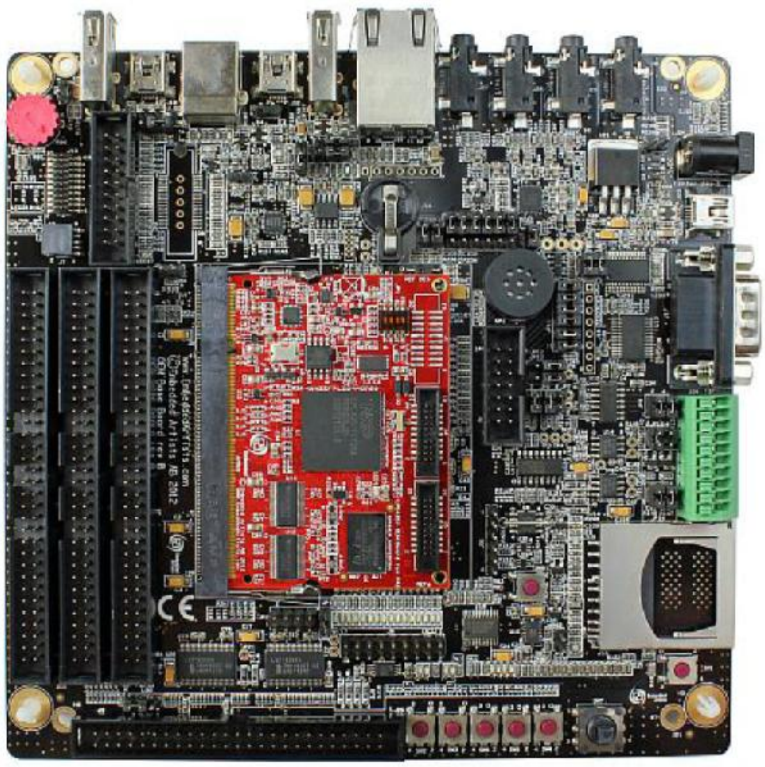
µTrace setup guide for Embedded Artists LPC4357 development board



About this document

This document will explain how to get up and running with the Lauterbach µTrace unit and the Embedded Artists LPC4357 development board. The document goes with a zip file containing the examples and configuration scripts described herein.

Two examples are provided: one runs a simple loop on the Cortex-M4 core of the device; the other runs a similar program on the Cortex-M4 and a very simple loop on the Cortex-M0 core in the device.

Pre-Requisites

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

Setup Procedures

Unzip the archive EA_LPC4357.zip so that it over-writes files in the $T32SYS directory. You may wish to take a backup of this directory beforehand.

Connect the CombiProbe header to Socket A on the µTrace.

Connect the CombiProbe header (using the MIPI34-MIPI20T adapter) to either Socket J1 on the SODIMM board (see Figure1) or to socket J10 on the main board (see Figure 2). For systems using off-chip trace, it is recommended to use J1 on the SODIMM due to the shorter track lengths of the trace signals.

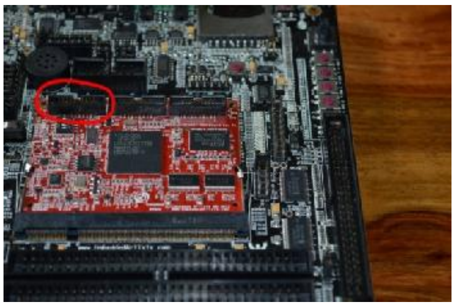


Figure 1: J1 on SODIMM

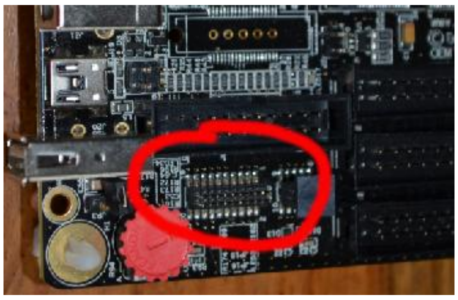
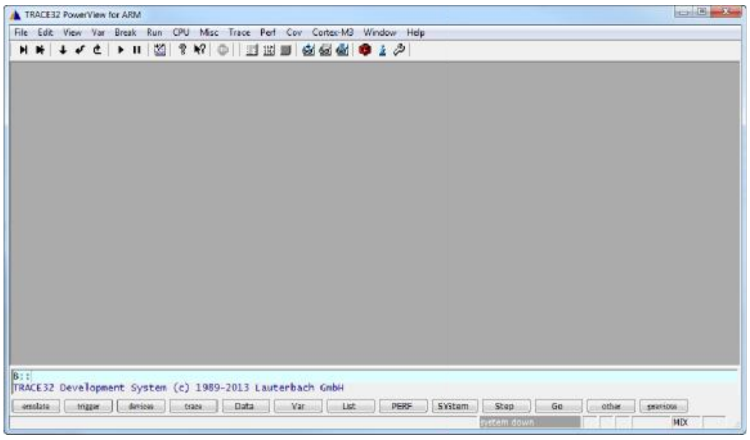


Figure 2: J10 on main board

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

Power on the µTrace and then power on the LPC4357 board.

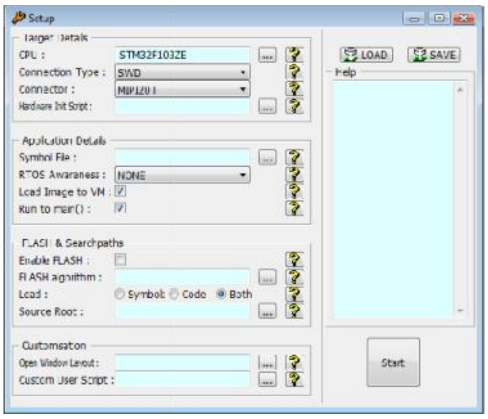
Start the TRACE32 software and you should see something like Figure 3 below.


TRACE32 PowerView for ARM
File Edit View Var Break Run CPU Misc Trace Perf Cov Cortex-M3 Window Help
emulate trigger devices trace Data Var List PERF SYStem Step Go other previous
system down
MIX

```
B::
TRACE32 Development System (c) 1989-2013 Lauterbach GmbH
```

Figure 3: TRACE32

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



CPU : STM32F103ZE
Connection Type : SWD
Connector : MIPI20T
Hardware Init Script :
Symbol File :
RTOS Awareness : NONE
Load Image to VM : [✓]
Run to main() : [✓]
Enable FLASH : [ ]
FLASH algorithm :
Load : Symbol Code [✓ Both]
Source Root :
Open Window Layout :
Custom User Script :

