Application Note: JN-AN-1193

JN516x Lauterbach JTAG Debugger

This Application Note details how to configure a debug environment for code running on a NXP JN516x wireless microcontroller using a Lauterbach JTAG Debugger.

An example application is provided along with instructions on how to configure boards from the JN516x-EK001 Evaluation Kit to allow the code running on them to be debugged using the Lauterbach JTAG Debugger. A description of the operation of the debugger IDE (Integrated Development Environment) is beyond the scope of this Application Note - full documentation is available from Lauterbach.

1 Introduction

This Application Note describes how to use the Lauterbach JTAG Debugger to debug code running on a JN516x wireless microcontroller fitted to a board from the JN516x-EK001 Evaluation Kit. This document refers to an example application for which source code is provided in the ZIP package of this Application Note. There is also a troubleshooting checklist in Appendix B which provides assistance in getting your own application to run with the debugger.

The following components are required:

- NXP DR1174 Carrier Board fitted with a DR1199 Generic Expansion Board (both available within the JN516x-EK001 Evaluation Kit)
- Lauterbach PowerDebug Module - any one of the following:
    - LA-3500 PowerDebug USB3.0
    - LA-7705 PowerDebug Ethernet
    - LA-7699 PowerDebug II
    - LA-7708 PowerDebug USB2.0
- Lauterbach JTAG Debug Cable - LA-3793 Beyond Debugger
- Lauterbach Beyond-to-NXP JTAG adapter
- Lauterbach IDE - TRACE32 PowerView

The evaluation kit and its components are described in the *JN516x-EK001 Evaluation Kit User Guide (JN-UG-3093)*.

The example software was developed using the JN516x Integrated Peripherals API. This API is described in the *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*.

Documentation for the Lauterbach hardware and software components can be found on the Lauterbach website: http://www.lauterbach.com.

2 Compatibility

The software and set-up instructions provided with this Application Note are intended for use with the following evaluation kit and SDK (Software Developer's Kit) versions:

Product Type	Part Number	Version or Build
Evaluation Kit	JN516x-EK001	-
BeyondStudio for NXP	JN-SW-4141	V1111
SDK Libraries	JN-SW-4163	V1052

3 Getting Started

3.1 System Overview

This Application Note uses the Lauterbach In-Circuit Debug (ICD) solution, which is a standalone system that can be used alongside the NXP development tools in order to debug software running on a JN516xx wireless microcontroller located on an NXP carrier board. This system is illustrated in the figure below.

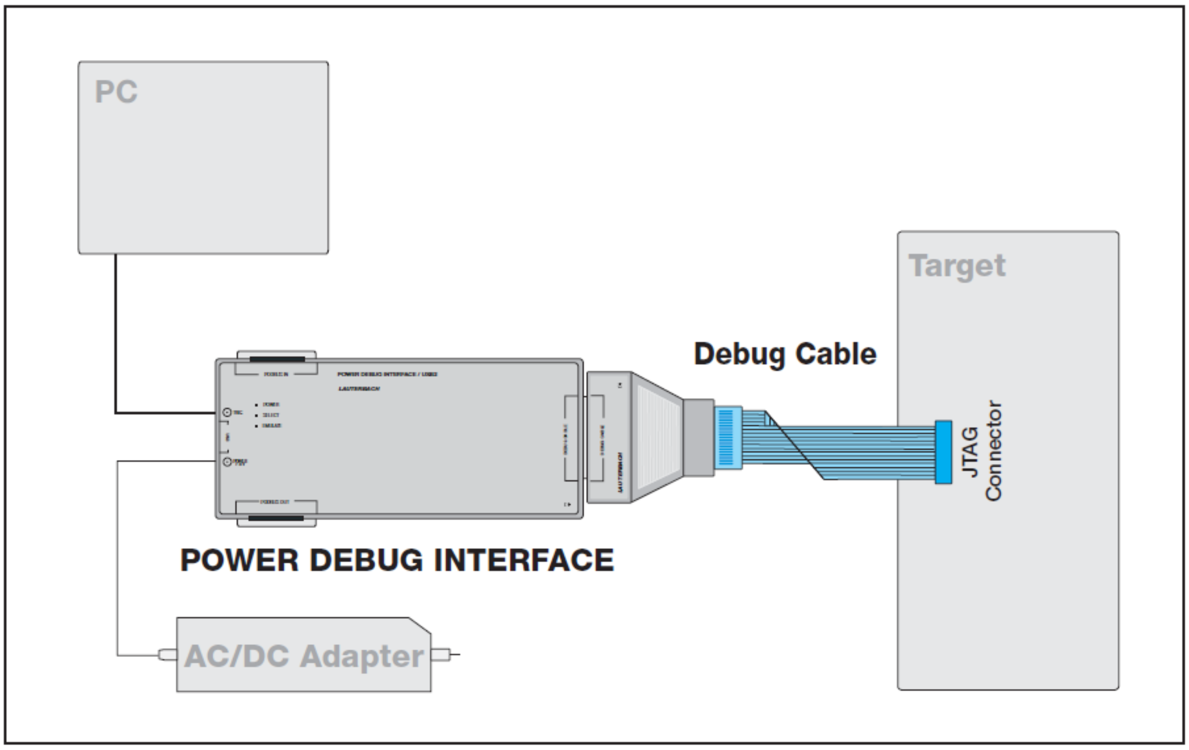


Figure 1: System Architecture

3.2 Debugging the Example Application

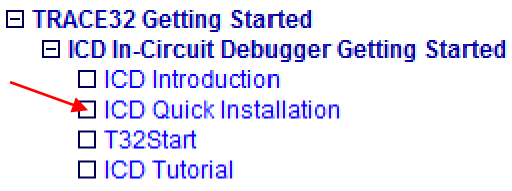
The following sub-sections detail the steps required to debug the example application provided with this Application Note using the Lauterbach JTAG Debugger.

3.2.1 Install the Lauterbach software

You will need the TRACE32 PowerView DVD. Follow the instructions from http://www.lauterbach.com.

Expand the **[+] Support** menu on the lefthand-side of the home page, expand **[+] Download Centre** and then select **TRACE32 Manuals**.

Finally, select **ICD Quick Installation**, as shown below:


⊟ TRACE32 Getting Started
⊟ ICD In-Circuit Debugger Getting Started
☐ ICD Introduction
☐ ICD Quick Installation
☐ T32Start
☐ ICD Tutorial

Note: During the installation process you will be asked to select the debug method. You should choose **In Circuit Debug (ICD)**. When asked for the CPU, you should choose the **BEYOND** option for correct customisation of the IDE for the JN516x.

3.2.2 Build the example application using the NXP SDK

Before you start to build the application, please ensure that you have following installed on your development PC:

- BeyondStudio for NXP (JN-SW-4141)
- JN516x IEEE802.15.4 SDK (JN-SW-4163)

Note: For the installation instructions, please refer to *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*

The above resources are available from the NXP Wireless Connectivity TechZone.

In order to build the supplied software, the Application Note folder (**JN-AN-1193-JN516x-Lauterbach-JTAG-Debugger**) must be placed under **<BeyondStudio for NXP Installation root>\workspace\**, where **<BeyondStudio for NXP Installation root>** is the path into which BeyondStudio for NXP was installed (by default, this is **C:\NXP\bstudio_nxp**). The **workspace** directory is automatically created when you start BeyondStudio for NXP.

