



LPCXpresso IDE SWO Trace Guide

Rev. 8.1 — 29 June, 2016








User guide



29 June, 2016

Copyright © 2013-2016 NXP Semiconductors

All rights reserved.

- 1. Trace Overview 1
 - 1.1. Serial Wire Output (SWO) 1
 - 1.2. SWO Trace : Views 2
 - 1.3. Starting SWO Trace 3
 - 1.3.1. Target Clock Speed 3
 - 1.3.2. Part specific configuration for SWO Trace 4
 - 1.3.3. SWO Config view  4
- 2. Profiling 5
 - 2.1. Overview 5
 - 2.2. SWO Profile view  5
- 3. ITM 7
 - 3.1. Overview 7
 - 3.2. Using the ITM to handle `printf` and `scanf` 7
 - 3.3. SWO ITM console view  7
 - 3.3.1. Toolbar 8
- 4. Interrupt tracing 9
 - 4.1. Overview 9
 - 4.2. SWO Interrupt Statistics view  9
 - 4.3. SWO Interrupt Trace view  10
 - 4.3.1. Zooming and panning 10
 - 4.3.2. Time axis 10
 - 4.3.3. Interrupt axis 11
 - 4.3.4. Interpretation 11
 - 4.3.5. Graph buffer depth 13
 - 4.4. SWO Interrupt Trace Table 13
 - 4.4.1. Columns 14
- 5. Data Watch Trace 16
 - 5.1. Overview 16
 - 5.2. SWO Data Watch view  16
 - 5.2.1. Item Display 18
 - 5.2.2. Trace Display 18
- 6. Performance Counters 19
 - 6.1. Overview 19
 - 6.2. SWO Performance Counters view  19
 - 6.3. Display features 20
- 7. Bandwidth considerations 21
 - 7.1. Overview 21
 - 7.2. SWO Stats View 21
- 8. Preferences 23
 - 8.1. Options 23
- 9. Appendix A – SWO Trace setup for LPC13xx 25
 - 9.1. To carry out SWO Trace on LPC13xx 25
 - 9.1.1. SWO pin and debug connector 25
 - 9.1.2. Enabling the trace clock 25
 - 9.1.3. SWO pinmux configuration 25
 - 9.1.4. Example setup code for LPC1315/16/46/47 parts 25
- 10. Appendix B – SWO Trace setup for LPC15xx 27
 - 10.1. To carry out SWO Trace on LPC15xx 27
 - 10.1.1. SWO pin and debug connector 27
 - 10.1.2. Enabling the trace clock 27
 - 10.1.3. SWO switch matrix configuration 27
- 11. Appendix C – SWO Trace setup for LPC5410x 29
 - 11.1. To carry out SWO Trace on LPC5410x 29
 - 11.1.1. SWO pin and debug connector 29

11.1.2. Debug probe	29
11.1.3. Enabling the trace clock	29
11.1.4. SWO pinmux configuration	30

1. Trace Overview

There are two different kinds of tracing technologies used within the LPCXpresso IDE. The trace functionality available depends on the features supported by your target and the features supported by your debug probe.

The forms of trace functionality supported by LPCXpresso IDE are:

- **Instruction Trace** - using on chip trace buffers
- **SWO Trace** - SWO using LPC-Link2 debug probe via CMSIS-DAP

Note that previous version of LPCXpresso IDE supported SWO trace via Redprobe+ (known as RedTrace), this feature is no longer present.

The latest information on trace and details about supported targets can be found at <https://community.nxp.com/message/630730>

The rest of this guide looks at SWO Trace, for information regarding instruction trace, see the Instruction Trace guide.

Note that SWO Trace may sometimes be referred to as SWV Trace (Serial Wire Viewer). Even the ARM documentation uses the two terms interchangeably.

1.1 Serial Wire Output (SWO)

ARM's Coresight debug architecture allows data to be sampled and streamed from the MCU to the host completely non-intrusively. This scheme allows events such as periodic PC sampling, interrupts etc. to be captured and transmitted by the debug probe **with no affect on MCU performance** and without the need for any code instrumentation or changes.

The Serial Wire Output (SWO) tools provide access to the memory of a running target and facilitate trace without needing to interrupt the target. Support for SWO is provided by all Cortex-M3 and M4 based MCUs. It requires just one extra pin in addition to the standard Serial Wire Debug (SWD) connection.



Note

Use of SWO requires connection to the target system using a compatible debug probe, currently LPC-Link2 with NXP CMSIS-DAP firmware. Cortex-M0 and Cortex-M0+ parts do not have SWO capabilities.



Note

The debug probe must be configured to connect using SWD not JTAG. This requirement is a feature of the ARM hardware.

LPCXpresso IDE presents target information collected using SWO from a Cortex-M3/M4 based system in a number of different views.

Table 1.1. SWO trace feature

Feature	SWO Trace
Data watch	yes*
Profile	yes
Interrupt Statistics	yes
Graphical Interrupt trace	Pro
ITM text console	yes

(*)Note that the LPC-Link2 SWO trace only allows one expression to be traced in the Free version of LPCXpresso. LPCXpresso Pro allows as many expressions to be traced as the hardware allows (typically four).

Items marked “Pro” are only available in the LPCXpresso Pro version.

1.2 SWO Trace : Views

The latest LPC-Link2 CMSIS-DAP firmware now provides access to the SWO trace information. Users can configure SWO trace and view the collected data in a set of views.

SWO Config

- Provides quick access to the SWO Trace views and shows the status of the SWO Trace connection.

SWO Profile

- Provides a statistical profile of application activity.

SWO ITM Console

- A debug console for reading and writing text to and from your application.

SWO Int Stats

- Provides counts and timing information for interrupts and interrupt handlers.

SWO Int Trace

- Plots a time line trace of interrupts.

SWO Int Table

- Lists the raw entry, exit and return SWO events for interrupt handlers.

SWO Data

- Provides the ability to monitor (and update) upto 4 memory location in real-time, without stopping the CPU.

SWO Counters

- Displays the target's performance counters.

SWO Stats

- Displays trace channel bandwidth usage and low level debug information related to the SWO Trace connection.

The **SWO Config** view is presented within the **Debug Perspective** or the **Develop Perspective** by default. This view can be used to easily show and hide the other SWO Trace views. Trace views that are not required may be closed to simplify the user interface. They may also be opened using the **Window -> Show View -> Other...** menu item.

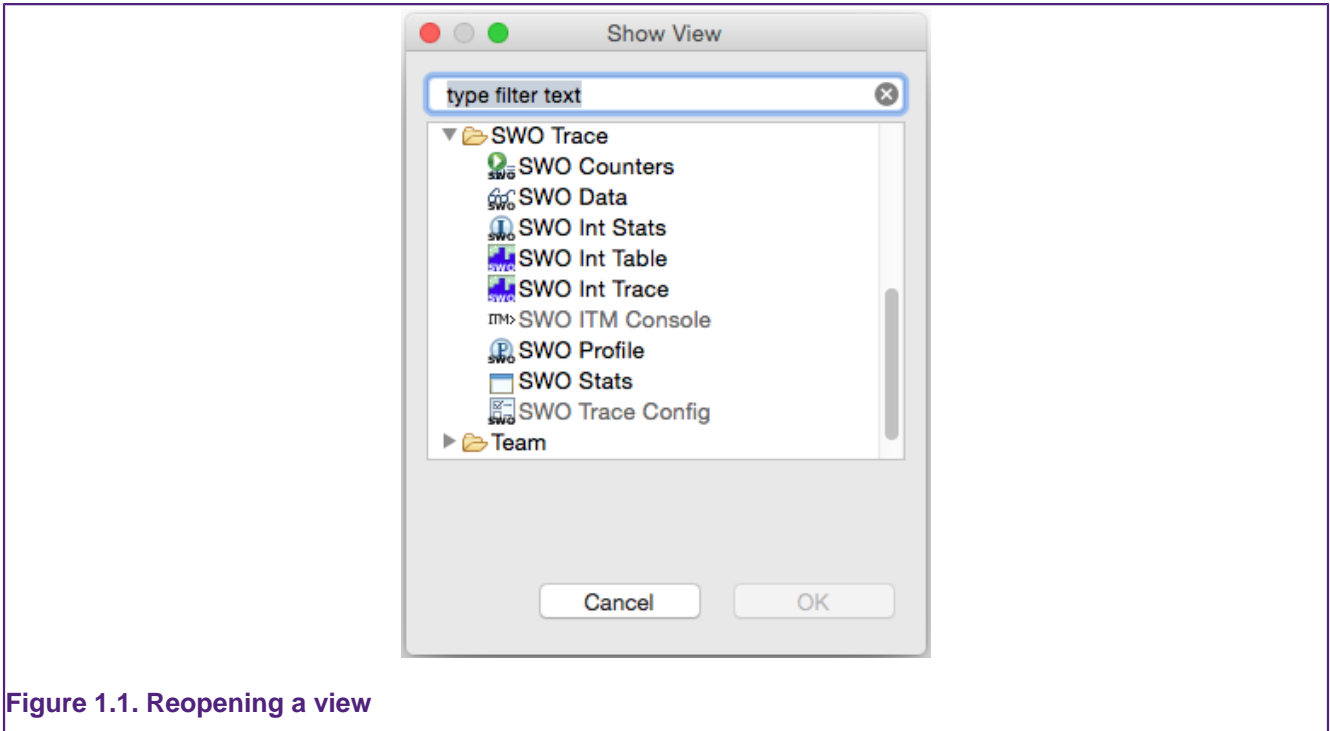






Figure 1.1. Reopening a view

1.3 Starting SWO Trace

To use SWO Trace's features, you must be debugging an application on a Cortex-M3/M4 based MCU, connected via a supported LPC-Link2 debug probe using the SWD protocol.