Figure 4: startup.cmm

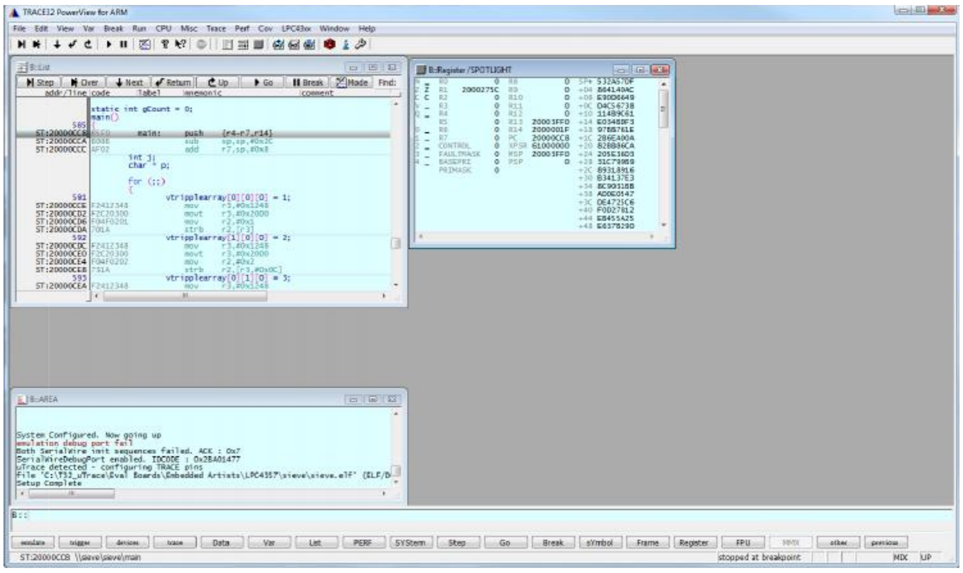
TRACE32 can be configured to call this script each time you start it up. To do this, edit the **t32.cmm** file in the $T32SYS directory. At the bottom of the file, just before the line that reads "ENDDO" add a line which reads

```
do startup.cmm
```

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

```
$T32SYS\Eval Boards\Embedded Artists\LPC4357
```

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


TRACE32 PowerView for ARM
File Edit View Var Break Run CPU Misc Trace Perf Cov LPC43xx Window Help
B::List
Step Over Next Return Up Go Break Mode Find:
B::Register /SPOTLIGHT
B::AREA
emulate trigger devices trace Data Var List PERF SYStem Step Go Break sYmbol Frame Register FPU MMX other previous
stopped at breakpoint
MIX UP

```
     addr/line code       label      mnemonic               comment
                static int gCount = 0;
                main()
           585  {
ST:20000CC8 B5F0   main:    push    {r4-r7,r14}
ST:20000CCA B08B            sub     sp,sp,#0x2C
ST:20000CCC AF02            add     r7,sp,#0x8
                      int j;
                      char * p;

                      for (;;)
                      {
           591               vtripplearray[0][0][0] = 1;
ST:20000CCE F2412348        mov     r3,#0x1248
ST:20000CD2 F2C20300        movt    r3,#0x2000
ST:20000CD6 F04F0201        mov     r2,#0x1
ST:20000CDA 701A            strb    r2,[r3]
           592               vtripplearray[1][0][0] = 2;
ST:20000CDC F2412348        mov     r3,#0x1248
ST:20000CE0 F2C20300        movt    r3,#0x2000
ST:20000CE4 F04F0202        mov     r2,#0x2
ST:20000CE8 751A            strb    r2,[r3,#0x0C]
           593               vtripplearray[0][1][0] = 3;
ST:20000CEA F2412348        mov     r3,#0x1248
```

```
N _  R0             0  R8            0  SP> 532A570F
Z Z  R1      2000275C  R9            0  +04 B64140AC
C C  R2             0  R10           0  +08 E90D6649
V _  R3             0  R11           0  +0C 04C56738
Q _  R4             0  R12           0  +10 11489C61
     R5             0  R13    20003FFD  +14 E03480F3
0 _  R6             0  R14    2000001F  +18 97B8761E
1 _  R7             0  PC     20000CC8  +1C 286EA00A
2 _  CONTROL        0  XPSR   61000000  +20 828B86CA
3 _  FAULTMASK      0  MSP    20003FFD  +24 205E36D3
4 _  BASEPRI        0  PSP           0  +28 31C79869
     PRIMASK        0                   +2C 89318916
                                        +30 834137E3
                                        +34 BC90318B
                                        +38 AD0E0547
                                        +3C DE4725C6
                                        +40 F0027812
                                        +44 E8455425
                                        +48 E637829D
```

```
System Configured. Now going up
emulation debug port fail
Both SerialWire init sequences failed. ACK : 0x7
SerialWireDebugPort enabled. IDCODE : 0x2BA01477
uTrace detected - configuring TRACE pins
File 'C:\T32_uTrace\Eval Boards\Embedded Artists\LPC4357\sieve\sieve.elf' (ELF/D
Setup Complete
```

```
B::
ST:20000CC8 \\sieve\sieve\main
```

Figure 5: Demo loaded

This will give you JTAG control of the target. The connection is via the Serial Wire Debug (SWD) interface to the Cortex-M4.

Trace Setup

In order to get ETM trace from the Cortex-M4 it is necessary to make a small modification to the SODIMM board. On the back of the SODIMM board is a small 0Ohm resistor as position SJ1. This can be seen in Figure 6. It is shown here in the default position connecting pads 1-2. The resistor must be moved to connect pads 2-3 to enable the off-chip trace signals.

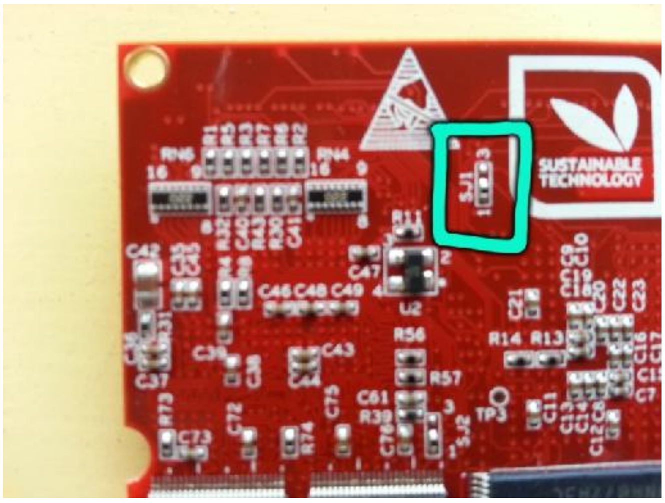


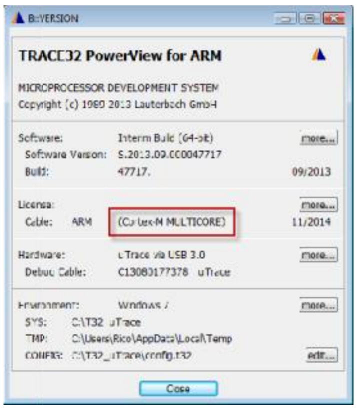
Figure 6: SODIMM resistor

The supplied configuration for the LPC4357 board assumes a working trace port and will setup the pin multiplexing and TRACE32 software accordingly. However, until this modification has been made no trace data can be collected.

