LPCXpresso v8 User Guide

Rev. 8.0 — 18 November, 2015 User guide

18 November, 2015

1. Introduction to LPCXpresso

LPCXpresso is a low-cost microcontroller (MCU) development platform ecosystem from NXP, which provides an end-to-end solution enabling embedded engineers to develop their applications from initial evaluation to final production.

The LPCXpresso platform ecosystem includes:

- The LPCXpresso IDE, a software development environment for creating applications for NXP's ARM based 'LPC' range of MCUs.
- The range of LPCXpresso development boards, which each include a built-in 'LPC-Link', 'LPC-Link2' or CMSIS-DAP debug probe. These boards are developed in collaboration with Embedded Artists.
- The standalone 'LPC-Link 2' debug probe.

This guide is intended as an introduction to using LPCXpresso, with particular emphasis on the LPCXpresso IDE. It assumes that you have some knowledge of MCUs and software development for embedded systems.

1.1 LPCXpresso IDE Overview of Features

The LPCXpresso IDE is a fully featured software development environment for NXP's ARM-based MCUs, and includes all the tools necessary to develop high quality embedded software applications in a timely and cost effective fashion.

The LPCXpresso IDE is based on the Eclipse IDE and features many ease-of-use and MCU specific enhancements. The LPCXpresso IDE also includes the industry standard ARM GNU tools enabling professional quality tools at low cost. The fully featured debugger supports both SWD and JTAG debugging, and features direct download to on-chip flash.

For the latest details on new features and functionality, visit http://www.lpcware.com/content/forum/lpcxpresso-latest-release

1.1.1 Summary of Features

- Complete C/C++ integrated development environment
  - Latest Eclipse-based IDE with many ease-of-use enhancements
    - Eclipse Mars (v4.6) and CDT (8.8)
  - IDE can be further enhanced with Eclipse plugins
    - CVS source control built in; Subversion, TFS, Git, and others available for download
  - Command-line tools included for integration into build, test, and manufacturing systems
- Industry standard GNU toolchain (v4.9.3), including
  - C and C++ compilers, assembler, and linker
  - Converters for SREC, HEX, and binary
- Fully featured debugger supporting JTAG and SWD
  - Built-in flash programming

- High-level and instruction-level debug
- Views of CPU registers and on-chip peripherals
- Support for multiple devices on JTAG scan-chain

- Library support
  - Redlib: a small-footprint embedded C library
  - Newlib: a complete C and C++ library
  - NewlibNano: a new small-footprint C and C++ library, based on Newlib
  - LPCOpen MCU software libraries
  - Cortex Microcontroller Software Interface Standard (CMSIS) libraries and source code
- Device-specific support for NXP's ARM-based MCUs (including Cortex-M, ARM7 and ARM9 based parts)
  - Automatic generation of linker scripts for correct placement of code and data into flash and RAM
  - Startup code and device initialization
  - No assembler required with Cortex-M based MCUs
- *Trace* [38]
  - Instruction trace via Embedded Trace Buffer (ETB) on certain Cortex-M3/M4 based MCUs or Micro Trace Buffer (MTB) on Cortex-M0+ based MCUs
  - SWO Trace on Cortex-M3/M4 based MCUs when debugging via LPC-Link2, providing functionality including
    - Profile tracing
    - Interrupt tracing
    - Datawatch tracing
    - Printf over ITM
  - Red Trace on Cortex-M3/M4 based MCUs when debugging via Red Probe+ (legacy)
- *Red State* [95] state machine designer and code generator
  - Graphically design your state machines
  - Generates standard C code
  - Configures NXP State Configurable Timer (SCT) as well as supporting software state machines

1.1.2 Supported debug probes

The following debug probes are supported by LPCXpresso IDE for general debug connections:

- LPC-Link (Original LPCXpresso board)

- LPC-Link 2 (with “CMSIS-DAP” firmware) - either the standalone debug probe or the version built into LPCXpresso V2 and V3 boards
- CMSIS-DAP enabled debug probes, such as LPC800-MAX, LPCXpresso824-MAX, LPCXpresso1769/CD, Keil ULINK-ME etc.
- Older debug probes
  - Red Probe / Red Probe+
  - RDB1768 development board built-in debug connector (RDB-Link)
  - RDB4078 development board built-in debug connector

For more information on debug probe support in LPCXpresso IDE, visit http://www.lpcware.com/content/faq/lpcxpresso/supported-debug-probes-mcus

Support for GDB server based debug connections is also provided. This feature enables support for 3rd party debug probes, such as Segger J-Link. When debugging with GDB server connections, some functionality may be disabled.

- For more information on using Segger J-Link with LPCXpresso, visit http://www.segger.com/nxp-lpcxpresso.html

1.1.3 LPCXpresso Development Boards

A major part of the LPCXpresso platform is the range of LPCXpresso boards which work seamlessly with the LPCXpresso IDE. These boards provide practical and easy to use development hardware to use as a starting point for your LPC Cortex-M MCU based projects.

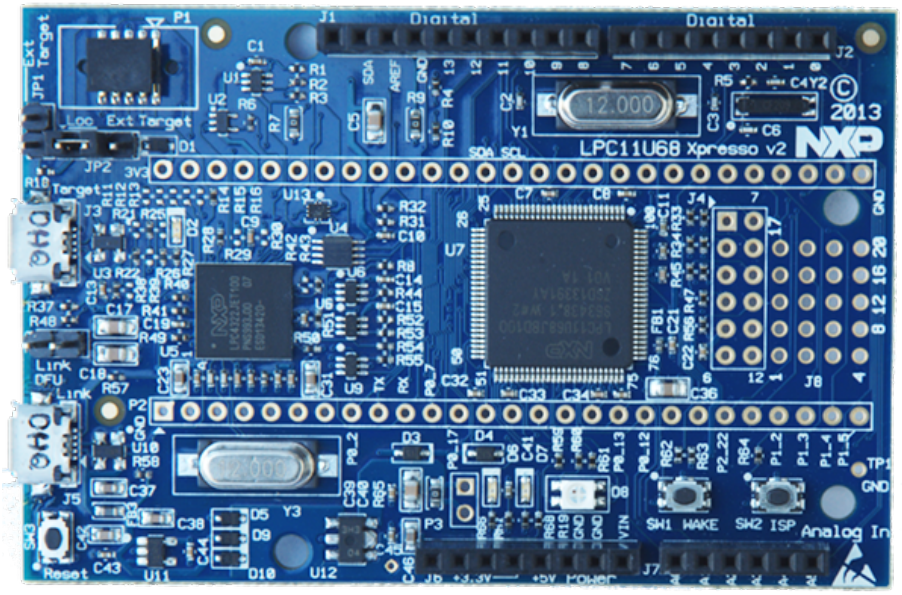


Figure 1.1. LPCXpresso V2 Board (LPCXpresso11U68)

For more information, visit:

http://www.lpcware.com/LPCXpressoBoards

2. Installation and Licensing

2.1 Host Computer Hardware Requirements

Before installation of the LPCXpresso IDE, you should make sure your development host computer meets the following requirements:

- A standard x86 PC with 2GB RAM minimum (4GB+ recommended) and 950MB+ of available disk space, running one of the following operating systems:
  - Microsoft® Windows Vista
  - Microsoft® Windows 7
  - Microsoft® Windows 8.1
  - Microsoft® Windows 10
  - Linux – Ubuntu 12 and later
  - Linux – Fedora 18 and later
- An x86 Apple Macintosh with 2GB RAM minimum (4GB+ recommended) and 950MB+ of available disk space, running one of the following Mac OS X versions:
  - 10.8.5 (or later) / 10.9.4 (or later) / 10.10 (or later)

Additional host platform notes:

- Starting with LPCXpresso v7.1, and following the discontinuation of support by Microsoft, Windows XP is no longer an officially supported platform. LPCXpresso **may** continue to work on Windows XP but this can no longer be guaranteed. LPCXpresso is no longer tested on Windows XP.
- Starting with LPCXpresso v7.4, Mac OS X 10.7 (Lion) is no longer an officially supported platform. LPCXpresso **may** continue to work on Mac OS X 10.7 but this can no longer be guaranteed. LPCXpresso is no longer tested on Mac OS X 10.7.
- Both 32-bit and 64-bit Windows / Linux systems are supported.
- The LPCXpresso IDE may install and run on other Linux distributions. However, only the distributions listed above have been tested. We have no plans to officially support other distributions at this time.
- A screen resolution of 1024x768 minimum is recommended.
- An internet connection is **required** to request, install, and activate license keys. When using the product, an internet connection is required to update the product and to download new examples.

2.2 Installation

When installing, all components of the LPCXpresso IDE are installed, but some functionality may be restricted by the currently installed license activation.

2.2.1 Windows

The LPCXpresso IDE is installed into a single directory, of your choice. Unlike many software packages, the LPCXpresso IDE does not install or use any keys in the Windows

Registry, or use or modify any environment variables (including PATH), resulting in a very clean installation that does not interfere with anything else on your PC. Should you wish to use the command-line tools, a command file is provided to set up the path for the local command window.

2.2.2 Linux

Due to the huge variation in capabilities of different Linux distributions, LPCXpresso is only tested and supported on recent distributions of Ubuntu and Fedora. The LPCXpresso IDE **may** work on other distributions but we cannot provide support if it does not work.

The installer is supplied as an executable that installs the LPCXpresso IDE components. The installer requires root privileges, although, once it is installed, no special privileges are required to run the LPCXpresso IDE. The installer will request a super-user password when it is started. Once installation has completed, we strongly recommend that your system is restarted — if you do not do this then some areas of the tools may not function correctly. The installer should be started from the command line, but will switch to GUI mode once the super-user password has been entered.

Installation on 32-bit distributions

The LPCXpresso IDE is a 32-bit product that should run on most 32-bit Linux distributions. On some distributions some additional packages may be required for USB compatibility (see below for the latest information). Note that GLIBC v2.15 or greater is required.

Enabling the LPCXpresso IDE internal Web Browser

On most recent Linux distributions, an additional package must be installed to enable the IDE's internal Web Browser. If this is not installed, any web pages displayed by the IDE will be displayed in the system default Web Browser.

Ubuntu

Run the following command to install the webkit package that enables the internal Web Browser:

```
sudo apt-get install libwebkitgtk-1.0-0
```

Fedora

Run the following command to install the webkit package that enables the internal Web Browser:

```
sudo yum install webkitgtk.i686
```

Installation on 64-bit distributions

The LPCXpresso IDE is a 32-bit product. For 64-bit versions of Linux, the 32-bit compatibility components must be installed. Note that all of these components must be installed, otherwise the installation program will not run and the LPCXpresso IDE will not function correctly. To install these components from the command line, follow the instructions below for your distribution.

Ubuntu 13.10 or later:

Ubuntu 13.10 no longer provides a convenient method to install all 32-bit compatible libraries, so they must all be installed individually. Those that are required for LPCXpresso 7 are as follows:

```
sudo apt-get install libgtk2.0-0:i386 libxtst6:i386 libpangox-1.0-0:i386 \
     libpangoxft-1.0-0:i386 libidn11:i386 libglu1-mesa:i386 \
     libncurses5:i386 libudev1:i386 libusb-1.0:i386 libusb-0.1:i386 \
     gtk2-engines-murrine:i386 libnss3-1d:i386 libwebkitgtk-1.0-0
```

Ubuntu 13.04 or earlier:

Ubuntu 13.04 and earlier provides a convenient method to install 32-bit compatible libraries. These can be installed using the follow command:

```
sudo apt-get install linux32 ia32-libs
```

Fedora

Install the following 32-bit libraries:

```
sudo yum install gtk2.i686 glibc.i686 glibc-devel.i686 libstdc++.i686 \
 zlib-devel.i686 ncurses-devel.i686 libX11-devel.i686 libXrender.i686 \
 libXrandr.i686 libusb.i686 libXtst.i686 nss.i686 libcanberra-gtk2.i686 \
 PackageKit-gtk3-module.i686 webkitgtk.i686
```

Other Linux Distros

While not officially supported, the LPCXpresso IDE has been reported to work on many other Linux Distros, including Linux Mint, openSUSE and Debian. When attempting to run on these Distros, remember that LPCXpresso is a 32-bit application and so various 32-bit compatible libraries must be installed. These include:

```
libgtk2.0-0:i386 libxtst6:i386 libpangox-1.0-0:i386 libpangoxft-1.0-0:i386 \
libidn11:i386 libglu1-mesa:i386 libncurses5:i386 libudev1:i386 \
libusb-1.0:i386 libusb-0.1:i386 gtk2-engines-murrine:i386 \
libnss3:i386
```

Also note that that the glibc 2.15 shared library is required.

We are unable to provide assistance when installing on other Distros, but the LPCXpresso forum on http://www.lpcware.com is a good place to search for information or post questions.

Post-installation issues

dfu-util fails to run

On some Linux systems, when booting LPC-Link or LPC-Link2, the supplied version of dfu-util may fail to execute. To resolve this you may need to install an additional component:

```
 sudo apt-get install libusb-0.1-4:i386 # Ubuntu
```

This should only be installed on systems where dfu-util fails to run.

Connection Refused error

When starting a debug session, on some Linux systems, a **"Connection refused"** error may be displayed. This happens because a critical system library is not installed where we expect to find it.

To resolve this issue, open a Terminal Window and execute the following commands (if running a 64-bit version of Linux, you must first install the 32-bit compatibility libraries as described above):

Note: The actual location of this library may depend on the distribution and version of Linux you are running.

Ubuntu

```
cd /lib/i386-linux-gnu
sudo ln -sf libudev.so.1 libudev.so.0
```

Fedora

```
cd /usr/lib/
sudo ln -sf libudev.so.1 libudev.so.0
```

Ubuntu-specific issues

When using the Unity interface, there may be an issue preventing some menu items from displaying in the LPCXpresso IDE (this does not affect the 'Classic' interface). To workaround this problem, create a shell script with the following content, and start the LPCXpresso IDE by running this script:

```
#!/bin/bash
export UBUNTU_MENUPROXY=0
/usr/local/<lpcxpresso_install_dir>/lpcxpresso/lpcxpresso
```

Fedora-specific issues

If SELINUX is used, it must be set to "permissive" mode to allow the LPCXpresso IDE to run.

2.2.3 Mac OS X

The LPCXpresso IDE installer is supplied as a Mac OS X `.pkg` installer file. Double click on the installer to install the LPCXpresso IDE into a subfolder of your Applications folder.

To start the LPCXpresso IDE, use the Mac OS X Launchpad. Alternatively click the **Open lpcxpresso** icon in the `/Applications/lpcxpresso_version` folder or run **lpcxpresso.app**, which can be found in the `lpcxpresso` subfolder of the main LPCXpresso IDE installation directory within `/Applications`.

2.2.4 Running under virtual machines

It is possible to install the LPCXpresso IDE within a virtual machine (VM) environment. Generally such installations cause few issues. However due to the nature of VMs the most likely problems relate to sharing of resources (USB, memory), and possible timeouts during debug operations.

In the unlikely event that you experience issues, we welcome reports but due to the nature of VM operation can offer no guarantee of resolution.

2.3 Licensing Overview

An LPCXpresso IDE installation will take one of the following three forms:

- **Unregistered** : the initial product state following a new customer installation, features are restricted, and code size is restricted to 8KB

- **Free Edition** : following installation of a Free Edition Activation code, maximum debug download size is limited to 256KB
- **Pro Edition** : following installation of Purchased Pro Edition Activation code, no restrictions

Note the current installation type can be identified by Help -> Display License Type

2.3.1 Users of earlier versions of LPCXpresso IDE

If you have previously been using LPCXpresso IDE v5.x or earlier, then note that your previous activation code is not compatible with this version. You will need to go through the activation process again in order to use this version.

If you have previously been using LPCXpresso IDE v6.x or v7.x, then your existing activation code is compatible with LPCXpresso IDE v8.x. There is no need to reactivate.

2.3.2 Users of Code Red Technologies Red Suite products

Red Suite activation codes are not compatible with LPCXpresso IDE. You will need to obtain an LPCXpresso Free or Pro Edition activation code in order to use LPCXpresso IDE.

2.4 Unregistered (UNREGISTERED) license

Initially after installation LPCXpresso IDE will run with a default Unregistered (UNREGISTERED) license. Most features of the product may be used, although some functionality is restricted, including the size of applications that you can build and debug (limited to 8Kbytes), and Red Trace functionality is disabled. Activate your product with a Free Edition or Pro Edition license to remove these restrictions.

2.5 Activating your product (LPCXpresso Free Edition)

A Free Edition activation code may be obtained, **free of charge**, by registering the LPCXpresso IDE. This provides a license to use the complete development environment with a 256KB debug download size limit.

Note: You will need to have created an account and logged on to the LPCWare website to be able to obtain a Free Edition activation code.

To activate your installation with a Free Edition license, from within LPCXpresso IDE:

1. Go to the menu entry **Help->Activate->Create Serial number and register (Free Edition)...**
   - Your product's serial number will be displayed
   - Write down the serial number, or copy it into the clipboard.
2. Press **OK** and a web browser will be opened on the Activations page
   - If you are already logged in to the website, the serial number will be completed for you.
   - If you are not logged in, you will need to login, navigate to http://www.lpcware.com/lpcxpresso/activate, and enter the product's serial number.
3. Press the button to Register LPCXpresso
   - Your LPCXpresso Activation code will be generated and displayed.

4. Go to the menu entry **Help->Activate->Activate (Free Edition)...**

5. Enter your activation code and Press **OK**.

   - This activates your product. The license type will be displayed and you will be able to use all the features of LPCXpresso, with a debug download limit of 256KB.

2.6 Activating your product (LPCXpresso Pro Edition)

A full, unrestricted activation code for the LPCXpresso IDE can be purchased via the menu entry **Help->Activate->Purchase from LPCXpresso webstore**. Once purchased, your activation code will be emailed to you.

When the activation code is received, follow the instructions below to activate your product. From within the LPCXpresso IDE:

1. Go to the menu entry **Help->Activate->Activate (Pro Edition)...**

2. Enter the Pro Edition activation code provided, and press **OK**

3. Enter your email address and provide a password of your choosing, and press **OK**.

Your product will now be activated. The license type will be displayed and you will be able to use all the features of LPCXpresso with no code size limits.

NOTES

1. The password should be kept safe, as it will be required should you wish reactivate your license (for example to move your license for LPCXpresso Pro Edition from one PC to another). Your email address will be used to send a reminder should you forget your password. We may also send occasional emails from which you may unsubscribe.

2. The first use of an activation code will also trigger an email from Softworkz (our licensing system partners) to the supplied email address. This email is important because it offers the option to create an account to centrally manage multiple seats of LPCXpresso. See http://www.lpcware.com/content/faq/lpcxpresso/initial-activation-email

2.6.1 Multi-Seat Activations

When you purchase LPCXpresso Pro Edition, you can choose to purchase a single or multi seat license. If you purchase a multi seat license, then you will be provided with a single activation code that can be used to activate LPCXpresso Pro Edition on multiple machines.

The major benefit of a multi seat activation code over multiple single seat codes is that this enables a single activation code to be made available to users whilst offering the ability to monitor and administer usage centrally.

Single seat licenses can be upgraded to multi-seat licenses by purchasing additional seats.

For additional information please visit:

http://www.lpcware.com/content/faq/lpcxpresso/central-administration-activation-codes

2.7 Further information on installation and licensing

Further information on LPCXpresso IDE installation, licensing and activation codes can be found in our FAQs at:

http://www.lpcware.com/faq/activation-licensing

3. LPCXpresso IDE Overview

3.1 Documentation and Help

The LPCXpresso IDE is based on the Eclipse IDE framework, and many of the core features are described well in generic Eclipse documentation and in the help files to be found on the LPCXpresso IDE's **Help -> Help Contents** menu. This also provides access to the LPCXpresso User Guide (this document), as well as the documentation for the compiler, linker, and other underlying tools.

To obtain assistance on using LPCXpresso visit

http://lpcware.com/lpcxpresso/support

3.2 Workspaces

When you first launch LPCXpresso IDE, you will be asked to select a Workspace, as shown in Figure 3.1.

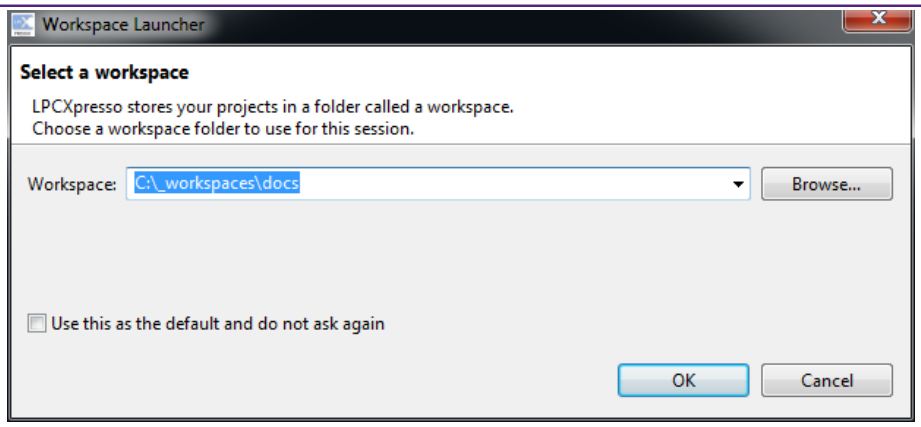


Figure 3.1. Workspace selection.

A workspace is simply a directory that is used to store the projects you are currently working on. Each workspace can contain multiple projects, and you can have multiple workspaces on your computer. The LPCXpresso IDE can only have a single workspace open at a time, although it is possible to run multiple instances in parallel — with each instance accessing a different workspace.

If you tick the **Use this as the default and do not ask again** option, then the LPCXpresso IDE will always start up with the chosen workspace opened. Otherwise you will always be prompted to choose a workspace.

It is also possible to change workspace whilst running the LPCXpresso IDE, using the **File -> Switch Workspace** option.

3.3 Perspectives and Views

The overall layout of the main LPCXpresso IDE window is known as a Perspective. Within each Perspective are many sub-windows, called Views. A View displays a particular set of data in the LPCXpresso environment. For example, this data might be source code,

hex dumps, disassembly, or memory contents. Views can be opened, moved, docked, and closed, and the layout of the currently displayed Views can be saved and restored.

Typically, the LPCXpresso IDE operates using the single **Develop Perspective**, under which both code development and debug sessions operate as shown in Figure 3.3. This single perspective simplifies the Eclipse environment, but at the cost of slightly reducing the amount of information displayed on screen.

Alternatively the LPCXpresso IDE can operate in a 'dual perspective' mode such that the **C/C++ Perspective** is used for developing and navigating around your code and the **Debug Perspective** is used when debugging your application.

You can manually switch between Perspectives using the Perspective icons in the top right of the LPCXpresso IDE window, as per Figure 3.2.

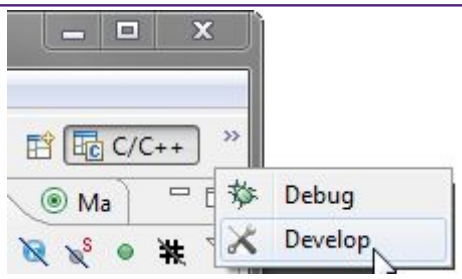


Figure 3.2. Perspective selection.

All Views in a Perspective can also be rearranged to match your specific requirements by dragging and dropping. If a View is accidentally closed, it can be restored by selecting it from the **Window -> Show View** dialog.

3.4 Major components of the Develop Perspective

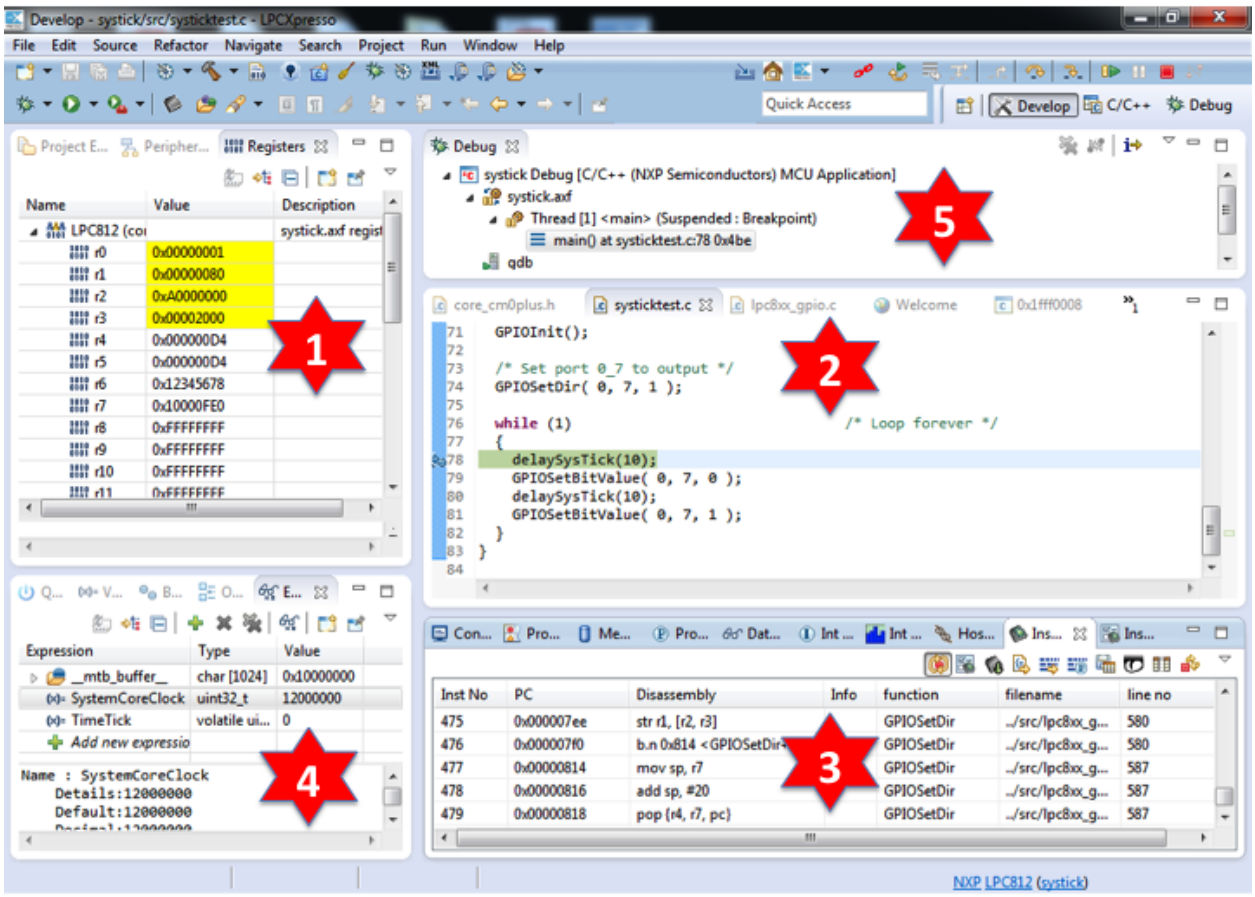


Figure 3.3. Develop Perspective (whilst debugging)

1. Project Explorer / Peripherals / Registers Views
   - The **Project Explorer** gives you a view of all the projects in your current 'Workspace'.
   - When debugging, the **Peripherals** view allows you to display the registers within Peripherals.
   - When debugging, the **Registers** view allows you to display the registers within the CPU of your MCU.
2. Editor
   - On the upper right is the editor, which allows modification and saving of source code. When debugging, it is here that you will see the code you are executing and can step from line to line. By pressing the `'i->'` icon at the top of the Debug view, you can switch to stepping by assembly instruction. Clicking in the left margin will set and delete breakpoints.
3. Console / Problems / Red Trace Views
   - On the lower right are the Console and Problems Views. The Console View displays status information on compiling and debugging, as well as semihosted program output. The Problem View (available by changing tabs) shows all compiler errors and will allow easy navigation to the error location in the Editor View.

- Located in parallel with the Console View are the various views that make up the Trace functionality of LPCXpresso IDE. The Trace views allow you to gather and display runtime information using the SWV technology that is part of Cortex-M3/M4 based parts. In addition, for some MCUs, you can also view instruction trace data downloaded from the MCU's Embedded Trace Buffer (ETB) or Micro Trace Buffer (MTB). The example here shows instruction trace information downloaded from an LPC812's MTB. For more information on Trace functionality, please see Section 8.1.

4. Quick Start / Variables / Breakpoints / Expressions Views

   - On the lower left of the window, the **Quickstart Panel** has fast links to commonly used features. This is the best place to go to find options such as Build, Debug, and Import.
   - Sitting in parallel to the 'Quickstart' view, the **Variable** view allows you to see the values of local variables.
   - Sitting in parallel to the 'Quickstart' view, the **Breakpoint** view allows you to see and modify currently set breakpoints.
   - Sitting in parallel to the 'Quickstart' view, the **Expressions** view allows you to add global variables and other expressions so that you can see and modify their values.

5. Debug View

   - The Debug view appears when you are debugging your application. This shows you the stack trace. In the 'stopped' state, you can click on any particular function and inspect its local variables in the Variables tab (which is located parallel to the **Quickstart Panel**).

4. Importing and Debugging example projects

4.1 Software drivers and examples

LPCOpen is now the preferred software platform for most NXP Cortex-M based MCUs, replacing the various CMSIS / Peripheral Driver Library / code bundle software packages made available in the past. LPCOpen has been designed to allow you to quickly and easily utilize an extensive array of software drivers and libraries in order to create and develop multifunctional products. Amongst the features of LPCOpen are:

- MCU peripheral device drivers with meaningful examples
- Common APIs across device families
- Thoroughly tested and maintained
- Commonly needed third party and open source software ports
- Support for Keil, IAR and LPCXpresso toolchains

The latest LPCOpen now available provides:

- MCU family specific download package
- Support for USB ROM drivers
- Improved code organization and drivers (efficiency, features)
- Improved support for LPCXpresso IDE

CMSIS / Peripheral Driver Library / code bundle software packages are still available, both within your LPCXpresso IDE install directory **\lpcxpresso\Examples\NXP** and also downloadable from NXP LPCware website. But generally these should only be used for existing development work. When starting a new evaluation or product development, we would recommend the use of LPCOpen.

More information on LPCOpen together with package downloads can be found at:

http://www.lpcware.com/lpcopen

4.2 Importing an Example project

The **Quickstart Panel** provides rapid access to the most commonly used features of the LPCXpresso IDE. Using the **Quickstart Panel**, you can quickly import example projects, create new projects, build projects and debug projects.

On the **Quickstart Panel**, click on the ‘Start Here’ sub-panel, and click on Import project(s).

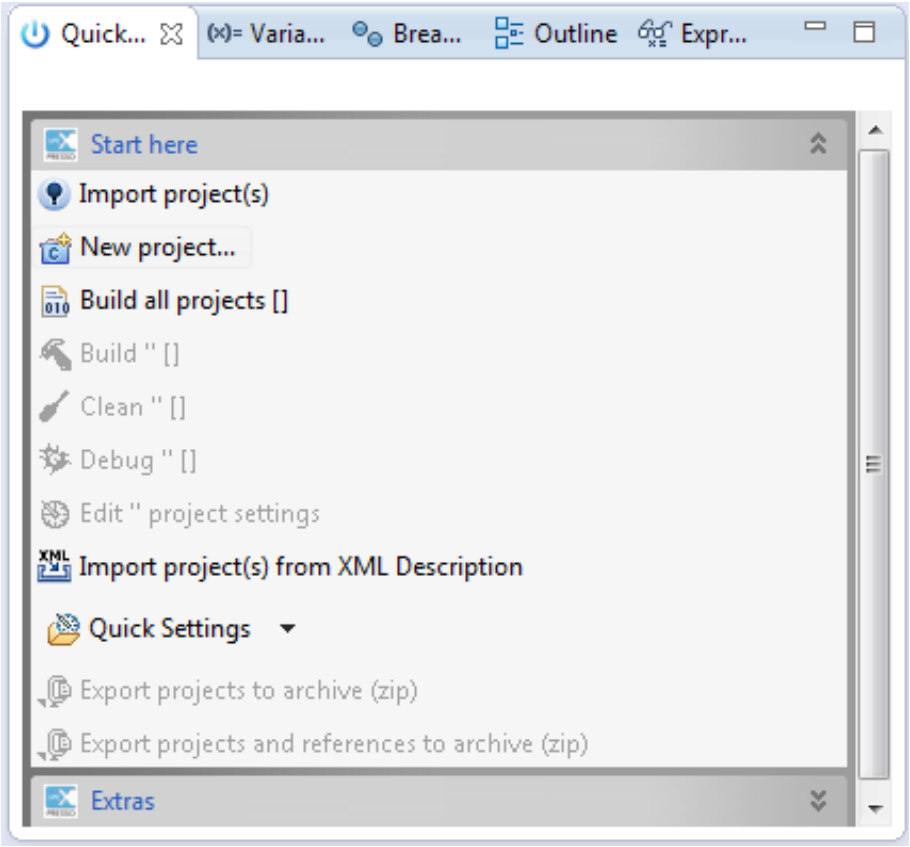


Figure 4.1. Import project(s)

As per Figure 4.2, from the first page of the wizard, you can

- Browse to locate Examples stored in zip archive files on your local system. These could be archives that you have previously downloaded (for example LPCOpen packages from the NXP LPCware website or the supplied, but deprecated, sample code bundles located within the Examples subdirectory of your LPCXpresso IDE installation.
- Browse to locate projects stored in directory form on your local system (for example, you can use this to import projects from a different workspace into the current workspace).
- Browse LPCOpen packages to visit LPCware and download appropriate LPCOpen package for your target MCU. This option will automatically open a web browser onto an appropriate links page on the NXP LPCware website.

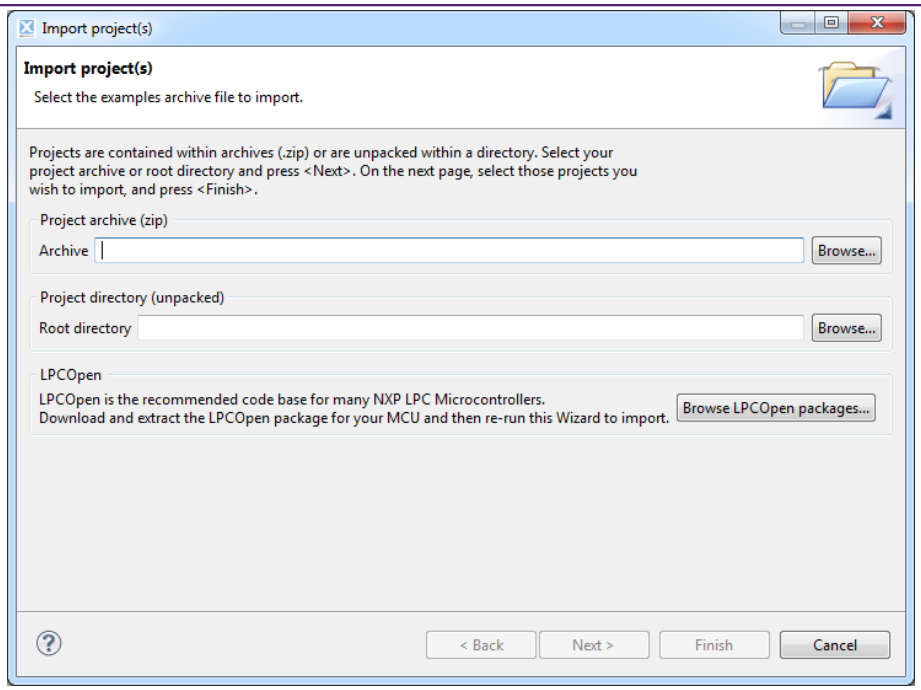


Figure 4.2. Import Examples

To demonstrate how to use the Import Project(s) functionality, we will now import the LPCOpen examples for the LPCXpresso4337 development board.

4.2.1 Importing Examples for the LPCXpresso4337 Development Board

First of all, assuming that you have not previously downloaded the appropriate LPCOpen package, click on the **Browse LPCOpen Packages**, which will open a web browser window. Click on **Download LPCOpen Packages** link, and then the link to the **LPCOpen v2.xx for LPC43xx family devices**, and then choose the download for the LPCXpresso4337 board.

Once the package has downloaded, return to the Import Project(s) dialog and click on the **Browse** button next to **Project archive (zip)** and locate the LPCOpen LPCXpresso4337 package archive previously downloaded. Select the archive, click **Open** and then click **Next**. You will then be presented with a list of projects within the archive, as shown in Figure 4.3.

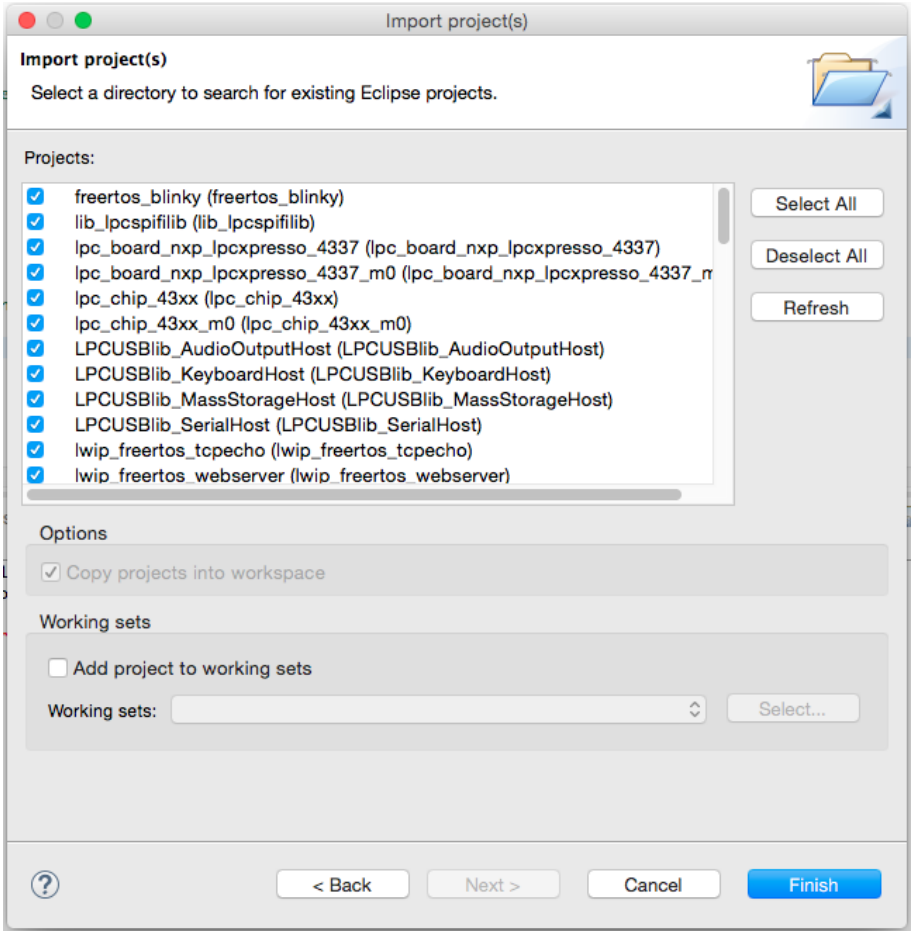


Figure 4.3. Select projects to import

Select the projects you want to import and then click **Finish**. The examples will be imported into your workspace.

Note that generally it is a good idea to leave all projects selected when doing an import from a zip archive file of examples. This is certainly true the first time you import an example set, when you will not necessarily be aware of any dependencies between projects. In most cases, an archive of projects will contain one or more library projects, which are used by the actual application projects within the examples. If you do not import these library projects, then the application projects will fail to build.

