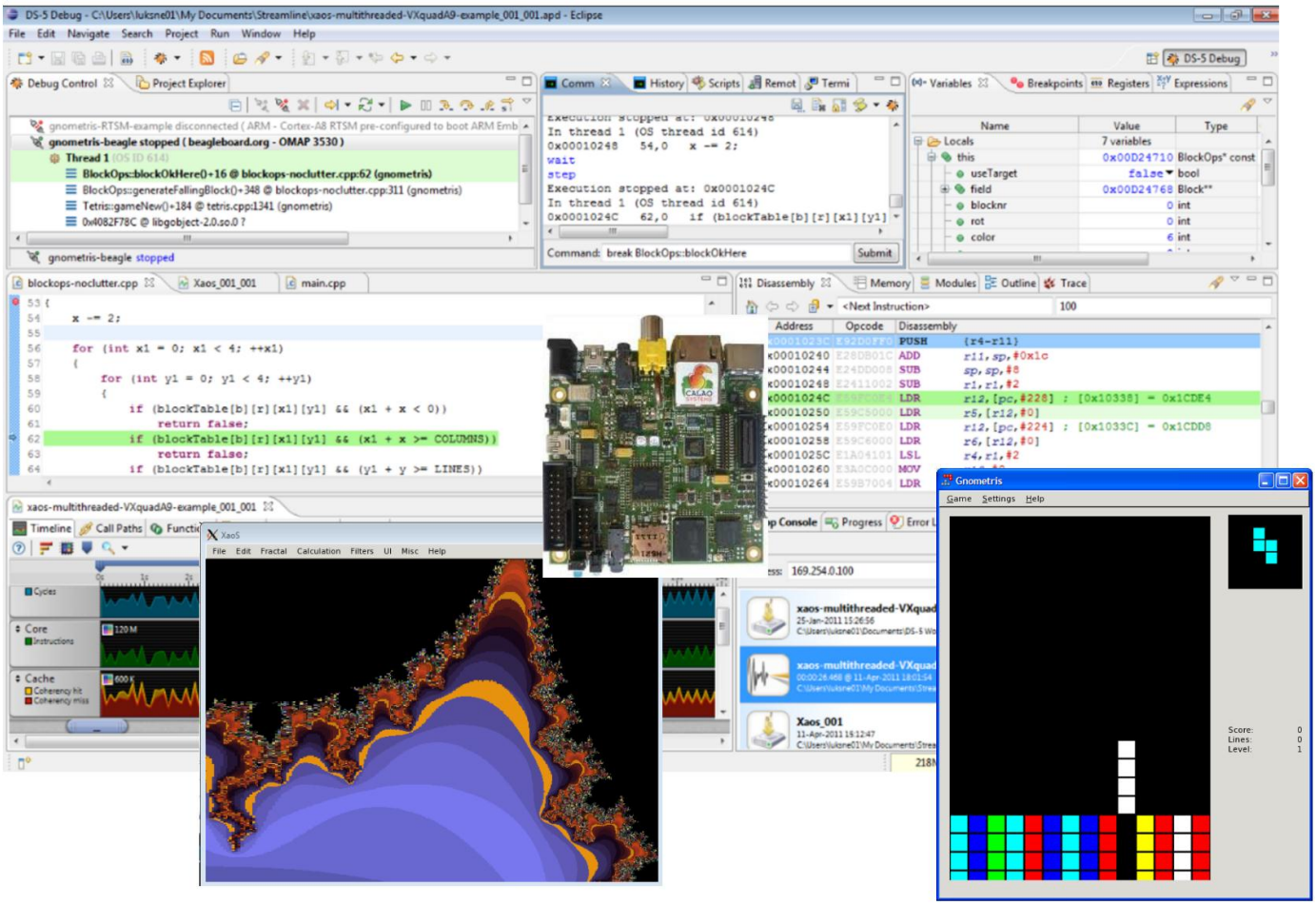
DS-5 Workshop: Linux Kernel and Application Debug, Trace and Profile on Snowball



This workshop demonstrates the features of the DS-5 Debugger by debugging and tracing U-Boot, the Linux kernel and a game called Gnometris that is an ARM Linux application running on the target. It also demonstrates the ARM Streamline profiler using an application called Xaos. It will introduce the views of DS-5 Debugger and ARM Streamline and demonstrate their features. Some of the features are used in explicit instructions that you are expected to follow and are marked with this symbol ⇨. Other features are only mentioned and are not critical to the flow of the workshop. You don't have to use them, but you are generally encouraged to try them out anyway.

We will use a Snowball board as the target, but DS-5 can also be used with any ARM Linux target that has networking or the Real Time System Models that come with DS-5. The Gnometris and Xaos applications that we will be using are supplied as examples with DS-5.

(version 31 January 2013; v5.13 d1622; HW; HWCortex-A9; DSTREAM; DSTREAMLinaro; DSTREAMSnowball; Snowball)

Contents

Preparation

If the host and target have already been setup for you, you can skip down to *Starting Eclipse* below. Otherwise you’ll need to start by installing some software.

Host Setup

DS-5 can be used on Windows and Linux hosts. If your host has not already been setup for you then you will need to follow the instructions in the appendix on page 87

Target Setup

For the workshop you’ll be supplied with a Snowball board that already has U-Boot, a Linux kernel and root filesystem on it. If you want to setup your own target please see *Appendix B: Snowball*: *Linaro Linux target setup* page 94.*Serial setup*

Starting Eclipse

⇨ **Start** > **All Programs** > **ARM DS-5** > **Eclipse for DS-5** and choose a location for the workspace where Eclipse projects will be stored. The default workspace location is fine.

⇨ If this is the first time you've started DS-5 you will see the "Welcome to ARM DS-5" home page; click **Go to the workbench**. You can get the **Welcome** page back if you want it later by choosing **Help** > **Welcome**.

Installing a DS-5 License

In order to use DS-5 for this workshop you will need to have a license for DS-5 Basic Edition, either a full or evaluation license. (A license for Professional Edition will also work.) If the workshop has been setup for you then you will already have a license. If you don’t have a license a dialog will open with an explanation and a button to **Open License Manager...**. You can get an evaluation or full license by opening the license manager using **Help** > **ARM License Manager...**, clicking the **Obtain License...** button and following the instructions. You will need to be connected to the internet. When you receive the license file, you need to add it by using the **Add License...** button.

Eclipse views (a brief digression)

The Eclipse window is divided into a number of rectangular *panes*. Each pane contains one or more *views*. Each view has a tab at the top with the view's name. For example, in the screenshot on the cover, **Debug Control**, **Project Explorer** and **Remote Systems** are three views in the same pane. Views of source files are named after their files, for example **main.cpp**.

You can easily rearrange panes and views. Dragging the dividing lines between panes changes their size. Clicking on a view's tab brings it to the front. You can drag a view's tab to move the view to another pane or to split a pane vertically or horizontally. You can even drag a view out of the Eclipse window entirely to create a new window. Double-clicking on the tabs of a pane will expand the pane to fill the window (maximize). Double-clicking it again will restore it down.

You can close a view by clicking the close button in its tab. If a view you want is not open, you can open it by using the **Window > Show view >** menu.

A particular grouping of views is called a perspective. For example, there is a **DS-5 Debug** perspective that we will be using, but there is also a C/C++ perspective and others. Eclipse provides ways to switch quickly between perspectives and to create your own custom perspectives from a group of views.

Some of the DS-5 views such as **Variables**, **Expressions**, **Registers, Functions** and **Modules** have columns. You can change the width of the columns by dragging the dividing lines in the column headers right or left. You can change which columns are displayed by right-clicking on the column headers and choosing from the context menu. In **Functions** and **Modules** you can sort by any of the columns by clicking on the column heading. If you shift-click a different columns you can set further minor sort keys which shows as the sort arrow and two or more dots.

There are more things you can do, such as minimizing panes and setting views to be pop-up "Fast Views", but that's probably enough to get you started.

Getting help (more digression)

You can press the **F1** key (or Shift+**F1** on Linux) at any time to get help about the current view. You can use **Help > Help Contents > ARM DS-5 Documentation** to view the documentation. You can access cheat sheets for various tasks using **Help > Cheat Sheets > ARM …**, for example importing the example projects. (But we won't be following those cheat sheets here.)

Bare-metal Debug and Trace (U-Boot)

First we are going to debug the U-Boot bootloader. It is some of the earliest code run by the processor after reset. It is responsible for initializing the hardware (for example, power, clocks, memory controllers, etc.), reading the kernel into RAM, setting up the kernel parameters (for example, the kernel command line, also known as the bootargs) and jumping to the kernel. U-Boot is a "bare-metal" program which means that it runs directly on the processor and does not use or need an operating system "beneath" it. U-Boot does not use the MMU and runs on a single core even in a multi-core system. U-Boot also has a command-line which we will use during our debugging.

Setting up your own target: Download, build and setup U-Boot

If you are running this workshop on your own host you will need to do some setup to build and debug U-Boot. You can download the U-Boot sources using the Ubuntu package manager on the target and then build it on the target. Once built, the sources and symbols can be copied to your host and used during the debug session. To complete your own setup please follow the *U-Boot download and debug setup* instructions in the appendix on page 94.

Connect the hardware

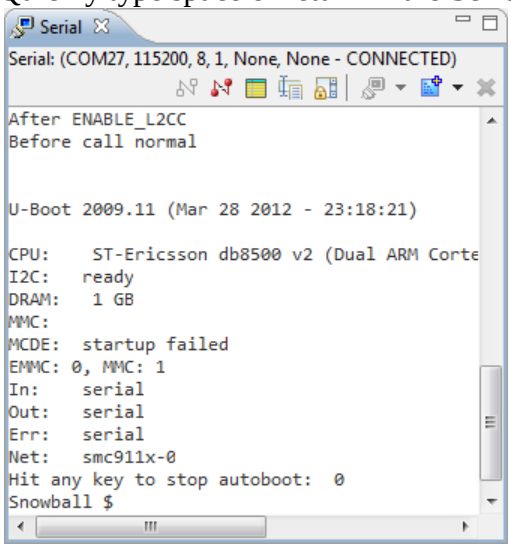
- ⇨ Connect the DSTREAM unit to its probe box using the 100-conductor ribbon cable. (Be sure to count the conductors! (Not really.))
- ⇨ Connect the DSTREAM unit to the host using a USB cable. The DSTREAM can also be connected via Ethernet. In this workshop we will use USB since we will be using Ethernet to connect to the target for application debug later. The DSTREAM is used to control the target when debugging at the bare-metal (U-Boot) and kernel level.
- ⇨ Connect the DSTREAM power (5V). Do not connect the target's power yet because we don't want it to boot yet.
- ⇨ Connect the JTAG port of the Snowball board to the port labelled **ARM JTAG 20** on the DSTREAM probe box using a 20-conductor ribbon cable.
- ⇨ Connect the 5V power cable to the target, but don't press the blue power button yet. Connect the serial port of the Snowball board to the host using a USB cable and connect the **Serial** view. See the in the appendix on page 92.

When the power button is pressed the target will run Xloader and U-Boot from the builtin eMMC. U-Boot will write various messages to the serial console and, if no input is given within a few seconds, load and boot the Linux kernel. To debug U-Boot, we want to stop it at its command line.

- ⇨ With the **Serial** view already connected, press the blue power button to boot the board.

U-Boot should begin printing.

- ⇨ Quickly type space or return in the **Serial** view to stop U-Boot at its command line:



If nothing happens, try disconnecting and reconnecting the target power again. When you remove power from the Snowball, the serial connection will be broken and the Serial view will begin repeating **`Bad file descriptor in nativeavailable`**. Click the Disconnect button () in the **Serial** view, reconnect the power (without pressing the power button) and then click the Connect button ().

You can try typing **`help`** and other U-Boot commands, but leave U-Boot at the command line when you finish experimenting.

Debugging U-Boot on the target

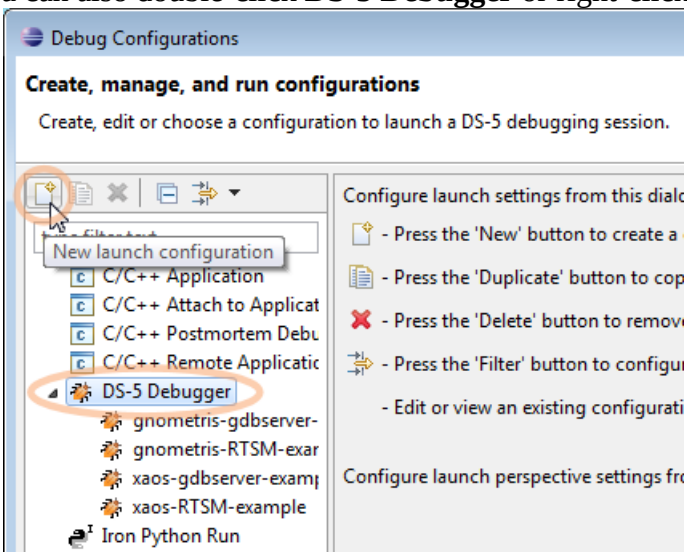
In this section we will use DS-5 to load the U-Boot we built on the target. We will load U-Boot into the target's RAM as the binary on the eMMC may be different from the one that we've built and have debug symbols for.

Next we will examine or create a debug configuration so that DS-5 can connect to the target.

- ⇨ Change to the **DS-5 Debug** perspective **Window > Open Perspective > Other… > DS-5 Debug**
- ⇨ Choose **Run > Debug Configurations...** then expand **DS-5 Debugger**

The **U-Boot** debug connection may already have been created for you in which case you can just select it, examine the settings, choose the DSTREAM unit and set the DTSL Options. (If the Gnometris and Xaos example projects have already been imported, you will see four debug configurations, **gnometris-gdbserver-example**, **gnometris-RTSM-example**, **xaos-gdbserver-example** and **xaos-RTSM-example** in the **Debug Configurations** dialog which we are going to ignore for now.)

- ⇨ If there is no **U-Boot** debug connection as a child of **DS-5 Debugger**, then create one: select **DS-5 Debugger** and click the **New launch configuration** button () to create a new debug configuration. (You can also double-click **DS-5 Debugger** or right-click it and choose **New** instead.)

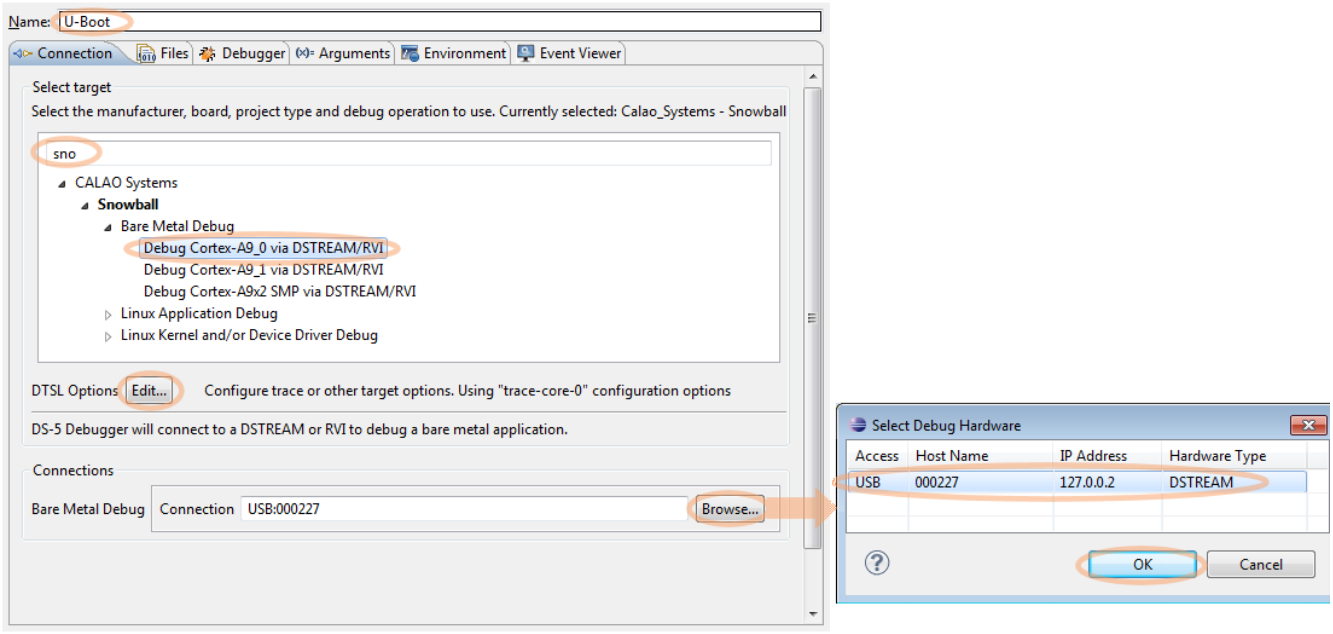


- ⇨ Give the debug configuration a name, I used ***U-Boot***.

In the **Connection** pane of the debug configuration we need to specify the platform and choose the DSTREAM. The plaforms list contains the large number of platforms that DS-5 supports by default arranged as a tree by vendor.

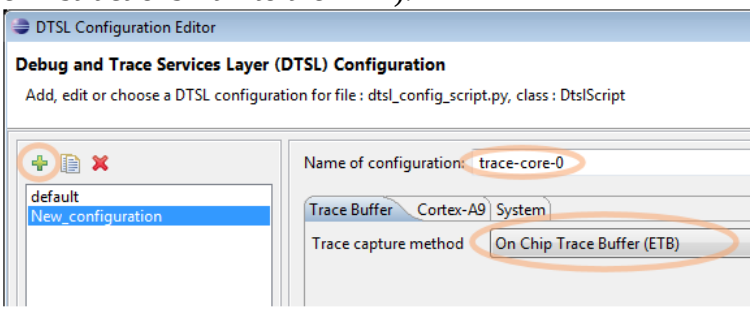
- ⇨ Type ***sno*** in the Filter platforms filter box so that the only the matching platforms are shown.
- ⇨ Expand **CALAO Systems > Snowball >Bare Metal Debug** in the platforms list.
- ⇨ Select **Debug Cortex-A9_0 via DSTREAM/RVI**.

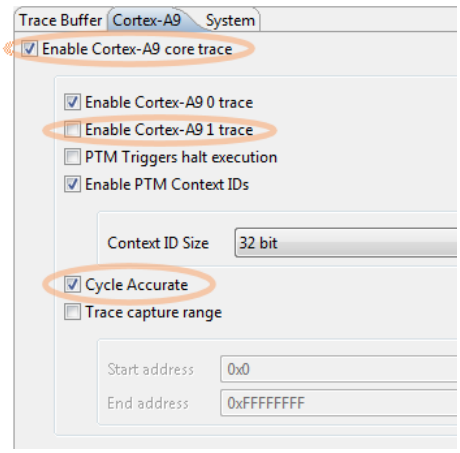
⇨ Click the **Browse** button and choose the DSTREAM unit and click **OK**. Your Connection number will be different than shown.



The tracing options are configured in a separate DTSL configuration. "DTSL" stands for Debug and Trace Services Layer. We'll enable tracing on core 0.

⇨ Click the **DTSL Options Edit…** button to open the **DTSL Configuration Editor** dialog box.
⇨ Click the Add button (✚) to create a new DTSL configuration.
⇨ Give the new DTSL configuration a name, I used ***trace-core-0***.
⇨ In the **Trace Buffer** pane, change the **Trace capture method** to **On Chip Trace Buffer (ETB)**. The Snowball doesn't have a connector for external (TPIU) trace.
⇨ In the **Cortex-A9** pane, check **Enable Cortex-A9 core trace** and uncheck **Enable Cortex-A9 1 trace** since U-Boot only runs on core 0.
⇨ Check **Cycle Accurate** so that the trace data will include cycle count (although this will mean that fewer instructions fit into the ETB).





You can leave **Enable PTM Context IDs** checked. U-Boot doesn't change the context ID so how we set it won't matter. You can disable **Cycle Accurate** trace if you want more instructions to fit into the ETB. It's also possible to setup an initial trace capture range to limit what instruction addresses are captured, but we don't want to do that.

⇨ Click **OK** in the **DTSL Configuration Editor** dialog box.

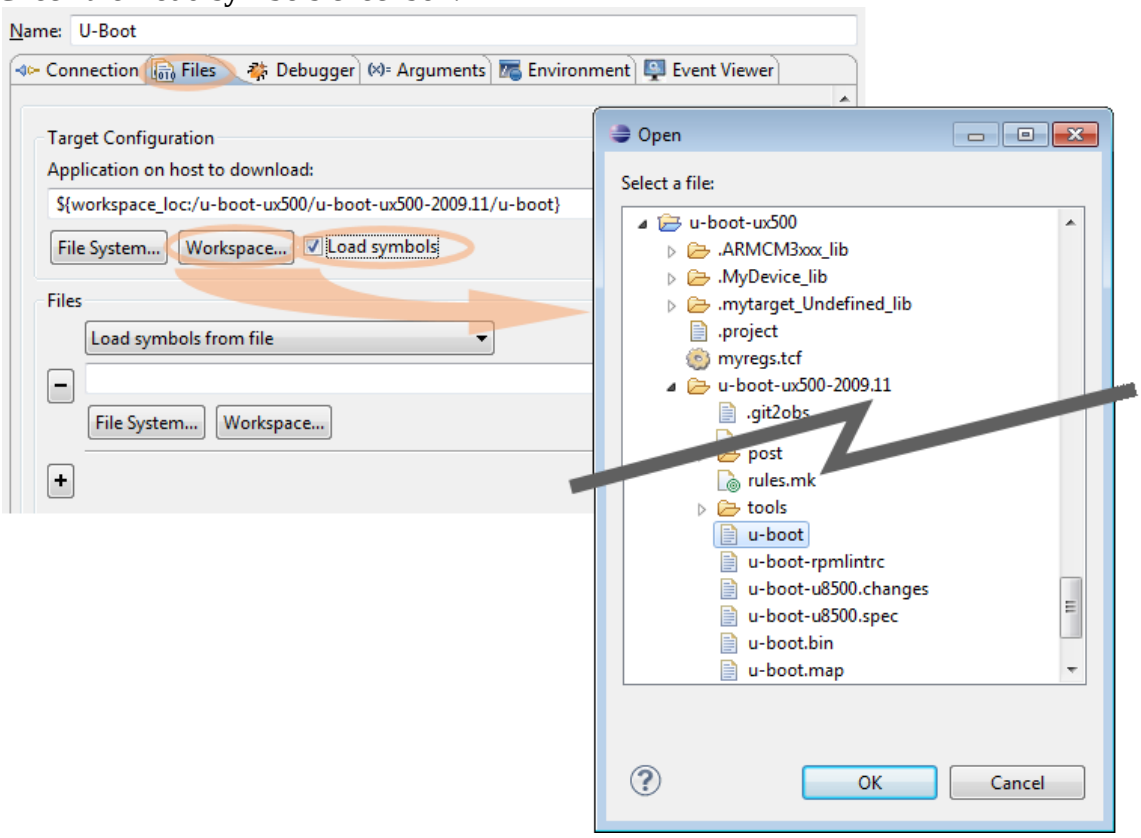
The **Debug** button is disabled because the debug configuration is not yet complete. The problem is explained near the top of the dialog: **[Debugger]: Debugging from a symbol, but no symbol files defined in the Files tab**. We will fill in the missing details in the other panes.

The **Files** pane of the debug configuration allows us to specify an application to download and symbols (debug information) to load. We're going to reload the U-Boot on the target so that we can be sure the U-Boot on the device matches the symbols we have.

We need to specify the location of the `u-boot` binary containing symbols (debug information):

⇨ In the **Files** tab under **Application on host to download**, click the **Workspace…** button and select the `u-boot` binary from the project `u-boot-ux500/u-boot-ux500-2009.11/u-boot`

⇨ Check the **Load symbols** checkbox:

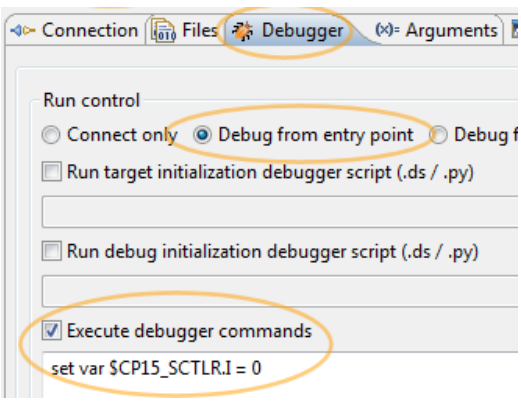


In the **Debugger** pane of the debug configuration:

⇨ Choose **Debug from entry point** so that DS-5 will load U-Boot and set the program counter to the entry point but not start running it.

⇨ We need to disable the I-Cache for this example. Check the **Execute debugger commands** and enter

```
set var $CP15_SCTLR.I = 0
```



You can specify debugger scripts and/or commands to execute as part of the connection process. The dialog fields have tooltips that explain at what point during connection the scripts and

commands are executed. An example would be to use the **Execute debugger commands** field to stop the target and load the debug symbols or disable some breakpoints and enable others. The script files can be written in either the simpler, gdb-like, DS-5 command line scripting language (`.ds` files) or the more powerful and complex scripting language Python (`.py` files).

You can specify the host working directory which affects certain debugger commands that access the host file system (for example `dump`, `log` and `source`). The default is fine.