You may start SWO Trace at any time while debugging your program. The program does **not** have to be stopped at a breakpoint or paused. Before the collection of data commences, SWO Trace may prompt you to enter the target clock speed.

Trace collection for each view is controlled by the buttons in its toolbar.

-  - starts trace collection for that view
-  - stops trace collection for that view
-  - deletes the collected trace from the view
-  - opens the **SWO Config** view

More than one view may be configured to collect trace at a time. However, the bandwidth of the trace channel is limited and may become saturated resulting in data loss. Try disabling other SWO Trace views if you do not see trace data coming through. The **SWO Stats** view provides an overview of the SWO channel load.

1.3.1 Target Clock Speed

Due to the way the Trace data is transferred by the target processor, setting the correct clock speed within the SWO Trace interface is essential to determine the correct baud rate for the data transfer. If the clock speed setting does not match the actual clock speed of the processor then data will be lost and or corrupted. This can result in no data being visualized or unexpected trace data.

The first time trace is used in a project you will be asked to enter the target clock speed. SWO Trace attempts to read the clock speed from the SystemCoreClock global variable

and will suggest that value if found. The SystemCoreClock is usually set to the *current* clock speed of the target. Care needs to be taken to ensure that it is used after the application has set the clock speed for normal operation, otherwise it may provide an inappropriate value. If that variable does not exist or has not been set to the core clock frequency you will need to manually enter the clock frequency.

Once set the target core clock speed is saved in the project configuration. The saved value can be viewed and changed from the **SWO Config** view.

1.3.2 Part specific configuration for SWO Trace



Most parts should not need any special configuration to use SWO trace. However, some may require additional configuration to enable SWO output.

The following parts require extra configuration:

- LPC13xx
- LPC15xx
- LPC5410x

Further details for each part can be found in the Appendix. Latest information is available from <https://community.nxp.com/message/630730>

1.3.3 SWO Config view

The **SWO Config** view displays the state of the SWO Trace system. From here you can open and close the other SWO Views using the show view  buttons and hide view  buttons for each SWO component.



Note

The clock speed can be changed from the **SWO Config** view. If the clock speed is entered incorrectly you may see unexpected trace data or no trace data.

The status of the SWO connection is also displayed. When SWO trace is correctly configured all lights will be green.

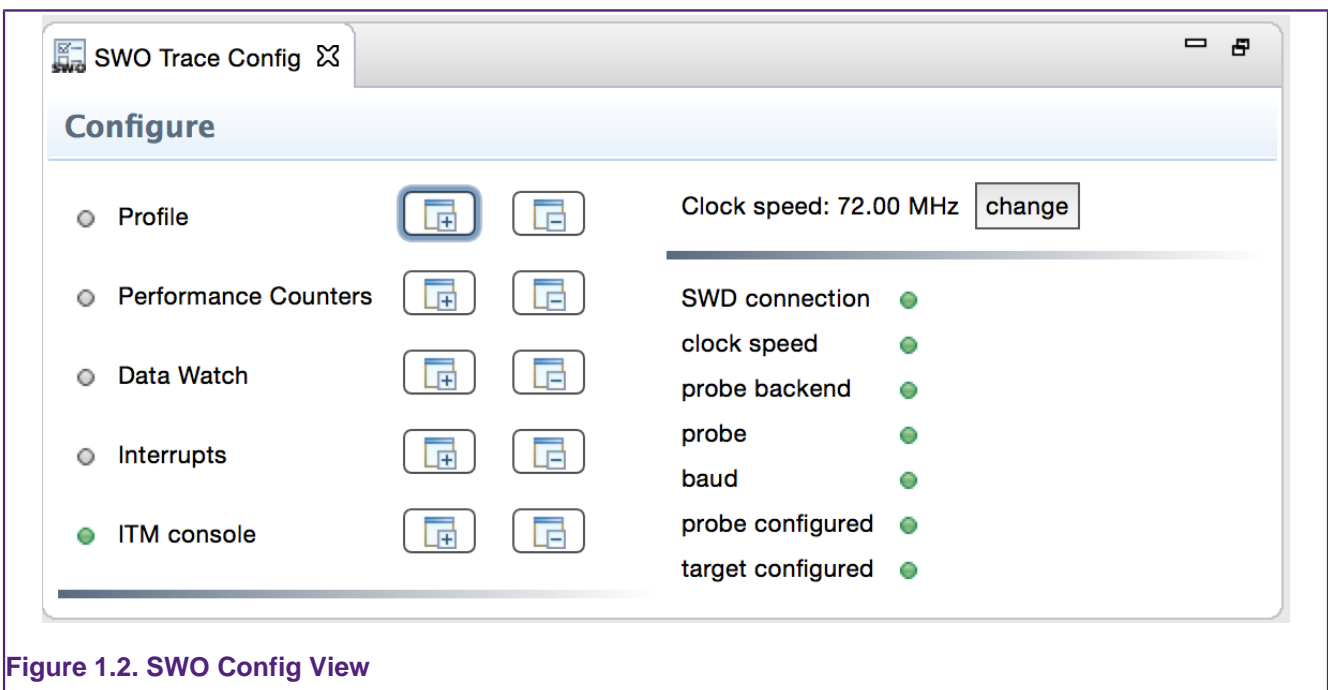


Figure 1.2. SWO Config View

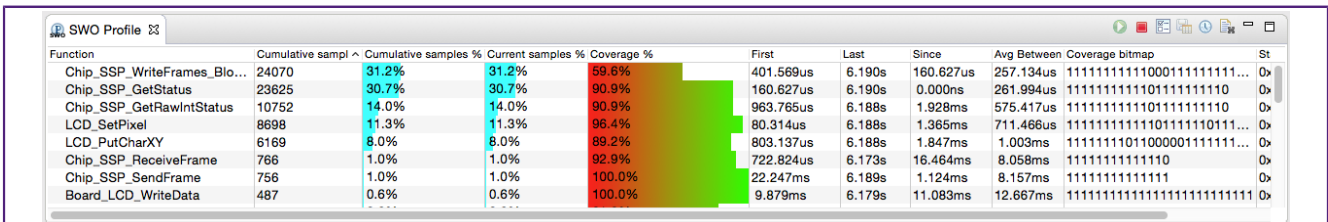
2. Profiling

2.1 Overview

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

2.2 SWO Profile view


The Profile view shows a profile of the code as it is running, providing a breakdown of time spent in different functions. An example screenshot is shown in Figure 2.1. Double clicking on a row will jump to the corresponding function definition in the source code. Clicking on a column title will sort by that column. Clicking a second time will reverse the sorting order.

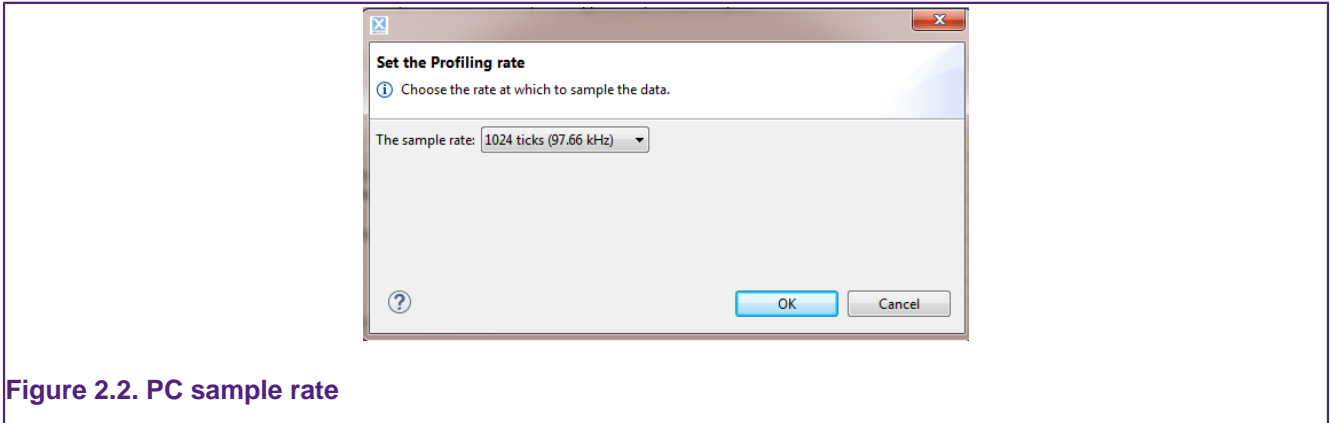


Function	Cumulative sampl	Cumulative samples %	Current samples %	Coverage %	First	Last	Since	Avg Between	Coverage bitmap	St
Chip_SSP_WriteFrames_Blo...	24070	31.2%	31.2%	59.6%	401.569us	6.190s	160.627us	257.134us	1111111111000111111111...	0x
Chip_SSP_GetStatus	23625	30.7%	30.7%	90.9%	160.627us	6.190s	0.000ms	261.994us	111111111101111111110	0x
Chip_SSP_GetRawIntStatus	10752	14.0%	14.0%	90.9%	963.765us	6.188s	1.928ms	575.417us	111111111101111111110	0x
LCD_SetPixel	8698	11.3%	11.3%	98.4%	80.314us	6.188s	1.365ms	711.466us	111111111101111110111...	0x
LCD_PutCharXY	6169	8.0%	8.0%	89.2%	803.137us	6.188s	1.847ms	1.003ms	1111111011000001111111...	0x
Chip_SSP_ReceiveFrame	766	1.0%	1.0%	92.9%	722.824us	6.173s	16.464ms	8.058ms	1111111111110	0x
Chip_SSP_SendFrame	756	1.0%	1.0%	100.0%	22.247ms	6.189s	1.124ms	8.157ms	1111111111111	0x
Board_LCD_WriteData	487	0.6%	0.6%	100.0%	9.879ms	6.179s	11.083ms	12.667ms	111111111111111111111111	0x