4.3 Building projects

Building the projects in a workspace is a simple case of using the **Quickstart Panel** to 'Build all projects'. Alternatively a single project can be selected in the Project Explorer View and built. Note that building a single project may also trigger a build of any associated library projects.

4.3.1 Build configurations

By default, each project will be created with two different 'build configurations' - **Debug** and **Release**. Each build configuration will contain a distinct set of build options. Thus a **Debug** build will typically compile its code with optimizations disabled (`-O0`) and **Release** will compile its code optimizing for minimum code size (`-Os`). The currently selected build configuration for a project will be displayed after its name in the Quickstart Panel's Build/ Clean/Debug options.

For more information on switching between Build Configurations, see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/change-build-config

4.4 Debugging a project

This description shows how to run the LPCOpen `nxp_lpcxpresso_4337_periph_blinky` example application for the LPCXpresso4337 development board. The same basic principles will apply for other examples and boards.

First of all you need to ensure that your LPCXpresso development board is connected to your computer. Note that some LPCXpresso development boards have two USB connectors fitted. Make sure that you have connected the lower connector marked DFU-Link.

When debug is started, the program is automatically downloaded to the target and is programmed into FLASH memory, a default breakpoint is set on the first instruction in `main()`, the application is started (by simulating a processor reset), and code is executed until the default breakpoint is hit.

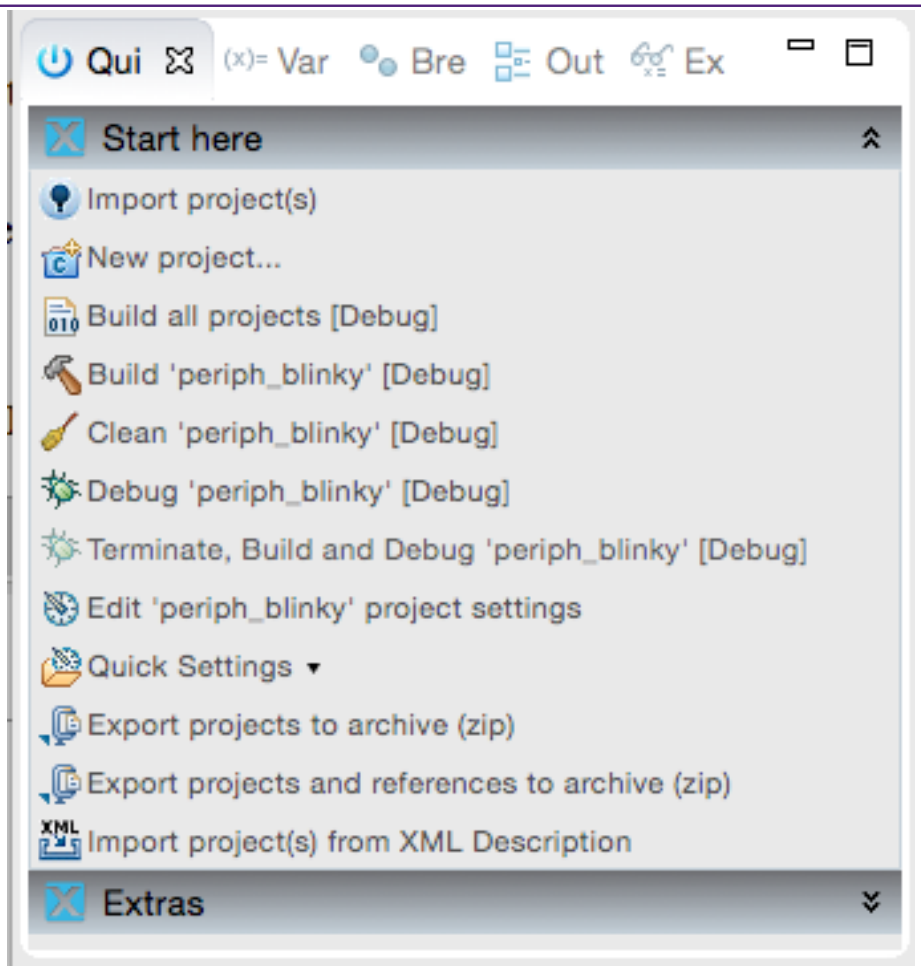


Figure 4.4. Launch debug session

To start debugging `nxp_lpcxpresso_4337_periph_blinky` on your target, simply highlight the project in the Project Explorer and then in the **Quickstart Panel** click on **Start Here** and select **Debug 'periph_blinky'**, as in Figure 4.4.

The LPCxpresso IDE will first build and then start debugging the application. Click on **OK** to continue with the download and debug of the 'Debug' build of your project.

4.4.1 Debug Emulator Selection Dialog

The first time you debug a project, the Debug Emulator Selection Dialog will be displayed, showing all supported probes that are attached to your computer. In this example, an LPC-Link2 and a LPCXpressoCD board have been found.

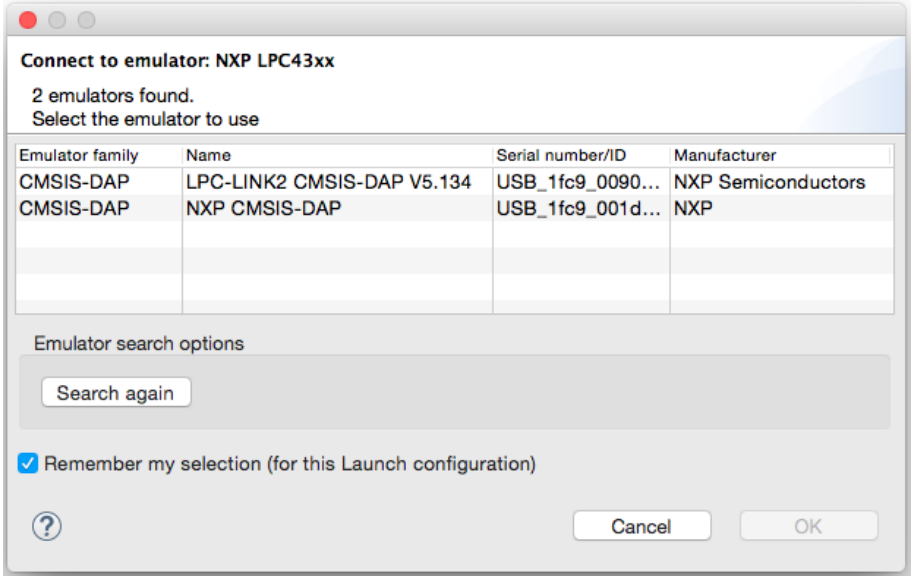


Figure 4.5. Attached probes

You now need to select the probe that you wish to debug through. In Figure 4.6 the LPC-Link2 has been selected, which is what we would do, for example, in order to debug an LPCXpresso4337 board.

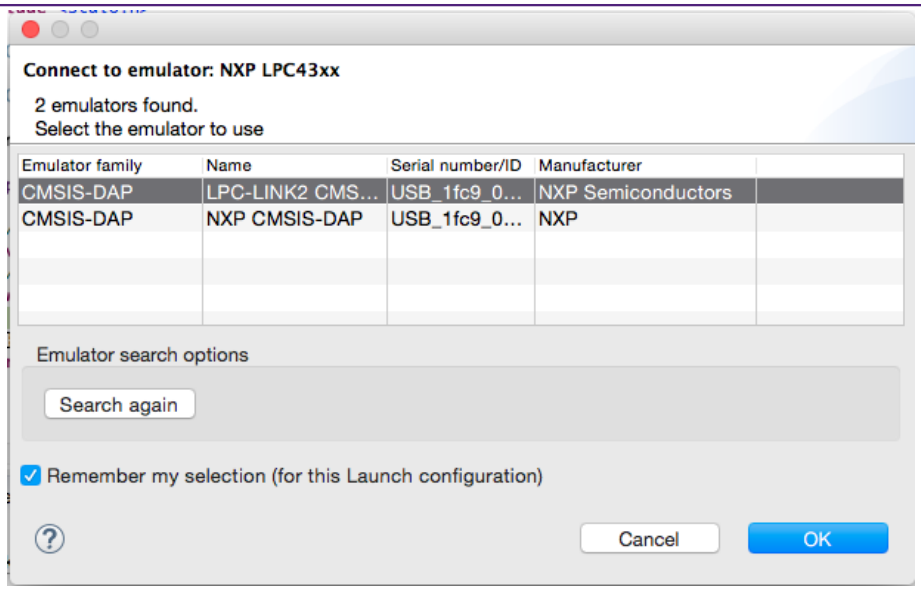


Figure 4.6. LPC-Link selected

For any future debug sessions, the stored probe selection will be automatically used, unless the probe cannot be found. In Figure 4.7 the previously selected LPC-Link2 is no longer connected.

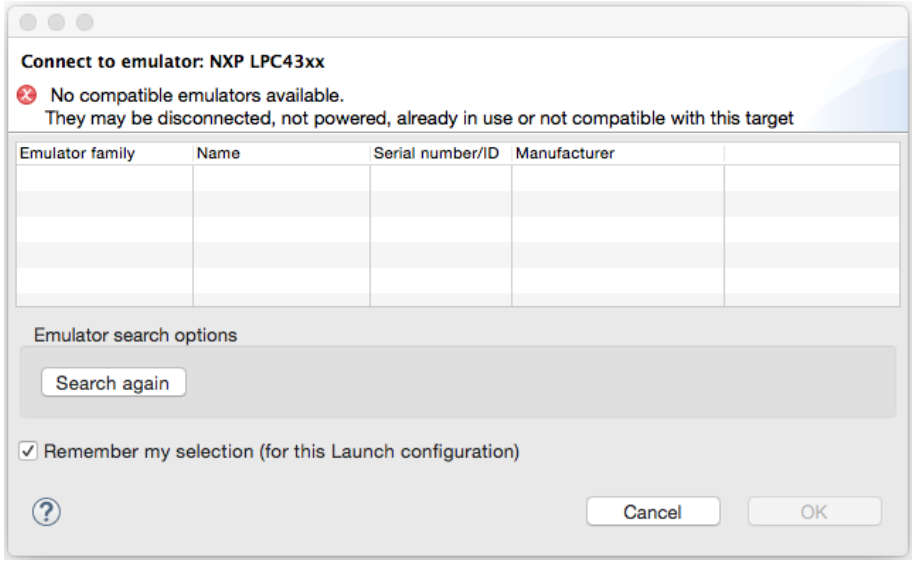


Figure 4.7. LPC-Link2 no longer connected

This might have been because you had forgotten to connect the probe, in which case connect it to your computer and select **Search again**.

The tools will then go and search for appropriate emulators. In Figure 4.8 LPC-Link2 has been detected again.

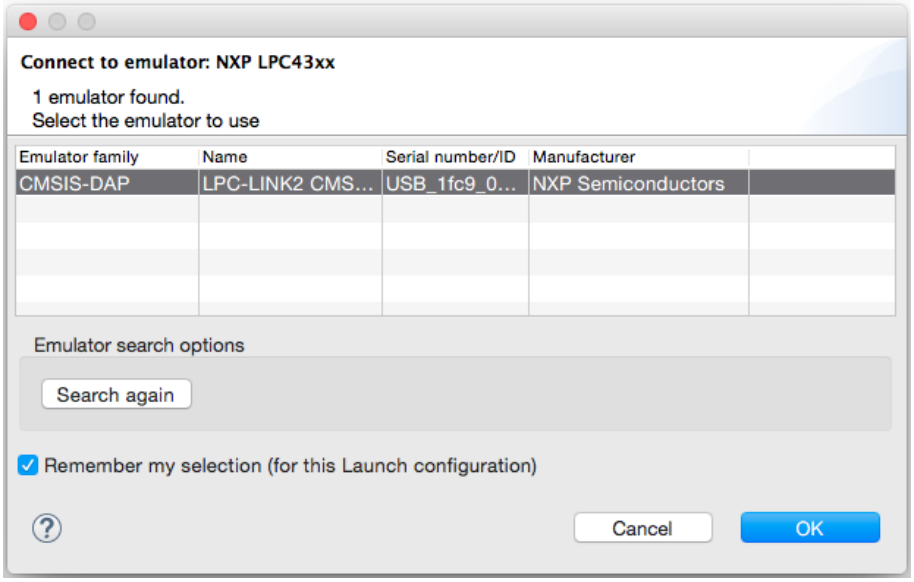


Figure 4.8. Attached probes – LPC-Link2

Notes:

- The “Remember my selection” option is enabled by default in the Debug Emulator Selection Dialog, and will cause the selected probe to be stored in the launch configuration for the current configuration (typically Debug or Release) of the current project. You can thus remove the probe selection at any time by simply deleting the launch configuration.

- You will need to select a probe for each project that you debug within a workspace (as well as for each configuration within a project).

- Storing the selected emulator (probe) in the debug configuration helps to improve debug startup time, It is also possible to turn off support for various debug emulators, which can further improve debug startup times. This can be configured on a workspace basis via the menu

```
(Windows or Linux hosts)
     Window -> Preferences -> LPCXpresso -> Debug Emulator Selection
(Mac OS X hosts)
     LPCXpresso -> Preferences -> LPCXpresso -> Debug Emulator Selection
```

4.4.2 Controlling execution

When you start a debug session, and if necessary you have selected the appropriate probe to connect to, your application is automatically downloaded to the target, a default breakpoint is set on the first instruction in `main()`, the application is started (by simulating a processor reset), and code is executed until the default breakpoint is hit.

Program execution can now be controlled using the common debug control buttons, as listed in Table 4.1, which are displayed on the global toolbar. The call stack is shown in the Debug View, as in Figure 4.9.

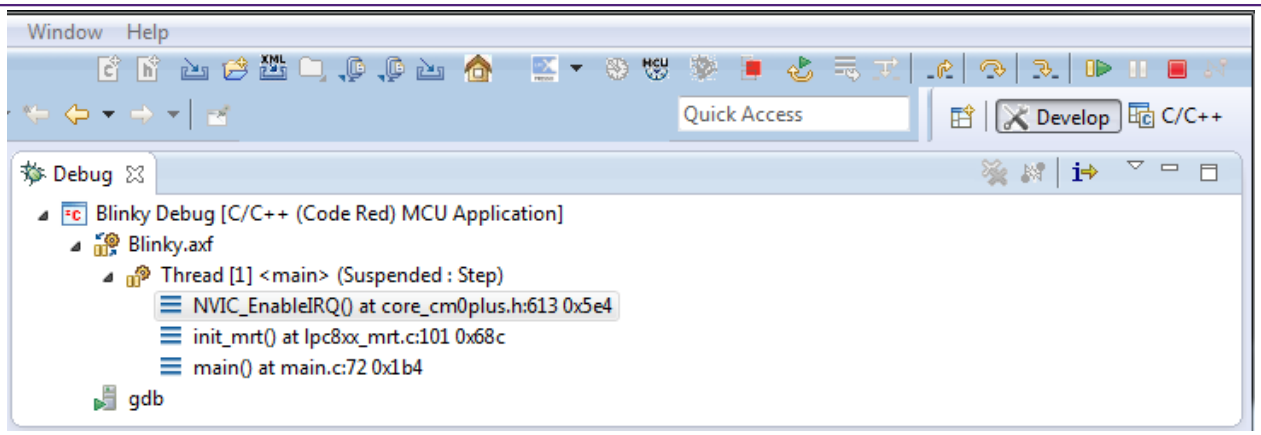


Figure 4.9. Debug Controls

Table 4.1. Program execution controls

Button	Description	Keyboard Shortcut
	Restart program execution (from reset)	
	Run/Resume the program.	F8
	Step Over C/C++ line.	F6
	Step into a function.	F5
	Return from a function	F7
	Stop the debugger.	Ctrl + F2
	Pause Execution of the running program.	
	Show disassembled instructions.	

Note – In LPCXpresso IDE v4 and earlier, the common debug control commands were found on the Debug View's local toolbar, rather than on the global toolbar. Moving them to the global toolbar allows the Debug View to be minimized when not needed – allowing more space on screen for other views to be displayed. You can choose to display the debug controls in the Debug View by selecting the "Show Debug Toolbar" option in the Debug View's "View Menu" (the downward-facing triangle on the top right of the view).

Setting a breakpoint

To set a breakpoint, simply double-click on the margin area of the line you wish to set a breakpoint on (before the line number).

Restart application

If you hit a breakpoint or pause execution and want to restart execution of the application from the beginning again, you can do this using the **Restart** button.

Stopping debugging

To stop debugging just press the **Stop** button.

If you are debugging using the **Debug Perspective**, then to switch back to **C/C++ Perspective** when you stop your debug session, just click on the **C/C++** tab in the upper right area of the LPCXpresso IDE (as shown in Figure 3.2).

5. Creating Projects using the Wizards

The LPCXpresso IDE includes many project templates to allow the rapid creation of correctly configured projects for specific MCUs.

5.1 Creating a project using the wizard

Click on the **New project...** option in the **Start here** tab of the **Quickstart Panel** to open up the Project Creation Wizard. Now select the MCU family for which you wish to create a new project. Note that in some cases a number of families of MCUs are grouped together at a top level – just open up the appropriate “expander” to get to the specific part family that you require. For example, in Figure 5.1 the top level group " `LPC11 / LPC12`" is used to hold all LPC11 and LPC12 part families.

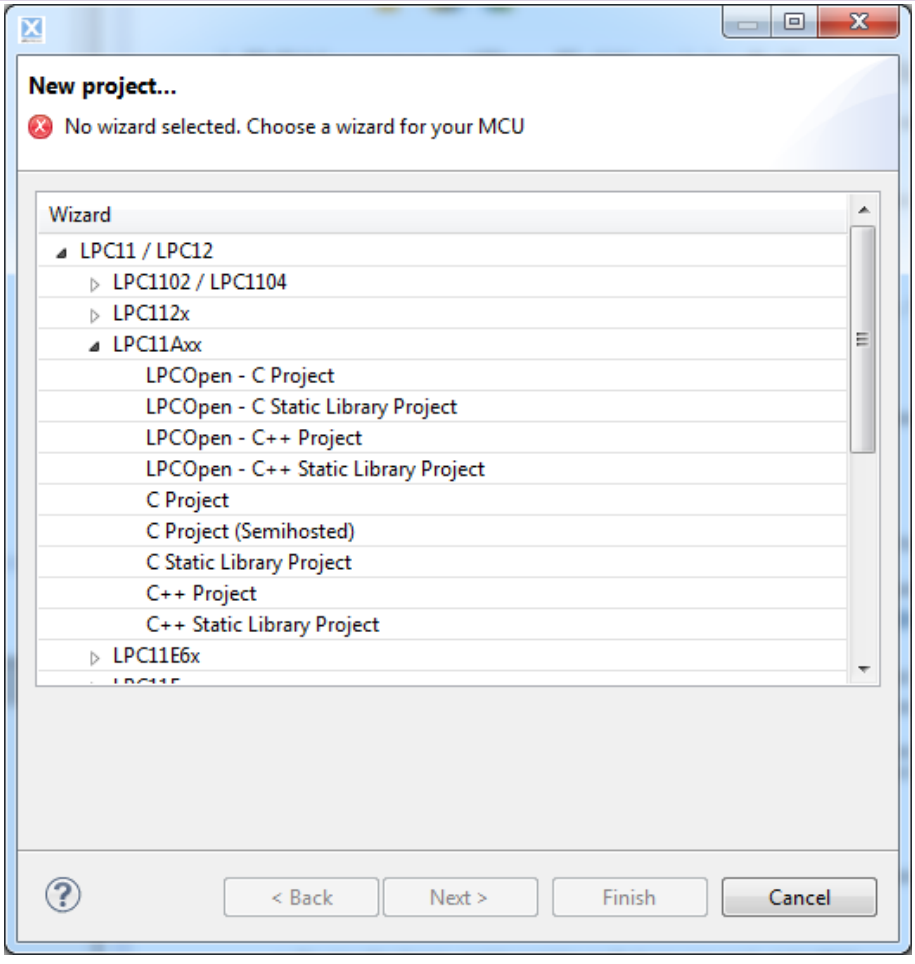


Figure 5.1. New Project: wizard selection

You can now select the type of project that you wish to create.

Most MCU families provide wizards for two forms of projects – LPCOpen and non-LPCOpen. For more details on LPCOpen, see Software drivers and examples [14] . For both forms of projects, the main wizards available are:

C Project

- Creates a simple C project, with the `main()` routine consisting of an infinite `while(1)` loop that increments a counter.

- For LPCOpen projects, code will also be included to initialize the board and enable an LED.

C++ Project

- Creates a simple C++ project, with the `main()` routine consisting of an infinite `while(1)` loop that increments a counter.

- For LPCOpen projects, code will also be included to initialize the board and enable an LED.

C Static Library Project

- Creates a simple static library project, containing a source directory, and optionally a directory to contain include files. The project will also contain a “liblinks.xml” file, which can be used by the smart update wizard on the context sensitive menu to create links from application projects to this library project. For more details, please see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/creating-linking-library-projects

C++ Static Library Project

- Creates a simple (C++) static library project, like that produced by the C Static Library Project wizard, but with the tools set up to build C++ rather than C code.

The non-LPCOpen wizards also include a further wizard:

Semihosting C Project

- Creates a simple “Hello World” project, with the `main()` routine containing a `printf()` call, which will cause the text to be displayed within the Console View of the LPCXpresso IDE. This is implemented using “semihosting” functionality. For more details, please see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/using-printf

Once you have selected the appropriate project wizard, you will be able to enter the name of your new project.

Then you will need to select the actual MCU that you will be targeting for this project. It is important to ensure that the MCU you select matches the MCU that you will be running your application on. This makes sure that appropriate compiler and linker options are used for the build, as well as correctly setting up the debug connection.

Finally you will be presented with one or more “Options” pages that provide the ability to set a number of project-specific options. The options presented will depend upon which MCU you are targeting and the specific wizard that you selected, and may also change between versions of the LPCXpresso IDE. Note that if you have any doubts over any of the options, then we would normally recommend leaving them set to their default values.

The following sections detail some of the options that you may see when running through a wizard.

5.1.1 LPCOpen Library Project Selection

When creating an LPCOpen based project, the first option page that you will see is the LPCOpen library selection page.

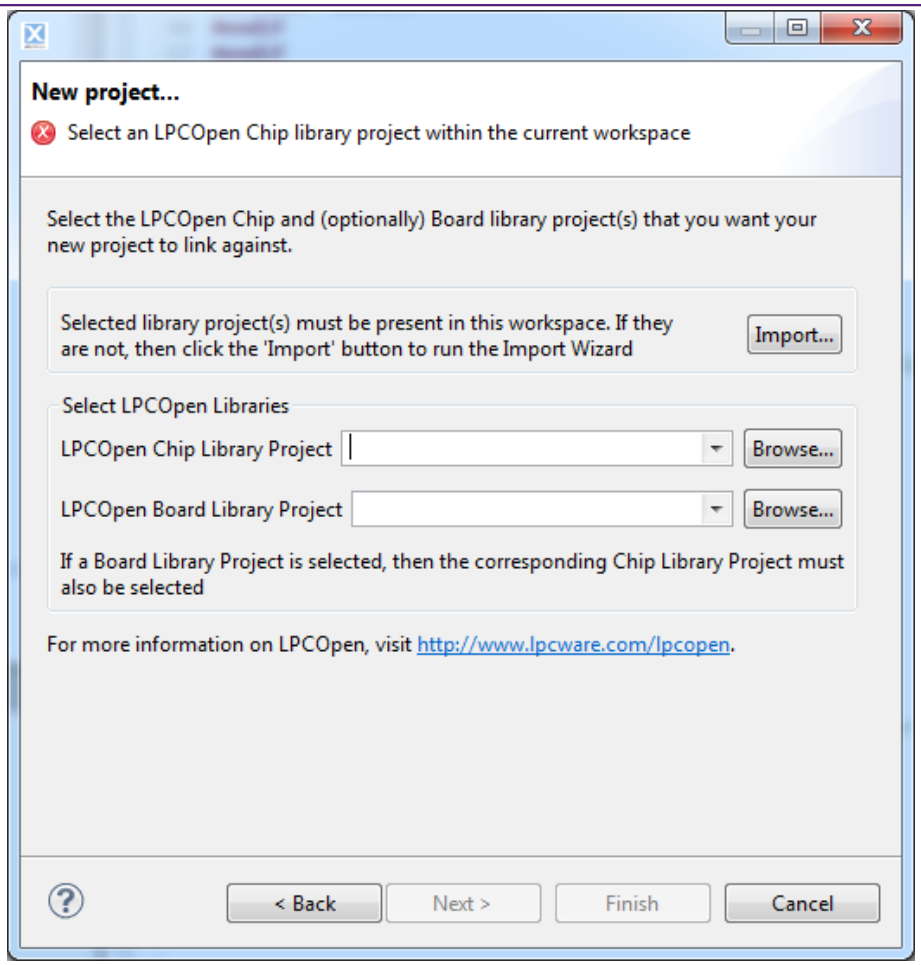


Figure 5.2. LPCOpen library selection

This page allows you to run the 'Import wizard' to download the LPCOpen bundle for your target MCU/board from the NXP LPCware website and import it into your workspace, if you have not already done so.

You will then need to select the LPCOpen Chip library for your MCU using the workspace browser (and for some MCU's an appropriate value will also be available from the drop down next to the Browse button). Note that the wizard will not allow you to continue until you have selected a library project that exists within the workspace.

Finally, you can optionally select the LPCOpen Board library for the board that your MCU is fitted to using the workspace browser (and again in some cases, an appropriate value may also be available from the drop down next to the Browse button). Although selection of a board library is optional, it is recommended that you do this in most cases.

5.1.2 CMSIS-CORE selection

For backwards compatibility reasons, the non-LPCOpen wizards for many parts provide the ability to link a new project with a CMSIS-CORE library project. The CMSIS-CORE portion of ARM's **Cortex Microcontroller Software Interface Standard** (or **CMSIS**) provides a defined way of accessing MCU peripheral registers, as well as code for initializing an MCU and accessing various aspects of functionality of the Cortex CPU itself. The LPCXpresso IDE typically provides support for CMSIS through the provision of CMSIS library projects. CMSIS-CORE library projects can be found in the Examples directory of your LPCXpresso IDE installation.

Generally, if you wish to use CMSIS-CORE library projects, you should use `CMSIS_CORE_<partfamily>` (which use components from ARM's CMSIS v3.20 specification). LPCXpresso IDE does in some cases provide libraries based on early versions of the CMSIS specification with names such as `CMSISv1p30_<partfamily>`, but these are not recommended for use in new projects.

The CMSIS library option within the LPCXpresso IDE allows you to select which (if any) CMSIS-CORE library you want to link to from the project that you are creating. Note that you will need to import the appropriate CMSIS-CORE library project into the workspace before the wizard will allow you to continue.

For more information on CMSIS and its support in the LPCXpresso IDE, please see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/cmsis-support .

Note - the use of LPCOpen instead of CMSIS-CORE library projects is recommended in most cases for new projects. (In fact LPCOpen actually builds on top of many aspects of CMSIS-CORE.) For more details see Software drivers and examples [14]

5.1.3 CMSIS DSP library selection

ARM's **Cortex Microcontroller Software Interface Standard** (or **CMSIS**) specification also provides a definition and implementation of a DSP library. The LPCXpresso IDE provides prebuilt library projects for the CMSIS DSP library for Cortex-M0/M0+, Cortex-M3 and Cortex-M4 parts, though a source version of it is also provided within the LPCXpresso IDE Examples.

Note - The CMSIS DSP library can be used with both LPCOpen and non-LPCOpen projects.

5.1.4 Peripheral Driver selection

For some parts, one or more peripheral driver library projects may be available for the target MCU from within the Examples area of your LPCXpresso IDE installation. The non-LPCOpen wizards allow you to create appropriate links to such library projects when creating a new project. You will need to ensure that you have imported such libraries from the Examples before selecting them in the wizard.

Note - the use of LPCOpen rather than these peripheral driver projects is recommended in most cases for new projects.

5.1.5 Code Read Protect

NXP's Cortex and ARM7 based MCUs provide a "Code Read Protect" (CRP) mechanism to prevent certain types of access to internal flash memory by external tools when a specific memory location in the internal flash contains a specific value. The LPCXpresso IDE provides support for setting this memory location. For more details see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/code-read-protect-crp .

5.1.6 Enable use of floating point hardware

Certain MCUs may include a hardware floating point unit (for example NXP LPC32xx, LPC407x_8x and LPC43xx parts). This option will set appropriate build options so that code is built to use the hardware floating point unit and will also cause startup code to enable the unit to be included.

5.1.7 Enable use of `Romdivide` library

Certain NXP Cortex-M0 based MCUs, such as LPC11Axx, LPC11Exx, LPC11Uxx and LPC12xx, include optimized code in ROM to carry out divide operations. This option enables the use of these Romdivide library functions. For more details see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/rom-divide .

5.1.8 Disable Watchdog

Unlike most MCUs, NXP's LPC12xx MCUs enable the watchdog timer by default at reset. This option disables this default behavior. For more details, please see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/lpc12-watchdog .

5.1.9 LPC1102 ISP Pin

The provision of a pin to trigger entry to NXP's ISP bootloader at reset is not hardwired on the LPC1102, unlike other NXP MCUs. This option allows the generation of default code for providing an ISP pin. For more information, please see NXP's application note, AN11015, "Adding ISP to LPC1102 systems".

5.1.10 Redlib Printf options

The "Semihosting C Project" wizard for some parts provides two options for configuring the implementation of printf family functions that will get pulled in from the Redlib C library:

- Use non-floating-point version of printf
  - If your application does not pass floating point numbers to `printf()` family functions, you can select a non-floating-point variant of printf. This will help to reduce the code size of your application.
  - For MCUs where the wizard does not provide this option, you can cause the same effect by adding the symbol `CR_INTEGER_PRINTF` to the project properties.
- Use character- rather than string-based printf
  - By default `printf()` and `puts()` make use of `malloc()` to provide a temporary buffer on the heap in order to generate the string to be displayed. Enable this option to switch to using "character-by-character" versions of these functions (which do not require additional heap space). This can be useful, for example, if you are retargeting printf() to write out over a UART – since in this case it is pointless creating a temporary buffer to store the whole string, only to then print it out over the UART one character at a time.
  - For MCUs where the wizard does not provide this option, you can cause the same effect by adding the symbol `CR_PRINTF_CHAR` to the project properties.

Note that if you only require the display of fixed strings, then using `puts()` rather than `printf()` will noticeably reduce the code size of your application.

5.1.11 Project created

Having selected the appropriate options, you can then click on the Finish button, and the wizard will create your project for you, together with appropriate startup code and a simple `main.c` file. Build options for the project will be configured appropriately for the MCU that you selected in the project wizard.

You should then be able to build and debug your project, as described in Section 4.3 and Section 4.4.

6. Memory Editor and User Loadable Flash Driver mechanism

6.1 Introduction

By default, the LPCXpresso IDE provides a standard memory layout for known MCUs. This works well for parts with internal flash and no external memory capability, as it allows linker scripts to be automatically generated for use when building projects, and gives built-in support for programming flash as well as other debug capabilities.

In addition, the LPCXpresso IDE supports the editing of the target memory layout used for a project. This allows for the details of external flash to be defined or for the layout of internal RAM to be reconfigured. In addition, it allows a flash driver to be allocated for use with parts with no internal flash, but where an external flash part is connected.

6.2 New in LPCXpresso v8.x Support for Multiple Flash regions

It is now possible to specify a flash driver for each region of defined flash memory. Now if a project is created that makes use of more than one flash region, including regions of different flash types, then LPCXpresso can automatically program the regions specified in a single operation.

For MCUs such as the LPC1800 and LPC4300 series, a typical use case could be to create an application to run from the MCU's internal flash that makes use of collateral in external SPI flash.

6.3 Memory Editor

The Memory Editor is accessed via the MCU settings dialog, which can be found at:

Project Properties -> C/C++ Build -> MCU settings

This lists the memory details for the selected MCU, and will, by default, display the memory regions that have been defined by the LPCXpresso IDE itself.

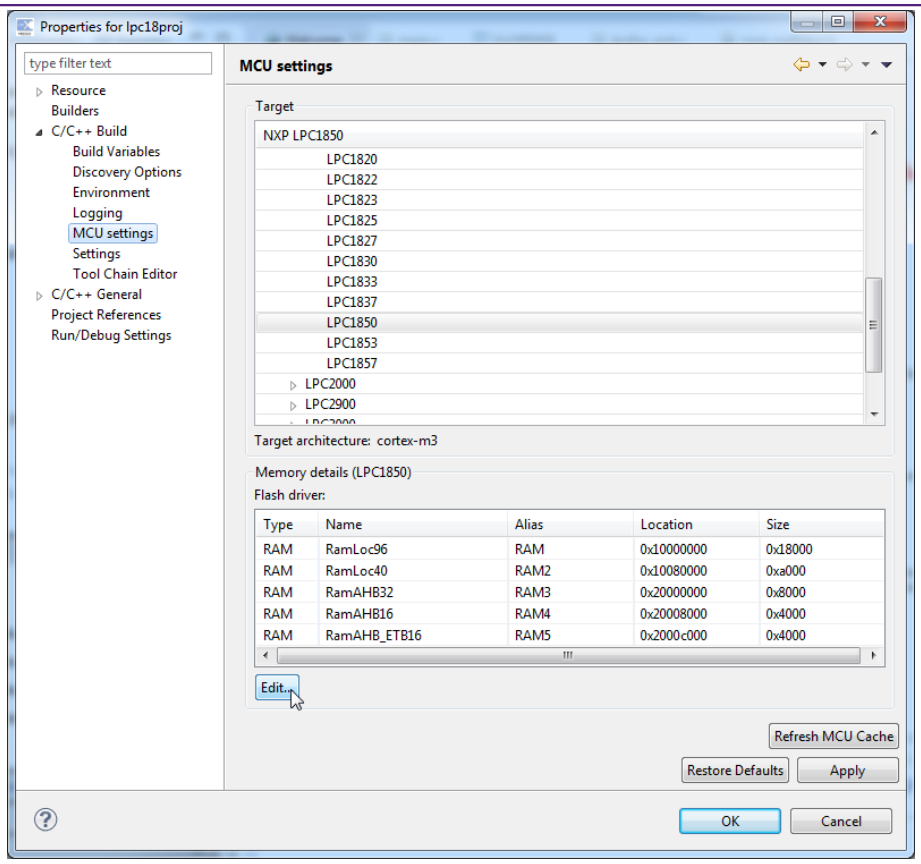


Figure 6.1. LPC1850 default memory regions

6.3.1 Editing a memory configuration

In the example below, we will show how the default memory configuration for an LPC4337 can be changed. Selecting the **Edit...** button will launch the **Memory configuration editor** dialog — see Figure 6.2.

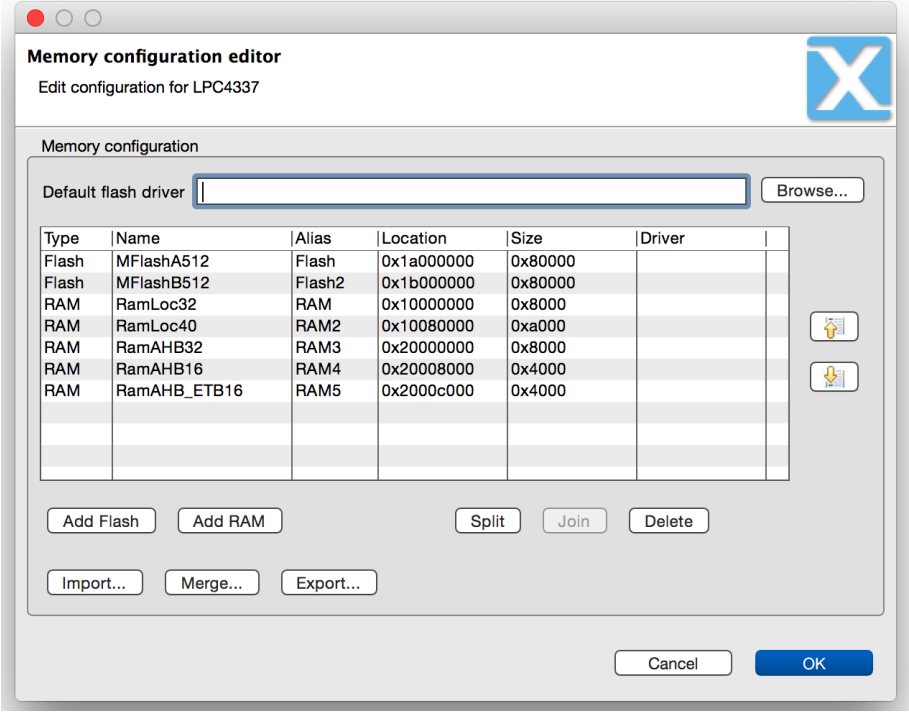


Figure 6.2. Memory configuration editor

Known blocks of memory, with their type, base location, and size are displayed. Entries can be created, deleted, etc by using the provided buttons — see Figure 6.2

Table 6.1. Memory editor controls

Button	Details
Add Flash	Add a new memory block of the appropriate type.
Add RAM	Add a new memory block of the appropriate type.
Split	Split the selected memory block into two equal halves.
Join	Joins the selected memory block with the following block (if the two are contiguous).
Delete	Delete the selected memory block.
Import	Import a memory configuration that has been exported from another project, overwriting the existing configuration
Merge	Import a partial memory configuration from file, merging it with the existing memory configuration. This allows you, for example, to add an external flash bank definition to an existing project.
Export	Export a memory configuration for use in another project.
Up / Down	Reorder memory blocks. This is important: if there is no flash block, then code will be placed in the first RAM block, and data will be placed in the block following the one used for the code (regardless of whether the code block was RAM or Flash).
Browse(Flash driver)	Select the appropriate driver for programming the flash memory specified in the memory configuration. This is only required when the flash memory is external to the MCU. Flash drivers for external flash must have a ".cfx" file extension and must be located in the \bin\flash subdirectory of the LPCXpresso IDE installation. For more details see User loadable flash drivers [33] .

The name, location, and size of this new region can be edited in place. Note that when entering the size of the region, you can enter full values in decimal, hex (by prefixing with `0x`), or by specifying the size in kilobytes or megabytes. For example:

- To enter a region size of 32KB, enter `32768`, `0x8000` or `32k`.

- To enter a region size of 1MB, enter `0x100000` or `1m`.

Note that memory regions must be located on four-byte boundaries, and be a multiple of four bytes in size.

The screenshot in Figure 6.3 shows the dialog after the “Add Flash” button has been clicked.

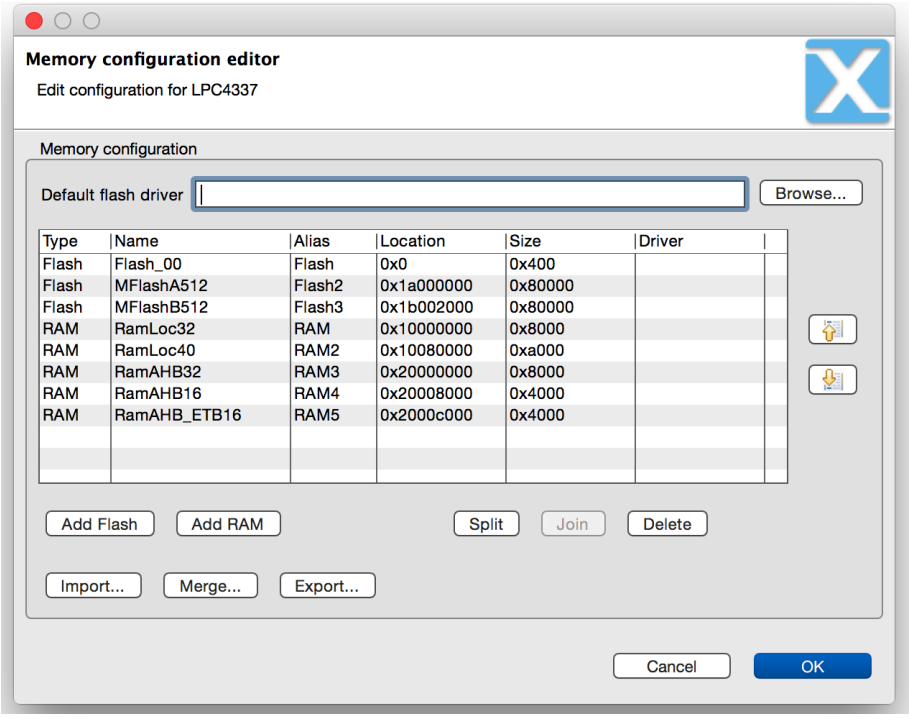


Figure 6.3. Effect of Add Flash

After updating the memory configuration, click **OK** to return to the MCU settings dialog, which will be updated to reflect the new configuration — see Figure 6.4.