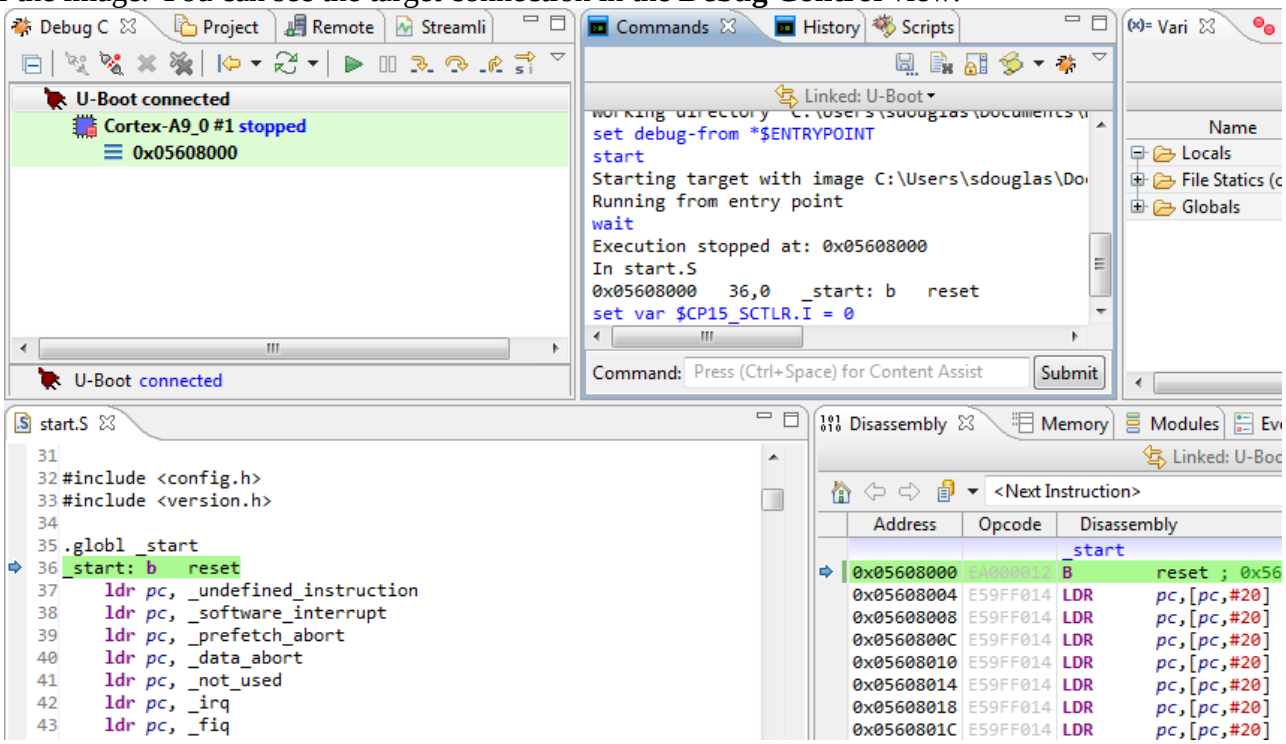
You can use the **Paths** list to specify directories that DS-5 should search for source files but we don't need to use it for U-Boot and can just leave it empty.

The **Arguments** and **Environment** panes don't apply to bare-metal debugging, so just leave them blank.

The **Event Viewer** pane is used to configure display of any ITM or STM information in the trace stream. We won't be using it on our target, so just leave it blank, too.

⇨ Click the **Debug** button. If the **Debug** button is disabled then Eclipse will put a message explaining why at the top of the **Debug Configurations** dialog. If the **Confirm Perspective Switch** dialog appears check **Remember my choice** and click **OK**.

DS-5 will connect to the target and load U-Boot into memory. The program counter will be set to the entry point of the image. You can see the target connection in the **Debug Control** view.



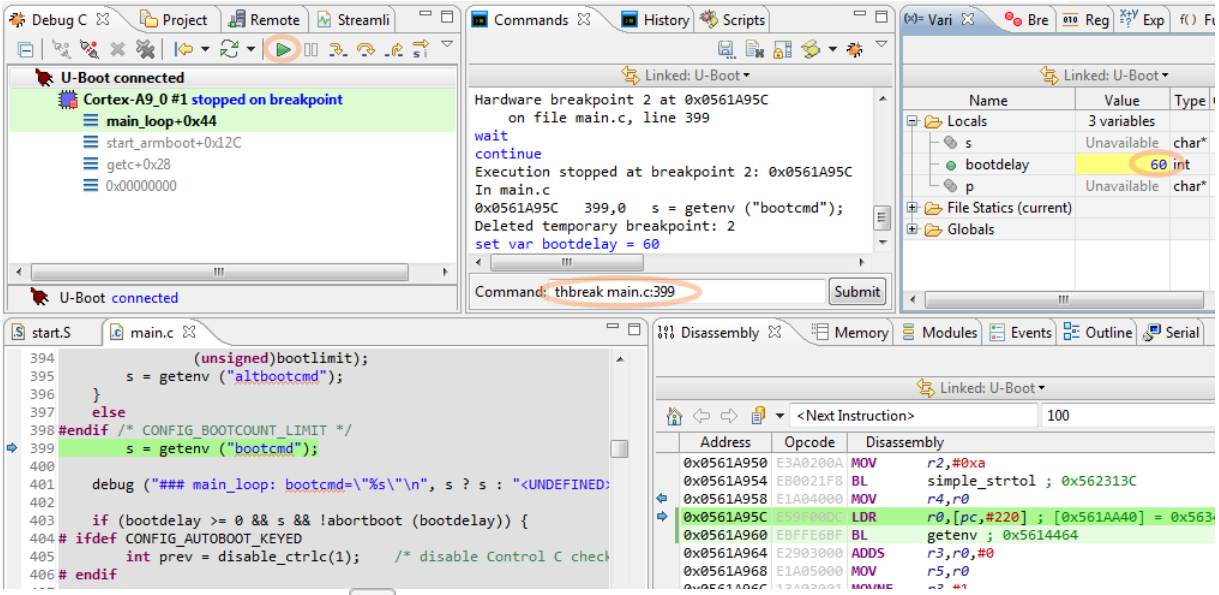
[When U-Boot runs you need to type a character to the serial port to stop it from autobooting Linux. In some cases the default boot delay can be very quick, and before you blink the Linux kernel boots. If this happens the debug connection will probably stop on an exception with the `PC` near address `0xFFFF0000` with the MMU on, which will prevent the debugger from loading the `u-boot` image. If this happens you need to disconnect the debug configuration, unpower and repower the board, re-establish the serial connection, push the blue button and try to be quicker.]

We can place a breakpoint in the `main_loop` function to stop boot process and change the boot delay time:

⇨ In the **Commands** view type `thbreak main.c:399` and press return to place a temporary hardware breakpoint in the function `main_loop`. Later we'll see how to set breakpoints in other ways.

⇨ Click the Continue button ( ) in the **Debug Control** view. The debugger will stop on line 399 in `main.c`. Because the breakpoint was temporary it was deleted automatically.

⇨ Find the variable `bootdelay` in the **Locals** folder of the **Variables** view and change its value to 60. Or you can type `set var bootdelay = 60` in the **Commands** view and press return.



⇨ Click the Continue button ( ).

Presto, in the serial console you should now see U-boot counting down from 60 seconds. This should be ample time to stop U-Boot from auto booting Linux.

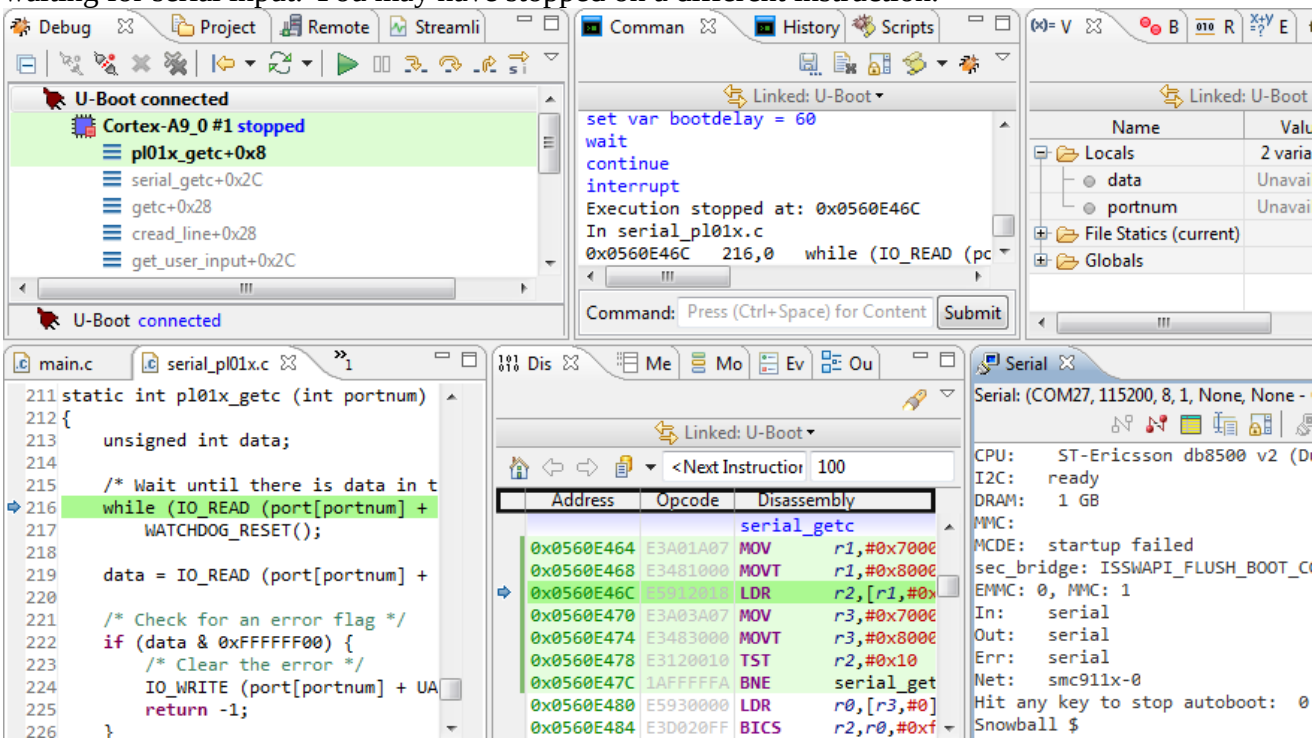
⇨ In the **Serial** view, press any key to stop U-Boot before it finishes counting down.

In the some of the steps above, we used the **Commands** view interface to control the debugger. Don't worry you don't need to remember all these commands. All the commands can be done through the DS-5 Debugger UI and we will go through them in the remainder of the workshop. And the command-line has pop up Content Assist help available, which we'll also show later.

⇨ Send a few characters to the target, but not return, by typing them into the **Serial** view. We'll see them in the debugger in a minute.

⇨ Click the Interrupt button ( ) in the **Debug Control** view to stop the target:

Now we can see that we've stopped in the function **`pl01x_getc`**, which seems reasonable as u-boot is waiting for serial input. You may have stopped on a different instruction.



[If you see no functions in the stack and the **`PC`** is at an address like **`0xC00xxxxx`** then you have not stopped in U-Boot and the kernel has booted. Reset the target (power off and on) and stop U-Boot from booting the kernel by typing to the **Serial** view.]

The current PC location is shown by an arrow ( ) in the left margin of both the source and **Disassembly** views and by the dark green highlighting. The light green highlighting in the **Disassembly** view shows all of the assembly instructions that correspond to the current source line.

If the processor supports debugging both TrustZone worlds then memory addresses for Normal world are prefixed with **`N:`**, while addresses in the Secure world are prefixed by an **`S:`**. If the processor doesn't support TrustZone, or has been configured so that it is only possible to debug Normal world then the addresses will not have a prefix.

Notice how the actions that you have performed so far, like **`interrupt`** have been recorded in the **Commands** and **History** views. If you want to automate that part of connecting to the target, you could copy them and paste them into the **Execute debugger commands** field of the **Debugger** pane of the debug connection.

Quick tour of the debugger

Now we'll show a few of the features of the various debugging views. Most of them will also be discussed in more detail in the application debugging section later.

In the **Debug Control** view we can see the call stack with the frame of the current function at the top and its caller below it and its caller's caller below that and so on. You can see in the **Variables** view that the current function, **`pl01x_getc`**, has no local variables.

Let's do some stepping:

⇨ Click the Step (into) button (; **F5**) in the **Debug Control** view.

This will continue the program until the current source line is finished. That will happen when you type a character to the target, so

⇨ Type a character to the **Serial** view.

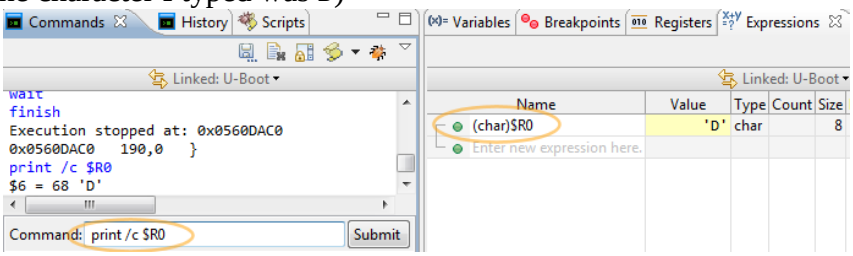
The target stops again on the **`data = IO_READ...`** statement because the step has finished.

⇨ Click the Step Out button (; **F7**). This will cause the target to continue until **`pl01x_getc`** returns to its caller, **`serial_getc`**.

The stack frame of **`pl01x_getc`** is no longer shown in the **Debug Control** view. The value returned will be in register **R0**.

⇨ Look in the **Registers** view and expand the **Core** registers folder. The value of the character you typed is in **R0**.

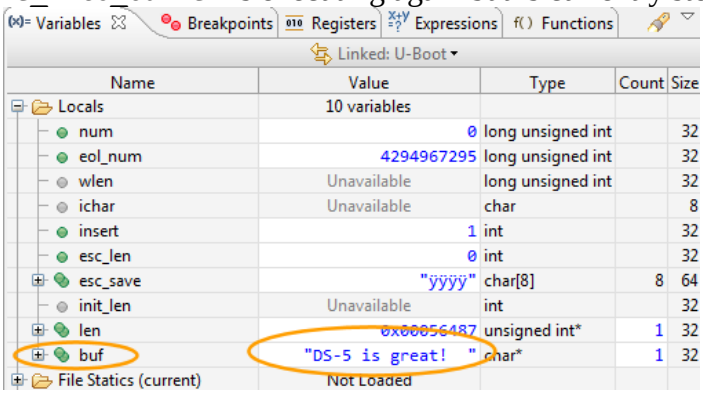
⇨ The ASCII number in hex is displayed in **R0**. You can quickly display the character by adding the expression **`(char)$R0`** to the **Expressions** view or typing **`print /c $R0`** in the **Commands** view console. (The character I typed was **`D`**)



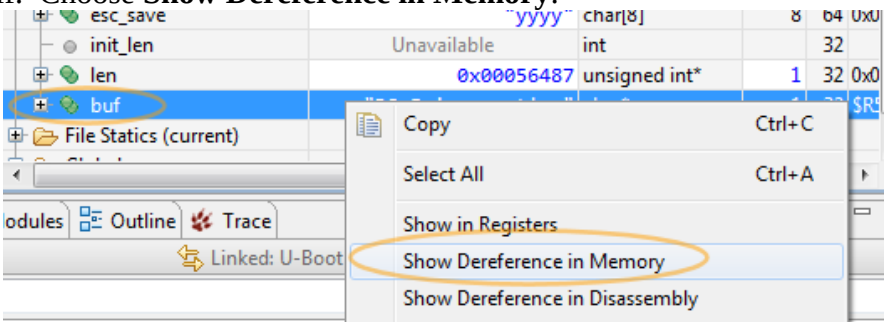
It is also possible to step by assembly instruction instead of by source line by using the toggle button (;). You can try it if you want, but set it back to stepping-by-source-line mode with the '**s**' dark ( ) afterwards.

⇨ Click the stack frame of **`cread_line`** to select it.

Now we can see the context of **`cread_line`** in the various views. The PC arrow and highlighting in the source and **Disassembly** views shows the instructions that will be executed when we return to this function. In the **Variables** view can see the **`buff`** variable which points to the text we've typed to the target. The variables view has a **Location** column that show us that the variable **`buff`** is held in register **`R5`** when **`readline_into_buffer`** is executing again but is currently stored in memory (on the stack):

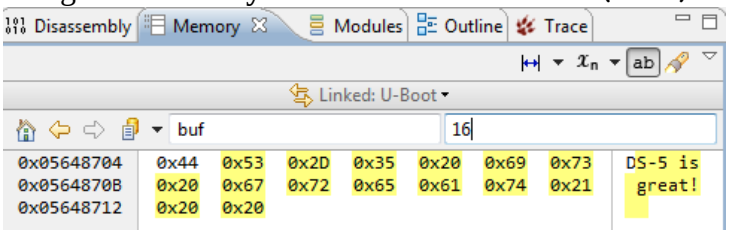


⇨ Right-click the variable **`buff`**. You can see choices in the context menu for formatting the variables is various ways, for example **Float** and **Hexadecimal**. There are also menu items to show the variable in a **Memory**, **Disassembly** or **Registers** view and to show the variable dereferenced in **Memory** or **Disassembly** views. Dereferencing shows what the variable points to instead of showing the variable itself. Choose **Show Dereference in Memory**.



The memory that **`buff`** is pointing to is displayed in a **Memory** view. Because it's not possible in C to know how many bytes **`buff`** is pointing at, only one byte is shown.

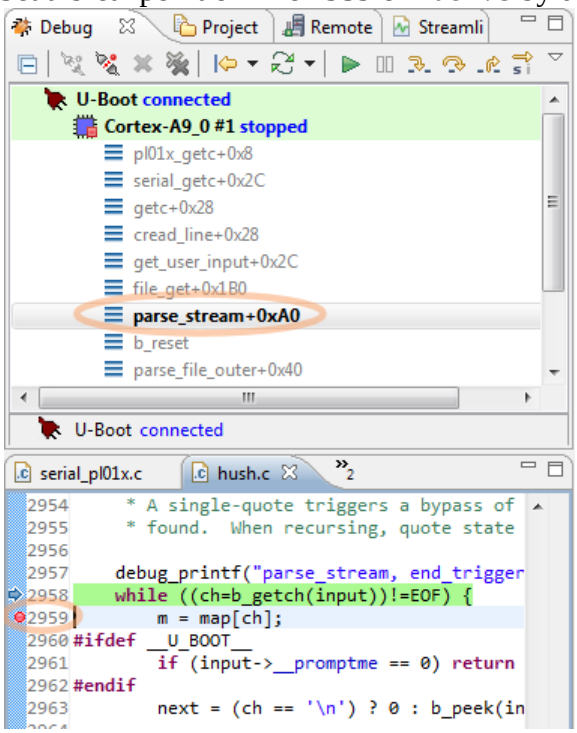
⇨ Change the **Memory** view size from **`sizeof *(buff)`** to **`16`**



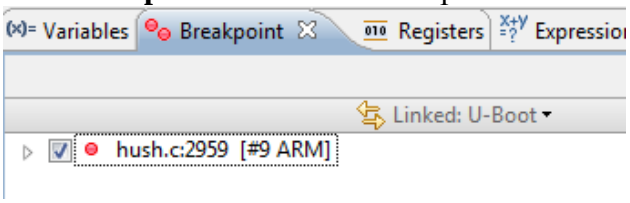
You can edit the memory either in hex or as characters. Making changes will not have any immediate effect on what is shown the **Serial** view, but it will change what U-Boot thinks you have typed when you press return.

⇨ Click the stack frame of **`parse_stream`** to select it.

⇨ Set a breakpoint on line 2959 of **`hush.c`** by double-clicking on the line number in the left margin.



You can see the breakpoint (●) appears in both the source and **Disassembly** views. The breakpoint is also listed in the **Breakpoints** view where it's possible to set conditions and other fancy stuff that we see later.



Trace view

⇨ If the **Trace** view is not already open, open it by choosing the **Window > Show view > Trace** menu item.

The **Trace** view already has some instructions in it, but since U-Boot has just been looping in **`pl01x_getc`** it's not very interesting. Let's collect some more interesting trace to examine in the **Trace** view.

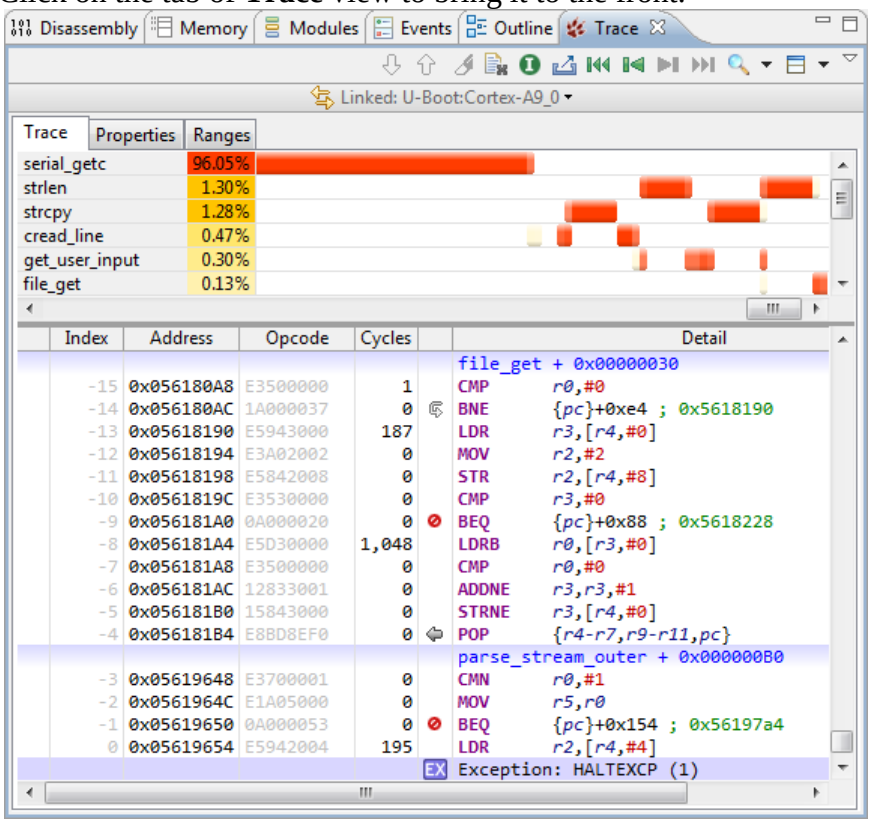
⇨ Click the Continue button (▶).

The target continues waiting for you to type the rest of the U-Boot command.

⇨ Type the return key into the **Serial** view.

The target hits the breakpoint in **`parse_stream`**.

⇨ Click on the tab of **Trace** view to bring it to the front.



By default, the **Trace** view collects all instructions executed up to the capacity of the1 trace buffer. Because the target has no external trace port, we're using an 8KB on-chip embedded trace buffer (ETB). The ETB is used as a circular buffer and holds a few 10s of thousands of instructions depending on the actual instruction sequences. If the trace buffer overflows (wraps) then only the most recent instructions will be shown when the target stops. If you need more control over when trace is collected, you can place trace start and stop points or trace trigger points, similarly to the way you can place breakpoints.

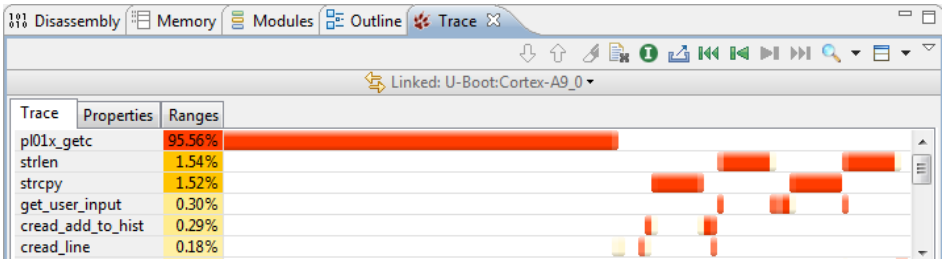
The **Trace** view shows a page of trace data at a time. The size of the page is initially 10,000 instructions but it can be set to any size between 1,000 and 1,000,000 instructions by using the **Set Trace Page Size…** menu item in the **Trace** view's drop-down menu ( ). You can use the buttons to move to the First ( ), Previous ( ), Next ( ) or Last ( ) page of trace. The First and Last buttons also change the order of the trace indices.

The **Trace** view is divided into three panes: **Trace**, **Properties** and **Ranges**:

- The **Trace** pane shows a history of functions and instructions that were executed in the current page of trace data. It is divided into two sections: on top is the **Navigation** section and on the bottom is the **Trace** section. These sections can be hidden or displayed using the drop-down menu ( ). You can drag the horizontal divider up and down to resize them.
- The **Properties** pane shows details of the trace capture and includes a **Stop Trace Capture on Trigger** checkbox. You may need to grow the view vertically to see it all.
- The **Ranges** pane allows you specify address ranges to limit tracing. You may need to grow the view vertically to see it all.

The **Navigation** section, at the top of the **Trace** pane, shows which functions were executed in the current page of trace data. It shows functions sorted by number of instructions executed and colored timelines of

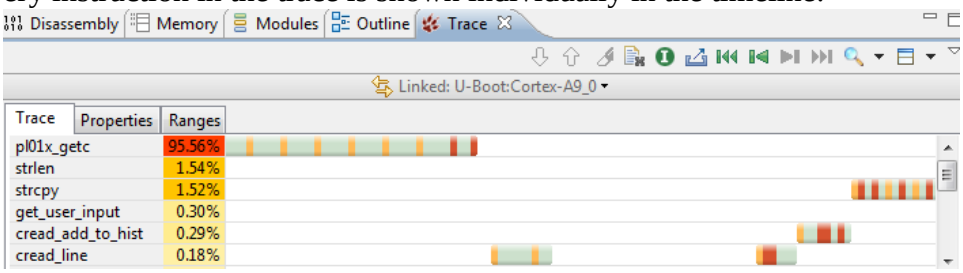
the execution. You can see the transfer of execution between functions:



You can zoom the **Navigation** section in and out in using the zoom drop-down menu ( ).

