.Back

79135	address	symbol	ti.back
-0028807885	T:2000008E	\\demo_sram\sieve\func2	
-0028807527	T:2000008E	\\demo_sram\sieve\func2	63.160us
-0028807164	T:2000008E	\\demo_sram\sieve\func2	62.976us
-0028806801	T:2000008E	\\demo_sram\sieve\func2	63.424us
-0028806445	T:2000008E	\\demo_sram\sieve\func2	63.200us
-0028806075	T:2000008E	\\demo_sram\sieve\func2	63.180us
-0028805711	T:2000008E	\\demo_sram\sieve\func2	63.040us
-0028805351	T:2000008E	\\demo_sram\sieve\func2	63.200us
-0028804994	T:2000008E	\\demo_sram\sieve\func2	62.936us
-0028804636	T:2000008E	\\demo_sram\sieve\func2	63.200us
-0028804270	T:2000008E	\\demo_sram\sieve\func2	63.152us
-0028803903	T:2000008E	\\demo_sram\sieve\func2	63.252us
-0028803545	T:2000008E	\\demo_sram\sieve\func2	63.200us
-0028803181	T:2000008E	\\demo_sram\sieve\func2	63.240us
-0028802810	T:2000008E	\\demo_sram\sieve\func2	62.960us
-0028802452	T:2000008E	\\demo_sram\sieve\func2	63.088us
-0028802087	T:2000008E	\\demo_sram\sieve\func2	63.292us
-0028801718	T:2000008E	\\demo_sram\sieve\func2	63.036us
-0028801357	T:2000008E	\\demo_sram\sieve\func2	63.364us
-0028800993	T:2000008E	\\demo_sram\sieve\func2	63.300us
-0028800626	T:2000008E	\\demo_sram\sieve\func2	62.852us
-0028800267	T:2000008E	\\demo_sram\sieve\func2	63.288us
-0028799904	T:2000008E	\\demo_sram\sieve\func2	62.976us
-0028799541	T:2000008E	\\demo_sram\sieve\func2	63.404us
-0028799183	T:2000008E	\\demo_sram\sieve\func2	63.260us

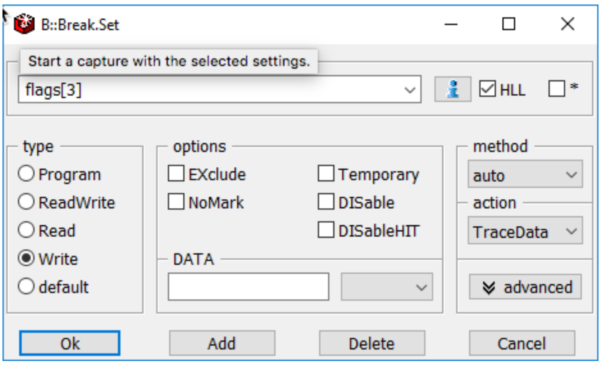
Distribution

Distribution shows the values that have been assigned to a variable during the sampling period. This requires data trace, which on Cortex-M requires the ITM/DWT to be configured correctly. Data trace packets consume more trace bandwidth than program flow trace, so care should be taken when monitoring data items not to cause overflows in the on chip FIFO of the target.

To monitor a variable, a breakpoint can be used to program the DWT to emit a data trace packet on reads, writes or any access. To monitor writes to HLL variable `flags[3]`, set a breakpoint like this:

```
Break.Set Var.ADDRESS(flags[3]) /Write /TraceData
```

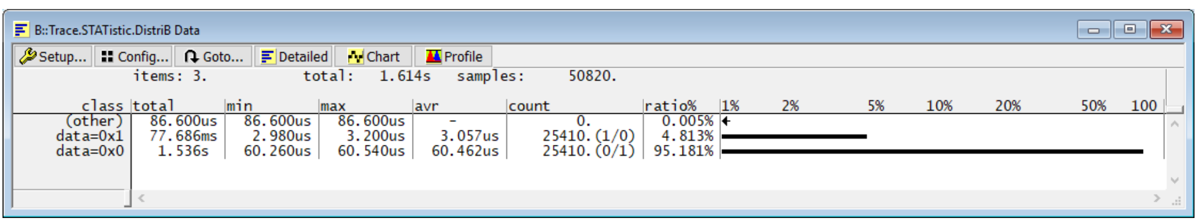
The **Var.ADDRESS()** macro returns the address of the 4th element of array flags. It can be used on any complex variable where a simple symbol table lookup will not find the address. The same breakpoint would be set in the UI like this.



The ITM needs to be set for Data Trace: **ITM.DataTrace ON** or via the GUI. Start the target to collect trace samples.

If the ETM trace is set to OFF (no ETM trace data generated) then the menu items under the **Perf** menu can be used to analyze the data. If ETM is on (Program flow trace data also generated) then the analysis cannot be performed by using the items under the **Perf** menu. When ETM is detected, the **Perf** will default to using ETM trace for analysis. Instead, use the ITMTrace series of commands. Both commands will be shown in the examples below. The generic Trace command can also be used.

Trace.STATisitc.DistriB Data

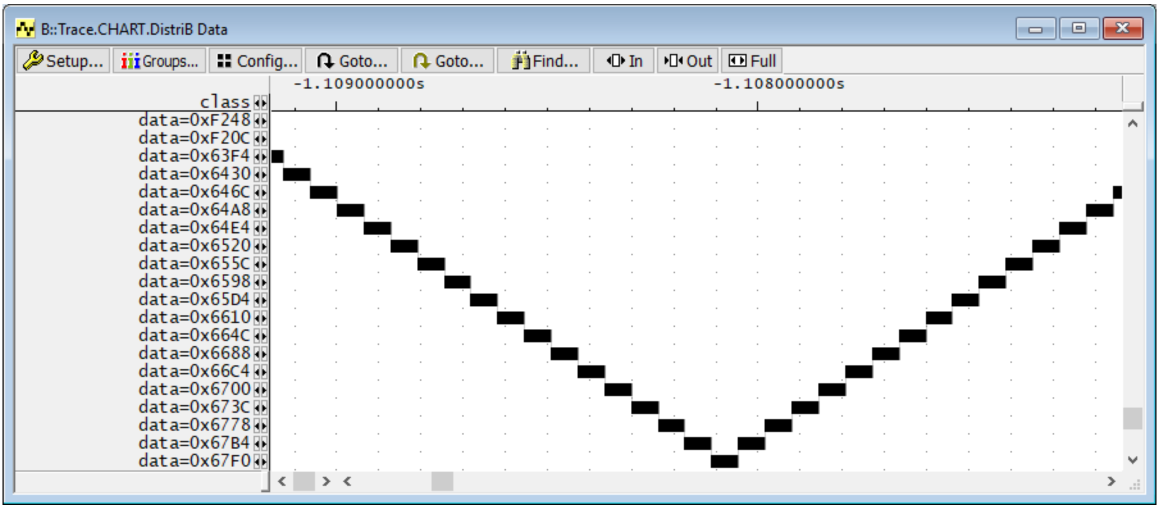


The image above shows that for 95.181% of the sampling time, the value of `flags[3]` was 0 and for 4.813% it was a 1. The (other) entry is the time at the start of the trace capture where the value of `flags[3]` was unknown as no writes had yet been captured.

Clicking the Config button will open a dialog to allow the user to change the sorting order and to add or remove different columns.

Clicking the **Chart** button will show how the values of the variable changed over time in a graphical format.

Trace.Chart.DistriB Data



This feature uses the **Trace.STATistic.DURation** command to measure the time between two arbitrary points; it is no longer constrained to function entry and the corresponding exit. For instruction timing **Trace.STATistic.AddressDURation** is better.

To show how long a variable contains one value before it switches to another use something like:

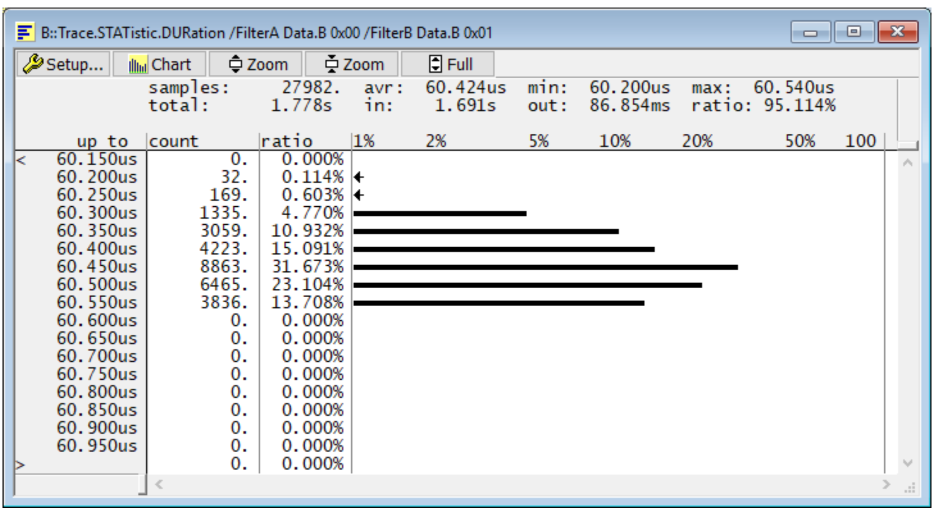
> **Trace.STATistic.DURation /FilterA Data.**<*width*> <*value*> **/FilterB Data.**<*width*> <*value*>

Where <*width*> can be:

- B - byte (8bits), can be split into B0, B1, B2 or B3 to represent a single byte in a 32bit access
- W - Word (16bits)
- L - long (32bits)
- Q - Quad (64bits)
- T - Triple (128 bits)

To measure the time between a variable changing from a 0 to a 1:

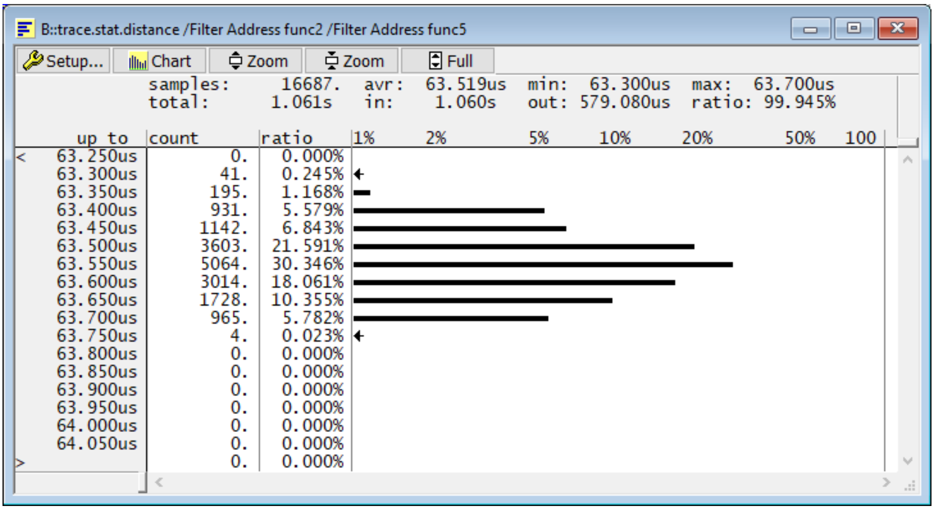
1. Set a breakpoint on the variable to generate trace data, for example:
   Break.Set Var.ADDRESS(flags[3]) /Write /TRACEDATA
2. Set **ITM.ON**
3. Set **ITM.DataTrace ON**
4. Collect trace data
5. Use **Trace.STATistic.DURation /FilterA Data.B 0x00 /FilterB Data.B 0x01** to show the results



This allows the user to time between two arbitrary points in the application code. This is done by using the **/Filter** options to the **<trace>.STATistic.DIStance** command.

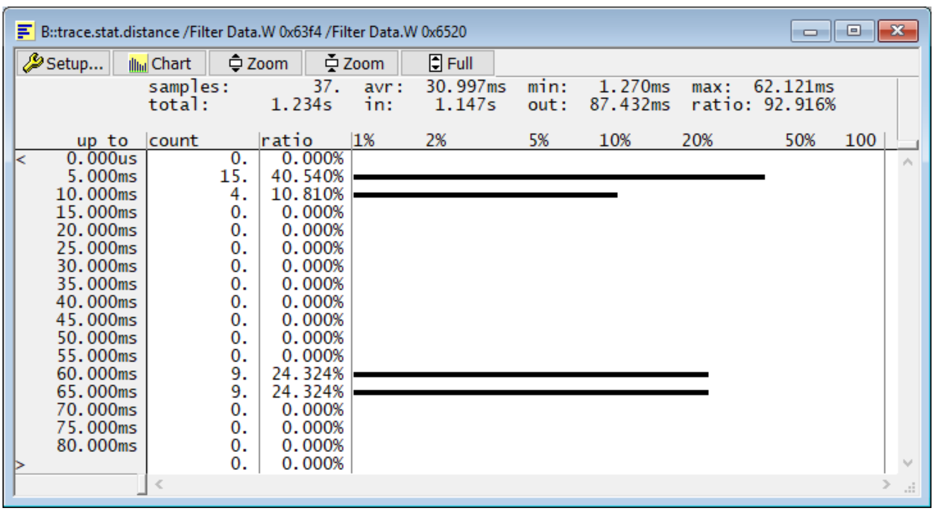
For example, if it is required to measure the distance between a call to `func2()` and the subsequent call to `func5()`:

Trace.STATistic.DIStance /Filter Address func2 /Filter Address func5



Filtering can also be performed using data values. Use the Data.<x> filter options instead of Address. For example, to filter on a variable containing the value 0x63f4 and the next time it contains the value 0x6520 use the command:

Trace.STATistic.DIStance /Filter Data.W 0x63F4 /Filter Data.W 0x6520



Trace and Groups

Trace data can be assigned to logical groups to aid analysis and filtering. Groups can be created with the **GROUP.Create** command and viewed with **GROUP.List**. For an overview of the **GROUP** functionality, refer to the **GROUP** command group.

Groups can be created around:

- Address Ranges
- Functions
- Modules
- Symbols
- Tasks

Grouping only affects the display of captured trace data, not the data itself.