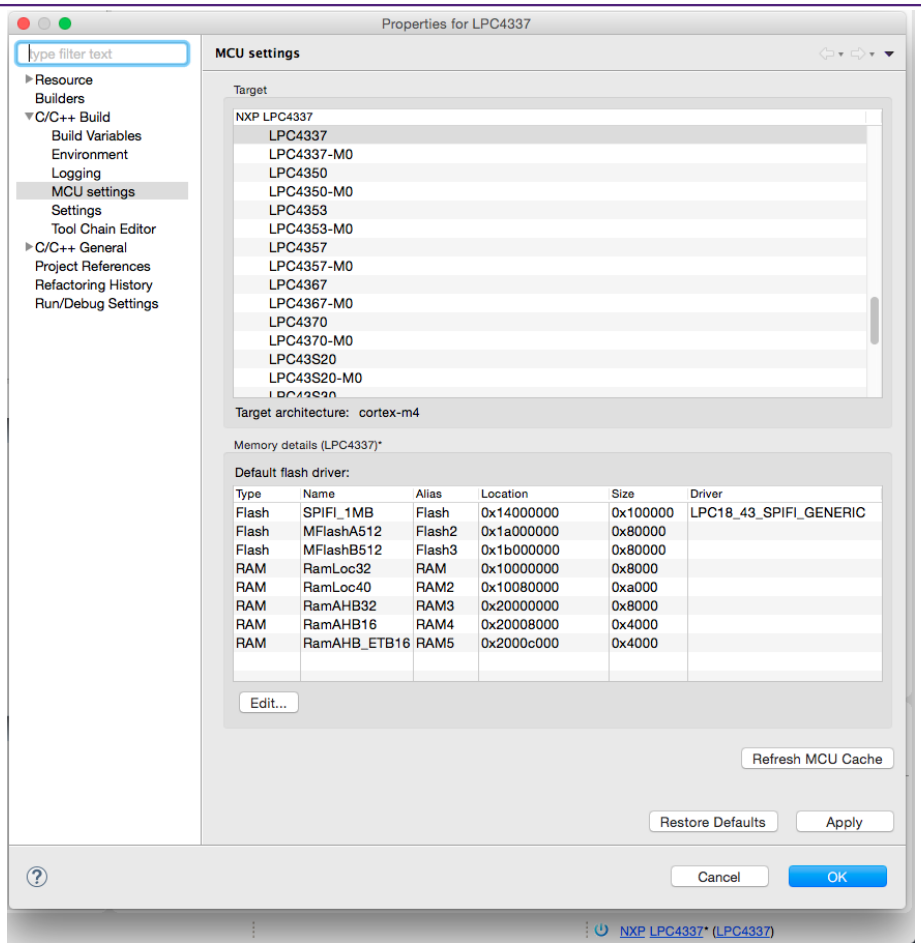


Figure 6.4. Updated MCU Settings

Here you can see that the region has been named SPIFI_1MB, and the default flash driver has been deleted and the Generic SPIFI driver selected for the newly created SPIFI_1MB region.

Note that once the memory details have been modified, the selected MCU as displayed on the LPCXpressoIDE “Status Bar” (at the bottom of the IDE window) will be displayed with an asterisk (*) next to it. This provides an indication that the MCU memory configuration settings for the selected project have been modified.

6.3.2 Restoring a memory configuration

To restore the memory configuration of a project back to the default settings, simply reselect the MCU type, or use the “Restore Defaults” button, on the MCU Settings properties page.

6.3.3 Copying memory configurations

Memory configurations can be exported for import into another project. Use the Export and Import buttons for this purpose.

6.4 User loadable flash drivers

Flash drivers for external flash must have a `.cfx` file extension and must be located in the `/bin/Flash` subdirectory of the LPCXpresso IDE installation.

Many flash drivers supplied with LPCXpresso IDE are for SPIFI flash devices. SPIFI memory is located in the MCU's memory map at `0x14000000`.

Earlier versions of LPCXPresso IDE provided a number of dedicated SPIFI flash drivers (targeted at one or more specific SPIFI devices). This meant that it was the user's responsibility to select the correct SPIFI flash driver to match the device fitted to their target hardware.

LPCXpresso IDE v7.9.0 introduced an improved flash driver mechanism for SPIFI flash. Now a project targeting SPIFI flash only has to specify a single Generic flash driver : "LPC18_43_SPIFI_GENERIC.cfx". When a flash programming operation is performed, this driver will first interrogate the SPIFI device and determine its type, size and configuration which are then reported back to the host debug driver. Using this information the correct optimised programming routines for the SPIFI device detected can then be used.

For a more complete list of supplied SPIFI flash drivers for the NXP LPC1800 and LPC4300 series of MCUs, please see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/lpc18-lpc43-spifi-flash-drivers

A small number of example flash drivers are also provided for parallel flash (connected to the MCU via the EMC). Such devices are located in the MCU's memory map at `0x1c000000`.

For example the `LPC18_43_Diolan_S29AL016J70T.cfx` driver is for use in programming the external 2MB parallel flash fitted to the Diolan LP1850-DB1 and LPC4350-DB1 boards.

Source code for external flash drivers is also provided in the `/Examples/FlashDrivers` subdirectory of the LPCXpresso IDE installation. These projects can be used as the basis of writing your own flash drivers for other devices.

Note that the `/bin/Flash` subdirectory may also contain some drivers for the built-in flash on some MCUs. It should be clear from the filenames which these are. Do not try to use these drivers for external flash on other MCUs!

6.5 Projects and Multiple Flash Regions

As we have seen earlier, LPCXpresso will automatically create memory maps to match the target MCU and select an appropriate flash driver for MCUs containing internal flash.

From LPCXpresso 8.0, there is extended support for the creation and programming of projects that span multiple flash devices. Previously only a single (default) flash driver could be specified for a project, now it is possible to specify a flash driver for each region of flash memory.

In Figure 6.5 we can see that a memory description can be created that describes both the address and size of the memories but also specifies the flash drivers needed to program each flash regions.

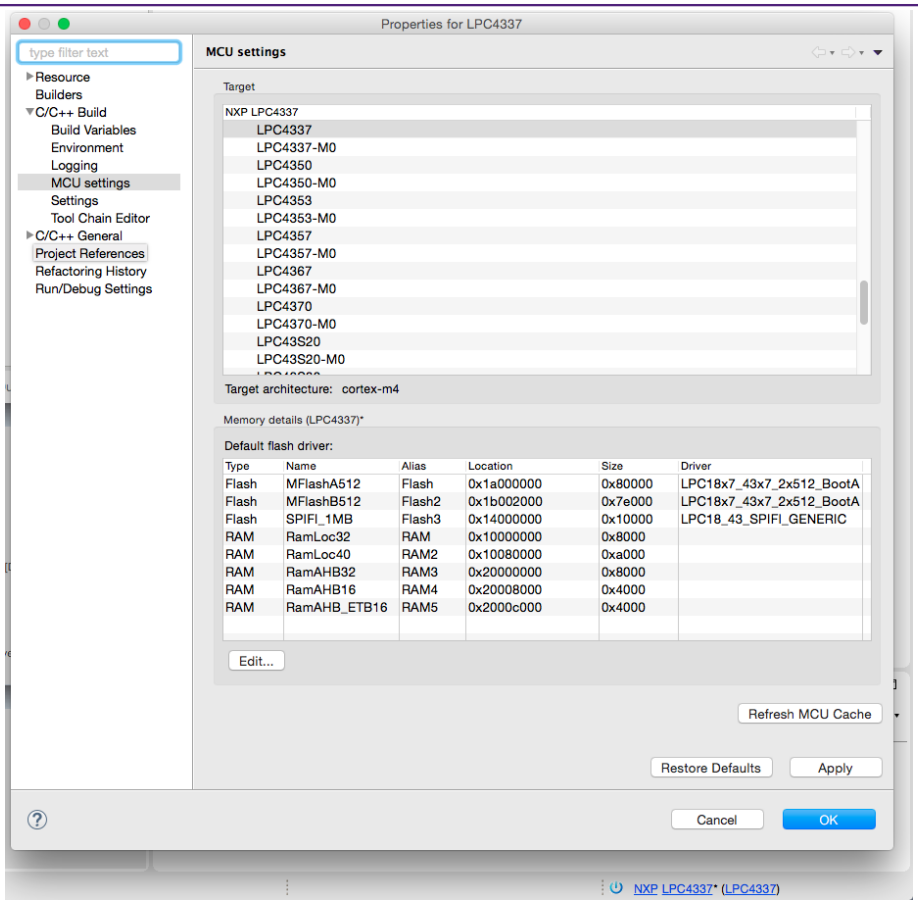


Figure 6.5. Multiflash MCU Settings

These per region flash drivers override any default flash driver if one is specified.

Note that some additional care may be required when creating projects of this type. In this example, we have chosen to boot from the flash memory at 0x1A000000 (BankA) and have offset the second Flash bank from its base address of 0x1B000000 by 8KB (where 8KB is the smallest block that can be programmed in the first 64KBs of device). This is to avoid a small risk that code or data located at the start of this flash may be misinterpreted by the MCU's bootloader.

For further information on creating project to make use of multiple banks of flash memory, please see the FAQ at:

https://www.lpcware.com/content/faq/lpcxpresso/coderodata-different-flash-blocks

6.6 Modifying memory configurations within the New Project Wizards

The New Project Wizards for LPC18xx, LPC43xx and LPC541xx parts allow the project's memory configuration to be edited from within the wizard itself. This allows you to add, delete or import memory blocks as required, as detailed in Editing a memory configuration [30], from within the project wizard itself.

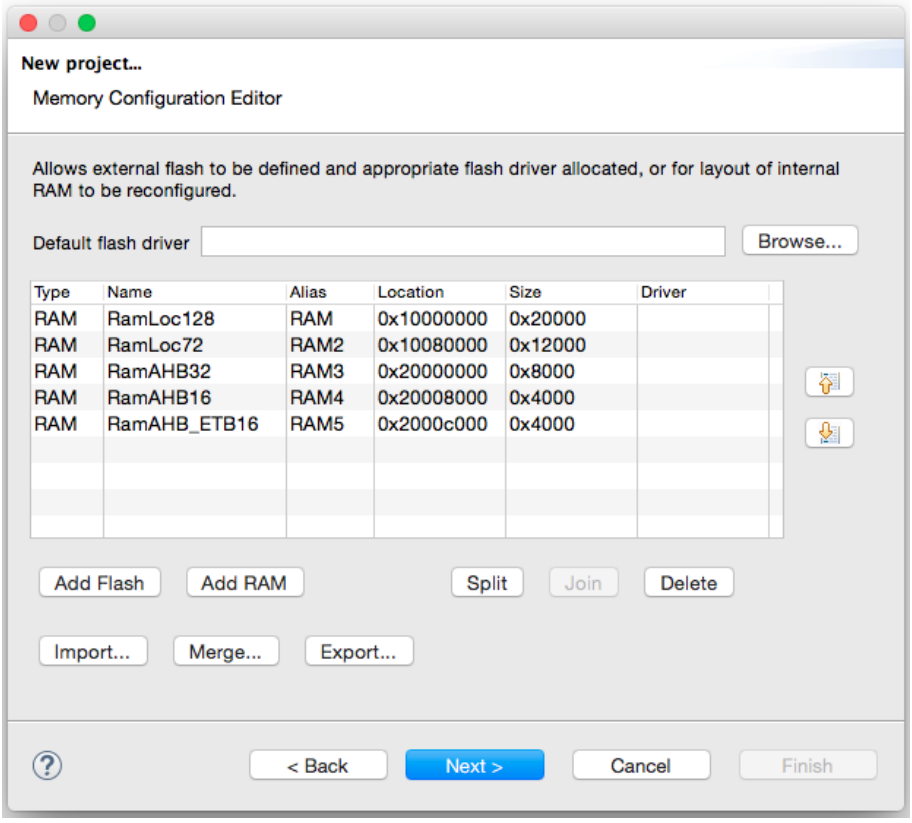


Figure 6.6. External Memory configuration within New Project Wizard

In addition, a number of memory configurations containing LPC18/43 external flash and flash driver driver definitions suitable for use with the Merge option of the memory configuration editor can be found at:

```
<install_dir>\lpcxpresso\Wizards\MemConfigs
```

Also, for backwards compatibility reasons, a number of full memory configurations, suitable for use with the Import option, can be found in the directory:

```
<install_dir>\lpcxpresso\Wizards\MemConfigs\NXP
```

but generally we would recommend the use of the ‘merge’ files instead.

7. Multicore projects

7.1 LPC43xx Multicore projects

The LPC43xx family of MCUs contain a Cortex-M4 ‘Master’ core and one (or more) Cortex-M0 ‘Slave’ cores. After a power-on or Reset, the Master core boots and is then responsible for booting the Slave core(s); hence the names Master and Slave. In reality Master and Slave only apply to the booting process; after boot, your application may treat any of the cores as the Master or Slave.

The LPCXpresso IDE allows for the easy creation of “linked” projects that support the targeting of LPC43xx Multicore MCUs. For more information on creating and using such multicore projects, please see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/lpc43xx-multicore-apps

7.2 LPC541xx Multicore projects

Some members of the LPC541xx family of MCUs contain a Cortex-M4 core and a Cortex-M0+ core (with the Cortex-M4 being the master, and the M0+ the slave) After a power-on or Reset, the Master core boots and is then responsible for booting the Slave core(s); hence the names Master and Slave. In reality Master and Slave only apply to the booting process; after boot, your application may treat any of the cores as the Master or Slave.

The LPCXpresso IDE allows for the easy creation of “linked” projects that support the targeting of LPC541xx Multicore MCUs. For more information on creating and using such multicore projects, please see the FAQ at

http://www.lpcware.com/content/faq/lpcxpresso/lpc541xx-multicore-apps

8. Trace

8.1 Trace Overview

There are three 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
- **Red Trace** - SWO using the Red Probe+ debug probe.

Note

The use of “Red Trace” (SWO Trace via Red Probe+) is now deprecated and support will be removed in a future LPCXpresso IDE release. Use SWO Trace via LPC-Link2 instead.

The latest information on trace and details about supported targets can be found on LPCWare.com at http://www.lpcware.com/content/faq/lpcxpresso/trace-overview

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

8.1.1 Instruction Trace

Instruction trace provides the ability to record a trace of executed instructions on certain Cortex-M0+, M3 and M4 based MCUs. Instruction trace data is captured and compressed by the ETM (or MTB) in real time. The amount of data that can be captured is dependent on the amount of memory allocated to the trace buffer on your target MCU.

To support the use of the Instruction Trace functionality, the target MCU must meet the following requirements:

- **Cortex-M3 and M4** based parts must implement both ARM’s Embedded Trace Macrocell (ETM) and an Embedded Trace Buffer (ETB). This means, for example, that Instruction Trace can be carried out with NXP’s LPC18xx and LPC43xx parts, but not with LPC17xx parts (which do not implement an ETB).
- **Cortex-M0+** must implement ARM’s Micro Trace Buffer (MTB). This means, for example, that Instruction Trace can be carried out with LPC8xx parts. Note – instruction trace is not supported on Cortex-M0 based parts.

Instruction trace is supported when using any supported debug probe, including LPC-Link, LPC-Link2 and Red Probe+.

8.1.2 Serial Wire Output

ARM’s Coresight debug architecture allows data to be sampled and streamed off the MCU to the host completely non-intrusively. This allows lower frequency events such as periodic PC sampling, interrupts etc to be captured and transmitted by the debug probe with no affect on 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 on top of 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 or Red Probe+. 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 restriction 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. There are two sets of these views corresponding to the two supported probes. If you are using a LPC-link2 probe you should use the **SWO Trace** views. If you are using the Red Probe+ debug probe you should use the **Red Trace** views.

The two probes provide similar functionality. **SWO Trace** is under active development and will have additional functionality added. **Red Trace** is a legacy product and is not under active development.

Table 8.1. SWO trace feature comparison

Feature	SWO Trace	Red Trace
Probe	LPC-Link2	Red Probe+
Data watch	yes*	yes
Profile	yes	yes
Interrupt Statistics	yes	yes
Graphical Interrupt trace	Pro	yes
ITM text console	yes	no
ITM Host Strings	in development	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.

8.2 SWO Trace : Views

The latest LPC-Link2 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) any memory location in real-time, without stopping the processor.

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, as per Figure 8.17.

8.2.1 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 it's 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.

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.

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

Please see this web page for further details: http://www.lpcware.com/content/faq/lpcxpresso/trace-overview

8.2.2 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.

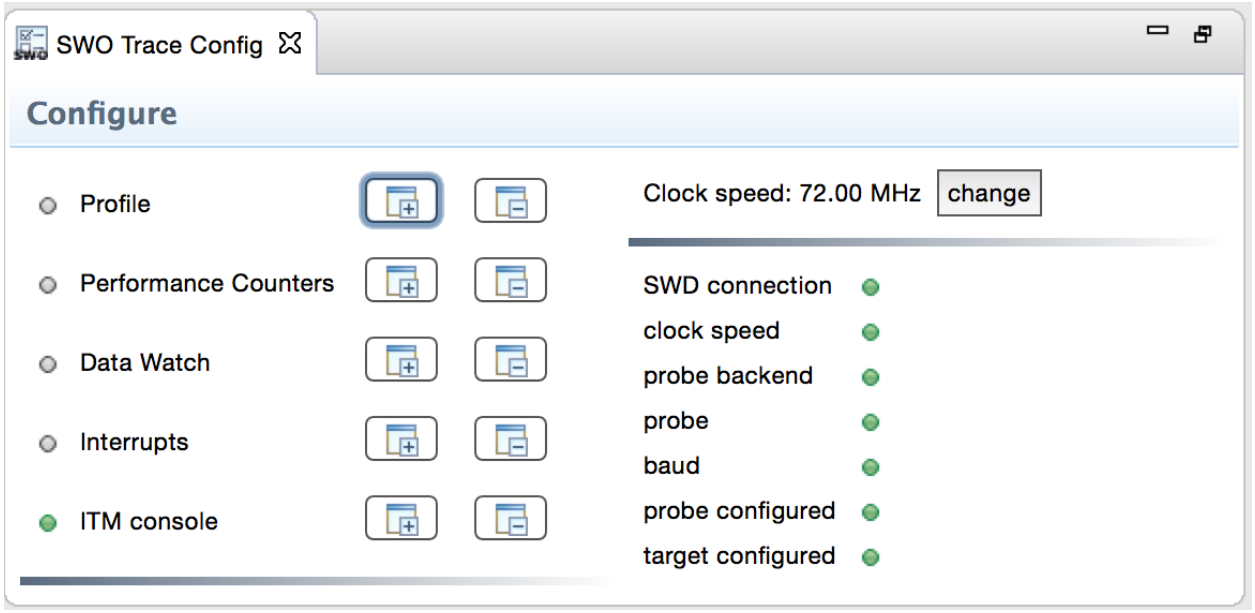


Figure 8.1. SWO Config View

8.3 SWO Trace : Profiling

8.3.1 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.

8.3.2 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 8.2. 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.

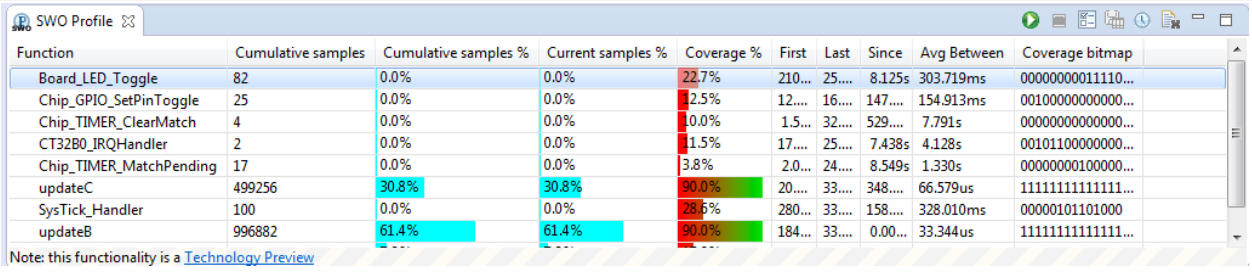


Figure 8.2. SWO Profile View

- **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 8.3.

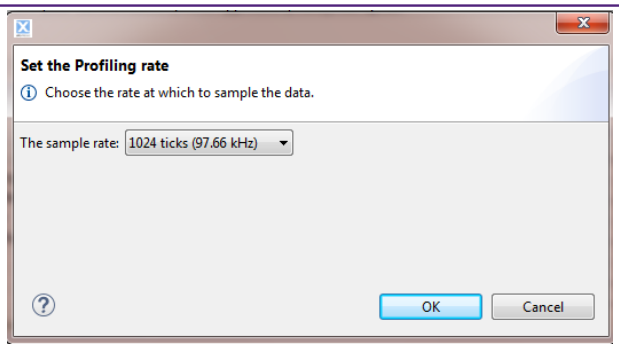


Figure 8.3. PC sample rate

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 (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.

8.4 SWO Trace : ITM

8.4.1 Overview

The ITM block provides a mechanism for sending data from your target to the debugger via the SWO stream. This communication is achieved though 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.

8.4.2 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) is 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 How to use ITM Printf FAQ on the LPCWare.com website.

8.4.3 SWO ITM console view

Data written to the ITM stimulus port 0 is presented in this view. An example screenshot is shown in Figure 8.4. 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`).

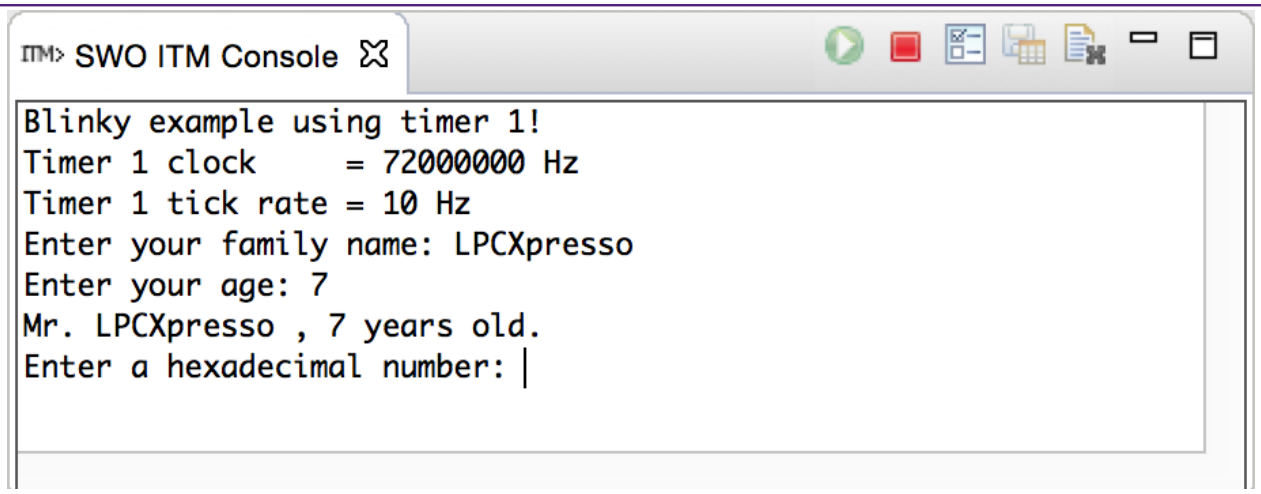


Figure 8.4. SWO ITM Console View

In addition to the standalone ITM Console view, the ITM console is also displayed as part of the standard console viewer, see Figure 8.4. 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.

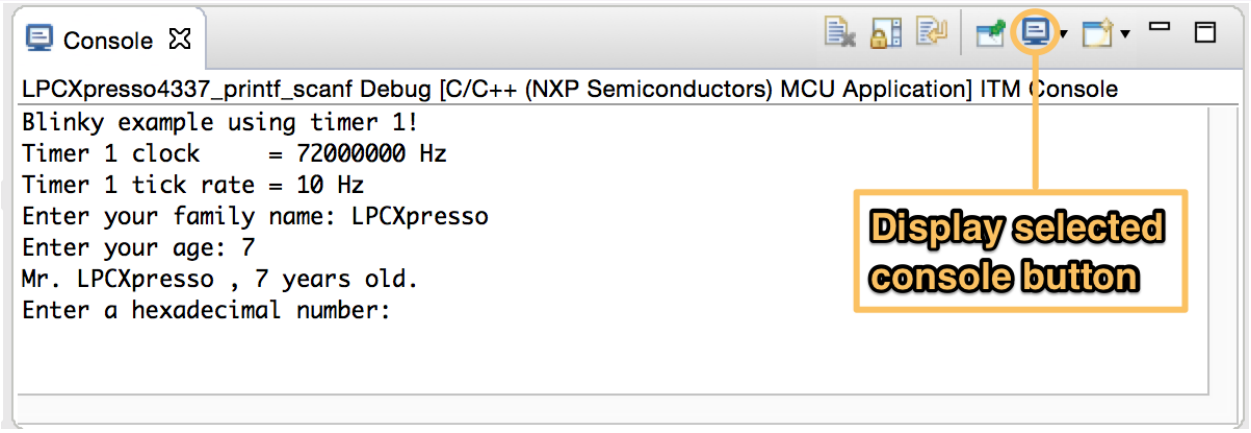


Figure 8.5. SWO ITM Console shown in the standard Console view

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.

8.5 SWO Trace : Interrupt tracing

8.5.1 Overview

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

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.

8.5.2 SWO Interrupt Statistics view

The Interrupt Statistics view displays counts and aggregated timing information for interrupt handlers. An example screenshot is shown in Figure 8.6.

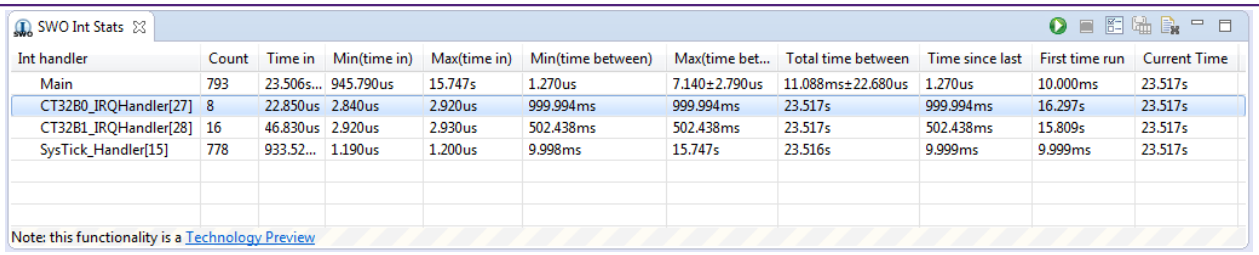


Figure 8.6. SWO Interrupt Statistics View

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.

8.5.3 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 8.7.

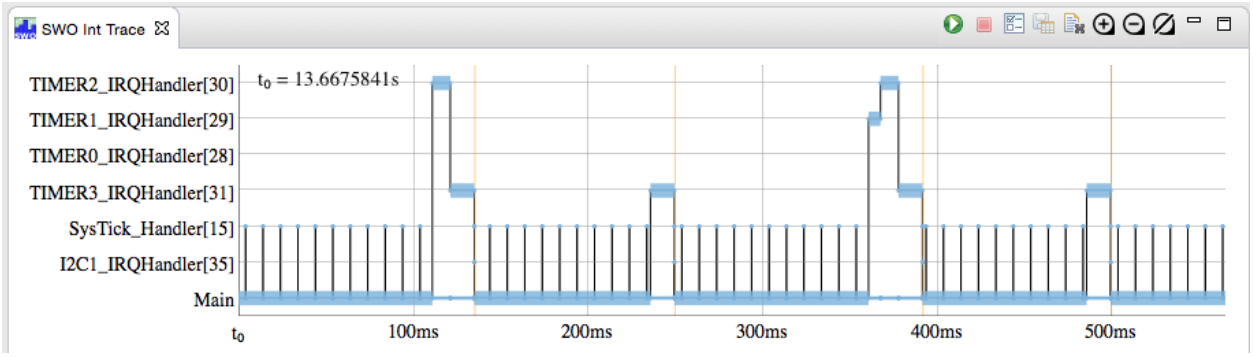


Figure 8.7. SWO Interrupt Statistics View

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

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 `t0`. The value of `t0` 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 `t0` changing as the plot moves along.

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.

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 8.8 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.

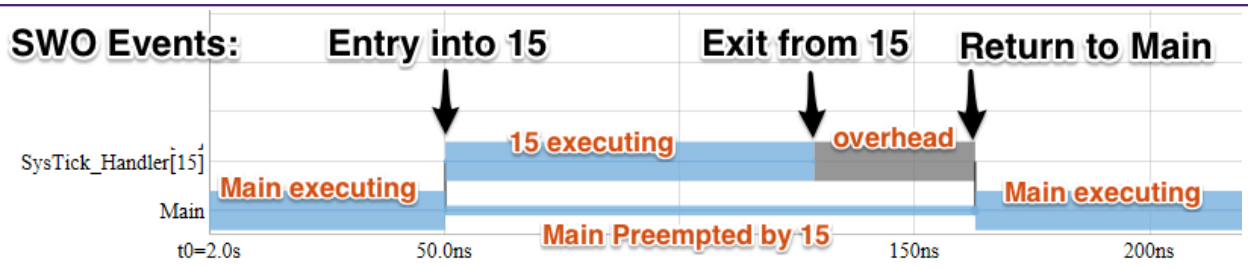


Figure 8.8. An Example exit entry and return

Tail chaining

Figure 8.9(a) shows an example of tail chaining, where one interrupt is exited and another entered without returning to `Main`. Figure 8.9(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.

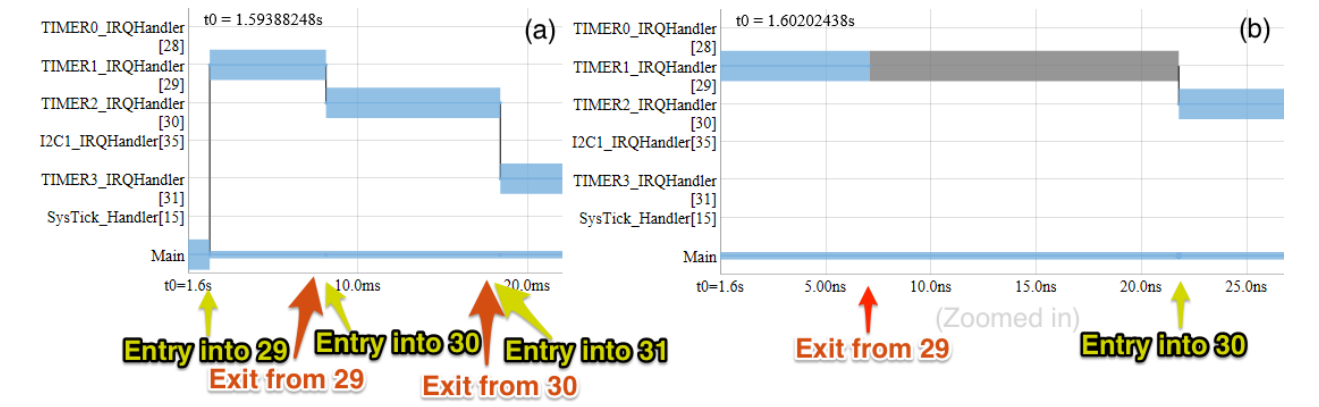


Figure 8.9. Tail chaining

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 8.10(a).

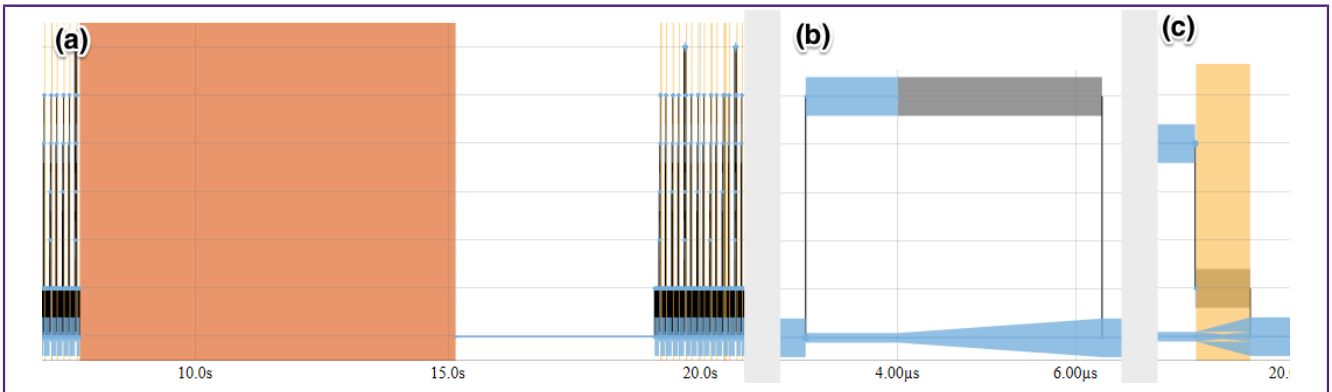


Figure 8.10. Overflows and 'late' time stamps

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 8.10(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 and these 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 8.10(c).

In addition to the overflow in Figure 8.10(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.

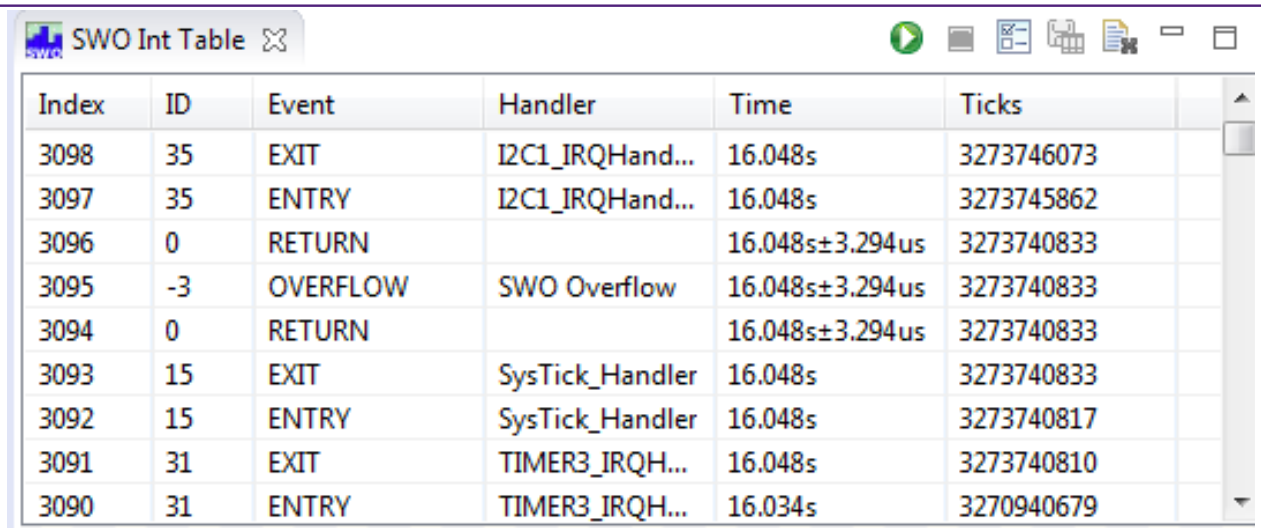
Graph buffer depth

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.

In the technology preview much larger buffer depths may result in the UI becoming very slow when plotting all points. This limitation will be addressed in a later releases.

8.5.4 SWO Interrupt Trace Table

The SWO Interrupt Trace Table view shows the collected interrupt events and their corresponding time stamps – see Figure 8.11.



Index	ID	Event	Handler	Time	Ticks
3098	35	EXIT	I2C1_IRQHand...	16.048s	3273746073
3097	35	ENTRY	I2C1_IRQHand...	16.048s	3273745862
3096	0	RETURN		16.048s±3.294us	3273740833
3095	-3	OVERFLOW	SWO Overflow	16.048s±3.294us	3273740833
3094	0	RETURN		16.048s±3.294us	3273740833
3093	15	EXIT	SysTick_Handler	16.048s	3273740833
3092	15	ENTRY	SysTick_Handler	16.048s	3273740817
3091	31	EXIT	TIMER3_IRQH...	16.048s	3273740810
3090	31	ENTRY	TIMER3_IRQH...	16.034s	3270940679

Figure 8.11. SWO Interrupt Trace Table View

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.

8.6 SWO Trace : Data Watch Trace

8.6.1 Overview

This view provides the ability to monitor (and update) any memory location in real-time, without stopping the processor. In addition up to 4 memory locations can also be traced 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.

8.6.2 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 8.12.

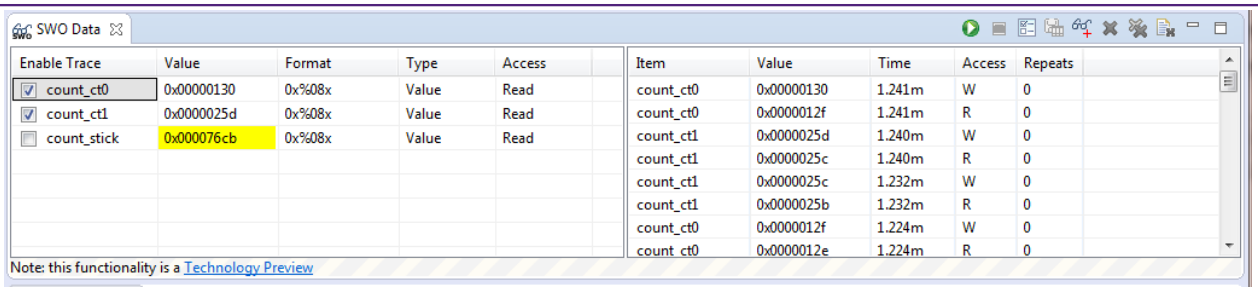


Figure 8.12. SWO Data Watch View

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 8.13.