To build the application, follow the instructions below:

1. Ensure that the project directory is located in

   <BeyondStudio for NXP Installation root>\workspace

2. Start the Eclipse platform and import the relevant as follows:
   - **a)** In BeyondStudio, follow the menu path **File>Import** to display the **Import** dialogue box.
   - **b)** In the dialogue box, expand **General**, select **Existing Projects into Workspace** and click **Next**.
   - **c)** Enable **Select root directory** and browse to the **workspace** directory.
   - **d)** In the **Projects** box, select the project to be imported and click **Finish**.
3. Build the application. To do this, ensure that the project is highlighted in the left panel of BeyondStudio and use the drop-down list associated with the hammer icon 🔨 in the toolbar to select the build configuration – once selected, the application will automatically build. The binary file will be created in the **Build** directory.

3.2.3 Load the binary file into the JN516x using the Flash programmer

The first time a device is programmed with a debug-enabled binary file, it must be done serially. You can do this using the integrated JN51xx Flash Programmer (described in the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*). Subsequent loads of debug-enabled binary files can then be performed using the Lauterbach IDE.

3.2.4 Enable JTAG debugging on the DR1174 Carrier Board

Remove the DR1199 Expansion Board from the DR1174 Carrier Board and ensure the JTAG jumper is in the enabled (**EN**) position, as shown in the photograph below, and then re-connect DR1199 to DR1174.

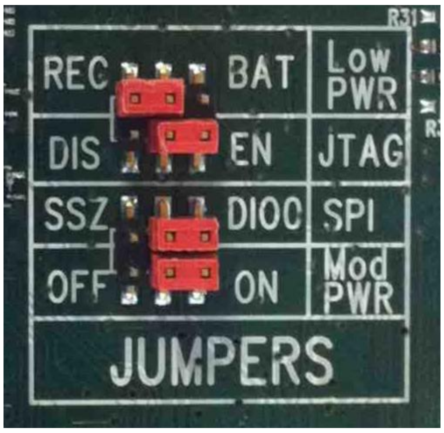


Figure 2: DR1174 JTAG Jumper Position

Note: Once the jumper is in the JTAG position, UART 0 will be disabled. You must therefore load the debug-enabled binary file before changing the jumper. You must change the jumper position to disable JTAG before using the Flash programmer.

3.2.5 Connect the Lauterbach Debug Module to the DR1174 board

Using the adapter board, plug the Debug cable onto the JTAG connector (**J6**) on the DR1174 Carrier Board. Ensure that pin 1 of the adapter board cable aligns with pin 1 of the DR1174 board.

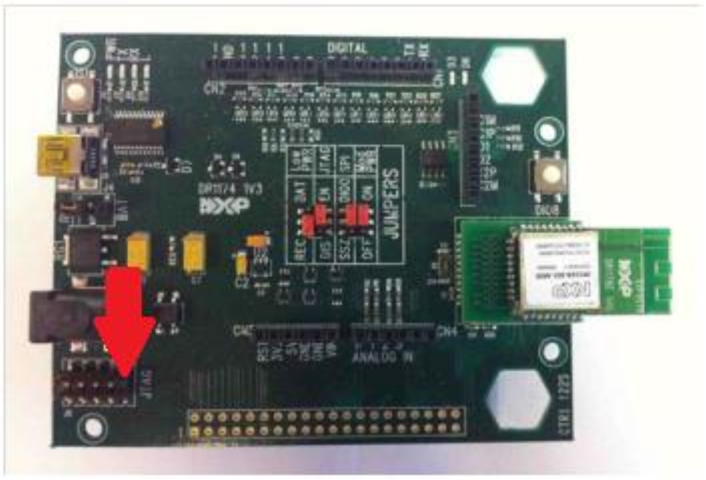


Figure 3: DR1174 JTAG Connector

3.2.6 Mount the DR1199 Generic Expansion Board onto the carrier board

The DR1199 Generic Expansion Board contains the LED which is controlled in the example application.

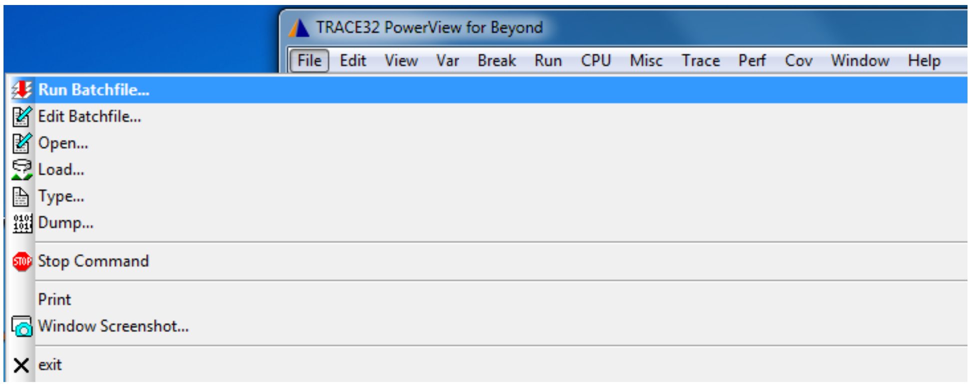
3.2.7 Apply power to the DR1174 Carrier Board

The carrier board can be powered by the USB connector, even though UART0 has been disabled.

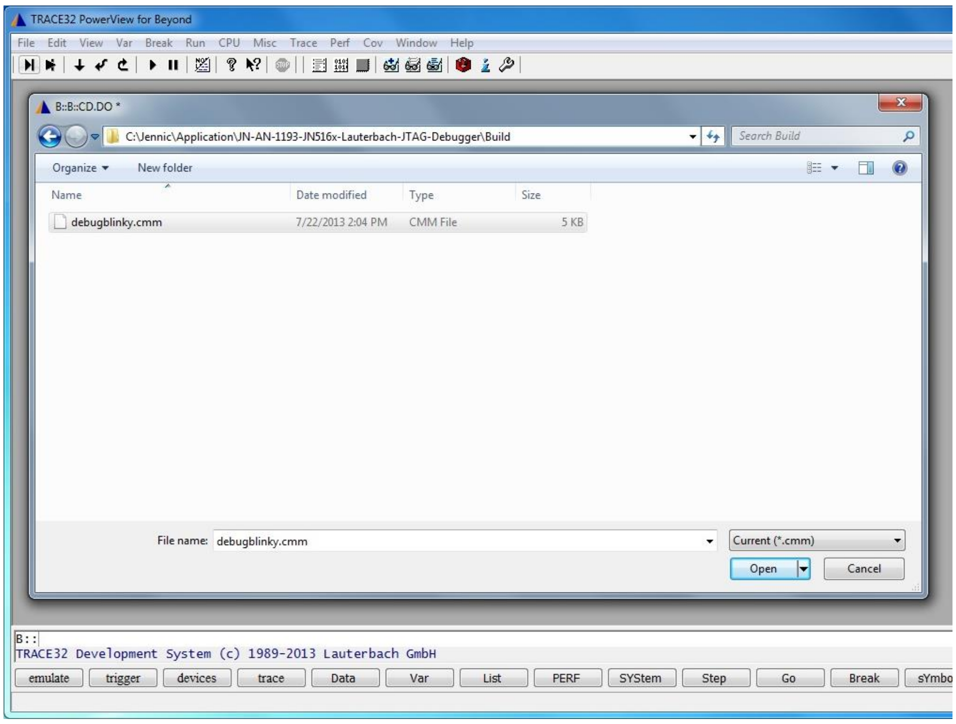
3.2.8 Start the Lauterbach TRACE32 IDE and run the start-up script

Start the Lauterbach TRACE32 PowerView application and then run the start-up script found in the Build folder as follows:

1. From toolbar menu, select **File** and then **Run Batchfile**.



2. Select the **debugblinky.cmm** file contained within the **Build** folder with path **<BeyondStudio for NXP Installation root>\workspace\ JN-AN-1193-JN516x-Lauterbach-JTAG-Debugger\Build** and then click the **Open** button.