When a group is created, TRACE32 automatically creates a base group called “other”. This contains everything that is not a part of any defined group.

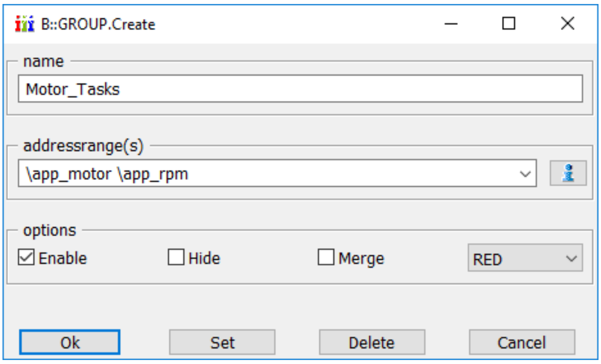
Grouping code into logical function blocks makes it easier to reduce the amount of trace data that users need to analyze and allows them to focus on the regions or interactions of interest.

Grouping by Modules

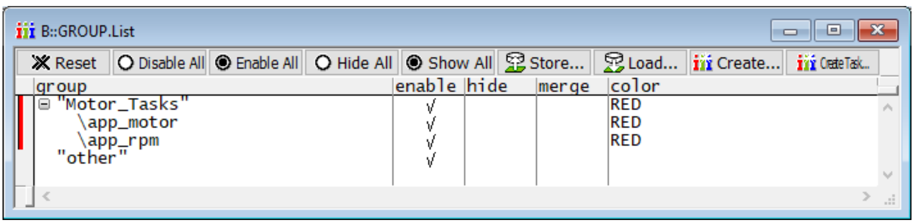
More than one item can be part of a group. In this example two object files will be allocated to a single group. Both sets of code are involved with the motor control functions of the target so it makes sense to group them logically like this.

GROUP.CreateTASK “Motor_Tasks” “\app_motor” “\app_rpm” /RED

GROUP.Create without any arguments will open the creation dialog. The image below shows it filled in to create the same group as the command above. Clicking the blue ‘i’ button will open a module browser window from which the user can select multiple modules.

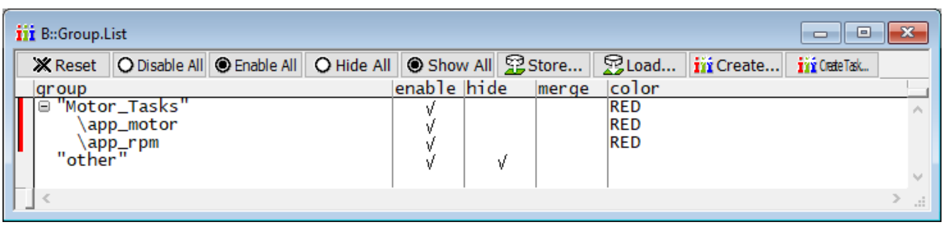


GROUP.List now shows the new group and the default “other” group.

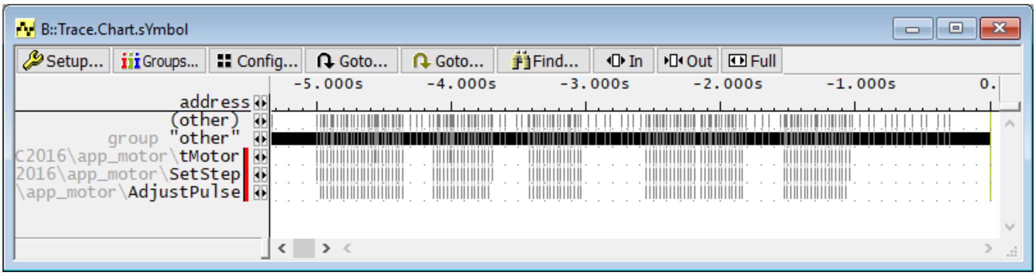


In the **GROUP.List** window, check the hide column for group “other”. Alternatively, use the command:

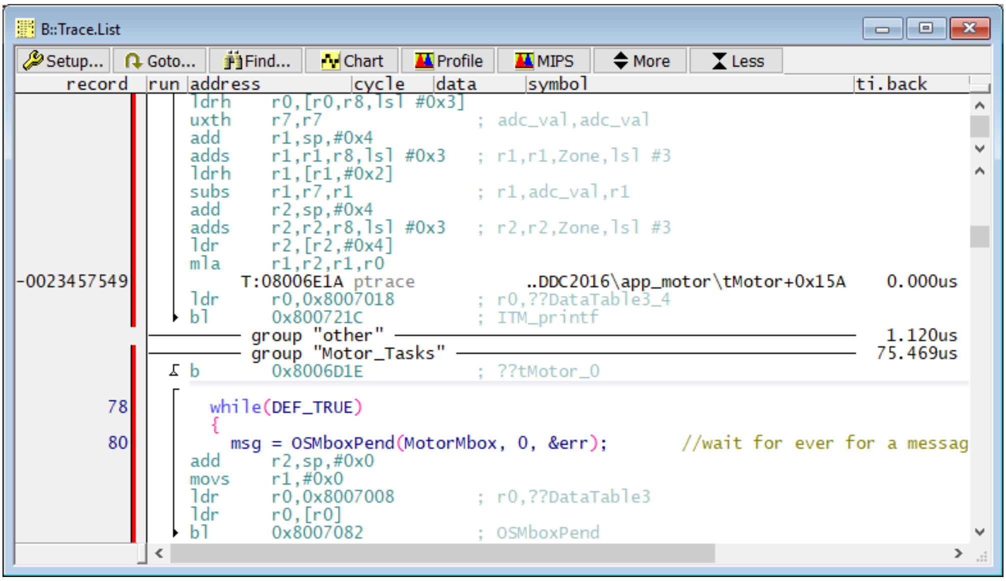
GROUP.HIDE “other”



Now the trace view windows will suppress the display of anything that is marked as hidden.

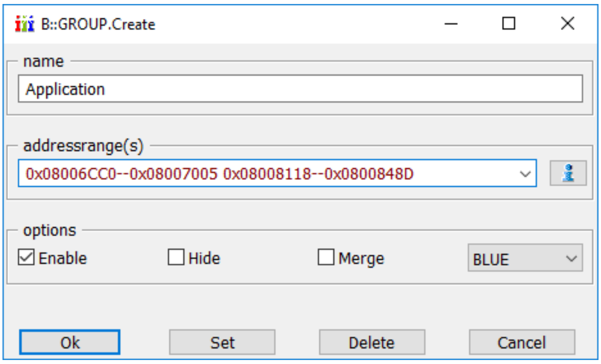


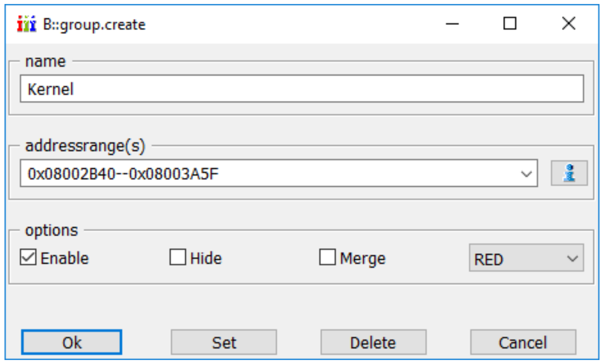
With the “other” group hidden, the **Trace.List** window looks like this. Different groups are color coded and the colors can be seen in the bar on the left side of the window. As control passes between groups a line is added to the trace listing to show this. In the example below 75.459us were taken up processing group “other” and before that 1.120us were spent processing in group “Motor_Tasks”.



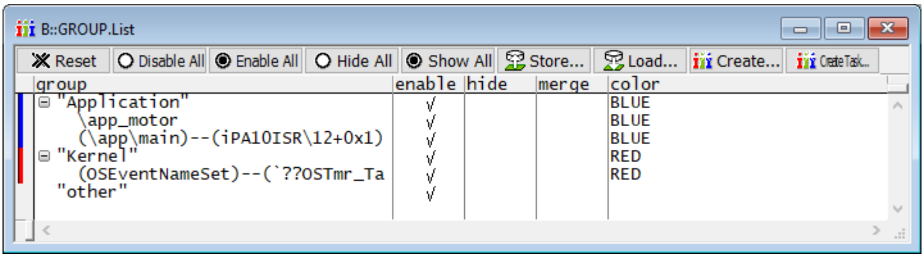
Grouping by Address Range

This example creates two groups: one for the application code and another for the kernel code. Multiple entries can be added to the addressrange(s) field, as shown in the “Application” example below. Whereas, the “kernel” entry only covers a single range.

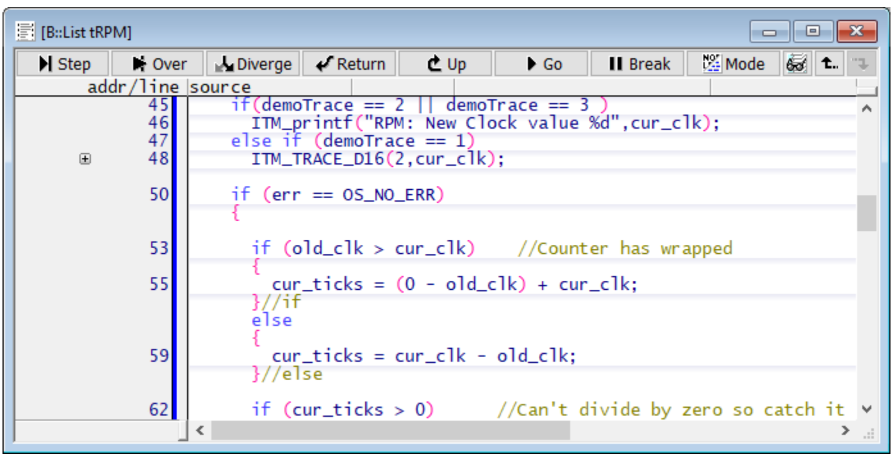




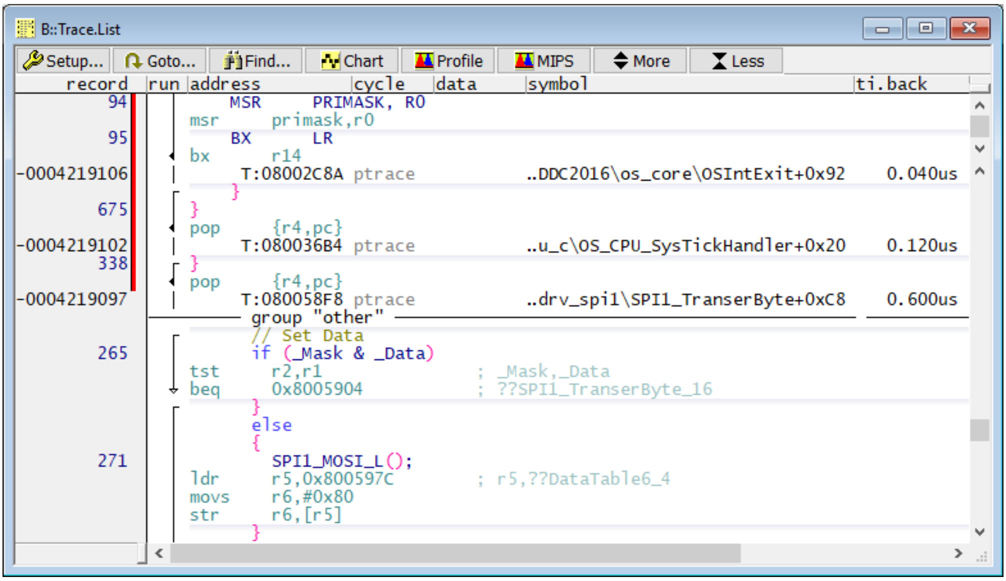
The groups can be listed using the **GROUP.List** command:



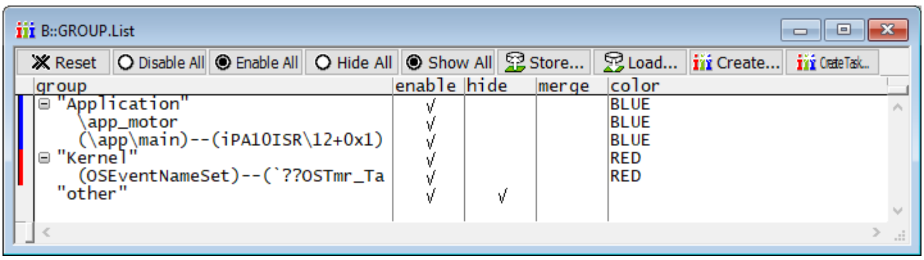
The “other” group is clearly visible here. The source view (**List**) windows now have an added color bar to show which group the code being viewed belongs to.



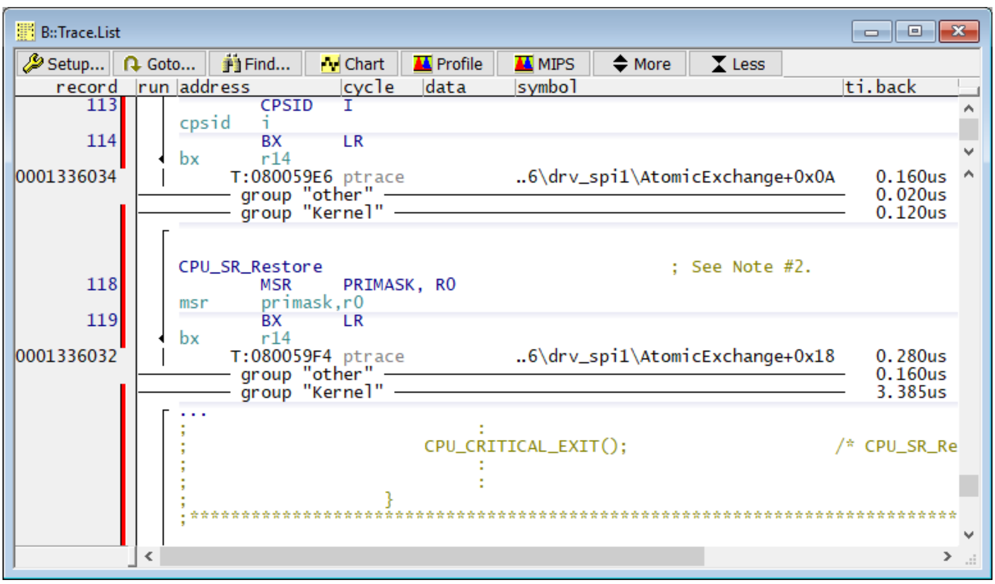
A trace listing window will show also show the groups color coded and will indicate where control passes from one group to another.