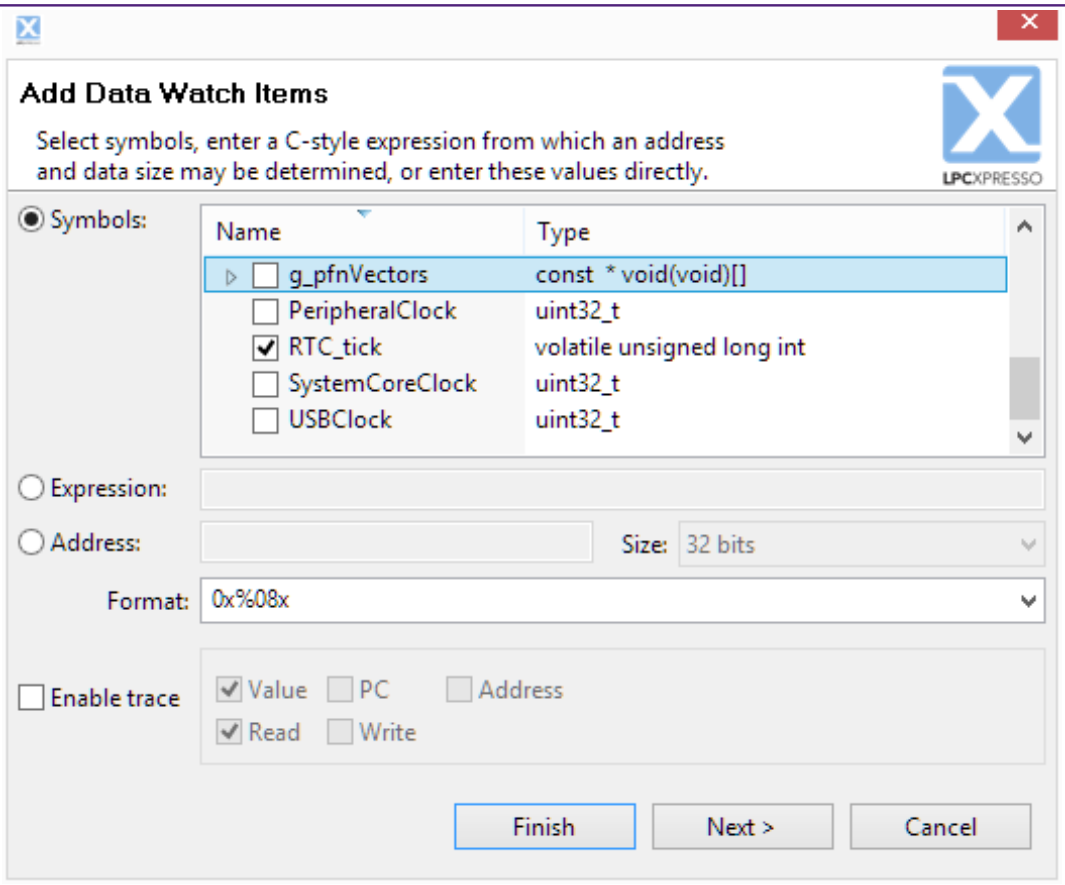


Figure 8.13. Add Data Watch Items

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

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

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

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

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

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

Item Display

As shown in Figure 8.12, the item display lists the data watch items that have been added. The following information is presented:

- **Enable Trace** - tracing of this item may be enabled or disabled using the checkbox. A maximum of 4 items may be traced at one time. Each traced item is given a color code, so that it may be picked out easily on the trace display (see Figure 8.27).
- **Value** - shows the current value of the item and may be edited to write a new value to the target. If the current value has changed since the last update, then it will be shown highlighted in yellow.
- **Format** - shows the printf-style expression used to format the value and may be edited
- **Type** - shows the trace type, which may be edited while trace is disabled:
  - **Value** - just the trace the value transferred to/from memory
  - **PC only** - just trace the PC of the instruction making the memory access
  - **PC and value** - trace the value and the PC
  - **Address** - trace the address of memory accessed
  - **Address and value** - trace the address and value
- **Access** - shows the access type, which may be edited while trace is disabled:
  - **Write** - just trace writes to the memory location
  - **Read** - just trace reads to the memory location
  - **Read & Write** - trace both reads and writes

Trace Display

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

8.7 SWO Trace : Performance Counters

8.7.1 Overview

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

8.7.2 SWO Performance Counters view

The Performance Counter view displays the target's Performance Counters as shown in Figure 8.14.

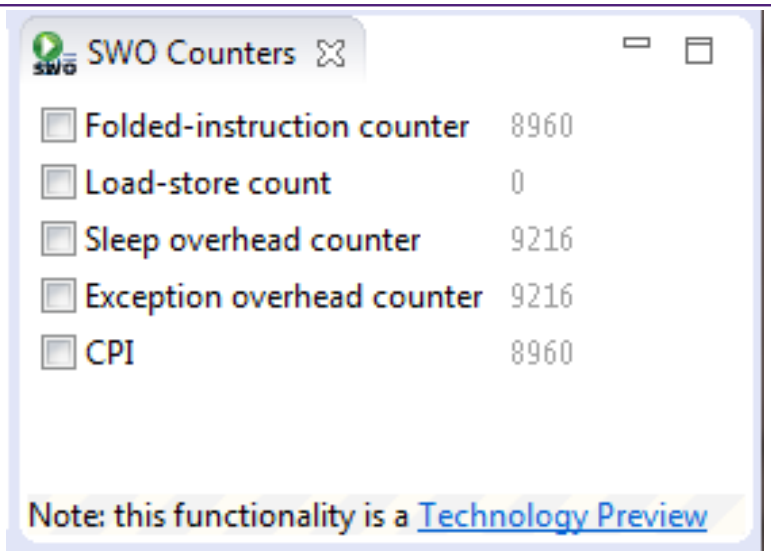


Figure 8.14. SWO Performance Counters View

The performance counters include:

- **Folded Instruction Counter**
  - Increments on any instruction that executes in zero cycles
- **Load-Store Counter**
  - 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**
  - 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**
  - 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**
  - 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.

8.8 SWO Trace: Bandwidth considerations

8.8.1 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.

8.8.2 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 loosing 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 here 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

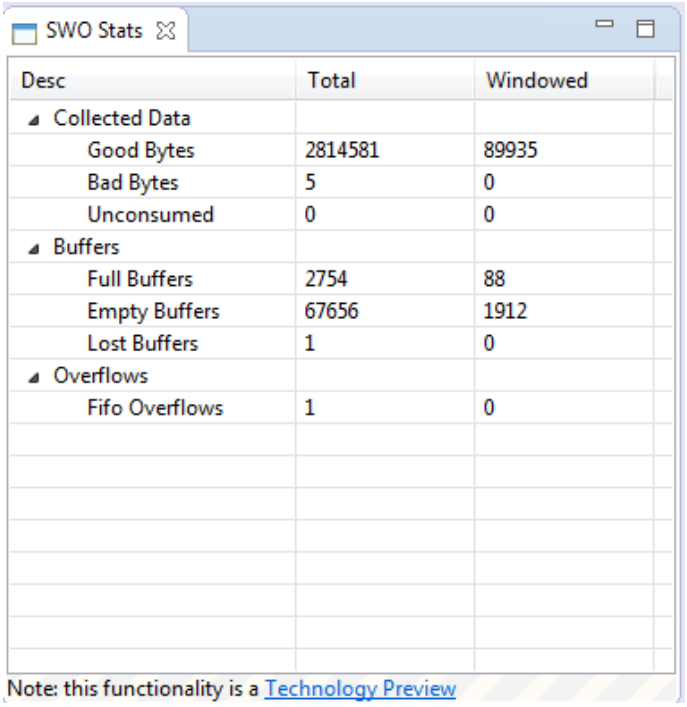


Figure 8.15. SWO Stats View

8.9 SWO Trace: Preferences

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

8.9.1 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.

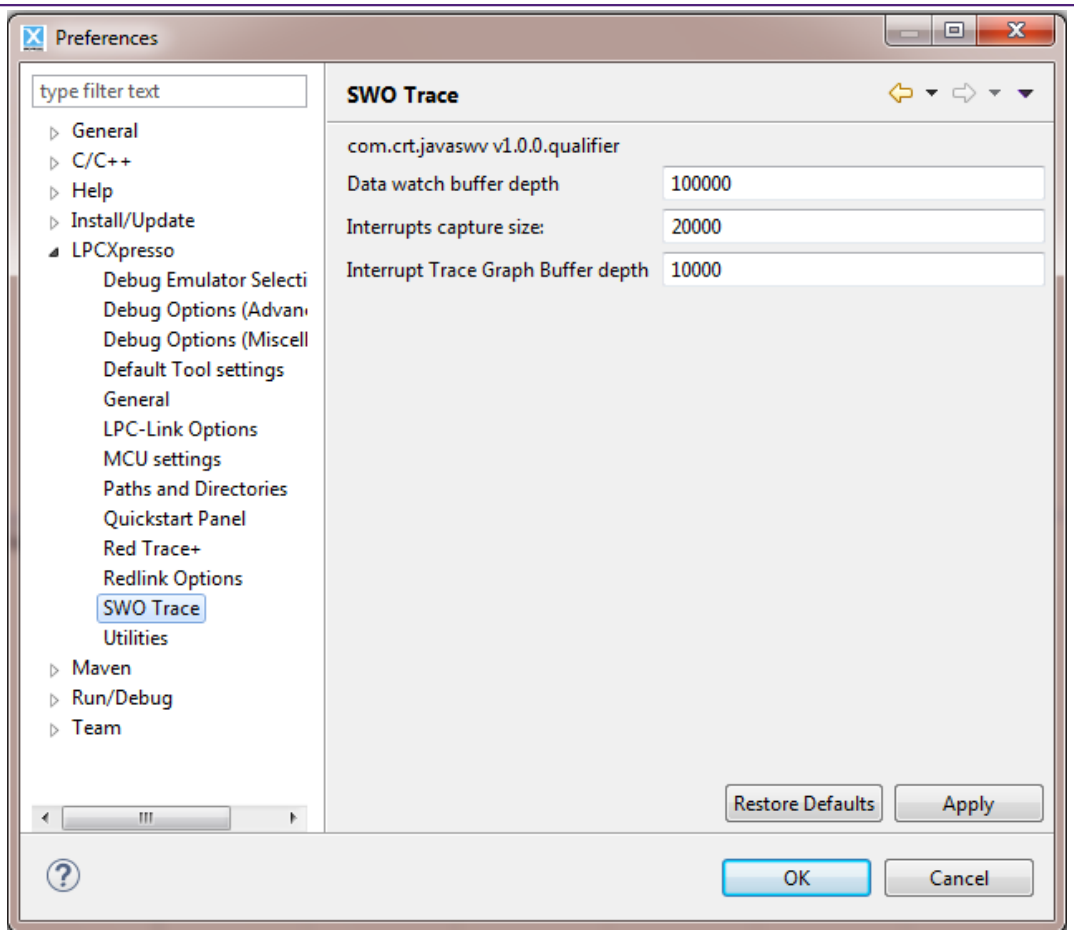


Figure 8.16. SWO Preferences

8.10 Red Trace : SWV Views

Note

The use of “Red Trace” (SWO Trace via Red Probe+) is now deprecated, and support will be removed in a future LPCXpresso IDE release. Use SWO Trace via LPC-Link2 instead.

Each trace view is presented within the **Debug Perspective** or the **Develop Perspective** by default. Their visibility can be toggle using the **toggle trace view** button.

Trace views that are not required may be closed to simplify the user interface. They may be re-opened using the **Window -> Show View -> Other...** menu item, as per Figure 8.17.

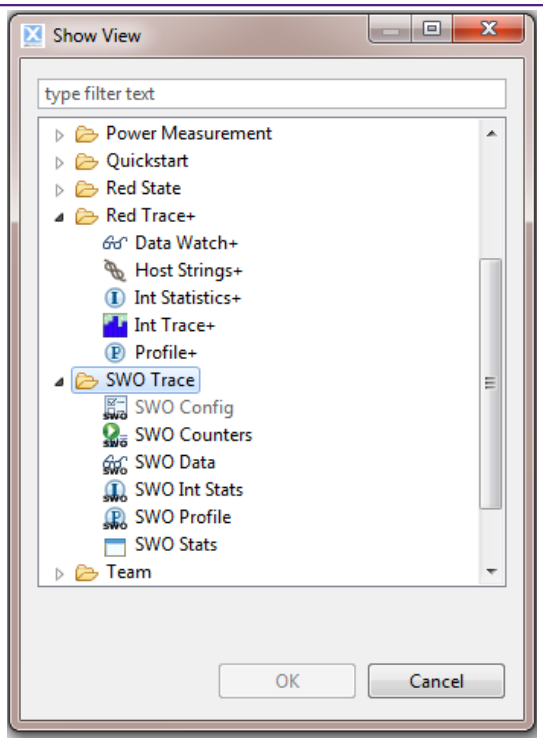


Figure 8.17. Reopening a view

The following trace views are provided:

Profile

- Provides a statistical profile of application activity.

Interrupt Statistics

- Provides counts and timing information for interrupt handlers.

Interrupt Trace

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

Data Watch

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

Host Strings

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

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

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

8.11 Red Trace : SWV Configuration

8.11.1 Starting Red Trace

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

You may start Red Trace at any time whilst debugging your program. The program does **not** have to be stopped at a breakpoint. Before the collection of data commences, Red Trace will prompt for settings related to your target processor, as per Figure 8.18.

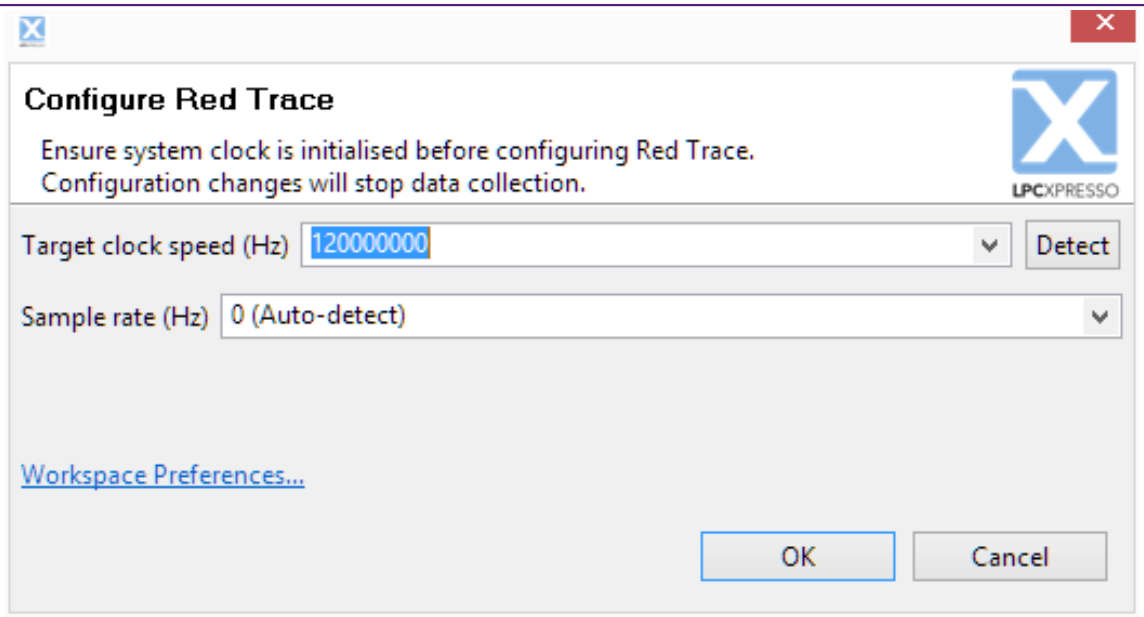


Figure 8.18. Configure Red Trace

Target Clock Speed

Due to the way the Trace data is transferred by the target processor, setting the correct clock speed within the Red 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, data will be lost, and the trace data will be meaningless.

Depending upon the target MCU and the debug probe being used, the **Detect** button may be able to detect the *current* clock speed of the target processor. But in other systems, you may need to input the clock speed manually.

Since the **Detect** button will obtain the current clock speed from 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. Clock set up code may be executed by the startup code run before the breakpoint at the start of main() is hit. For example in NXP projects which use CMSIS – Cortex Microcontroller Software Interface Standard – this will be done by the startup code calling the SystemInit() function. However in some projects clock setup may be done within the main() function itself.

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

Sample rate

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

8.11.2 Start Trace

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

8.11.3 Refresh

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

8.11.4 Settings

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

8.11.5 Reset Trace

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

8.11.6 Save Trace

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

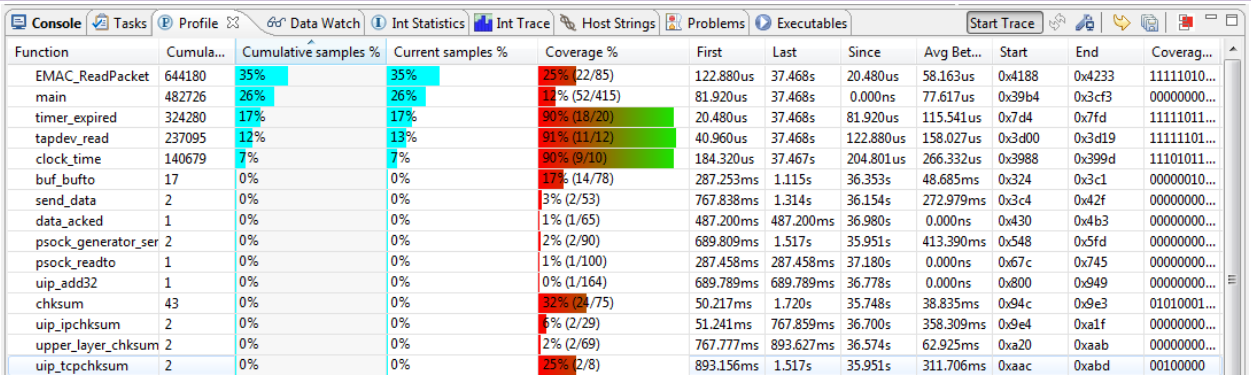
8.12 Red Trace : Profiling

8.12.1 Overview

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

8.12.2 Profile view

The Profile view gives a profile of the code as it is running, providing a breakdown of time spent in different functions. An example screenshot is shown in Figure 8.19.



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

Figure 8.19. Profile View

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

Note:

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

8.13 Red Trace : Interrupt tracing

8.13.1 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.

8.13.2 Interrupt Statistics view

The Interrupt Statistics view displays counts and timing information for interrupt service handlers. An example screenshot is shown in Figure 8.20.

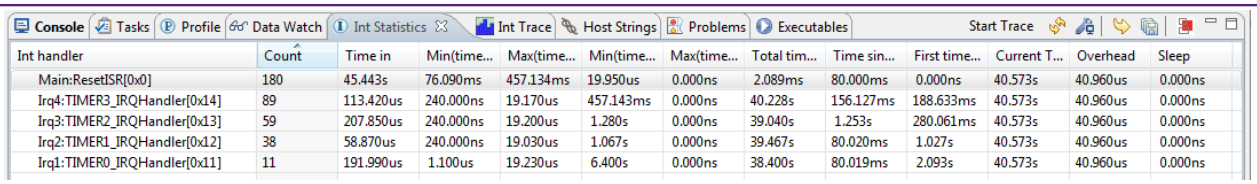


Figure 8.20. Interrupt Statistics View

Information displayed includes:

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

8.13.3 Interrupt Trace view

The Interrupt Trace view provides a time-based graph of interrupts and exceptions and shows their nesting and when the exception is dismissed. This gives a visual representation of time in interrupt service routines and the transitions between them. An example screenshot is shown in Figure 8.21.

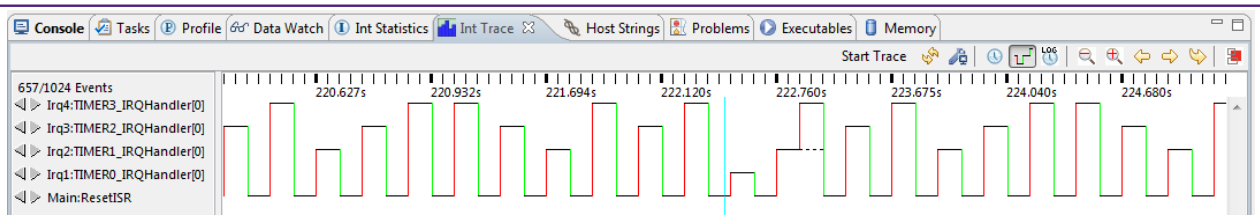


Figure 8.21. Interrupt Trace View

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

The buttons in the top right corner deserve further explanation and are enlarged in Figure 8.22.

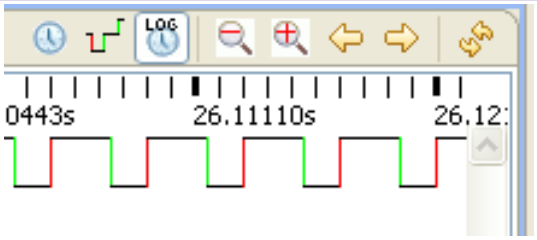


Figure 8.22. Interrupt trace display controls

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

The panel on the left of the Interrupt Trace view lists the interrupt service routines in the application being debugged, as per Figure 8.23.

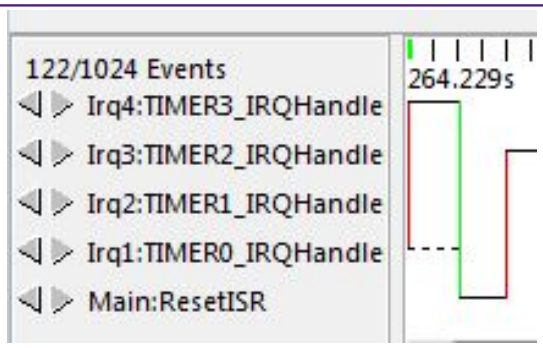


Figure 8.23. Interrupt handlers

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

8.14 Red Trace : Data Watch Trace

8.14.1 Overview

This view provides the ability to monitor (and update) any memory location in real-time, without stopping the processor. In addition up to 4 memory locations can also be traced,

allowing all access to be captured. Information that can be collected includes whether data is read or written, the value that is accessed and the PC of the instruction causing the access.

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

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

Warning:
Data Watch Trace and Instruction Trace [38] cannot be used simultaneously as they both require use of the DWT unit.

8.14.2 Data Watch view

The view is split into 2 sections – with the **item display** on the left and the **trace display** on the right, as per Figure 8.24.

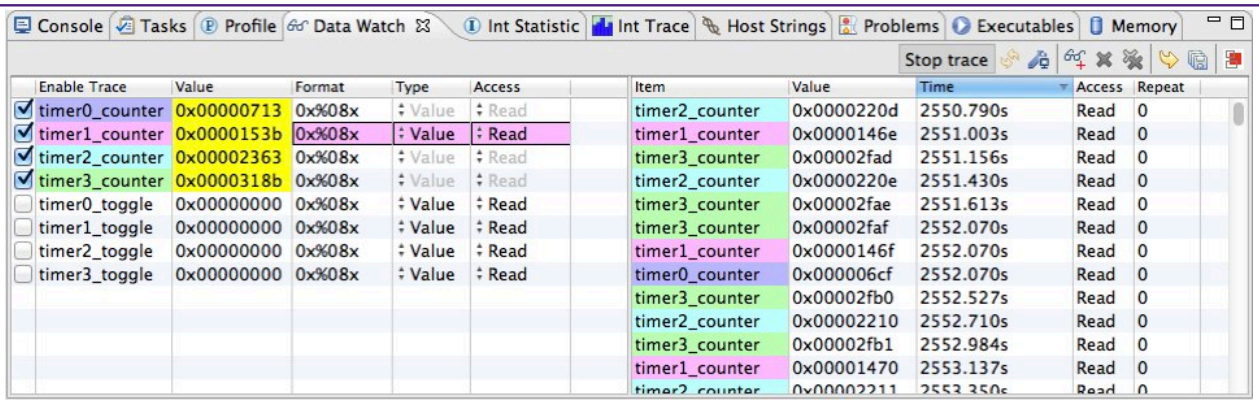


Figure 8.24. Data Watch View

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 8.25.

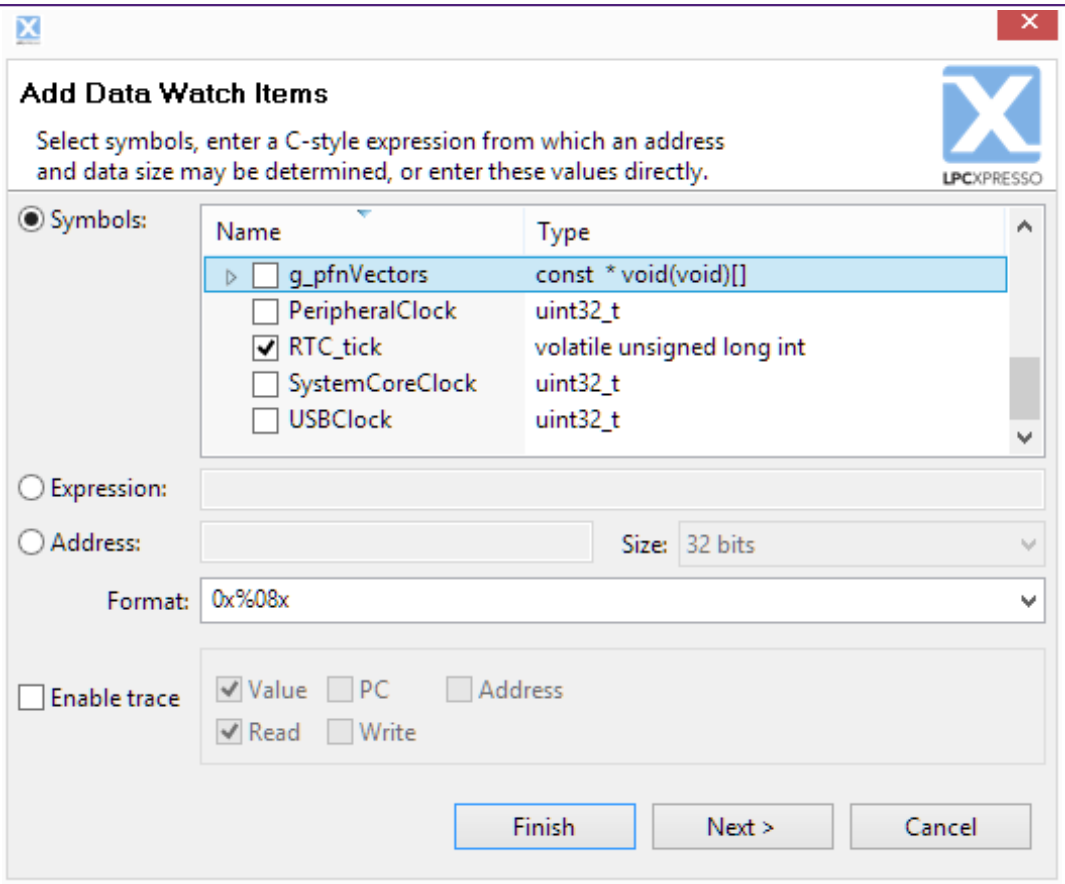


Figure 8.25. Add Data Watch Items

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

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

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

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

Note:

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

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

Item Display

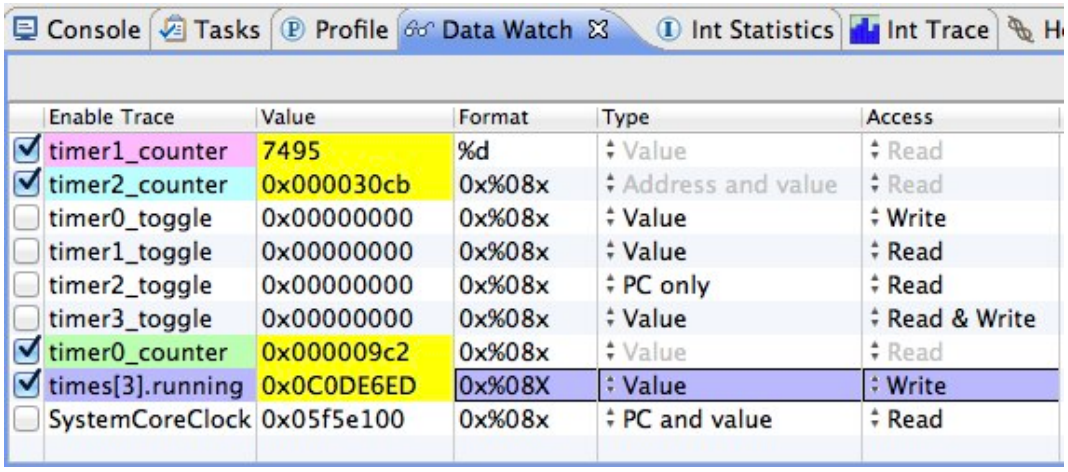


Figure 8.26. Data Watch Item Display

As shown in Figure 8.26, the item display lists the data watch items that have been added. The following information is presented:

- **Enable Trace** - tracing of this item may be enabled or disabled using the checkbox. A maximum of 4 items may be traced at one time. Each traced item is given a color code, so that it may be picked out easily on the trace display (see Figure 8.27).
- **Value** - shows the current value of the item and may be edited to write a new value to the target. If the current value has changed since the last update, then it will be shown highlighted in yellow.
- **Format** - shows the printf-style expression used to format the value and may be edited
- **Type** - shows the trace type, which may be edited while trace is disabled:
  - **Value** - just the trace the value transferred to/from memory
  - **PC only** - just trace the PC of the instruction making the memory access
  - **PC and value** - trace the value and the PC
  - **Address** - trace the address of memory accessed
  - **Address and value** - trace the address and value
- **Access** - shows the access type, which may be edited while trace is disabled:
  - **Write** - just trace writes to the memory location
  - **Read** - just trace reads to the memory location
  - **Read & Write** - trace both reads and writes

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

Trace Display

The trace display shows the traced values of the memory locations. Each location being traced is given a particular color code, to allow all access to it to be easily picked out. See Figure 8.27.

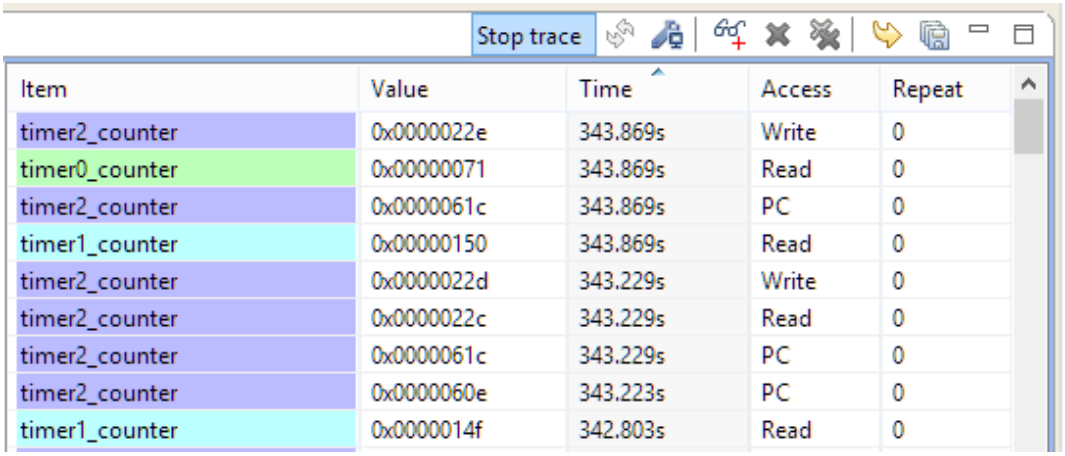


Figure 8.27. Data Watch trace display

8.15 Red Trace : Host Strings (ITM)

8.15.1 Overview

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

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

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

8.15.2 Defining Host Strings

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

Host Strings files are associated with the **Host Strings Editor** and the new file is opened within this editor automatically when the **New File wizard** finishes, as per Figure 8.28.

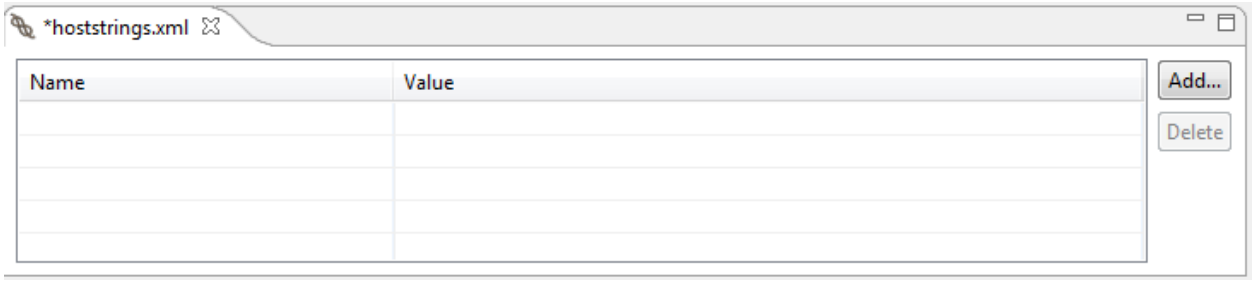


Figure 8.28. Blank Host Strings Editor

Each host string describes a message to be presented within the Host Strings view within the LPCXpresso IDE, plus the formatting information for a single parameter that is sent from the target for presentation within the message. Click on the **Add...** button within the editor to add a new host string, as per Figure 8.29.

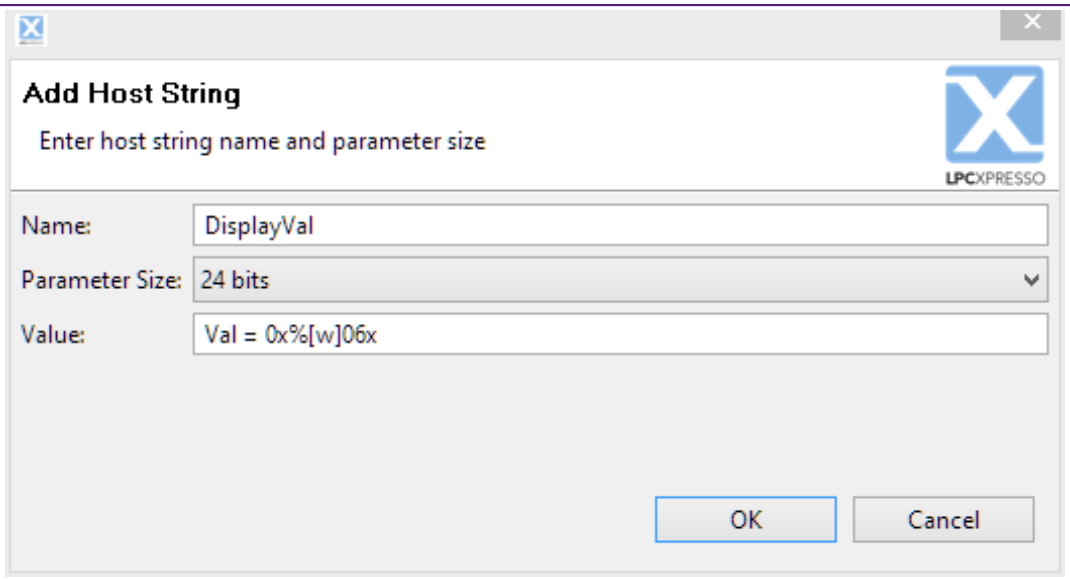


Figure 8.29. Adding a new Host String Name

Name

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

Parameter Size

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

Value

- String that will be displayed in the Host Strings view within the LPCXpresso IDE when the Host Strings Macro is executed on the target.

- By default, when you add a new Host String, the value will be set to the name of the Host String, plus the parameter in hexadecimal format. However this value can be edited as required.
- The format of the parameter is `%[inf][width]ctrl`, where
  - `inf` = `b`, `h` or `w` for byte, half, word (truncated to 24-bits).
  - `ctrl` = `u` or `d` (for decimal), `x` or `X` (for hexadecimal), or `b` (for binary).
  - `width` (optional) a number to specify the minimum number of characters to be printed and whether this should be zero-padded (similar to `printf`).
- Host strings providing incorrect formatting information are marked with a warning triangle within the editor.

An example set of Host Strings within the Host Strings Editor are shown in Figure 8.30.

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

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

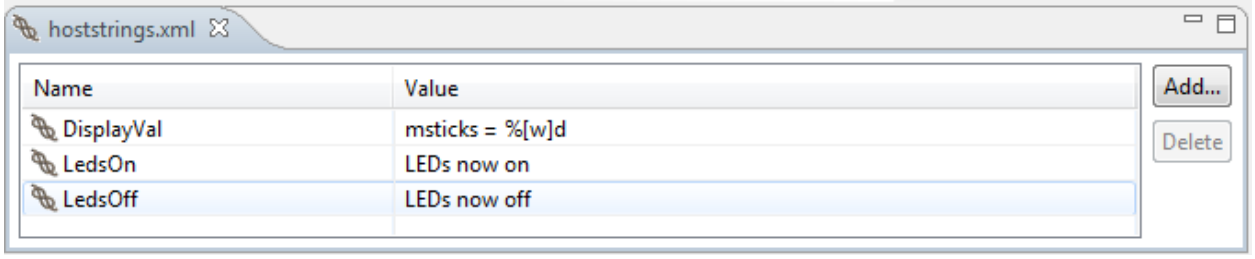


Figure 8.30. Populated Host Strings Editor

8.15.3 Building the Host Strings macros

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

8.15.4 Instrumenting your code

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

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

Note that the above assumes that the source code for your project is located in a subdirectory immediately below the root directory of your project (where the Host String files

are located). If your project is more complex, then it may be better to add `${ProjDirPath}` to your project include paths using the menu:

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

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

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

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

```
HS_DisplayVal(msticks);
```

8.15.5 Host Strings view

The Host Strings view presents the message associated with a host string each time the associated host string macro is called within your code. The message includes the formatted value of the parameter passed into the macro.
Figure 8.31 shows example Host Strings output, as generated by running the `RDB1768cmsis2_HostStringsDemo` project, which can be found in the `RDB1768cmsis2.zip` example bundle provided with the LPCXpresso IDE.

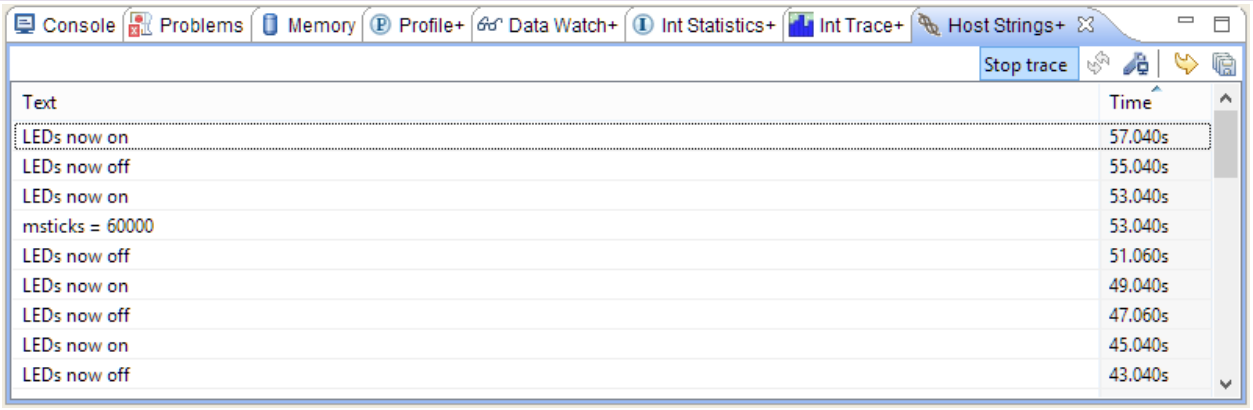


Figure 8.31. Host Strings output

8.16 Instruction Trace

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

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

This documentation is divided into four parts.

- Getting Started [72] - Tutorials to help you learn how to use instruction trace

- Concepts [76] - Technical background of instruction tracing
- Reference [83] - Details of the interface and operation of instruction trace
- Troubleshooting [90] - solutions to common challenges you may face

8.16.1 Getting Started

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

Configuring the Cortex-M0+ for Instruction Trace

Instruction trace on Cortex-M0+ targets requires that some SRAM need to be reserved. The Micro Trace Buffer (MTB) that provides the Instruction Trace capabilities stores the execution trace to SRAM. It is therefore necessary to reserve some SRAM to ensure that user data is not overwritten by trace data. This configuration is **not** required for instruction trace on **Cortex-M3** and **Cortex-M4** MCU parts.