⇨ Use the zoom drop-down menu ( ) to change to **1:1** resolution and scroll all to the right end (most recent).

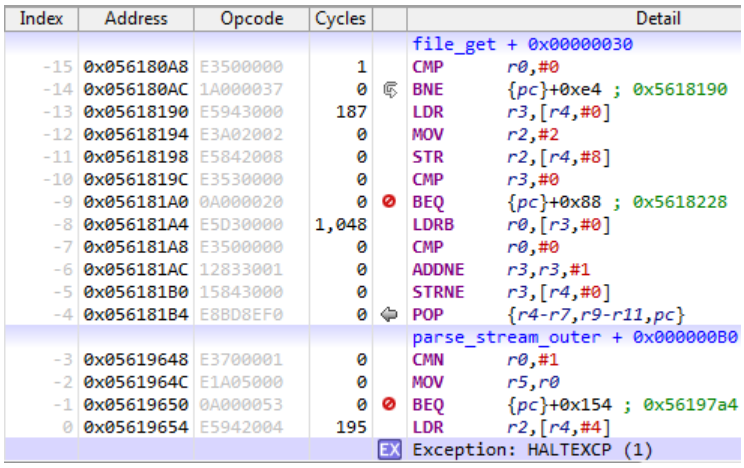
Now every instruction in the trace is shown individually in the timeline:



In order to show more instructions the trace data is not cycle accurate and does not include memory access addresses or values.

Cortex-A9 trace data (PTM) does not include cycle count information for individual instructions (and by default, none at all) so the coloring represents the instruction class (memory access, branch, ALU, ...). The Cortex-A9 trace hardware does not allow tracing memory accesses (addresses or values).

The **Trace** section, at the bottom of the **Trace** pane, shows the traced functions and instructions in the order they were executed.



Index	Address	Opcode	Cycles		Detail
					file_get + 0x00000030
-15	0x056180A8	E3500000	1		CMP r0,#0
-14	0x056180AC	1A000037	0		BNE {pc}+0xe4 ; 0x5618190
-13	0x05618190	E5943000	187		LDR r3,[r4,#0]
-12	0x05618194	E3A02002	0		MOV r2,#2
-11	0x05618198	E5842008	0		STR r2,[r4,#8]
-10	0x0561819C	E3530000	0		CMP r3,#0
-9	0x056181A0	0A000020	0	⊘	BEQ {pc}+0x88 ; 0x5618228
-8	0x056181A4	E5D30000	1,048		LDRB r0,[r3,#0]
-7	0x056181A8	E3500000	0		CMP r0,#0
-6	0x056181AC	12833001	0		ADDNE r3,r3,#1
-5	0x056181B0	15843000	0		STRNE r3,[r4,#0]
-4	0x056181B4	E8BD8EF0	0		POP {r4-r7,r9-r11,pc}
					parse_stream_outer + 0x000000B0
-3	0x05619648	E3700001	0		CMN r0,#1
-2	0x0561964C	E1A05000	0		MOV r5,r0
-1	0x05619650	0A000053	0	⊘	BEQ {pc}+0x154 ; 0x56197a4
0	0x05619654	E5942004	195		LDR r2,[r4,#4]
				EX	Exception: HALTEXCP (1)

The **Cycles** column shows zero for most instructions. This is because of the way PTM works: it only produces cycle counts on "waypoints" (essentially conditional or unpredicted branches) which gives the number of cycles since the previous waypoint. The cycle counts are much higher than expected here since the caches are disabled and peripherals (UART) are being accessed.

On Cortex-A9, conditional branches (and non-PC-relative branches) that are skipped because their condition codes are not met are shown with ⊘. The PTM trace information does not record whether other conditional instructions (for example ADDNE) are skipped or not.

The last instruction in the **Trace** section (index 0) is the last instruction executed before the target stopped on the breakpoint (which is shown by the **`HALTEXCP`**. At index -4, we can see that the **`file_get`** function returned to **`parse_stream`**.

⇨ Click on the **`MOV r5,r0`** instruction at index -2.

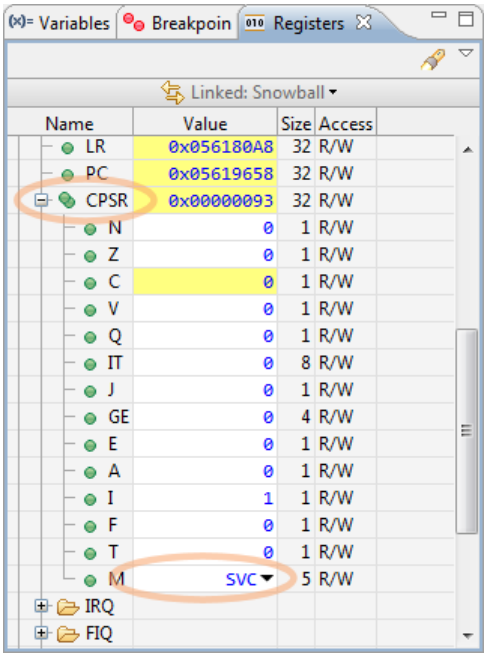
Notice that the corresponding line of source code becomes highlighted in blue the source view. Also a cross-section marker is shown in the corresponding point in the **Navigation** pane timeline.

⇨ Click and drag up and down in the **Trace** pane. Notice how both the navigation cross-section marker and blue highlighting in the source view follow along. You can also move the selection around the **Trace** pane with the up- and down-arrow, page-up and -down, home and end keys.

⇨ Click and drag right and left in the **Navigation** pane. Notice again how both the navigation cross-section marker and blue highlighting in the **Trace** pane and source view follow along. You can also move the cross-section marker around the **Navigation** pane with the right- and left-arrow keys.

You can use the toggle button (;) to change from showing both functions and instructions to showing functions only

⇨ Double click breakpoint indicator ( ) in the left margin of `hush.c` to delete the breakpoint. You can also delete the breakpoint by selecting it in the **Breakpoints** view and typing the delete key or using the Delete button ( ).



Registers view

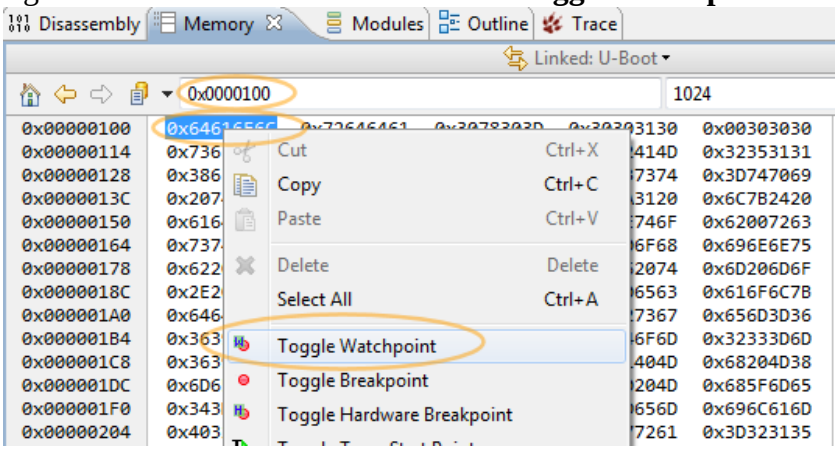
When doing bare-metal or kernel debugging via DSTREAM, you can use the **Registers** view to access all of the registers of the ARM processor, including other modes (**IRQ**, **FIQ**, ...) and the system control coprocessor (**CP15**). For example, you can find out what the current mode is by expanding the **CPSR** register in the **Core** folder and looking at the **M** (mode) field. We can see that U-Boot is using **SVC** (Supervisor) mode. You could also change the processor mode using the drop-down menu, but that's not a good idea.

Watchpoints

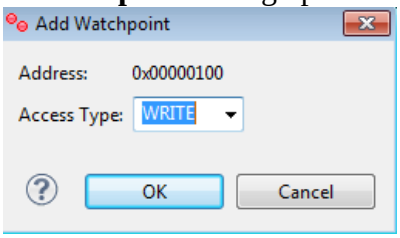
A watchpoint stops the target when a particular memory location is read or written. (Another name for them is "data breakpoints".) Before U-Boot starts the kernel it writes the kernel parameters into RAM at address `0x00000100` as we'll see later. Let's place a watchpoint on that address so that we can find the code that does the writing.

⇨ Bring the **Memory** view to the front and type `0x00000100` in the Add7ress field.

⇨ Right-click on the first word and choose **Toggle Watchpoint** from the context menu.



The **Add Watchpoint** dialog opens:



⇨ Choose **WRITE** or **ACCESS** as the **Access Type** and click **OK**.

You can now see the watchpoint in the **Memory** view and the **Breakpoints** view.

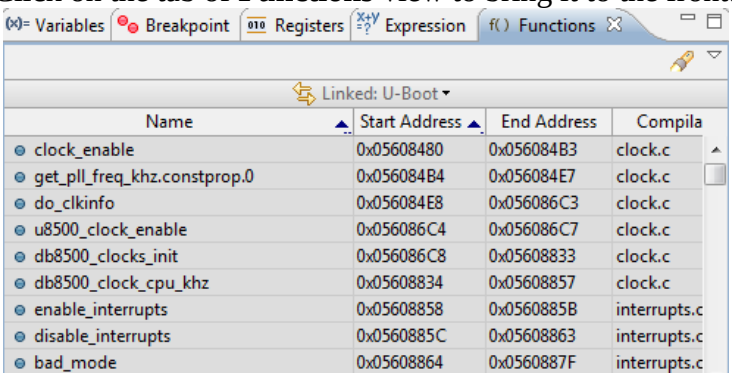
⇨ Click the Continue button ( ).

U-Boot tries to process the command you just typed and prompts for a new one. The watchpoint will trigger later when it writes the kernel parameters in to RAM.

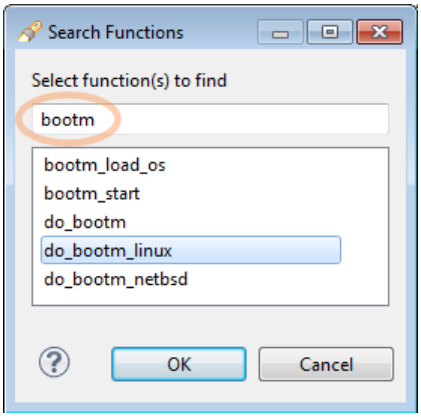
Functions view

Now we'll arrange to stop after U-Boot has loaded the linux kernel and is about to transfer control to it in the function **`do_bootm_linux`**.

⇨ Click on the tab of **Functions** view to bring it to the front:



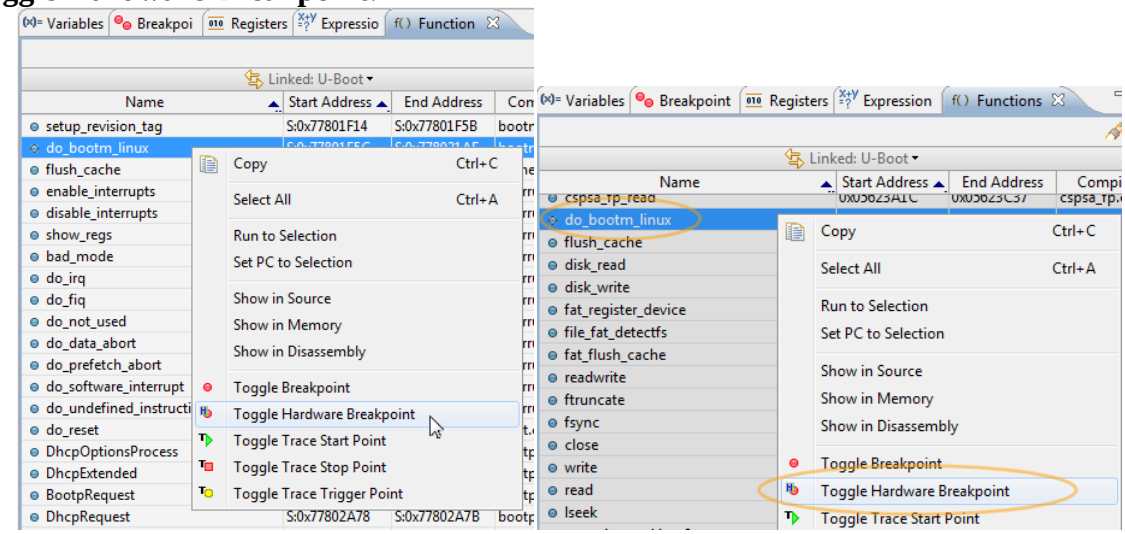
The **Functions** view shows information about all of the functions in the debug symbols we have loaded including the start and end addresses. You can change the sorting of the **Functions** view by clicking on the column headings. You can use the **Filters...** command in the view's drop-down menu ( ) to control which images and compilation units are shown.



⇨ Click on the Search button ( ) of the **Functions** view to open the **Search Functions** dialog then type some of the function's name, select **`do_bootm_linux`** and click **OK**:

The entry for **`do_bootm_linux`** is selected. We can set a hardware breakpoint on it even while the target is running.

⇨ Right-click on the entry for **`do_bootm_linux`** and choose **Toggle Hardware Breakpoint**:



Notice that the entry for **`do_bootm_linux`** now shows that there is a hardware breakpoint set ( ). The breakpoint is also shown in the **Breakpoints** view.

There are also Search buttons ( ) in the **Variables**, **Expressions**, **Registers**, **Disassembly** and **Memory** views.

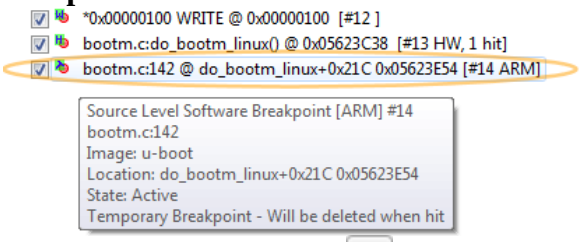
⇨ Type the command `boot` followed by the return key into the **Serial** view to tell U-Boot to load and boot the Linux kernel.

The target will stop at the beginning of `do_bootm_linux`. If you didn't delete the breakpoint in `parse_outer`, the target will stop there first and you should press the Continue button to get to `do_bootm_linux`. By the time U-Boot gets to `do_bootm_linux` it will have already loaded the kernel into RAM at some physical address and written some details to the console. All addresses are physical because U-Boot hasn't turned on the MMU.

Line 142 of of `bootm.c` is where U-Boot transfers control to the kernel.

⇨ Click on line 142 to move the selection there; then right-click on the line and choose **Run to Selection** from the context menu. Be sure to move the selection to line 142 first.

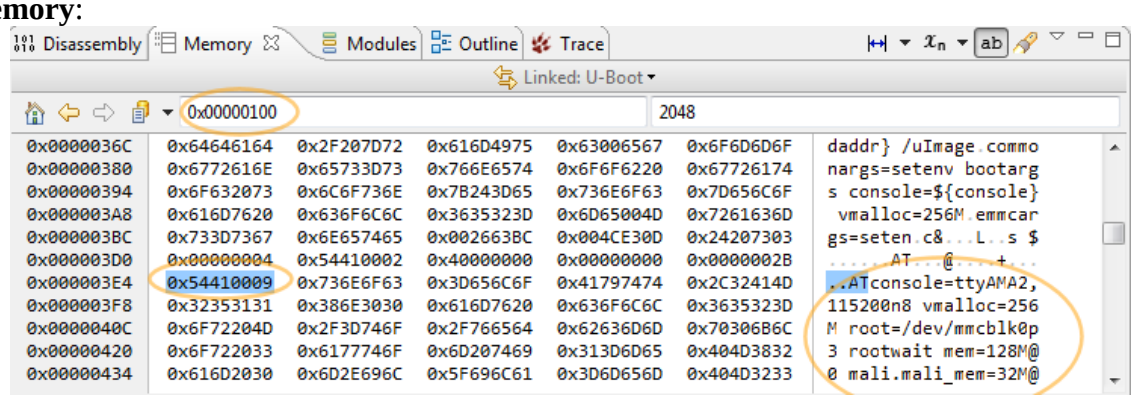
We don't reach, line 142 though. It turns out that our watchpoint triggers first in the function `setup_start_tag` (which has been inlined into `do_bootm_linux`) and it constructing the kernel parameters. The temporary breakpoint on line 142 that was created by **Run To Selection** is still present (and can be seen in the **Breakpoints** view. Get information about a breakpoint by hovering the cursor over it.)



⇨ Click the Continue button ( ) and the target will reach line 142.

From the source code we can see that U-Boot is going to pass three parameters to the kernel: zero (in `R0`), the machine ID, `machid`, (in `R1`) and a pointer to the kernel parameters, `bd->bi_boot_params` (in `R2`). You can see the value of `machid` by hovering the mouse over it. You can also see the value of the variables in the **Variables** view. The last parameter is a pointer to the kernel parameters in memory which contains the kernel command line (bootargs) and other information.

⇨ Select `bd->bi_boot_params` in the source then right-click on it and choose **Show Dereference in Memory**:



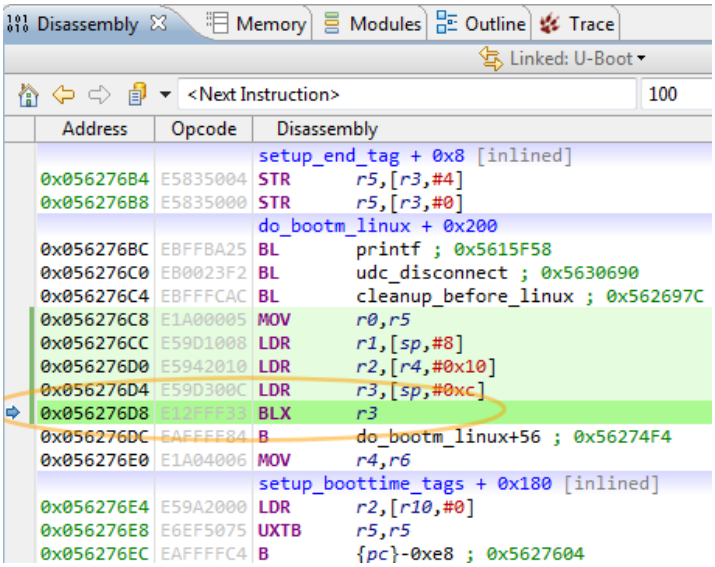
In the **Memory** view (you may need to scroll down) we can see the kernel command line string in the middle of the kernel parameters preceded by the tag value `0x54410009`: `console=ttyAMA2,115200n8 vmalloc=256M root=/dev/mmcblk0p3 rootwait`.

⇨ Click on the tab of the **Disassembly** view to bring it to the front.

If you scroll the **Disassembly** view up a few lines you can see that the function `setup_end_tag` has been inlined into `do_bootm_linux`. This is indicated by the lable `setup_end_tag +0x10 [inlined]` and the green color of the addresses. Now we're going to step carefully by assembly instructions.

⇨ Click the stepping by toggle button in the **Debug Control** view so that it is in stepping-by-instructions mode with the '**i**' dark ( ).

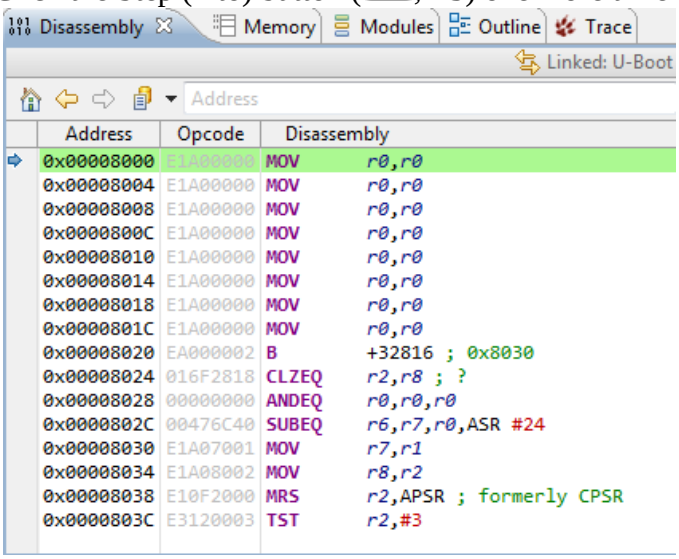
⇨ Click the Step (into) button (; **F5**) three times until the PC arrow is on the **`BLX R3`** instruction that transfers execution to the kernel:



⇨ Click on the tab of the **Registers** view to bring it to the front.

See that **`R0`** contains zero, **`R1`** contains the machine ID and **`R2`** is a pointer to the kernel parameters and **`R3`** is a pointer to the kernel code (as a physical address). Although the compiler doesn't know it, the kernel will never return to **`do_bootm_linux`**.

⇨ Click the Step (into) button (; **F5**) one more time to step to the kernel:



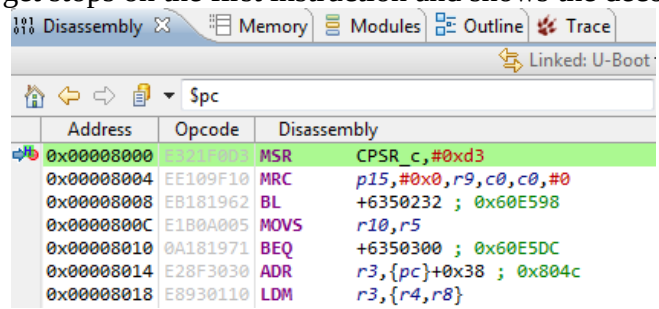
The kernel image that U-Boot loads into RAM is known as **`uImage`**. It has a 64 byte header created by the U-Boot tool **`mkimage`** and is followed by a compressed version of the kernel wrapped by a self-decompressor which is the code we see here and is known as a **`zImage`**. The **`zImage`** works by decompressing the kernel code in-place and then jumping back to the beginning again. We'll set a **hardware** breakpoint on the first instruction and continue the target to let the decompression happen.

⇨ Click the Step (into) button (; **F5**) once to step to the next instruction.

⇨ Right-click on the first instruction of the compressed kernel (**`0x00008000`**) and choose **Toggle Hardware Breakpoint**. Using a regular software breakpoint won't work since the decompressor is going to overwrite this instruction.

⇨ Click the Continue button ( ).

The target stops on the first instruction and shows the decompressed kernel code:



Since we're done debugging U-Boot we will delete any breakpoints and watchpoints we still have and unload its debug symbols.

- Click the Delete All button ( ) in the **Breakpoints** view.
- Click the stepping by toggle button in the **Debug Control** view so that it is back in stepping-by-source-line mode with the '**s**' dark ( ).
- Close the various U-Boot source views, if you want to get them out of your way.

Now we're ready to start debugging the kernel.

Kernel and module debug

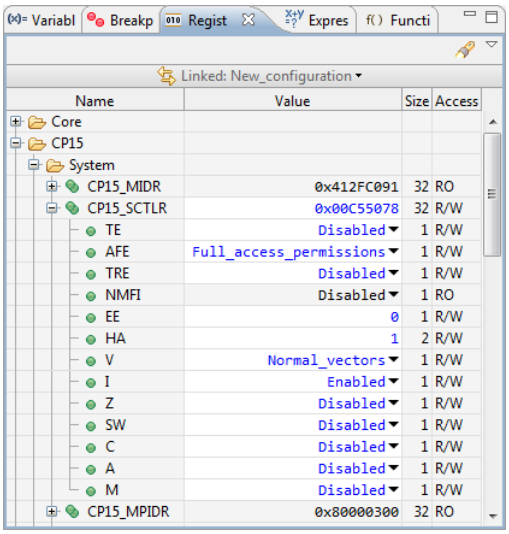
When U-Boot transfers execution to the kernel the MMU is still off and all addresses are physical addresses. Building the MMU tables and turning on the MMU are some of the first things that the kernel does, but it can still be useful to debug the kernel before the MMU is on.

Setting up your own target: Download the Linux kernel symbols and sources

If you are running this workshop on your own host you will need to do some setup to debug the Linux kernel. You can download the kernel sources and symbols using the Ubuntu package manager on the target and then build the kernel module example on the target. Once built, the sources and symbols can be copied to your host and used during the debug session. To complete your own setup please follow the instructions in the appendix *Linux kernel download and debug setup* in the appendix on page 95; and *Kernel module debug (modex) build and setup* in the appendix on page 97.

Debugging the kernel before the MMU is on

There is no requirement to debug the kernel before the MMU is on unless you're trying to find a bug there or are an exceptionally curious person. I'm guessing you're exceptionally curious so we'll debug it just a bit. Similarly, if we wanted to get to this point without bothering to debug U-Boot first we could just set a hardware breakpoint at `0x00008000` and let U-Boot run until we reached it twice ("twice" because of the decompression).