One or more groups can be removed from the trace display windows by clicking the "hide" column in the **GROUP.List** window or by using the command **GROUP.HIDE** *<name>*.



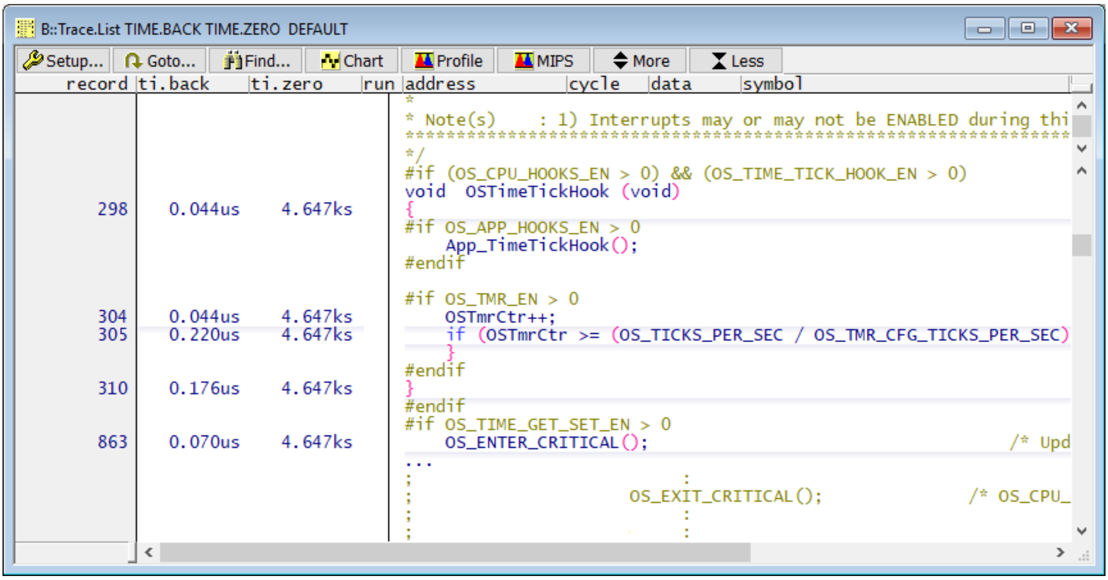
The "other" group is now suppressed from display.



Timing

A very detailed set of analyses can be performed using the **PERF** commands (**“Analyzing the Results”**, page 30) but measuring time from a significant event or between two points can be performed using the timing columns in the **Trace.List** window.

The default **Trace.List** window includes the **TIme.Back** column. This is the time taken from the last trace entry to this one. Additional timing columns can be added to any trace listing window, for example:

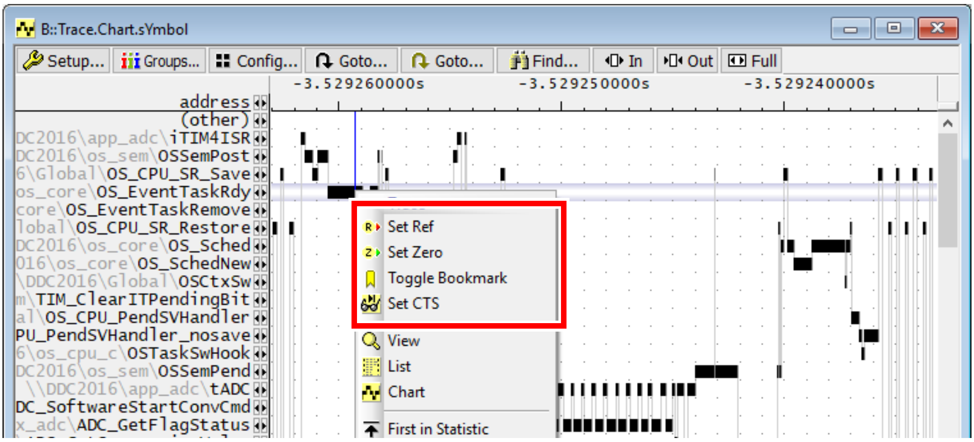


The most commonly used timing options are listed in the table below. Other, less common, options are described in the documentation for **Trace.List**.

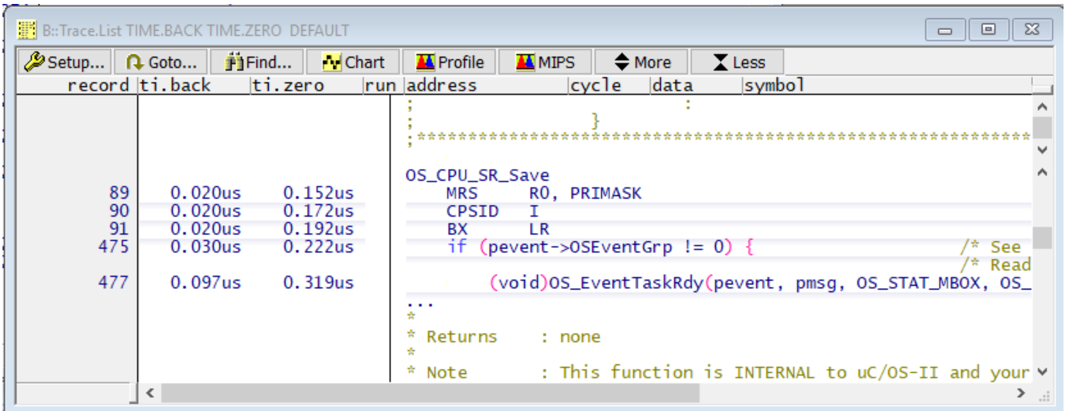
TIme.Back	The time elapsed since the last entry.
TIme.Fore	The time elapsed from this entry until the next entry.
TIme.REF	The time elapsed since a user-defined reference point.
TIme.Zero	The time elapsed since the debug session started or a user-defined Zero point.
TIme.Trigger	The time elapsed since the trace trigger event.

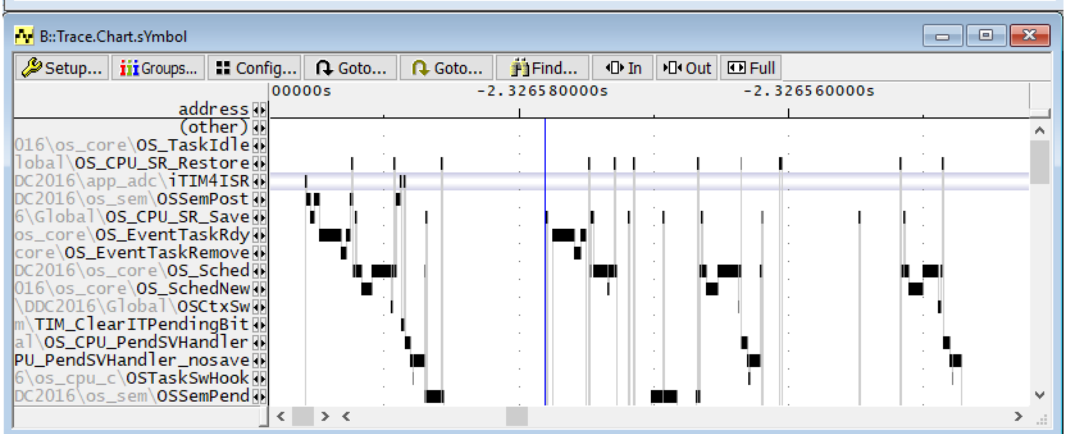
Also, bear in mind that some times can be negative as the item may have occurred before the timing point.

Some marker points can be user-defined or moved by the user. To do this, right-click in any windows that shows a view of the captured trace data and select the appropriate marker from the pop-up menu.



The two images below show how setting the **TI.Zero** marker in the **Trace.Chart** window will affect the timing in the **Trace.List** window.



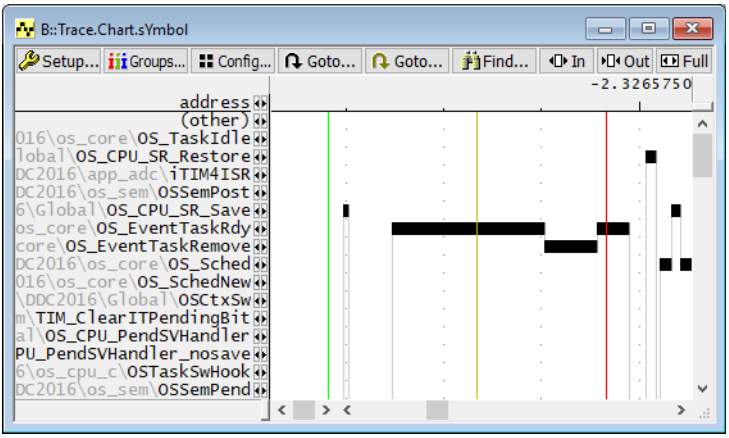


After setting the Zero point in the Chart window, the values in the TI.Zero column in the List window change to show the timings relative to the new Zero point.

Different markers are shown in different colors.

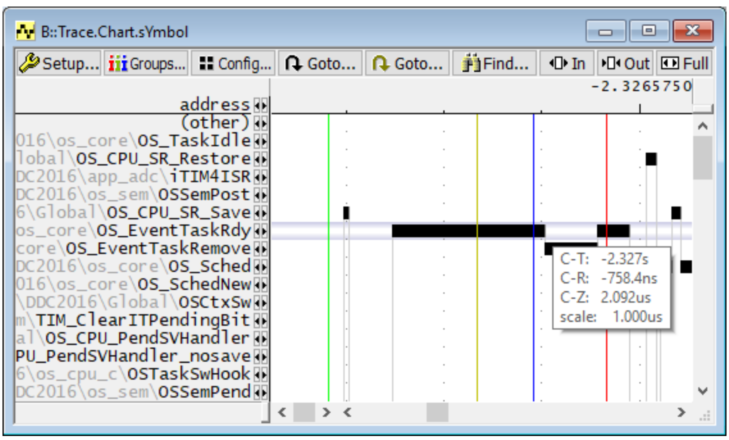
Green	Zero marker
Red	User-defined REF marker
Yellow	User-defined bookmarks.

All three can be seen in the picture below.



A blue line represents a user placed cursor and is used for tracking between trace view windows.

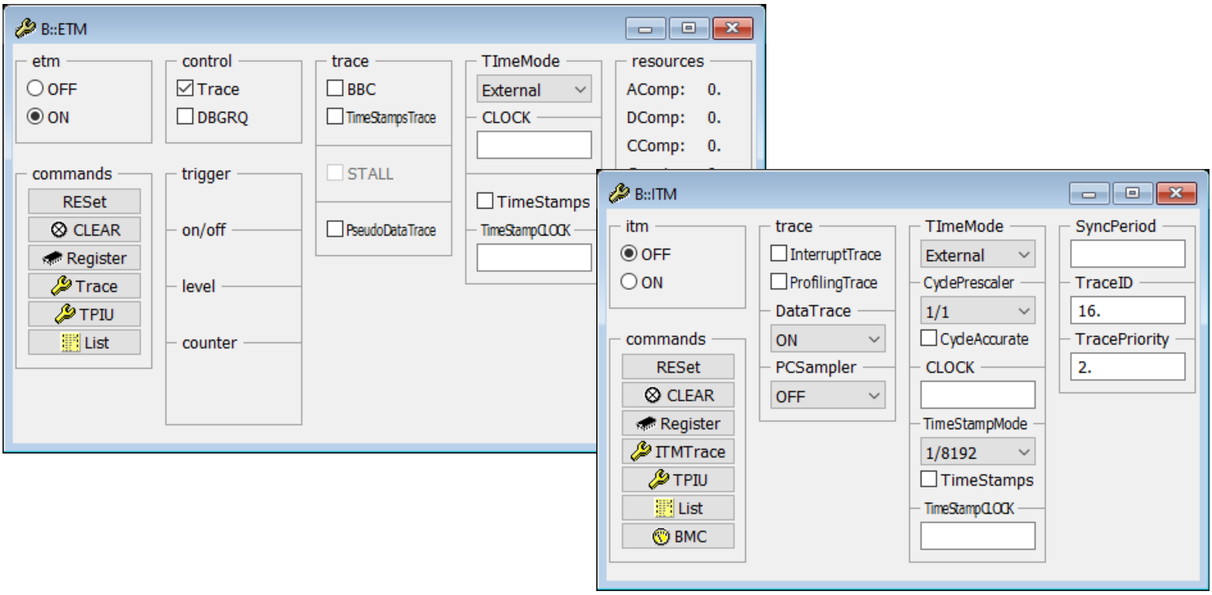
Left-clicking anywhere in the **Trace.Chart** window will create a pop-up which shows the timing information:



C-T	Current time (selected by cursor) to Trigger point.
C-R	Current time (selected by cursor) to REF marker.
C-Z	Current time (selected by cursor) to Zero point.

Trace Based Code Coverage

The manual **"Application Note for Trace-Based Code Coverage"** (app_code_coverage.pdf) gives a detailed introduction to the trace-based code coverage. However, the manual does not contain details about the architecture-specific setups. Here is an overview of the setups for the ETM.