LPCXpresso's new project wizard will automatically includes a file called `mtb.c` for parts which support MTB trace. This file places an array called `__mtb_buffer__` into memory at the required alignment. The MTB will then be configured to use this space as its buffer allowing it to record execution trace without overwriting user code or data. The array `__mtb_buffer__` should not be used within your code since the MTB will overwrite any data entered into it.

The enablement and size of the `__mtb_buffer__` is controlled by three symbols:

- `__MTB_DISABLE` - If this symbol is defined, then the buffer array for the MTB will not be created.
- `__MTB_BUFFER_SIZE` - Symbol specifying the size of the buffer array for the MTB. This must be a power of 2 in size, and fit into the available RAM. The MTB buffer will also be aligned to its 'size' boundary and be placed at the start of a RAM bank (which should ensure minimal or zero padding due to alignment).
- `__MTB_RAM_BANK` - Allows MTB Buffer to be placed into specific RAM bank. When this is not defined, the "default" (first if there are several) RAM bank is used.

To change or add these symbols click on Quickstart Panel->Quick Settings->Defined Symbols.

If you have a project for an MTB supported part which does not have the `mtb.c` file in it already you simply need to copy the `mtb.c` file from existing project or create a new project and copy the file from there.

Warning

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

8.16.2 Trace the most recently executed instructions

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

Note:

Instruction Trace depends on optional components in the target. These components may or may not be available on the LPC device you are working with. See the Instruction Trace Overview [76] for more information.

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

Step 0: Configure memory if using a Cortex-M0+ parts If you are using a Cortex-M0+ part with an MTB you must first configure the memory usage of your code to avoid conflicts. See the Configuring the Cortex-M0+ for instruction trace[72] for instructions on how to do this.

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

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

Step 2: Configure the trace buffer

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

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

If the **Record Trace Continuously** button was grayed out, or you encountered error when trying to set check out the troubleshooting guide [90] .

Step 3: Download the content of the buffer

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

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

Step 4: Review the captured trace

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

1. Toggle the **Link to source** button so that it is selected
2. Toggle the **link to disassembly** so that it is selected too
3. Select a row in the **Instruction Trace** view
4. Use the up and down cursor keys to scroll through the rows in the **Instruction Trace** view.

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

Step 5: Highlight the captured instructions

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

- Toggle the **Profile information** button 

See Toggle profile information [86] for more information.

Stop trace when a variable is set

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

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

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

- Cortex M3 or M4 using ETB [74]

Note – For Cortex-M0+ based systems, this functionality requires additional hardware to be implemented alongside the MTB. LPC8xx does not provide this.

Cortex M3 or M4 using ETB

To trace the instructions that resulted in the write of the unexpected value we are going to have trace continuously enabled with the ETB acting as a circular buffer. Next we will set up a trigger [80] to stop trace being written to the ETB after the value `0xff` gets written to the variable we are interested in. The trigger event requires two DWT comparators. One of the comparators watches for any write to the address of the variable and the other watches for the value `0xff` being written to any address. The position of the trigger in the trace is set to capture a small amount of data after the trigger and then to stop putting data into the ETB.

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

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

Step 2: Find the address of the variable

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

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

Step 3: Enable trace

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

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

Step 4: Enable stalling

To make sure that no packets get lost if the ETM becomes overwhelmed we enable stalling. Setting the **stall level** to 14 bytes mean that the processor will stall when there are only 14 bytes left in the formatting buffer. This setting ensures that we do not miss any data, however, it comes at the cost of pausing the CPU when the ETM cannot keep up with it. See stalling [78] in the Concepts section for more information.

1. Check the **Stall processor** check box
2. Drag the slider to set the **Stall level** to 14 bytes

Step 5: Configure watchpoint comparator

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

Step 6: Configure the value written

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

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

Step 7: Configure the trigger condition

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

Step 8: Prevent trace from being recorded after the trigger

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

The configured view should look like Figure 8.34.

Step 9: Configure and resume the target

Now press the green check button  to apply the configuration to the ETM and ETB. Resume the target after the configuration has been applied. Once the target resumes, the buffer will start filling with instructions. Once the buffer is filled the newest instructions will overwrite the oldest. When the value `0xff` is written to the focal variable, the **trigger counter** will start to count down on every word written to the buffer. Once the **trigger counter** reaches zero, no further trace will be recorded, preserving earlier trace.

Step 10: Pause target and download the buffer

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

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

8.16.3 Concepts

Instruction Trace Overview

MTB Concepts

The CoreSight Micro Trace Buffer for the M0+ (MTB) provides simple execution trace capabilities to the Cortex-M0+ processor. It is an optional component which may or may not be provided in a particular MCU. See *Supported Targets* [90]

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

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

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

- Enabling the MTB [76] - options for controlling the MTB
- MTB memory configuration [77] - how the MTB uses memory
- MTB Watermarking [77] - reacting to the buffer filling up
- MTB Auto-resume [77] - combining multiple trace buffer captures

Warning

The MTB does not have its own dedicated memory. The memory map used by the target must be configured so that some RAM is reserved for the MTB. The MTB must then be configured to use that reserved space as described in the Configuring the Cortex-M0+ for instruction trace[72] in the Getting Started section.

Enabling the MTB

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

The MTB can also be enabled and disabled by two external signals TSTART and TSTOP. On the LPC8xx parts, these are driven by the "External trace buffer command" register:

```
// Disable trace in "External trace buffer command" register
LPC_SYSCON->EXTTRACECMD = 2;
 :
 :
// Renable trace in "External trace buffer command" register
LPC_SYSCON->EXTTRACECMD = 1;
```

Note that if TSTART and TSTOP are asserted at the same time TSTART takes priority.

MTB memory configuration

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

Since the MTB uses the same SRAM used by global variables, the heap and stack, care must be taken to ensure that the target is configured so that the MTB's memory and the memory used by your code do not overlap. For more information see Configuring the Cortex-M0+ for instruction trace [72] in the Getting Started Guide.

MTB Watermarking

The MTB watermarking functionality allows the MTB to respond to the buffer filling to a given level by stopping further trace generation or halting the execution of the target. The watermark level and the actions to perform can be set in the **Instruction Trace Config** view [87]. The defined action is performed when the Watermark level matches the MTB's write pointer value. The halt action can be augmented with the **auto-resume** behavior.

MTB Auto-resume

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

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

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

Tip:

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

MTB Downloading Trace

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

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

Embedded Trace Macrocell

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

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

Stalling

The Embedded Trace Macrocell [78] (ETM) uses as relatively small FIFO buffer to store formatted packets. These packets are then copied to the embedded trace buffer [79] (ETB). This FIFO buffer can overflow when the rate of packets being written into it exceed the rate at which they can be copied out to the ETB.

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

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

Stalling is enabled from the **Instruction Trace Config** view [88]. Check the **Stall processor** box and select a stall level with the slider.

ETM events

Events are boolean combinations of two event resources [78]. These events can be used to control when tracing is enabled or to decide when the trigger will occur for example. The available event resources are dependent on the chip vendor's implementation.

Events can be specified in the **Instruction Trace Config** view [88]. For details on how to build these events see ETM event configuration [89] .

Event Resources

Event resources are used by the ETM [78] to control the ETM's operation. They can be combined to form ETM events [78].

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

- Watchpoint comparators [79] — match on addresses and data values
- Counters [79] — match when they reach zero
- Trace start/stop unit [79] — combine multiple input
- External [79] — match on external signals

- Hard wired [79] — always match

Watchpoint comparator event resource

Watchpoint comparators are event resources that facilitate the matching of addresses for the PC and the data access as well as matching data values. They are implemented by the Data Watchpoint and Trace [82] (DWT) component.

Counter event resource

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

Trace start/stop unit event resource

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

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

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

To use this method

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

External event resource

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

Hard wired event resource

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

Embedded Trace Buffer

The Embedded Trace Buffer (ETB) makes it possible to capture the data that is being generated at high speed in real-time and to download that data at lower speeds without the need for an expensive debug probe. Red Trace Instruction trace on Cortex-M3 and Cortex-M4 targets is facilitated by the ETB in conjunction with the Embedded Trace Macrocell [78] (ETM).

The ETB and ETM are optional components in the Cortex-M3 and Cortex-M4 targets. Their implementation is vendor specific. When they are implemented the vendors may implement different subsets of the ETM's and ETB's features. Instruction trace will automatically detect which features are implemented for your target; however, note that not all of the features listed in this guide may be available on your target.

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

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

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

Triggers

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

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

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

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

Trace after

When the slider is at the far left, the **words before trigger** will be zero and the **words after trigger** will be equal to the number of words which can be stored in the ETB. This corresponds to the situation where the region of interest occurs after the trigger condition. Once the ETB receives the trigger packet the trigger counter, which was equal to **words after trigger** will count down with every subsequent word written to the ETB, until it reaches zero. The trigger packet will be early in the trace and the instruction trace will include all instructions from when the trigger event occurred, until the ETB buffer filled up.

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

Trace around

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

and some from after the trigger. Use the slider to balance the amount of trace before and after the trigger.

Trace before

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

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

Timestamps

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

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

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

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

Debug Request

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

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

Output all branches

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

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

This option is not usually required as the Instruction trace tool is able to reconstruct the program flow using just the indirect branches and the memory image of the executed code. This option dramatically increases the number of packets that are output and can result in FIFO overflows, resulting in data loss or reduced performance if stalling [78] is enabled. It can also make synchronization harder (e.g. in triggered traces) as you can end up with fewer I-Sync packets in the ETB.

Data Watchpoint and Trace

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

Warning:
Instruction Trace and Data Watch Trace [64] cannot be used simultaneously as they both require use of the DWT unit.

Instruction address Comparator

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

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

Note:
Instruction addresses must be half-word aligned

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

Data access address Comparators

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

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

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

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

Data Value comparator

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

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

- Data Value Write — only true if a value is written that is equal to the Match Value

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

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

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

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

Cycle Count

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

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

8.16.4 Reference

Instruction trace view

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

It should look like Figure 8.32.

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

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

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

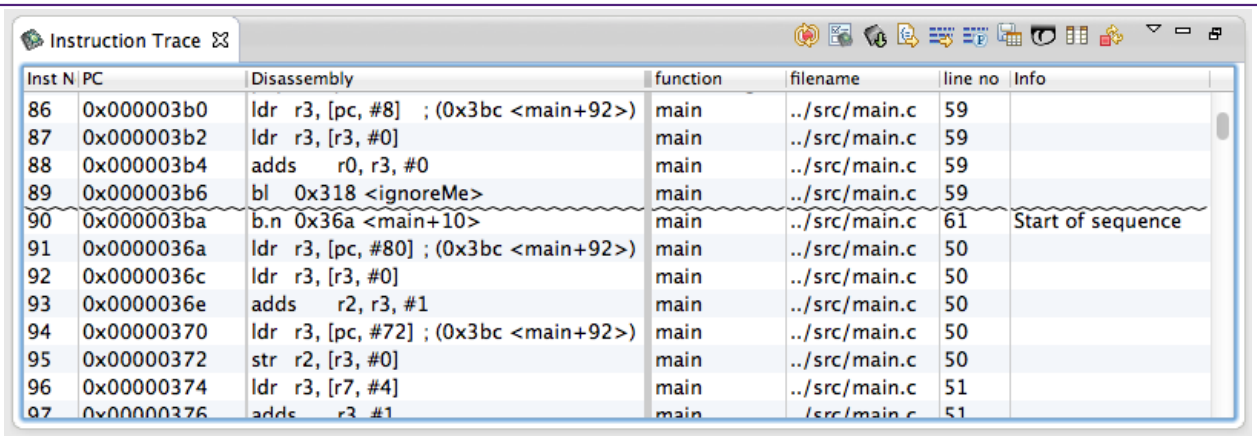


Figure 8.32. The instruction trace view

Instruction Trace view Toolbar buttons

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

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

Record trace continuously

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

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

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

Show Instruction Trace config view 

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

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

- Cortex M0+ MTB [87]
- Cortex M3 ETB [88]
- Cortex M4 ETB [88]

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

Download trace buffer 

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

Notes

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

Link to source 

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

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

Link to disassembly 

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

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

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

Toggle profile information

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

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

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

Tip:

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

Save trace

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

Jump to trigger

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

Stop auto-resume

When using the MTB auto-resume feature [77], the user may not have time to press the suspend button or the **Cancel** button in the download trace progress dialog box as the target is being rapidly suspended and restarted. Pressing the **Stop auto-resume** button will turn off the auto-resume feature, so that the target will suspend the next time the MTB reaches its watermark level without resuming.

Select columns

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

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

Table 8.2. Instruction view column descriptions

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

Instruction Trace Config view for the MTB

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

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

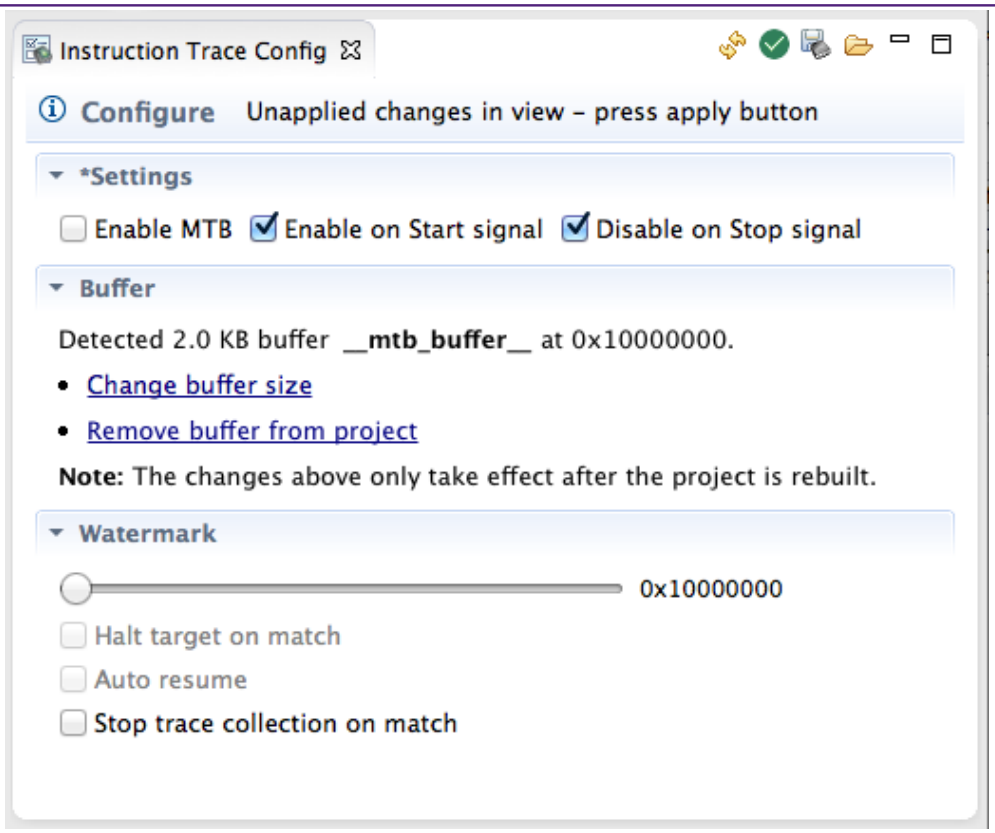


Figure 8.33. Instruction Trace config view for MTB instruction trace

Configuring the buffer

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

Enabling

The three check-boxes in the top section of the view control whether the MTB is enabled or not. The first check-box **Enable MTB** can be used to directly enable or disable the MTB. The second two check-boxes control whether or not the MTB is affected by start and stop signals `TSTART` and `TSTOP` which can be triggered by software using the target's external trace buffer command register `EXTTRACECMD`.

Buffer

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

See *MTB memory configuration* [77] for more information and *Configuring the Cortex-M0+ for instruction trace* [72] for an example.

Watermark

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

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

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

Note

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

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

Viewing the state of the MTB

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

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

Instruction Trace Config view for the ETB

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

Changes made to the MTB configuration are only applied when the **Apply** button is pressed. A star appended to a section title indicates that it contains unapplied changes.

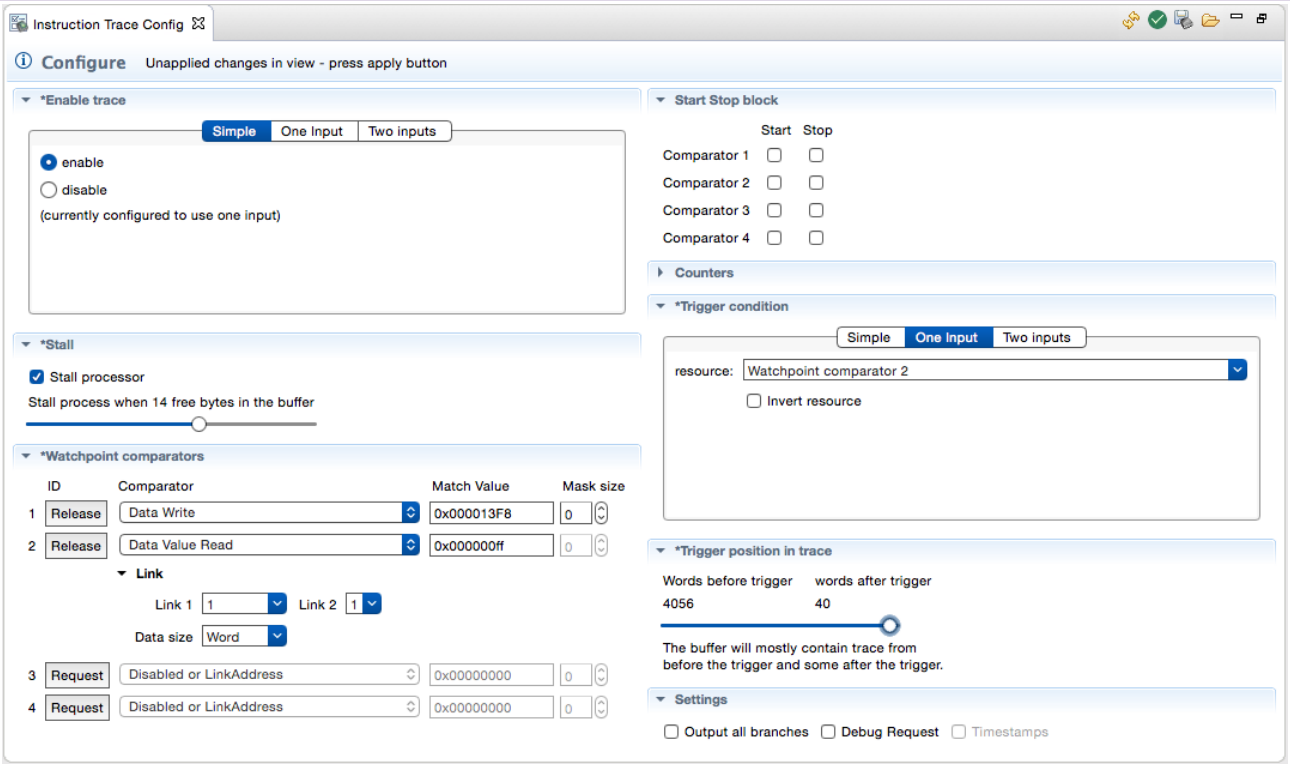


Figure 8.34. The advanced configuration dialog for the ETB

ETM Event configuration

An ETM event [78] is a boolean combination of up to two event resource inputs[78]. The **Instruction Trace Config** view provides an easy way to build these events by allowing the user to choose the complexity of the event. These are used for trace enablement and the trigger condition for example.

- The simplest option is a binary enabled/disabled choice. This is accessed by selecting the **Simple** tab. Select **enable** for the Event to always be true or **disable** for the event to always be false. See Figure 8.35.

- To use a single event resource select the **One Input** tab. An event resource can be chosen from the drop-down and the event will be true when the event resource is true. Checking the **Invert resouce** box will cause the Event to be true when the event resource is false, and visa-versa. See Figure 8.36.

- To combine two event resources select the **Two Inputs** tab. The events can be chosen from the drop-downs and the logical combination operation selected. As with the **One Input** tab the resources can be inverted. See Figure 8.37.

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

Figure 8.35. Simple ETM event configuration

Figure 8.36. One Input ETM event configuration

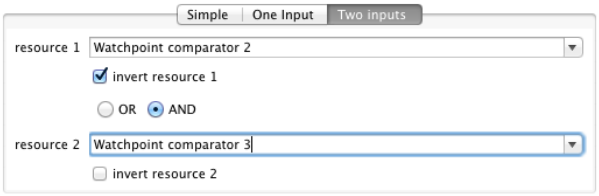


Figure 8.37. Two Inputs ETM event configuration

Supported targets

Instruction Trace is not supported on all targets. It only works on targets which have the necessary hardware. Instruction trace is currently supported on the following targets from NXP:

- LPC8xx (MTB)
- LPC11U6x (MTB)
- LPC18xx (ETM / ETB)
- LPC43xx (ETM / ETB)

8.16.5 Troubleshooting

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

General

Instruction Trace claims target not supported when it should be

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

You can also double check that your target is in *Supported Targets* [90]

MTB

Target crashes when MTB is enabled

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

Target keeps resuming itself and I cannot stop it

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

See *Auto-resume* [77] in the Concepts section for more information.

ETB

Trigger Packet missing from trace even though trigger occurred

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

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

8.17 Comparator sharing

Watchpoints, Data Watch trace and Instruction trace all make use of hardware debug comparators. These comparators can match PC values, data values and address accesses which can be used to pause the execution of a target; output a message over the SWO; or to start an instruction trace buffer recording for example.

There are a limited number of these hardware debug comparators implemented on a part, for example the DWT unit of a Corex-M3 would typically have four. A specific comparator can only be used by one component at a time. To use a comparator for a particular component, either watchpoint, data watch or instruction trace, it must be requested from the IDE. If all of the comparators are currently in use the request will be denied. Comparators need to be explicitly released by the component that has successfully requested it to be availble to another component. Different comparators may be used by different components at the same time.

8.17.1 Watchpoints – requesting and releasing comparators

Watchpoints are similar to breakpoints, but instead of halting execution on a specific line number, they halt execution when a specific variable is accessed.

Comparators for watchpoints are requested automatically when the debugger resumes execution of the target. If there are insufficient comparators available the error message in the figure Figure 8.38 below will be shown.

The comparators for watchpoints are released when the target suspend excution

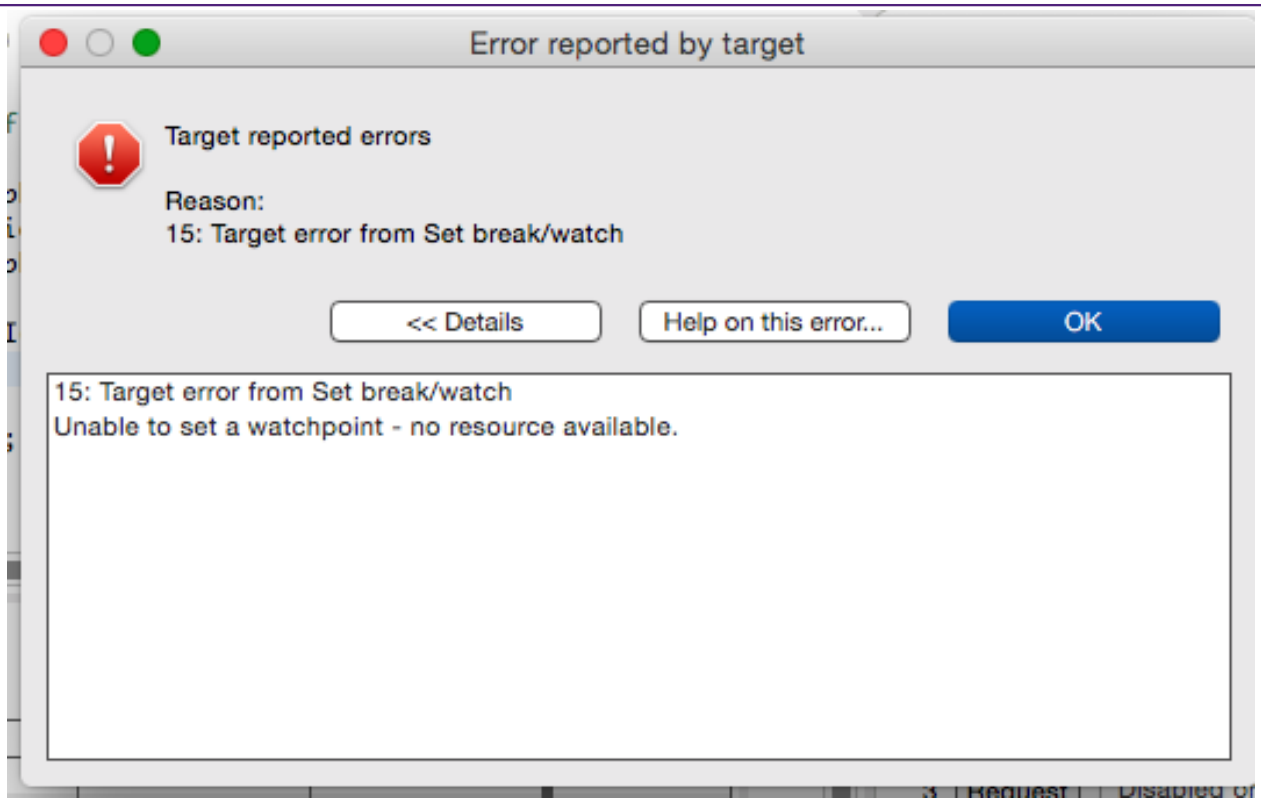


Figure 8.38. Unable to set watchpoints

8.17.2 SWO Data Watch comparators

The SWO data watch comparators emit information about a data access over the SWO channel.

A comparator is requested when a check is marked next to an item in the Data Watch view [51]. The tool selects the first free comparator to use. If no comparators are available you will be alerted with the error message in figure Figure 8.39.

When an item is unchecked in the Data Watch view [51] its compartor is released and will become available to other components.

See Data Watch Trace [51] for more information about using SWO Data Watch Trace.

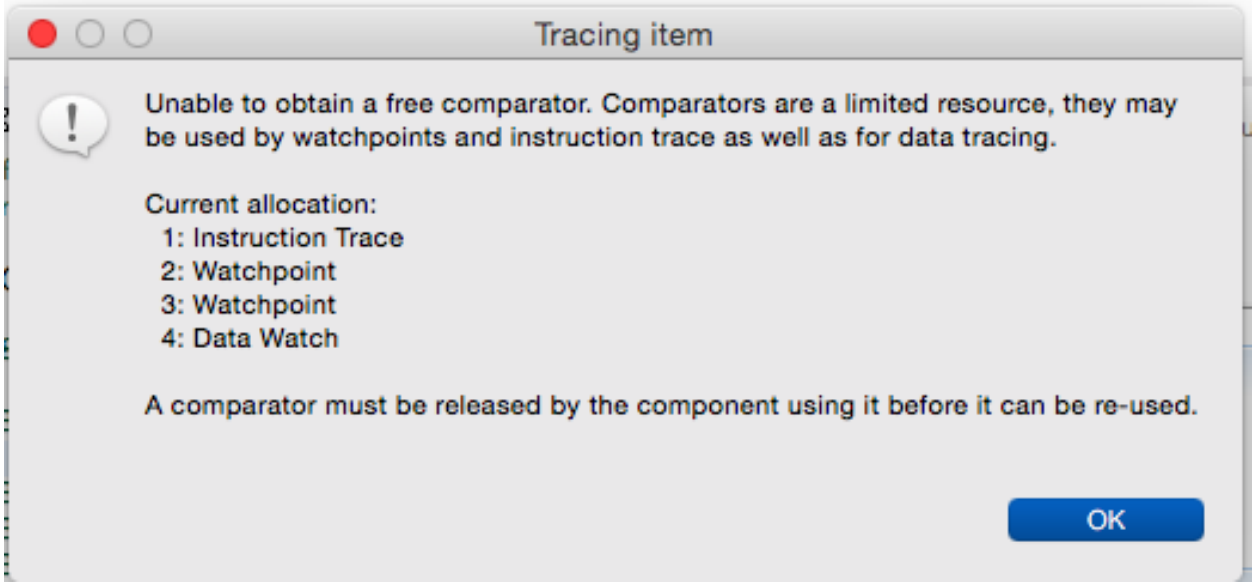


Figure 8.39. Unable to set watchpoints

8.17.3 Instruction Trace

The comparators can be used to control the starting and stopping of instruction trace buffers.

To use a comparator select the **Request** button corresponding to the comparator to use in the **Watchpoint comparators** section (see figure Figure 8.40). If that specific comparator is already in use you will see an error message (see figure Figure 8.41). Once it has been sucessfuly requested it can be configured and the **Request** buttin is replaced with a **Release** button.

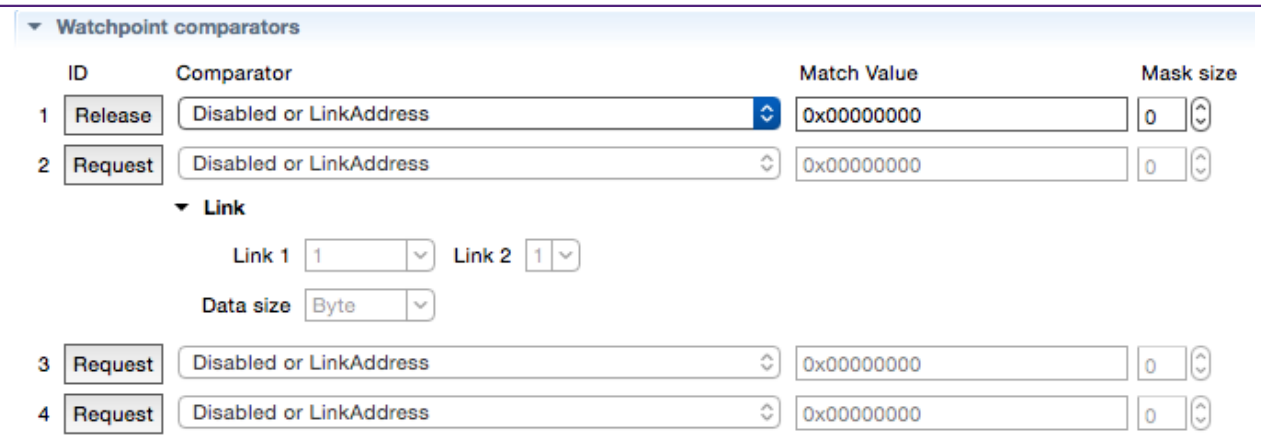
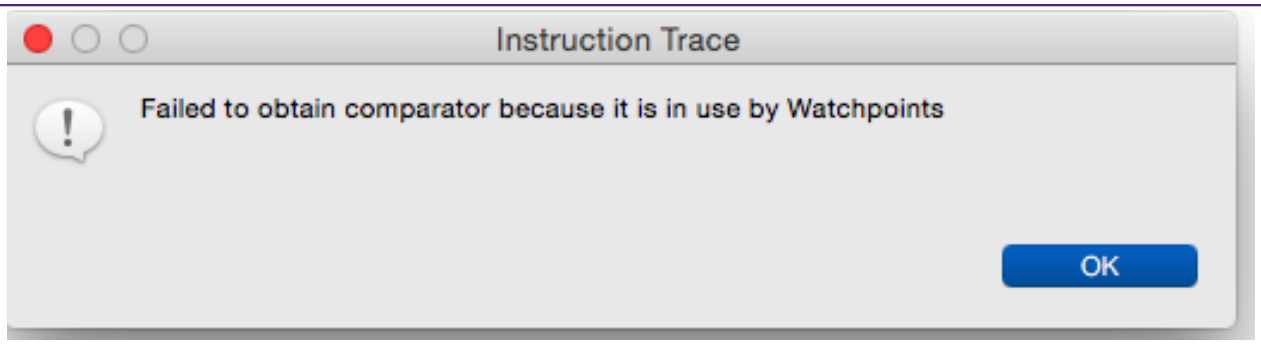


Figure 8.40. Instruction Trace configuration

Press the **Release** button to stop using the comparator for Instruction trace and aloow other components to use it. To find out more about Instruction Trace please see the Instruction Trace Guide [71]



Error message shown when the user tries to request a comparators that is already in use.

Figure 8.41. Unable to set watchpoints

9. Red State SCT tool

9.1 Red State Overview

Red State is a graphical tool incorporated into the LPCXpresso IDE for designing state machines and automatically generating the code required to implement the state machine. The state diagram is built in the Red State editor by drawing states and the transitions between them. You can easily set conditions under which the transitions will occur and associate events with the transitions, such as setting of outputs.

Red State can be used to create and generate code for state machines for NXP's **State Configurable Timer** (**SCT**) peripheral (which is available on certain MCUs) as well as for software-only state machines (which can be run on any MCU).

The Red State related material in this User Guide has three main parts: two tutorials and a reference section. SCT state machine tutorial [95] provides a walk-through for using Red State to implement a state machine on NXP's SCT. Software state machine tutorial [106] provides a walk-through for creating a software state machine. Subsequent sections provide a reference for Red State.

9.1.1 The NXP State Configurable Timer

Red State supports NXP's State Configurable Timer Peripheral (SCT) that combines a 32-bit Timer/Counter with a configurable state-machine. The SCT integrates its internal state with timer values, inputs and outputs to trigger events. These events can change the internal state of the SCT or the state of output pins. They can affect the timer — starting or stopping it and capturing its value. It can also generate interrupt requests to the processor or perform a DMA request.

Red State provides an intuitive interface for building SCT state machines entirely within the LPCXpresso IDE. It provides easy access to the powerful features of the SCT through a simple graphical interface. All the code that is necessary to configure your design on the SCT can be generated with a single click.

9.1.2 Software State Machine

Red State can generate a software state machine in C from your state machine diagram. You can define input and output variables and functions to call. Making changes to the state machine is simple and you can regenerate all the required code with a single click, avoiding the risk of introducing errors into your business logic.

9.1.3 Integrating a state machine with your project

Once the state machine design is completed, code can be generated from it to be included in the build. Code is only generated when the process is initiated using the **Generate Code** command. For example, if code had been generated for a state machine and the state machine was subsequently altered, the user would need to click on **Generate Code** again.

This document includes step-by-step instructions for creating and using state machines with the SCT and for running in software.

9.2 Red State : SCT state machine tutorial

This section will take you through the steps required to generate a state machine in Red State for the SCT target.

A state machine can only be added to an existing project in the LPCXpresso IDE. The first step is to import the base project to extend in this tutorial. The second step is to create a new state machine and add it to the project using the **Red State Wizard**. The final step is to use the integrated Red State graphical editor to design the state machine and generate the SCT configuration to run on the target.

9.2.1 Prerequisites

This tutorial is based on the `blinky` example provided by NXP. It is intended to be run on the Hitex 1850 board using Rev A silicon. This project will sequentially toggle outputs, which we name `LED1` to `LED4`. Note that the outputs on the Hitex board do not come connected to LEDs. Connect LEDs or a logic analyzer to the outputs to observe the state machine in action.

9.2.2 Creating a new project for the SCT

To get started, import the SCT base project from the example projects. First, select Import projects from the **Quickstart Panel** and press the **Browse** button corresponding to the **Project Archive** edit box. Next, navigate to the `/NXP/LPC1000/LPC18xx` folder and choose the file `LPC1850A_Hitex.zip`. Now press the **Next** button.

The archive contains several projects, but we only require the following three. The `CMSISv2pxx_LPC18xx_DriverLib` is required to access the LPC1850. The `LPC1850A_HitexA4_SCT_Base` project contains the code required to set up the system control unit. To use this project you must add your own state machine. The `LPC1850A_HitexA4_SCT_Blinky` project contains the completed SCT state machine that is the result of following this tutorial.

To follow along with this walk-through, select the `CMSISv2pxx_LPC18xx_DriverLib` and `LPC1850A_HitexA4_SCT_Base` projects. To work from the completed state machine, select `CMSISv2pxx_LPC18xx_DriverLib` and `LPC1850A_HitexA4_SCT_Blinky`. The rest of this tutorial will assume that the former configuration is being used.

9.2.3 Adding a new SCT state machine to the project

Now that the project is set up, the next step is to add the state machine to it. The state machine is described in an `.rsm` file in the project. Create the file with the Red State Machine file generator. Make sure that the `src` folder containing the C source files in the `LPC1850A_HitexA4_SCT_Base` project is selected and then select **New -> Other** from the file menu or press the **New** button in the toolbar to bring up the Wizard selection dialog.

Type `state` in the filter box to find the **Red State Machine file generator** or just expand the **Red State** folder (see Figure 9.1).

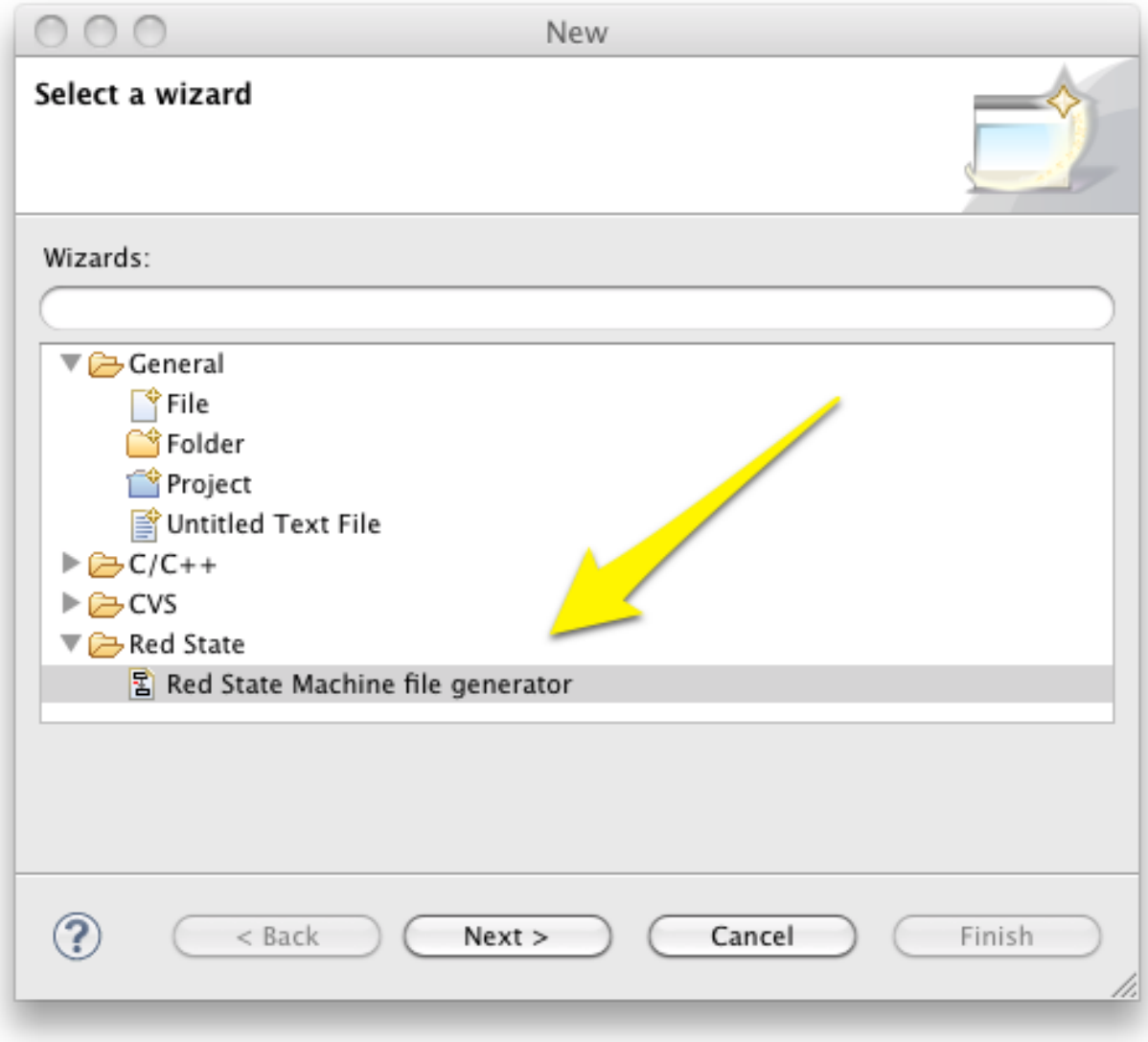


Figure 9.1. Creating a new state machine

Select the **Red State Machine file generator** and press the **Next** button. In the **New State Machine** dialog, make sure that the `src` folder is selected. Enter the file name for the state machine file (e.g. `blinky`) and press **Next**. Note that this name will have `.rsm` appended to it and will be added to the project. If an `rsm` file with that name already exists in the workspace, an error message will appear, and you will need to choose a different name for the file.