Multi-Core Example

The second example loads code which is executed on both the Cortex-M4 and the Cortex-M0 in the LPC4357 device. To debug both cores requires that the µTrace has a multi core license added to it. This can be checked by starting TRACE32 and selecting **`About TRACE32...`** from the **`Help`** menu. Figure 7 shows a multi core system with the license highlighted. If your system does not have a multi core license, please contact your local Lauterbach sales office. Contact details can be found at:

[`http://www.lauterbach.com/sales`](http://www.lauterbach.com/sales)


B::VERSION

TRACE32 PowerView for ARM

MICROPROCESSOR DEVELOPMENT SYSTEM
Copyright (c) 1989 2013 Lauterbach GmbH

```
Software:              Interim Build (64-bit)
  Software Version:    S.2013.09.000047717
  Build:               47717.                      09/2013

License:
  Cable:      ARM      (Cortex-M MULTICORE)        11/2014

Hardware:              µTrace via USB 3.0
  Debug Cable:         C13080177378   uTrace

Environment:           Windows 7
  SYS:       C:\T32_uTrace
  TMP:       C:\Users\Rico\AppData\Local\Temp
  CONFIG:    C:\T32_uTrace\config.t32
```

Figure 7: Multi core license

Before launching the multi core example some changes need to be made to your **`config.t32`** file which is located in the T32SYS directory. These changes will not affect the single core functionality but will extend TRACE32 to allow two instances of the software to connect to the same µTrace unit. This is required to debug both cores in an Asymmetric Multi Processor (AMP) setup like this.

Edit **`config.t32`** in your favourite text editor and at the bottom of the file add a new section that looks like this. The blank line above and below are very important - don't miss them.

```

IC=NETASSIST
PORT=22000

```

Alter the **`PBI=`** section so that it now looks like this. Again, don't forget the blank line above and below.

```

PBI=
USB
CORE=1

```

Save the changes.

Now take a copy of your **config.t32** file and call it **config-2nd.t32**. Place it in your $T32SYS directory. This will need editing as well. The full contents of my **config.t32** file are shown here. Make sure the TMP declaration is correct for your setup; it should be the same as the one in your **config.t32** file. The main changes are summarised below:

```
; Environment variables
OS=
ID=T32-2nd
TMP=C:\Users\rc\AppData\Local\Temp
SYS=C:\T32_uTrace

PBI=
USB
CORE=2

IC=NETASSIST
PORT=22001

; Printer settings
PRINTER=WINDOWS

; Screen fonts
SCREEN=
FONT=SMALL
HEADER=TRACE32 2nd Core
```

OS=
    Change ID=
    Change TMP

PBI=
    CORE=2

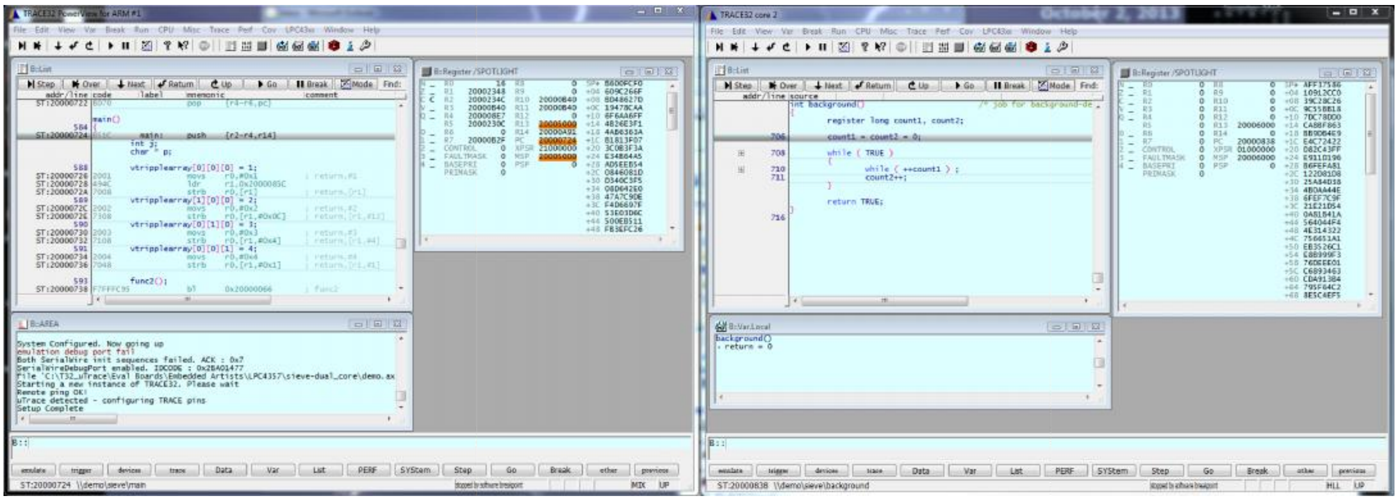
IC=
    Change PORT=

SCREEN=
    Add HEADER=

As before, connect the hardware and launch TRACE32. Run the **startup.cmm** script, if your system is not already configured to auto run this. Click the LOAD button and browse for the **dual_core_demo.t32ini** file in the

$T32SYS\Eval Boards\Embedded Artists\LPC4357

Click the big start button. After the first core has been initialised and code loaded a second instance of TRACE32 will be started and connected to the Cortex-M0 core. The symbols will be loaded and you will end up with two instances of TRACE32, looking like Figure 8. These have been resized to make the image fit here.



```
System Configured. Now going up
emulation debug port fail
Both SerialWire init sequences failed. ACK : 0x7
SerialWireDebugPort enabled. IDCODE : 0x2BA01477
file 'C:\T32_uTrace\Eval Boards\Embedded Artists\LPC4357\sieve-dual_core\demo.ax
Starting a new instance of TRACE32. Please wait
Remote ping OK!
uTrace detected - configuring TRACE pins
Setup Complete
```

```
int background()                        /* job for background-de
{
        register long count1, count2;

706     count1 = count2 = 0;

708     while ( TRUE )
        {
710             while ( ++count1 ) ;
711             count2++;
        }

        return TRUE;
716 }
```

Figure 8: Multi core Configuration

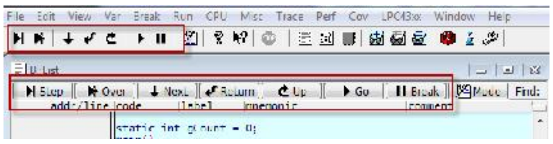
The target has been configured so that each core can be single stepped independently but when either core starts or stops (user controlled or via a breakpoint) the other core will also start or stop. These settings can be adjusted using the **Synch** command. Please refer to the TRACE32 documentation.

Tutorials

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

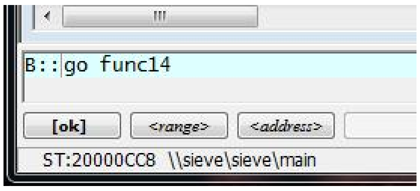
Run Control

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


File Edit View Var Break Run CPU Misc Trace Perf Cov LPC43xx Window Help
Step
Over
Next
Return
Up
Go
Break
Mode
Find:
addr/line code
label
mnemonic
comment

```
static int plcount = 0;
```