Figure 2.1. SWO Profile View

- **Cumulative samples:** This is the total number of PC samples that have occurred while executing the particular function. A relatively high number of samples indicates a larger function or more frequently executed functions.
- **Cumulative samples (%):** This is the same as above, but displayed as a percentage of total PC samples collected.
- **Current samples (%):** This column is a legacy item. In SWO trace it is identical to the Cumulative samples(%).
- **Coverage (%):** Percentage of instructions in the function that have been seen to have been executed.
- **Coverage Bitmap:** the coverage bitmap has 1 bit for each half-word in the function. The bit corresponding to the *address* of each sampled PC is set. Most Cortex-M instructions are 16-bits (one half-word) in length. However, there are some instructions that are 32-bits (two half-words). The bit corresponding to the second halfword of a 32-bit instruction will *never* be set.
- **First:** This is the first time (relative to the start time of tracing) that the function was sampled.
- **Last:** This is the last time (relative to the start time of tracing) that the function was sampled.
- **Since:** It is this long since you last saw this function. (current – last)
- **Avg Between:** This is the average time between executions of this function.

The PC sampling rate is configured using the **Rate button** . The rate can be configured to be from one in every 64 instructions to one in every 16384 instructions. Note however that at the higher sample rates the SWO channel will be overwhelmed resulting in data loss. See Figure 2.2.



The summary of the PC count by function depends on being able to map a PC sample to a function. The function name can only be determined if the source code is available (this may not be the case for library code or ROM code for example). If code is dynamically loaded the profile will be inaccurate as samples may get attributed to the wrong functions.

**Note:**

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

3. ITM

3.1 Overview

The ITM block provides a mechanism for sending data from your target to the debugger via the SWO stream. This communication is achieved through a memory-mapped register interface. Data written to any of 32 stimulus registers is forwarded to the SWO stream. Unlike other SWO functionality, using the ITM stimulus ports requires changes to your code and so should not be considered non-intrusive.

3.2 Using the ITM to handle `printf` and `scanf`

The ITM stimulus registers facilitate `printf` style debugging. LPCXpresso uses the CMSIS standard scheme of treating any data written to stimulus port 0 (0xE0000000) as character data. A minor addition to your project can redirect the output of `printf` to this port.

A special global variable is used to achieve `scanf` functionality, which allows the debugger to send characters from the console to the target. The debugger writes data to the global variable named `ITM_RxBuffer` to be picked up by `scanf`.

To use this functionality with an LPC Open project please visit the FAQ 'How to use ITM Printf' at <https://community.nxp.com/message/630624>.

3.3 SWO ITM console view ^{ITM>}

Data written to the ITM stimulus port 0 is presented in this view. An example screenshot is shown in Figure 3.1. The view shows the ITM console for the active debug session. Once the target is terminated the view is cleared.

Text entered into this console is sent to the target if a suitable receiving buffer exists (specifically the global `int32_t ITM_RxBuffer`).

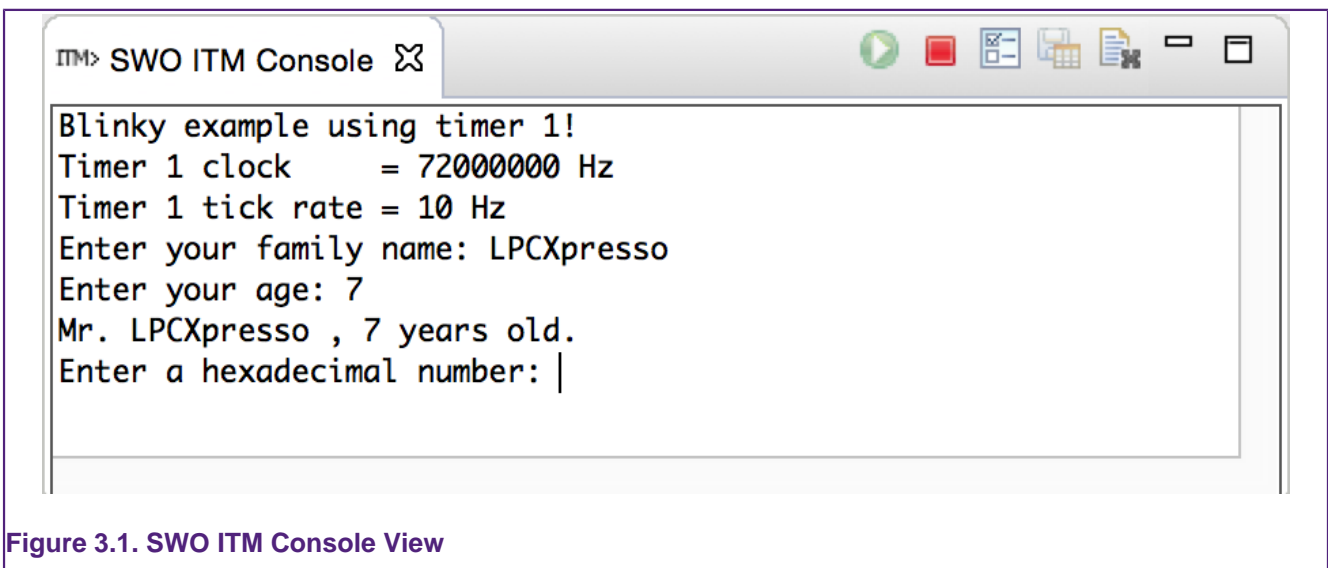


Figure 3.1. SWO ITM Console View

In addition to the standalone ITM Console view, the ITM console is also displayed as part of the standard console viewer, see Figure 3.1. It can be displayed by selecting the "Display Selected Console" button and choosing the console named "<your project> **ITM Console**". This view persists after the target is terminated, unlike the standalone ITM console view. Note that the standard console viewer switches automatically between consoles to show

consoles that are being written to. This switching can be disorienting, as the ITM console is easily lost among the other consoles displayed there. It is easier to keep track of the standalone ITM console.

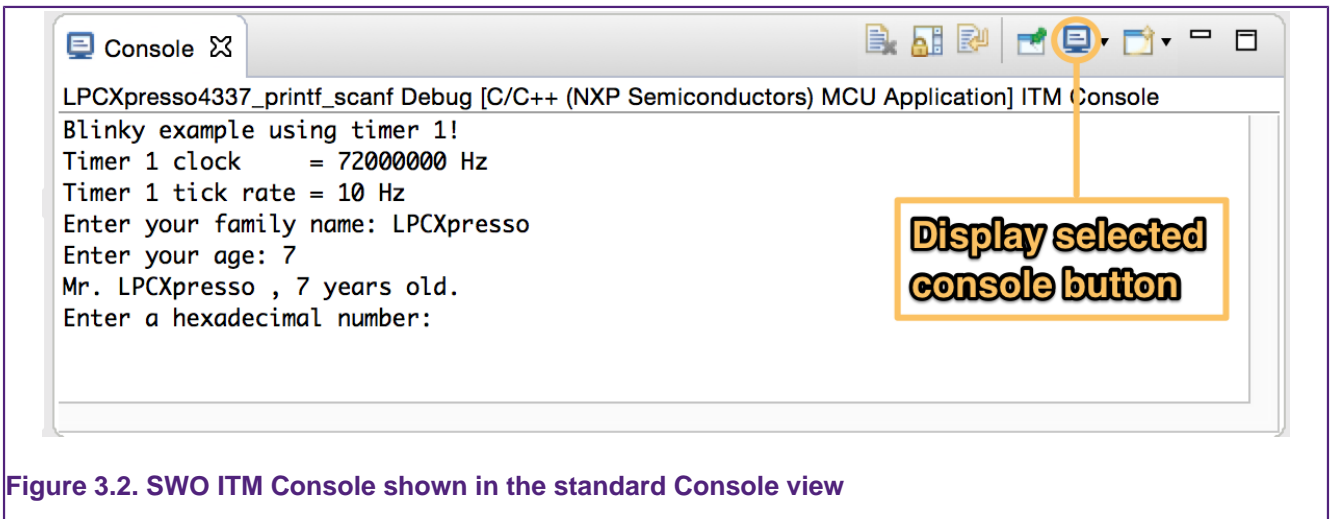






Figure 3.2. SWO ITM Console shown in the standard Console view

3.3.1 Toolbar

-  enable stimulus port 0 and start collecting data.
-  disable stimulus port 0.
-  switch to the SWO config view.
-  clear the ITM data and console.

The start and stop buttons in the ITM Console View enable and disable stimulus port 0.