Pressing the **Finish** button would have resulted in the use of the default settings — creating an SCT state machine with a **unified timer** and `initial` and `always` states pre-populated.

For this example, leave the options in their default conditions except for the **unified timer** option. Only the low timer will be used, so deselect the **unified timer** option. You also need to choose the target MCU, for example **LPC18xx**.

Hitting **Finish** creates a new state machine and opens it in the **Red State editor**. The perspective is automatically switched to the Red State perspective. This perspective shows the different views associated with editing the state machine. It is opened automatically when an `rsm` file is loaded (or when a new `rsm` file is created from the Wizard). The perspective can be switched using the menus by going to **Window -> Open Perspective**, or by using the perspective toolbar in the top right of the LPCXpresso IDE.

9.2.4 The blinky state machine overview

The blinky state machine example is based on NXP's blinky example in their *FSM Designer for the State Configurable Timer* documentation. It toggles four outputs sequentially, with the direction determined by an input signal. The target should be set up with four LEDs on outputs (named LED1, LED2, LED3 and LED4) and two inputs (named DOWN and RESET).

This example uses only the low (L) counter of the SCT, activating one of four outputs at a time. Each time there is a counter match, the output advances. With four LEDs on the outputs, we should observe the following sequence:

LED1 -> LED2 -> LED3 -> LED4 -> OFF -> LED1 -> LED2 -> ...

If the DOWN input is high, the direction of the sequence is reversed:

LED1 -> OFF -> LED4 -> LED3 -> LED2 -> LED1 -> OFF -> LED4 -> ...

If the RESET input is high, all outputs are set to off. When RESET goes low, the sequence starts again — either from LED1 if DOWN is low or from LED4 if down is high.

9.2.5 Naming outputs and inputs

The **Outputs for State Machine** and **Inputs for State Machine** views contain the default inputs and outputs for the SCT. The blue text indicates that it is a fixed property. The black text is editable. Start by giving descriptive names to the output pins 0 — 3 that will represent the LEDs.

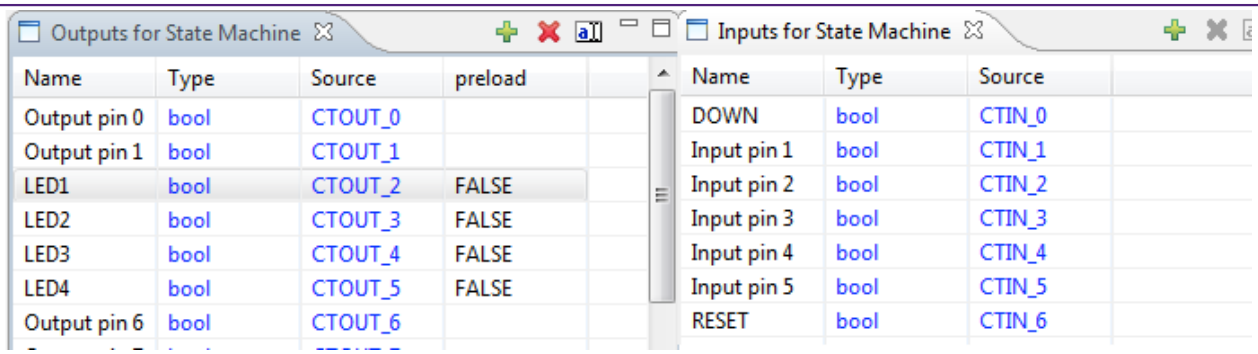


Figure 9.2. The output and input views for the *blinky* state machine

Click on **Output pin 2** in the Name column and type LED1 to rename it. Next click in the preload column and select **FALSE** from the drop-down list. The output is initialized to this value on a restart. Name the next three output pins to correspond to LED2, LED3, and LED4 and set their preload values to **FALSE** (see left panel of Figure 9.2).

Note

Two outputs cannot have the same name. Attempting to change a name to one that is already being used by another output will be ignored and the output will keep its original name.

Naming the inputs is very similar to naming the outputs. Assign the name DOWN to input pin 0 (**CTIN_0**) and RESET to input pin 6 (**CTIN_6**). Inputs have no preset value (see right panel of Figure 9.2).

9.2.6 Matching the timer

A key feature of the SCT is the triggering of events based on the value of the timer. To access this feature add a **match** input that will be high when the timer is equal to some value:

1. Add a Low match by pressing the  button in the input toolbar. This will add a new input. Change its name to `maxcount` and its type to **Match Low**.

2. Add another input to store a value to compare with the low counter. Name it `speed` and leave its type as `const int`. Enter the value `40000` in the source column.

3. In the row for `maxcount`, choose `speed` from the drop-down in the source column.

The input `maxcount` will be high when the `L` counter is equal to the value of `speed`.

9.2.7 The states

The next step is to add the states that the SCT will be using. The blinky state machine has four distinct states corresponding to each LED exclusively being on, and one state corresponding to no LEDs being on. It also has an extra *virtual* state described below. The graphical editor is used to configure the required states and set their names.

Special states

The SCT has several reserved names for states corresponding to special states. Choosing **include initial state** and/or **include ALWAYS state** in the **State Machine Wizard** automatically adds these special states to the state machine when it is created. You can also include them manually by adding a state and giving it the correct reserved name. NXP described these states as follows:

`H_ALWAYS`, `L_ALWAYS` (split counter mode), `U_ALWAYS` (unified counter mode):

> These are pseudo (or “virtual”) states that do not get mapped into a state register value for the SCT state machine. It is just a graphical convenience to represent events that are state independent, or in other words, are considered to be valid in all defined states

`H_ENTRY`, `L_ENTRY` (counter split mode), `U_ENTRY` (unified counter mode)

> These represent the initial value of the state register the SCT will have after configuration. It is a useful feature as you might want the state machine to start from a user defined condition. If not specified, the SCT will be left in the default configuration after reset, that is, start from state zero. Note that the tool will map the state numbering at its convenience, so use the ENTRY feature if the starting state is of relevance for your application

Deleting a state

Red State editor now contains four boxes, representing states, with the labels `H_ENTRY`, `L_ENTRY`, `H_ALWAYS` and `L_ALWAYS`. Since this example only uses the `L` counter, delete the `H_ENTRY` and `H_ALWAYS` states by right-clicking on them in the graphical editor and choosing **delete** from the context menu.

Adding states

The L_ENTRY state corresponds to the state with all the LEDs off. To add the four states, corresponding to each of the LEDs being exclusively on, do the following:

1. Click on the **State** button in the **Red State editor**

2. Click where you would like to place it in the editor (see Figure 9.3).

3. Rename it by clicking on it once to select it, and then clicking on it a second time to edit the name.

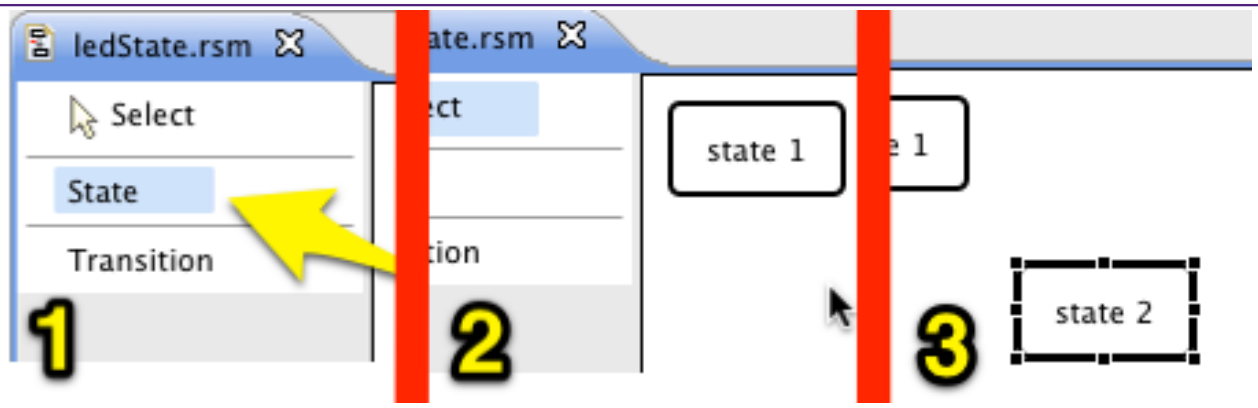


Figure 9.3. Adding a new state

Add the four states and name them `LED1on`, `LED2on`, `LED3on` and `LED4on`. As with the naming of inputs and outputs, state names have to be distinct. Red State ignores any attempt to change the name of a state to that of an existing state.

Drag the states to arrange them on the canvas as shown in Figure 9.4.

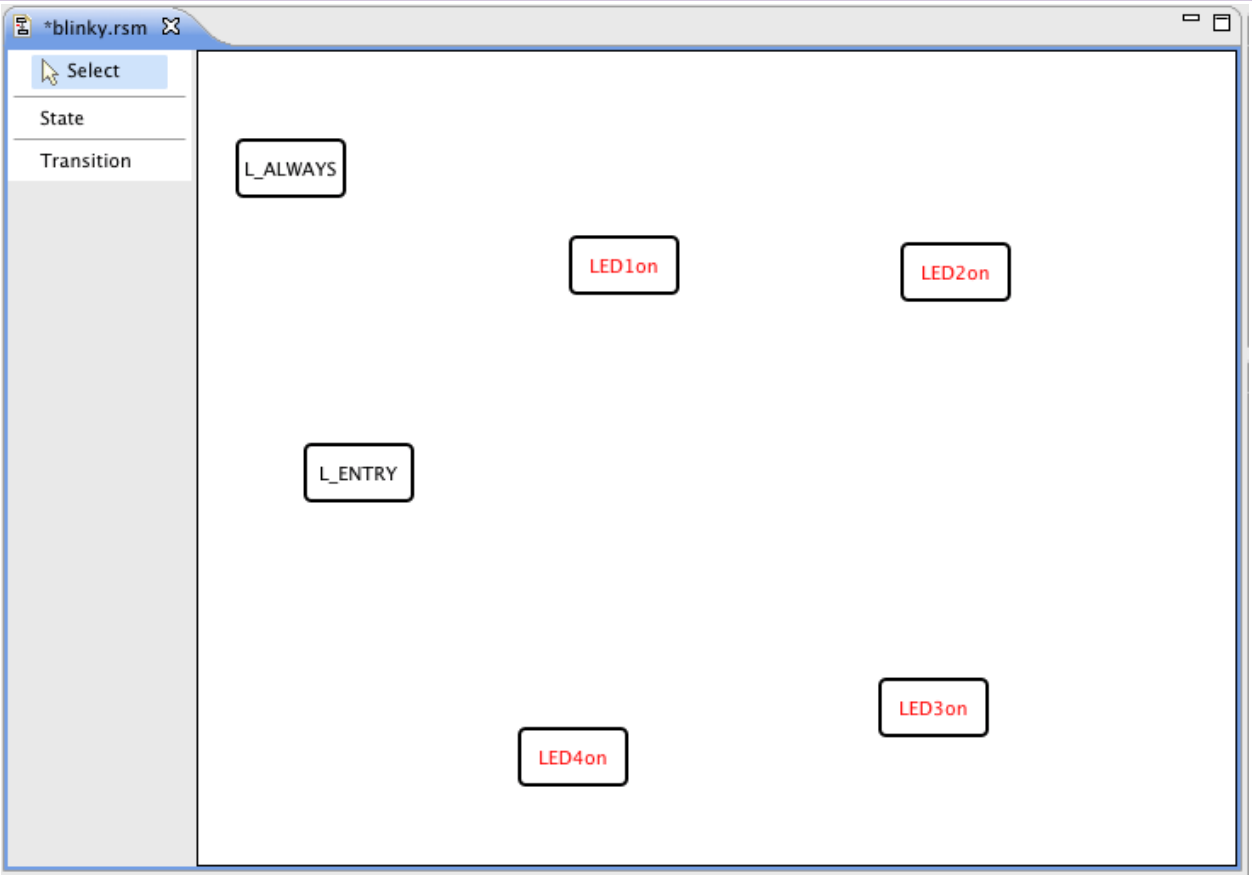


Figure 9.4. The states for the *blinky* example

The red text of the state labels indicates that there is a problem with that state. Hovering the cursor above a state will display any errors associated with that state. For example see Figure 9.5 where Red State is warning that the state machine can never enter this state. This problem will be addressed in the following sections.

Figure 9.5. Error indication

9.2.8 Adding transitions

Creating a new transition

Whenever the RESET input is high, the state machine should enter the L_ENTRY state, regardless of the state machine's current state. Adding a transition from the L_ENTRY state is a convenience that avoids the necessity of drawing transitions from every other state when such a global transition is required.

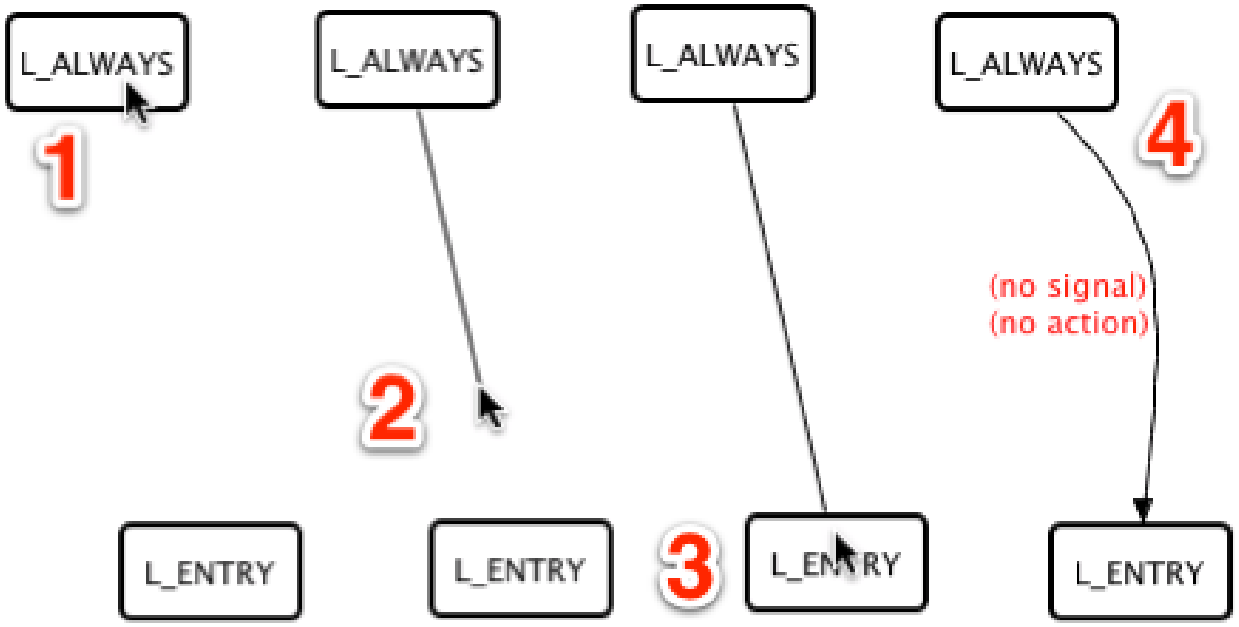


Figure 9.6. Adding a transition between states

To add a transition from L_ALWAYS to L_ENTRY select the Transition tool then:

1. Click on the starting state for the transitions (L_ALWAYS)
2. Move the pointer to the end state for the transition (L_ENTRY)
3. Click on the end state for the transition
4. Red State creates the new transition.

The label of the new transition is highlighted in red because it does not have a signal associated with it. See Figure 9.6.

Adding a signal to a transition

Create a signal to link the RESET input with the new transition by pressing the add button in the **Signals** panel. Next, double-click on the new signal and rename it Reset. Finally,

right-click on `Reset` and then select Input and then **RESET** from the pop-up menu – see Figure 9.7.

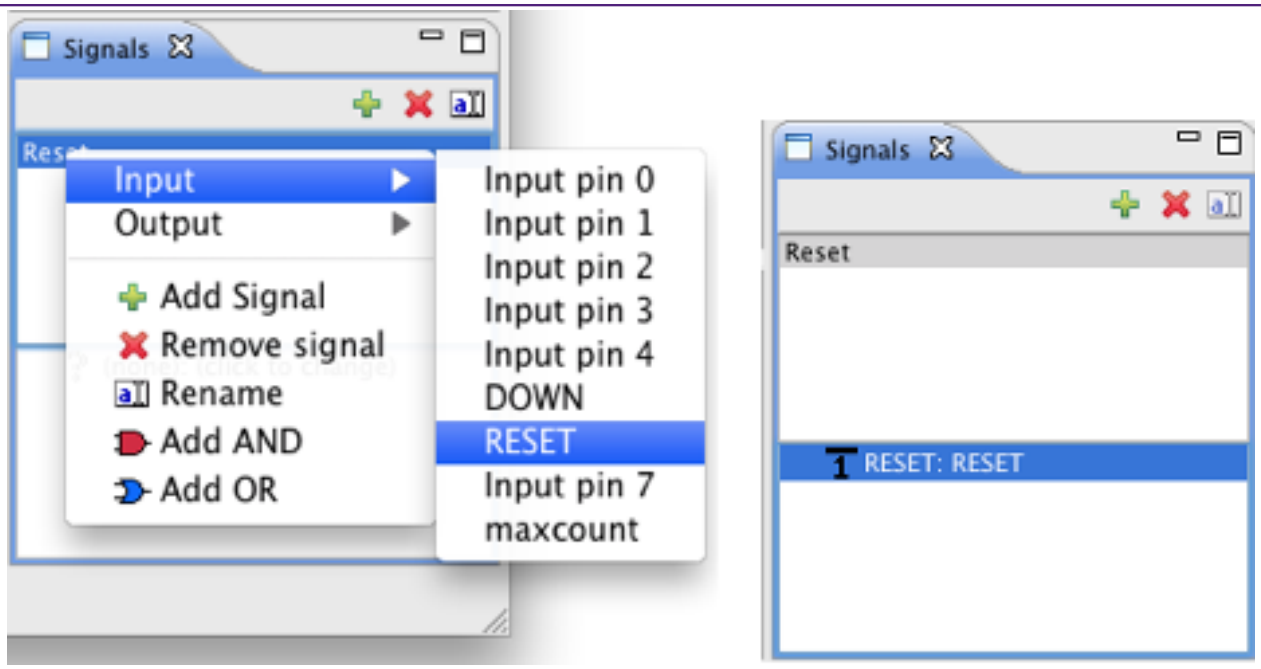


Figure 9.7. Adding a reset signal

The bottom half of the **Signals** panel now shows the composition of the signal selected in the top half. Notice the **1** icon next to the input name in the bottom half. This icon indicates that the signal will fire when the `RESET` input is high. Right-clicking on it and choosing an option from the **Set I/O condition** submenu can change this behavior.

To add the signal to the transition, right-click on the transition's label and choose **Reset** from the **Set Signal** context menu.

Adding action elements to transitions

The reset transition should turn off all the LEDs. A transition can have at most one **action** associated with it. This **action**, however, can have multiple **actions elements** associated with it. For example, an **action element** might set or clear an output pin.

Right-click on the **RESET** transition label and choose **Add Action -> CLEAR -> LED1**. Repeat for the other three LEDs (see Figure 9.8).

Adding action elements from the context menu automatically creates an action to contain the chosen action element if the transition has no action associated with it. Otherwise, it appends the chosen action element to the existing action.

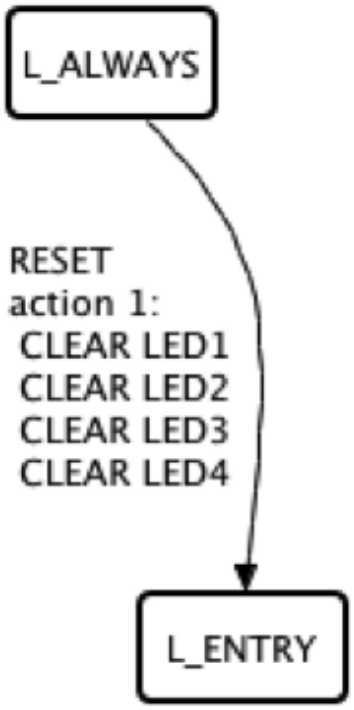


Figure 9.8. The RESET transition

To keep the state machine in the `L_ENTRY` state while the RESET signal is high, even if other transitions are fired, set the transition **priority** property. Enter `100` in the priority column for this transition in the **State Table** view (see Figure 9.9). The state machine transitions to the state defined by the transition with the highest priority when the signals for multiple transitions from the current state are satisfied simultaneously. Note that the SCT performs all actions associated with each satisfied transition.

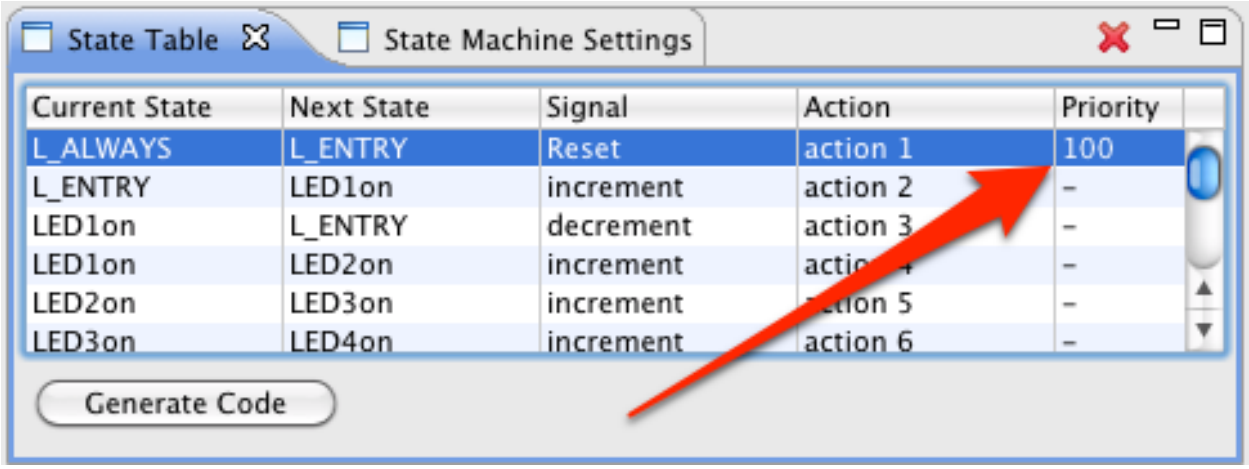


Figure 9.9. Setting the priority of a transition

Turning on `LED1`

The next transition we'll add will be from `L_ENTRY` to `LED1on`. It occurs when the counter reaches speed — i.e. when `maxcount` is high — and when the DOWN button is not pressed. When this transition occurs we turn on LED1.

To build this transition, start by adding a new signal in the Signals panel and name it increment. Since we want to combine two inputs, right-click on increment and choose **Add AND**. You'll notice that there are two (**none**) elements now in the bottom half of the signals view. Right-click on one of these and choose **DOWN** from the **Input** menu. Right-click on DOWN again and change its I/O condition from HIGH to LOW in the **I/O condition** context menu. Now right-click on the other element and choose `maxcount` from the **Input** context menu. See Figure 9.10.

In the editor use the **Transition** tool to create a new transition from `L_ENTRY` to `LED1on`. Right-click on the transition label and set the signal to increment. Finally, turn on `LED1` by choosing **Add Action -> SET -> LED1**.

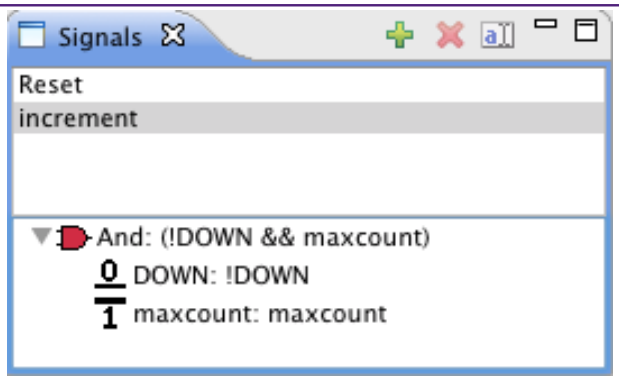


Figure 9.10. A composite signal

Turning off `LED1`

The next transition we will add is from `LED1on` back to `L_ENTRY`. This transition should happen if we are in state `LED1on` and the DOWN button is pressed (i.e. the DOWN signal is high) and the counter has reached speed.

Add another signal and name it decrement. Repeat the steps for the increment signal but leave the I/O condition for DOWN as HIGH. Draw the transition from `LED1on` to `L_ENTRY` and set decrement as its signal. Add the action element **CLEAR LED1** to turn off the LED.

Remaining transitions

Add the remaining transitions, with action elements to turn off the current LED and turn on the next one according to whether the DOWN input is HIGH or LOW. Note that you can reuse your increment and decrement signals — there is no need to create multiple copies of the same signal.

Finally, we want to reset the counter every time that it reaches the value defined by speed. Add a signal that fires when `maxcount` is high, then add a loop-back transition on `L_ALWAYS` that uses that signal and calls the function **Limit low counter**. This function instructs the counter that it has reached its limit and should reset to zero. This behavior is the default for the counter reaching its limit — see NXP's documentation on the SCT for other behaviors.

The completed blinky state machine should look like Figure 9.11.

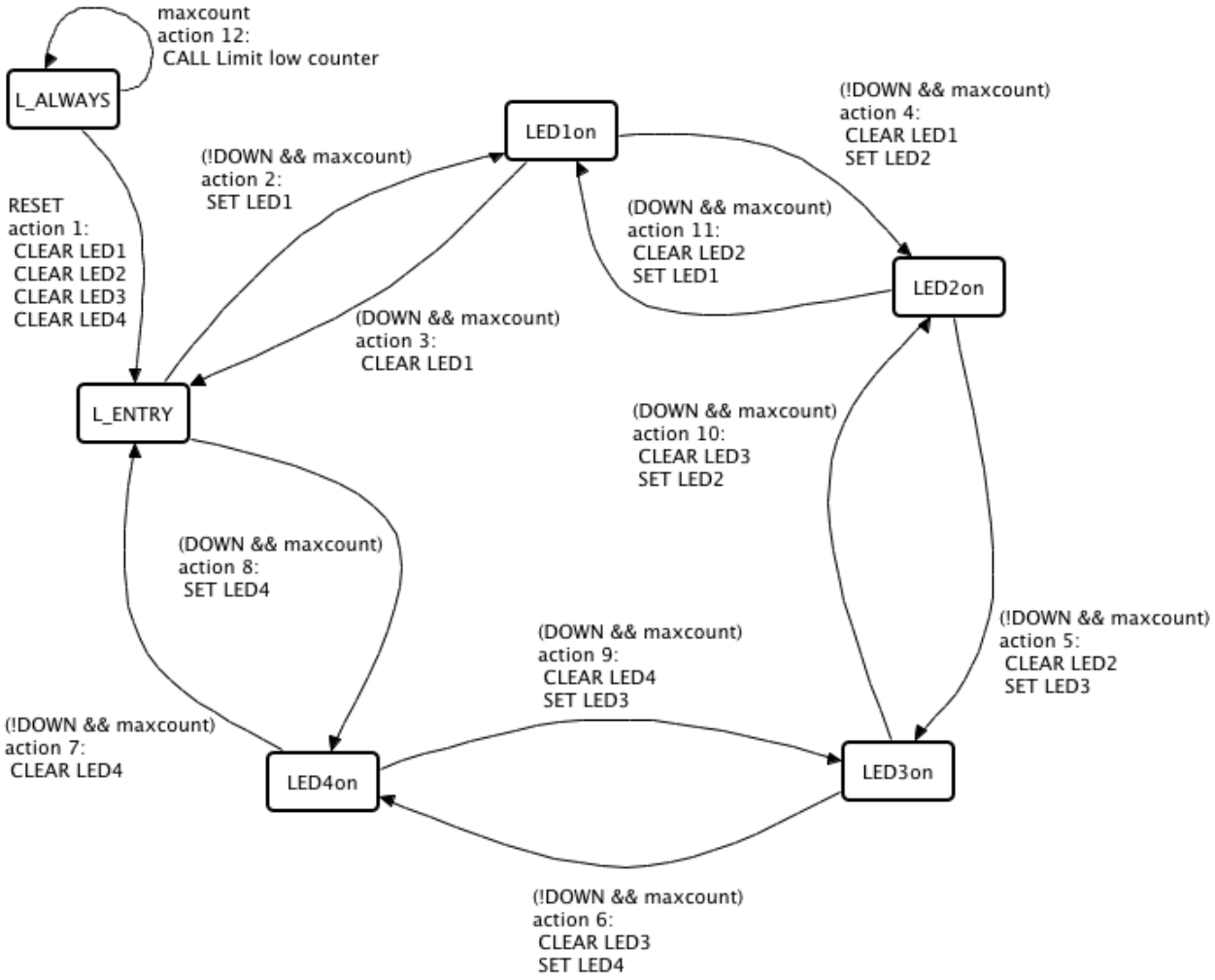


Figure 9.11. The complete state diagram for the SCT blinky example

9.2.9 Generating the configuration code

The SCT is configured by setting values to registers. The code that performs this configuration is created by Red State when the **Generate Code** button in the **State Table** is pressed or when **Generate Code** is selected from the context menu in the editor. This code can now be run on the target.

Note:
If the state machine is altered, the code has to be regenerated — the code generation is not performed automatically when you build your project.

Files generated

When the **Generate Code** button is pressed four files are generated, overwriting any previous versions. The first file is the `smd` file, which contains the description of the state machine to be processed by NXP's parser. NXP's parser then generates the `sct_fsm.h` and `sct_fsm.c` files that contain the configuration code for setting up the SCT. Red State also generates the `sct_user.h` that sets up the constants used in the `sct_fsm.h` and `sct_fsm.c` files.

Issues and warnings

The state machine needs to operate within the constraints of the SCT. Red State generates warnings if the state machine cannot be implemented on the SCT. The configuration files are not be generated or updated in this case.

Most issues will be detected as you construct the state machine and will be highlighted by red text in the editor. Hovering the mouse cursor over the warnings will provide more information in a pop-up box.

Some problems may only be detected when code is generated. These errors will be listed in the message box alerting you to the failure.

9.2.10 Incorporating with your code

The Red State tool generates the code to program the state machine defined in the editor into the SCT registers. The function `sct_fsm_init()` must be called to program the SCT. This function is declared in the `sct_fsm.h` header file. Since the SCT can only be programmed when it is stopped and will need some configuration in your code. This configuration can be done using macros included from `lpc18xx_sct.h` or `lpc43xx_sct.h`. This part of the configuration is not automatically generated and is implemented in the `sct_main.c` file for this tutorial.

9.3 Red State : Software state machine tutorial

9.3.1 Software state machine tutorial overview

This section goes through the process of using Red State to create a software state machine and deploying it to a target. This example uses the RDB1768 debug board, extending a base project provided in the LPCXpresso IDE.

Building a traffic light example

We are going to make a traffic light system for pedestrians to cross a busy street. This will be implemented using four outputs hooked up to LEDs and one input hooked up to a button. We'll assume that the first three LEDs correspond to the red, amber and green traffic lights and the fourth lights up a walk sign for the pedestrians.

The traffic lights are green until a pedestrian presses a button. This tutorial implements the USA traffic light transitions:

green only -> yellow only -> red only -> green only...

When a pedestrian presses the button it will change the traffic lights from green to red, then turn on the walk sign for a period. Then the walk sign will be turned off and the traffic lights set to green.

9.3.2 Creating a new project

Importing the base project

Start by importing the library project for the RDB1768 board. Select **import project(s)** from the **Quickstart Panel** (visible in the **C/C++ perspective**) — see Figure 9.12.

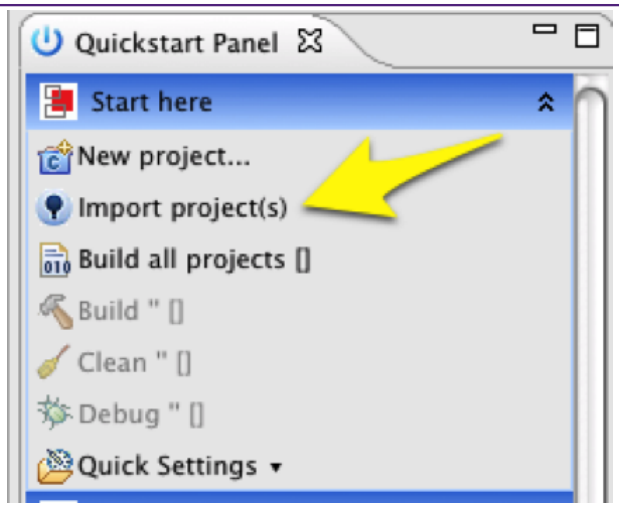


Figure 9.12. Importing projects

Press browse for the project archive zip text box and locate the `RDB1768cmsis2.zip` file in the `Examples/NXP/LPC1000/LPC17xx/` folder. Select it then press **Next** to get a list of projects contained in the archive.

The archive contains several projects, we are only interested in three of them. The `CMSISv2xx_LPC17xx` project contains the required code to use the board. The `RDB1768cmsis2_RedStateLedTraffic_base` project provides the starting point for the tutorial. The `RDB1768cmsis2_RedStateLedTraffic` project contains the working solution that would be generated by following this tutorial.

Select the three project mentioned above and click **Finish** to import them into the workspace. This tutorial assumes that you are using the `RDB1768cmsis2_RedStateLedTraffic_base` project.

9.3.3 Extending the LED Traffic base project

Switch to the **C/C++ perspective** and look at the `main_ledflash.c` file. Elements of the file are excluded using the preprocessor command `#if`. Setting the value defined to `ADD_REDSTATE_CODE` to `1` will enable the state machine code. Since we have not generated the code yet it should be left set to `0`. In this case, the main while loop contains an infinite loop which increments an integer. This section demonstrates how to replace the content of that while loop with the state machine created by Red State.

Add the state machine to the project

The next step is to use the **New state machine Wizard** to add the state machine. Make sure that the `src` folder containing the C source files is selected and then select **New->Other** from the file menu or press the **New** button in the toolbar to bring up the Wizard selection dialog.

Type `state` in the filter box to find the **Red State Machine file generator** — see Figure 9.13.

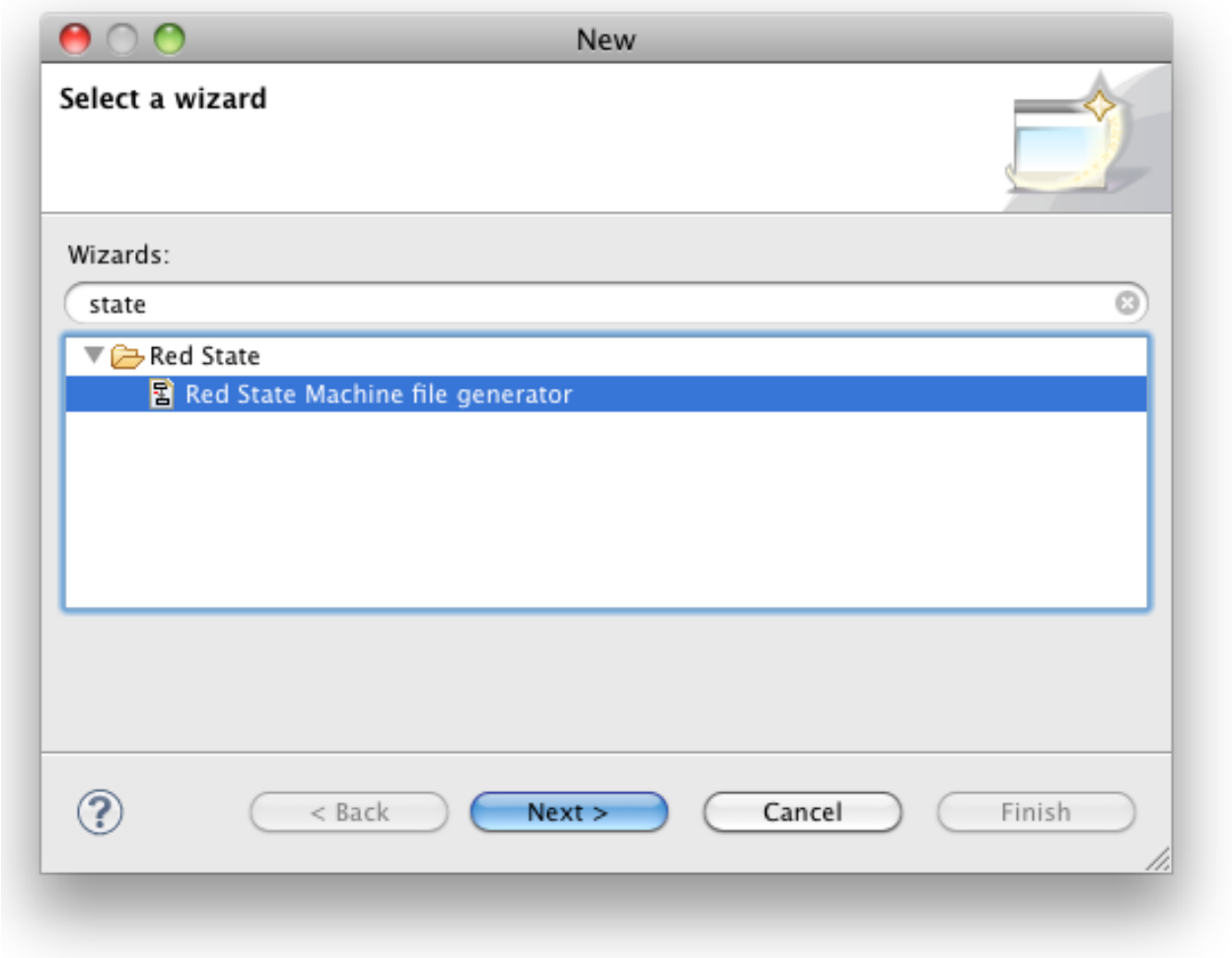


Figure 9.13. Red State Machine Wizard

Select **Red State Machine file generator** and press **Next**. In the **New State Machine** dialog make sure that the `src` folder is selected. Then enter the file name for the state machine file (e.g. `ledState`) and press **Next** (pressing **Finish** straight away would create the default SCT state machine rather than a software state machine).