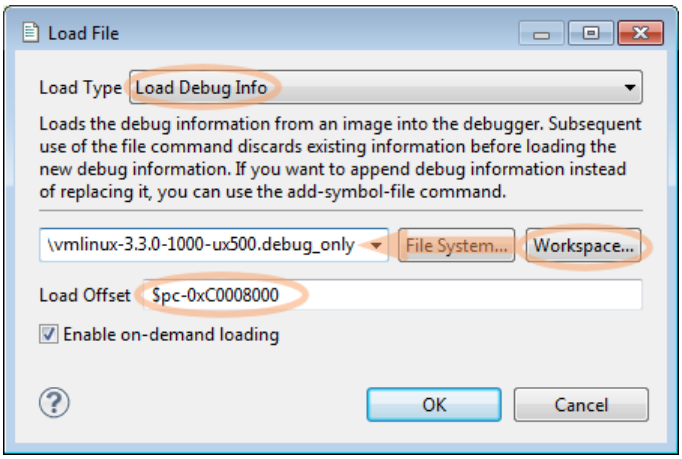


⇨ Bring the **Registers** view to the front; expand the **CP15** and **System** folders and the **CP15_SCTRL** (System Control) register.

You can see that that **CP15_SCTRL** register has the **M** bit clear ("Disabled") so the MMU is currently disabled.

Debugging the kernel before it turns on the MMU is very similar to debugging U-Boot, but because the kernel debug symbols in the `vmlinux` file are the virtual addresses we need to load them with an offset so that they are valid with the MMU off: The virtual address of the first instruction is `0xC0008000` and the physical address is `$pc` so the offset needed is `$pc-0xC0008000` (or `0x00008000-0xC0008000` if the PC is somewhere else).

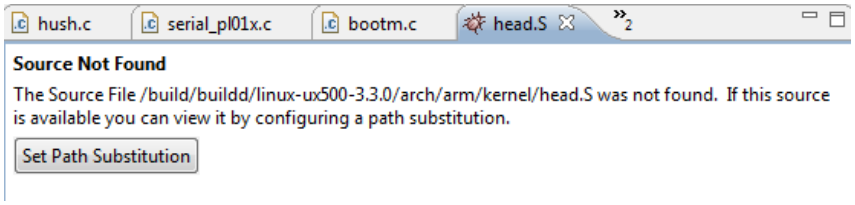
⇨ Choose **Load...** from the **Debug Control** view's drop-down menu ( ) and then choose **Load Debug Info**; click the **Workspace...** button then select `kernel\vmlinux-3.3.0-1000-ux500.debug_only`, which has the kernel debug symbols; put `$pc-0xC0008000` in the **Load Offset** field. If you've stopped the target at a different location than the first instruction of the kernel you'll need to use `0x00008000` instead of `$pc`. Click the **OK** button:



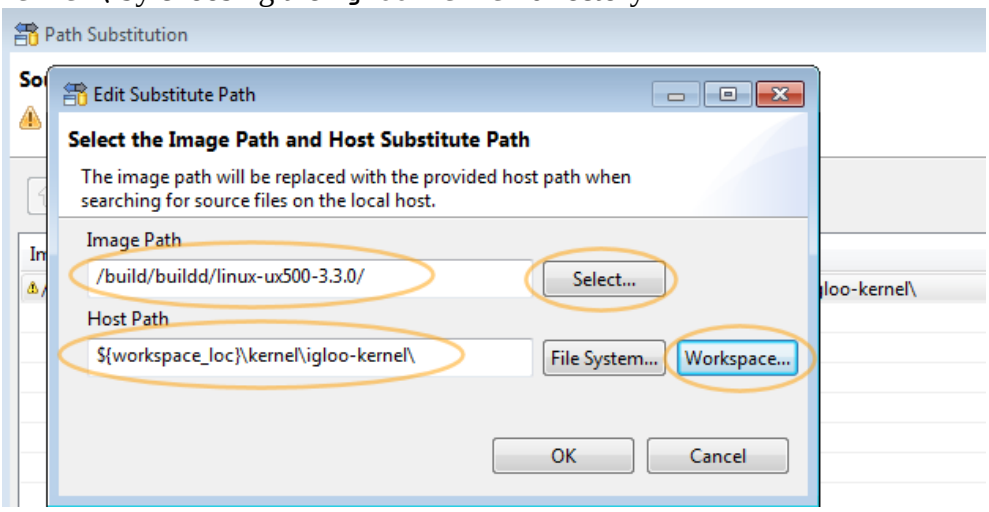
The **Disassembly** and source views update to show the kernel symbols and sources. Most of the early kernel source is written in assembly code.

Because we chose **Load Debug Info** the U-Boot debug symbols that we had loaded before were discarded. The command that appears in the **Commands** view is `file` instead of `add-symbol-file`. We could easily copy that command to a script if we needed to do it often.

You may need to set a substitute path if the source code for the kernel is not resolved. For example you should see



- ⇨ Click the **Set Path Substitution** button in the source view
- ⇨ Select the **Image Path** as **`/build/buildd/linux-ux500-3.3.0/`**
- ⇨ Use the **Workspace…** button to select the **Host Path** as **`${workspace_loc}\kernel\igloo-kernel\`** by choosing the **`igloo-kernel`** directory



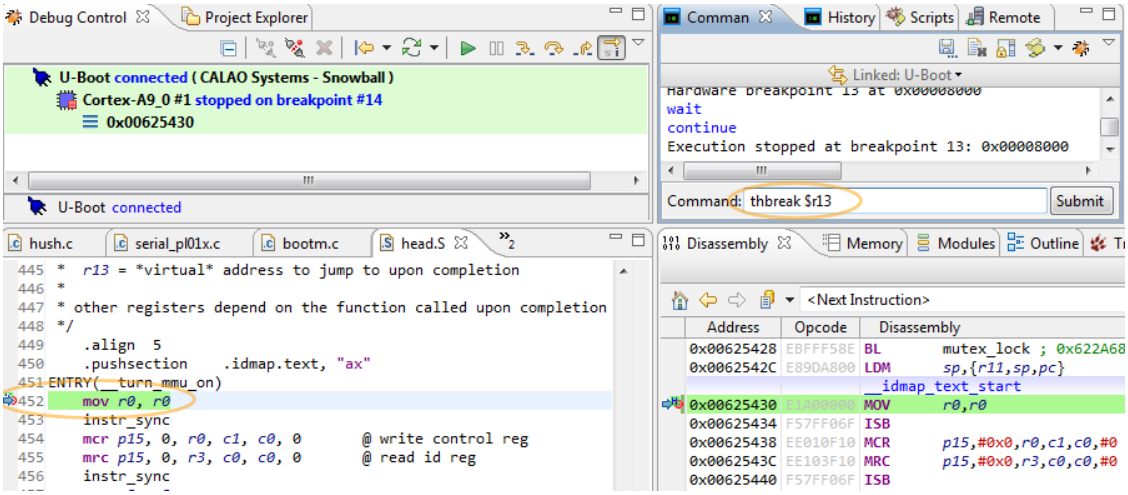
You can now step and do the normal debugging things to see what the kernel is doing before the MMU gets turned on. We'll just set a breakpoint on the function **`__turn_mmu_on`** which is where the kernel, well, turns the MMU on. In the **Functions** view, the address of **`__turn_mmu_on`** should be **`0x00625430`** If the value is different then you may have mistyped the **Load Offset**. Another reason that the value could be different is if you're using a different version or configuration of the kernel. If the address is not **`0x00625430`** then execute the command **`file`** in the **Commands** view to discard the debug symbols and then reload them.

- ⇨ Use the **Functions** view to put a hardware breakpoint on **`__turn_mmu_on`** or, for variety, use the command **`hbreak __turn_mmu_on`** in the **Commands** view.
- ⇨ Click the **Continue** button to run to it ( ).

When we get to **`__turn_mmu_on`** the MMU is still off, but register **`R13`** has the virtual address of the first instruction executed after the MMU is turned on (usually it holds the stack pointer). If you want, you can look in the **Trace** view to see what has been executing recently. You may notice that the symbol displayed in the **Dissassembly** view is not **`__turn_mmu_on`**. The reason is that at this location there are two symbols, and the debugger is displaying the first (from line 450).

[If, instead of reaching **`__turn_mmu_on`**, the debugger stops with the **`PC`** near address **`0xFFFF0000`** (that is, in the exception vectors) then the breakpoint has been missed and the kernel has started executing and the debugger has stopped the target on a Data Abort or other exception. The **Serial** view will show the kernel message output. You'll need to disconnect, reset the target, stop U-Boot from launching the kernel by typing to it and reconnect the U-Boot debug configuration.]

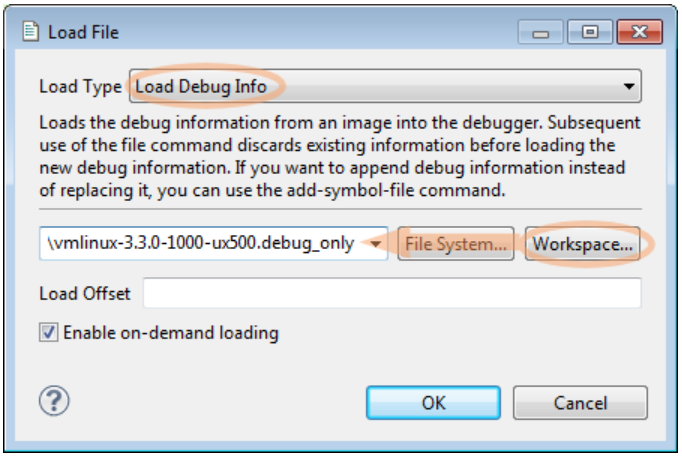
- ⇨ Delete the breakpoint that is on **`__turn_mmu_on`** because when the MMU is turned on it will be in the wrong place and will only be able to cause problems.
- ⇨ Place a temporary hardware breakpoint at the address that **`R13`** is pointing at, by typing the command **`thbreak *$r13`** in the **Command** field of the **Commands** view and pressing return. (Actually the **`*`** is optional.)



⇨ Click the Continue button ( ).

When the target stops the MMU is on but the symbols we loaded with an offset are now wrong. The breakpoint gets deleted when it is hit because it's a temporary breakpoint. You can see that the **M** bit in **CP15_SCTRL** in the **Registers** view is now set ("Enabled").

⇨ Reload the kernel symbols without an offset. Choose **Load...** from the **Debug Control** view's drop-down menu ( ) and then choose **Load Debug Info**; click the **Workspace...** button then select **`kernel\vmlinux-3.3.0-1000-ux500.debug_only`**, which has the kernel debug symbols; don't put any value in the **Load Offset** field and then click the **OK** button.



⇨ Use the **Functions** view to put a breakpoint on **`start_kernel`** or, for variety, use the command **`break start_kernel on`** in the **Commands** view.

⇨ Click the Continue button ( ).

The **`start_kernel`** function initialises the remainder of the Linux kernel after the MMU is turned on. The majority of the kernel components are initialised in this function. As the memory map for the Linux kernel is now initialised we can enable the DS-5 Linux kernel operating support to aid our debug session.

Debugging the kernel initialisation after the MMU is on

So far we've been debugging using the **U-Boot** bare-metal debug configuration that we created. This works for the early kernel because the kernel itself is essentially a bare-metal application; that is, the kernel does not rely on any lower-level operating system.

Now it's time to examine or create a debug configuration for the kernel when the MMU is on.

⇨ With the target still stopped, click the **Disconnect From Target** button ( ) in the **Debugger Control** view to disconnect the **U-Boot** debug configuration.

⇨ Choose **Run > Debug Configurations...** then expand **DS-5 Debugger**.

The **Kernel** debug connection may already have been created for you in which case you can just select it, examine the settings and choose the DSTREAM unit.

⇨ If there is no **Kernel** debug connection as a child of **DS-5 Debugger**, then select **DS-5 Debugger** and click the **New launch configuration** button ( ) to create a new debug configuration. (You can also double-click **DS-5 Debugger** or right-click it and choose **New** instead.)

⇨ Give the debug configuration a name, I used ***Kernel***.

In the **Connection** pane of the debug configuration we need to specify the platform and choose the DSTREAM.

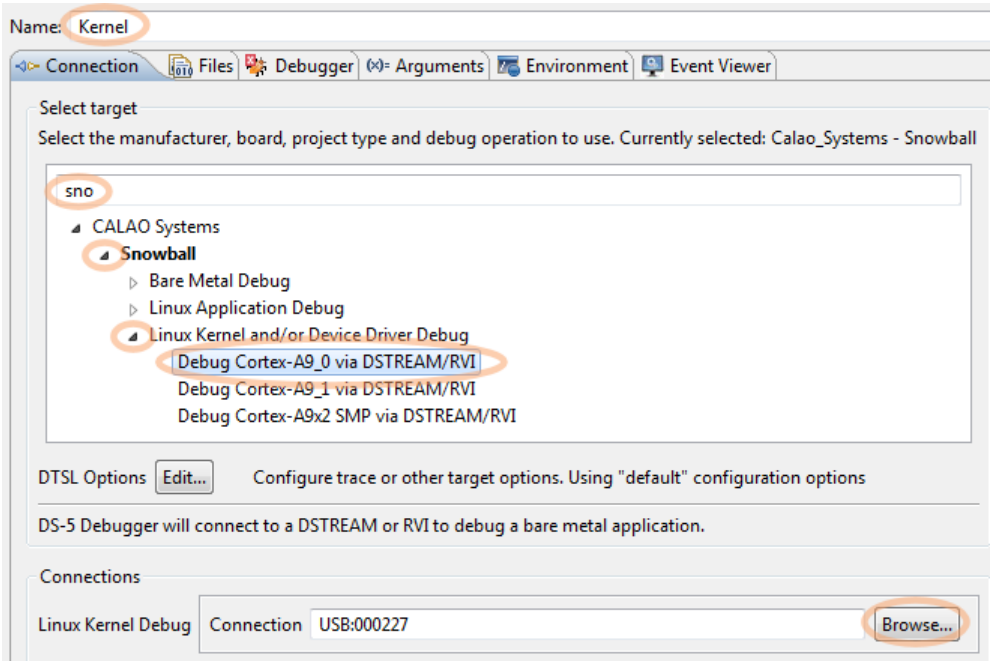
⇨ Type ***sno*** in the Filter platforms filter box so that the only the matching platforms are shown.

⇨ Expand **CALAO Systems > Snowball > Linux Kernel and/or Device Driver Debug** in the platforms list.

⇨ Select **Debug Cortex-A9_0 via DSTREAM/RVI**.

We choose kernel-only trace because decoding the trace information requires reading the instructions from the target which is not always possible for a Linux application that isn't the current process.

⇨ Click the **Browse** button and choose the DSTREAM unit and click **OK**. Your Connection number will be different than shown.



We'll leave the **Files** pane of the debug configuration blank so that we can be sure that the target is stopped when we load the debug symbols. We'll load them in a script in the **Debugger** pane. The kernel was (and can only be) built on a Linux host. As we are using a Ubuntu image we can use the Linux package manager to get the symbols for the Linux kernel.

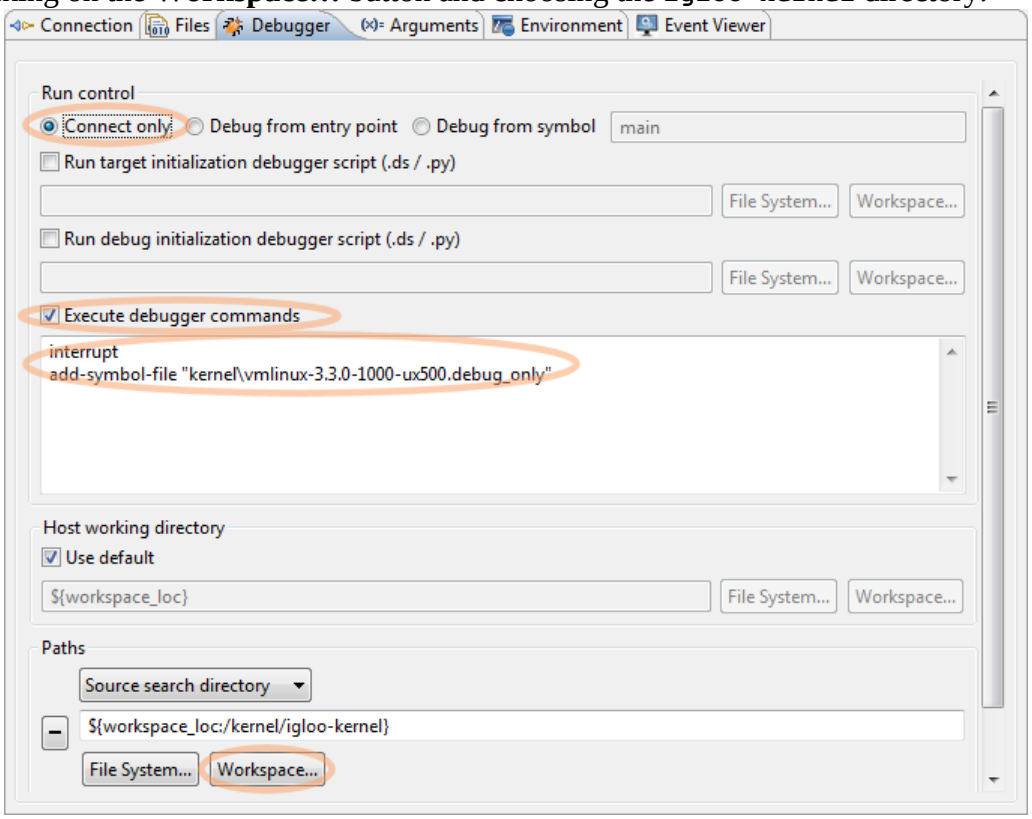
In the **Debugger** pane of the debug configuration:

⇨ Choose **Connect only** so that DS-5 will just attach to the target which is already running the kernel.

⇨ Check **Execute debugger commands** and type these commands into the field:

```
interrupt
add-symbol-file "kernel\vmlinux-3.3.0-1000-ux500.debug_only"
```

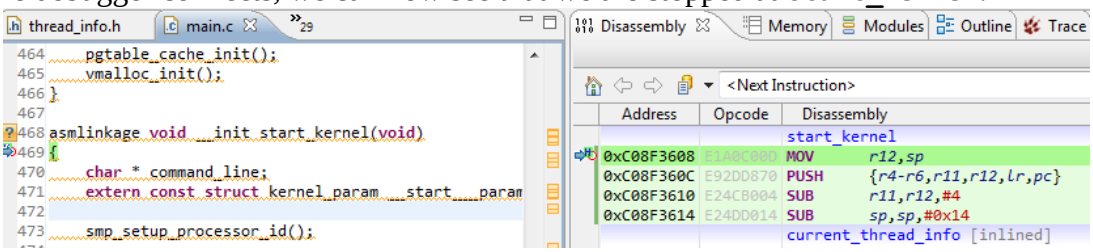
⇨ In the **Paths** list select **Source search directory** as **`${workspace_loc:/kernel/igloo-kernel}`** by clicking on the **Workspace…** button and choosing the **`igloo-kernel`** directory:



The **Arguments** and **Environment** panes don't apply to kernel debugging either, so just leave them blank. And again, we won't be using the **Event Viewer** pane, so just leave it blank, too.

⇨ Click the **Debug** button. If the **Debug** button is disabled then Eclipse will put a message explaining why at the top of the **Debug Configurations** dialog.

When the debugger connects, we can now see that we are stopped at **`start_kernel`**:



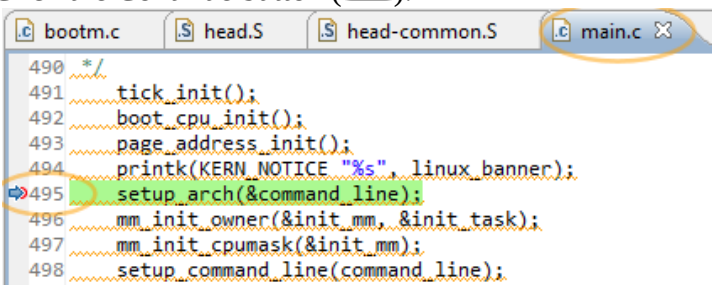
This kernel debug configuration will behave in a similar way to the bare-metal debug configuration that we have been using so far but will also allow us to see the processes and threads after we've loaded the kernel debug symbols. Another difference is that, by default, when doing bare metal debug DS-5 intercepts processor exceptions like Data Abort and in kernel debug it lets the target handle them. The interception behaviour can be changed by using the **Manage Signals** command in the **Breakpoints** view's drop-down menu ( ).

The extra orange underlining in the kernel source views is due to the C/C++ indexer not fully understanding the include paths and macros used to build the kernel. You can disable it in **Window > Preferences > General > Annotations > C/C++ Indexer Markers**.

There are no processes yet, but we can step and debug the rest of the kernel with the MMU on. In the **Breakpoints** view, you can also see that DS-5 has set some debugger internal breakpoints so that it can track the loading and unloading of kernel modules. Next let's debug some of the kernel initialisation.

⇨ Double click on side bar next to line 495 in **`main.c`** to set a breakpoint

⇨ Click the Continue button ( ).

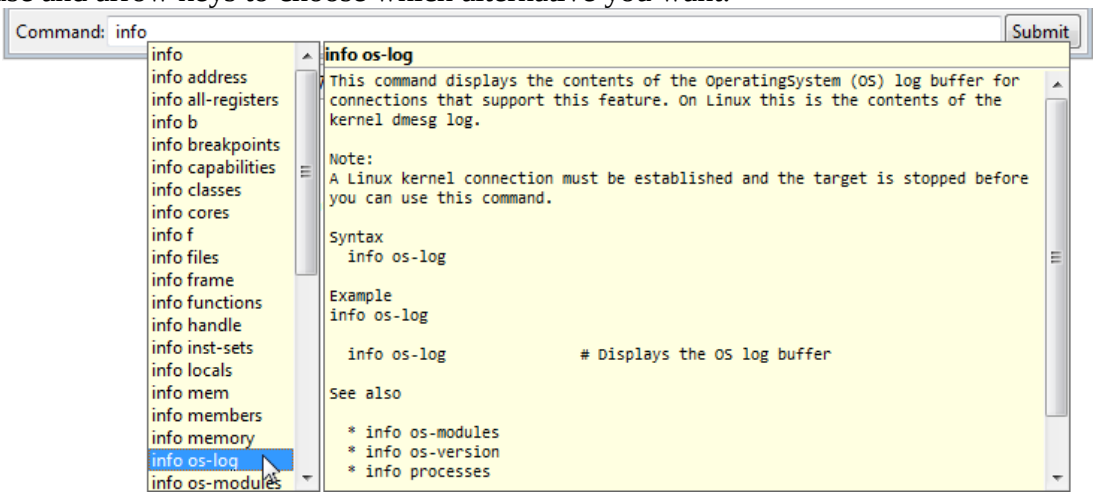


On line 494 a kernel **`printk`** function was executed. These kernel messages are normally directed to the serial port to aid with the debug of kernel start-up. As we are still at a very early stage in the kernel boot process the serial port driver for the kernel is not initialised yet. For that reason the message is not displayed over the serial port connection yet.

You can use the **`info`** command to get information about the kernel even if the target doesn't have a serial console. You can use Content Assist (Ctrl+Space) to get help on commands as you are typing them.

INFO: In Windows, if foreign language support is enabled, the Ctrl+Space key combination is used to change between languages inside a text box (for example English to Chinese). You can change the key combination to something else in Eclipse by going to **Window > Preferences > General > Keys > Content Assist** to change the key combination to something else.

⇨ Type **`info`** into the **Command** field of the **Commands** view and then type Ctrl+Space. This activates the Content Assist which shows the possible completions and help for each. You can use the mouse and arrow keys to choose which alternative you want:



⇨ Try the **`info os-version`** and **`info os-log`** commands. The **`info os-log`** command shows the kernel message buffer. (**`printk`**; **`dmesg`**);

⇨ Click the **Continue** button ( ) and let the target run.

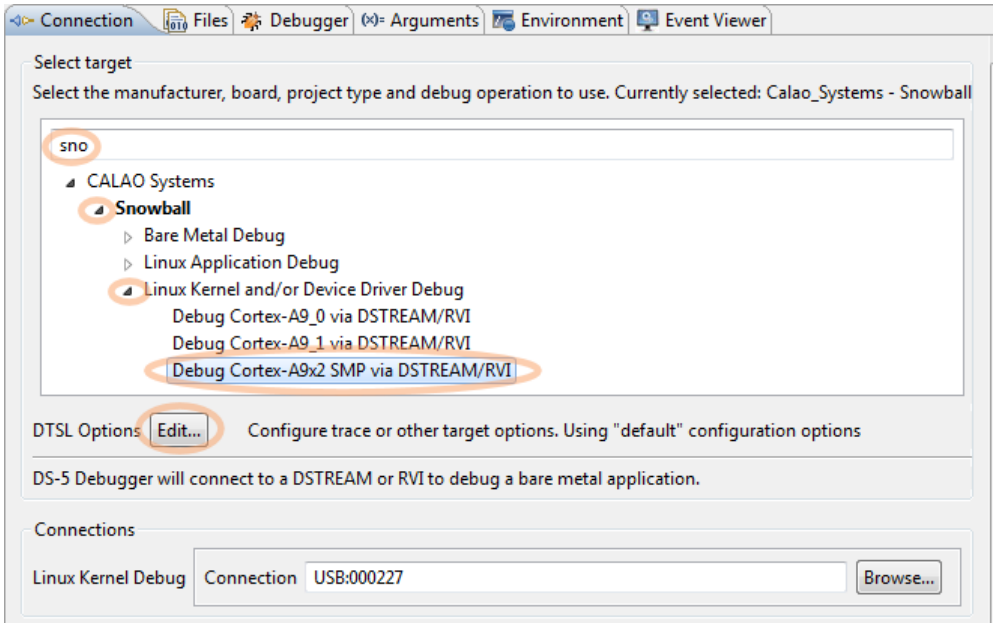
One of the next initialisation sequences of the Linux kernel is to boot the secondary processors and assign kernel threads to the cores. We will disconnect and then reconnect DS-5 with an SMP connection:

⇨ Click the Disconnect From Target button ( ) in the **Debug Control** view to disconnect from the running target.