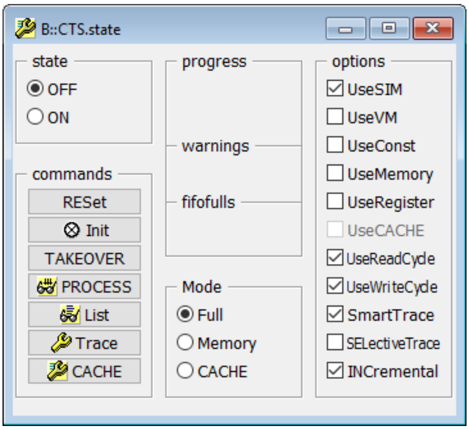


The following settings are recommended:

```
ETM.Trace ON                    ; Enable program flow trace
ETM.TImeMode External           ; Enable tool timestamp
ITM.OFF                         ; Switch the ITM off
```

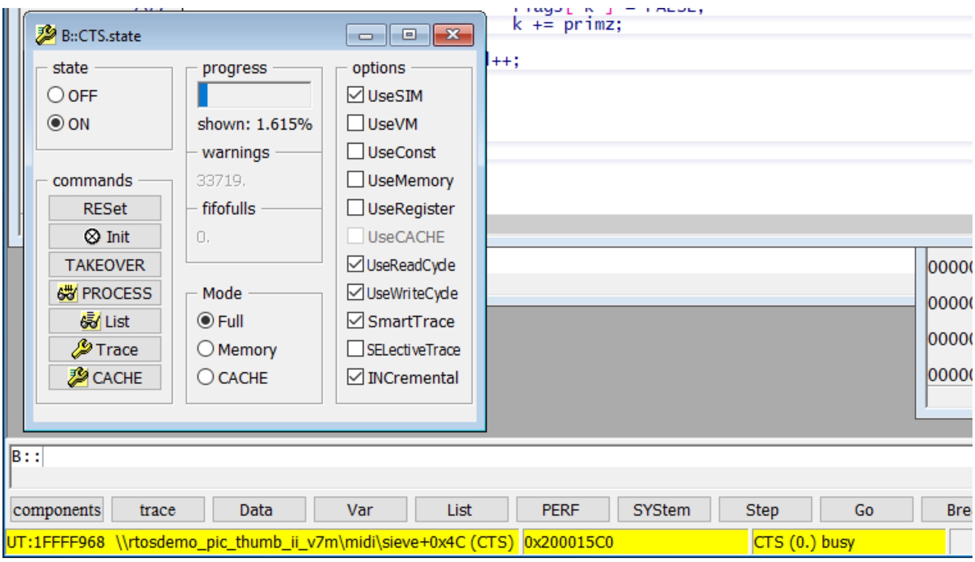
Trace Based Debugging

Once a set of trace data has been captured it is possible to walk through the history of the recording, often filling in missing details. The Context Tracking System (CTS) is used to perform these functions. CTS is accessed from the Trace menu or via the command **CTS.state**.

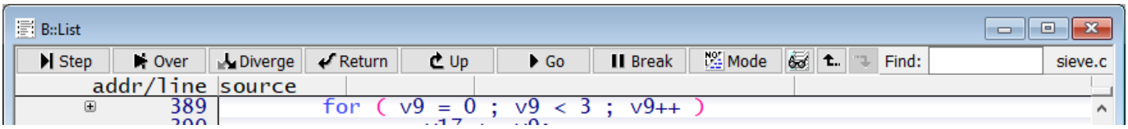


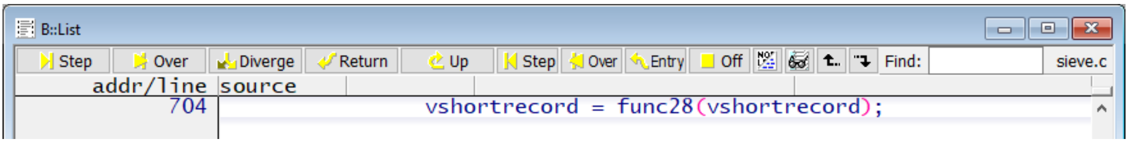
More information about trace based debugging can be found here: **"Trace-based Debugging"** in General Commands Reference Guide C, page 163 (general_ref_c.pdf).

Once the mode is set to ON (via the GUI or the command **CTS.ON**), the CTS system analyses the recorded trace in more detail, often filling in missing data accesses and areas where FIFO overflows occurred. This may take a few minutes depending upon the size of the captured trace and the speed of the host PC and available RAM. It will look something like this:



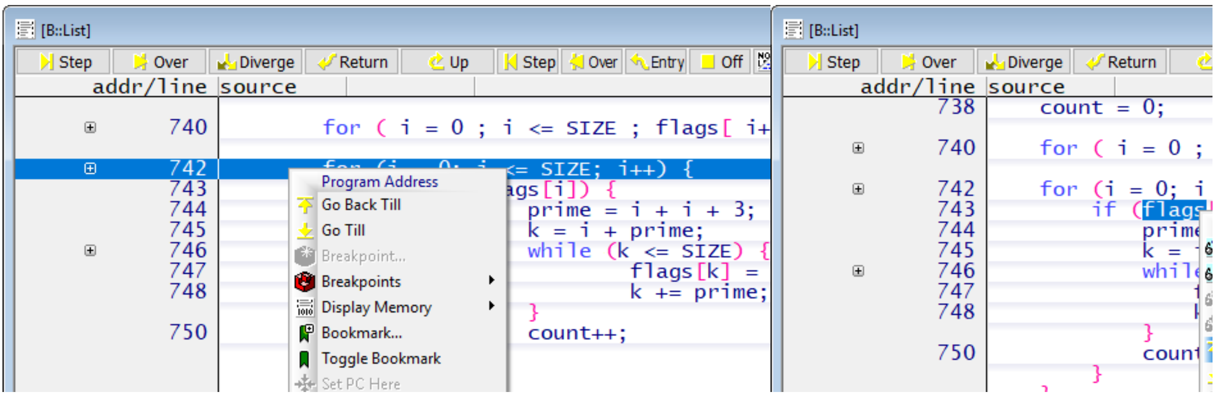
When CTS is active, the TRACE32 status bar and control icons in the List window change color to yellow. New buttons appear in the List window tool bar, allowing the user to step forwards or backwards through the code and to run back to a function entry point. Compare the two images below.



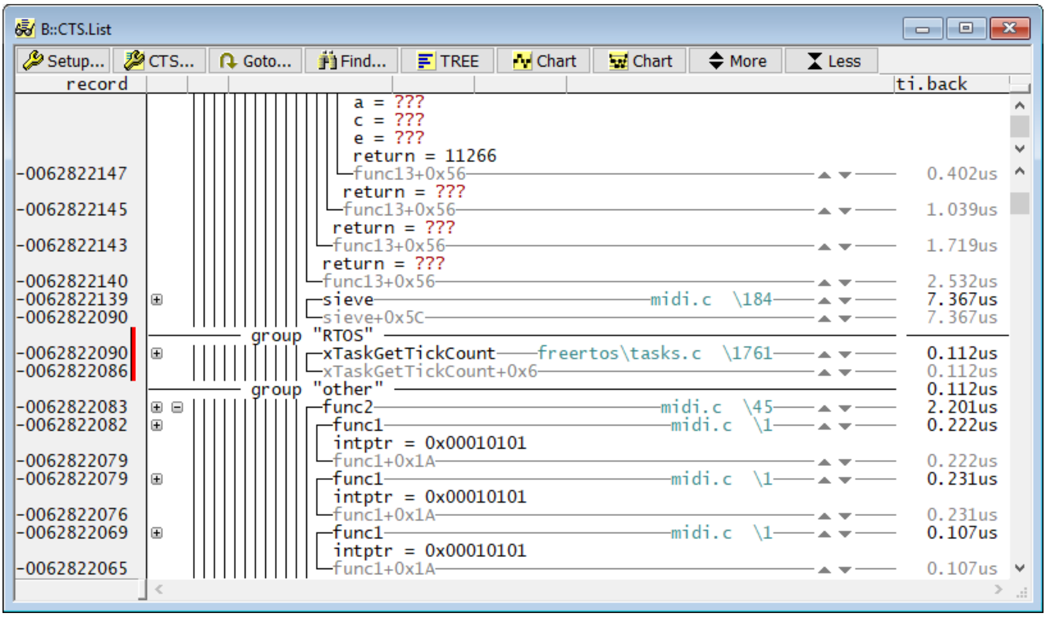


Stepping backwards or forwards through the code shown in the List window will cause any register, memory or variable display windows to be updated to show the re-constructed values as they would have been at that point in history.

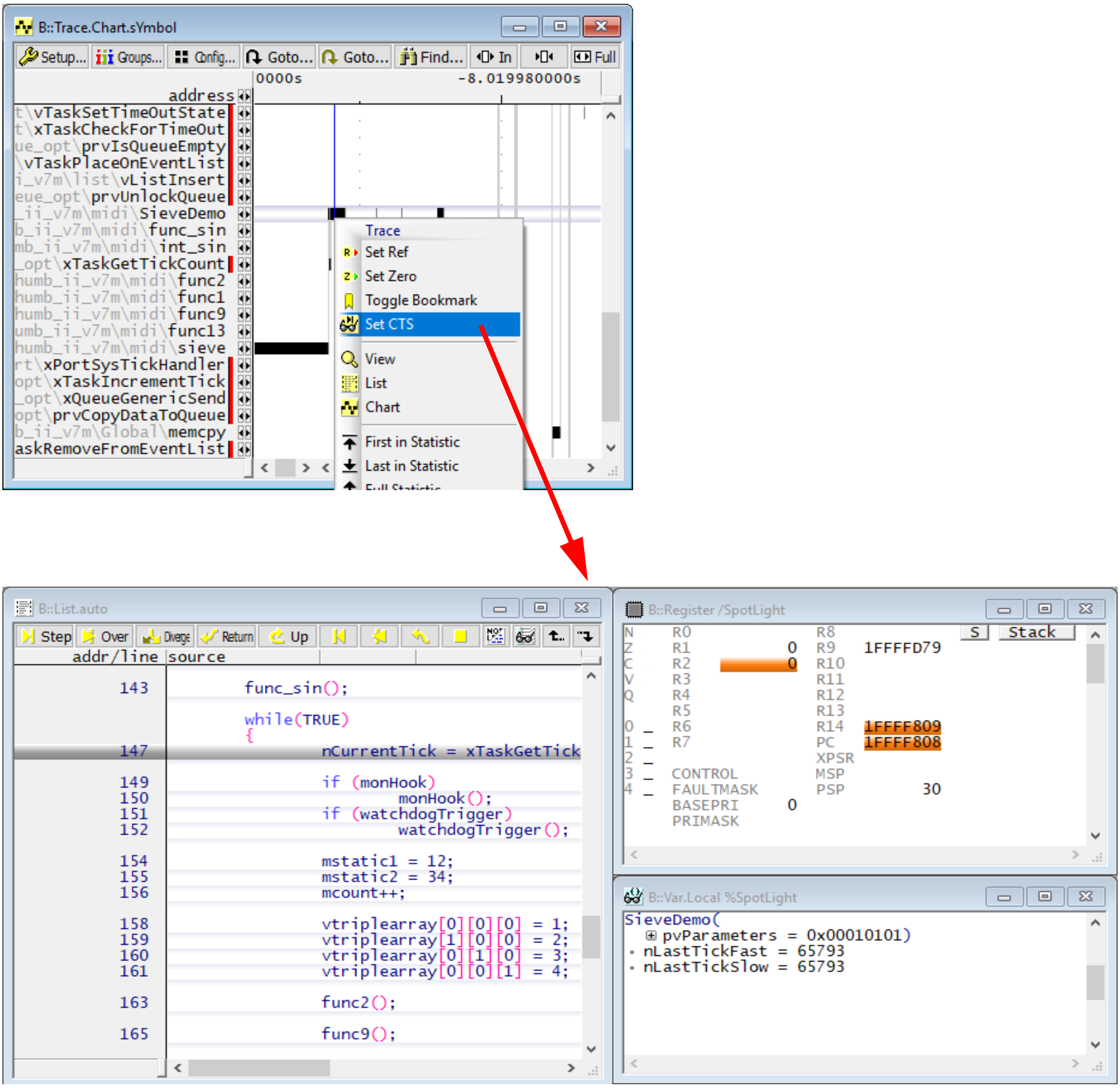
The right-click menu also has new options added to allow users to go forwards or backwards to a point or run forwards or backwards to an access on a variable.



Clicking the List button in the **CTS.state** window opens the **CTS.List** window. This shows a nested view of the code. Where possible, arguments and return values have been reconstructed using the current state of the target and the captured trace data. Timing for each node is represented in the ti.back column. Each node can be opened or closed to show more detail. Where a piece of data cannot be reconstructed a series of ??? will appear in all windows that show CTS data.



In any trace display window, the right-click menu has a Set CTS option. Selecting this will cause the current state to jump to that point in history with as much of the context reconstructed as possible. This allows the use of high level tools such as chart windows to locate an issue and then zoom in to see in detail what occurred at that time.



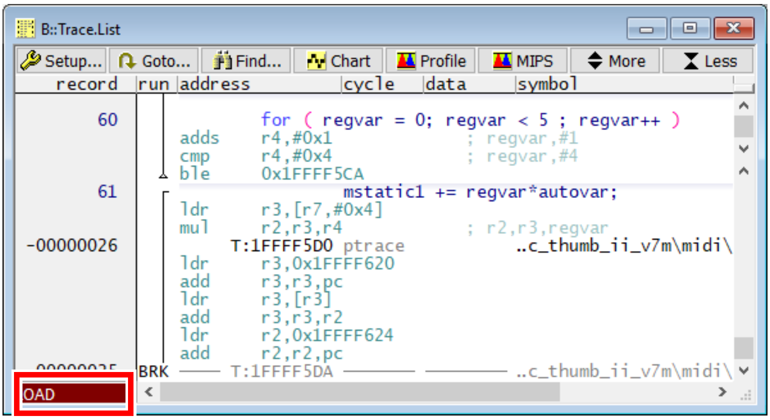
Once CTS is active, all trace analysis features will work with the reconstructed data. This is often better than the captured data as any gaps may be filled in but will be no worse.

Off-line Analysis

A set of trace data can be saved for later, off-line analysis. To save the trace data, use **Trace.SAVE** *<file>*. In most cases only the /ZIP option is useful. Saved trace data files have the extension ".ad".