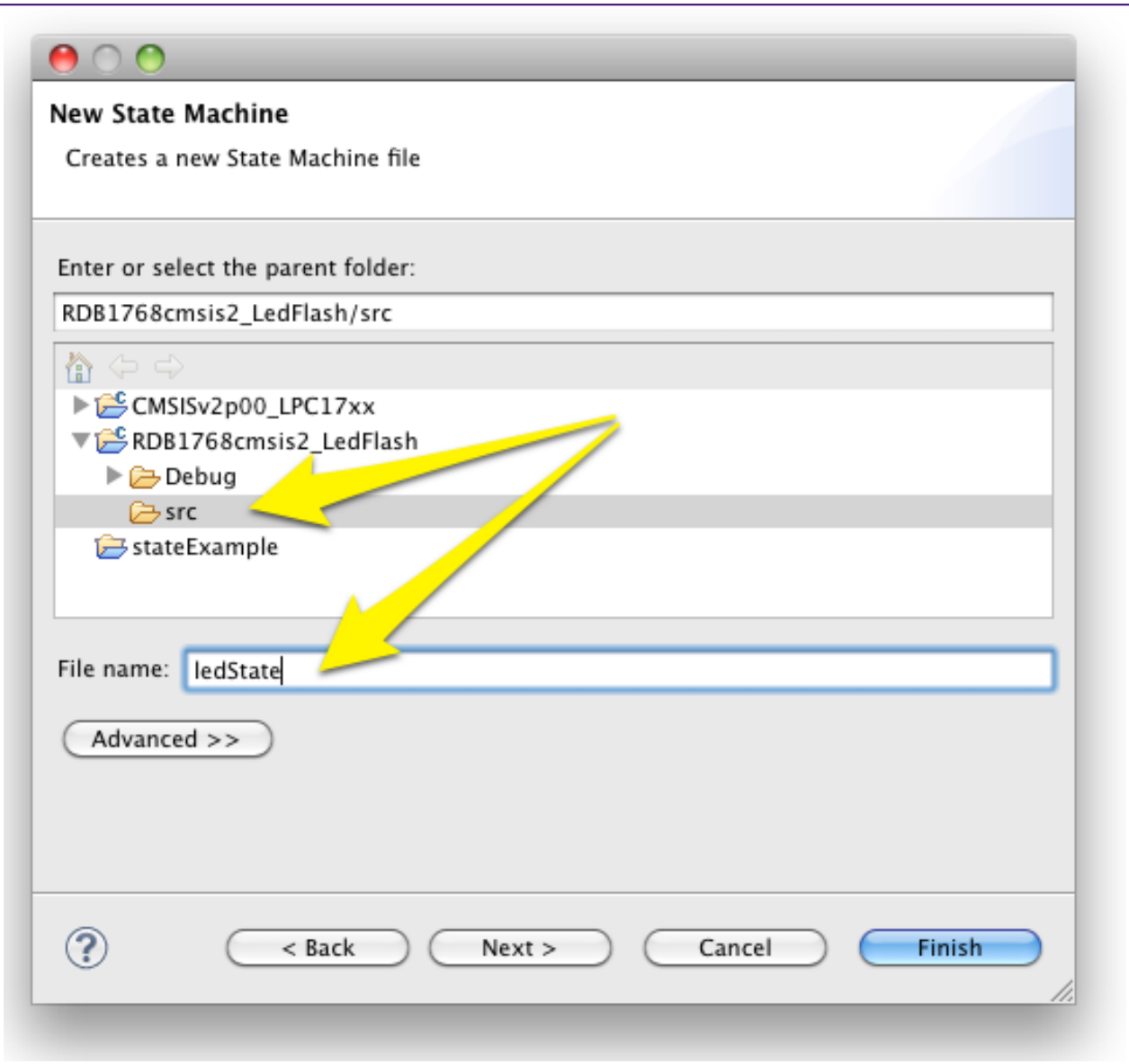


Figure 9.14. Choosing a filename for a new state machine

Select **Software state machine** from the **Target** drop-down list, and keep the other default options. For a full description of the options see the *Software State Machine Wizard options* [122] .

Press **Finish** to complete the Wizard. the LPCXpresso IDE will switch to the Red State perspective. The newly created state machine `.rsm` file is added to your source folder and opened in the **State Machine diagram editor**.

Finally, select the **State Machine Settings** view (usually behind the **State Table** view). In the prefix field enter the text `pf`. This string will be used to prefix components in the generated C code. Prefixes are used to avoid naming conflicts between different state machines in the same project.

Adding states to the State Machine

The traffic light will be modeled using six states – see Table 9.1. The Wizard should have already created the initial state: `state 1`. Create the remaining 5 states in the editor by

doing the following: first, click on the **state** button in the **state machine diagram editor**; second, click in the editor where you'd like to place the new state and then release the mouse button — see Figure 9.15.

The new state is automatically assigned a name (`state 2` in this case). Rename the state by single-clicking to select it and then single-click it again. Note that double-clicking will not work — two single-clicks must be performed in succession.

Table 9.1. States used by Traffic Light example

State	Description
state 1	The initial state of the state machine
Green	The green light is on and all others are off
Yellow	Only the yellow light on
Red	
Walk	The walk and red lights are on
Don't walk	The red light is on only (for after the Walk light was been on.)

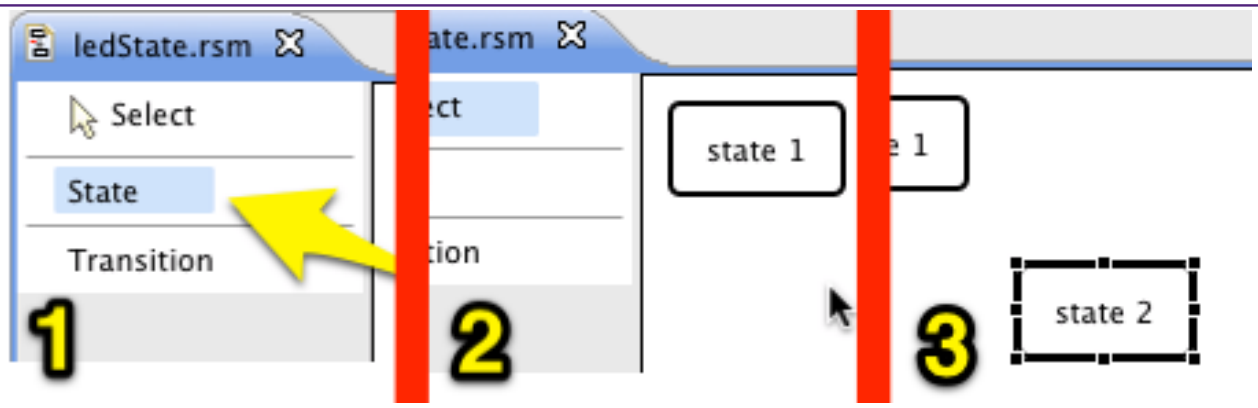


Figure 9.15. Adding a new state in the State machine editor

Rename `state 2` to `Green` and add the remaining four states in the same way, naming them as shown in Table 9.1. States can be repositioned by dragging them and can be resized by dragging the handles of a selected state.

You should now have the states laid out in the state machine editor as in Figure 9.16.

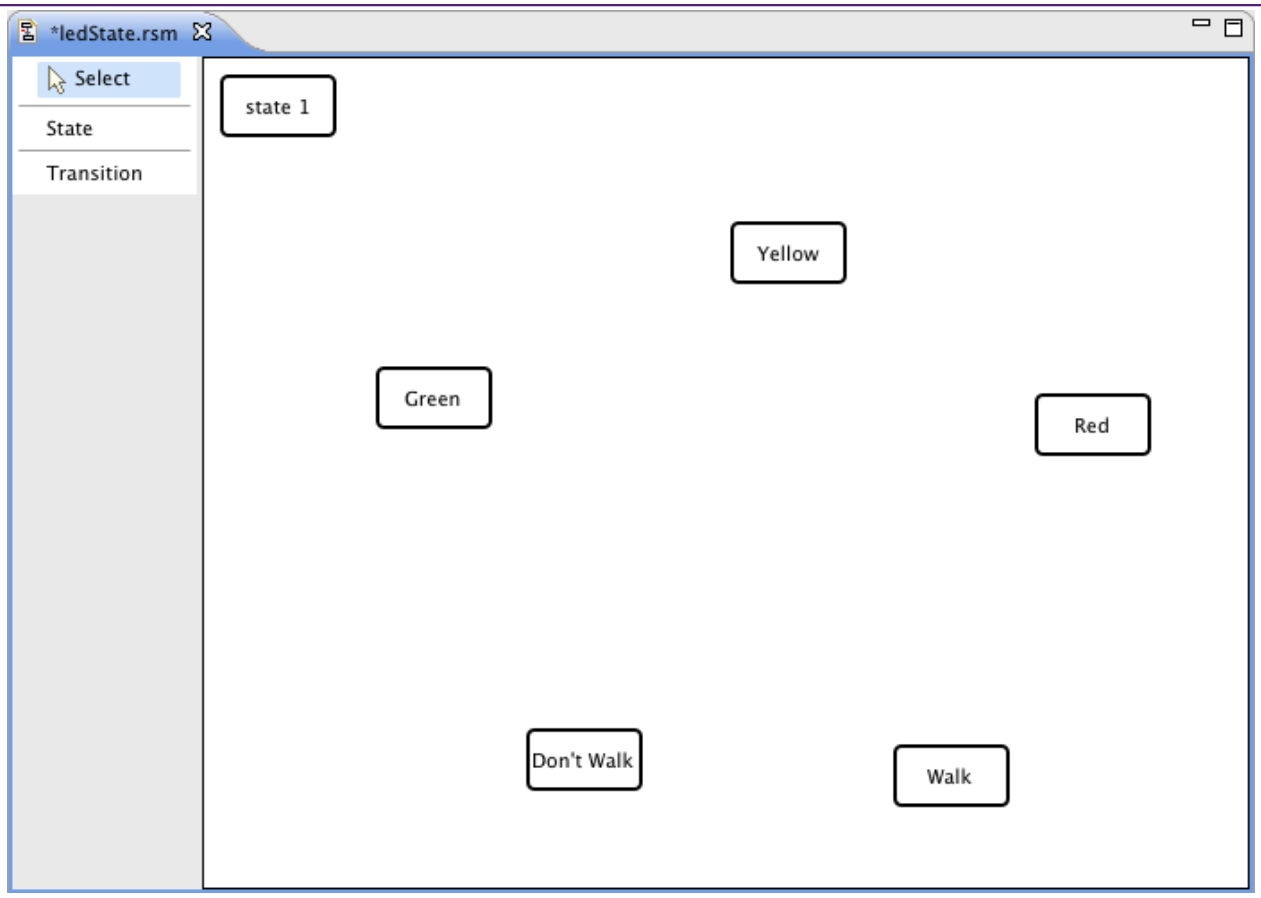


Figure 9.16. Layout of the states in the State machine editor for the software state machine.

Adding inputs

The values of inputs can be read by the state machine but cannot be set by the state machine. The traffic lights will have two inputs: the button for the pedestrians to press to change the lights and a reset signal. Add these two inputs in the **Inputs for State Machine** panel by pressing the add button twice. Delete accidentally added inputs by selecting their row and pressing the **Delete** button.

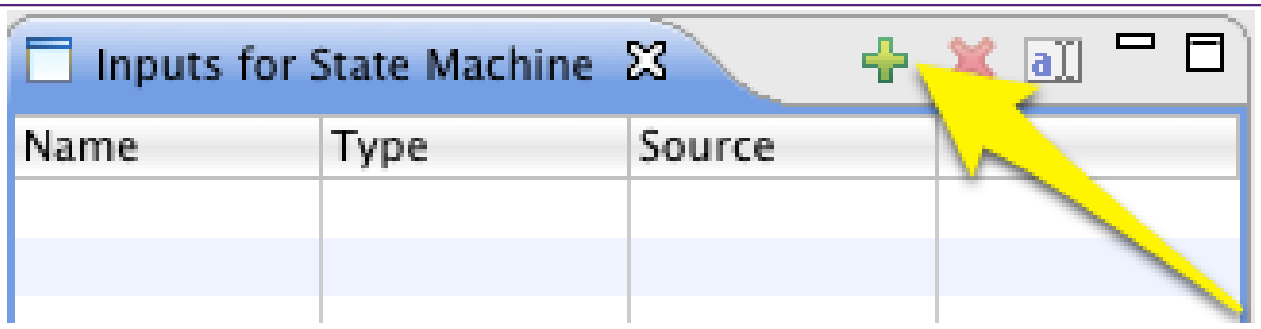


Figure 9.17. Adding inputs for the Software State Machine

Edit the names of the inputs by clicking in the Name column, naming one `buttonPushed` and the other `reset`. The source column defines the type of these inputs — keep their source as “int”.

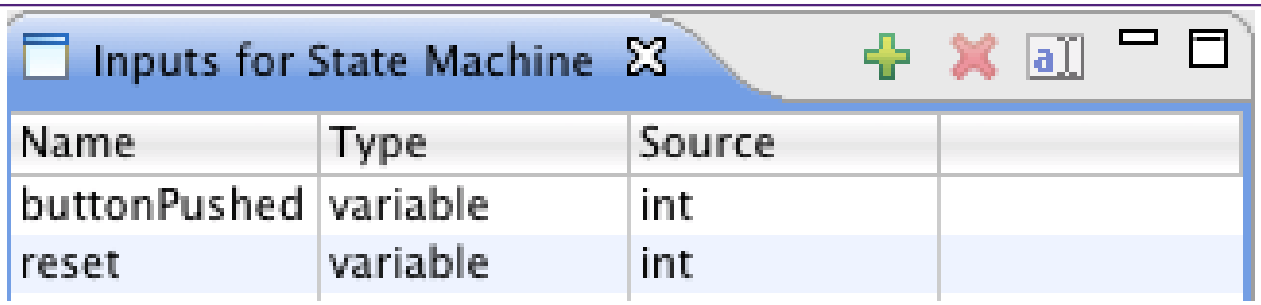


Figure 9.18. Added inputs for the Software State Machine

Adding outputs

The **Outputs for State Machine** panel includes variables that can be read and set by the state machine as well as functions that may be called by the state machine. This example uses functions to turn the LEDs on and off as well as variables for timing the different stages of the lights.

Use the **Add** button in the **Outputs for State Machine** panel to add 10 outputs. Name two of the variables `timer` and `duration` to be used for the timing of the lights. Click on their name to rename them in the table, keeping the type as **variable** and the source as **int**. Enter `0` (the numeral zero) in the preload column for both variables. The variables will be initialized to these preload values when the state machine is reset.

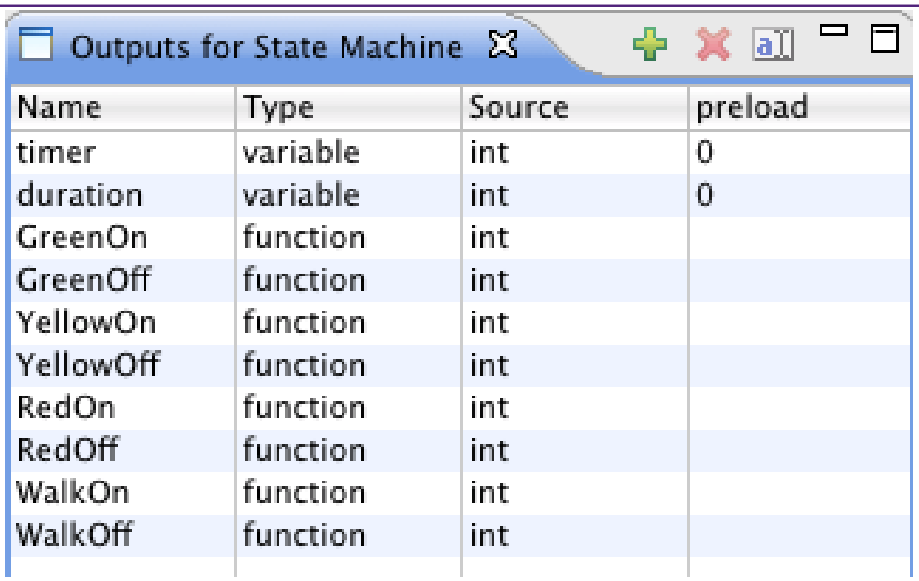


Figure 9.19. Added outputs for the Software State Machine

Next create the functions. Each light needs two functions: one to turn it on and one to turn it off. Edit the next row of the output panel to set the name to `GreenOn` and the type to **function**. The source column is ignored for functions. Make the rest of the outputs into functions for turning on and off the different lights — see Figure 9.19.

The Initial State and the Reset signal

After a reset, the state machine enters the initial state named `state 1`. The software state machine requires that the initial state and a reset signal be defined. If you selected **include**

initial state in the Wizard there will be a state called `state 1` already in the diagram. The state machine will be initialized this is the state on reset. If **include RESET signal** was selected in the Wizard, then there will also be a signal called `RESET` in the **Signals** View.

When the RESET signal is detected the current state of the state machine transitions to the initial state and all outputs are set to their preset values.

The initial state and the RESET signal are defined in the **State Machine Settings** view that is the tab behind the **State Table** by default — see Figure 9.20.

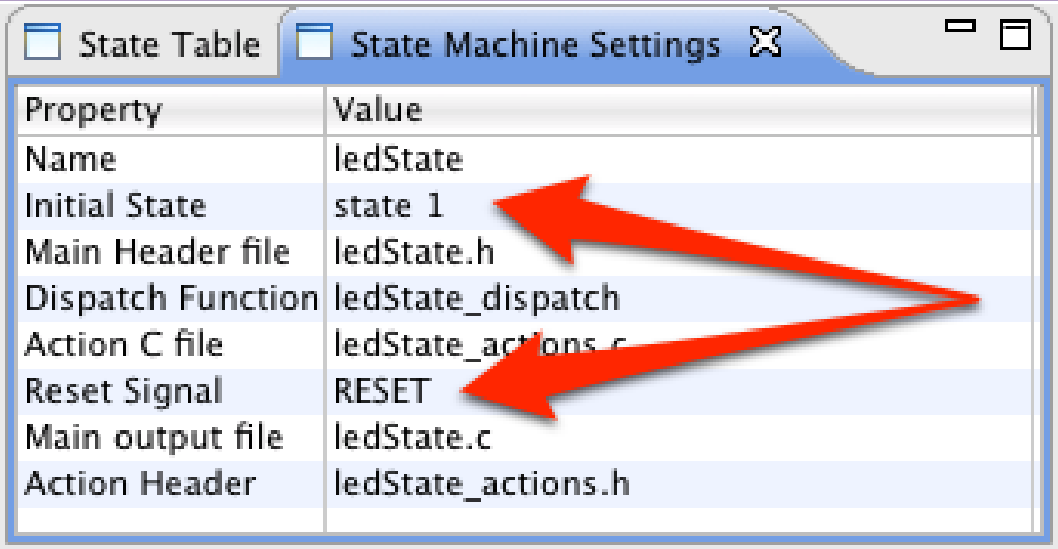


Figure 9.20. Setting the initial state and the reset signals in the State Machine Setting panel

Adding a transition

From the initial state, the state machine will transition to the `Green` state and call the appropriate light functions to set just the green light to be on. Add a transition from `state 1` to the `Green` state by choosing the Transition tool, click on `state 1` and then on the `Green` state — see Figure 9.21.

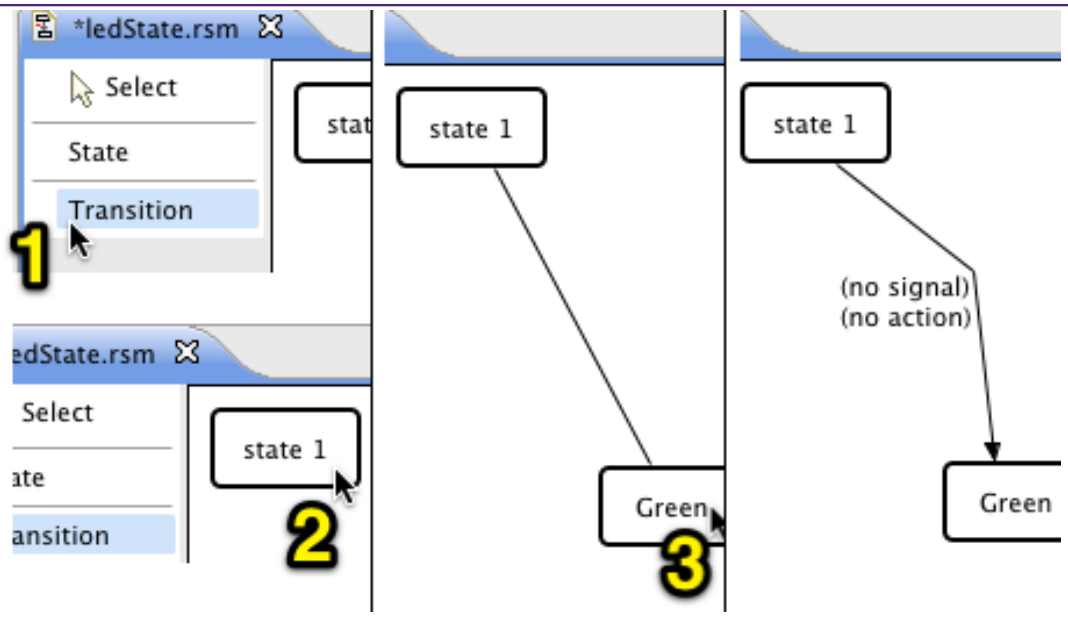


Figure 9.21. Adding a transition to the state diagram

Creating a signal

Select the **Add signal** button in the Signals panel to add a new signal. Double-click on the name to change it and enter `always`. Make this signal always evaluate as true by comparing a variable to itself. Select the row in the bottom half of the signal panel that says "(**click to change**)". Right-click on it and select **Add MATCH** from the context menu — see Figure 9.22.

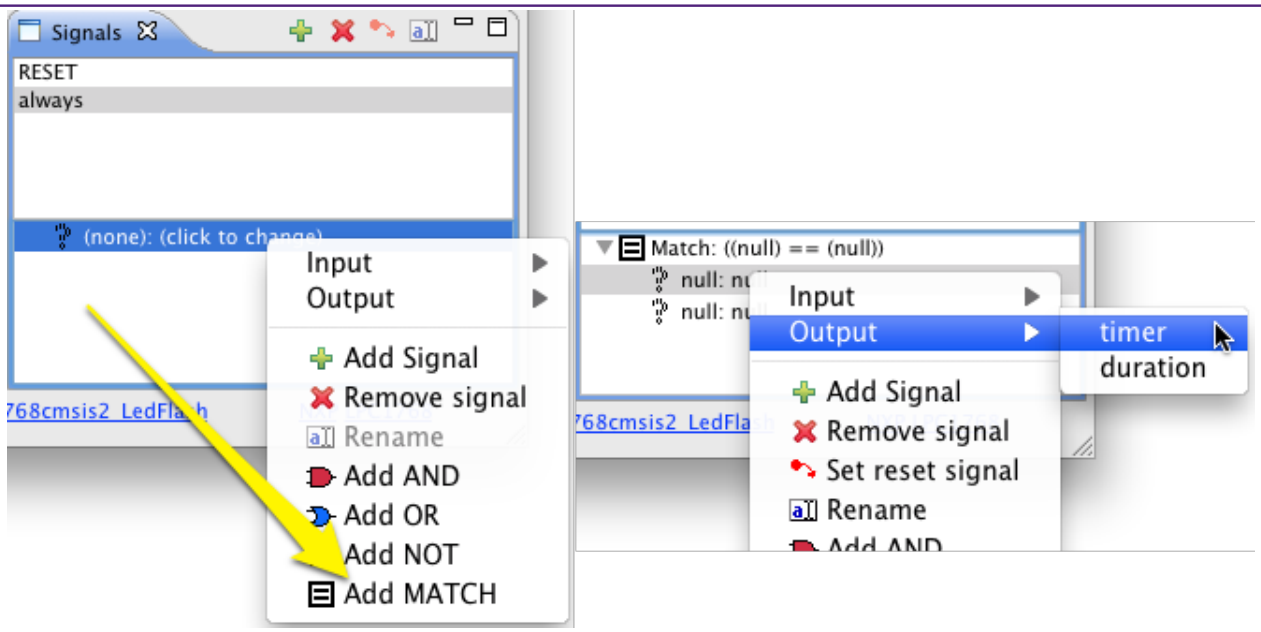


Figure 9.22. Adding a match condition to a signal

The match compares two elements. Initially these are both null and have to be changed. Right-click on the first null and choose **Output -> timer** from the context menu. Make sure that the element to be set is selected in the bottom half before right-clicking; otherwise the whole signal may get replaced.

Do the same for the other `null` value to build the match condition: `(timer == timer)`.

Adding a signal to a transition

Add the `always` signal to the transition from `state 1` to `Green` by right-clicking on the transition label (the text beside the arrow that says (**no signal**)) and selecting **Set Signal -> always** from the context menu.

Figure 9.23. Adding a signal to a transition

Adding actions to a transition

A single action is associated with a transition. That action can be composed of multiple action elements, allowing multiple outputs to be set and functions called when the transition's signal is fired and the state machine is in the transition initial state.

When the state machine transitions from `state 1` to `Green`, the green light should turn on and the other lights should all be off. To implement this behavior right-click on the **no action** text beside the arrow and select **Add Action -> CALL -> GreenOn**. Then repeat this procedure to call the functions that turn off the other lights.

Remove accidentally added action elements by right-clicking on the transition label and choosing the item from the **delete action element** list. Note that the **Delete** option in the context menu will delete the entire transition.

The first transition should now be labeled with the text:

```
(timer == timer)
action 1:
 CALL GreenOn
 CALL YellowOff
 CALL RedOff
 CALL WalkOff
```

Transition on button press

The next transition is fired when the user presses the button – that is, when the input `buttonPushed` is non-zero. Add a signal and name it `didPush`. Right-click on the signal element in the bottom half and set it by selecting **Input -> buttonPushed** from the context menu.

Next add a transition from `Green` to `Yellow`. Click on **Select** to switch from transition adding mode and right-click on the transition label to set the signal to `didPush`.

Add the actions to turn off the green light and turn on the yellow light using the context (right-click) menu.

A loop-back transition, incrementing a counter, will be used to hold the light on yellow for a period. First, we set the timer variable to zero by adding the action **Set timer**. To do this you right-click on the transition label, select **Add Action -> SET -> timer**. To enter the value we need to edit the action element in the **Action list**. The transition label identifies the action associated with it on its second line, for example `action 2` in this case. Select `action 2` in the Action List and you'll see the **SET** operation in the lower half of the **Action List**. Click in the **Value** column and enter the number `0` — see Figure 9.22.

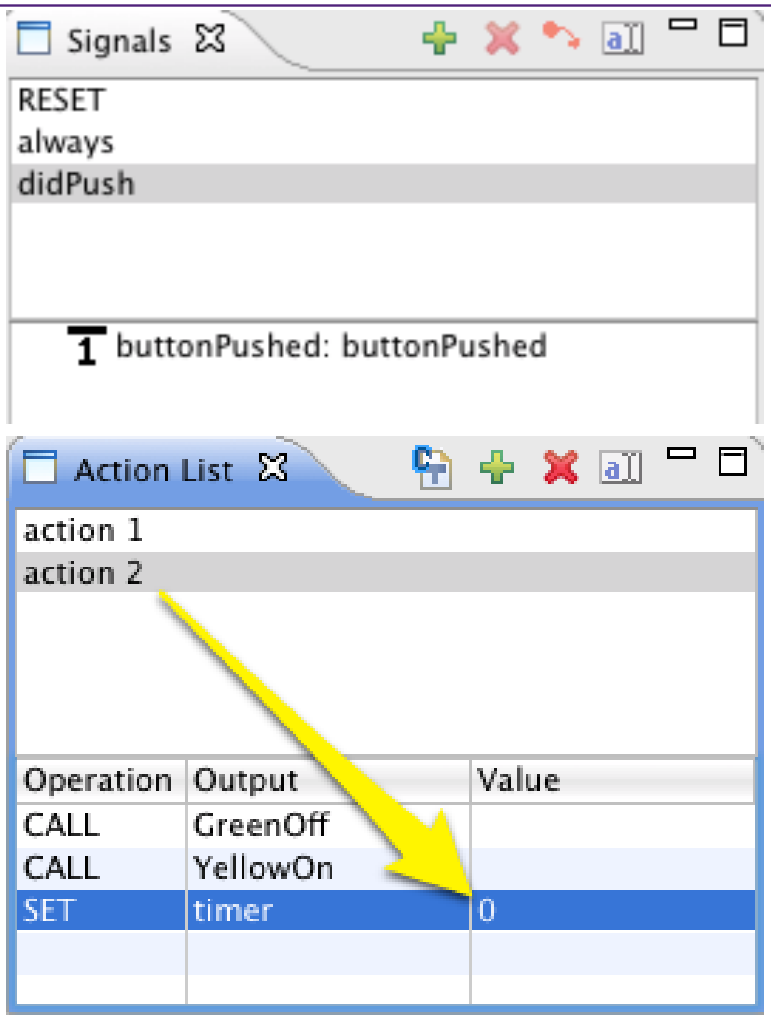


Figure 9.24. Adding a match condition to a signal

Now set the `duration` variable to `500` by following the same process.

In the `Yellow` state we'll add a loop-back transition which increments the timer while the timer does not equal the variable `duration`. Add a new signal and name it `wait`. Add a **Not condition** and then a **match condition** to the signal using the context menu. Add `timer` and `duration` to the **Match condition**. Add another function in the **Output** list and name it `incTimer`, remembering to set its type to **function**.

To make the loop-back transition, select **Transition** as usual and single-click on the `Yellow` state. Single-click on the state again to finish the transition. Add the wait signal to it and the call the `incTimer` function.

Add a new signal and name it `continue`. Edit the signal in the Signal panel so that it matches the timer to the variable `duration`. Drag the loop-back transition label so that it is above the `Yellow` state. Now add a transition from the `Yellow` state to the `Red` state with signal `continue` and add actions to turn off the Yellow light, turn on the Red light, set the variable `timer` to zero and the variable `duration` to 250.

The loop-back holds the state machine in the `Yellow` state until it has looped `duration` number of times. Then the `continue` signal makes the state machine transition to the `Red` state.

Add similar transitions to move through the remaining states, pausing in them for appropreate periods — see Figure 9.25.

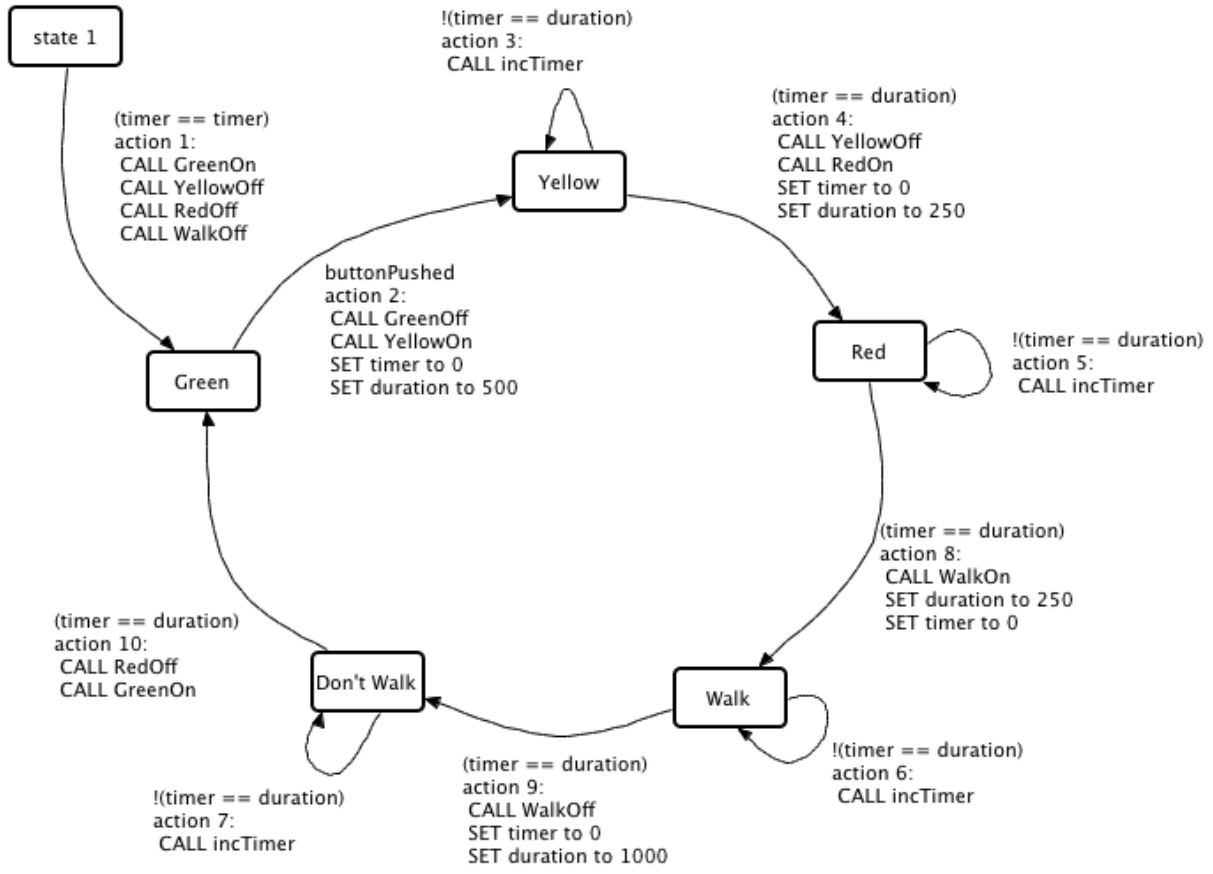


Figure 9.25. Layout of the traffic light software state machine.

9.3.4 Integrating a state machine with existing code

We now have our completed state machine. Next we need to generate the C code for the state machine and hook it up to the existing code which turns LEDs on and off and handles external interrupts.

The state machine is updated by calling its dispatch function with inputs and outputs. The name of the dispatch function can be set in the **State Machine Settings** panel. Its default name is `ledState_dispatch` for this example. The inputs and outputs for the state machine are defined by two `structs` in the main header file (e.g. `ledState.h`).

Editing `main`

The `main` function in `main_ledflash.c` contains a while loop which increments a variable. We will be replacing the content of that while loop with a call to the dispatch function.

First, we need to set up the inputs and outputs. The input struct will be a global variable whose type is defined in `ledState.h`. Add the `ledState.h` header file to `main_ledflash.c`.

```
#include "ledState.h"
```

Then enter the following before the main function call:

```
pf_inputs input; // global access to the input
```

Next, we initialize the inputs between initializing the LEDs and the start of the while loop. Also, add the local output `struct` and initialize it by calling the dispatch function with the reset signal set to `1`. This reset will set the current state to the initial state and fill the output with their preload values.

```
// initialise the inputs and outputs:
input.buttonPushed = 0;
input.reset = 1;

pf_outputs out;
ledState_dispatch(input, &out);
input.reset = 0;
```

Replace the body of the while loop with a call to the delay function and the call to the dispatch function:

```
while(1) {
 short_delay(10); // slow it down
 ledState_dispatch(input, &out);
 i++;
}
```

Generating the state machine code

We now need to generate the code for the dispatch function. With the state machine editor selected, press the **Generate Code** button in the **State Table** panel. This should add four files to your project. The names of these files are defined in the **State Machine Settings** panel. By default you should have the following files:

- `ledState.h` — Header file for the logic of the state machine.
- `ledState.c` — The logic of the state machine. Contains the dispatch function and the input and output data structures.
- `ledState_actions.h` — The header file for the action functions.
- `ledState_actions.c` — The functions that are called by the state machine.

The `ledState_actions.c` file is initially empty. It can be filled with the required template functions by pressing the **Generate Action C Template** button in the **Actions** panel or by right-clicking in the **State Machine editor** window and selecting **Generate Action Template**.

Warning

Generate Action Template will overwrite any existing content in your `ledState_actions.c` file as well as regenerating the other three files.

You are expected to edit the `ledState_actions.c` file. The other files are regenerated each time you press the **Generate Code** button. You are strongly advised not to edit these other files, as your changes will get overwritten.

Editing the actions C file

Press the **Generate Action C Template** button to make the function bodies for the action functions. The LedFlash example contains the `leds.h` and `leds.c` files which contain code for turning on and off the LEDs. Include the `leds.h` in the `ledState_actions.c` file.

```
#include "leds.h"
```

The LEDs can be turned on by the function `led_on(int)` and off using `led_off(int)` passing it a reference to the LED as the parameter. The LEDs are identified by the constants defined in `leds.h`, see Table 9.2.

Table 9.2. The LED ids

Constant	Representation
LED_3	Red
LED_2	Yellow
LED_5	Green
LED_4	Walk Sign

Note that the unusual ordering of the LEDs is due to their layout on the RDB board.

Enter the appropriate calls for turning the LEDs on and off in the function bodies.

```
/* Action: GreenOff */
void act_GreenOff()
{
 led_off (LED_5); //  green led
}

/* Action: GreenOn */
void act_GreenOn()
{
 led_on (LED_5); //  green led
}

/* Action: RedOff */
void act_RedOff()
{
 led_off (LED_3); // Red
}

...
```

Accessing the outputs

The output variables are stored in a `struct` defined in the `ledState.h` file. Its name is of the form `prefix_outputs`, where `prefix` is defined in the **State Machine Settings** view.

A pointer to the output struct is passed to the dispatch function. From within the action functions you can get this pointer by calling the `prefix_getOutput()` function. We use this pointer to increment the timer value:

```
/* Action: incTimer */
void pf_incTimer()
{
 pf_outputs * out = pf_getOutput();
 out->timer++;
}
```

Setting inputs in interrupt handlers

The state machine will now happily sit in its `Green` state. Now we need to hook up an external interrupt so that the state machine knows when a pedestrian has pushed a button. We use code imported from the `RDB1768cmsis2_ExtInt` project — the `eint0.c` and `eint0.h` files. This code is included in our base project.

Include the `eint0.h` header file in the `main_ledflash.c` file.

```
#include "eint0.h"
```

Next add the following line to initialize the interrupt immediately after the LEDs have been initialized.

```
// Setup External Interrupt 0 for RDB1768 ISP button
EINT0_init();
```

In the `eint0.c` file replace the `#include "leds.h"` with `#include "ledState.h"`. Declare `input` as an extern global with the following line:

```
extern pf_inputs input;
```

Replace the call to `leds_invert()` from the `EINT0_IRQHandler()` in the `eint0.c` file with

```
input.buttonPushed = 1;
```

Now when the ISP button on the board is pressed, the `buttonPushed` input will be set to `1`. Finally we need to have it reset to `0` after the dispatch function has processed the event. This reset is accomplished by setting the `buttonPushed` input to `0` after the dispatch function is called. Place the following line after the call to `ledState_dispatch()` in the while loop in the main function:

```
input.buttonPushed = 0;// clear button push
```

Running on the target

Now the traffic lights are ready to be deployed to the target. Switch to the C/C++ perspective and choose **Debug 'RDB1768...'** from the **Quickstart Panel**.

The LED beside the LCD will light up when this example is running on the RDB1768 board. Pressing the button labeled ISP (between the Ethernet socket and the LCD) makes the LEDs change in the traffic light sequence — see Figure 9.26.