4. Interrupt tracing

4.1 Overview

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

SWO interrupt tracing provides precise timing for entry and exits of interrupt handlers without any code instrumentation or processor overhead. The SWO stream reports three different events and the times at which they occurred:

- Entry to *i* - when the handler for interrupt *i* is initially executed
- Exit from *i* - when the handler *i* finishes or is preempted by an interrupt of higher priority
- Return to *i* - when the handler for interrupt *i* is returned to after being preempted.

4.2 SWO Interrupt Statistics view

The Interrupt Statistics view displays counts and aggregated timing information for interrupt handlers. An example screenshot is shown in Figure 4.1.

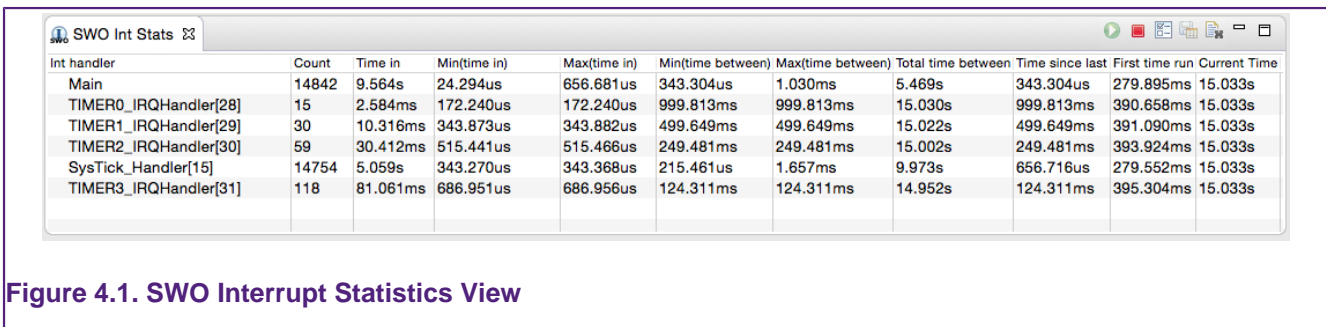


Figure 4.1. SWO Interrupt Statistics View

Information displayed includes:

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

Also see the **Overhead** and **Sleep** performance counters.

4.3 SWO Interrupt Trace view

The Interrupt Trace view plots a time line showing the entry, exit and return events of interrupt handlers. It is useful for debugging exception priorities and seeing how interrupts may be interacting in your running code.



Note

To see the chart in Linux you may need to install addition dependencies to enable the SWT Browser widget. See here for more information. For example in Ubuntu run `sudo apt-get install -y libwebkitgtk-1.0-0`

An example screenshot is shown in Figure 4.2.

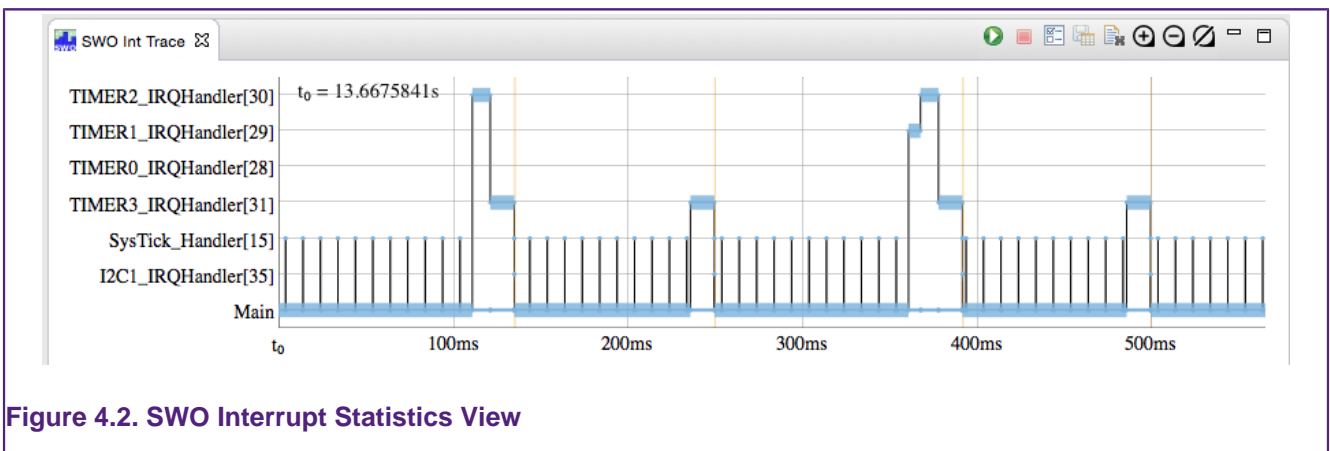





Figure 4.2. SWO Interrupt Statistics View

4.3.1 Zooming and panning

You can interact with the chart using the mouse by clicking and dragging inside the chart:

- Zoom in on time axis: click and drag horizontally to highlight the region to zoom into
- Zoom out to see all data: double click in graph
- Pan left and right: hold down **shift** as you click and drag on the graph

Additionally, there are the buttons in the toolbar to change the time scale:

-  2x zoom in
-  2x zoom out
-  reset time scale to show all data

4.3.2 Time axis

Time is plotted along the x-axis at the bottom of the graph. It is labeled relative to the origin of the chart t_0 . The value of t_0 is plotted in the top left of the chart.

When the chart is panned, by pressing shift and dragging the chart using the mouse, the grid lines may not appear to move but you should see t_0 changing as the plot moves along.

4.3.3 Interrupt axis

Each observed interrupt is listed on the vertical axis. The labels consist of the name of the handler with the number of the interrupt prepended to it. For example `SysTick_Handler[15]` refers to interrupt 15, which is handled by the function `SysTick_Handler`. The special entry `Main` represents all code executed outside of the Handler mode.

4.3.4 Interpretation

Each row in the chart represents the execution state of the labeled interrupt handler. No horizontal line indicates that the interrupt handler is not being executed.

Executing an interrupt from outside handler mode

Figure 4.3 shows an example of a SysTick handler that increments an integer and returns. The figure has been labeled to show how the SWO events are represented.

- Initial state
 - Initially the processor is executing user code, as indicated by the thick line in the far left of the `Main` row.
- Entry into 15 (`SysTick_Handler`)
 - The SysTick interrupt handler is entered at time $2s+50ns$ and an ENTRY event is sent over the SWO with the corresponding time stamp. This entry is represented in the chart by a vertical black line connecting the previously executing code's row (the `Main` row in this case) to the newly entered handler's row (the `SysTick_Handler` row).
 - The execution of `SysTick_Handler` is represented by the thick blue line.
 - The thin blue line in the `Main` row represents that `Main`'s execution is suspended, but not completed.
- Exit from 15 (`SysTick_Handler`)
 - When `SysTick_Handler` completes at $2s+125ns$ an EXIT event is generated with a corresponding time stamp.
 - The EXIT event is represented by the end of the thick blue line.
 - After exiting there is an overhead before the execution of `Main` can resume, this is represented by a thick gray line in the row of the handler which has just exited.
 - **Note** *This example was chosen to clearly show the overhead. The small code size of handler makes the overhead seem relatively large. The overhead is only 7 cycles here though.*
- Return to `Main`
 - When execution of code in `Main` begins at $2s + 160ns$ a RETURN event is generated with a corresponding time stamp.
 - A thin black vertical line connects the handler which is being returned from to the one now executing.
 - A thick horizontal line in the `Main` row shows that it is being executed.

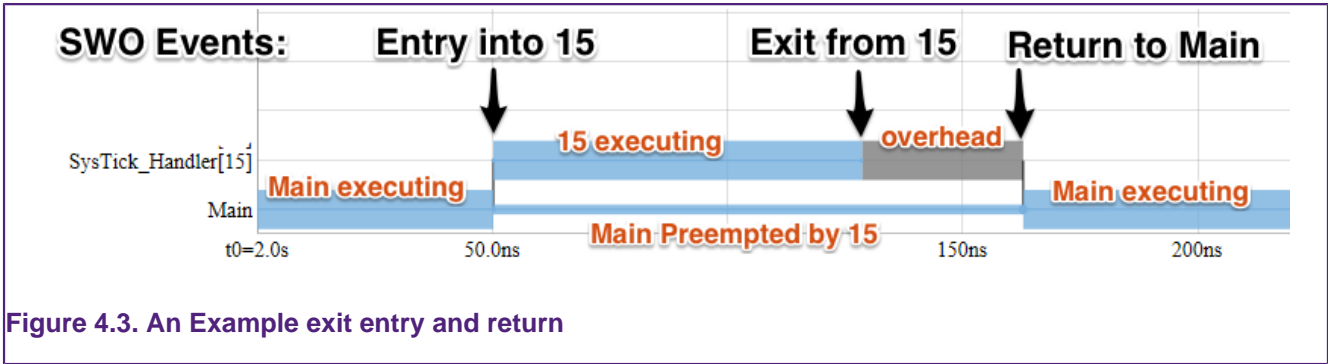


Figure 4.3. An Example exit entry and return

Tail chaining

Figure 4.4(a) shows an example of tail chaining, where one interrupt is exited and another entered without returning to Main. Figure 4.4(b) shows a zoomed in view of the first tail chaining allowing the overhead to be visible. The figure has been labeled to show how the SWO events are represented.