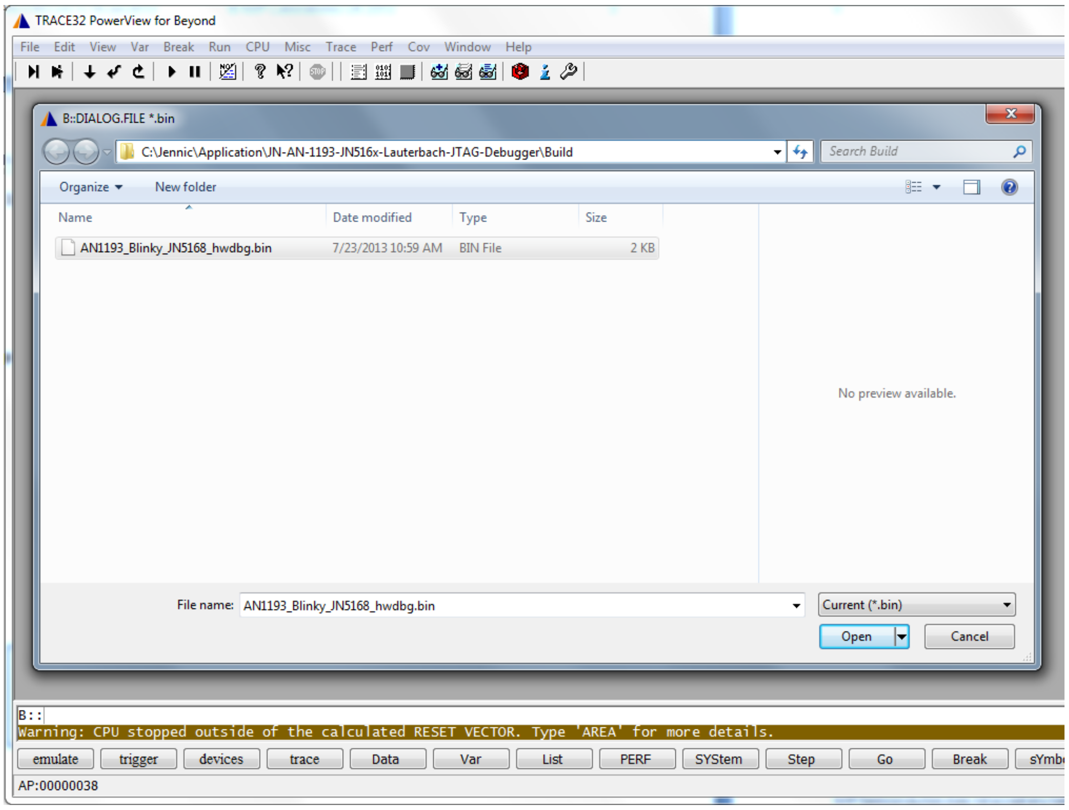
C:\Jennic\Application\JN-AN-1193-JN516x-Lauterbach-JTAG-Debugger\Build

Name	Date modified	Type	Size
debugblinky.cmm	7/22/2013 2:04 PM	CMM File	5 KB

File name: debugblinky.cmm
Current (*.cmm)

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

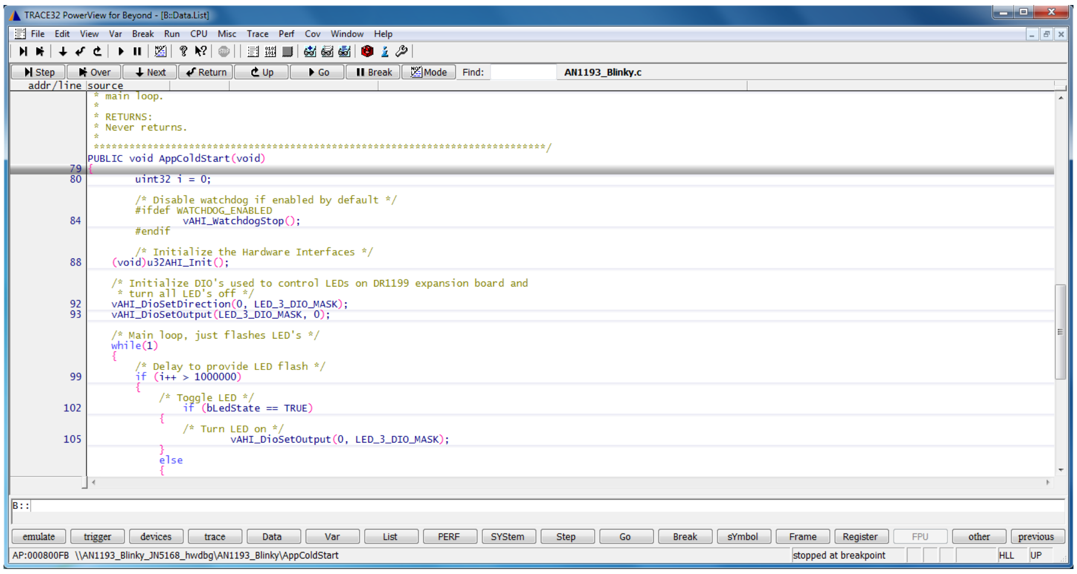
3. Another dialog box will now open allowing a binary file for download to be selected. Select the **AN1193_Blinky_JN5168_hwdbg.bin** file contained within the **Build** folder with path **<BeyondStudio for NXP Installation root>\workspace\JN-AN-1193-JN516x-Lauterbach-JTAG-Debugger\Build** and then click the **Open** button.



```
C:\Jennic\Application\JN-AN-1193-JN516x-Lauterbach-JTAG-Debugger\Build

Name                            Date modified        Type       Size
AN1193_Blinky_JN5168_hwdbg.bin  7/23/2013 10:59 AM   BIN File   2 KB

File name: AN1193_Blinky_JN5168_hwdbg.bin        Current (*.bin)

B::
Warning: CPU stopped outside of the calculated RESET VECTOR. Type 'AREA' for more details.
AP:00000038
```

TRACE32 PowerView for Beyond
File Edit View Var Break Run CPU Misc Trace Perf Cov Window Help
B::DIALOG.FILE *.bin
Search Build
Organize
New folder
No preview available.
Open
Cancel
emulate trigger devices trace Data Var List PERF SYStem Step Go Break sYmbol

4. The selected binary file will be downloaded to the JN5168 device and the bootloader will automatically start. The debugger stops execution at the **AppColdStart()** entry point. The TRACE32 IDE will display the example application source code.



```
 * main loop.
 *
 * RETURNS:
 * Never returns.
 *
 ****************************************************************************/
PUBLIC void AppColdStart(void)
{
        uint32 i = 0;

        /* Disable watchdog if enabled by default */
        #ifdef WATCHDOG_ENABLED
                vAHI_WatchdogStop();
        #endif

        /* Initialize the Hardware Interfaces */
    (void)u32AHI_Init();

    /* Initialize DIO's used to control LEDs on DR1199 expansion board and
     * turn all LED's off */
    vAHI_DioSetDirection(0, LED_3_DIO_MASK);
    vAHI_DioSetOutput(LED_3_DIO_MASK, 0);

    /* Main loop, just flashes LED's */
    while(1)
    {
        /* Delay to provide LED flash */
        if (i++ > 1000000)
        {
            /* Toggle LED */
                if (bLedState == TRUE)
            {
                /* Turn LED on */
                        vAHI_DioSetOutput(0, LED_3_DIO_MASK);
            }
            else
            {
```

```
B::
AP:000800FB  \\AN1193_Blinky_JN5168_hwdbg\AN1193_Blinky\AppColdStart    stopped at breakpoint    HLL  UP
```

TRACE32 PowerView for Beyond - [B::Data.List]
File Edit View Var Break Run CPU Misc Trace Perf Cov Window Help
Step Over Next Return Up Go Break Mode Find: AN1193_Blinky.c
addr/line source
emulate trigger devices trace Data Var List PERF SYStem Step Go Break sYmbol Frame Register FPU other previous

3.2.9 Debug the example application

The example application can now be debugged by setting breakpoints, single-stepping and watching the values of variables, etc. It is recommended that you follow the Lauterbach training, which can be accessed via:

- **Help->Training Manuals->Debugger**
- **Help->Training Manuals->Hll Debugging**

Instructions concerning the operation of the Lauterbach TRACE32 IDE can also be found at: http://www.lauterbach.com/manual.html.