⇨ Right-click on the **Kernel** debug configuration in the **Debug Control** view and choose **Debug Configurations...** to edit the configuration.

⇨ Type ***sno*** in the Filter platforms filter box so that the only the matching platforms are shown.

⇨ Change from **Debug Cortex-A9_0 via DSTREAM/RVI** to **Debug Cortex-A9x2 SMP via DSTREAM/RVI**.

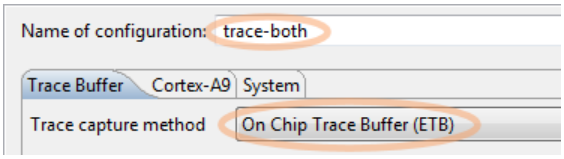


We will setup the tracing options in a DTSL configuration.

⇨ Click the **DTSL Options Edit…** button to open the **DTSL Configuration Editor** dialog box.

⇨ Click the Add button (+) to create a new DTSL configuration.

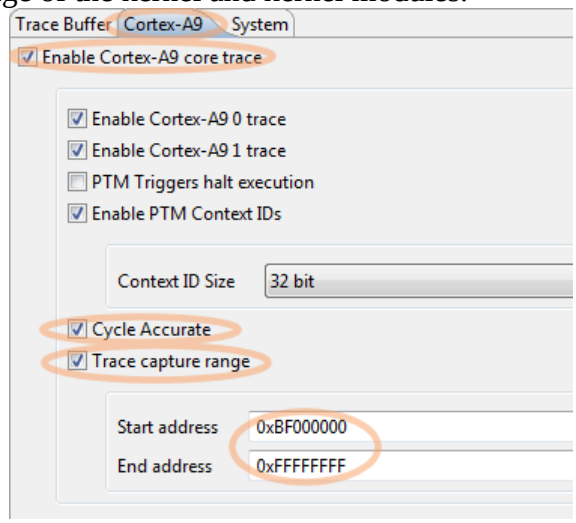
⇨ Give the new DTSL configuration a name, I used ***trace-both***.



⇨ In the **Trace Buffer** pane, change the **Trace capture method** to **On Chip Trace Buffer (ETB)**. The Snowball doesn't have a connector for external (TPIU) trace

⇨ In the **Cortex-A9** pane, check **Enable Cortex-A9 core trace** and **Cycle Accurate** so that the trace data will include cycle count (although this will mean that fewer instructions fit into the ETB).

⇨ Check **Trace capture range** and set the range to **`0xBF000000-0xFFFFFFFF`** which is the address range of the kernel and kernel modules.

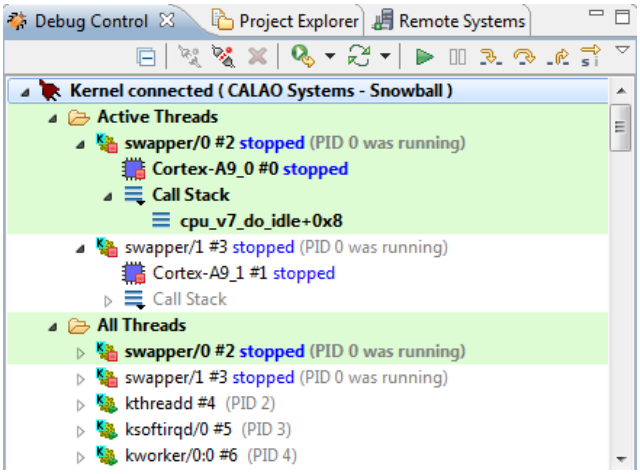


⇨ Click **OK** in the **DTSL Configuration Editor** dialog box.

⇨ Click the **Debug** button to reconnect to both processors on the target.

In the **Debug Control** view we can now also see the processes and threads instead of just the cores that we saw when debugging the bare-metal U-Boot.

⇨ Expand the **All Threads** folder in the **Debug Control** view to see all the processes and threads:



You can expand each thread to see the thread's stack. You can change the debugger's focus from one thread to another by clicking on the different threads and stack frames. You can change the way child threads are displayed by choosing **Flat** or **Hierarchical** from the **Thread Presentation** submenu of the **Debug Control** view's drop-down menu ( ). From the drop-down menu you can also select **Always Show Cores** to display the cores ( ) like in bare-metal debugging as well.

Kernel space threads are shown with a ( ) icon and user space threads are shown with a ( ) icon.

Reading the thread information for all threads takes some time, so it's a good idea to leave the **All Threads** folder collapsed when you don't need it.

- ⇨ Collapse the **All Threads** folder to hide the non-active threads.
- ⇨ Click the Continue button ( ) to let the target run so that the kernel can finish booting.

For the next part of the workshop you will need to build the DS-5 kernel module example against the kernel on your target and copy it back to the host. If this has not been done for you, please see in the appendix on page 97 about how to set this up.

When the kernel finished booting it will prompt you to login.

- ⇨ Type `ls` to the **Serial** view window to list the files in the current directory.

The file **`modex.ko`** is a kernel module that we can debug. Kernel modules are like shared libraries for the kernel. First, we will load its debug symbols:

- ⇨ Choose **Load...** from the **Debug Control** view's drop-down menu ( ); leave **Add Symbol File** selected; click the **Workspace...** button and then choose **`kernel_module\modex.ko`**, which has debug symbols; click the **Open** button and then click the **OK** button.

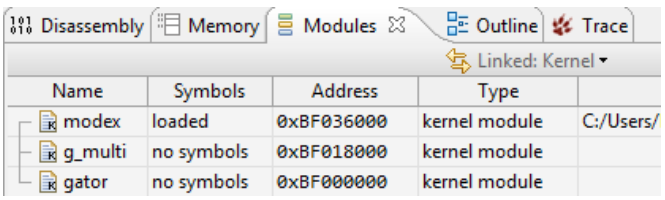
We use **Add Symbols File** instead of **Load Debug Info**, because **Load Debug Info** would discard the currently load debug symbols (from **`vmlinux`**). There will be a warning in the **Commands** view that **`modex.ko`** is not loaded yet. That's ok, we will load it soon.

- ⇨ Look in the **Modules** view.

We can see that **`modex`** is not in the list.

- ⇨ Type `insmod /home/linaro/kernel_module/modex.ko` to the **Serial** view to load (insert) the module.

There will be a message in the Commands view that the symbols have been loaded and **`modex`** now appears in the **Modules** view.



⇨ Click the Interrupt button ( ) to stop the target.

⇨ Use the **Commands** view to set a breakpoint by executing the command `break modex_write`.

⇨ Click the Continue button ( ) and let the target run.

⇨ Type `echo A > /dev/modex` to the **Serial** view. The `A` can be any character you want.

The kernel will call the `modex_write` function which will process the input after hitting the breakpoint. You can now step and debug the kernel module.

⇨ Click on line 84, with the call to `printk`, to move the selection there; then right-click on the line and choose **Run to Selection** from the context menu. Be sure to move the selection to line 84 first.

⇨ Look in the **Variables** view to see that the value of `x` holds the first character that you echoed.

⇨ Click the Step Over button (; **F6**) to execute the call to `printk`.

⇨ See that the `printk` message appears in the kernel messages, type `info os-log` in the **Commands** view. If the kernel messages are still being directed to the serial port then they will also appear in the **Serial** view.

⇨ When you're finished with your investigations, delete any breakpoints and then click the Continue button ( ) and let the target run.

⇨ With the target still running, click the **Disconnect From Target** button ( ) in the **Debugger Control** view to disconnect the **Kernel** debug configuration.

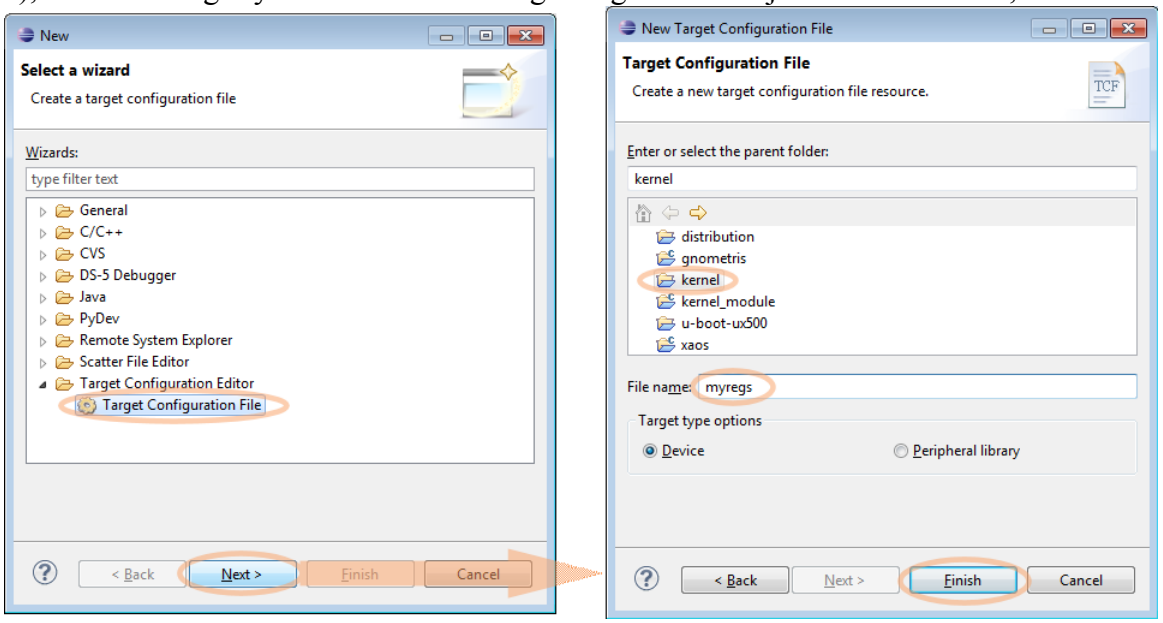
Peripheral Registers

The debugger can also display memory-mapped peripheral registers in the **Registers** view. DS-5 knows the peripheral registers for some platforms, but not for Snowball. If the peripheral register descriptions are available in a standard format (CMSIS-SVD, RVD BCD or Lauterbach PER) DS-5 can import/convert them to its format (`.tcf`) and use them. There are a collection of CMSIS-SVD files at http:://cmsis.arm.com. We don't have a file to import for Snowball, so will use DS-5's **Target Configuration Editor** to create a `.tcf` file with a few peripheral registers to show what kinds of things are possible.

Create a new file `myregs.tcf` in the `kernel` project.

⇨ Select **File > New > Other...** (Ctrl+**N**), expand **Target Configuration Editor** and select **Target Configuration File**; click **Next >**.

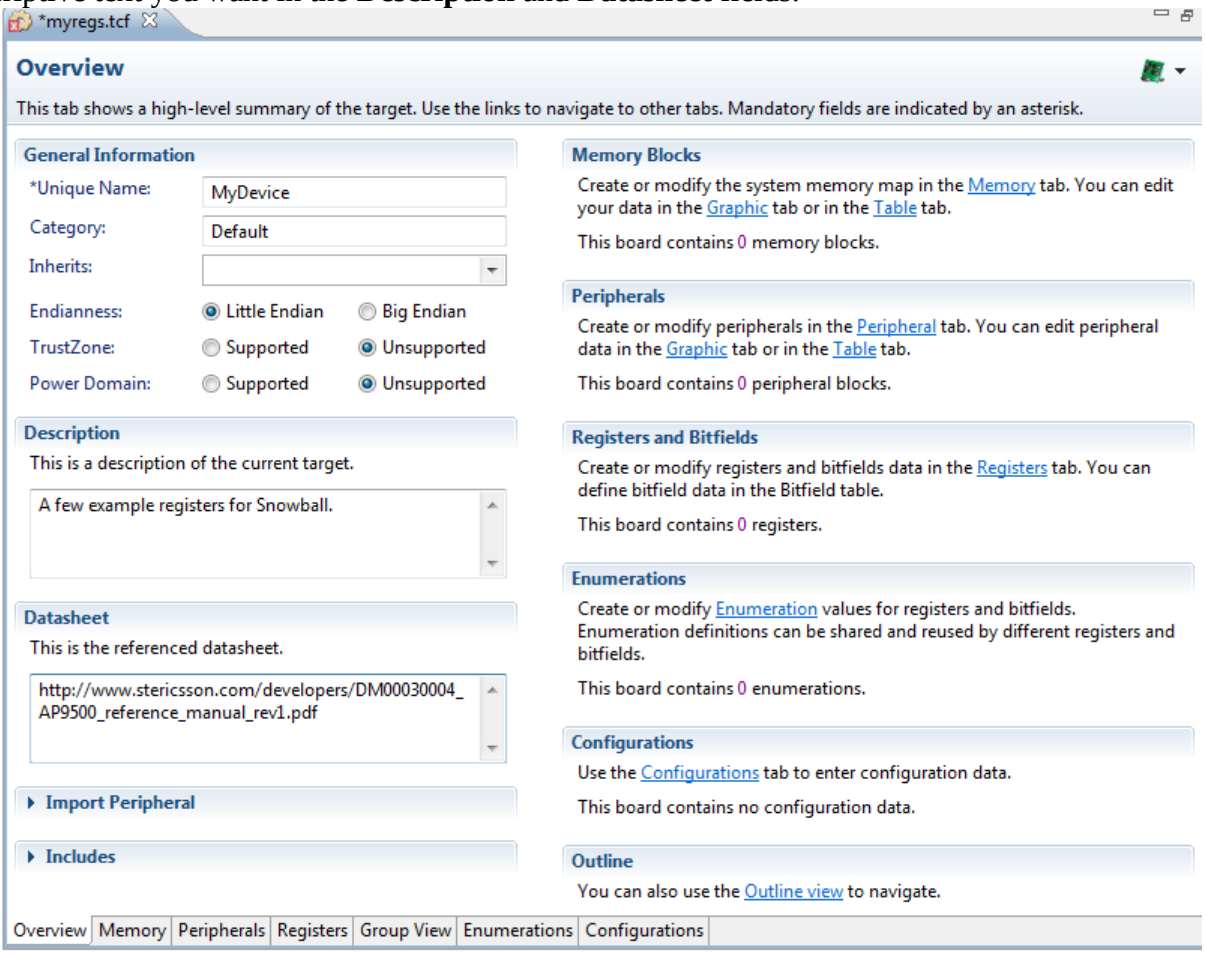
⇨ Select the `kernel` project; type the filename *myregs*. If we wanted to be able to reuse the registers in multiple targets we could choose to create a **Peripheral library** file (`.pcf`) instead of a **Device** file (`.tcf`), but that's slightly more work for a single target so we'll just use the default, **Device**.



⇨ Click **Finish**. The file is created and the **Target Configuration Editor** opens to the **Overview** pane.

⇨ Double click the **myregs.tcf** tab to zoom up the editor to fill the whole Eclipse window.

⇨ Fill in *MyDevice* for the **Unique Name** for the device (must be a legal C identifier) and type any descriptive text you want in the **Description** and **Datasheet** fields:



The **Target Configuration Editor** has a number of panes which are used to describe various aspects of the target: **Memory**, **Peripherals**, **Register**, **Enumerations**, **Group View** and **Configurations**. They are described on the right side of the **Overview** pane along with links. There are also tabs along the bottom of the view for switching panes. You can also navigate a `.tcf` file using the standard Eclipse **Outline** view.

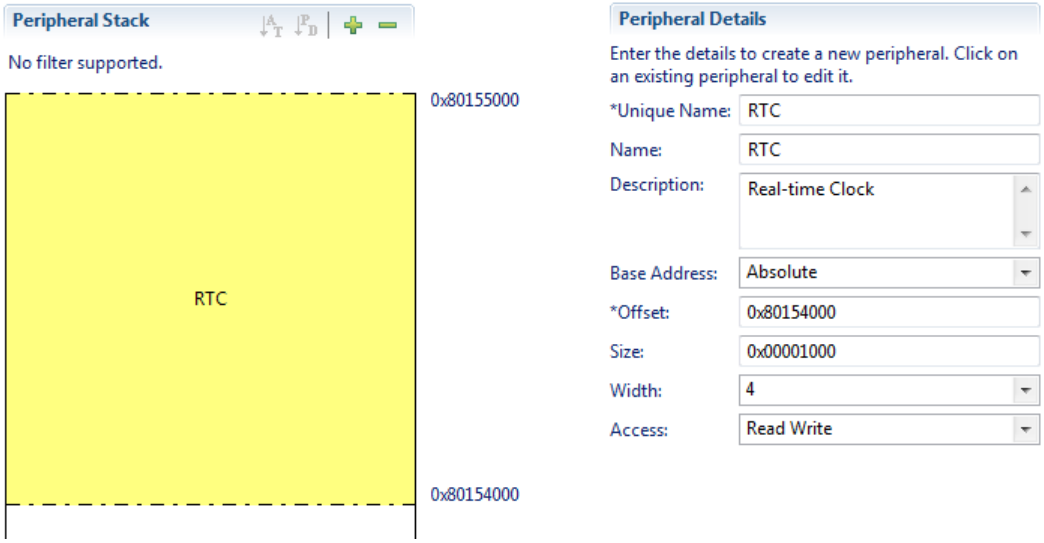
We don't need to create any memory regions so we'll skip the **Memory** pane. We'll start by creating a peripheral, the RTC (real time clock).

⇨ Click on the **Peripherals** tab or link to view the **Peripherals** pane.

⇨.Fill in the details for the peripheral:

Unique Name: *RTC*
Base Address: **Absolute** (use the drop-down menu)
Offset: `0x80154000`
Size: `0x1000`
Width: `4` (menu)

Access: Read Write (menu)



The **Offset** value is the physical address of the peripheral as discovered in the datasheet for the processor. The graphic view displays the peripheral and its address range.

The **Peripherals** pane (and the **Memory** pane) can be viewed either in the default graphic form or as a table. You can see them as table by clicking the Switch button ( ) at the top. The table view allows copying and pasting rows and columns much like a spreadsheet.

Now we'll add registers to the peripheral.

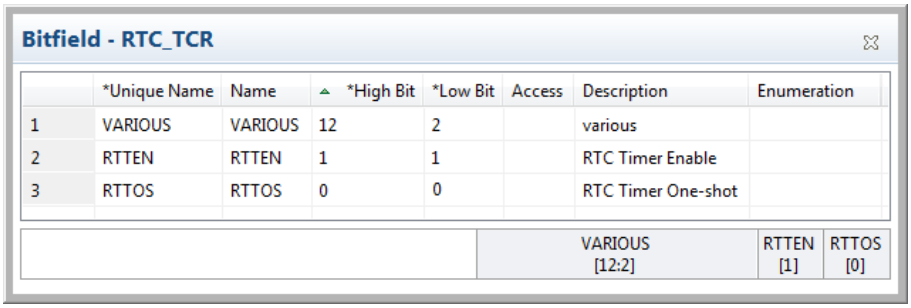
⇨ Click the **Registers** tab and add these three registers. You need to choose the peripheral in the **Peripheral** column before you can choose it for the **Base Address**. Don't fill in the **Access Size** and **Access** columns.

	*Unique Name	Name	Base Address	*Offset	*Size	A...	Access	Description	Peripheral
1	RTC_TDR	RTC_TDR	RTC	0x00000020	0x00000004			Timer Data	RTC
2	RTC_TLR1	RTC_TLR1	RTC	0x00000024	0x00000004			Timer Load	RTC
3	RTC_TCR	RTC_TCR	RTC	0x00000028	0x00000004			Timer Control	RTC

The **Offset** value is the offset from the base address of the peripheral.

This is already sufficient to be able to access these registers, but we'll go further and create handy bitfield descriptions for **RTC_TCR**.

⇨ Select **RTC_TCR**; click the Edit Bitfield... button ( ) or right-click and choose **Edit Bitfield...** from the context menu and enter these bitfields:



Notice the picture of the bitfields at the bottom of the bitfield editor.

Like the peripheral addresses, the bitfield definitions can be discovered from the processor datasheet. Bitfields can also have an Enumeration which assigns names to values, for example DISABLED=0, ENABLED=1. Since the interesting bitfield is just one bit wide, we'll leave creating an enumeration for it as an exercise to the interested reader. (Hint: create the enumeration then come back to the bitfield editor.)

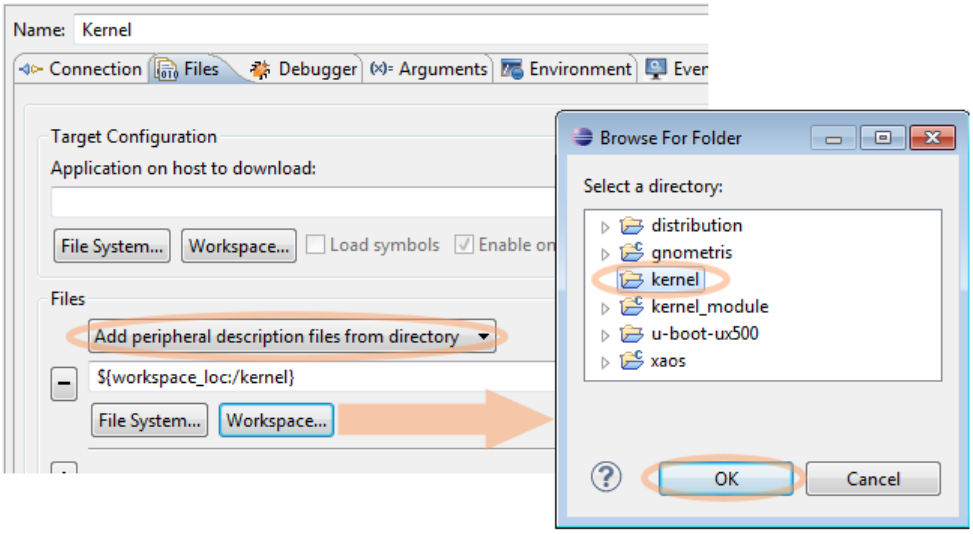
⇨ Close the bitfield editor ( ) and save the file by clicking the Save button (; Ctrl+S). You can double-click the **myregs.tcf** tab to zoom it back down. You can close the file if you want.

Now we've finished creating the `.tcf` file and we can add it to our debug configuration so that we can use it.

⇨ If there is a connected debug configuration then disconnect it ( ).

⇨ Right-click on the **Kernel** debug configuration in the **Debug Control** view and choose **Debug Configurations...**.

⇨ In the Files list in the **Files** pane change the popup to **Add peripheral description files from directory**; click **Workspace...** and choose the `kernel` project which is the parent directory of the `.tcf` file we created:



⇨ Click **Debug** to connect to the target.

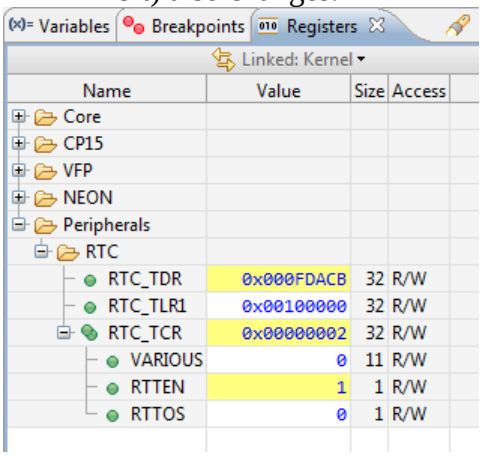
When the target is stopped, the **Registers** view now contains a **Peripherals** folder which contains the registers and bitfields we've just created.

⇨ Expand the **Peripherals** and **RTC** folders and the **RTC_TCR** register.

We can see the bitfields we created in the **RTC_TCR** register. You can hover the mouse over a register or bitfield to see its full name and any text that you put in the **Description** column in the editor. To use the timer in the RTC we need to first set the timer load register (**RTC_TLR1)** to some value and then set the timer enable bit (**RTTEN**) in the timer control register (**RTC_TCR**).

⇨ Set the **RTC_TLR1** register to `0x100000` (any reasonably large value will do).

⇨ Then, set the **RTTEN** bit in the **RTC_TCR** register to `1`. The value of **RTC_TCR** (which contains the **RTTEN** bit) also changes.



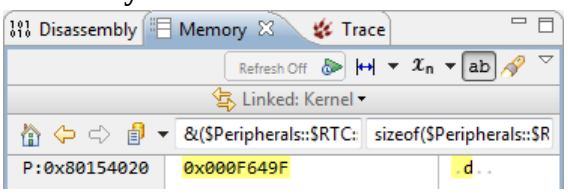
When it is enabled, the RTC copies the value from **RTC_TLR1** to **RTC_TDR** and begins counting it down at a rate of 32KHz. When **RTC_TDR** reaches zero the RTC copies the value again and repeats.

⇨ Step the target a few times and you will see the RTC_TDR register updating "by itself".

Since the peripheral register has a memory address we can also view it in a **Memory** view.

⇨ Right-click on the **RTC_TDR** register and choose **Show in Memory** from the context menu.