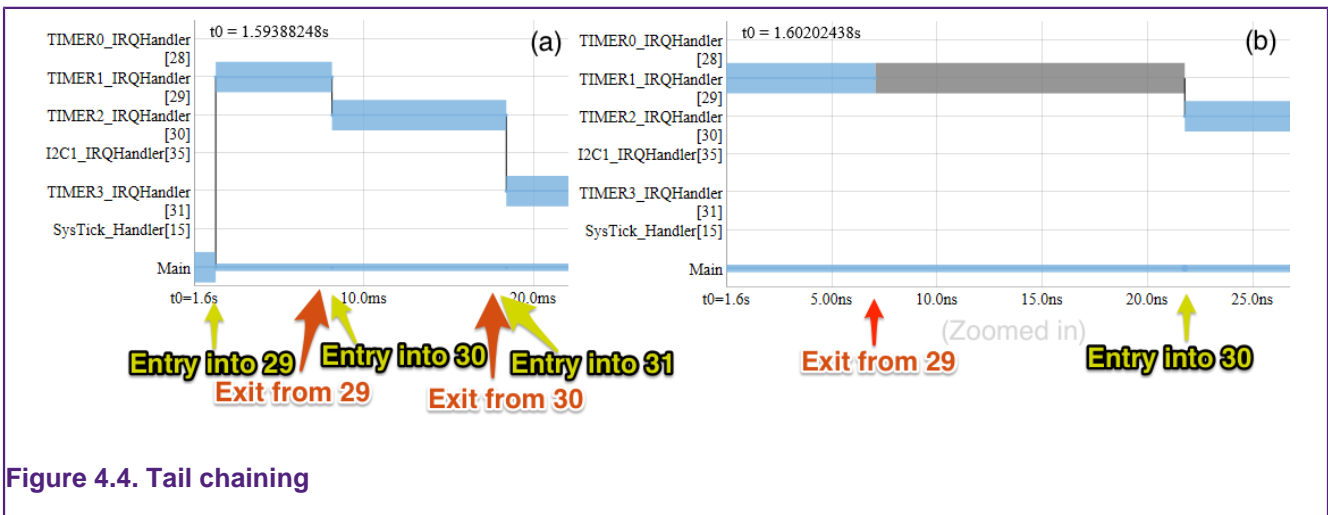


Figure 4.4. Tail chaining

Buffer breaks

When data is not collected from the debug probe by the IDE as quickly as it is being generated, some data is lost. Typically this would happen when data collection were paused for a while and then resumed, or when there is an unusually high amount of processing being performed by the IDE. Lost buffers are represented in the chart by an orange block – see Figure 4.5(a).

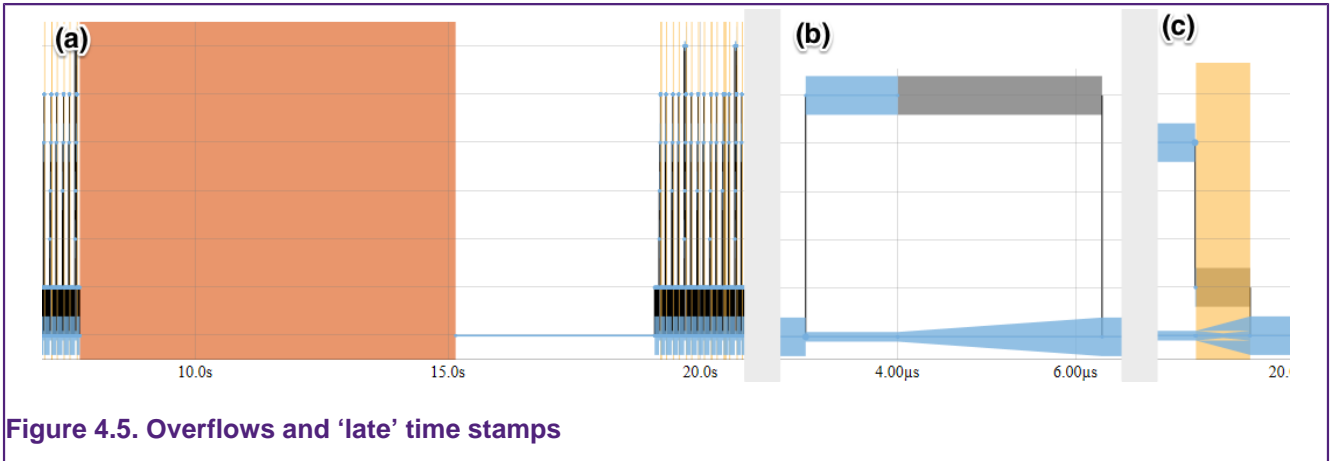


Figure 4.5. Overflows and 'late' time stamps

Time ranges

Usually an event has a precise time stamp associated with it. In some case the debug circuit may be unable to generate a matching time stamp. In these cases it may generate a 'late' time stamp corresponding to a time after the last event occurs. This 'late' time stamp tells us that the event happened occurred between the previous time stamp and the 'late' time stamp. We represent this in the chart as a tapered line.

Figure 4.5(b) show a representation of a 'late' time stamp. The ENTRY event into and EXIT event from the handler have precise time stamps. The return event has a 'late' time stamp. Since the EXIT's time stamp is at 4us and the 'late' time stamp is just after 6us, the chart show's `Main`'s execution state, as represented by the thickness of the line in the bottom row, increase from the preempted-state-thickness, to the executing-state-thickness between 4us and 6us.

FIFO overflows

The debug circuit on the target formats events into packets that drain via a FIFO over the SWO line. If too many events happen close together, this FIFO can overflow resulting in data loss. When this occurs, the overflow is represented as a yellow block on the chart – see Figure 4.5(c).

In addition to the overflow in Figure 4.5(c) we see the effect of two 'late' time stamped events. Two handlers are tail chained (the second handler is much shorter than the others and looks just like a dot). Both the EXIT of the second handler and RETURN into the `Main` at the bottom are associated with 'late' packets. This means that the preemption of `Main` ends within the range and so tapers, becoming smaller. Similarly, the re-entry into `Main` is represented by an increasing tapering from the preemption-width to the executing-width.

4.3.5 Graph buffer depth

The buffer depth can be configured via the **Interrupt Trace Graph Buffer depth** option in Preferences -> LPCXpresso -> SWO Trace. The default depth is 5000 and corresponds to the number of start and ends for each handler. Note that one handler's buffer may fill up before another.

large buffer depths may result in the UI becoming very slow when plotting all points. This limitation will be addressed in later releases.

4.4 SWO Interrupt Trace Table

The SWO Interrupt Trace Table view shows the collected interrupt events and their corresponding time stamps – see Figure 4.6.

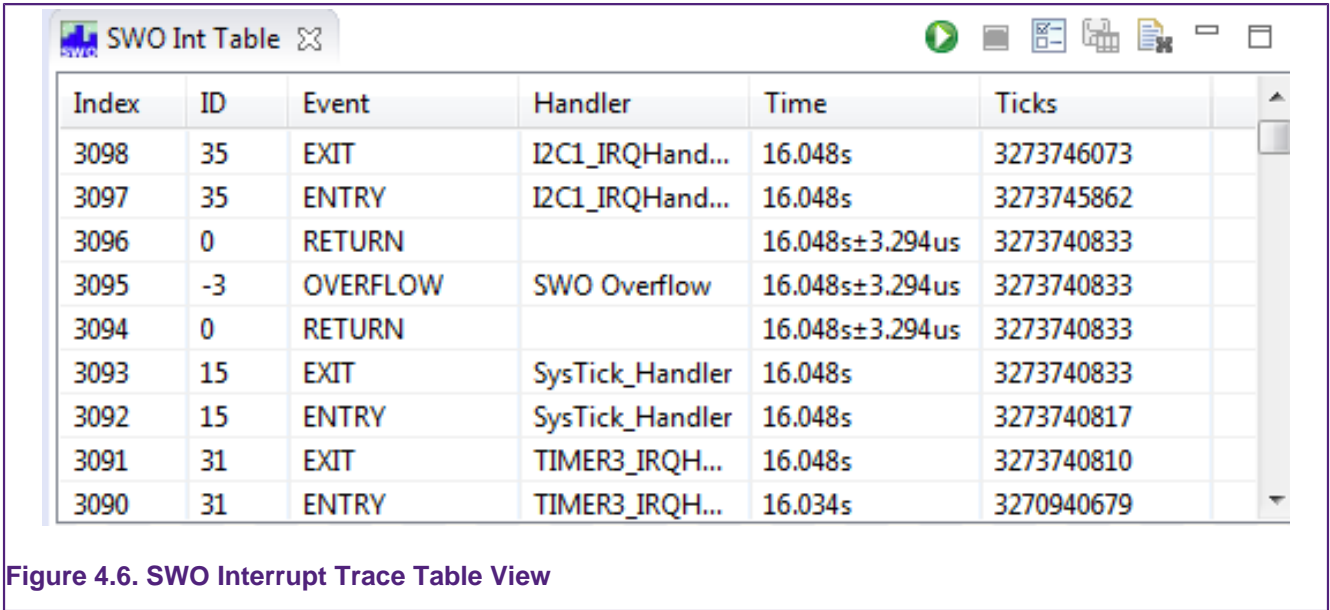


Figure 4.6. SWO Interrupt Trace Table View

4.4.1 Columns

- Index
 - Sequential ID for each event
- ID
 - Interrupt handler ID if greater than zero; or:
 - **0** - Out of Handler mode (i.e. normal execution)
 - **-1** - Unknown ID
 - **-2** - Inconsistent state (data corruption)
 - **-3** - SWO packet formatter FIFO overflow
 - **-4** - Break in data stream (dropped buffer(s))
- Event
 - **ENTRY** - execution of the associated interrupt handler begins
 - **EXIT** - execution of the associated interrupt handler ends
 - **RETURN** - re-entry into executing handler after preemption
 - **OVERFLOW** - data lost (see ID code for more info)
- Handler
 - The assigned interrupt handler or the interrupt handler ID if no source for the handler can be located.
- Time
 - The time stamp associated with the event expressed in seconds
 - May be a range – if so it will have a +/- term

- Ticks
 - The time stamp expressed as a number of CPU clock ticks.
 - For time stamps representing a range this is the start of the range.