Figure T1: Run Control Buttons



```
B::go func14
```

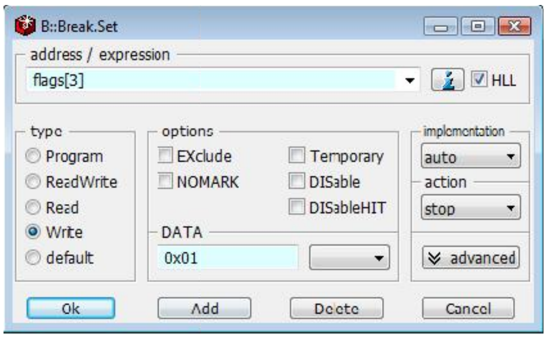
```
ST:20000CC8  \\sieve\sieve\main
```

Figure T2: The Go command

Breakpoints

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

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



address / expression: flags[3] [✓ HLL]

type: Program, ReadWrite, Read, [✓ Write], default

options: EXclude, NOMARK, Temporary, DISable, DISableHIT

DATA: 0x01

implementation: auto

action: stop

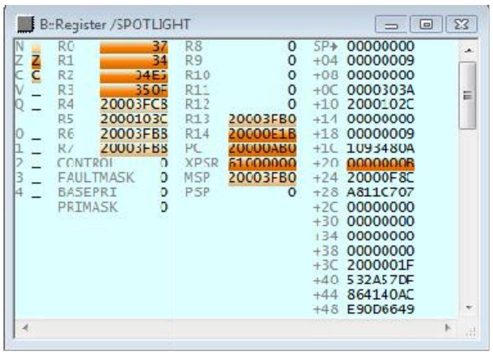
Figure T3: Break.set

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

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

Registers

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

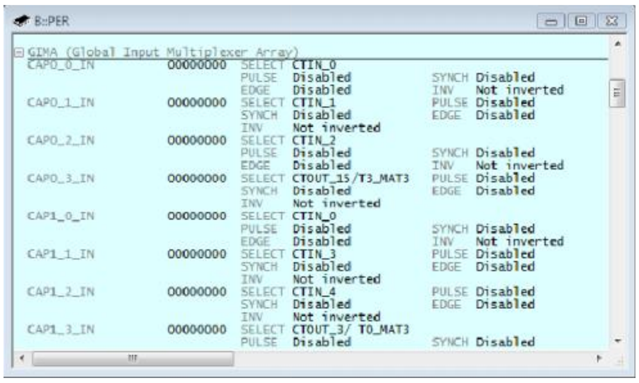


```
N    R0          37  R8           0  SP> 00000000
Z Z  R1          34  R9           0  +04 00000009
C C  R2         34E3 R10          0  +08 00000000
V _  R3         350F R11          0  +0C 0000303A
Q _  R4     20003FC8 R12          0  +10 2000102C
     R5     2000103C R13   20003FB0  +14 00000000
0 _  R6     20003FB8 R14   20000E1B  +18 00000009
1 _  R7     20003FB8 PC    20000AB0  +1C 1093480A
2 _  CONTROL       0 XPSR  61000000  +20 0000000B
3 _  FAULTMASK     0 MSP   20003FB0  +24 20000F8C
4 _  BASEPRI       0 PSP          0  +28 A811C707
     PRIMASK       0                 +2C 00000000
                                     +30 00000000
                                     +34 00000000
                                     +38 00000000
                                     +3C 2000001F
                                     +40 532A57DF
                                     +44 864140AC
                                     +48 E90D6649
```

Figure T4: Highlighted Registers

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

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



```
B::PER
⊟ GIMA (Global Input Multiplexer Array)
  CAP0_0_IN   00000000  SELECT CTIN_0
                        PULSE  Disabled            SYNCH Disabled
                        EDGE   Disabled            INV   Not inverted
  CAP0_1_IN   00000000  SELECT CTIN_1              PULSE Disabled
                        SYNCH  Disabled            EDGE  Disabled
                        INV    Not inverted
  CAP0_2_IN   00000000  SELECT CTIN_2
                        PULSE  Disabled            SYNCH Disabled
                        EDGE   Disabled            INV   Not inverted
  CAP0_3_IN   00000000  SELECT CTOUT_15/T3_MAT3    PULSE Disabled
                        SYNCH  Disabled            EDGE  Disabled
                        INV    Not inverted
  CAP1_0_IN   00000000  SELECT CTIN_0
                        PULSE  Disabled            SYNCH Disabled
                        EDGE   Disabled            INV   Not inverted
  CAP1_1_IN   00000000  SELECT CTIN_3              PULSE Disabled
                        SYNCH  Disabled            EDGE  Disabled
                        INV    Not inverted
  CAP1_2_IN   00000000  SELECT CTIN_4              PULSE Disabled
                        SYNCH  Disabled            EDGE  Disabled
                        INV    Not inverted
  CAP1_3_IN   00000000  SELECT CTOUT_3/ T0_MAT3
                        PULSE  Disabled            SYNCH Disabled
```

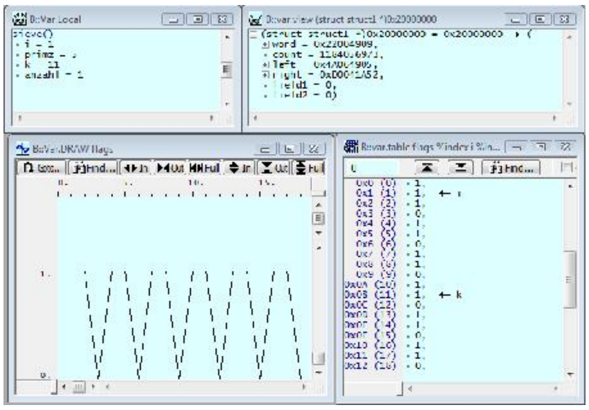
Figure T5: Peripheral Registers

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

Variables

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

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



B::Var.Local
B::var.view (struct struct1 *)0x20000000
B::Var.DRAW flags
Goto...
Find...
In
Out
Full
B::var.table flags %index i %in...
Find...

```
sieve()
 · i = 1
 · primz = 3
 · k = 11
 · anzahl = 1
```

```
⊟ (struct struct1 *)0x20000000 = 0x20000000 → (
   ⊞ word = 0x22004909,
   · count = 1164016971,
   ⊞ left = 0x47000905,
   ⊞ right = 0xD0041A52,
   · field1 = 0,
   · field2 = 0)
```

```
0x0  (0)   · 1,
0x1  (1)   · 1,   ← i
0x2  (2)   · 1,
0x3  (3)   · 0,
0x4  (4)   · 1,
0x5  (5)   · 1,
0x6  (6)   · 0,
0x7  (7)   · 1,
0x8  (8)   · 1,
0x9  (9)   · 0,
0x0A (10)  · 1,
0x0B (11)  · 1,   ← k
0x0C (12)  · 0,
0x0D (13)  · 1,
0x0E (14)  · 1,
0x0F (15)  · 0,
0x10 (16)  · 1,
0x11 (17)  · 1,
0x12 (18)  · 0,
```