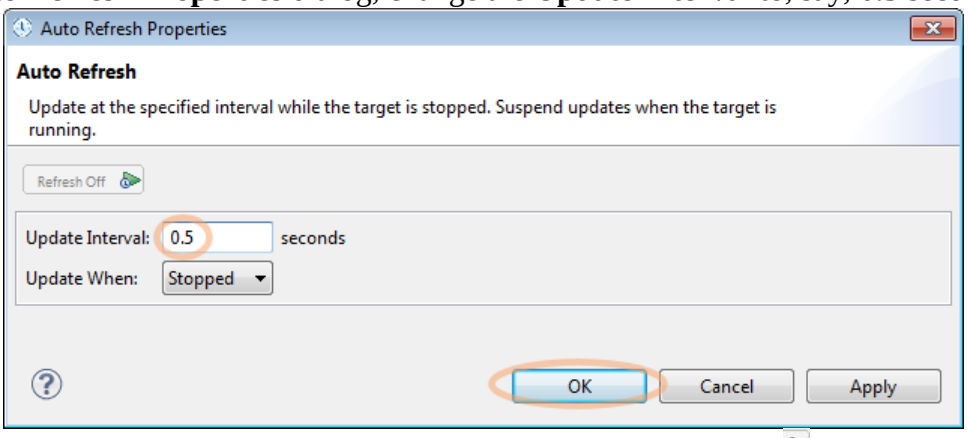
A Memory opens showing the register value. The address is shown with `P:` to indicate that it is a physical address.



⇨ Choose **Refresh** from the **Memory** view's drop-down menu ( ) and see the value changing.

The **Memory** view also supports timed auto refresh.

⇨ Click on the left part of the Timed auto refresh button (Refresh Off) that says "Refresh Off" to open the **Auto Refresh Properties** dialog; change the **Update Interval** to, say, **0.5** seconds and then click OK:



⇨ Now click on the right part of the Timed auto refresh button ( ) to start the automatic refresh. When you're finished watching the value change, click it again to stop the refresh.

⇨ When you're finished with your investigations, click the Continue button ( ) and let the target run.

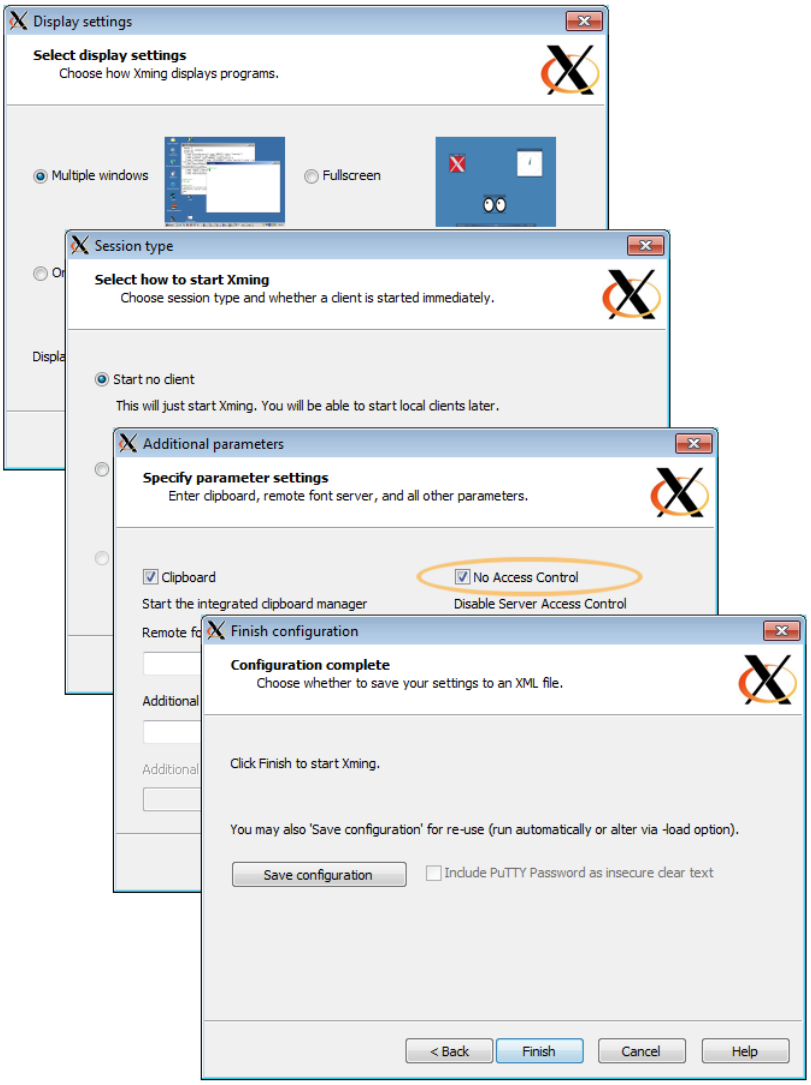
⇨ With the target still running, click the **Disconnect From Target** button ( ) in the **Debugger Control** view to disconnect the **Kernel** debug configuration.

Next we're going to do the application debug and profiling sections. They use an Ethernet connection to the target and do not use the DSTREAM. You can remove the DSTREAM power and even disconnect it from the target.

Starting the X server on the Host

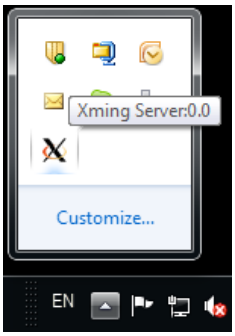
We're going to start the Xming X server on the Windows host so that the Gnometris Linux application running on the target can open a window on the host. If you are using a Linux host, or you have a monitor, keyboard and mouse attached to your target, then you don't need to use Xming.

⇨ Start the XLaunch wizard from **Start > All Programs > Xming > XLaunch**. Choose **Multiple Windows**, "**Next >**", **Start no client**, "**Next >**", check **No Access Control**, "**Next >**" and click the **Finish** button.



Note: Using **No Access Control** is simple, but insecure. Any machine connected to the same network as your host will be able to open windows on the host.

When started, Xming will not initially display any window. The only indication that it is running will be an icon in the Notification area of the task bar.



If you have trouble with the target and host communicating and your host is running a firewall you may need to configure it to allow network traffic from the target (for example, make IP address `169.254.0.100` a "friend").

Importing Gnometris

We need to import the two example projects **distribution** and **gnometris** if they have not been imported already.

⇨ Look in the **Project Explorer** view and if **distribution** and **gnometris** do not appear there, follow the instructions for importing them in *Importing projects* in the appendix on page 87. Also follow the instructions for changing the Gnometris sources in the appendix on page 89.

If **Project > Build Automatically** is checked, the `gnometris` project will be built automatically after it has been imported. The build produces two files at the top level of the project: `gnometris`, which is the application, and `libgames-support.so` which is a shared library used by the application. These two files will contain debug information. Copies of these two files with the debug information removed are created in the `stripped` subdirectory of the project. The project contains pre-built copies of these four files which will be overwritten when you build the project. It also contains backup copies of the four files named `gnometris_backup` and `libgames-support.so_backup` which do not get rebuilt. We won't be using these backup files.

Connecting to the Target

Next, we'll establish a connection to the target using Eclipse's Remote Systems Explorer (RSE) so that we can browse its file system and create a terminal connection. The Remote Systems view is part of the **DS-5 Debug** perspective, so we'll switch to that perspective now. It's possible that the RSE connection has already been created in the **Remote Systems** view for you.

⇨ Choose **Window > Open Perspective > DS-5 Debug**. If **DS-5 Debug** isn't listed then you are already in the **DS-5 Debug** perspective. You can also switch perspectives by using the buttons on the Perspective toolbar (

).

⇨ If the **Remote Systems** view is not open, you can open it by choosing **Window > Show View > Other... > Remote Systems > Remote Systems** in any perspective.

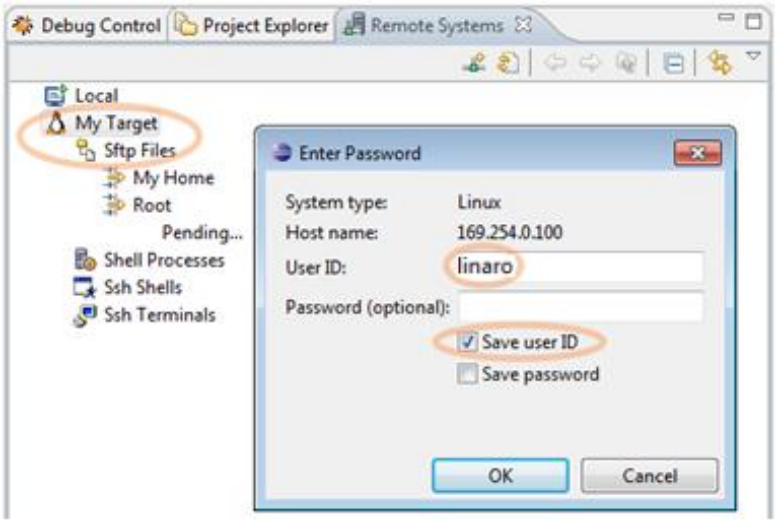
⇨ Click on the tab of the **Remote Systems** view to bring it to the front.

⇨ We won't be using the **Local** connection, so you can collapse it.

⇨ If there is no **My Target** connection, create one by following the instructions in the appendix on page 90.

⇨ Browse the target's file system; Expand **My Target > Sftp files > Root**. If the connection has **Files** instead of **Sftp Files**, then the connection was not created correctly and you should **Disconnect** it, **Delete** it and recreate it.

⇨ Enter User ID=**linaro**, Password=**linaro**; check **Save user ID** and click the **OK** button. There will be a few authentication dialogs; accept them.



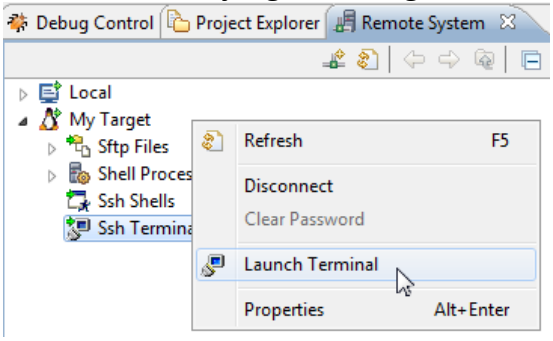
You can also expand **My Home** to browse the home directory of the user, which is `/home/linaro` for the linaro user.

You can copy files to and from the target by dragging them between the **Remote Systems** view and the **Project Explorer** view or Windows Explorer windows. As we'll see below, it's also possible to copy files to the target automatically as part of the debug configuration.

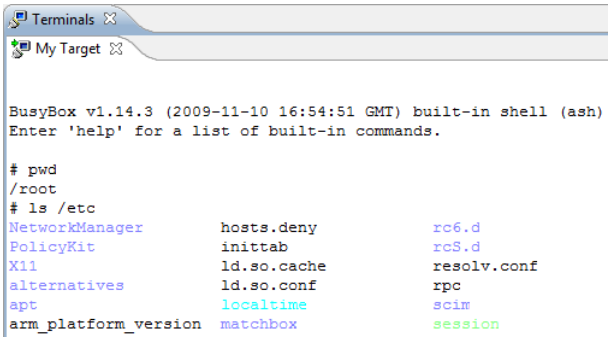
If you want, you can double click a text file on the target, for example `/etc/bash.bashrc`, and view it or even edit it in Eclipse – but it's probably best if you don't save any changes unless you are sure that you know what you are doing.

Now we'll create a terminal connection so that we can execute commands easily on the target. You can Collapse the **Sftp Files** to get them out of the way.

⇨ Create a Terminal by right-clicking on **Ssh Terminals** and choosing **Launch Terminal**.



This will open a **Terminals** view in Eclipse that can be used to execute commands on the target. (This is different from, but confusingly similar to the **Serial** view). The picture below shows the output from the example target image included with DS-5. If your target is running a different distribution the output will be different.



You can use **Launch Terminal** more than once if you want to have multiple terminal sessions to the target.

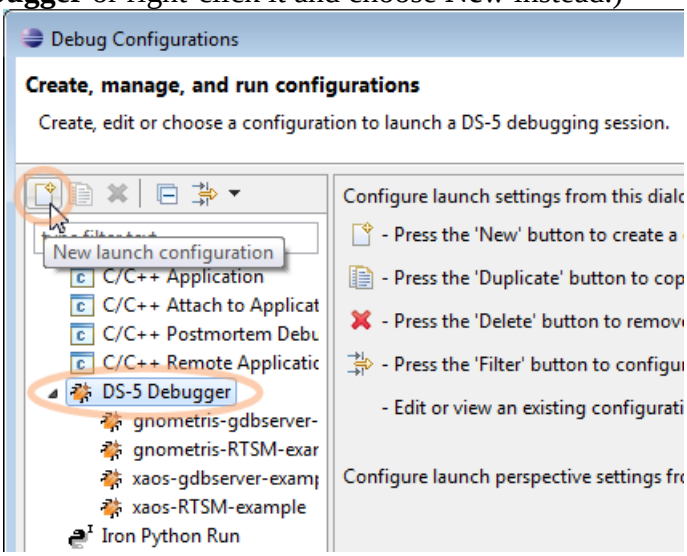
Debugging Gnometris on the Target

If you've just done the U-Boot and kernel debugging some of the following will already be familiar, but the views are discussed in significantly more detail.

The Gnometris example project includes two debug configurations, **gnometris-gdbserver-example,** and **gnometris-RTSM-example**, for running the application on the Real Time System Model (RTSM). You can try the RTSM debug configuration later if you want, but we are going to ignore them for now and make our own. If the Xaos example has already been imported, you'll also have debug configurations named **xaos-RTSM-example** and **xaos-RTSM-example** which we're also going to ignore.

Since we are using a Snowball board as our target, we'll create our own debug configuration. The **gnometris** debug connection may already have been created for you in which case you can just select the existing connection and examine the settings.

⇨ Choose **Run > Debug Configurations...** then select **DS-5 Debugger** and click the **New launch configuration** button ( ) to create a new debug configuration. (You can also double-click **DS-5 Debugger** or right-click it and choose **New** instead.)



⇨ Give the debug configuration a name, I used ***gnometris***.

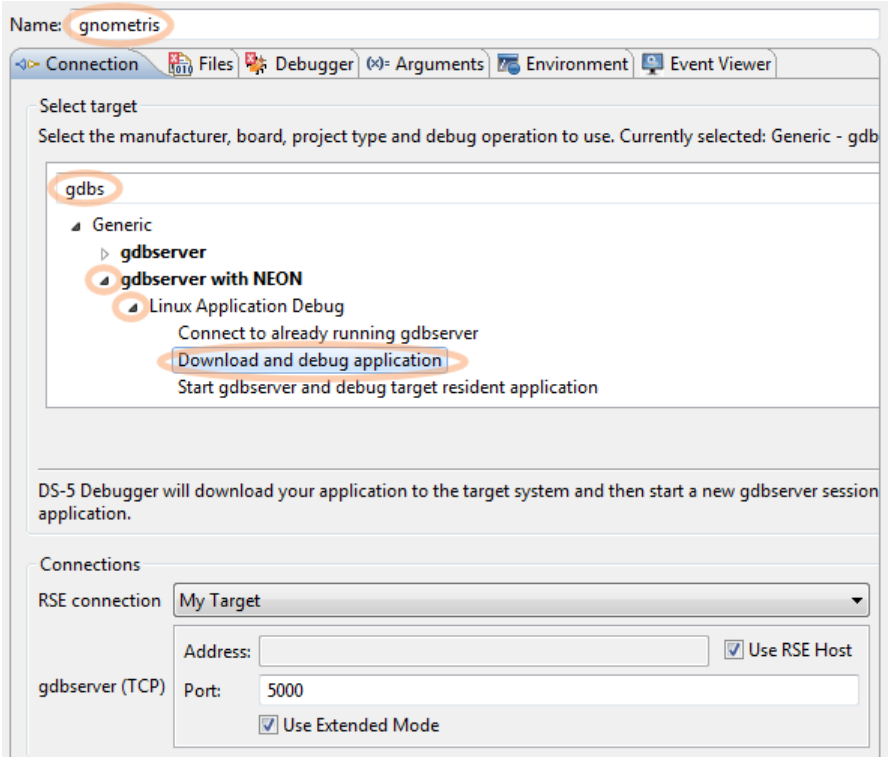
In the **Connection** pane of the debug configuration we need to specify the platform and IP Address of the target.

⇨ Type ***gdbs*** in the Filter platforms filter box so that the only the matching platforms are shown.

⇨ Expand **Generic > gdbserver with NEON > Linux Application Debug** in the project list.

⇨ Choose the **Download and debug application**. **My Target** will already be the selected **RSE configuration** and **Use RSE Host** and **Use Extended Mode** will be checked. Leave the **Port** number as

5000.

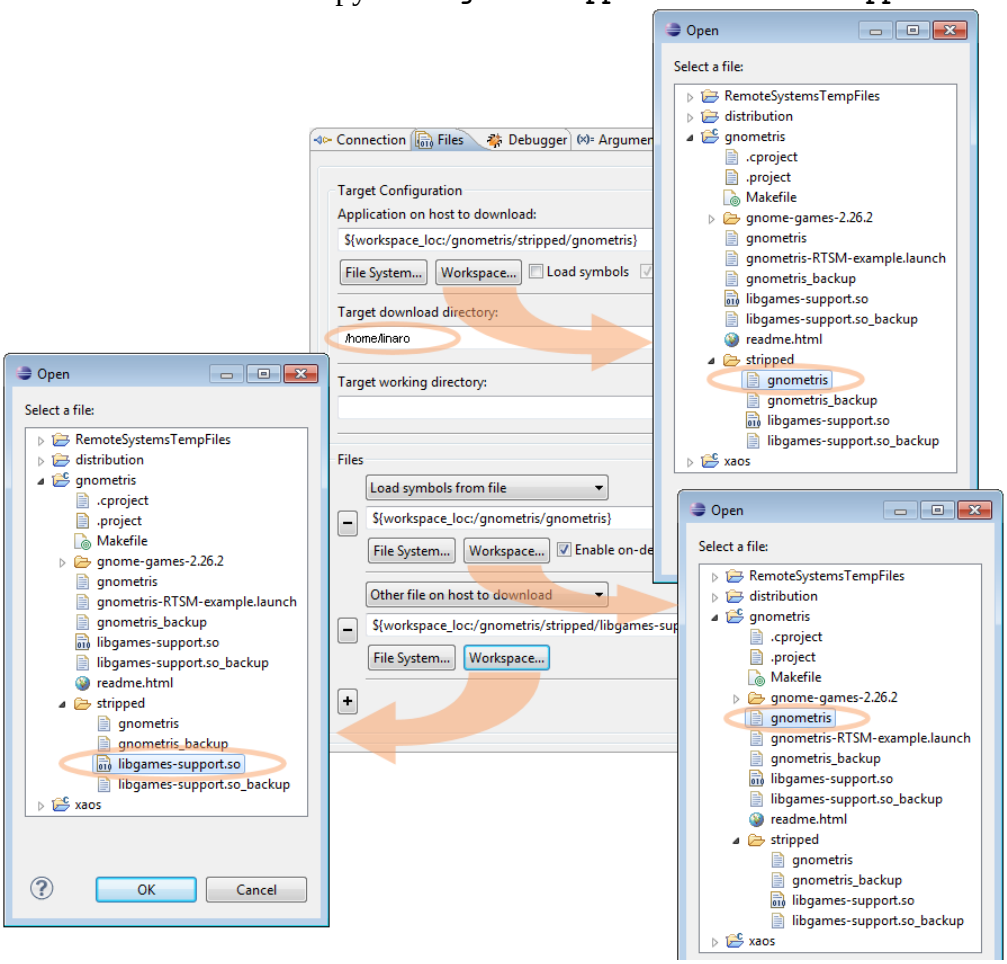


The **Debug** button is disabled because the debug configuration is not yet complete. The problem is explained near the top of the dialog: **[Files]: No target download directory details entered**. We will fill in the missing details in the other panes.

In the **Files** pane of the debug configuration we need to specify the application to download, the target directory to download it to, the location of the symbols (debug information) and any additional files to download.

- ⇨ For **Application on host to download**, use the **Workspace...** button to choose the copy of **`gnometris`** in the **`stripped`** sub-directory. The symbols are not needed on the target. The unstripped version will also work, but it will take longer to download and take up more space on the target. Do not use the **`gnometris_backup`** copy because it will not include the changes made earlier.
- ⇨ Put **`/home/linaro`** in the **Target download directory**. This is the home directory of the root user. Any other writable directory, like **`/tmp`**, would work as well. The application and shared library, which we'll specify below, will be downloaded to this directory when we start debugging.
  We can leave the **Target working directory** empty. By default the download directory will be used as the working directory.
- ⇨ In the first entry in the **Files** list, choose **Load symbols from file** and use the **Workspace...** button to choose the copy of **`gnometris`** with symbols (the copy *not* in the **`stripped`** sub-directory).
- ⇨ Click the add button () to add a second entry to the **Files** list.

⇨ In the second entry in the **Files** list, choose **Other file on host to download** and use the **Workspace...** button to choose the copy of `libgames-support.so` in the `stripped` sub-directory.

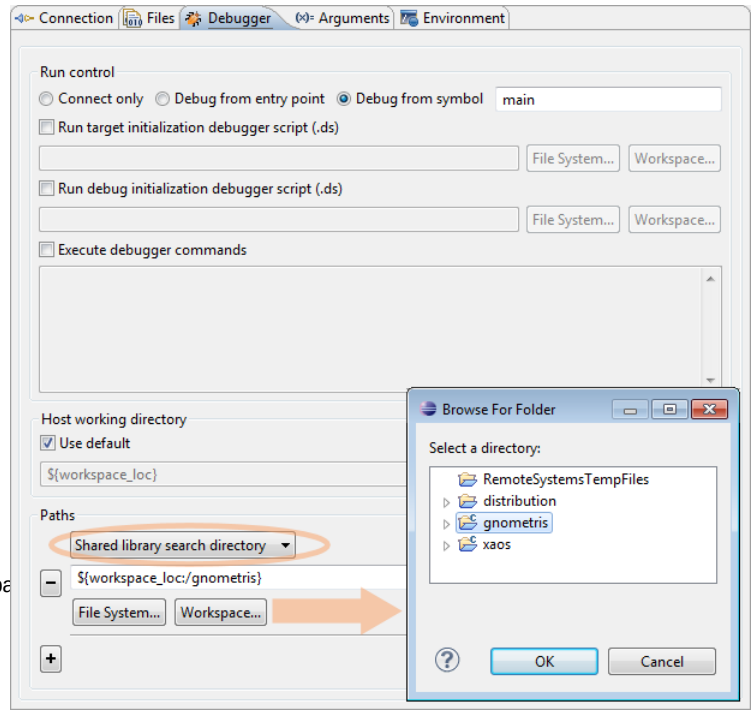


You do not need to add the symbol file for the shared library because the debugger will search for it when it sees the shared library get loaded at runtime, but you could add another entry to the **Files** list to load symbols from the unstripped copy of `libgames-support.so` and the debugger would use it instead of searching.

In the **Debugger** pane of the debug configuration:

We are going to use **Debug from symbol** so that DS-5 will begin debugging by running the application and stopping at **main**.

You can specify debugger scripts and/or commands to execute as part of the connection process. The dialog fields have tooltips that explain at what point during connection the scripts and commands are executed. An example would be to use the **Execute debugger commands** field to disable some breakpoints and enable others. The script files can be written in either the simpler, gdb-like, DS-5 command-line scripting language (`.ds` files) or the more powerful and complex scripting language Python (`.py` files). But we don't need to use them, so we'll just leave them disabled.



You can specify the host working directory which affects certain debugger commands

pa

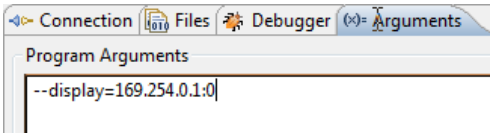
that access the host file system (for example **`dump`**, **`log`** and **`source`**). The default is fine.

You may need to scroll down or grow the dialog by dragging the lower right corner to see the entry in the **Paths** list.

⇨ In the first entry in the **Paths** list, choose **Shared library search directory** and use the **Workspace...** button to choose the **`gnometris`** project directory. The debugger will search this directory for symbols of shared libraries when they are loaded as the application starts or runs. In our case the debugger will find the copy of **`libgames-support.so`** with symbols there.

In the **Arguments** pane of the debug configuration:

⇨ Type **`--display=169.254.0.1:0`** (no spaces) in the **Program Arguments** field:

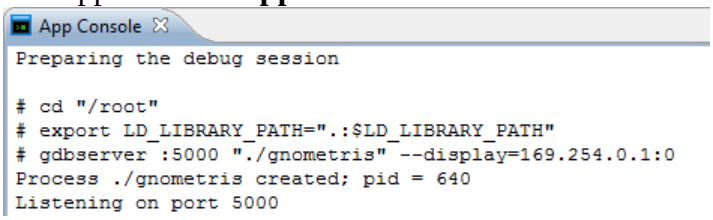


The **`--display=169.254.0.1:0`** argument tells Gnometris to open its window on the X server running on the host. If you want Gnometris to display on the target's own X server, you would use **`--display=:0`** instead.

We could use the **Environment** pane to set any environment variables that we needed on the target. For example we could set **DISPLAY** instead of using the **`--display`** command-line argument. In this case, we can leave the **Environment** pane blank.

⇨ Click the **Debug** button. If the **Debug** button is disabled then Eclipse will put a message explaining why at the top of the **Debug Configurations** dialog. If the **Confirm Perspective Switch** dialog appears check **Remember my choice** and click **OK**.

DS-5 will download the application and shared library and start running it using gdbserver. Some progress messages will appear in the **App Console** view:



The game will start running and stop at main() before it has opened its window. If the game does not start, there may be error messages in the **App Console** view that indicate what is causing the problem. The debugger will look like the screen shot below.