Note that if you rebuild your application binary file, it can be downloaded via JTAG.

4 Debugging a Sleeping Application

The JTAG debug registers are not preserved over sleep. Additional steps are therefore required to enable debugging to continue after sleep.

4.1 Enable Polling

The debugger can continuously poll the JTAG interface when the JN516x device is asleep. When the device wakes and enables the debug registers, the debugger can then connect. The **debugblinky.cmm** file already has polling enabled via the line:

```
SYS.Option LPMDebug ON
```

4.2 Preserving the Contents of Registers through Sleep

The file **AN1193_Sleep.c**, located at **<BeyondStudio for NXP Installation root>\workspace \JN-AN-1193-JN516x-Lauterbach-JTAG-Debugger\Source\AN1193_Sleep.c**, contains an example application that sleeps. The file scope variable below is used to store the register contents during sleep:

```
uint32 u32DebugRegisters[AHI_STORE_DEBUG_REGISTER_COUNT];
```

The register contents are saved just before sleep by the line:

```
vAHI_StoreDebug(u32DebugRegisters);
```

The application then sleeps with RAM held so that the contents of the above variable are preserved. Further information on sleep modes and the wake timers can be found in the *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*.

When the application wakes, the bootloader calls the **AppWarmStart()** function. The registers are restored during the warm start by:

```
vAHI_RestoreDebug(u32DebugRegisters);
```

Note: The JenOS PWRM library contains calls to store and restore the debug registers. ZigBee PRO applications therefore do not require these calls. JenNet-IP applications do not use the PWRM library and so do require the debug registers to be preserved.

4.3 Building the Example Application

The following sub-sections detail the necessary steps to build the example sleep application.

Using the drop down list associated with the hammer icon in Eclipse, select AN1193_Sleep to build the example sleeping application.

4.4 Downloading and Running the Example Application

Run the start-up script **debugblinky.cmm** as described above. This time, select the **AN1193_Sleep_hwdbg.bin** file.

Note that running the script downloads the new application via JTAG without requiring the Flash programmer.

Start the application by pressing **Go**.

The LED should flash. When the LED is off, the device is awake and executing a busy wait loop. When the LED is on, the device is asleep.

4.4.1 Debugging a Sleeping Application

At the bottom of the Lauterbach application, a green bar will change from “Running” to “Running (not responding)” as the device sleeps.

Pressing **Break** while in the “Running” state will allow the debugger to gain control. Breakpoints can be set. If a breakpoint is set, it will be hit even if it is not executed until after the device has slept and woken.

Pressing **Break** while in the “Running (not responding)” state will result in an error displayed in red: “Access timeout, processor running”.

Attempting to run the script **debugblinky.cmm** to load another program or to restart the application will result in an error if the device is asleep when the script is run. To avoid this, manually reset the target by pressing the “Reset” button before running the script.

Appendix C describes how to modify the hardware so that the debugger can drive the reset line automatically.

5 Using Debug (DBG) Module with Lauterbach

5.1 Using UART1

When the JTAG debugger is connected, it is no longer possible to use UART0 for displaying debug information. Debug information can either use UART1 or a terminal within the Lauterbach IDE. The configuration for UART1 with the evaluation kit depends on the expansion board.

The Generic Expansion Board (DR1199) has a USB connector to access UART1. This can be accessed by initialising the DBG function as shown below:

```
DBG_vUartInit(DBG_E_UART_1, DBG_E_UART_BAUD_RATE_115200);
```

When using the LCD Expansion Board (DR1215), UART1 must be configured to single-wire (transmit only) mode:

```
vAHI_UartTxOnly(E_AHI_UART_1, TRUE);
DBG_vUartInit(DBG_E_UART_1, DBG_E_UART_BAUD_RATE_115200);
```

The yellow wire (RXD) of an FDTI serial cable may then be connected to the IO pin labelled Digital 4 towards the top-right of the expansion board.

When using the Lighting/Sensor Expansion Board (DR1175), the UART must be configured to use the alternative output pins:

```
vAHI_UartSetLocation(E_AHI_UART_1, TRUE);
vAHI_UartTxOnly(E_AHI_UART_1, TRUE);
DBG_vUartInit(DBG_E_UART_1, DBG_E_UART_BAUD_RATE_115200);
```

The yellow wire (RXD) of an FDTI serial cable may then be connected to the IO pin labelled 14 on the top-left of the expansion board.

Note that an FDTI cable is not supplied as part of the evaluation kit.

5.2 Using the Lauterbach Terminal

The Lauterbach terminal may be used as an alternative to UART1.

The sample application, **AN1193_Terminal.c**, demonstrates how to use **DBG_vPrintf()** debugging to send output to a terminal within the Lauterbach IDE.

The Debug (DBG) module is described in the *JenOS User Guide (JN-UG-3075)*.

To initialise the Lauterbach terminal, instead of initialising the UART via

```
DBG_vUartInit(DBG_E_UART_1, DBG_E_UART_BAUD_RATE_115200);
```

Initialise the Lauterbach DBG module instead via the function below:

```
DBG_vLauterbachInit();
```

The DBG library may be configured to flush data to the terminal is several ways, via the **DBG_u32Flags** variable.

If the **DBG_FLAG_AUTO_FLUSH** flag is set in **DBG_u32Flags**, the **DBG_vPrintf()** function calls **DBG_vFlush()** at the end of each line.

If the **DBG_FLAG_FLUSH_WHEN_FULL** flag is set in **DBG_u32Flags**, the **DBG_vPrintf()** function calls **DBG_vFlush()** when the output buffer (256 bytes) is full.

The flush function calls a trap instruction to trigger a software breakpoint. The Lauterbach debug probe intercepts this trap instruction and transfers data between the terminal and the DBG module.

Note: The hardware debugger will stall the CPU for approximately 2 ms while the software breakpoint is being processed. This may occasionally cause protocol stacks to behave differently and will affect the real-time behavior of user applications. For example, the Touchlink message exchange in the Zigbee Light Link Application Note is likely to fail.

The IDE indicates this intrusive action in the status line. Next to the running state, the red "I" informs about **I**ntrusive actions of the debugger. This is similar to "S" which means **S**low as the IDE interacts with the target (e.g. software emulated breakpoints).