Trace files can be re-load with **Trace.Load** *<file>*.

Normal trace analysis features work exactly as if they were using a recently captured "live" trace. The only difference is that the trace display windows now contain a red "LOAD" mnemonic. These can be seen in the image below.



A more detailed analysis may often be achieved if the contents of the RAM are saved along with the CPU registers at the point where the trace had finished being captured. This can be done like this:

```
Data.SAVE.Binary <file> <address_range>
STOre <file> Register
```

TRACE32 software can also be installed or configured as an instruction set simulator. In this way the saved trace can be analyzed off-line.

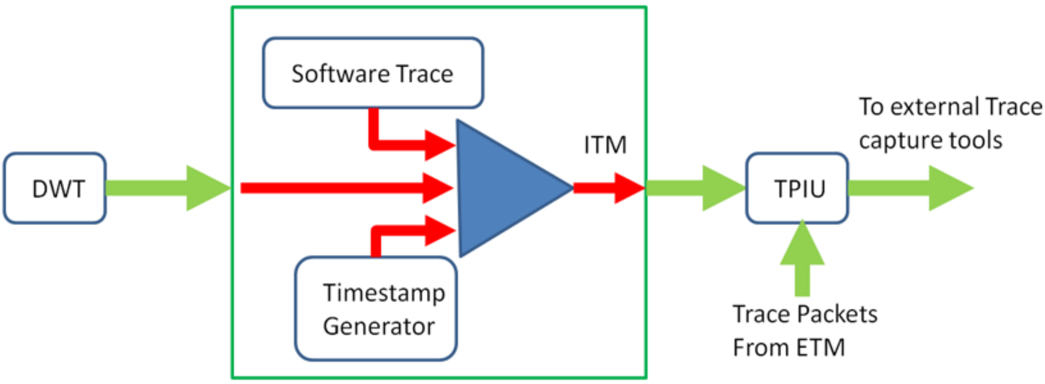
To analyze saved trace data:

1. Start TRACE32 in simulator mode, set the CPU type and system mode to UP.
   - **SYStem.CPU**
   - SYStem.Up
2. Load the saved RAM contents.
   - **Data.LOAD.Binary** *<file> <address>*
3. Load the saved CPU registers
   - **DO** *<save_reg_file>*
4. Load application symbols
   - **Data.LOAD.Elf** *<symbol_file>* **/NoCODE**
5. Load the saved trace file

- **Trace.Load** *<file>*.ad

- The DWT provides some useful additions to the basic program flow trace, although not all will be covered here as we will focus only on those features that provide a trace like output.
- PC Sampler
- Data Trace capability
- Interrupt Trace
- ETM Trigger

The block diagram below shows how the DWT is organized to inject trace packets into the ITM stream for eventual spooling to the TPIU and then to the trace port (ETM or SWV). However, if the TPIU is configured for SWV then it will not transmit ETM packets; only the ITM packets will be transmitted.

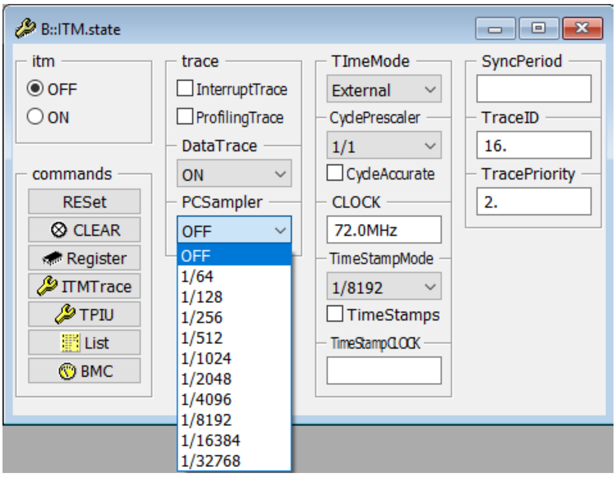


Each trace source is given a unique 7-bit Trace ID (ATID) so that the streams can be separated by the debug tools.

For details on time stamping please refer to **"Time Stamping"**, page 72.

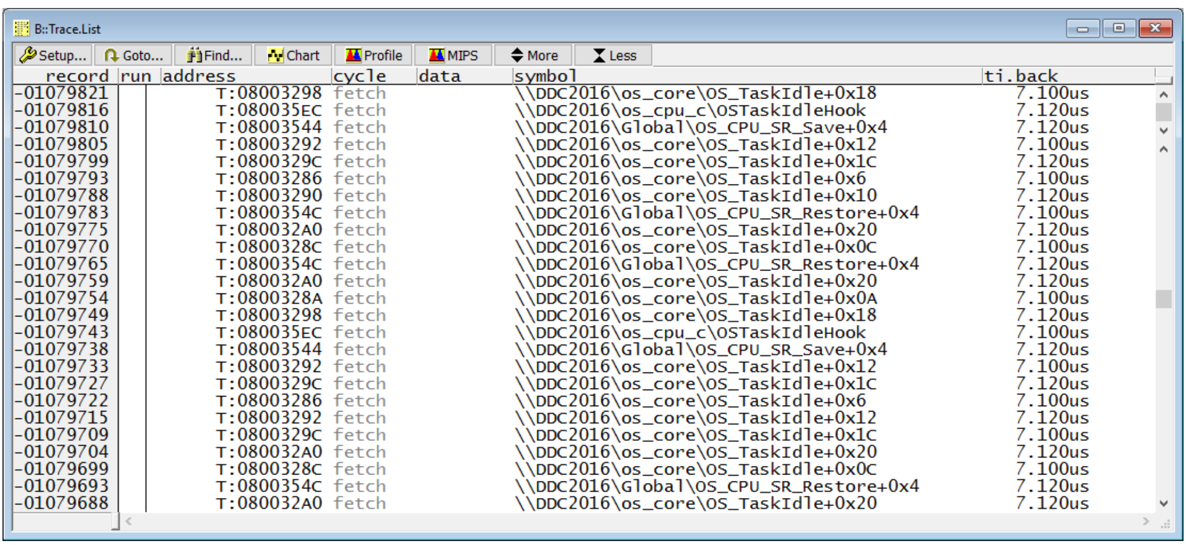
This feature is not a true program flow trace in the way that MTB or ETM will allow full capture of program flow. The PC Sampler will periodically sample the Program Counter and emit the address as a trace packet to the ITM. The sampled PC is emitted via the ITM which makes this technique ideal for systems which only support the ITM or Serial Wire Viewer for trace as it can provide some data about program flow. The program flow may not be 100% accurate; items that can complete in less than the selected number of cycles to sample may not show in the data at all. The PC can be sampled every 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 or 32768 clock cycles.

This feature is configured in the **ITM.state** window. There is no separate DWT configuration window as the trace events from the DWT require the ITM block to also be present on the device or they cannot be emitted. Since the two are so tightly linked, the **ITM.state** window allows the user to configure both the ITM and DWT blocks of the Cortex-M device.

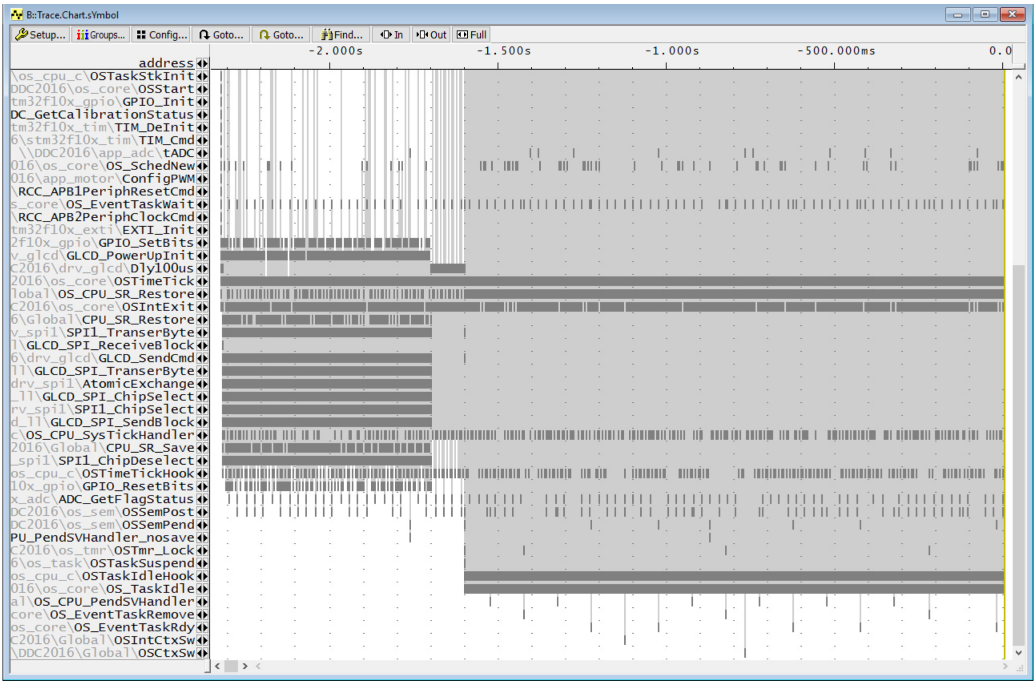


Set the frequency of the PCSampler (via the GUI or the **ITM.PCSampler** command) and enter the value of the CPU clock to enable timing of the data.

Run the target to generate and sample the data then use the command **Trace.List** to show the results.

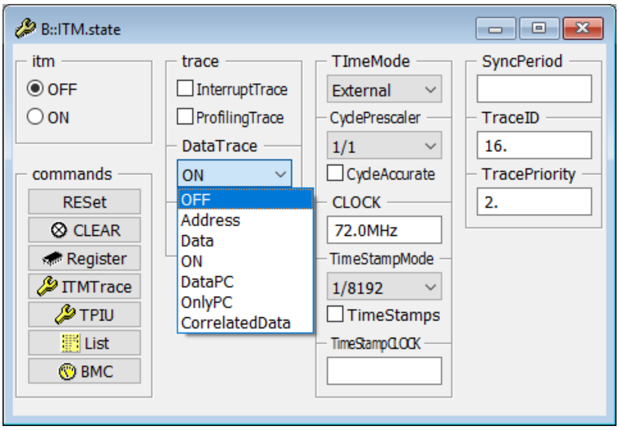


Clicking the **Chart** button or using the command **Trace.Chart.sYmbol** will show the functions against time and look like this.



One of the features of the DWT is to be able to inject data trace events into the ITM for inclusion into the trace stream. The data events are configured using the TRACE32 breakpoint interface. More information can be found here: **"On-chip Breakpoints"**, page 84.

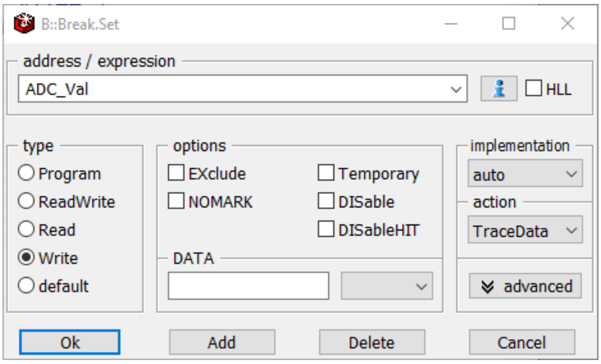
Set the **ITM.DataTrace** mode.



Several modes are supported:

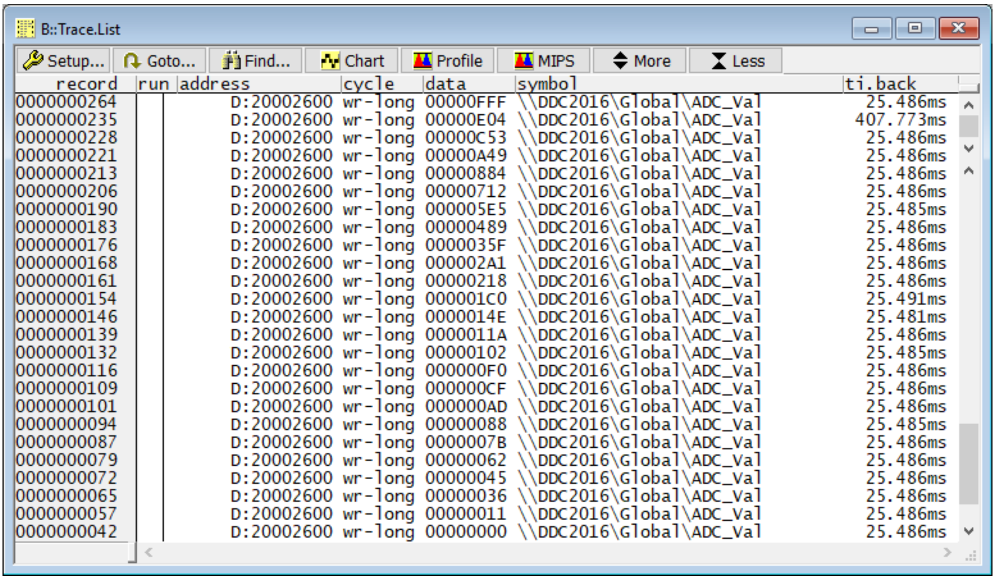
OFF	No Data Trace information will be generated.
Address	If the data address matches a DWT comparator, address information about the data access will be emitted as a trace packet.
Data	If the data address matches a DWT comparator, the data value will be emitted as a trace packet.
ON	If the data address matches a DWT comparator, address and data information about the data access will be emitted as a trace packet.
DataPC	If the data address matches a DWT comparator, address and data information about the data access will be emitted along with the address of the instruction that issued the data access.
OnlyPC	If the data address matches a DWT comparator, address of the instruction that performed the data access will be emitted.
CorrelatedData	Produces the same information as DataPC but the streams (ETM & ITM) are merged in TRACE32.

To generate data trace, set a TraceData breakpoint on the value to be traced.