Figure T6: Variable Views

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

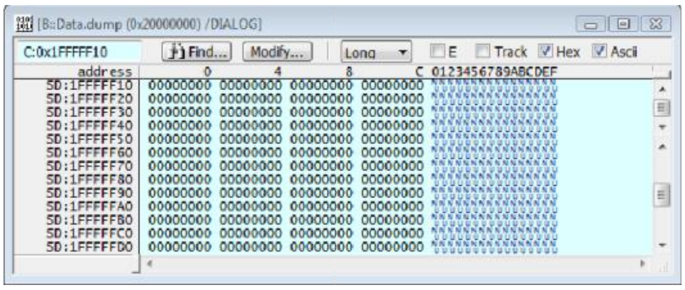
```
go sieve
```

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

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

Memory

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



```
[B::Data.dump (0x20000000) /DIALOG]
C:0x1FFFFF10   Find...  Modify...  Long   E  Track  [✓] Hex  [✓] Ascii
      address         0        4        8        C  0123456789ABCDEF
  SD:1FFFFF10  00000000 00000000 00000000 00000000
  SD:1FFFFF20  00000000 00000000 00000000 00000000
  SD:1FFFFF30  00000000 00000000 00000000 00000000
  SD:1FFFFF40  00000000 00000000 00000000 00000000
  SD:1FFFFF50  00000000 00000000 00000000 00000000
  SD:1FFFFF60  00000000 00000000 00000000 00000000
  SD:1FFFFF70  00000000 00000000 00000000 00000000
  SD:1FFFFF80  00000000 00000000 00000000 00000000
  SD:1FFFFF90  00000000 00000000 00000000 00000000
  SD:1FFFFFA0  00000000 00000000 00000000 00000000
  SD:1FFFFFB0  00000000 00000000 00000000 00000000
  SD:1FFFFFC0  00000000 00000000 00000000 00000000
  SD:1FFFFFD0  00000000 00000000 00000000 00000000
```

Figure T7: Memory Dump

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

Try the following:

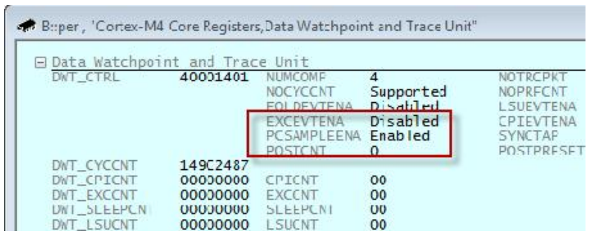
```
var.view flags /SPOTLIGHT
data.dump flags /dialog /SPOTLIGHT
```

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

Performance Analysis

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

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



```
B::per , 'Cortex-M4 Core Registers,Data Watchpoint and Trace Unit"
⊟ Data Watchpoint and Trace Unit
  DWT_CTRL       40001401  NUMCOMP      4               NOTRCPKT
                           NOCYCCNT     Supported       NOPRFCNT
                           FOLDEVTENA   Disabled        LSUEVTENA
                           EXCEVTENA    Disabled        CPIEVTENA
                           PCSAMPLEENA  Enabled         SYNCTAP
                           POSTCNT      0               POSTPRESET
  DWT_CYCCNT     149C2487
  DWT_CPICNT     00000000  CPICNT       00
  DWT_EXCCNT     00000000  EXCCNT       00
  DWT_SLEEPCN    00000000  SLEEPCNT     00
  DWT_LSUCNT     00000000  LSUCNT       00
```

Figure T8: PC Snoop

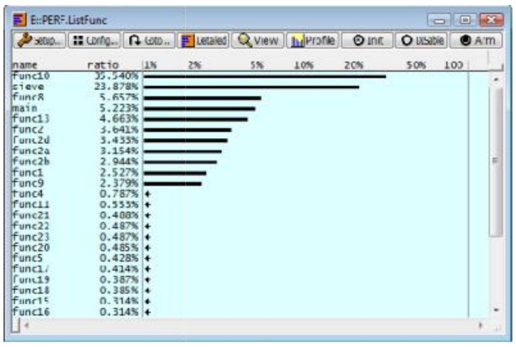
If this is available non-intrusive metrics can be collected. Set the METHOD in the Perf window to **`Snoop`**.

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

To view relative function runtime analysis:

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

An example is shown in figure T9.



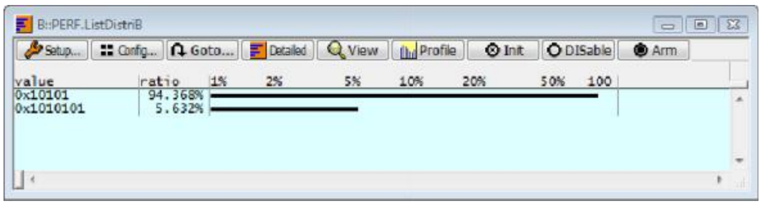
```
B::PERF.ListFunc
Setup...  Config...  Goto...  Detailed  View  Profile  Init  Disable  Arm
name      ratio     1%   2%   5%   10%   20%   50%  100
func10    35.540%
sieve     23.878%
func8      5.657%
main       5.223%
func13     4.663%
func2      3.641%
func2d     3.433%
func2a     3.154%
func2b     2.944%
func1      2.527%
func9      2.379%
func4      0.787%
func11     0.553%
func21     0.488%
func22     0.487%
func23     0.487%
func20     0.485%
func5      0.428%
func17     0.414%
func19     0.387%
func18     0.385%
func15     0.314%
func16     0.314%
```

Figure T9: PC Snoop

To view data values:

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

An example is shown in figure T10.



```
value          ratio     1%    2%     5%    10%    20%     50%   100
0x10101         94.368%
0x1010101        5.632%
```

Figure T10: Data Snoop

On-Line Help

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

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

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

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

Trace Examples

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

Basic Trace Collection

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

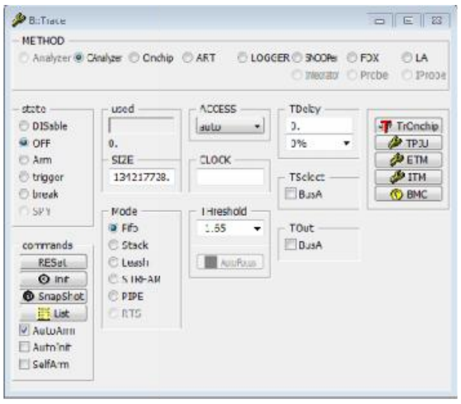
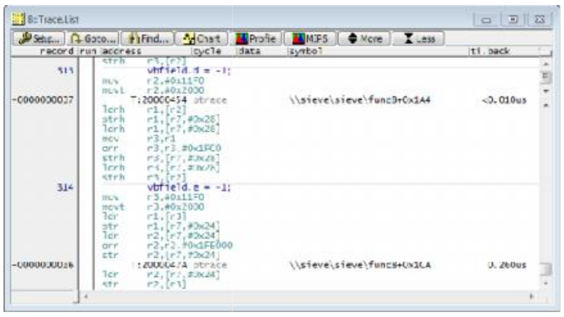

```
B::Trace
METHOD  ( ) Analyzer  (•) CAnalyzer  ( ) Onchip  ( ) ART  ( ) LOGGER  ( ) SNOOPer  ( ) FDX  ( ) LA
        ( ) Integrator  ( ) Probe  ( ) IProbe
state        used              ACCESS       TDelay
( ) DISable  0.                auto         0.
(•) OFF      SIZE              CLOCK        0%
( ) Arm      134217728.                     TSelect
( ) trigger                                 [ ] BusA
( ) break    Mode              THreshold
( ) SPY      (•) Fifo          1.65         TOut
             ( ) Stack                      [ ] BusA
commands     ( ) Leash         AutoFocus
RESet        ( ) STREAM
Init         ( ) PIPE
SnapShot     ( ) RTS
List
[✓] AutoArm
[ ] AutoInit
[ ] SelfArm
```