5. Data Watch Trace

5.1 Overview

This view provides the ability to monitor (and update) any memory location in real-time, without stopping the processor. In addition up to 4 memory locations can also be traced in LPCXpresso Pro or 1 in LPCXpresso Free edition, allowing all accesses to be captured. Information that can be collected includes whether data is read or written, the value that is accessed and the PC of the instruction causing the access.

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

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

5.2 SWO Data Watch view

The view is split into 2 sections – with the **item display** on the left and the **trace display** on the right, as per Figure 5.1.

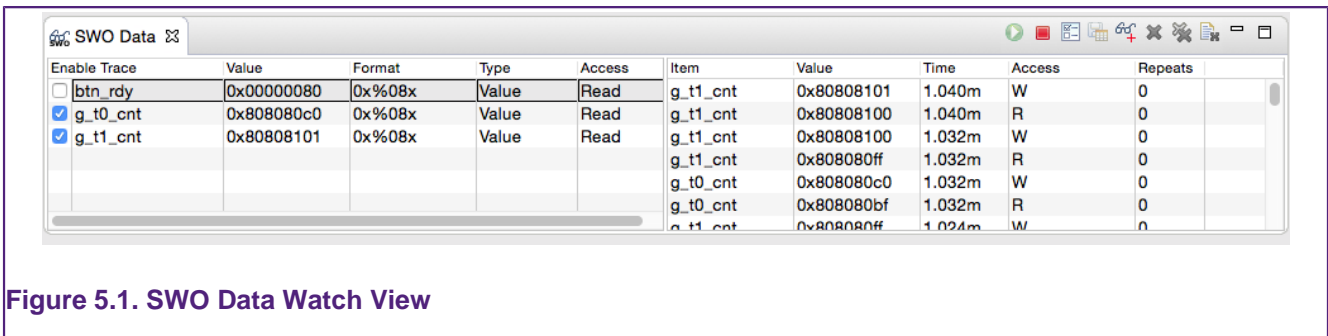



Figure 5.1. SWO Data Watch View

Use the **Add Data Watch Items** button  to display a dialog to allow the memory locations that will be presented to be chosen, as per Figure 5.2.

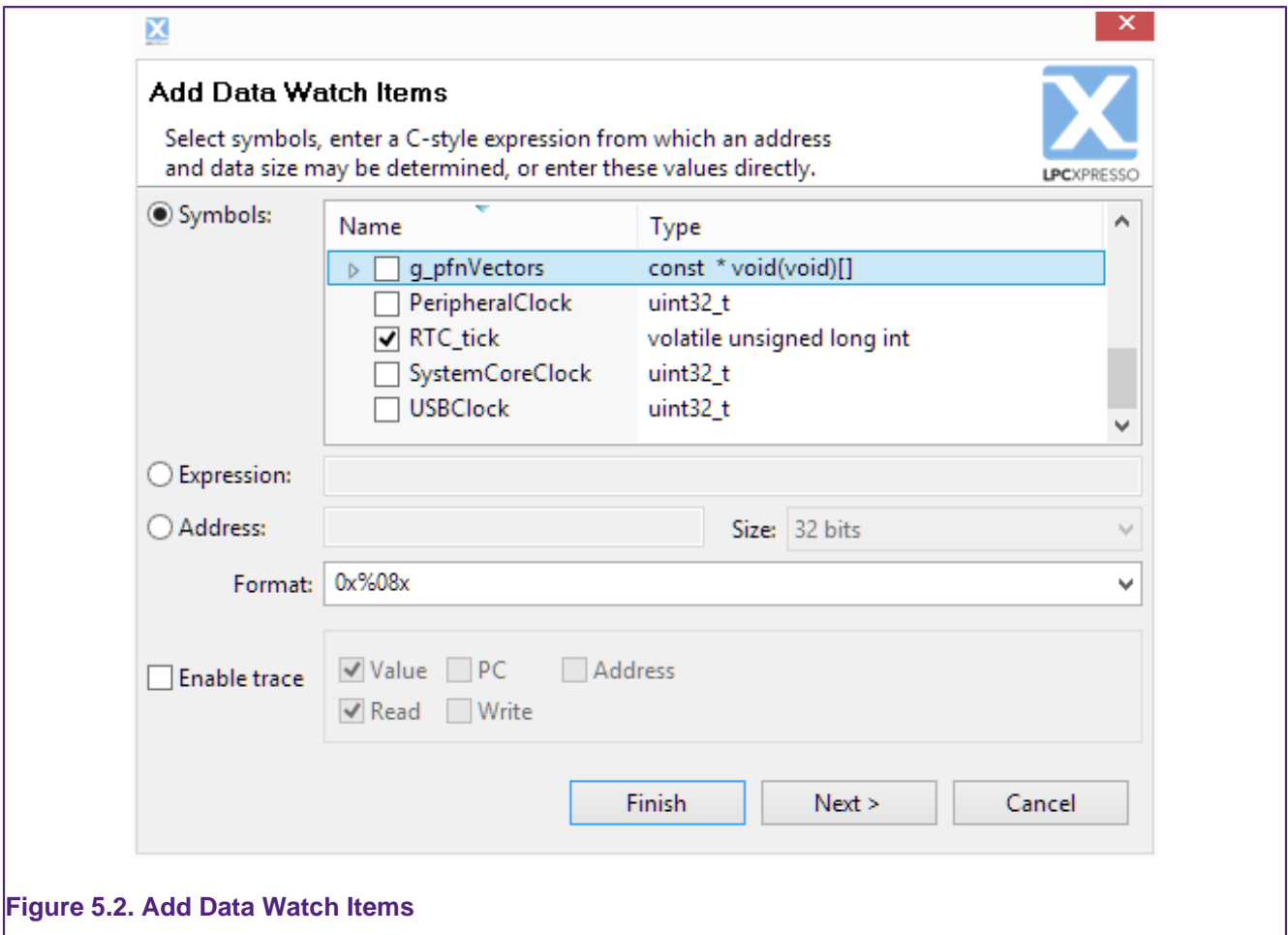


Figure 5.2. Add Data Watch Items

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


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

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

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



Note:

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

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

5.2.1 Item Display

As shown in Figure 5.1, the item display lists the data watch items that have been added. The following information is presented:

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

5.2.2 Trace Display

The trace display shows the traced values of the memory locations.

6. Performance Counters

6.1 Overview

There are several counters available in the Cortex-M3 and Cortex-M4 processors to help analyze the performance of the target application.

6.2 SWO Performance Counters view

The Performance Counter view displays the target's Performance Counters as shown in Figure 6.1.

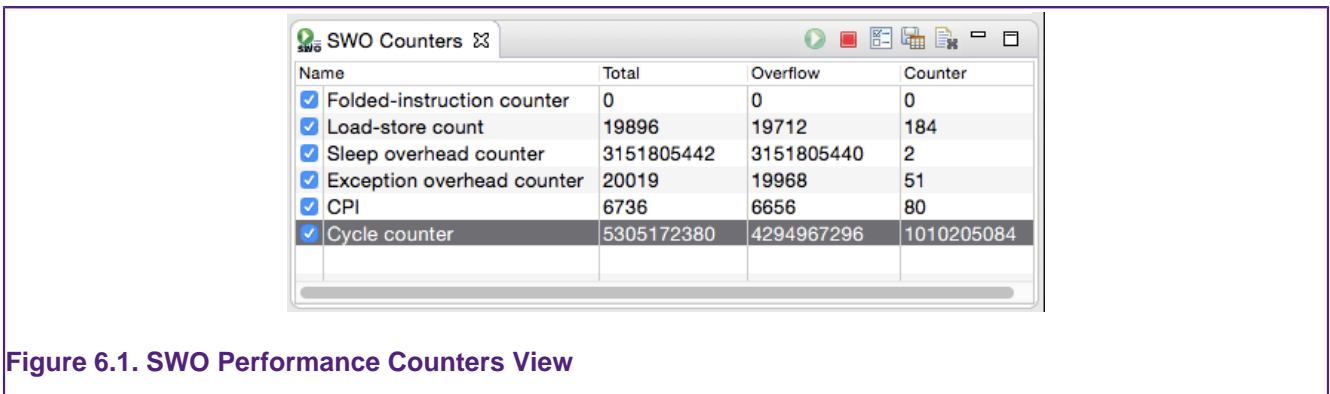


Figure 6.1. SWO Performance Counters View

The performance counters are described here as follows:

- **Cycle counter (CYCCNT)**
 - Increments on each clock cycle when the processor is not halted in debug state.
- **Folded Instruction Counter (FOLDCNT)**
 - Cycles saved by instructions which execute in zero cycles.
- **Load-Store Counter (LSUCNT)**
 - increments on each additional cycle required to execute a multi-cycle load-store instruction. It does not count the first cycle required to execute any instruction.
- **Sleep Overhead Counter (SLEEPcnt)**
 - increments on each cycle associated with power saving, whether initiated by a WFI or WFE instruction, or by the sleep-on-exit functionality.
- **Exception Overhead Counter (EXCCNT)**
 - increments on each cycle associated with exception entry or return. That is, it counts the cycles associated with entry stacking, return unstacking, preemption, and other exception-related processes.
- **CPI (CPIcnt)**
 - increments on each additional cycle required to execute a multi-cycle instruction, except for those instructions recorded by Load-store count. It does not count the first cycle required to execute any instruction. The counter also increments on each cycle of any instruction fetch stall.