Use: **Break.Set ADC_Val /Write /TraceData**

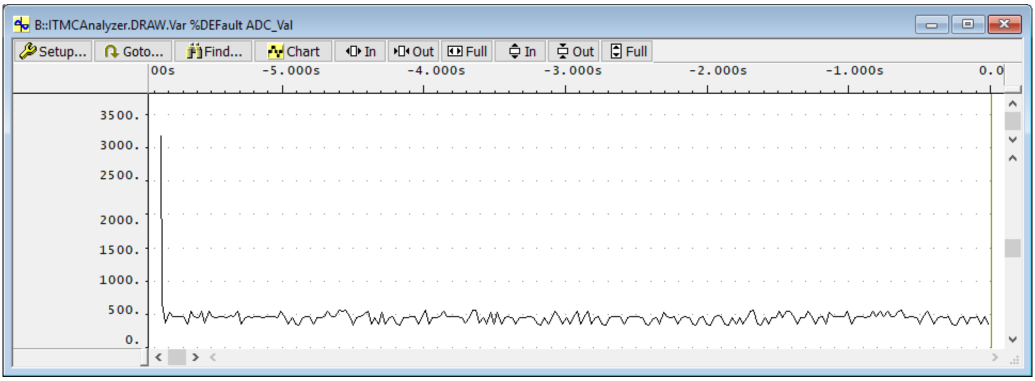
Run target to collect data. Show results:



The traced data values can be shown as a graph as the change against time. Use the command

Trace.DRAW.Var %DEFault ADC_Val

to show this:



To see where in the code a data access came from:

1. Switch **ITM.DataTrace DataPC**

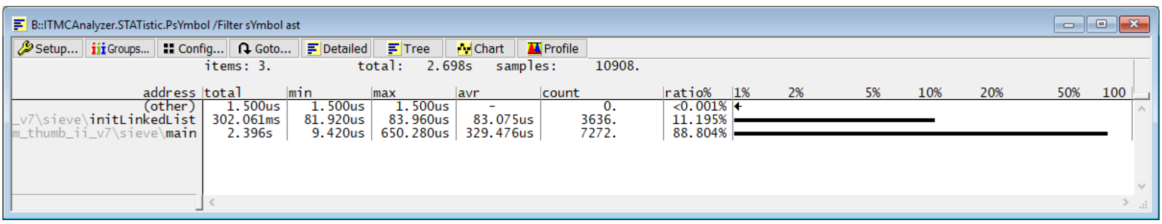
2. Set the breakpoint to ReadWrite

   - **Break.Set** *<var>* **/ReadWrite /TraceData**

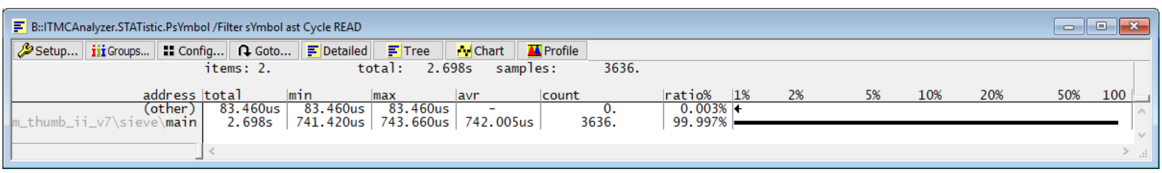
3. Run the target to collect the trace samples.

The results can be shown filtered by any kind of access: read, write or any. The command **Trace.STATistic.PsYmbol** is used.

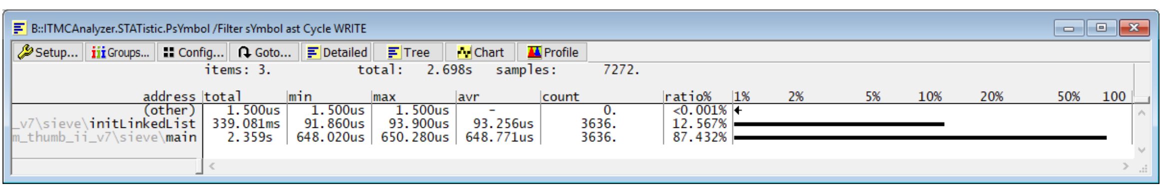
To show all accesses: **Trace.STATistic.PsYmbol /Filter sYmbol** *<var>*



To show only read accesses: **Trace.STATistic.PsYmbol /Filter** *<var>* **Cycle READ**



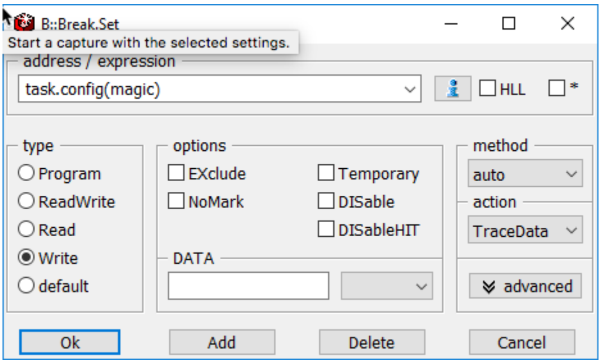
To show only write accesses: **Trace.STATistic.PsYmbol /Filter** *<var>* **Cycle WRITE**



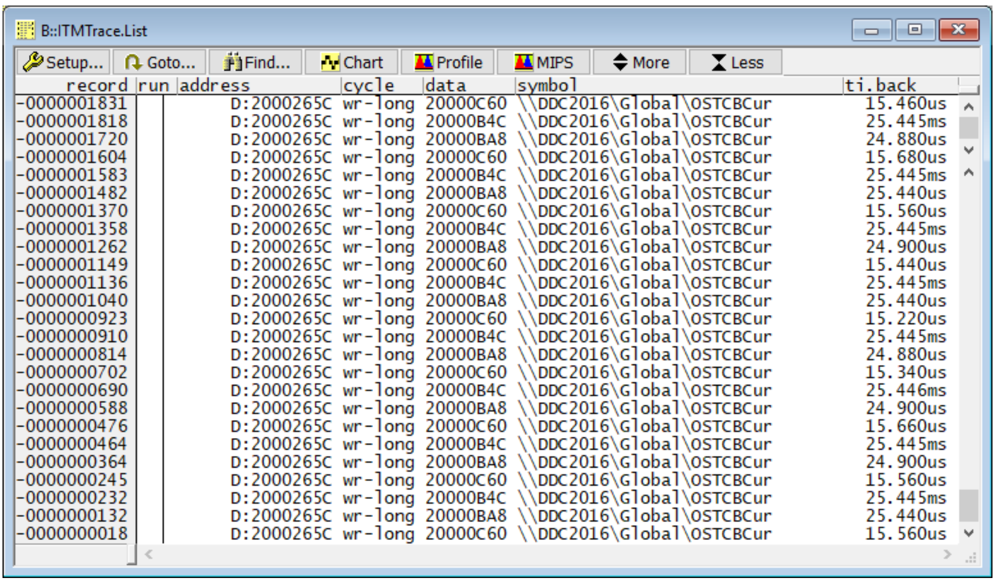
Task/Thread switch Tracing

If your design uses an RTOS, task switches can be traced. If the RTOS is supported by a TRACE32 OS Awareness you can use the address **TASK.CONFIG(magic)**. If your RTOS is not supported by TRACE32 you can use the address of the variable that holds the currently executing thread ID.

Configure **ITM.DataTrace Data** and set a breakpoint (**Break.Set TASK.CONFIG(magic) /Write /TraceData**) and run the target to collect trace samples.

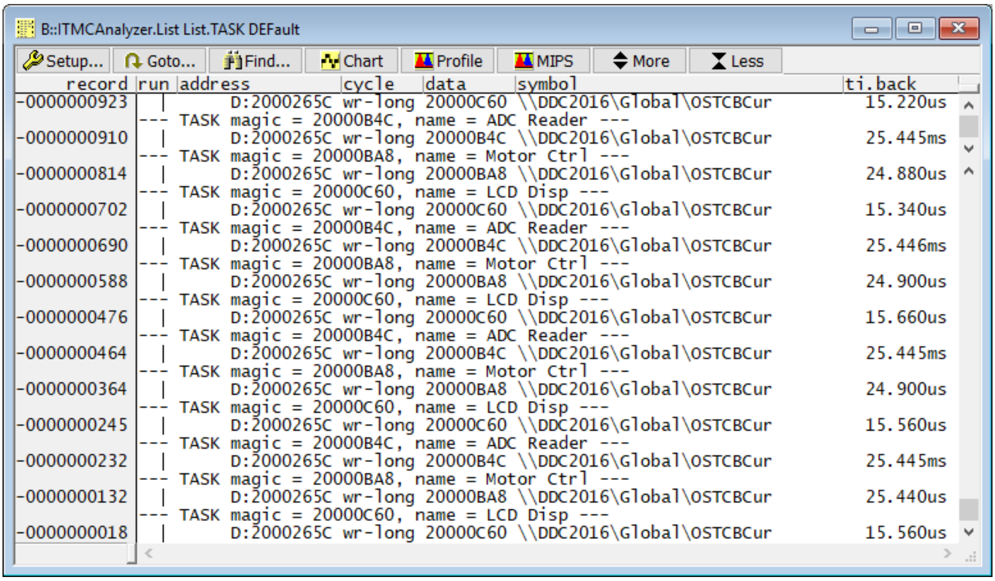


A list of all writes (all task switches) can be seen with **ITMTrace.List**.

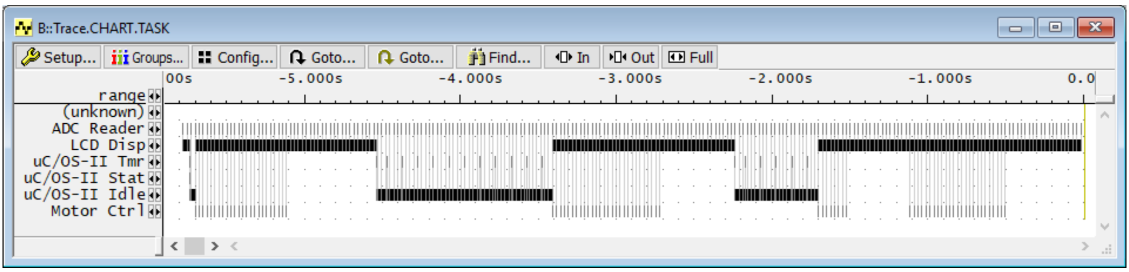


The ti.back column shows how long each task or thread was running before it was switched out for a new one.

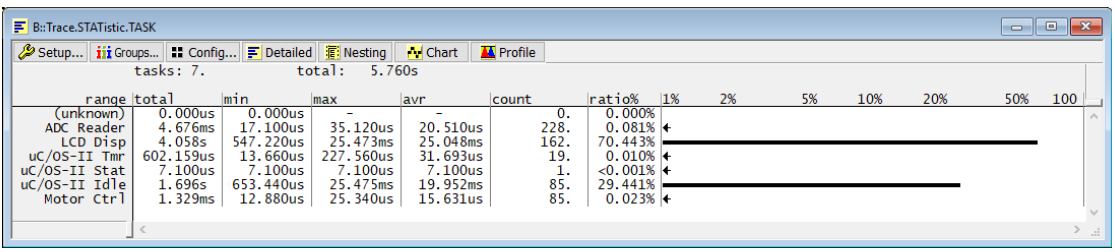
A more detailed view can be made using: **ITMTrace.List List.TASK DEFault**



To view how tasks or threads changed over time, use: **Trace.Chart.TASK**



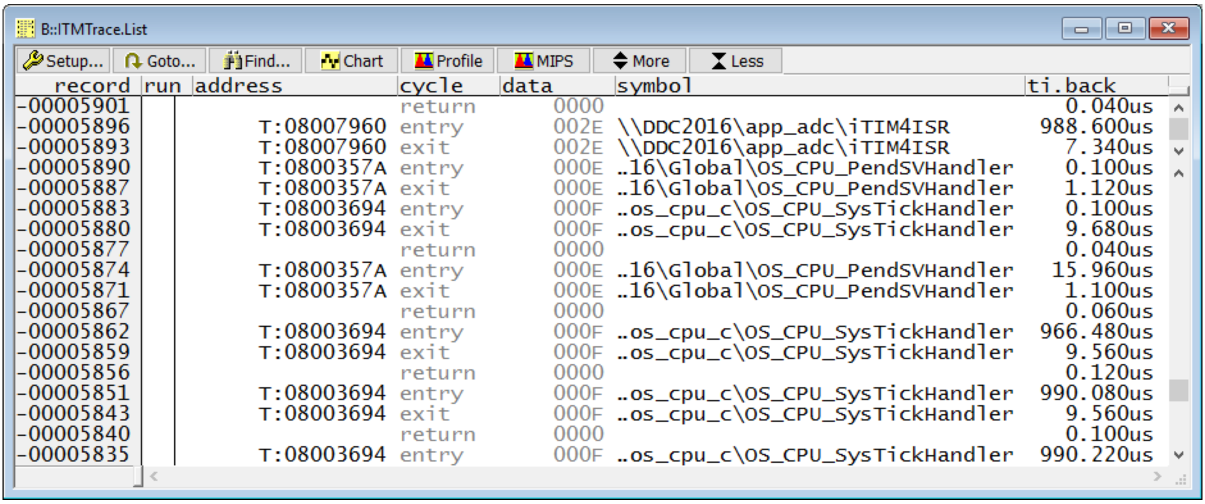
To view run time information for each task, use **Trace.STATistic.TASK**



The DWT can be programmed to generate trace events for each interrupt entry and exit point.

Select InterruptTrace from the ITM configuration window or use the command **ITM.InterruptTrace ON**.

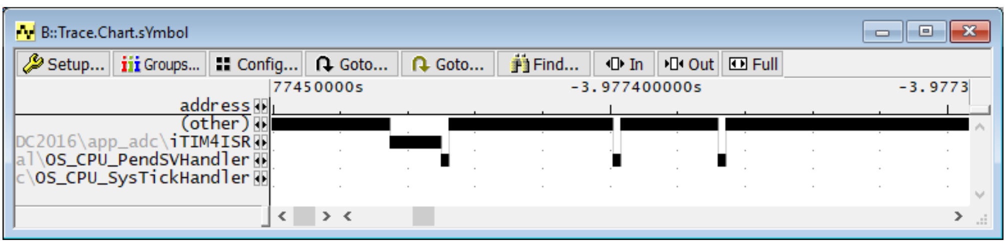
To reduce the likelihood of internal trace FIFO overflows, switch off all other trace sources (ProfilingTrace, DataTrace and PCSampler). Run the target to sample the trace data. The results can be displayed using the **Trace.List** command and look like this.