Figure T11: Trace Configuration

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

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



```
    record run address        cycle   data     symbol                              ti.back
                strh    r3,[r7]
       313                      vbfield.d = -1;
                mov     r2,#0x11F0
                movt    r2,#0x2000
-0000000037          T:20000454 ptrace         \\sieve\sieve\func3+0x1A4            <0.010us
                ldrh    r1,[r2]
                strh    r1,[r7,#0x28]
                ldrh    r1,[r7,#0x28]
                mov     r3,r1
                orr     r3,r3,#0x1FC0
                strh    r3,[r7,#0x28]
                ldrh    r3,[r7,#0x28]
                strh    r3,[r2]
       314                      vbfield.e = -1;
                mov     r3,#0x11F0
                movt    r3,#0x2000
                ldr     r1,[r3]
                str     r1,[r7,#0x24]
                ldr     r2,[r7,#0x24]
                orr     r2,r2,#0x1FE000
                str     r2,[r7,#0x24]
-0000000026          T:2000047A ptrace         \\sieve\sieve\func3+0x1CA             0.260us
                ldr     r2,[r7,#0x24]
                str     r2,[r3]
```

Figure T12: Program flow trace

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

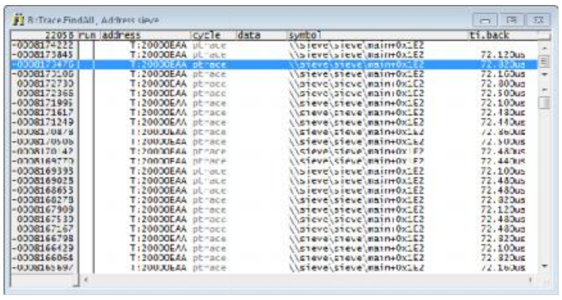

```
B::Trace.FindAll , Address sieve
       22058 run address       cycle   data  symbol                        ti.back
-0008174222        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2
-0008173845        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.120us
-0008173476        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.320us
-0008173106        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.160us
-0008172730        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.800us
-0008172366        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.500us
-0008171995        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.100us
-0008171617        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.480us
-0008171249        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.440us
-0008170878        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.860us
-0008170506        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.500us
-0008170142        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.480us
-0008169770        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.440us
-0008169393        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.100us
-0008169025        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.480us
-0008168653        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.480us
-0008168278        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.820us
-0008167909        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.120us
-0008167533        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.480us
-0008167167        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.480us
-0008166798        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.820us
-0008166429        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.100us
-0008166064        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.820us
-0008165697        T:20000EAA ptrace         \\sieve\sieve\main+0x1E2      72.160us
```

Figure T13: Search Results

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

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

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

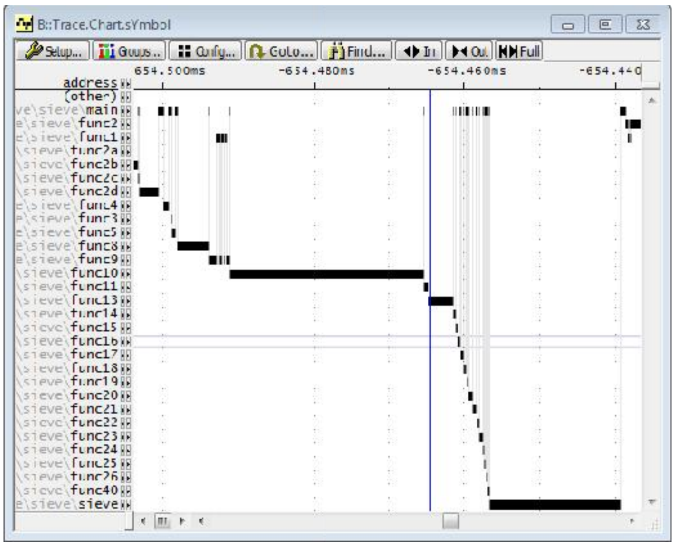


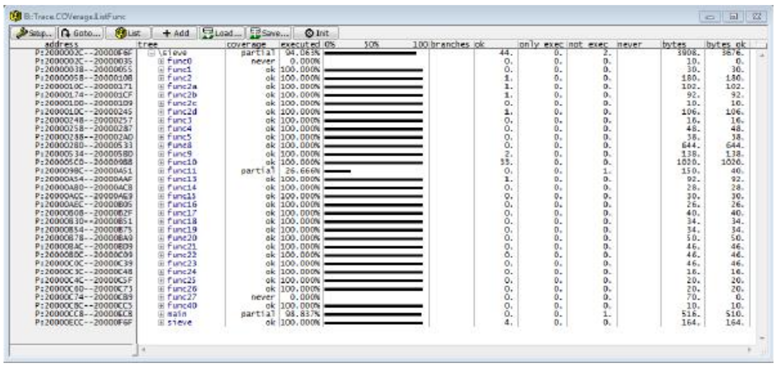
Figure T14: Trace Chart

The zoom can be controlled in a number of ways:

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

Code Coverage

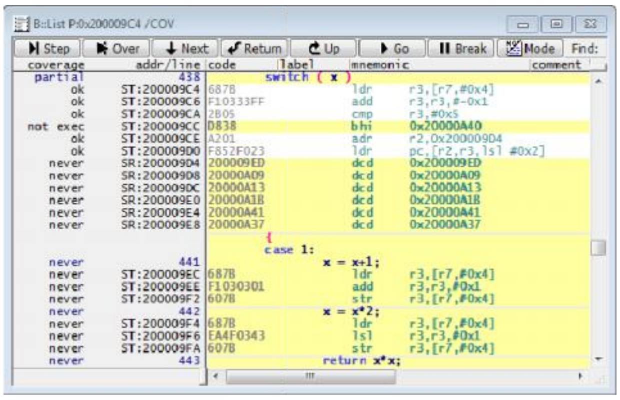
With program flow trace available it is easy to get code coverage information. Select **`Add Tracebuffer`** from the **`Cov`** menu. This will add the contents of the current trace buffer to the existing code coverage database. Like this multiple test runs can be aggregated. Now select **`List functions`**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