5.3 Script Settings for Lauterbach Terminal

The terminal method **BufferQUICK** configures the debugger for use with the terminal. The general syntax is:

```
TERM.METHOD BufferQUICK <AddressOfTrap> <BufferTargetToGui>
<BufferGuiToTarget>
```

The line below should be used to configure the terminal with the required symbols:

```
TERM.METHOD BQUICK ADDRESS.OFFSET(sYmbol.BEGIN(DBG_vLbSpot))
ADDRESS.OFFSET(Var.ADDRESS(DBG_au8LbPutBuffer))
ADDRESS.OFFSET(Var.ADDRESS(DBG_au8LbGetBuffer))
```

The BQUICK terminal mode is a new feature that has been added for the JN516x device. It is only available in the Lauterbach IDE from the 2013.08 version. Use **Help->About** in the Lauterbach IDE to verify the installed version and contact Lauterbach technical support for an updated version, if required.

The example script **debugterminal.cmm** contains additional terminal related commands. The script has a TERM.RESet near the start that deletes any existing terminal windows/data. This is required to allow the Flash memory to be reprogrammed. After the TERM.METHOD, the following lines configure the terminal window with a scroll-back, and set its position and name in the IDE.

```
TERM.Scroll On
term.size 50 3000
WinPOS ,,,,,,myterm
TERM
```

5.4 Building and Running the Example Application

The example application can be built by selecting the target from the drop down list associated with the hammer icon in Eclipse.

The application should be run with a different script: **debugterminal.cmm**, as this script contains the terminal commands above. The script has a hard-coded filename rather than prompting to select the binary file.

Upon running the script, there should be a terminal window and a **Data.List** window. You may need to resize the windows to see them both. You can use WinPOS commands in your script to automatically reposition the windows.

Pressing **GO** in the **Data.List** window will start the application. The serial output should appear in the **TERM** window.

After looping 10 times, an assert is triggered. Selecting **View->Stack frame with locals** brings up a stack trace which can be used to locate the function that triggered the assertion.

6 Enabling Debug in your own Application

Upon start-up (or reset), the JN516x bootloader examines the header of the application image stored in Flash memory to determine whether JTAG-based debugging has been enabled and, if so, which set of DIO pins are to be used as the JTAG interface.

To set the appropriate bits in the image header, re-link the application with the following build options added to the Makefile or passed via the command line:

```
DEBUG=HW
DEBUG_PORT=UART0
```

The `DEBUG=HW` option sets the debug enable bit in the application header. By default all compiler optimisations and link time optimisation are turned off for debug builds. To reenable compile optimisations you may use the option below.

```
DEBUG=HW_NOOPT
```

Note that this can sometimes compromise single-stepping and variable examination from the C code.

Link time optimisation is turned off for debug builds because the aggressive optimisations it performs compromise debugging. This will also make the output binary larger. If the binary will no longer fit into the device, you may have to re-enable link time optimisation. To do this, add the following flags to the CFLAGS and LDFLAGS in the application Makefile:

```
CFLAGS += -flto
LDFLAGS += -flto
```

The `DEBUG_PORT=UART0` setting instructs the bootloader to use the UART0 pins (DIO4, DIO5, DIO6 and DIO7) as the JTAG interface. This option should be selected if the debugger is to be connected to J6 on the DR1174 board.

Note: When JTAG-based debugging is enabled using the `DEBUG=HW` or `DEBUG=HW_SIZEOPT` build options, the application will no longer boot unless a debugger is connected.

6.1 .cmm Script

Example scripts for the JN516x hardware can be found in the installation at:

C:\T32\demo\beyond\hardware\jn5168\big_endian\

When debugging your own application, it is recommended to base the **.cmm** script on **flashdemo.cmm** in this directory, rather than **blinky.cmm**.

6.2 Bootloader Symbols

The bootloader contains functions such as **memcpy()** and some Flash access functions that are used by the JN516x Integrated Peripherals API. To allow the stack unwinding to work correctly through these functions, it is necessary to load the bootloader symbols. For example, if there is an exception caused by an incorrect parameter to **memcpy()**, it is useful to be able to see the function which called **memcpy()**. **Debugblinky.cmm** contains the line below to load these symbols:

```
Data.Load.ELF "BootLoader_JN5168.elf" /NoClear /NoCODE /NoRegister
```

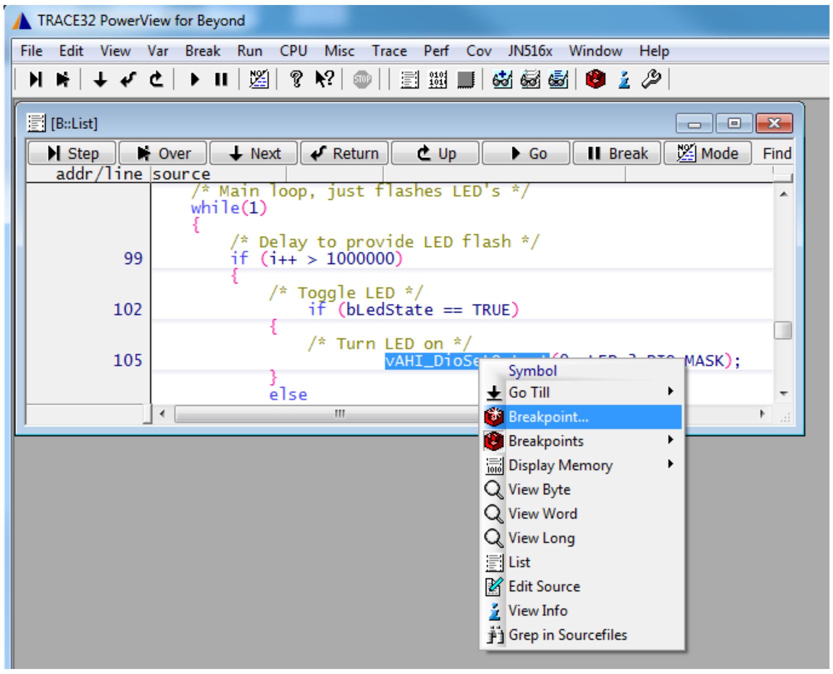
The bootloader symbols are provided in the **Build** folder of the Application Note.

Appendix A: Checking the Debugger with Demo Code

The following procedure checks basic debugger operation with the source code supplied as part of this Application Note. It is assumed that the user has completed all the steps in Section 3.2 “Debugging the Example Application”.

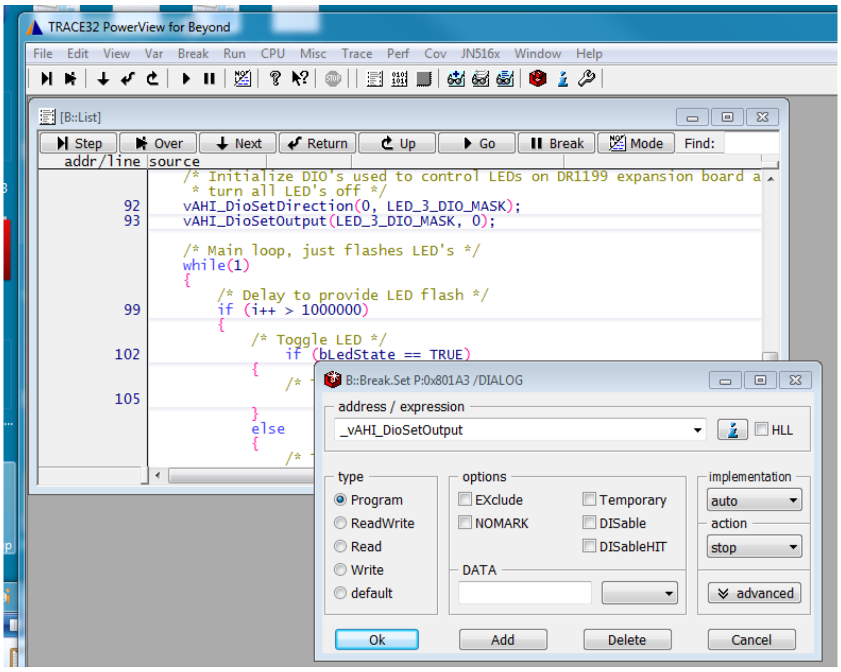
Trace32 PowerView should be displaying the B::List view (Source View).