The number of instructions can be calculated as:

$$\text{number of instructions} = \text{CYCCNT} - \text{CPICNT} - \text{EXCCNT} - \text{SLEPCNT} - \text{LSUCNT} + \text{FOLDCNT}$$

Various other calculations and inferences can be made using these counters. For example, the time spent spent executing instructions vs the time sleeping will give an indication of the how hard the CPU is working.

$$\% \text{ time awake} = ((\text{CYCCNT} - \text{SLEPCNT})/\text{CYCCNT}) \times 100$$

6.3 Display features

Whenever the CPU is running, the performance counters will be continuously updated internally. However, with trace enabled within the Performance Counters view, an individual counter will only be sampled when its check box is ticked. When this is done the initial value of all counters held within the debugger will be set to count up from zero.

The total value for any displayed counter is calculated by adding two components:

- the current value of the counter within the CPU - an overflow value

These counters change very rapidly so all counters apart from the Cycle counter will only display their total values when the target is paused.

7. Bandwidth considerations

7.1 Overview

SWO trace allows the user to use multiple SWO functions at the same time. For example, the interrupt trace and profile trace can be running at the same time. The SWO pipeline is composed of five parts which get the trace data from the target into LPCXpresso:

1. the trace hardware on the target generates the data
2. the probe reads this data from the target via the SWO pin
3. The probe caches the data read to be sent to LPCXpresso IDE
4. LPCXpresso IDE requests data from the probe over USB
5. LPCXpresso IDE decodes the SWO stream and displays it to users.

Data loss can occur at any of these steps when they become overloaded. This can happen if the target is configured to generate more trace data than can be handled. When the SWO channel becomes overloaded it is recommended that trace be reconfigured to reduce the load. A user could stop using a component altogether, or reduce the sample rate in profile for example.

Data loss can result in inaccurate timing information and the introduction of corrupted data. It is therefore generally a good idea to adjust your SWO settings to minimize data loss.

7.2 SWO Stats View

The **SWO stats** view provides a low level display of the utilization of the different parts of the SWO pipeline. This view allows users to identify any bottlenecks in the SWO pipeline which may indicate that they are overloading the SWO channel. SWO data is collected into buffers and sent a buffer at a time to the LPCXpresso IDE.

For metric two numbers are presented: the total for the entire SWO session in the “Total” column and the data for the last 2000 collected buffers (corresponding to the last 2s) in the “Windowed” column.

This list will help you interpret the presented statistics:


- Good bytes and Bad bytes
 - Shows how much of the data is being detected as valid, expected SWO packets.
 - A few bad bytes can be expected in normal operation
 - If the number of bad bytes is greater than or the same order of magnitude as good bytes it suggests that the SWO stream is corrupted.
 - This can often be caused when SWO trace is configured with the wrong target clock speed
- Full buffers and Empty buffers
 - This presents the USB utilization. The windowed statistics are most useful here.
 - The more full buffers that there are relative to the empty buffers, the heavier the USB load is.

- The windowed statistics for the full and empty buffers add up to 2000.
- If full = 700 and empty = 1300 you have a lot of head room in the USB channel and should not be losing data there
- If full = 1978 and empty = 22 the USB channel is nearly fully utilized and bursts of data may saturate the USB channel resulting in lost data.
- Seeing the number of empty buffers increase and but no full buffers when you are expecting to see data implies that there target may not be configured correctly – your target may require additional configuration see 'Overview of Trace support in LPCXpresso IDE at <https://community.nxp.com/message/630730> for more information
- Lost buffers
 - This shows the number of buffers of SWO data which were collected by the probe from the target, but were not sent over USB before being overwritten with new data.
 - It is possible to lose buffers even if the USB channel is not fully saturated.
 - A high number of lost buffers relative to full buffers indicates that the SWO channel is overloaded
 - A small number of lost buffers is likely to occur in normal operation
- Overflow packets
 - When the trace hardware on the target generates more data than it can send out of the SWO pin, it generates an Overflow packet.
 - This indicates that the SWO channel is overloaded.
 - Some overflow packets can be expected but most of the time you should aim to have 0 overflow packets in the windowed statistics

Desc	Total	Windowed
▼ Collected Data		
Good Bytes	2181293	194381
Bad Bytes	1022	0
Unconsumed	0	0
▼ Buffers		
Full Buffers	2135	190
Partial Buffers	20292	1810
Empty Buffers	8	0
Lost Buffers	0	0
▼ Overflows		
Fifo Overflows	0	0

Figure 7.1. SWO Stats View

8. Preferences

There are several user configurable options for SWO trace which can be access via the Preferences menu item at:  Preferences -> LPCXpresso -> SWO Trace.

8.1 Options

These options apply to the entire workspace and are persisted between IDE restarts.

- **Data watch buffer depth**
 - The depth of the data watch ring buffer
 - This is the number of data points which will be kept in the history
 - The data watch ring buffer acts like a FIFO containing the most recent events
- **Interrupts capture size**
 - The depth of the interrupt event ring buffer
 - These events are viewable in the **SWO Interrupt Table** view
- **Interrupt Trace Graph Buffer depth**
 - The depth of the ring buffers for the **SWO Interrupt Trace** view
 - If the user interface slows down too much when plotting the entire collected dataset try reducing this value.

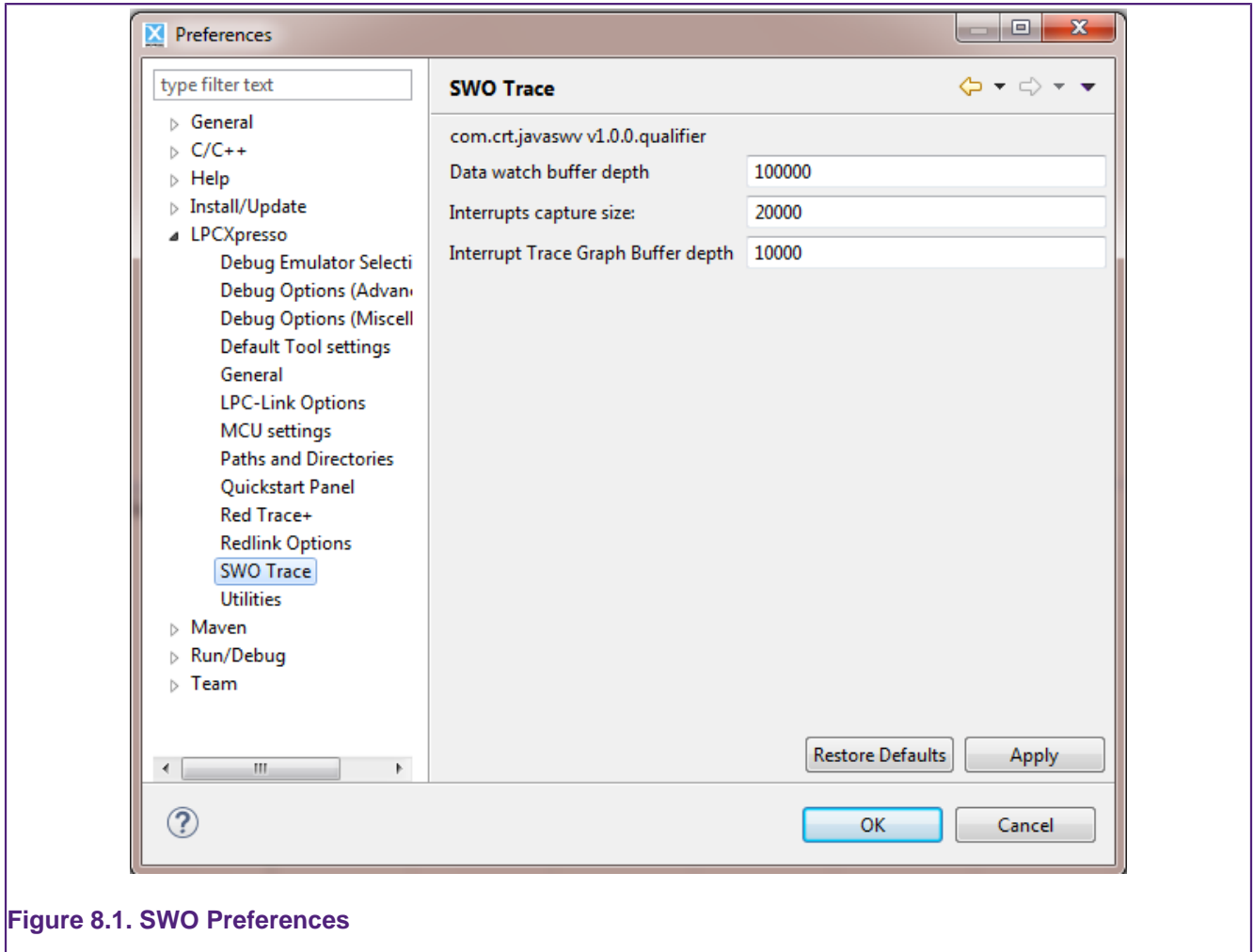


Figure 8.1. SWO Preferences

9. Appendix A – SWO Trace setup for LPC13xx

9.1 To carry out SWO Trace on LPC13xx

1. Ensure that your board has SWO pinned out to the debug connector
2. Turn on the trace clock within the MCU, as this is not enabled by default
3. Configure the MCU pinmuxing, so that the SWO signal is accesible on the appropriate pin of the MCU

This should typically be done within each of your application projects (or library projects that your projects link against).