B::ITMTrace.List

Setup... Goto... Find... Chart Profile MIPS More Less

record	run	address	cycle	data	symbol	ti.back
-00005901			return	0000		0.040us
-00005896		T:08007960	entry	002E	\\DDC2016\app_adc\iTIM4ISR	988.600us
-00005893		T:08007960	exit	002E	\\DDC2016\app_adc\iTIM4ISR	7.340us
-00005890		T:0800357A	entry	000E	..16\Global\OS_CPU_PendSVHandler	0.100us
-00005887		T:0800357A	exit	000E	..16\Global\OS_CPU_PendSVHandler	1.120us
-00005883		T:08003694	entry	000F	..os_cpu_c\OS_CPU_SysTickHandler	0.100us
-00005880		T:08003694	exit	000F	..os_cpu_c\OS_CPU_SysTickHandler	9.680us
-00005877			return	0000		0.040us
-00005874		T:0800357A	entry	000E	..16\Global\OS_CPU_PendSVHandler	15.960us
-00005871		T:0800357A	exit	000E	..16\Global\OS_CPU_PendSVHandler	1.100us
-00005867			return	0000		0.060us
-00005862		T:08003694	entry	000F	..os_cpu_c\OS_CPU_SysTickHandler	966.480us
-00005859		T:08003694	exit	000F	..os_cpu_c\OS_CPU_SysTickHandler	9.560us
-00005856			return	0000		0.120us
-00005851		T:08003694	entry	000F	..os_cpu_c\OS_CPU_SysTickHandler	990.080us
-00005843		T:08003694	exit	000F	..os_cpu_c\OS_CPU_SysTickHandler	9.560us
-00005840			return	0000		0.100us
-00005835		T:08003694	entry	000F	..os_cpu_c\OS_CPU_SysTickHandler	990.220us

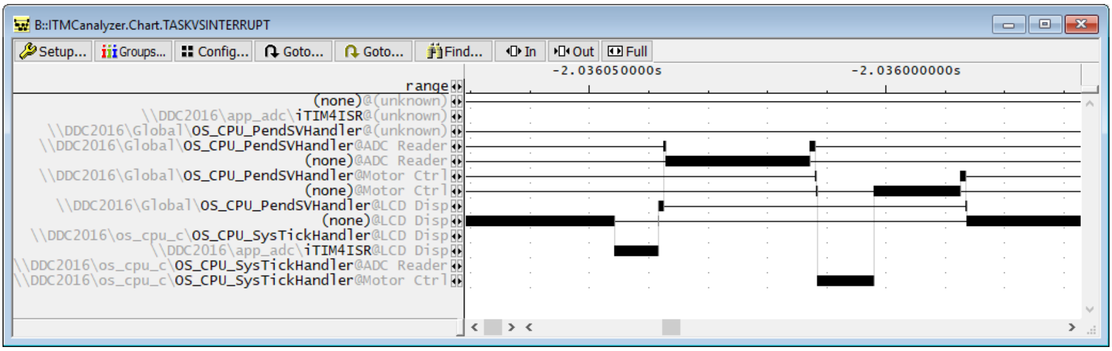
The ti.back column shows the time from the previous sample to this one, so iTIM4ISR was running for 7.340 us. – this is the difference between the entry marker and the exit marker. Clicking the Chart button (or using the command **Trace.Chart.sYmbol**) will show a graphical representation of interrupt nesting against time and may look like this. The “other” row is any code executing on the target that is not an interrupt service routine.



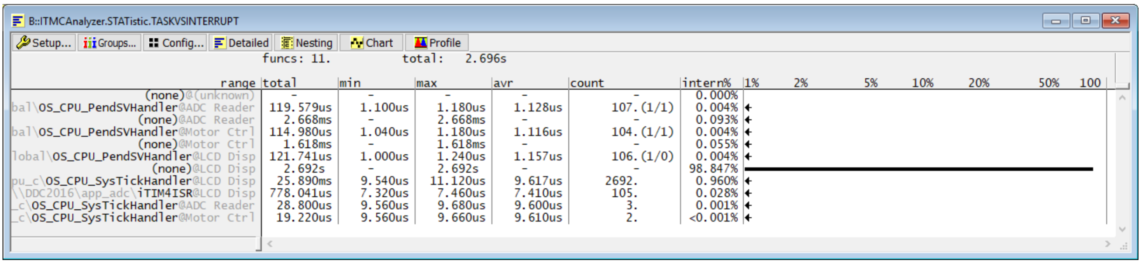
Adding a breakpoint to generate a data trace packet on task switches will allow us to analyze which tasks or threads were interrupted. Set:

```
ITM.InterruptTrace ON
ITM.DataTrace Data
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

The results can be displayed with: **<trace>.Chart.TASKVSINTERRUPT**



Or with **Trace.STATistic.TASKVSINTERRUPT**



ETM Trigger

The DWT contains an ETM Trigger capability. This is hidden from the user and is accessed using the /TraceTrigger breakpoint type. More information can be found in:

“Trace Filtering”, page 24

“Trace Control by Filter and Trigger” in Training Arm CoreSight ETM Tracing, page 91 (training_arm_etm.pdf)

The ITM organizes trace from three main areas:

1. Software Generated Trace
2. Integrates packets from DWT into the trace stream
3. Generates timestamp packets for insertion into the trace stream

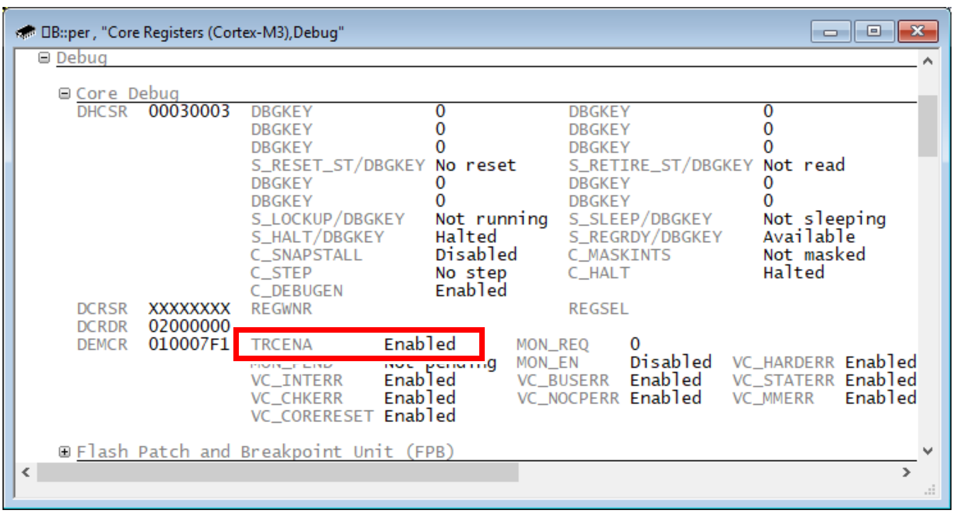
If packets arrive simultaneously from more than one of these sources, the ITM is responsible for arbitration and prioritizes them in the order represented by the list above.

The ITM is an optional component and may not be included in all designs (a read to a non-existent ITM register will return 0x00000000). It also requires a TPIU to output any data.

Here is a list of important ITM registers. All registers are fully accessible in Privileged mode and all registers can be read in user mode. If the corresponding bits in **ITM_TPR** are set then user mode also has write access to **ITM_STIM[31..0]** and **ITM_TER**.

0xE0000FF0 - 0xE0000FFC	Component Identification Registers (CID[3..0])
0xE0000FD0 - 0xE0000FEC	Peripheral Identification Registers (PID[7..0])
0xE0000FB0	Lock Access Register (ITM_LAR)
0xE0000E80	Trace Control Register (ITM_TCR)
0xE0000E40	Trace Privilege Register (ITM_TPR)
0xE0000E00	Trace Enable Register (ITM_TER)
0xE0000000 - 0xE000007C	Stimulus Port Registers (ITM_STIM[31..0])

Before the ITM can be used it must be enabled and unlocked. TRACE32 will enable the ITM automatically by setting the **TRCENA** bit in the Debug Exception and Monitor Control Register.



Unlocking is done by writing `0xC5ACCE55` to **ITM_LAR**.

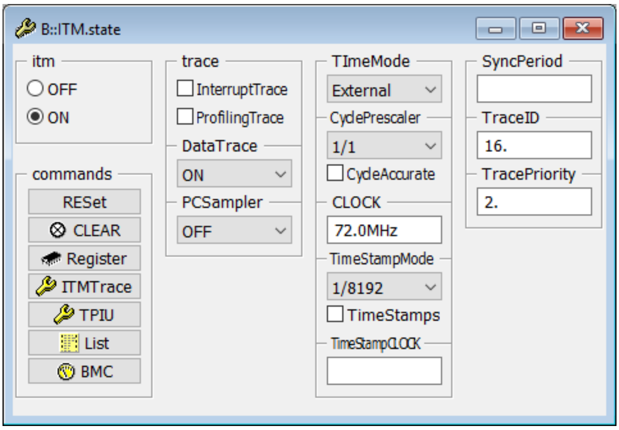
Writing a value to any of the ITM stimulus ports causes a data packet to be generated and fed to the TPIU for inclusion into the trace stream. First, the ITM must be unlocked. An example is shown below.

The Cortex-M CMSIS includes functions for writing data to the ITM but macros are more efficient and reduce the overhead of using the ITM at runtime.

```
static volatile unsigned int  *ITM_BASE =
                   (volatile unsigned int *)0xE0000000;

#define ITM_ENABLE_ACCESS  { ITM_BASE[0x3EC]=0xC5ACCE55; }
#define ITM_TRACE_PRIV     ITM_BASE[0x390]
#define ITM_TRACE_ENABLE   ITM_BASE[0x380]

#define ITM_TRACE_D8(_channel_,_data_) { \
  volatile unsigned int *_ch_=ITM_BASE+(_channel_); \
  while ( *_ch_ == 0); \
  (*((volatile unsigned char *)(_ch_)))=(_data_); \
}
#define ITM_TRACE_D16(_channel_,_data_) { \
  volatile unsigned int *_ch_=ITM_BASE+(_channel_); \
  while ( *_ch_ == 0); \
  (*((volatile unsigned short *)(_ch_)))=(_data_); \
}
#define ITM_TRACE_D32(_channel_,_data_) { \
  volatile unsigned int *_ch_=ITM_BASE+(_channel_); \
  while ( *_ch_ == 0); \
  *_ch_ = (_data_); \
}


int main()
{
  hardware_setup();
  ITM_ENABLE_ACCESS;
  ITM_TRACE_PRIV          = 0;
  ITM_TRACE_ENABLE        = 0xFFFFFFFF;
```

Data is written to one of the 32 stimulus ports using the ITM_TRACE_D* macros. The Cortex-M makes no assumptions about the contents of each stimulus port and leaves the assignment of them up to the user. For example, one could use each port for a different thread or task within the embedded application.

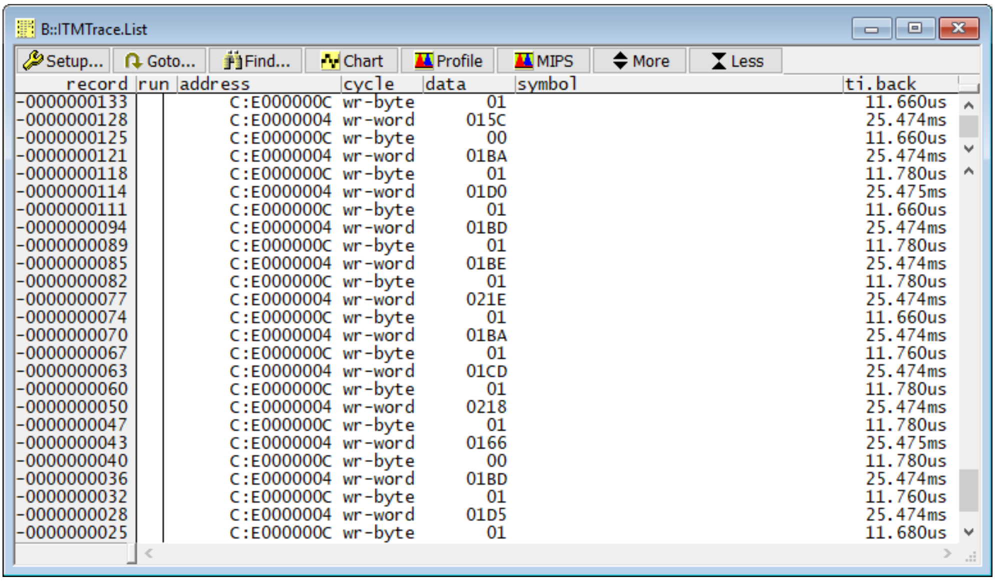
The ITM can be configured from the main ITM window which can be opened using the menu item, clicking the ITM button on the main trace configuration window or by using the **ITM.state** command. The window looks like this.



To capture software generated trace:

1. Switch **ITM.ON**
2. Set Trace.CLOCK to the CPU clock frequency. More information about timing can be found here:**"Time Stamping"**, page 72
3. In the main trace configuration window (**Trace.state**), set
   - METHOD to either Analyzer or CAnalyzer. Only those options which the tools support will be available.
   - Set the state to OFF (**Trace.OFF**)
   - Set AutoArm ON (**Trace.AutoArm ON**). This will start and stop trace sampling as the core starts and stops.
   - Set AutoInit ON (**Trace.AutoInit ON**). This will clear any existing trace data from the buffer before capturing new data.
   - Set Mode to FIFO, Stack or Leash. A discussion of how this affects the trace buffer can be found here: **"Trace Buffer Management"**, page 10.
4. Start the target to capture trace data. The AutoArm option will ensure that capture starts and stops with the target.
5. The target can be halted manually or via a breakpoint.

Show the results with **ITMTrace.List**.


B::ITMTrace.List

Setup... Goto... Find... Chart Profile MIPS More Less

record	run	address	cycle	data	symbol	ti.back
-0000000133		C:E000000C	wr-byte	01		11.660us
-0000000128		C:E0000004	wr-word	015C		25.474ms
-0000000125		C:E000000C	wr-byte	00		11.660us
-0000000121		C:E0000004	wr-word	01BA		25.474ms
-0000000118		C:E000000C	wr-byte	01		11.780us
-0000000114		C:E0000004	wr-word	01D0		25.475ms
-0000000111		C:E000000C	wr-byte	01		11.660us
-0000000094		C:E0000004	wr-word	01BD		25.474ms
-0000000089		C:E000000C	wr-byte	01		11.780us
-0000000085		C:E0000004	wr-word	01BE		25.474ms
-0000000082		C:E000000C	wr-byte	01		11.780us
-0000000077		C:E0000004	wr-word	021E		25.474ms
-0000000074		C:E000000C	wr-byte	01		11.660us
-0000000070		C:E0000004	wr-word	01BA		25.474ms
-0000000067		C:E000000C	wr-byte	01		11.760us
-0000000063		C:E0000004	wr-word	01CD		25.474ms
-0000000060		C:E000000C	wr-byte	01		11.780us
-0000000050		C:E0000004	wr-word	0218		25.474ms
-0000000047		C:E000000C	wr-byte	01		11.780us
-0000000043		C:E0000004	wr-word	0166		25.475ms
-0000000040		C:E000000C	wr-byte	00		11.780us
-0000000036		C:E0000004	wr-word	01BD		25.474ms
-0000000032		C:E000000C	wr-byte	01		11.760us
-0000000028		C:E0000004	wr-word	01D5		25.474ms
-0000000025		C:E000000C	wr-byte	01		11.680us

The individual channels can be split out for display by using the **ITM.PortFilter** command. For example, to view data from just channel 1 use:

```
ITM.PortFilter 0x01
ITMTrace.FLOWPROCESS
ITMTrace.List
```

To view additional channels, use **ITM.PortFilter** with a binary value for the channels you wish to include. TRACE32 uses 0yXXXXXXXX to indicate a binary value. For example:

```
ITM.PortFilter 0y0011010
ITMTrace.FLOWPROCESS
ITMTrace.List
```

A complete source code example can be found under:

`~~/demo/arm/hardware/stm32/stm32f3/custom_itmprintf`

An example `itm_printf()` function is shown below.

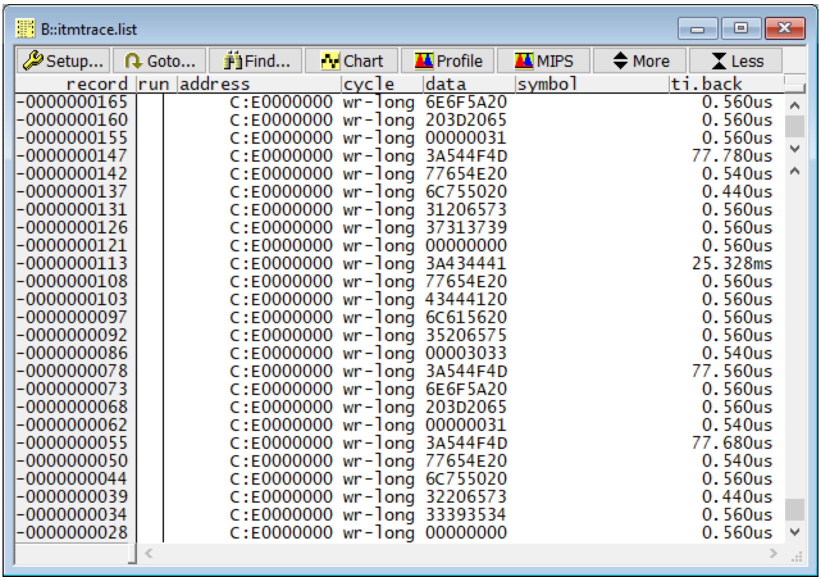
```
#include <stdarg.h>
#include "itm_printf.h"

extern int vsprintf(char *buf, const char *fmt, va_list args);
void ITM_printf(const char *format,...)
{
    union {
  char c[100];
  unsigned int i[25];
    } line;
    unsigned int v;
    int i,j,l;
    va_list ap;
    va_start(ap, format);
    l=vsprintf(&(line.c[0]),format,ap);
    l++;
    l++;
    va_end(ap);
    i=0;
    j=0;
    while (i<l)
    {
        v=line.i[j];
        i+=4;
        j++;
        if (i>l)
            v&=(0xFFFFFFFF>>((i-l)*8));
        ITM_TRACE_D32(0,v);
    }
}
```

NOTE:	The version of `vsprintf()` provided by your compiler MUST be thread safe and re-entrant. If not, consider protecting it with a mutex or similar.

This function can be called in the application just like a regular printf().

```
ITM_printf("ADC: New ADC value %d", new_ADC);
```

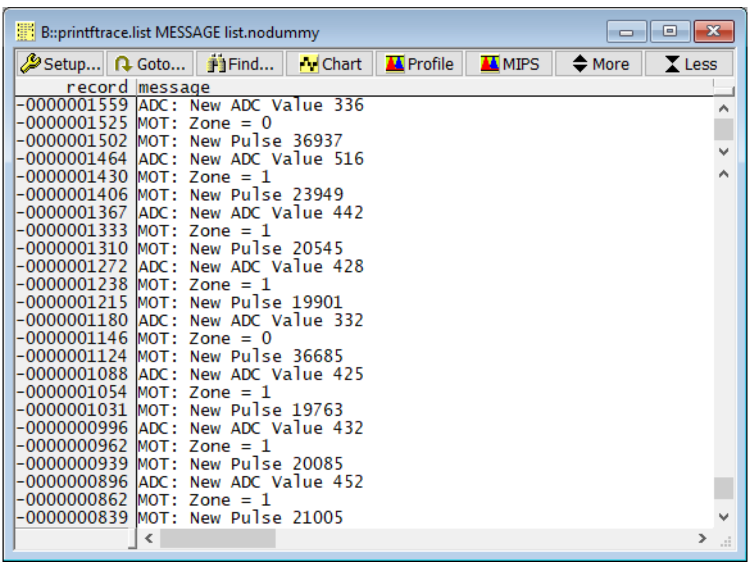
With ITM set to collect data trace, a standard trace listing will look like this.



record	run	address	cycle	data	symbol	ti.back
-0000000165		C:E0000000	wr-long	6E6F5A20		0.560us
-0000000160		C:E0000000	wr-long	203D2065		0.560us
-0000000155		C:E0000000	wr-long	00000031		0.560us
-0000000147		C:E0000000	wr-long	3A544F4D		77.780us
-0000000142		C:E0000000	wr-long	77654E20		0.540us
-0000000137		C:E0000000	wr-long	6C755020		0.440us
-0000000131		C:E0000000	wr-long	31206573		0.560us
-0000000126		C:E0000000	wr-long	37313739		0.560us
-0000000121		C:E0000000	wr-long	00000000		0.560us
-0000000113		C:E0000000	wr-long	3A434441		25.328ms
-0000000108		C:E0000000	wr-long	77654E20		0.560us
-0000000103		C:E0000000	wr-long	43444120		0.560us
-0000000097		C:E0000000	wr-long	6C615620		0.560us
-0000000092		C:E0000000	wr-long	35206575		0.560us
-0000000086		C:E0000000	wr-long	00003033		0.540us
-0000000078		C:E0000000	wr-long	3A544F4D		77.560us
-0000000073		C:E0000000	wr-long	6E6F5A20		0.560us
-0000000068		C:E0000000	wr-long	203D2065		0.560us
-0000000062		C:E0000000	wr-long	00000031		0.540us
-0000000055		C:E0000000	wr-long	3A544F4D		77.680us
-0000000050		C:E0000000	wr-long	77654E20		0.540us
-0000000044		C:E0000000	wr-long	6C755020		0.560us
-0000000039		C:E0000000	wr-long	32206573		0.440us
-0000000034		C:E0000000	wr-long	33393534		0.560us
-0000000028		C:E0000000	wr-long	00000000		0.560us

To display the results correctly, use **PrintfTrace** instead.

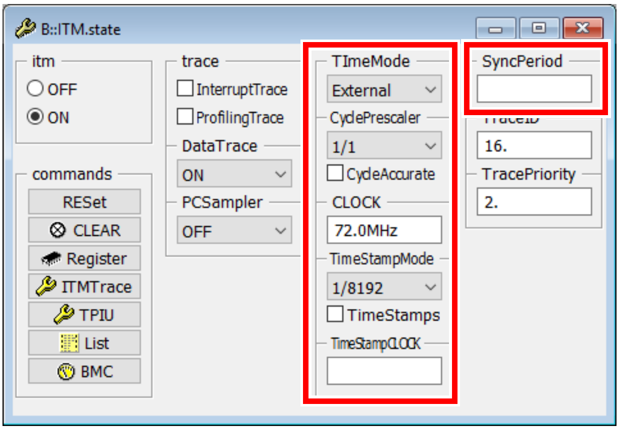
```
PrintfTrace.List MESSAGE List.NoDummy
```



record	message
-0000001559	ADC: New ADC Value 336
-0000001525	MOT: Zone = 0
-0000001502	MOT: New Pulse 36937
-0000001464	ADC: New ADC Value 516
-0000001430	MOT: Zone = 1
-0000001406	MOT: New Pulse 23949
-0000001367	ADC: New ADC Value 442
-0000001333	MOT: Zone = 1
-0000001310	MOT: New Pulse 20545
-0000001272	ADC: New ADC Value 428
-0000001238	MOT: Zone = 1
-0000001215	MOT: New Pulse 19901
-0000001180	ADC: New ADC Value 332
-0000001146	MOT: Zone = 0
-0000001124	MOT: New Pulse 36685
-0000001088	ADC: New ADC Value 425
-0000001054	MOT: Zone = 1
-0000001031	MOT: New Pulse 19763
-0000000996	ADC: New ADC Value 432
-0000000962	MOT: Zone = 1
-0000000939	MOT: New Pulse 20085
-0000000896	ADC: New ADC Value 452
-0000000862	MOT: Zone = 1
-0000000839	MOT: New Pulse 21005

Time Stamping

To provide any kind of timing analysis the trace decoders need some time stamping information. There are several options, each with its own set of advantages and drawbacks.



ITM.TImeMode	Controls the timing information used by the trace decoders. Timestamps can be off, generated by the ITM or generated by the TRACE32 hardware as each packet is decoded. An option exists to combine internal and external timestamps to allow better correlation with other trace sources.
ITM.CyclePrescaler	This sets the divider from the core clock to generate the timestamp clock for ITM messages. ITM.CLOCK still contains the core clock value.
ITM.CycleAccurate	If this option is enabled, the ITM will insert cycle count packets into the trace stream. Timestamps are based upon the cycle count and the value of the core clock. If this option is not selected, TRACE32 hardware tools will generate timestamps for packets as they are de-queued from the trace interface.
ITM.TimeStampCLOCK	This is the value of the global timestamp clock and is used for multi-core systems.
ITM.TimeStampMode	Determines whether the timestamp clock is derived form the CPU clock or TPIU clock.
ITM.TimeStamps	Enable global timestamp packets.
ITM.CLOCK	This is the frequency of the cpu core clock. It is used in conjunction with CycleAccurate mode to calculate how long between trace events.
ITM.SyncPeriod	Controls how frequently (number of clock cycles) a synch packet is generated. The default is 1024.

In most cases, the default options are the correct ones to choose.

Stream Mode

ITM trace data can also be streamed to a file on the local hard drive. The configuration is very similar to that of ETM which is explained here: **“ETM Stream Mode”**, page 17.

Trace data can also be streamed to a shared library for custom processing or handling. An example of this can be found under `~~/demo/arm/hardware/xmc/xmc4500/custom_itmprintf`. The full source code for the DLL or shared library is in the DLL sub-directory, along with make files and instructions on how to build for your host operating system. How to create a build such a DLL or shared library is beyond the scope of this document.

To load a shared object for custom trace processing use the command

<trace>.CustomTraceLoad “name” <*file*>

To use, set **Trace.Mode PIPE**. The DLL is responsible for managing its own display of the data that it has processed. The TRACE32 api allows for the creation of a custom command to display the data in text only format in a window inside TRACE32. If a more complex display of the data then this is required it is the responsibility of the DLL to do this. By using stream mode like this, the trace data is not stored but parsed on-the-fly.

Pipe Mode

The ITM trace data can be streamed to a pipe (Named pipe in Windows, pipe or FIFO in Linux or MacOS). This allows for a host application to react in almost real-time to data being generated by the target. There is no buffer to fill, so theoretically, the trace could be of unlimited duration.

An example of creating a Windows forms application with a named pipe using Visual Studio (tested on Windows 7, 8 and 10 with Visual Studio 2015 and 2017) can be found under `~~/demo/arm/hardware/stm32/stm32f3/custom_itmprintf`.

The script in this directory shows how to configure TRACE32 for streaming ITM data to a pipe.

To stream ITM data to a named pipe:

1. Set **Trace.Mode PIPE**
2. Launch the Host OS application that will create and open the pipe
3. Connect TRACE32 to the pipe with **Trace.PipeWRITE** "*<pipe>*"
4. Start the target

The ITM data is exported from TRACE32 to the pipe in a packet with the following format.

Byte	Meaning
0	The size of the rest of the packet in bytes. This byte does not count towards the total length.
1 - 8	Timestamp in Little Endian format
9	Message type: data = 1 indicates there are 8 bits data = 2 indicates there are 16 bits data = 3 -indicates there are 32 bits
10	ITM Channel ID
11 - 14	Payload in Little Endian format.

Up to 8 pipes can be simultaneously supported by TRACE32. Calling **Trace.PipeWRITE** without any arguments will close all open pipes.