1. Move the cursor to line 105 and click on **vAHI_DioSetOutput**. Right-click to add a breakpoint, as shown below.


TRACE32 PowerView for Beyond
File Edit View Var Break Run CPU Misc Trace Perf Cov JN516x Window Help
[B::List]
Step Over Next Return Up Go Break Mode Find
Symbol
Go Till
Breakpoint...
Breakpoints
Display Memory
View Byte
View Word
View Long
List
Edit Source
View Info
Grep in Sourcefiles

```
addr/line source
          /* Main loop, just flashes LED's */
          while(1)
          {
              /* Delay to provide LED flash */
99            if (i++ > 1000000)
              {
                  /* Toggle LED */
102                   if (bLedState == TRUE)
                  {
                      /* Turn LED on */
105                           vAHI_DioSe                    MASK);
                  }
                  else
```

2. This displays the ‘set breakpoint’ dialogue box, as shown below. Click **OK**.


TRACE32 PowerView for Beyond
File Edit View Var Break Run CPU Misc Trace Perf Cov JN516x Window Help
[B::List]
Step Over Next Return Up Go Break Mode Find:
Ok
Add
Delete
Cancel
advanced

```
addr/line source
          /* Initialize DIO's used to control LEDs on DR1199 expansion board a
           * turn all LED's off */
92        vAHI_DioSetDirection(0, LED_3_DIO_MASK);
93        vAHI_DioSetOutput(LED_3_DIO_MASK, 0);

          /* Main loop, just flashes LED's */
          while(1)
          {
              /* Delay to provide LED flash */
99            if (i++ > 1000000)
              {
                  /* Toggle LED */
102                   if (bLedState == TRUE)
                  {
                      /*
105
                  }
                  else
                  {
                      /*
```

B::Break.Set P:0x801A3 /DIALOG

address / expression: _vAHI_DioSetOutput ☐ HLL

type: [✓ Program] ReadWrite Read Write default

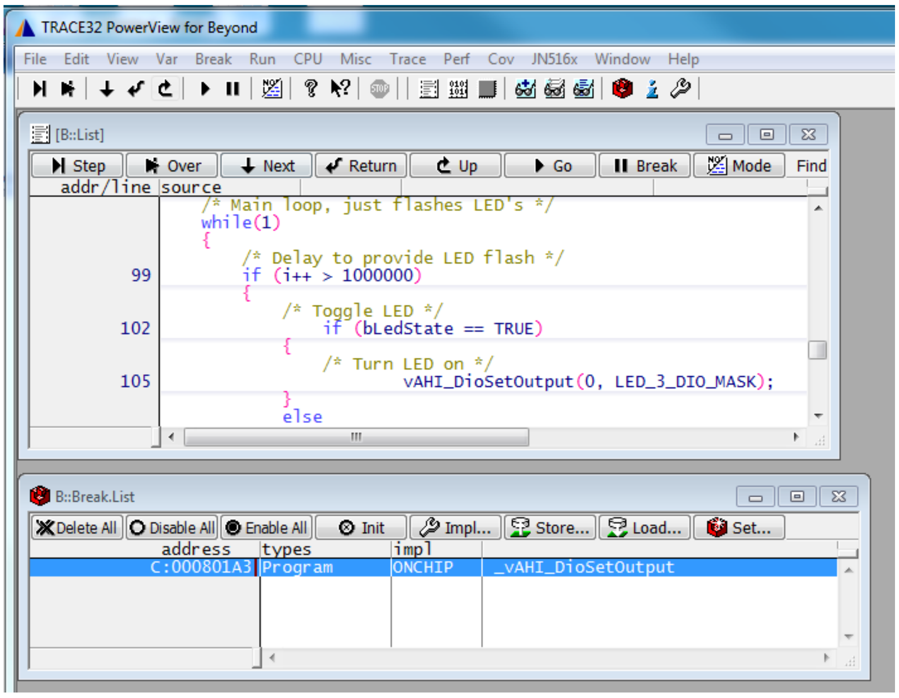
options: ☐ EXclude ☐ NOMARK ☐ Temporary ☐ DISable ☐ DISableHIT

DATA:

implementation: auto

action: stop

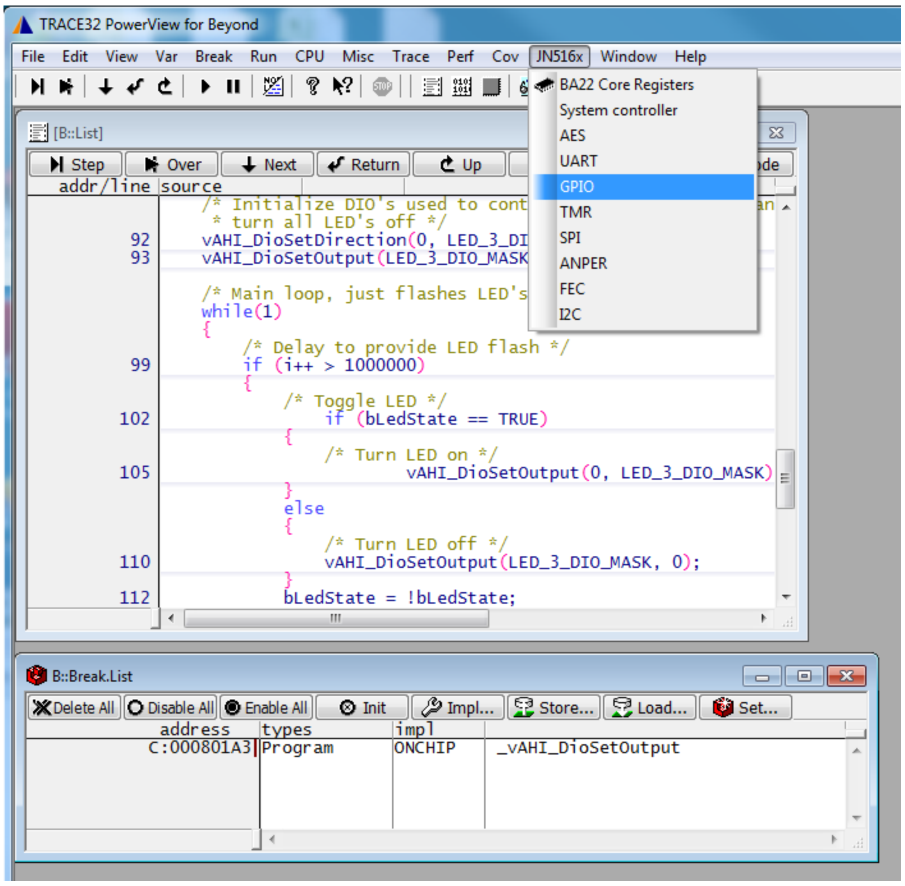
3. List the breakpoints by clicking the ‘red box’ icon on the top menu.