When you create a debug configuration, like **gnometris**, in the **Debug Configurations** dialog it also appears in the **Debug Control** view. The **gnometris-RTSM-example** debug configuration that was imported with Gnometris appears in the **Debug Control** view. Since we're not going to use it for the time-being, you can select **gnometris-RTSM-example** in the **Debug Control** view and click the Remove Connection button ( ). You can also right-click on the configuration and choose **Remove Connection** from the context menu . You can also remove **xaos-RTSM-example** if it's there. We'll see later how you could get it back if you want.

If you want to restart the debug session from the beginning for some reason, you can click the Debug from main() button ( ) in the **Debug Control** view. Or you can disconnect and reconnect the debug configuration:

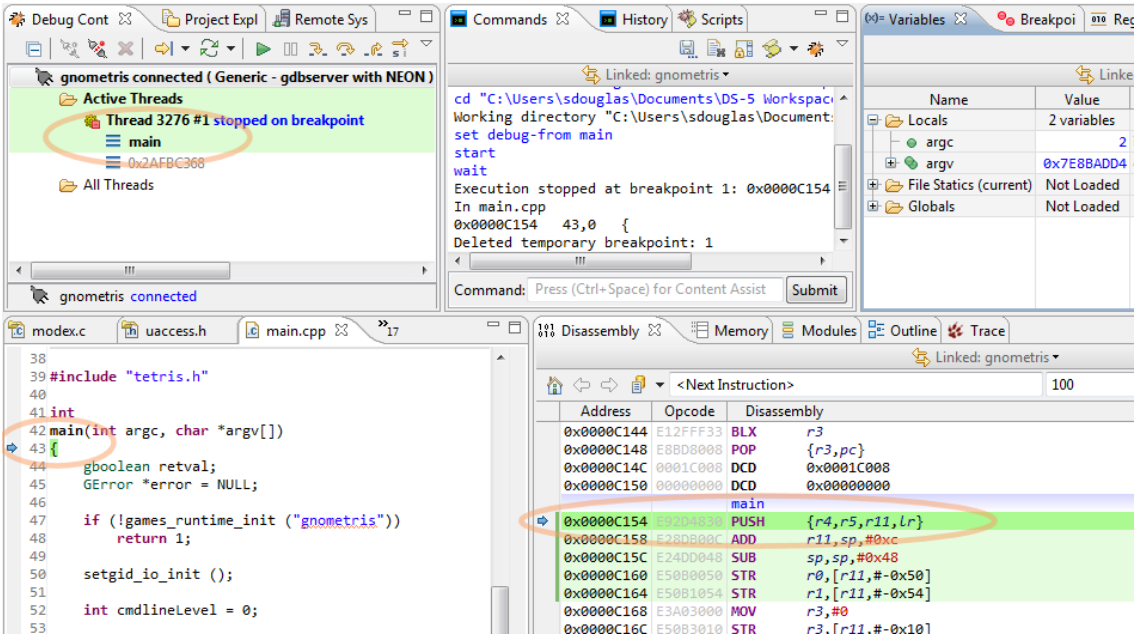
1. With the debug configuration selected in the **Debug Control** view, click the **Disconnect From Target** button ( ).

2. Click the connect button ( ) to re-connect. You can also double-click the configuration or open the **Debug Configurations** dialog, choose the debug configuration there and click the **Debug** button.

Any breakpoints that were set in the configuration will be remembered between connections.

Detailed Debugging

Because **Debug from symbol** is set to **`main`** in our debug configuration, when the DS-5 Debugger connects to the gdbserver that it starts on the target, it will stop the application at **`main`**.



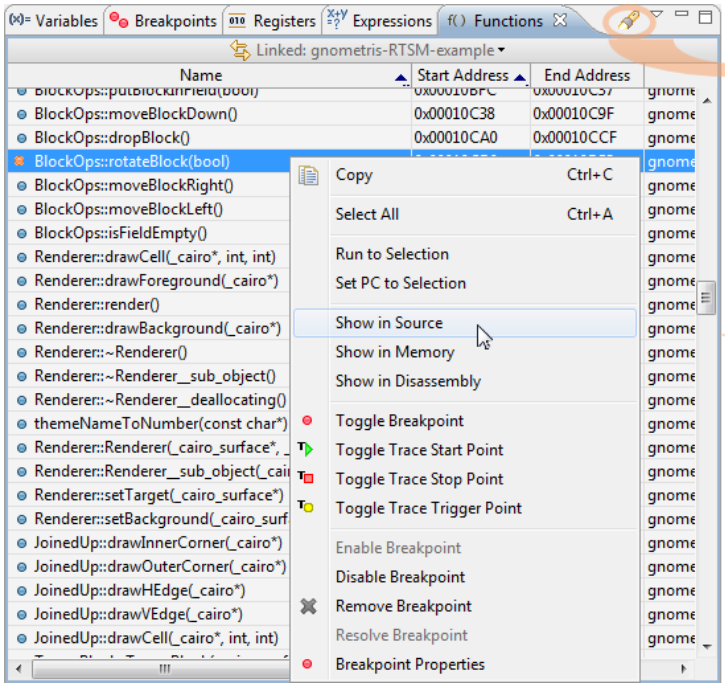
When the game stops at **`main`** it will not have opened its window yet.

Functions view

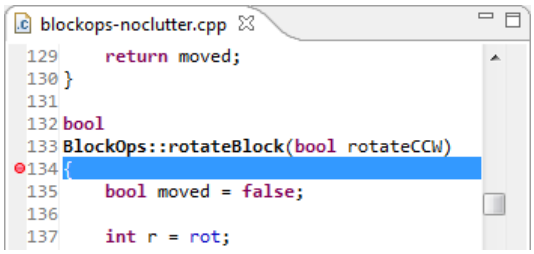
We'll use the **Functions** view to set a breakpoint on **`BlockOps::rotateBlock()`** which is called when the user rotates the current block, by pressing up-arrow.

⇨ Click on the tab of the **Functions** view to bring it to the front; then click on the Search button ( ) to open the **Search Functions** dialog; type **rot** as the search text (or any other part of the name **BlockOps::rotateBlock**); select **`BlockOps::rotateBlock(bool)`** and click OK.

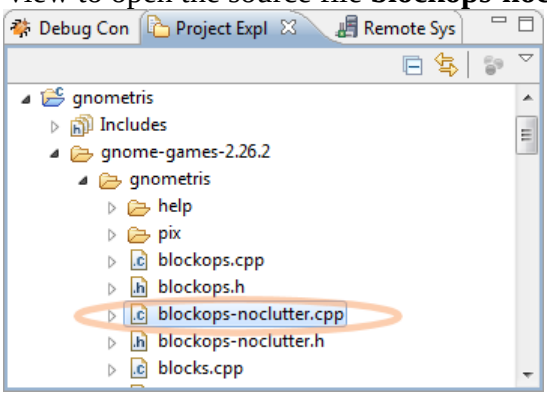
The function **`BlockOps::rotateBlock()`** is now selected in the Functions view.



⇨ Double click on **`BlockOps::rotateBlock()`** to set a breakpoint on it. The breakpoint symbol (●) will appear. Then right-click on **`BlockOps::rotateBlock()`**; then choose **Show in Source** from the context menu.

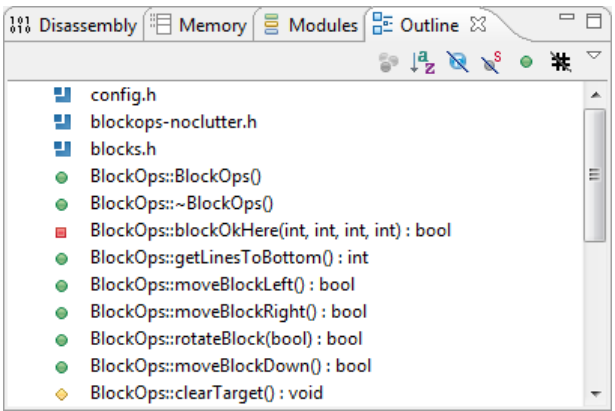


We can see that the breakpoint we've set is on line 134 of **blockops-noclutter.cpp**. We could also use the **Project Explorer** view to open the source file **blockops-noclutter.cpp** by double-clicking on it.



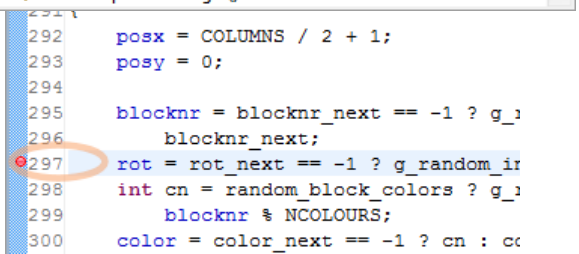
Outline view (another brief digression)

You can use the handy **Outline** view to quickly navigate the front-most source view. Clicking on a function name will scroll the source view so that the beginning of the function is visible.



The **Outline** view has a button that toggles between sorting the view alphabetically (), and showing the list in file order. There are also buttons that show or hide different kinds of entries. You can get a tooltip description of the buttons by "hovering" the mouse pointer over them.

Breakpoints



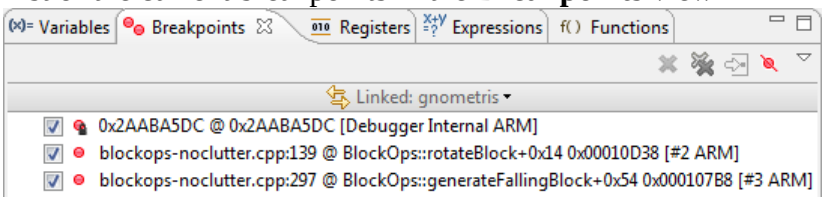
You can also set a breakpoint by double-clicking in the left margin of a source or disassembly line where the breakpoint symbol (●) will appear. You can delete a breakpoint by double-clicking it, too.

⇨ Set a breakpoint by double-clicking in the left margin on line 297 in `BlocksOps::generateFallingBlock()`. This function is in the same file, **blockops-noclutter.cpp**. You can go directly to a line by number by using the **Navigate > Go To Line...** (Ctrl+**L**) command.

`BlocksOps::generateFallingBlock()` is called when Gnometris wants to create a new block

Breakpoints view

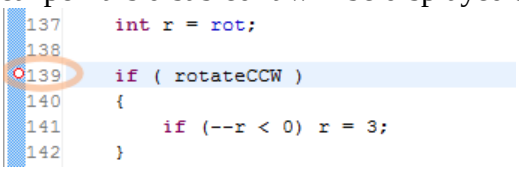
There is a list of the current breakpoints in the **Breakpoints** view



Besides the breakpoints you've set, you can see the debugger internal breakpoints that DS-5 uses.

(Possibly) Fun things to do with breakpoints in the Breakpoint view

- You can disable each breakpoint individually without removing it by unchecking it. When a breakpoint is disabled it will be displayed as hollow.



- There is also a button to globally skip all breakpoints ( ). The individual enable status is remembered while the breakpoints are globally skipped.
- There are buttons to delete individual breakpoints ( ) and delete all breakpoints ( ).
- There is also a button ( ) that will open the source file(s) of the selected breakpoint(s). Double-clicking a breakpoint in the breakpoint view will also open its source file.
- There is a **Breakpoints** view drop-down menu ( ). You can use the commands in it to import and export the breakpoints from/to an XML file. You can control the sorting of the **Breakpoints** view.
- You can use copying (Ctrl+C or right-click) in the breakpoints view to get a text list of the selected breakpoints.
- You can paste or drag text into the breakpoints view to set breakpoints. For example, you can select the name of a function being called in a source view and drag it to the breakpoints view to set a breakpoint.
- You can also set a breakpoint by dragging a function from the **Outline** view into the Breakpoints view.

New views, linking and multiple configurations

You can choose **New Breakpoints View** in the **Breakpoints** view's drop-down menu to create a new breakpoint view. Most views have a similar command. There are a few reasons why you might want to do this. Some views, for example the **Memory** view can be set to show different regions and you might want to see multiple **Memory** views at once. Some views, like the Variables view can be "frozen" and you might want to see some frozen versions and an unfrozen version at the same time.

The reason you might want more than one **Breakpoints** view is different. Although we won't do it in this workshop, DS-5 Debugger can have multiple debug configurations running at the same time on the same or different targets. The multiple debug configurations and their threads and stacks appear in the **Debug Control** view. By default the other views change the information that they are displaying according to which debug configuration is currently selected in the **Debug Control** view. This is shown by **Linked:** in

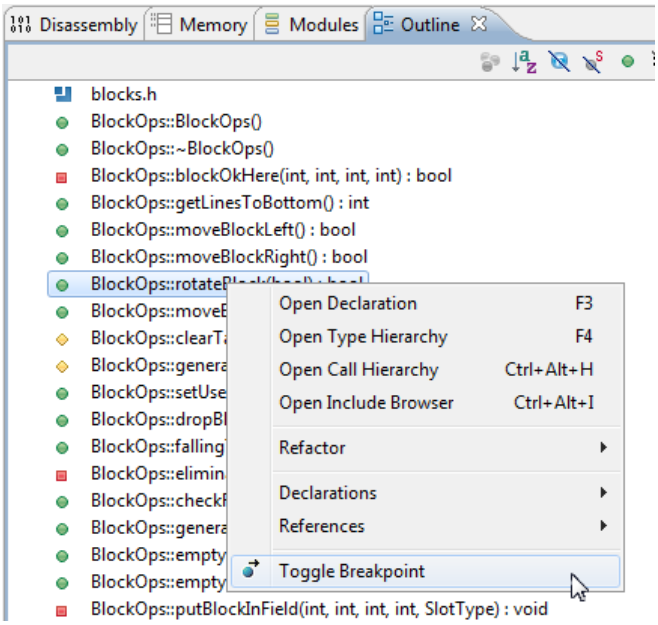
the link menu near the top of each view 

. You can use this menu to change a view so that it "sticks" with one debug configuration instead of changing to show the information about the currently selected configuration. So if you had two configurations you could use a second **Breakpoints** view to see the breakpoints in both configurations at the same time.

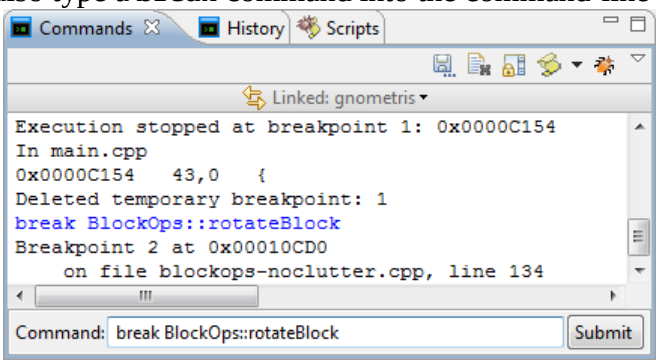
You can reset all the views to the default **Linked:** setting by using the **Reset DS-5 views to Linked** menu item in the **Debug Control** view's drop-down menu ().

More ways to set breakpoints

You can also right-click on a function in the **Outline** or **Functions** views and choose the **Toggle Breakpoint** menu item.



You can also type a **`break`** command into the command-line at the bottom of the **Commands** view.



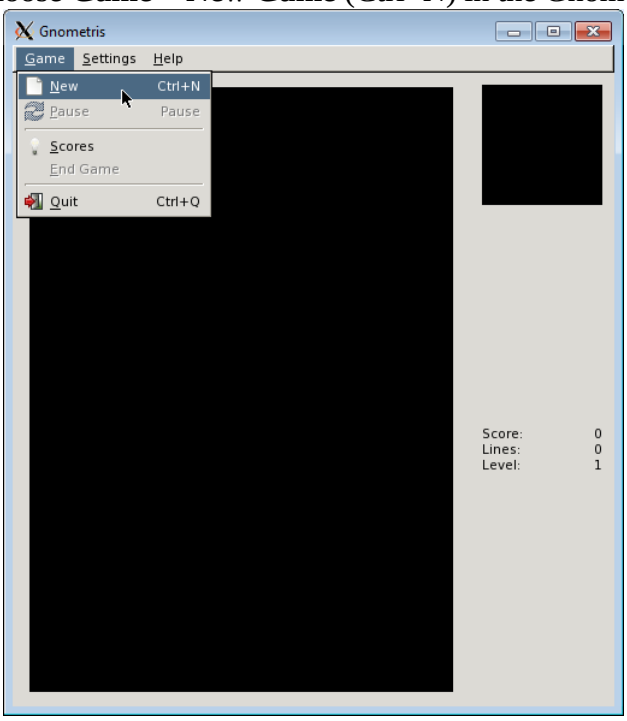
You can use a function name, a file name and line number (for example, **`blockops-noclutter.cpp:134`**) or an address (for example, **`*0x11690`**) as the argument to the **`break`** command.

We'll see more features of breakpoints and more ways to set them later. We'll also see more uses of the **Commands** view later.

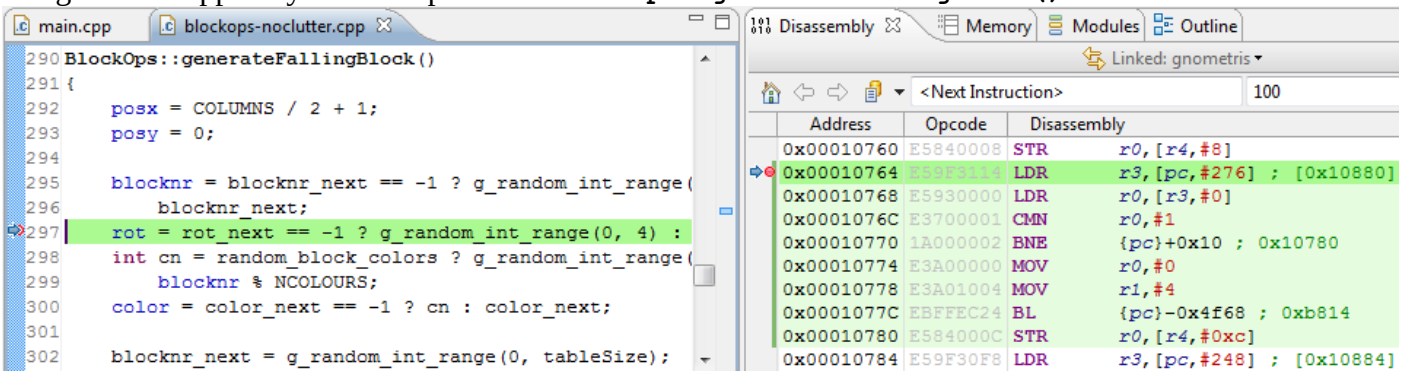
Source and Disassembly views

- ⇨ If you've changed the breakpoints since you set them, make sure the two breakpoints are set, enabled (checked) and not being skipped.
- ⇨ Click on the tab of the **Debug Control** view to bring it to the front, if it is not already in front. Click the "beating" Continue button () in the **Debug Control** view. The game begins running and, after a bit, opens its window.

⇨ Choose **Game > New Game** (Ctrl+N) in the Gnometris window.

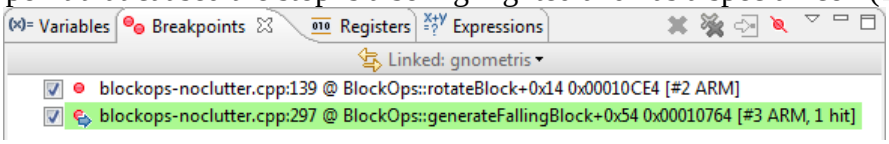


The game is stopped by the breakpoint in **`BlocksOps::generateFallingBlock()`**.



The current PC location is shown by an arrow (⇨) in the left margin of both the source and **Disassembly** views and by the dark green highlighting. The light green highlighting in the **Disassembly** view shows all of the assembly instructions that correspond to the current source line.

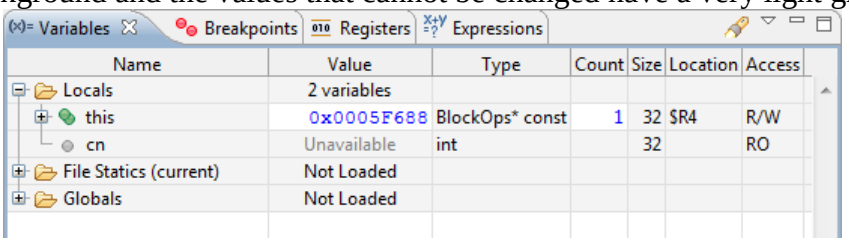
The breakpoint that caused the stop is also highlighted and has a special icon (⇨) in the **Breakpoints** view:



We'll see more features of source and **Disassembly** views later

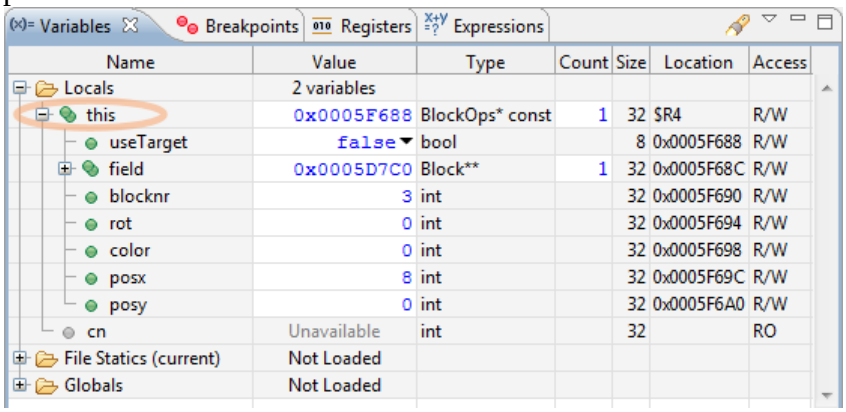
Variables view

The **Variables** view shows the local, file static and global variables. The values that can be changed have a white background and the values that cannot be changed have a very light gray background.



The **Locals** part of the view shows the arguments and local variables to the function. Because of the current location of the PC the variable **`cn`** is not available yet. We will be able to see it after we have stepped past its initialization on line 298. Since **`BlocksOps::generateFallingBlock()`** is a C++ member function it has a **`this`** argument that is a pointer to a **`BlocksOps`** structure.

⇨ Expand **`this`** to see its data members

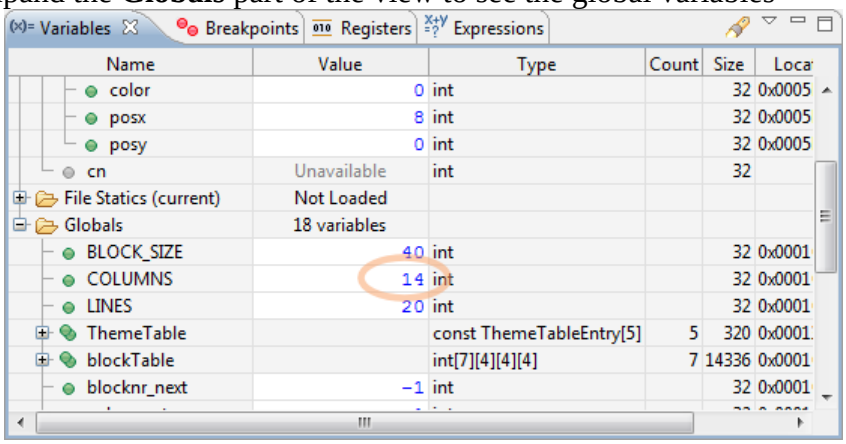


Name	Value	Type	Count	Size	Location	Access
Locals	2 variables					
this	0x0005F688	BlockOps* const	1	32	$R4	R/W
useTarget	false	bool		8	0x0005F688	R/W
field	0x0005D7C0	Block**	1	32	0x0005F68C	R/W
blocknr	3	int		32	0x0005F690	R/W
rot	0	int		32	0x0005F694	R/W
color	0	int		32	0x0005F698	R/W
posx	8	int		32	0x0005F69C	R/W
posy	0	int		32	0x0005F6A0	R/W
cn	Unavailable	int		32		RO
File Statics (current)	Not Loaded					
Globals	Not Loaded					

The **Location** column shows that **`this`** is currently in register R4 and it shows the memory addresses of the data members. Notice that the **`rot`** and **`color`** data members have the value zero, since they were initialized to those values in the **`BlocksOps`** constructor. They will be set as we execute the next few source lines. The **Size** column shows the size of each variable in bits. The **Access** column shows if the variable or member is read-only (**RO**) or read/write (**R/W**).

The **`field`** data member has type **`Block**`** and is a pointer to an array of **`COLUMNS`** **`Block*s`** that was set in the constructor. **`COLUMNS`** is a global variable; we'll find out its value.

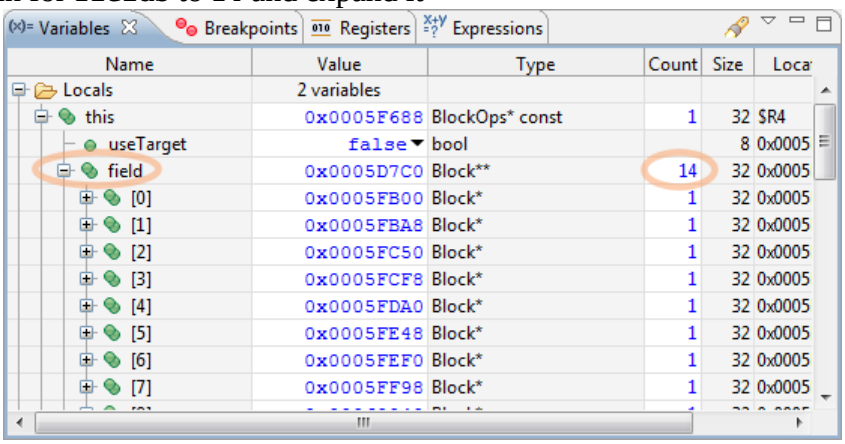
⇨ Expand the **Globals** part of the view to see the global variables



Name	Value	Type	Count	Size	Loca
color	0	int		32	0x0005
posx	8	int		32	0x0005
posy	0	int		32	0x0005
cn	Unavailable	int		32	
File Statics (current)	Not Loaded				
Globals	18 variables				
BLOCK_SIZE	40	int		32	0x0001
COLUMNS	14	int		32	0x0001
LINES	20	int		32	0x0001
ThemeTable		const ThemeTableEntry[5]	5	320	0x0001
blockTable		int[7][4][4][4]	7	14336	0x0001
blocknr_next	-1	int		32	0x0001

We can see that the value of **`COLUMNS`** is **`14`**.