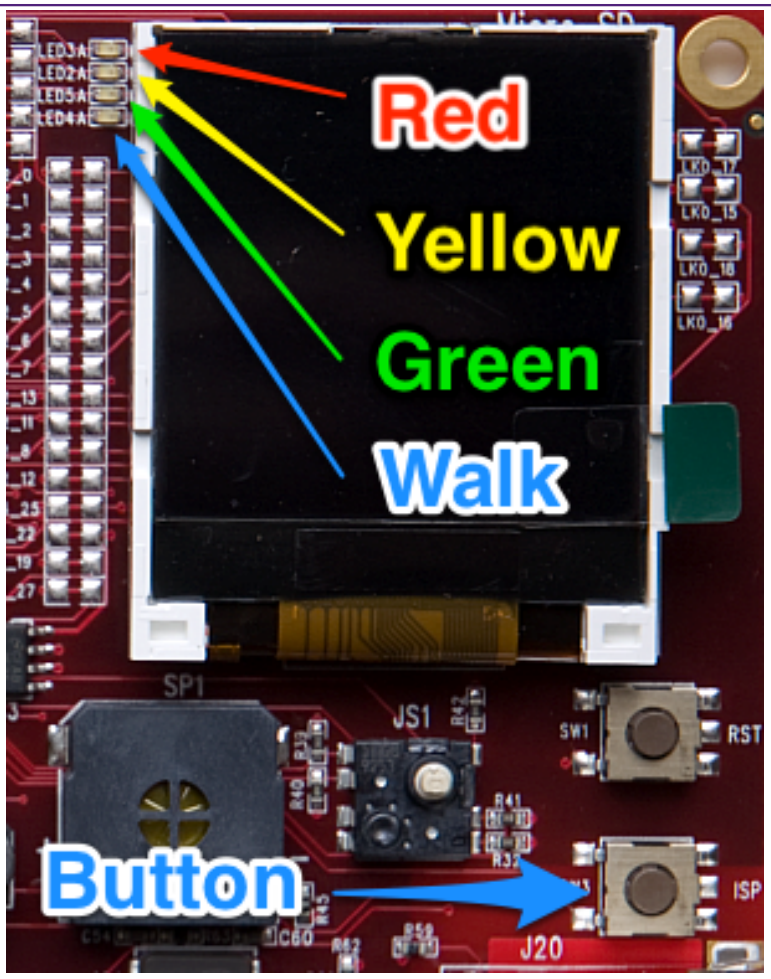


Figure 9.26. The LEDs on the RDB1768 board.

Other examples

Be sure to check out other examples in the RedState examples folder. The project `RDB1768cmsis2_RedStateTrafficLCD` shows how the same state machine can be used with different peripherals.

9.4 Red State : New state machine Wizard

The **New State Machine Wizard** can create either SCT based state machines or software state machines. Select the option you require from in the **Target** box after you have chosen the location to store the file.

9.4.1 SCT Wizard options

The **New State Machine Wizard** has several options for the creating an SCT based state machine. See Figure 9.26

- **Name**: A descriptor
- **Target**: Choose between an SCT state machine and a software state machine. For the SCT you also choose the MCU and on parts that contain multiple SCTs, the ID of the SCT to program.
- **unified timer**: The SCT can be configured as a unified 32-bit counter or it can be split into two 16-bit counters (the low counter and the high counter).

- **include initial state**: If checked, special initial states will be automatically added to the state machine. If the timer is unified, an initial state named U_ENTRY will be added. If the timer is split, two initial states named L_ENTRY and H_ENTRY will be added. These names correspond to the low and high state machines respectively.
- **include ALWAYS state**: If checked, a special *virtual* state is added to the state machine. This is in fact independent of the SCT's state and allows global or state-independent events to be triggered and represented in the state editor.
- **Main file**: This is the name of the intermediary file that will be generated from the state diagram. It will be generated relative to the folder that the state machine is saved in.

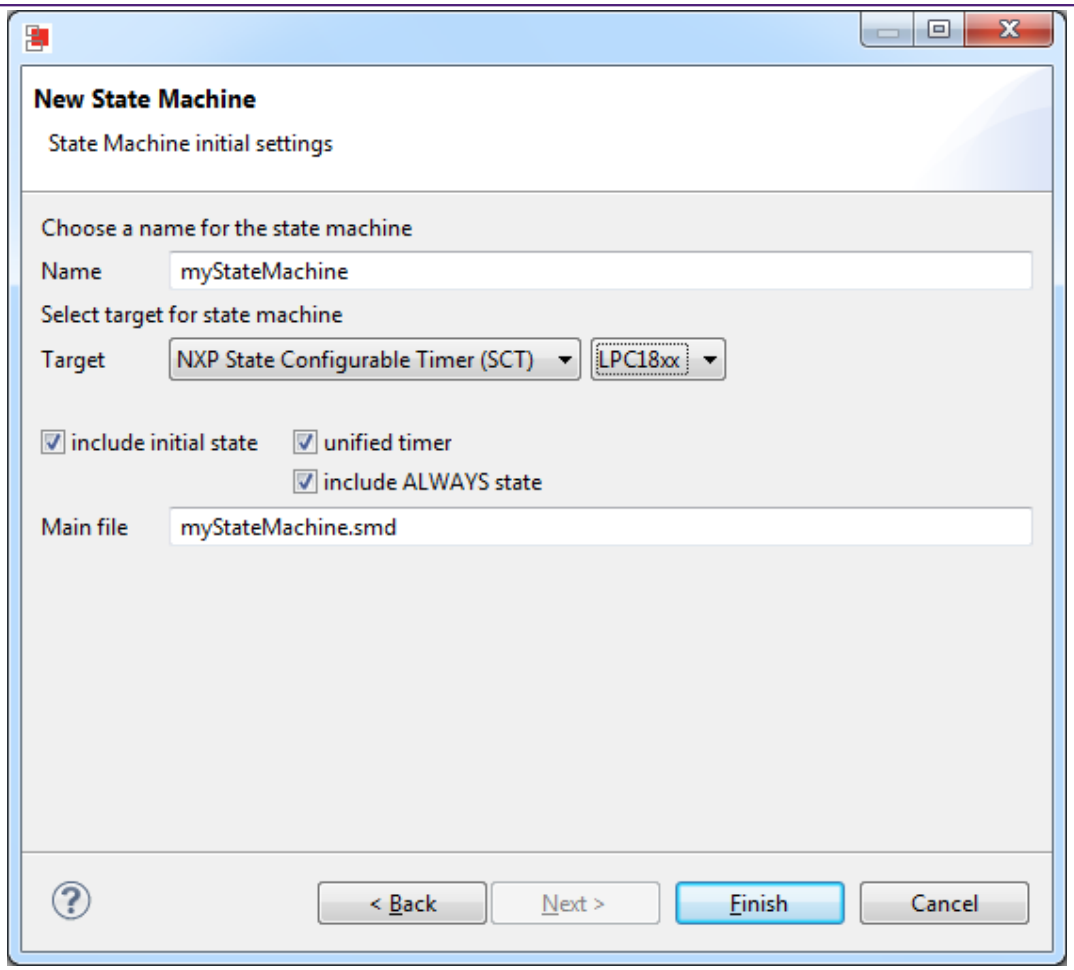


Figure 9.27. The New State Machine Wizard settings for the SCT.

9.4.2 Software State Machine Wizard options

When **Software State Machine** is chosen from the **Target** drop-down you can configure the following options:

- **Name**: a descriptor
- **Target**: choose between and SCT state machine and a software state machine.
- **include initial state**: if checked, a state will be automatically include as set as the initial state.
- **include RESET signal**: if checked, a signal will be automatically included in the new state machine and set as the RESET signal. The RESET signal is a special signal which

sets the state to the initial state and resets any output whenever it is fired regardless of the state machine's current state.

- **Main file**: what to call the main C file which will be generated by the plug-in. It contains the logic of the state machine. A corresponding header file is also created.
- **Action c file**: The state machine may call functions. These functions have to be in the action c file. The plug-in automatically generates the prototypes in the header file and can optionally generate a template for the function bodies.
- **Dispatch function name**: this is the name of the function that processes the logic of the state machine. Its inputs are passed to it in a special `struct` (defined in the main file's header).
- **Prefix**: a prefix used for naming key data structures and functions to ensure that there are no conflict between other state machines.

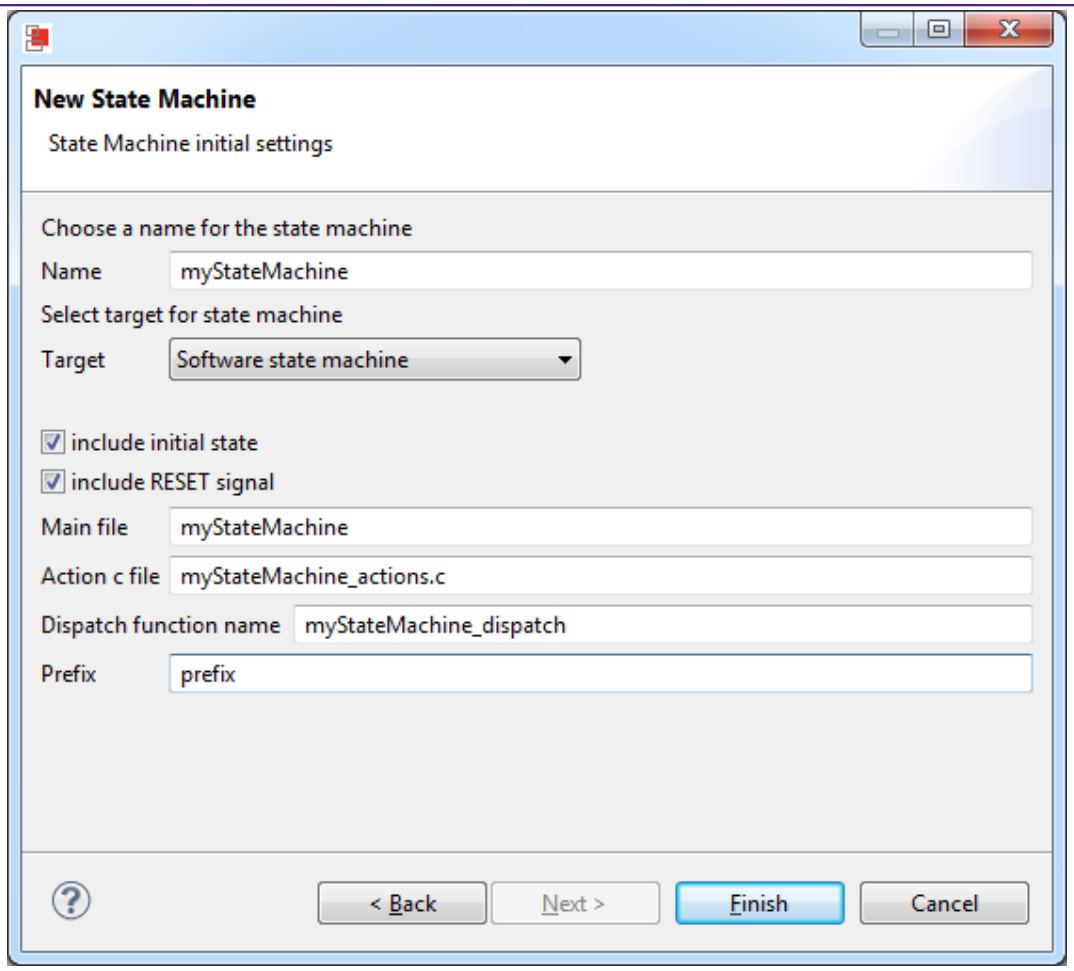


Figure 9.28. The New State Machine Wizard settings for the software state machine.

9.5 Red State : The state machine editor

9.5.1 Overview

The state machine editing perspective consists of a graphical editor and 6 views. the LPCXpresso IDE automatically switches to this perspective when you load an `rsm` file. In this section we will give you a brief overview of the main components of Red State's editor.

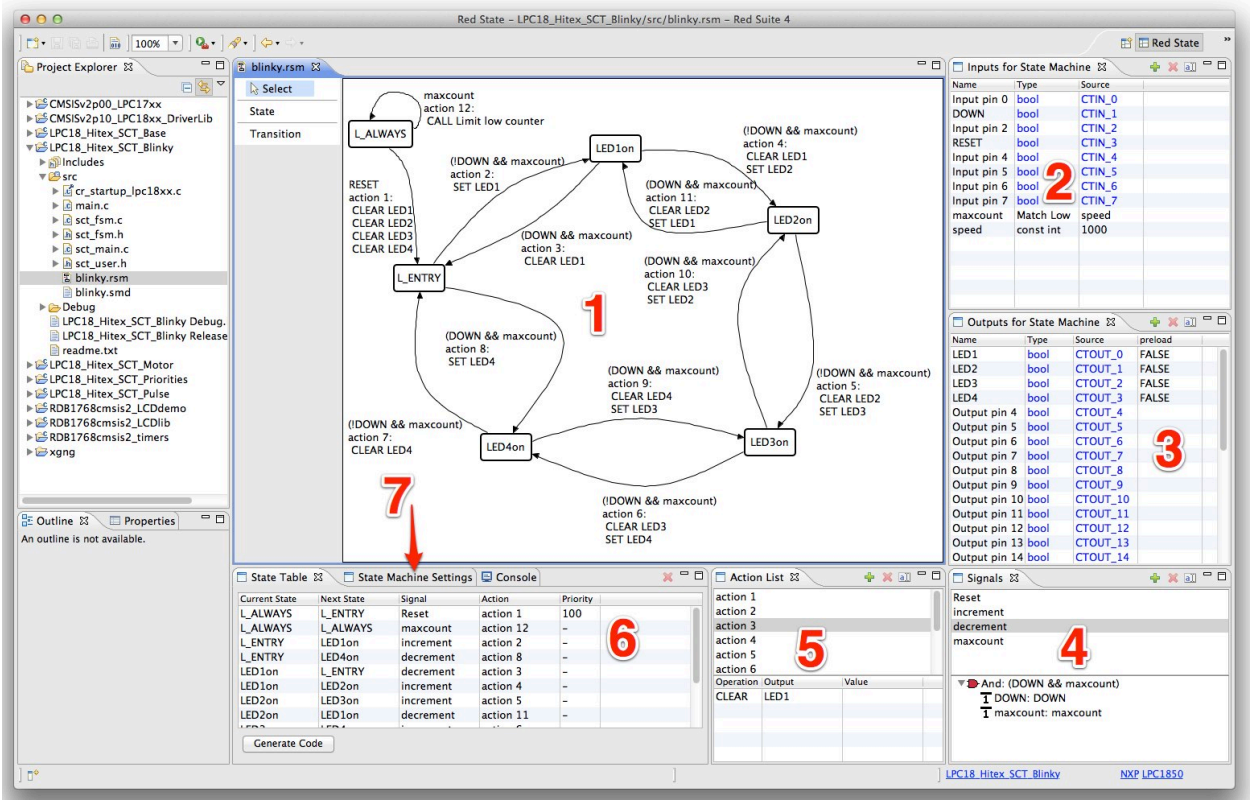


Figure 9.29. The Red State perspective

The main window is the **state machine diagram editor** — see number 1 in Figure 9.29. The editor is associated with `.rsm` files. This is where the state machine is graphically represented. Boxes represent states. Arrows show transitions between states. You can add states and transitions by selecting them from the pallet on the left. The transitions have signals and actions associated with them. Any red text in the state machine diagram indicates a problem with the set up. You can hover your cursor over the text for the full error description. Use the drop-down in the toolbar to set the zoom level of the diagram.

The **Inputs for State Machine** view — see number 2 in Figure 9.29 — defines the state machine's inputs. The state machine cannot directly affect inputs. Each input has a name, a type and a source. The name can be any string, but no two inputs can have the same name in the same state machine. Internally any non-alpha numeric characters are treated as underscores so the names `input 1`, `input#1` and `input_1` would be considered identical. Different state machines may allow different types. The source describes where an input is coming from. Note that you may not be able to edit some properties of an input. Fixed properties are written in blue text.

The **Outputs for State Machine** view — see number 3 in Figure 9.29 — defines the outputs available to the state machine. In contrast to the inputs, the state machine may directly affect outputs, for example by setting pins or calling functions. The Output view is very similar to the Input view but includes an additional column. This fourth column is the "preload" column where you may optionally enter initial values of the outputs. Note that outputs also include functions that can be called. Like the input view, blue text indicates non-editable fields.

The **Signals** view — see number 4 in Figure 9.29 — allows you to create and edit the triggers for transitions. A signal consists of a logical combination of inputs and outputs. It is split into two halves: the top lists the signal names which can be associated with a transition;

and the bottom provides the detailed view of the signal and allows it to be constructed via the context (“right-click”) menu.

The **Action List** view — see number 5 in Figure 9.29 — allows you to create and edit compound actions. Only one action can be associated with a transition, but this action may consist of multiple action elements. Action elements can include setting variables or outputs and calling functions. Like the Signals view, it is split into two parts. The top part lists the actions, and the bottom part lists the action elements associated with the action selected in the top part. Elements can be added and deleted using the context menu. A single compound action may be used by multiple transitions.

The **State Table** view — see number 6 in Figure 9.29 — lists all of the transitions in the state machine and their actions and signals. You can create new states and transitions by typing into the highlighted “(click to add)” row. Transitions can also be edited here.

The **State Machine Settings** view — behind the **State Table** view, see number 7 in Figure 9.29 — allows the user to change the name of the output file. In the case of the software state machine, the initial state and reset signals can be set here too. A prefix can also be included to allow multiple state machines to be used in the same project. For the SCT, the target MCU and the name of the header file included in the `sct_user.h` file is set here. The target SCT can be changed here which is useful if you want to copy an existing state machine file which was designed for a different SCT.

9.5.2 States

Once you have created a new state machine you can create states using either the **Diagram Editor** or the **State Table**.

Creating

In the **Diagram Editor** you select **state** from the pallet on the left and place it by clicking in the diagram. This will create a new state with an automatically generated name.

In the **State Table** you can create states by typing a name in the current or next state column. This will start to create a transition using the name you entered. If there is already a state with that name in the state machine, it will be used in the transition, otherwise a new state will be created.

Naming

A state can be renamed in the **Diagram Editor** by single-clicking on the state to select it, and then single-clicking on it again. The state names are required to be unique and not blank.

There are a few restrictions on the naming of states for SCT state machines. In the SCT there are several reserved names that indicate special states. NXP describes them as follows:

> **`H_ALWAYS`, `L_ALWAYS` (split counter mode), `U_ALWAYS` (unified counter mode)** :
>
> These are pseudo (or “virtual”) states which do not get mapped into a state register value for the SCT state machine. It is just a graphical convenience to represent events which are state independent, or in other words are considered to be valid in all defined states
>
> **`H_ENTRY`, `L_ENTRY` (counter split mode), `U_ENTRY` (unified counter mode)**

These represent the initial value of the state register the SCT will have after configuration. It is a useful feature as you might want the state machine to start from a user defined condition. If not specified, the SCT will be left in the default configuration after reset, that is, start from state zero. Note that the tool will map the state numbering at its convenience, so use the ENTRY feature if the starting state is of relevance for your application.

Resizing

The states can be resized in the diagram by selecting them and then dragging the handles at their corners. If the full state name does not fit inside the state box it will show ellipsis (...) at the end of the visible part of the name.

Deleting

States can be deleted from the **Diagram Editor** by selecting them and then either choosing **delete** in the edit menu or by right-clicking on them and choosing **delete** from the context menu.

Setting initial state

In the software state machine, the name of the initial state can be set in the **State Machine Settings** view.

9.5.3 Transitions

Red State graphically represents transitions as arrows connecting two different states, or looping back to the same state. They also have a text label describing the signal and any actions associated with the transition. A transition is composed of five elements: a *current state*, a *next state*, a *signal*, an *action* and a *priority*. When the state machine's *current state* matches the transition's *current state* and the transition signal evaluates as true the transition can occur.

It is possible for the conditions of multiple transitions to be satisfied at the same time. In that case the state machine will transition to the next state defined by the transition with the highest priority. The SCT performs all actions from all satisfied transitions. In the software state machine only the actions associated with the highest priority transition will get called. If the transitions have equal priority the transition which will get fired is undefined, but one transition will occur.

Adding transitions

Transitions can be added in the **Diagram Editor** by selecting **Transition** from the pallet on the left. First, select the starting state for the transition, and then select the end state for the transition.

A transition can be added using the **State table** by entering a starting and finishing state name in the last row of the table. When entering state names into the **State Table**, you are able to select existing states from the drop-down list or type a name in. If the typed name does not match an existing state then a new state is created.

The transition is represented in the diagram as an arrow. It has an information box associated with it that indicates the signal and actions associated with it.

Deleting transitions

To delete a transition you can select either the transition's information box or part of the arrow representing the transition and choose **delete** from the edit menu or the right-click menu.

You may delete a transition in the **State Table** view by selecting it and pressing the delete button  in the **State Table**'s toolbar.

Adding signals to a transition

Once a signal [128] has been built it can be added to a transition in the **Diagram Editor** by right-clicking on the transition's information box and choosing the signal from the **Set signal** sub-menu. To change the signal, simply set it to a different signal.

In the **State Table** view you can change or set a signal by entering its name in the signal column for that transition. If the signal you enter does not exist, one with that name will be created.

Adding actions to a transition

The action associated with a transition can be set in the **State Table** in a similar way to setting the signal.

In the **Diagram Editor** you can add individual action elements directly to a transition. If the transition has no action already associated with it, a new action will be created to contain the action elements. If there is already an action associated with this state then the elements will be added to that action. Note that any other transitions using that action will also have the new action elements added to them.

Action elements can also be removed from an action by right clicking on the transition information box and choosing **delete action element**.

Appearance of transitions

Dragging the selection handles associated with a transition alters its visual appearance in the state machine **Diagram Editor**. Selecting either the arrow representing the transition or the label associated with it will display the selection handles and highlight the arrow. The arrow initially has five selection handles — three larger ones and two smaller ones — see Figure 9.29. Dragging the first handle onto another state will change the transition's **current state** — 1 in Figure 9.29. Similarly dragging the last handle onto another state will change the transition's **next state** — 4 in Figure 9.29. Dragging the middle handle will vary how much the arrow bends — 3 in Figure 9.29. Drag one of the two smaller handles to create a new bend in the line — 2 in Figure 9.29. You can create multiple bends in the arrow by dragging the smaller selection handles.

To remove a bend in the arrow, drag the larger handle between two other large handles and it will become a small handle again.

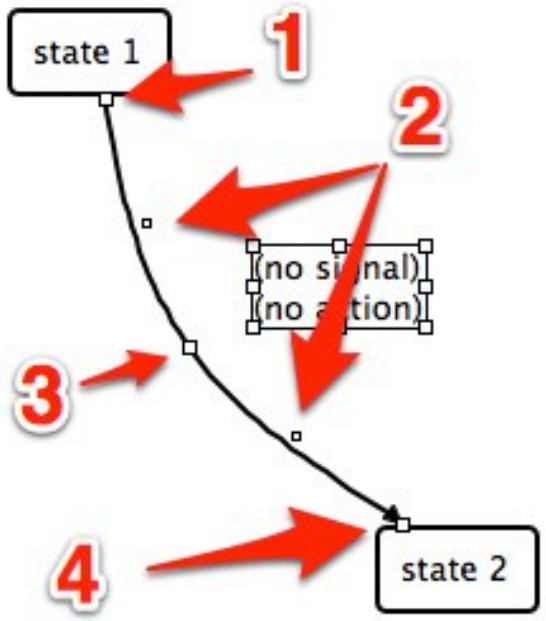


Figure 9.30. Selection handles on the transition arrow

9.5.4 Signals

A signal is a logical combination of inputs and outputs. Signals are constructed in the **Signal** view. To create a new signal, single-click the **Add signal** button in the **Signal** view. This will add a new signal name to the list in the top half. You can rename the signal by double-clicking on it or using the **Rename** button in the **Signal** view. No two signals should have the same name.

The logic of the signal is set in the bottom half of the **Signal** view. Initially a signal will have an entry saying `(none)`. Right-clicking on that element will allow you to replace it with an appropriate input or output. You can also combine elements using AND and OR.

Note that setting a signal will overwrite the selected element. If the signal name (in the top half) is the active selection then the root of your signal will get replaced. If you just want to set a sub-element (like one half of an AND condition for example) make sure that the element you want to replace is the selected element. Elements that have children (other subcomponents) will try to use the selected element as a child when it is added. For example, if an input variable is selected in the bottom part of the **Signal** view and you apply an AND to it, the input signal will be replaced by the AND, but one of the children of the AND condition will be the original input variable.

The SCT has several constraints restricting what can be used as a signal. A signal may only consist of one of the following:

- a single input or output
- a single timer match condition
- a logical combination of one match condition and one input or output

In the SCT you can specify the **I/O condition** for an input or output. Once you have added the input or output to the signal, right-click on it again and choose an option from the **I/O**

condition menu to set whether the signal will be fired when it is **high**, **low**, on the **rising edge**, or on the **falling edge**.

9.5.5 Actions

An action consists of a name and a set of action elements associated with it. Action elements perform tasks like setting an output variable's state or calling a function. A single action may be associated with one or more transitions.

A new action can be created by pressing the **Add action** button  in the **Action List**, typing a name into a transition's action column in the **State Table** or by adding action elements to a transition without an action in the **Diagram Editor**.

Once an action is created it can have elements added to it in the Action List by selecting it in the top half and then using the right-click menu in the bottom half of the view.

In the SCT the following action elements are available:

- **SET** - sets an output pin to be high
- **CLEAR** - sets an output pin to be low
- **TOGGLE** - calls a set and a clear on an output pin so pin will toggle if conflict resolution is appropriately set
- **CALL** - perform a special SCT action, like starting a timer or firing an interrupt
- **CAPTURE** - capture a timer's value into the specified capture register

In a software state machine you can add the following elements:

- **SET** - sets an output variable to a value you enter in the value column in the bottom of the Action List
- **CALL** - call a function you defined in the Output view

Action elements can be removed by selecting them and choosing remove action from the right-click menu – take care to ensure that the element is selected and not just the Action otherwise you may delete the entire action and all action elements associated with it.

Action elements may be added or removed from transitions in the **Diagram Editor** by selecting the transition information box and using the right-click menu.

9.5.6 Inputs

Inputs are read-only elements that are used in signals. In the SCT these inputs are mostly predefined and correspond to the pins used by the SCT. In a software state machine you are free to define inputs of any type.

The names of the inputs can be freely set with the usual requirement of uniqueness. Any items in the input view that are written in blue are not user editable items. They are not editable in order to enforce limitations of the state machine. In the SCT for example, you may change the name associated with input pin 3 but you can't change its source or type, nor can you delete it from the inputs.

SCT inputs

Inputs for the SCT have the following types:

- **bool** — a binary input from a pin. The pin numbers are defined in the source and are named CTIN_n, where n is the pin number.

- **const int** — a constant integer to be used in match conditions, its source is the integer value it is to be set to.
- **Match Low** (or **High**, or **Unified**) — a match condition which compares the timer specified in its type name with the const int input variable named in its source.

New **const ints** and **Match ...** conditions can be added in the **Input** view, but the input pins are fixed.

Software state machine inputs

In the software state machine the inputs can only be of type **variable**. The C type of the variable is defined in the source column. It should be noted that these variables are considered as constants that can't be directly set by the state machine.

9.5.7 Outputs

Outputs are elements of the state machine that may be used in actions.

SCT outputs

In the SCT they may have the following types:

- **bool** — a binary output pin. Its pin number is defined by its source name with `CTOUT_n` corresponding to pin number `n`. An initial value can be set for the output by choosing a value in the preload column.
- **function** — special SCT specific calls
- **Capture register** — a capture register to capture the timer indicated by its source

As with the inputs, most of the pre-populated outputs are not editable except for their names. Additional outputs can be added using the **Add output** button  in the **Output** view. These extra outputs are limited to be `Capture registers` or an `IRQ`.

- To add another interrupt request, add a new output and set its **type** to `function` and its **source** to `IRQ`.
- To add a capture register, add a new output and set its **type** to `Capture register` and choose which timer to capture.

When using IRQs with the SCT, care must be taken to correctly identify the event associated with an IRQ. When the state machine fires an IRQ it is handled by the SCT IRQ handler — e.g. `void SCT_IRQHandler (void)`. The handler must use the SCT event flag register `LPC_SCT->EVFLAG` to identify which event raised the interrupt. The i th bit of this register corresponds to the i th event and is `1` if that event has occurred since reset or since a `1` was last written to it. Setting a bit to `1` will clear it.

The code generated by Red State includes macro definitions linking IRQs with events. These take the form of your IRQ name prefixed with `SCT_IRQ_EVENT_`. To acknowledge the interrupt, the handler should clear the corresponding bit as follows:

```
LPC_SCT->EVFLAG = (1 << SCT_IRQ_EVENT_samplingComplete);
```

Where `SCT_IRQ_EVENT_samplingComplete` is a generated `#define` in the `sct_fsm.h` made by the parser file corresponding to an IRQ named `samplingComplete` in the state diagram.

Since there is not necessarily a one-to-one relationship between events and transitions (multiple transitions may be handled by a single event) it can be appropriate to use the

same IRQ on multiple transitions. However, if the parser assigns IRQs with the same name to different events, the macro may get defined twice to different values. For example you may find the following in `sct_fsm.h`:

```
#define SCT_IRQ_EVENT_samplingComplete  (5)
#define SCT_IRQ_EVENT_samplingComplete  (6)
```

In this case, add a new IRQ output in Red State and replace one of the existing IRQs in the state diagram with the new one.

Software state machine outputs

In a software state machine you can have the following types:

- **variable** — like the input generic variables their type is entered in the source column. A preload value can be assigned to initialize the variable.
- **function** — a function to be called. It is defined in the action header file and the body should be written in the main action C file. Currently passing of parameters to the function is not supported.

Preset values

Variables in both the SCT and software state machine can have a preset value. You can set a preset value by entering it into the **preset** column in the **Output** view. Variables are initialized to their preset values when the state machine is reset.

Note that preload values are not type checked by the code generator. The content of the preload cell will be directly entered into the generated code as the RHS of an assignment. For example, to initialize an output of type char with the character `a` you need to enter `'a'` in the preload cell including the single quotes, whereas to initialize an `int` you would simply directly enter a number e.g. `1337` without any quotes, etc.

9.6 Red State : Limitations

Red State uses NXP's SCT tool parser to generate the configuration code for the SCT. NXP's documentation notes the following limitations with the parser:

> The ultimate goal of the tool is to allow specifying every detail of a complex SCT design. For the time being, the following limitations are present:
>
> - The SCT global configuration (like the operating mode, the clocks configuration) has still to be specified manually within the application code. All specified match and acquired capture values are relative to those global clock settings
> - The conflict resolution register setup is not fully included in the state machine configuration. The default hardware setup after reset is to take no action in case of conflicts for events trying to drive the same output at the same time.
>   - In the current implementation, it is possible to choose the toggle action for those outputs which are set and cleared simultaneously at a specific point in time. This is done by specifying the pin=toggle attribute in the FZM drawing.
>   - In case of multiple events driving the same outputs, the programmer needs to manually modify the generated code to override the current

settings and program the conflict resolution register as desired, as this feature is not included in the current tool version (1.x)

- The conflict enable register is also not programmed and needs to be written by the programmer if it is required to trigger a no change conflict interrupt

- DMA0 and DMA1 support is potentially included in the tool and the parser, but it has not been extensively tested

—SCT-Tools FSM Designer for the State Configurable Timer, Rev. 2.0, 09 June 2011, NXP Semiconductors

9.7 Red State : Frequently Asked Questions

9.7.1 How do I migrate from a Red State project created in Red Suite / LPCXpresso v4 to one created in this version

The code generated by Red State in the current version is different from that generated within Red Suite / LPCXpresso v4. It was updated to allow multiple state machines to be created in the same project. Each state machine defines a prefix for its key data structures and functions to ensure that there are no conflicts between them.

The prefix is set in the **State Machine Setting** view — see number 7 in Figure 9.29.

This section details how to update an existing project to work with the new prefix naming convention. We assume that you are working from an existing project, with a software state machine generated by Red State under Red Suite 4. We will use the refactoring tools to update the old code. This refactoring should be done **before** regenerating the state machine code in the LPCXpresso IDE. This procedure is only required **once** and is designed to ensure that any code you have written that interacts with the state machine uses the new naming scheme.

Once these steps have been completed your existing project will be compatable with Red State in newer versions of LPCXpresso. Be sure to back up your code before applying these refactorings.

Note : Red State projects created using Red Suite / LPCXpresso after v4 do not need to be converted in this manner.

Step 1 — Choose a new prefix

Choose a short prefix to use for your state machine. This prefix will be applied to various variables and functions. Any two statemachines with the same prefix may conflict with each other. A prefix can be blank. If the prefix is not blank, an underscore character will be appended to it. For example if the prefix `mySM` was chosen, the `enum` containing the state ids would be named `mySM_state`. If a blank prefix was chosen the same generated `enum` would be named `state`.

Note

In this refactoring guide we will assume a prefix named `mySM` was used. You may choose to use a different prefix.

Step 2 — Refactoring the state data structures

Open the header file corresponding to the **main output file**. The main output file's name is listed in the **State Machine Setting** view. This header file contains declarations for the `enum rs_state`; two `structs` : `inpinputSignal` and `redStateOutput` as well as several function

declarations. In this step we need to refactor the name of the `enum rs_state` as well as the names representing the different states, `STA_STATE_1` for example.

By using the the built-in refactoring tools we can update all references to the elements we want to change in one go.

Warning

The refactoring tool may attempt to apply the refactoring to all global elements in your **workspace** that match the signature of the element being refactored. Use the refactoring preview option (**CTRL + ENTER**) to ensure that the refactoring is only applied to the correct projects.

- The `enum rs_state` needs to be renamed to `mySM_state`.

  1. Highlight the text `rs_state` in the main output header file.
  2. Choose the menu item **Refactor -> Rename**.
  3. Enter the new name `mySM_state`
  4. Press **CTRL + ENTER** to see the refactor preview and apply the change to the current project.

- For each element of the `enum mySM_state` rename their prefix from `STA_` to `S_` using the same refactoring tool. For example `STA_STATE_1` would become `S_STATE_1`.

  - Note that this step is not required if you do not use the state `enum` directly in the code you have written: that is, you do not use the `getState()` function. These names are updated any time the code is generated.

Step 3 — Refactor the input and output `structs`

The input and output `struct` in the main output header file generated in Red Suite 4 have prefixes such as `i_` on inputs and `v_` on outputs. These have been removed in subsequent versions of LPCXpresso IDE. Use the rename refactoring tool as before to remove these prefixes from your code; for example, `i_reset` would become `reset` and `v_timer` would become `timer`.

Step 4 — Refactor the input and output `typedef`

Use the refactoring tool as before to rename the following

- Rename the `typedef` for the input from `inputSignal` to `mySM_inputs`.
- Rename the `typedef` for the output from `redStateOutput` to `mySM_outputs`.

Step 5 — Refactor the preload function and the getter functions

Use the refactoring tool as before to rename the following functions

- Rename the preload function from `preload` to `mySM_preload`.
- Rename the function `getState` to `mySM_getState`.
- Rename the function `getInput` to `mySM_getInput`.
- Rename the function `getOutput` to `mySM_getOutput`.

Step 6 — Refactor the action function names

In Red Suite / LPCXpresso v4 the function names in the `..._action.h` and `..._action.c` files were prefixed with `act_`. In the current version of LPCXpresso IDE these functions use the same prefix used elsewhere. For example `act_GreenOn` would become `mySM_GreenOn`.

Use the refactoring tool to rename the `act_...` fuctions in the `..._actions.h` file, replacing the `act_` prefix with your new prefix, e.g. `smMS_`.

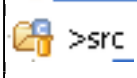
Step 7 — Generate code in the current version of LPCXpresso IDE

These refactoring should now have made your existing code compatable with the Red State code generated in this version of LPCXpresso IDE. The final step is to generate the state machine code. This code generation can be perfomred by right-clicking in the **State machine diagram editor** and choosing **Generate code**, or by pressing the **Generate code** button in the **State Table** view.

Review the generated files to ensure that the new names match those you manually refactored to.

10. Appendix A – File Icons

Table 10.1. File Icons

Icon	Meaning
	C language source file.
	C language header file.
	Source folder (generally use for source files, including headers)
	Folder (use for none-source files)
	Source folder with modified build properties
	Source file with modified build properties
	Project Makefile. Note that the LPCXpresso IDE may automatically generate these.
	C language source file that has been excluded from the project build. Hollowed out character in icon.
	General text file. Often used for files such as linker script files which end with the extension '.ld'.
	The blue double arrows at the top of the icon denote that this particular source file has different project build properties than the rest of the project.
	The orange/gold 'storage drive' denotes that this file is connected to a CVS repository.
	The right arrow denotes that the file or directory has been modified locally and may need committing to the CVS repository. I.e. The local copy is more up to date than the repository.
	
	This is an executable file.
	Directory containing executables.
	C Project Directory (Project Explorer View)
	C Project Directory linked to a CVS repository. (Project Explorer View)

11. Appendix B – Glossary of Terms

Table 11.1. Explanation of the meaning of terms used in this document.

Term	Meaning
Build artifact	The 'Build Artifact' is the final product of all the build steps (with the exception of any post dump files produced with 'objdump'). The Build Artifact is correctly set as the name of a new project by default, but may be changed manually. This can be useful, if for example you have copied an existing project (cloned it) and you want the new project executable to have the name of your new copy of the project.
Code Red Technologies	Company responsible for Red Suite and LPCXpresso IDEs. Acquired by NXP Semiconductors in April 2013.
Debug Target	The development board (evaluation board) or debug probe connected to a board.
DWARF	Debug with Attributed Record Format. Developed along with ELF but actually independent of the Object file format. DWARF is a format defined for carrying debug information in object files.
ELF	Executable and Linking Format. This is the object code file format used by our development toolchain and most microprocessor toolchains.
GCC	GNU Compiler Collection. C/C++ compiler and related tools used by LPCXpresso IDE.
LPC-Link	Low end debug probe for NXP microcontrollers, provided integrated into the NXP LPCXpresso development boards
LPCXpresso IDE	A software development environment for creating applications for NXP's range of ARM based 'LPC' range of Microcontrollers (MCUs). Originally a reduced functionality version of Red Suite for specific NXP microcontrollers.
Newlib	Open source C99 runtime library. Can optionally be used instead of Redlib for C projects, and required for C++ projects.
Project	A collection of source files and settings
Perspective	In the LPCXpresso IDE, a perspective is a particular collection of 'Views' that are grouped together to be suitable for a particular use. For example the 'Develop' perspective, the 'C/C++ programming' perspective and the 'Debug' perspective.
Red Probe™	Debug probe produced by Code Red Technologies that is compatible with both SWD/SWV and JTAG debug targets. For Cortex-Mx based devices, it will normally default to using SWD/SWV. For other ARM processors it will default to using JTAG.
Red Probe+™	Higher performance version of Red Probe.
Red Suite™	The Code Red Technologies IDE (Integrated Development Environment) based on Eclipse with extensions for embedded development. Supported MCUs from a number of vendors. No longer in development.
Red Suite NXP Edition	Version of Red Suite for NXP microcontrollers only, with variants with 256KB and 512KB code download limits. Acted as a low cost upgrade path for the free LPCXpresso IDE. No longer in development
Redlib™	Proprietary non-GNU C90 runtime library (with some C99 extensions). Generally provides reduced code size compared to Newlib.
Semihosting	The ability to use IO on your debugger host system for your target embedded system. For example a 'printf' will appear in the console window of the debugger.
SWD	Serial Wire Debugging (Single Wire Debugging).This is a debug connection technology available on Cortex-M based parts that allows debug through just 2-wires unlike 6 for JTAG.
SWV (Red Trace™)	Serial Wire Viewing. This is an additional feature to SWD that allows the LPCXpresso IDE to give real-time tracing visualization and views from Cortex-M3/M4 based devices. SWV is only available if SWD is being used.
View	A 'View' is a window in the LPCXpresso IDE that shows a particular file or activity. A 'Perspective' is the layout of many 'Views'.
Workspace	The LPCXpresso IDE organizes groups of projects into a 'Workspace'. A workspace is stored as a directory on your host PC and has subdirectories containing individual projects.