TRACE32 PowerView for Beyond
File Edit View Var Break Run CPU Misc Trace Perf Cov JN516x Window Help
[B::List]
Step Over Next Return Up Go Break Mode Find
addr/line source
B::Break.List
Delete All Disable All Enable All Init Impl... Store... Load... Set...

```
/* Main loop, just flashes LED's */
while(1)
{
    /* Delay to provide LED flash */
    if (i++ > 1000000)
    {
        /* Toggle LED */
            if (bLedState == TRUE)
        {
            /* Turn LED on */
                    vAHI_DioSetOutput(0, LED_3_DIO_MASK);
        }
        else
```

address	types	impl	
C:000801A3	Program	ONCHIP	_vAHI_DioSetOutput

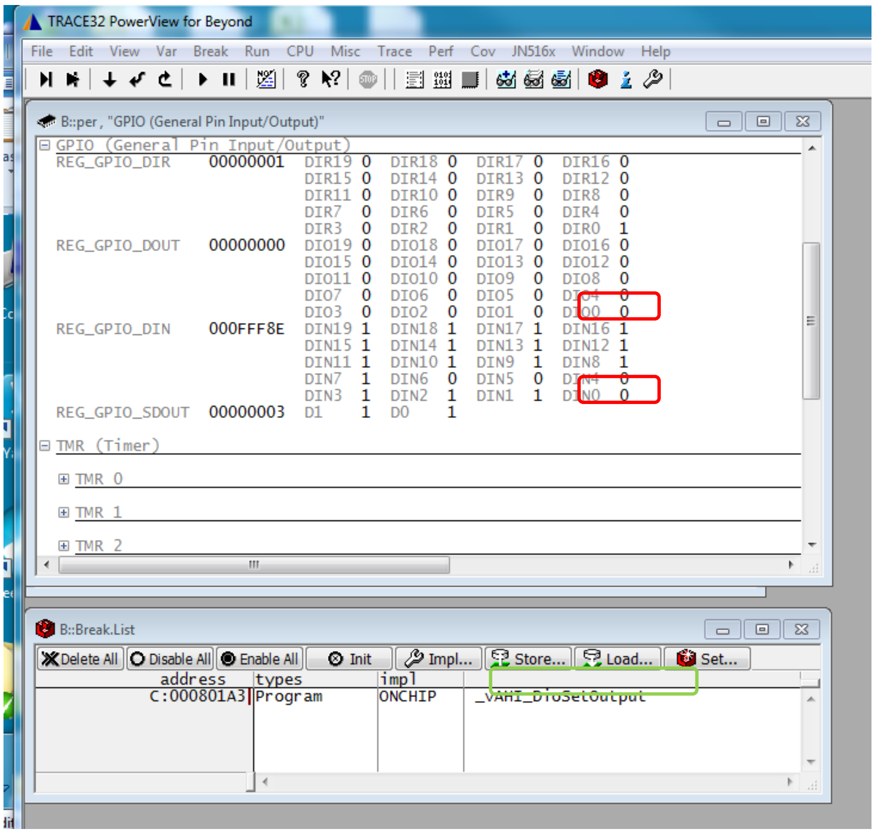
4. Next, select the GPIO view.


TRACE32 PowerView for Beyond
File Edit View Var Break Run CPU Misc Trace Perf Cov JN516x Window Help
BA22 Core Registers
System controller
AES
UART
GPIO
TMR
SPI
ANPER
FEC
I2C
[B::List]
Step Over Next Return Up
addr/line source
B::Break.List
Delete All Disable All Enable All Init Impl... Store... Load... Set...

```
/* Initialize DIO's used to cont
 * turn all LED's off */
vAHI_DioSetDirection(0, LED_3_DI
vAHI_DioSetOutput(LED_3_DIO_MASK

/* Main loop, just flashes LED's
while(1)
{
    /* Delay to provide LED flash */
    if (i++ > 1000000)
    {
        /* Toggle LED */
            if (bLedState == TRUE)
        {
            /* Turn LED on */
                    vAHI_DioSetOutput(0, LED_3_DIO_MASK)
        }
        else
        {
            /* Turn LED off */
            vAHI_DioSetOutput(LED_3_DIO_MASK, 0);
        }
        bLedState = !bLedState;
```

address	types	impl	
C:000801A3	Program	ONCHIP	_vAHI_DioSetOutput

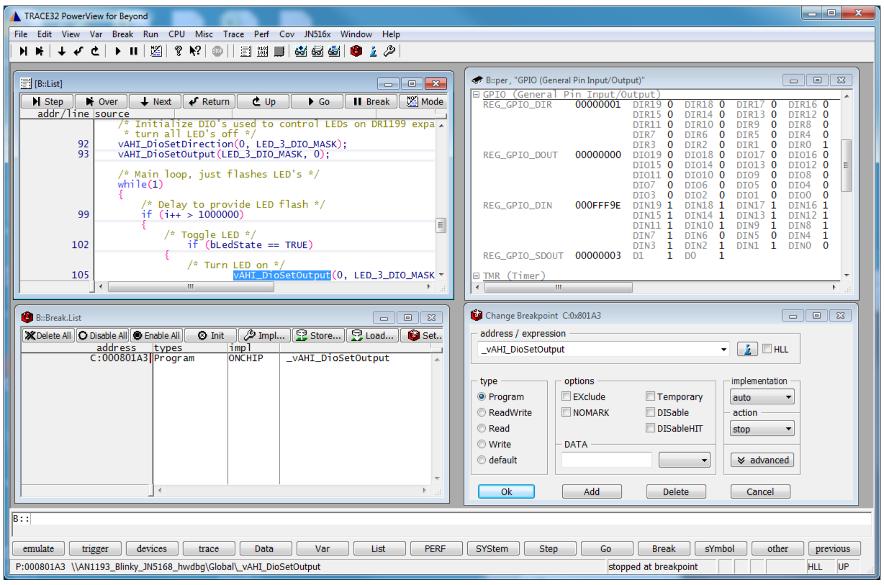
5. This displays a ‘GPIO registers monitor’ window in which the LED drive can be seen on DIO0. Note that both the output and input will change as the LED is driven.



```
B::per , "GPIO (General Pin Input/Output)"
⊟ GPIO (General Pin Input/Output)
  REG_GPIO_DIR     00000001  DIR19 0  DIR18 0  DIR17 0  DIR16 0
                             DIR15 0  DIR14 0  DIR13 0  DIR12 0
                             DIR11 0  DIR10 0  DIR9  0  DIR8  0
                             DIR7  0  DIR6  0  DIR5  0  DIR4  0
                             DIR3  0  DIR2  0  DIR1  0  DIR0  1
  REG_GPIO_DOUT    00000000  DIO19 0  DIO18 0  DIO17 0  DIO16 0
                             DIO15 0  DIO14 0  DIO13 0  DIO12 0
                             DIO11 0  DIO10 0  DIO9  0  DIO8  0
                             DIO7  0  DIO6  0  DIO5  0  DIO4  0
                             DIO3  0  DIO2  0  DIO1  0  DIO0  0
  REG_GPIO_DIN     000FFF8E  DIN19 1  DIN18 1  DIN17 1  DIN16 1
                             DIN15 1  DIN14 1  DIN13 1  DIN12 1
                             DIN11 1  DIN10 1  DIN9  1  DIN8  1
                             DIN7  1  DIN6  0  DIN5  0  DIN4  0
                             DIN3  1  DIN2  1  DIN1  1  DIN0  0
  REG_GPIO_SDOUT   00000003  D1    1  D0    1
⊟ TMR (Timer)
  ⊞ TMR 0
  ⊞ TMR 1
  ⊞ TMR 2
```

```
B::Break.List
       address  types    impl
   C:000801A3   Program  ONCHIP    _vAHI_DioSetOutput
```