9.1.1 SWO pin and debug connector

On LPC13xx parts, the SWO signal is accessible via pin P0_9 of the MCU. Thus you will need to check the schematic for you board to ensure that SWO is connected to the debug connector.

9.1.2 Enabling the trace clock

The current startup code generated by the new project wizards for LPC13xx parts in LPCXpresso IDE v7.7 (and earlier) does not yet contain code to optionally turn on the trace clock. This is intended for a future LPCXpresso IDE release. In the meantime, you can use the code below for doing this.

9.1.3 SWO pinmux configuration

On LPC13xx parts, the SWO signal is accessible via pin P0_9 of the MCU

Thus the pinmux settings for your project need to ensure that this pin is configured to use the SWO function. In particular this means that this pin is no longer available for use as the MOSI0 for the SSP0 peripheral. Example code to do this is given below.

Note that the default LPCOpen codebase for LPC1343 and LPC1347 setup P0_9 as MOSI0 in board_sysinit.c of their board library project.

9.1.4 Example setup code for LPC1315/16/46/47 parts

Adding the following code to your main() function, after the call to Board_Init() if using LPCOpen, should allow SWO trace to function:

```
volatile unsigned int *TRACECLKDIV = (unsigned int *) 0x400480AC;
volatile unsigned int *IOCON_PIO_0_9 = (unsigned int *) 0x40044024;
// Write 1 to TRACECLKDIV - Trace divider
*TRACECLKDIV = 1;
// Write 0x93 to I/O configuration for pin PIO0_9 to select ARM_TRACE_SWV
*IOCON_PIO_0_9 = 0x93;
Example setup code for LPC1311/13/42/43 parts
```

Adding the following code to your main() function, after the call to Board_Init() if using LPCOpen, should allow SWO trace to function:

```
volatile unsigned int *TRACECLKDIV = (unsigned int *) 0x400480AC;
```

```
volatile unsigned int *IOCON_PIO_0_9 = (unsigned int *) 0x40044064;
// Write 1 to TRACECLKDIV - Trace divider
*TRACECLKDIV = 1;
// Write 0x93 to I/O configuration for pin PIO0_9 to select ARM_TRACE_SWV
*IOCON_PIO_0_9 = 0x93;
```

10. Appendix B – SWO Trace setup for LPC15xx

10.1 To carry out SWO Trace on LPC15xx

1. Ensure that your board has SWO pinned out to the debug connector
2. Turn on the trace clock within the MCU, as this is not enabled by default
3. Configure the switch matrix, so that the SWO signal is accessible on an appropriate pin of the MCU

This should typically be done within each of your application projects (or library projects that your projects link against).

10.1.1 SWO pin and debug connector

The first revisions of the LPCXpresso1549 board did not provide for connecting the SWO pin on the debug connector. You will need a revision C (or later) of this board to carry out SWO trace.

To identify which revision of board you have, look in the top right of the back of the board which should be marked:

```
LPCXpresso1549
OM13056 v2 Rev C
```

If you are using a different board, you will need to check the schematics to identify if SWO is connected to the debug connector.

10.1.2 Enabling the trace clock

The startup code generated by the new project wizards for LPC15xx in LPCXpresso IDE v7.7 (and later) contain code to optionally turn on the trace clock. By default this code will be enabled by the new project wizard, but if not required can be removed by defining the compiler symbol `DONT_ENABLE_SWVTRACECLK`.

If you are using different startup code (for instance, as provided in the projects in the LPCOpen v2.08c package for the LPCXpresso1549 board), then you can add the following code to your project to turn on the trace clock (for example right at the start of `main()` or in the startup file before the call to `main()`):

```
volatile unsigned int *TRACECLKDIV = (unsigned int *) 0x400740D8;
*TRACECLKDIV = 1;
```

Note that later versions of the LPCOpen package for LPC15xx should contain appropriate trace clock setup code within the startup file.

10.1.3 SWO switch matrix configuration

The SWO signal from the Cortex CPU is not assigned to a fixed pin on the LPC15xx MCU. Thus it is necessary to correctly configure the switch matrix.

For example, if you are using LPCOpen v2.08c package for the LPCXpresso1549 board (and have a revision C or later board), then you will need to modify `board_sysinit.c` in the `lpc_board_nxp_lpcxpresso_1549` project to add SWO to the switch matrix setup thus:

```
STATIC const SWM_GRP_T swmSetup[] = {
/* USB related */
{(uint16_t) SWM_USB_VBUS_I, 1, 11}, /* PIO1_11-ISP_1-AIN_CTRL */
/* UART */
{(uint16_t) SWM_UART0_RXD_I, 0, 13}, /* PIO0_13-ISP_RX */
{(uint16_t) SWM_UART0_TXD_O, 0, 18}, /* PIO0_18-ISP_TX */
/* SWO signal */
{(uint16_t) SWM_SWO_O, 1, 2}, /* PIO01_2-SWO */
};
```

Note that later versions of the LPCOpen package for LPC15xx should contain appropriate SWO switch matrix setup code for the LPCXpresso1549 board in the board library.

11. Appendix C – SWO Trace setup for LPC5410x

11.1 To carry out SWO Trace on LPC5410x

1. Ensure that your board has SWO pinned out to the debug connector
2. If using the built-in LPC-Link2 debug probe of a LPCXpresso54102 board, ensure that the probe is configured for DFU booting
3. Turn on the trace clock within the MCU, as this is not enabled by default
4. Configure the MCU pinmuxing, so that the SWO signal is accesible on the appropriate pin of the MCU

This should typically be done within each of your application projects (or library projects that your projects link against).

Note that SWO Trace on LPC5410x MCUs can only be carried out for debug sessions connected to the Cortex-M4 CPU, not to the Cortex-M0+.

11.1.1 SWO pin and debug connector

On LPCXpresso54102 boards, pin P0_15 is connected to SWO of the debug connector.

If you are using a different board, you will need to check the schematics to identify if SWO is connected to the debug connector (and on which pin from the MCU).

11.1.2 Debug probe

The LPCXpresso54102 board is typically shipped with CMSIS-DAP firmware installed into the built-in LPC-Link2 debug probe that does not provide the necessary channels for SWO trace data transfer back to the host debugger. You need to configure your board so that the LPCXpresso IDE can softload “redlink” firmware. This is done by fitting a jumper on JP5, then repowering the board. For more details, please see <http://www.nxp.com/pages/LPCXPRESSO-BOARDS> .

11.1.3 Enabling the trace clock

The startup code generated by the new project wizards for LPC5410x in LPCXpresso IDE v7.7 (and later) contain code to optionally turn on the trace clock. By default this code will be enabled by the new project wizard, but if not required can be removed by defining the compiler symbol DONT_ENABLE_SWVTRACECLK.

If you are using different startup code (for instance, as provided in the projects in the LPCOpen v2.14.1 package for the LPCXpresso5410x board), then you can add the following code to your project to turn on the trace clock (for example right at the start of main() or in the startup file before the call to main()) :

```
volatile unsigned int *TRACECLKDIV = (unsigned int *) 0x400000E4;
volatile unsigned int *SYSAHBCLKCTRLSET = (unsigned int *) 0x400000C8;
// Write 0x00000001 to TRACECLKDIV (0x400000E4) - Trace divider
*TRACECLKDIV = 1;
// Enable IOCON peripheral clock (for SWO on PI00-15 or PI01_1)
// by setting bit13 via SYSAHBCLKCTRLSET[0] (0x400000C8)
*SYSAHBCLKCTRLSET = 1 << 13; // 0x2000
```

Note that later versions of the LPCOpen package for LPC5410x should contain appropriate trace clock setup code in the startup file.

11.1.4 SWO pinmux configuration

On LPCXpresso54102 boards, pin P0_15 is connected to SWO of the debug connector. This needs configuring in the pinmux settings for your project.

If you are using the examples from the LPCOpen v2.14.1 package for the LPCXpresso5410x board, then you need to modify the pinmux settings in `board_sysinit.c` of the `lpc_board_lpcxpresso_54102` library project, so that the SWO settings are not commented out by the `#if 0 ... #endif` clause (as they are by default). Thus after modifying the code, it should look like:

```
{0, 15, (IOCON_FUNC2 | IOCON_MODE_INACT | IOCON_DIGITAL_EN)}, /* SWO */
#if 0 /* Debugger signals, do not touch */
{0, 16, (IOCON_FUNC5 | IOCON_MODE_INACT | IOCON_DIGITAL_EN)}, /* SWCLK_TCK */
{0, 17, (IOCON_FUNC5 | IOCON_MODE_INACT | IOCON_DIGITAL_EN)}, /* SWDIO */
#endif
```

Note that later versions of the LPCOpen package for LPC5410x should contain appropriate SWO pinmux setup code for the LPCXpresso54102 board in the board library.