address	tree	coverage	executed	branches ok	only exec	not exec	never	bytes	bytes ok
P:2000002C--20000F6F	\sieve	partial	94.063%	44.	0.	2.		3908.	3676.
P:2000002C--20000035	func0	never	0.000%	0.	0.	0.		10.	0.
P:20000038--20000055	func1	ok	100.000%	0.	0.	0.		30.	30.
P:20000058--2000010B	func2	ok	100.000%	1.	0.	0.		180.	180.
P:2000010C--20000171	func2a	ok	100.000%	1.	0.	0.		102.	102.
P:20000174--200001CF	func2b	ok	100.000%	1.	0.	0.		92.	92.
P:200001D0--200001D9	func2c	ok	100.000%	0.	0.	0.		10.	10.
P:200001DC--20000245	func2d	ok	100.000%	1.	0.	0.		106.	106.
P:20000248--20000257	func3	ok	100.000%	0.	0.	0.		16.	16.
P:20000258--20000287	func4	ok	100.000%	0.	0.	0.		48.	48.
P:20000288--200002AD	func5	ok	100.000%	0.	0.	0.		38.	38.
P:200002B0--20000533	func8	ok	100.000%	0.	0.	0.		644.	644.
P:20000534--200005BD	func9	ok	100.000%	2.	0.	0.		138.	138.
P:200005C0--2000098B	func10	ok	100.000%	55.	0.	0.		1020.	1020.
P:2000098C--20000A51	func11	partial	26.666%	0.	0.	1.		150.	40.
P:20000A54--20000AAF	func13	ok	100.000%	1.	0.	0.		92.	92.
P:20000AB0--20000ACB	func14	ok	100.000%	0.	0.	0.		28.	28.
P:20000ACC--20000AE9	func15	ok	100.000%	0.	0.	0.		30.	30.
P:20000AEC--20000B05	func16	ok	100.000%	0.	0.	0.		26.	26.
P:20000B08--20000B2F	func17	ok	100.000%	0.	0.	0.		40.	40.
P:20000B30--20000B51	func18	ok	100.000%	0.	0.	0.		34.	34.
P:20000B54--20000B75	func19	ok	100.000%	0.	0.	0.		34.	34.
P:20000B78--20000BA9	func20	ok	100.000%	0.	0.	0.		50.	50.
P:20000BAC--20000BD9	func21	ok	100.000%	0.	0.	0.		46.	46.
P:20000BDC--20000C09	func22	ok	100.000%	0.	0.	0.		46.	46.
P:20000C0C--20000C39	func23	ok	100.000%	0.	0.	0.		46.	46.
P:20000C3C--20000C4B	func24	ok	100.000%	0.	0.	0.		16.	16.
P:20000C4C--20000C5F	func25	ok	100.000%	0.	0.	0.		20.	20.
P:20000C60--20000C73	func26	ok	100.000%	0.	0.	0.		20.	20.
P:20000C74--20000CB9	func27	never	0.000%	0.	0.	0.		70.	0.
P:20000CBC--20000CC5	func40	ok	100.000%	0.	0.	0.		10.	10.
P:20000CC8--20000ECB	main	partial	98.837%	0.	0.	1.		516.	510.
P:20000ECC--20000F6F	sieve	ok	100.000%	4.	0.	0.		164.	164.

Figure T15: Function level code coverage

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

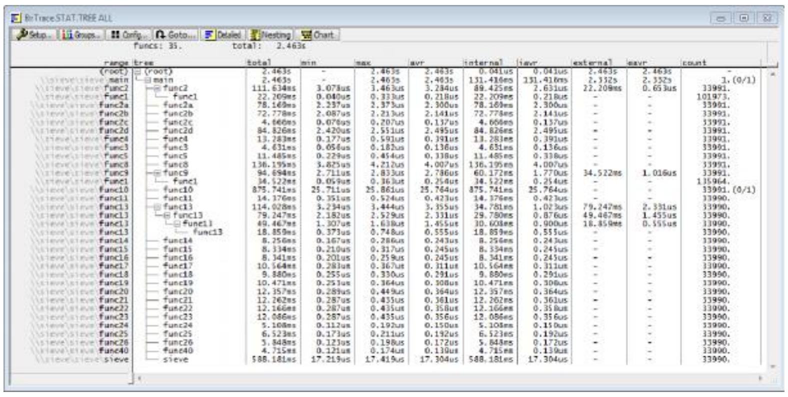


```
coverage       addr/line  code        label    mnemonic          comment
partial              438                  switch ( x )
      ok     ST:200009C4  687B                 ldr     r3,[r7,#0x4]
      ok     ST:200009C6  F10333FF             add     r3,r3,#-0x1
      ok     ST:200009CA  2B05                 cmp     r3,#0x5
not exec     ST:200009CC  D838                 bhi     0x20000A40
      ok     ST:200009CE  A201                 adr     r2,0x200009D4
      ok     ST:200009D0  F852F023             ldr     pc,[r2,r3,lsl #0x2]
   never     SR:200009D4  200009ED             dcd     0x200009ED
   never     SR:200009D8  20000A09             dcd     0x20000A09
   never     SR:200009DC  20000A13             dcd     0x20000A13
   never     SR:200009E0  20000A1B             dcd     0x20000A1B
   never     SR:200009E4  20000A41             dcd     0x20000A41
   never     SR:200009E8  20000A37             dcd     0x20000A37
                                          {
                                          case 1:
   never             441                           x = x+1;
   never     ST:200009EC  687B                 ldr     r3,[r7,#0x4]
   never     ST:200009EE  F1030301             add     r3,r3,#0x1
   never     ST:200009F2  607B                 str     r3,[r7,#0x4]
   never             442                           x = x*2;
   never     ST:200009F4  687B                 ldr     r3,[r7,#0x4]
   never     ST:200009F6  EA4F0343             lsl     r3,r3,#0x1
   never     ST:200009FA  607B                 str     r3,[r7,#0x4]
   never             443                           return x*x;
```