6. Click on **vAHIDioSetOutput** in the B::Break.List window to display the breakpoint dialogue box and then resize the four windows as shown below.



```
[B::List]
addr/line source
          /* Initialize DIO's used to control LEDs on DR1199 expa
           * turn all LED's off */
      92  vAHI_DioSetDirection(0, LED_3_DIO_MASK);
      93  vAHI_DioSetOutput(LED_3_DIO_MASK, 0);

          /* Main loop, just flashes LED's */
          while(1)
          {
              /* Delay to provide LED flash */
      99      if (i++ > 1000000)
              {
                  /* Toggle LED */
     102              if (bLedState == TRUE)
                  {
                      /* Turn LED on */
     105                  vAHI_DioSetOutput(0, LED_3_DIO_MASK
```

```
B::per , "GPIO (General Pin Input/Output)"
⊟ GPIO (General Pin Input/Output)
  REG_GPIO_DIR     00000001  DIR19 0  DIR18 0  DIR17 0  DIR16 0
                             DIR15 0  DIR14 0  DIR13 0  DIR12 0
                             DIR11 0  DIR10 0  DIR9  0  DIR8  0
                             DIR7  0  DIR6  0  DIR5  0  DIR4  0
                             DIR3  0  DIR2  0  DIR1  0  DIR0  1
  REG_GPIO_DOUT    00000000  DIO19 0  DIO18 0  DIO17 0  DIO16 0
                             DIO15 0  DIO14 0  DIO13 0  DIO12 0
                             DIO11 0  DIO10 0  DIO9  0  DIO8  0
                             DIO7  0  DIO6  0  DIO5  0  DIO4  0
                             DIO3  0  DIO2  0  DIO1  0  DIO0  0
  REG_GPIO_DIN     000FFF9E  DIN19 1  DIN18 1  DIN17 1  DIN16 1
                             DIN15 1  DIN14 1  DIN13 1  DIN12 1
                             DIN11 1  DIN10 1  DIN9  1  DIN8  1
                             DIN7  1  DIN6  0  DIN5  0  DIN4  1
                             DIN3  1  DIN2  1  DIN1  1  DIN0  0
  REG_GPIO_SDOUT   00000003  D1    1  D0    1
⊟ TMR (Timer)
```

```
B::Break.List
       address  types    impl
   C:000801A3   Program  ONCHIP    _vAHI_DioSetOutput
```

Change Breakpoint C:0x801A3

address / expression: _vAHI_DioSetOutput ☐ HLL

type: [✓ Program] ReadWrite Read Write default

options: ☐ EXclude ☐ Temporary ☐ NOMARK ☐ DISable ☐ DISableHIT

implementation: auto

action: stop

```
P:000801A3 \\AN1193_Blinky_JN5168_hwdbg\Global\_vAHI_DioSetOutput    stopped at breakpoint    HLL  UP
```

7. Press the Go [⏵] button. The code executes until a call is made to the API function that changes the DIO state. Each time the button is pressed, you will see the LED toggle its state and the ‘GPIO registers monitor’ window will reflect the activity on the DIO that drives the LED. The B::Break.List window will highlight the listed breakpoint as being reached.

Appendix B: Troubleshooting

If you have not previously used the Lauterbach debugger with the NXP JN516x-EK001 Evaluation Kit, you are advised to follow the step-by-step guide in this Application Note. If you have previously followed this guide and are now having problems with your own application, check the following:

1. The JTAG jumper is correctly set

The JTAG jumper on the DR1174 Carrier Board must be set to match the type of application that is currently loaded. If a non-debug application is currently loaded, the jumper must be set to **DIS** so that the Flash programmer can connect and download an application to unlock JTAG. If the current application unlocks JTAG, the jumper must be set to **EN** so that the debugger can connect to JTAG. Refer to Section 3.2.4.

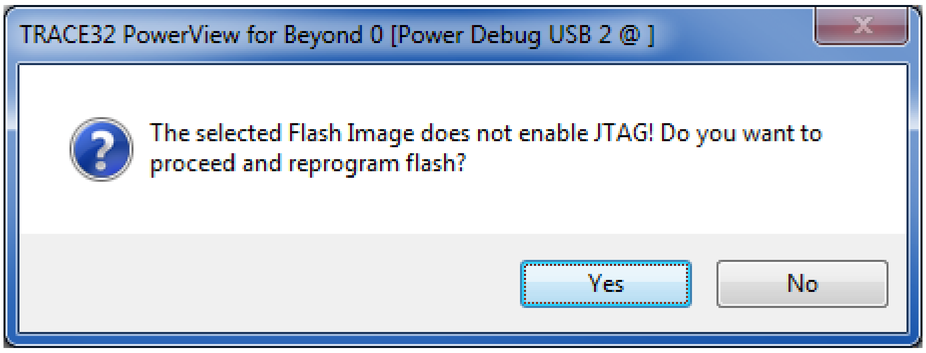
2. The currently loaded application unlocks JTAG in the bootloader

The bootloader determines whether the JTAG interface or UART0 is enabled in an application. If the application currently loaded has been built without JTAG enabled, it will not be possible to connect the Lauterbach debugger to download a new application. In this case, an application which unlocks JTAG must be downloaded using the JN51xx Flash Programmer. It is not necessary to download the application that you are currently debugging. Any application built for debug can be used to unlock JTAG. Once the debugger connects, it can then update the Flash memory with the new application. The application built as part of this Application Note can be used as a JTAG unlock application. To ensure that your application has been built with JTAG, follow the steps in Section 6.

The **flashdemo.cmm** script below checks that JTAG is enabled before programming a new image:

C:\T32\demo\beyond\hardware\jn5168\big_endian\flashdemo.cmm

If JTAG is not enabled in the application, the following warning is displayed:


TRACE32 PowerView for Beyond 0 [Power Debug USB 2 @]

The selected Flash Image does not enable JTAG! Do you want to proceed and reprogram flash?

If you click ‘Yes’ to proceed, the application will be loaded and the debugger will no longer be able to access the JTAG interface. You will need to switch the JTAG jumper back to DIS to enable the UART for Flash programming and debugging.

3. The Lauterbach debug adapter cable has pin 1 correctly aligned on the DR1174 Carrier Board

The connection on the Carrier Board is not polarised. See the diagram in Section 3.2.5.

Appendix C: Additional Hardware Modifications

Details on the JTAG adapter can be found at:

http://www.lauterbach.com/frames.html?adnxp.html

Additional functionality is available that has not been tracked on the DR1174 Carrier Board. If you are designing your own hardware you can include the following:

1. Reset Control

The Lauterbach debug hardware drives NRESET to reset the JN516x device. This is pin 8 of the 10-way connector. Connecting NRESET to the reset line on the JN516x device is useful when the JTAG hardware does not allow reset. In particular, the JTAG hardware is disabled when the JN516x device is in sleep mode, as described in Section 4.

2. Automatic Unlock

The bootloader executes a short delay with JTAG enabled, if the MISO (program) line of the JN516x device is held during boot. This gives the Lauterbach hardware time to connect, even if the currently programmed application does not unlock the JTAG interface. Note that the JTAG jumper on the Carrier Board must still be set to **EN**. The MISO line is pin 7 on the adapter cable. This line is driven via an open collector circuit on the adapter board so that the connection is isolated if the line is not being driven. The **jtag_unlock.cmm** script below can be run to drive the MISO line during boot:

C:\T32\demo\beyond\hardware\jn5168\big_endian\jtag_unlock.cmm

Revision History

Version	Notes
1.0	First Beta version
1.1	Second Beta version
1.2	First full release version
1.3	Updated to use Digital 4 for UART 1 with LCD Expansion Board (DR1215)
1.4	Updated for BeyondStudio

Important Notice

NXP Semiconductors

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com