Figure T16: Low level code coverage

Performance Analysis

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



B::Trace.STAT.TREE ALL

funcs: 35.   total: 2.463s

range	tree	total	min	max	avr	internal	iavr	external	eavr	count
(root)	(root)	2.463s	-	2.463s	2.463s	0.041us	0.041us	2.463s	2.463s	-
\\sieve\sieve\main	main	2.463s	-	2.463s	2.463s	131.416ms	131.416ms	2.332s	2.332s	1.(0/1)
\\sieve\sieve\func2	func2	111.634ms	3.078us	3.463us	3.284us	89.425ms	2.631us	22.209ms	0.653us	33991.
\\sieve\sieve\func1	func1	22.209ms	0.040us	0.333us	0.218us	22.209ms	0.218us	-	-	101973.
\\sieve\sieve\func2a	func2a	78.169ms	2.237us	2.373us	2.300us	78.169ms	2.300us	-	-	33991.
\\sieve\sieve\func2b	func2b	72.778ms	2.087us	2.213us	2.141us	72.778ms	2.141us	-	-	33991.
\\sieve\sieve\func2c	func2c	4.666ms	0.070us	0.207us	0.137us	4.666ms	0.137us	-	-	33991.
\\sieve\sieve\func2d	func2d	84.826ms	2.420us	2.551us	2.495us	84.826ms	2.495us	-	-	33991.
\\sieve\sieve\func4	func4	13.283ms	0.177us	0.591us	0.391us	13.283ms	0.391us	-	-	33991.
\\sieve\sieve\func3	func3	4.631ms	0.056us	0.182us	0.136us	4.631ms	0.136us	-	-	33991.
\\sieve\sieve\func5	func5	11.485ms	0.229us	0.454us	0.338us	11.485ms	0.338us	-	-	33991.
\\sieve\sieve\func8	func8	136.195ms	3.825us	4.212us	4.007us	136.195ms	4.007us	-	-	33991.
\\sieve\sieve\func9	func9	94.694ms	2.711us	2.833us	2.786us	60.172ms	1.770us	34.522ms	1.016us	33991.
\\sieve\sieve\func1	func1	34.522ms	0.059us	0.363us	0.254us	34.522ms	0.254us	-	-	135964.
\\sieve\sieve\func10	func10	875.741ms	25.711us	25.861us	25.764us	875.741ms	25.764us	-	-	33991.(0/1)
\\sieve\sieve\func11	func11	14.376ms	0.351us	0.524us	0.423us	14.376ms	0.423us	-	-	33990.
\\sieve\sieve\func13	func13	114.028ms	3.234us	3.444us	3.355us	34.781ms	1.023us	79.247ms	2.331us	33990.
\\sieve\sieve\func13	func13	79.247ms	2.182us	2.529us	2.331us	29.780ms	0.876us	49.467ms	1.455us	33990.
\\sieve\sieve\func13	func11	49.467ms	1.307us	1.638us	1.455us	30.608ms	0.900us	18.859ms	0.555us	33990.
\\sieve\sieve\func13	func13	18.859ms	0.373us	0.748us	0.555us	18.859ms	0.555us	-	-	33990.
\\sieve\sieve\func14	func14	8.256ms	0.167us	0.286us	0.243us	8.256ms	0.243us	-	-	33990.
\\sieve\sieve\func15	func15	8.334ms	0.210us	0.317us	0.245us	8.334ms	0.245us	-	-	33990.
\\sieve\sieve\func16	func16	8.341ms	0.201us	0.259us	0.245us	8.341ms	0.245us	-	-	33990.
\\sieve\sieve\func17	func17	10.564ms	0.283us	0.367us	0.311us	10.564ms	0.311us	-	-	33990.
\\sieve\sieve\func18	func18	9.880ms	0.255us	0.330us	0.291us	9.880ms	0.291us	-	-	33990.
\\sieve\sieve\func19	func19	10.471ms	0.253us	0.364us	0.308us	10.471ms	0.308us	-	-	33990.
\\sieve\sieve\func20	func20	12.357ms	0.289us	0.449us	0.364us	12.357ms	0.364us	-	-	33990.
\\sieve\sieve\func21	func21	12.262ms	0.287us	0.435us	0.361us	12.262ms	0.361us	-	-	33990.
\\sieve\sieve\func22	func22	12.166ms	0.287us	0.435us	0.358us	12.166ms	0.358us	-	-	33990.
\\sieve\sieve\func23	func23	12.086ms	0.287us	0.435us	0.356us	12.086ms	0.356us	-	-	33990.
\\sieve\sieve\func24	func24	5.108ms	0.112us	0.192us	0.150us	5.108ms	0.150us	-	-	33990.
\\sieve\sieve\func25	func25	6.523ms	0.173us	0.211us	0.192us	6.523ms	0.192us	-	-	33990.
\\sieve\sieve\func26	func26	5.848ms	0.123us	0.198us	0.172us	5.848ms	0.172us	-	-	33990.
\\sieve\sieve\func40	func40	4.715ms	0.121us	0.174us	0.139us	4.715ms	0.139us	-	-	33990.
\\sieve\sieve\sieve	sieve	588.181ms	17.219us	17.419us	17.304us	588.181ms	17.304us	-	-	33990.

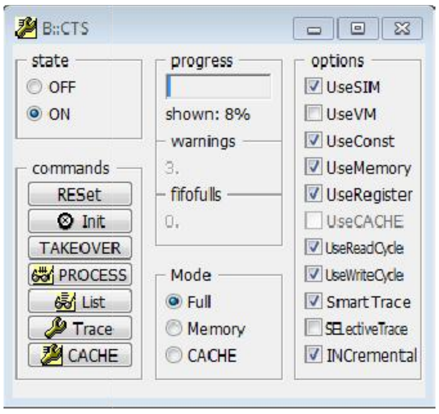
Figure T17: Detailed Performance Measurements

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

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

Trace Based Debugging

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



B::CTS

state: OFF [✓ ON]

progress: shown: 8%

warnings: 3.

fifofulls: 0.

Mode: [✓ Full] Memory CACHE

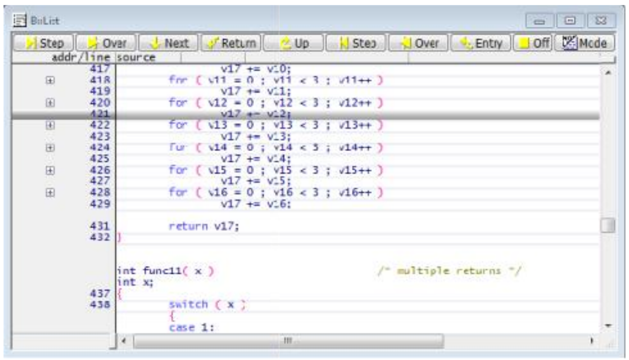
options: [✓ UseSIM] UseVM [✓ UseConst] [✓ UseMemory] [✓ UseRegister] UseCACHE [✓ UseReadCycle] [✓ UseWriteCycle] [✓ SmartTrace] SElectiveTrace [✓ INCremental]

Figure T18: CTS Settings

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

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

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



B::List

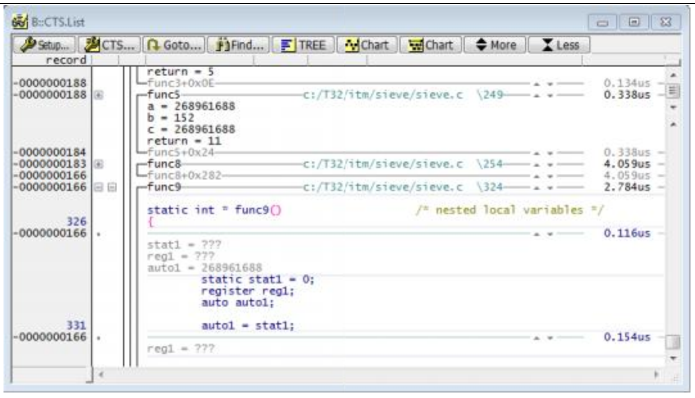
```
 addr/line source
  417                v17 += v10;
  418          for ( v11 = 0 ; v11 < 3 ; v11++ )
  419                v17 += v11;
  420          for ( v12 = 0 ; v12 < 3 ; v12++ )
  421                v17 += v12;
  422          for ( v13 = 0 ; v13 < 3 ; v13++ )
  423                v17 += v13;
  424          for ( v14 = 0 ; v14 < 3 ; v14++ )
  425                v17 += v14;
  426          for ( v15 = 0 ; v15 < 3 ; v15++ )
  427                v17 += v15;
  428          for ( v16 = 0 ; v16 < 3 ; v16++ )
  429                v17 += v16;

  431          return v17;
  432 }


      int func11( x )                        /* multiple returns */
      int x;
  437 {
  438          switch ( x )
               {
               case 1:
```

Figure T19: New Run Control Buttons

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

- Go Till
- Go Back Till

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



```
             return = 5
-0000000188  func3+0x0E                                              0.134us
-0000000188  func5          c:/T32/itm/sieve/sieve.c  \249           0.338us
             a = 268961688
             b = 152
             c = 268961688
             return = 11
-0000000184  func5+0x24                                              0.338us
-0000000183  func8          c:/T32/itm/sieve/sieve.c  \254           4.059us
-0000000166  func8+0x282                                             4.059us
-0000000166  func9          c:/T32/itm/sieve/sieve.c  \324           2.784us

             static int * func9()           /* nested local variables */
         326 {
-0000000166                                                          0.116us
             stat1 = ???
             reg1 = ???
             auto1 = 268961688
                     static stat1 = 0;
                     register reg1;
                     auto auto1;

         331         auto1 = stat1;
-0000000166                                                          0.154us
             reg1 = ???
```

Figure T20: CTS.List view

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

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

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