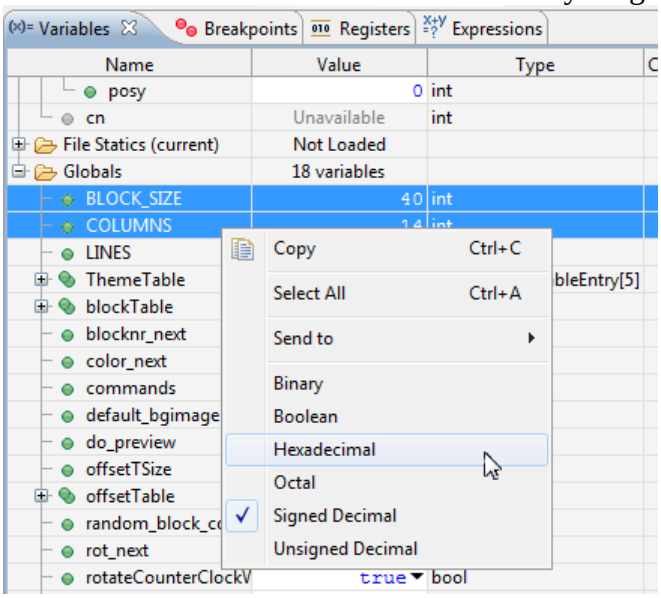
⇨ To see all 14 elements of the array that the data member **`fields`** is pointing to, we can set the **Count** column for **`fields`** to **`14`** and expand it



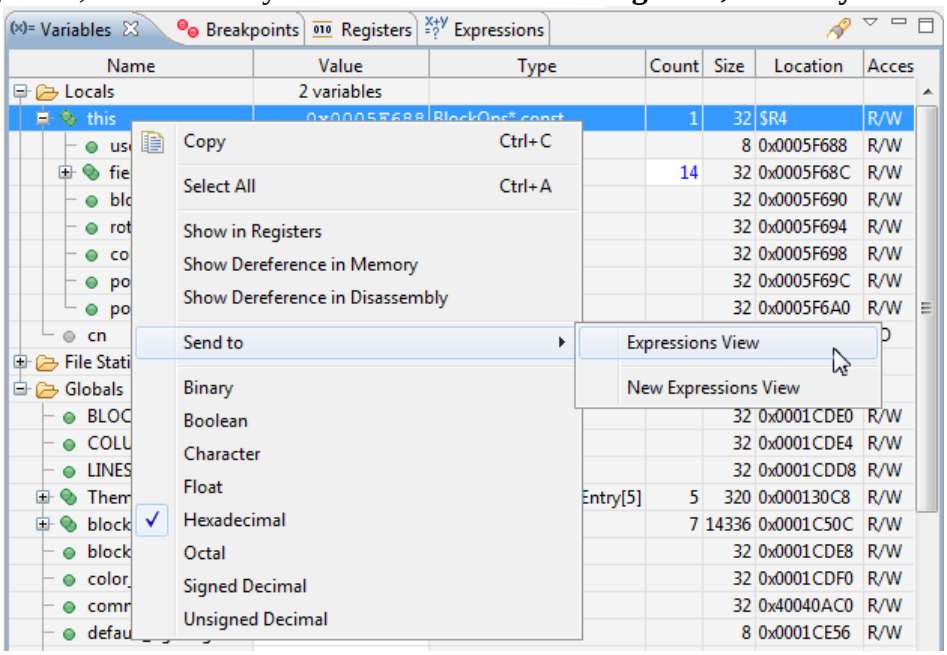
Name	Value	Type	Count	Size	Loca
Locals	2 variables				
this	0x0005F688	BlockOps* const	1	32	$R4
useTarget	false	bool		8	0x0005
field	0x0005D7C0	Block**	14	32	0x0005
[0]	0x0005FB00	Block*	1	32	0x0005
[1]	0x0005FBA8	Block*	1	32	0x0005
[2]	0x0005FC50	Block*	1	32	0x0005
[3]	0x0005FCF8	Block*	1	32	0x0005
[4]	0x0005FDA0	Block*	1	32	0x0005
[5]	0x0005FE48	Block*	1	32	0x0005
[6]	0x0005FEF0	Block*	1	32	0x0005
[7]	0x0005FF98	Block*	1	32	0x0005

⇨ Collapse the data member **`fields`** to hide the array elements again; we aren't really interested in them.

You can change the format used to show values by selecting one or more rows, right-clicking and choosing a new format from the context menu. It won't hurt anything if you want to try it out.



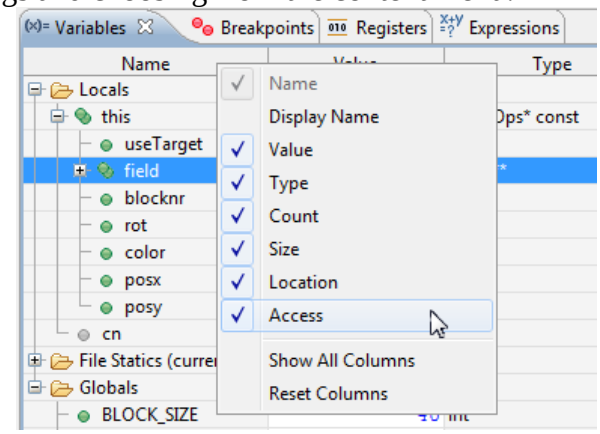
Depending on the type and location of the variables selected, the context menu will have some of the following items; feel free to try them out. We'll see the **Registers, Memory** and **Expressions** views later.



- If the variable is stored in a register then **Show in Registers** will select it in the **Registers** view.
- If the variable is stored in memory then **Show in Memory** will make it visible in the **Memory** view and **Show in Disassembly** will make it visible in the **Disassembly** view.
- If the variable has a pointer type then **Show Dereference in Memory** will show the memory that it points to in a **Memory** view and **Show Dereference in Disassembly** will show the memory that it points to in a **Disassembly** view.
- You can choose **Send to > Expressions View** to add the selected variables to an existing **Expressions** view or **Send to > New Expressions View** to create a new **Expressions** view containing the selected variables.

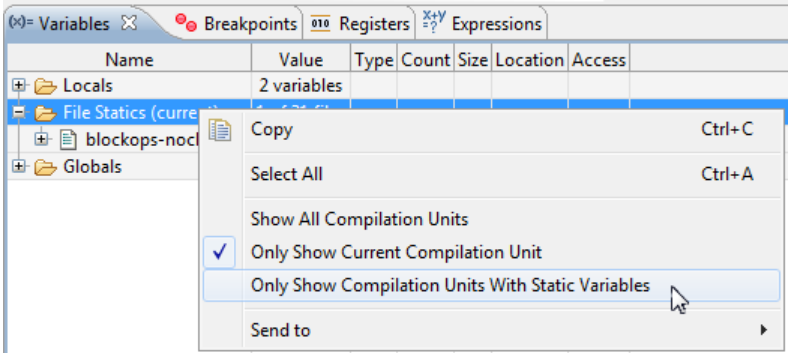
The values displayed will change as we step through the program execution. Changed values are shown with yellow highlighting. We can also change the values if we want (not now though, we'll do that later). You can choose values for variables of Boolean and enum type by using a drop-down menu.

You can change which columns are displayed in the **Variables** view by right-clicking on the column headings and choosing from the context menu.

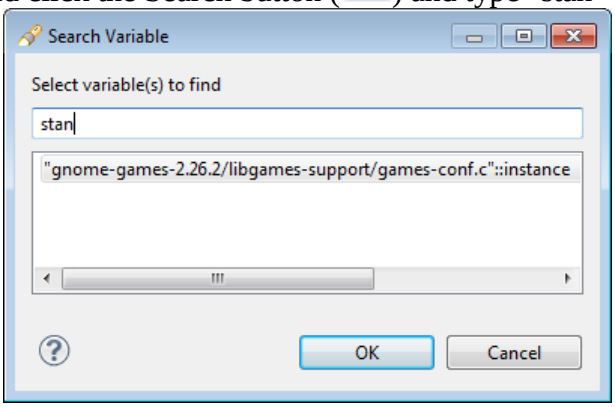


You can also change which columns are displayed in the **Registers**, **Expressions** and **Modules** views this way.

The **File Statics** are the variables that are declared with the C/C++ `static` keyword and are local to a file. The current file, **blockops-noclutter.cpp** does not have any file statics. You can change which files are listed under **File Statics** by right-clicking on the **File Statics** row and choosing from the context menu.

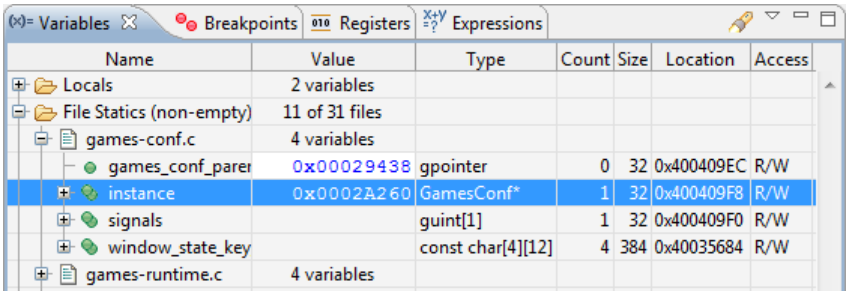


The **Variables** view has a Search button () at the top which opens a dialog that makes it easy to find a variable if you know part of its name. For example, say we we're looking for a variable named `instance`; we could click the Search button () and type "stan" (without the double quotes):



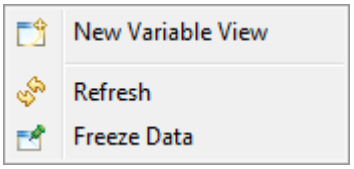
This shows us there is only one matching variable (which happens to be a file static in the shared library). (If you don't see it then something is wrong in the debug configuration that is preventing the unstripped shared library from being found.) When we select it and click **OK** (or double-click it) then the variable is

shown in the **Variables** view:



The Search button is also available in the **Registers**, **Expressions**, **Functions**, **Memory** and **Disassembly** views.

The **Variables** view also has a drop-down menu ( ) which contains these commands:



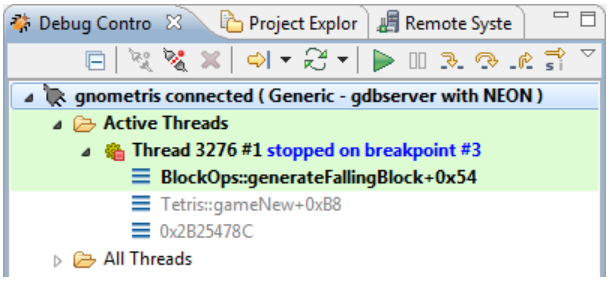
- The **New Variables View** command creates a new, unfrozen **Variables** view so that you can have more than one **Variables** view at the same time.
- The **Refresh** command causes the values to be re-read from the target. This could be useful for example when examining values that are changed by hardware or another process.
- The **Freeze Data** command prevents the values from being updated so that you can later see the values as they were at the time they were frozen.

The **New View**, **Refresh** and **Freeze Data** commands are also available in drop-down menus in other views: **Disassembly**, **Expressions**, **Memory**, and **Registers**.

Later we'll see the **Expressions** view which is similar to the **Variables** view, but can display more complex expressions and just the expressions that you specify.

Stepping

The stepping commands are buttons in the **Debug Control** view:



- Step (into) (; **F5**): the Step command steps to the next source line or assembly instruction but *into* function calls
- Step Over (; **F6**): the Step Over command steps to the next source line or assembly instruction and *steps over* function calls
- Step Out (; **F7**): the Step Out command executes the rest of the current function and stops when it returns to its caller

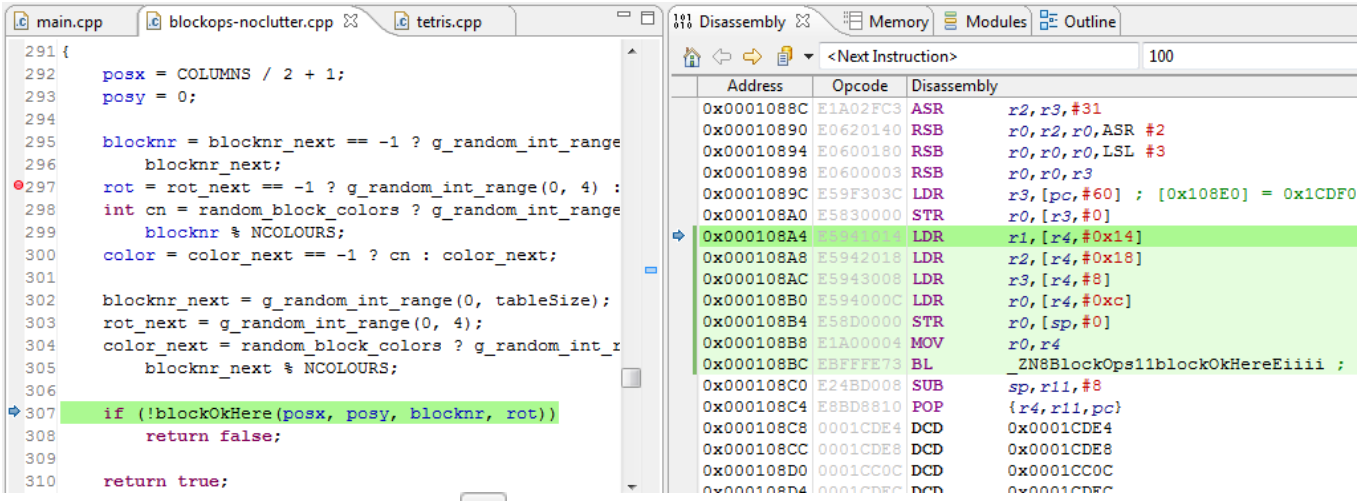
Stepping can happen either by source line or assembly instruction. The **Debug Control** view has a toggle button (;) that you can use to switch between the two modes. By default stepping is by source line with the '**s**' dark ( ).

⇨ In stepping-by-source-line mode ( ), click the Step Over button (; **F6**). The execution advances to the next source line by executing all of the assembly instructions that are highlighted with light green in the **Disassembly** view. You can see in the **Variables** view that the `rot` data member has been set to random integer from zero to 3. (The new PC arrow position is a bit strange, but it's up to the compiler (in this case gcc) to tell the debugger the correct line number information.)

⇨ Click Step Over again. The local variable `cn` gets a location (register R0) and a value.

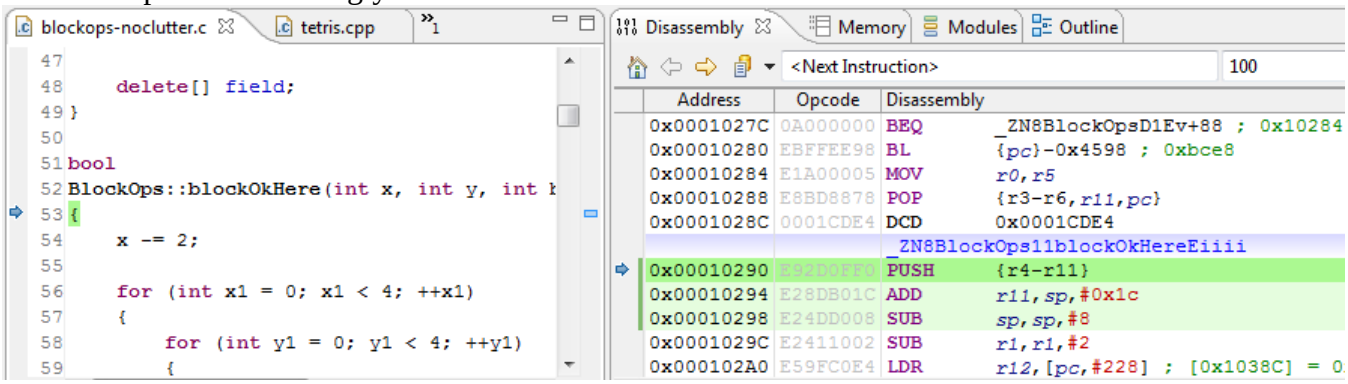
⇨ Click Step Over a third time. The `color` data member has been set to the value of `cn`.

⇨ Click Step Over three more times; until the `if` statement on line 307 is the next one to be executed. Instead of stepping, you can click on the source line and then right-click and choose **Run to Selection** from the context menu. If you step too far, just click the Continue button ( ) and wait until the current block finishes falling and the next block is generated and the breakpoint is hit again.



➪ Click the Step (into) button ( ) instead of Step Over.

The execution steps into the call to **`BlockOps::blockOkHere()`** and the **Variables** and **Disassembly** views are updated accordingly.

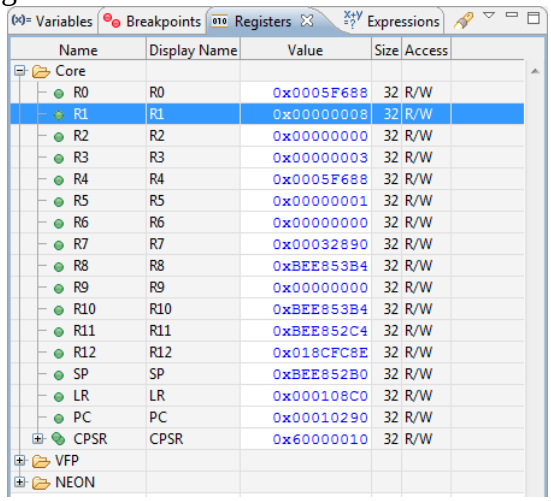


We can see from the green highlighting in the **Disassembly** view that using the Step Over command in stepping-by-source-line mode ( ) will step two assembly instructions. You can try switching to stepping-by-assembly-instruction mode ( ). In this mode, Step Over (; **F6**) will step just one assembly instruction. Remember to switch the stepping mode back to stepping-by-source-line mode to avoid confusion later.

Registers view

In the **Variables** view we can see that **`x`** has the value 8 and is located in register **`R1`**.

➪ Right-click on **`x`** in the Variables view and choose **Show in Registers**.



Sure enough, **`R1`** has the value 8.

➪ Select the **`0x00000008`** and change the value of **`R1`** to 9. This change won't hurt the game.

➪ Look back at **`x`** in the **Variables** view; the new value is shown.

⇨ While looking at the **Registers** view, step the application a few times. You can see the register values changing and the changed values highlighted in yellow. You can see the changes in the **Variables** view as well.

You can select one or more registers (the whole row, not just the value) and right-click to change the display format. You can expand the CPSR register and see and change the individual fields. Changing the CPSR could have bad effects on the application, so it's probably not a good idea to change them just now. The ARM core in the target has hardware floating point registers that are shown inside the **VFP** and **NEON** folders. Since Gnometris doesn't do much floating point we won't do any more with them, but feel free to have a look.

When you have a register selected you can right-click on it to get a context menu with **Show Memory Pointed to by *register***, **Show Disassembly Pointed to by *register*** and **Send to** items. The **Show Memory/Disassembly Pointed to by *register*** commands display the memory starting at the address in the selected register in a **Memory** or **Disassembly** view. Like the **Variables** and **Expressions** views, the **Send to** submenu has commands that add the selected registers to an existing or new **Expressions** view.

Also like the **Variables** and **Expressions** views, you can use a context menu by right-clicking on the column headings to choose which columns are displayed.

As mentioned earlier, the **Registers** view has a Search () button for finding registers, and a drop-down menu () with **Refresh**, **Freeze Data** and **New Registers View** commands like the **Variables** and **Expressions** views do.

Debug Control view

The **Debug Control** view shows the connected debug configurations and possibly some disconnected ones. In our case there is just one interesting configuration, the **`gnometris`** configuration that we created earlier. If you remove a debug configuration from the Debug Control view, you can get it back either by going to the Debug Configurations dialog or by using Add Configuration (without connecting)... in the drop-down menu () of the Debug Control view.

Call Stack

At the first level underneath the debug configuration are the threads; Gnometris has just one. Under the thread are the frames of the call stack which represent the executing functions. The currently executing function is listed first followed by its caller then its caller's caller and so on.

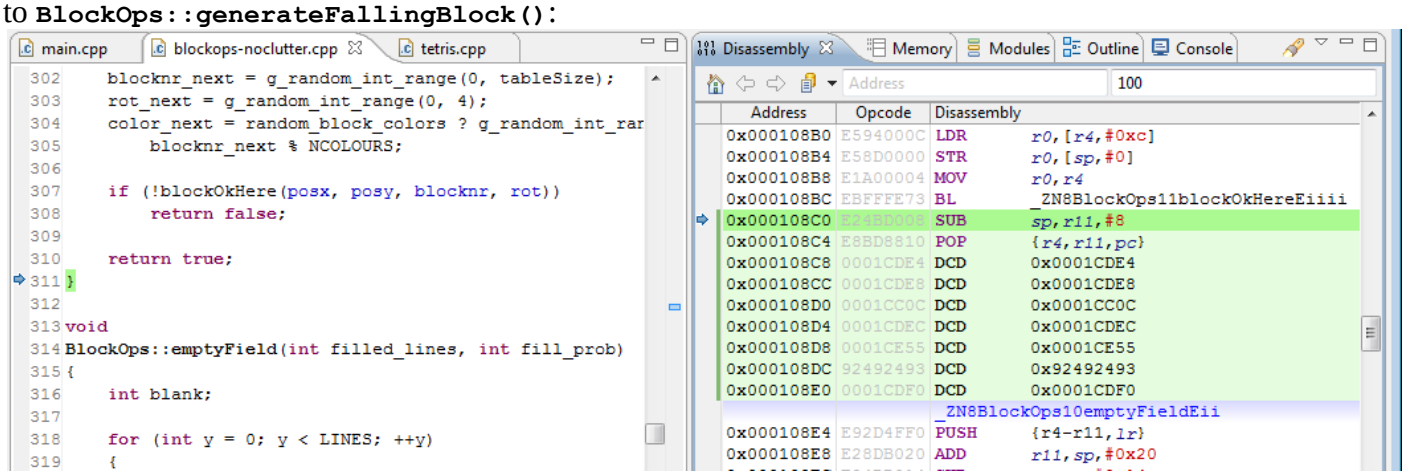
The current configuration, thread and function frame are highlighted in green. It is possible to change the debugger's "focus" to a different configuration, thread or function frame by clicking on it.

⇨ Click on the **`BlockOps::generateFallingBlock()`** frame. The source and **Disassembly** views update to show where execution will resume when control returns to the selected frame. The **Variables** view updates to show the local variables of the selected frame. The values in the **Expressions** and **Registers** views also change to show the value they will have when control returns to the selected frame.

If there are no symbols loaded for a particular level in the call stack then the debugger will not be able to show the function name or the caller – this is why the last level is just an address. Later we'll see how we can load the symbols for a specific shared library.

We can use the Step Out command to run the application until the current top frame, **`BlockOps::blockOkHere()`**, returns to its caller

⇨ Click the Step Out button (; **F7**); the application runs until **`BlockOps::blockOkHere()`** returns to **`BlockOps::generateFallingBlock()`**:



The **`BlockOps::blockOkHere()`** frame is gone from the call stack. Since the return value from a function is in register R0, we can use the **Registers** view to see that **`BlockOps::blockOkHere()`** has returned a value of **`true`** (1) which is also the value that **`BlockOps::generateFallingBlock()`** is going to return.

You can step out of more than one frame by right-clicking on the frame you want to step out to and choosing **Step Out to This Frame**.

Play the game

⇨ Click on the Continue button ( ).

This lets the game run until it hits another breakpoint (or quits). The blocks will start falling. Play the game for a bit. You can move the falling block left or right using the arrow keys. The down-arrow key will make the block drop quickly. The up-arrow key causes the block to rotate and when you press it the game will hit the breakpoint we set on **`BlockOps::rotateBlock()`**.

Let's disable the two breakpoints so we can play without interruption for a bit. We need the target to be stopped to change the breakpoints.

⇨ Stop the game. There are three ways to do this (which may have already happened)

1. Type up-arrow; this will cause the game to stop at the **`BlockOps::rotateBlock()`** breakpoint.
2. Wait until the block stops falling; this will cause the game to begin to generate a new block and stop at the **`BlockOps::generateFallingBlock()`** breakpoint.
3. Click the Interrupt button ( ) in the **Debug Control** view.

⇨ Click the Skip All button ( ) in the **Breakpoints** view. Alternatively, you could disable both breakpoints by unchecking them individually. You can't skip or disable the debugger internal breakpoints.

⇨ Click on the Continue button ( ) to continue the game.

Play the game until you're bored and want to continue debugging.

⇨ Stop the game by pressing the Interrupt button ( ) in the **Debug Control** view. This will stop the game. Since Gnometris spends most of its time sleeping, you will probably stop it in the C library (**`libc.so.6`**).

⇨ Reenable the breakpoint on **`BlockOps::rotateBlock()`** and disable the breakpoint on **`BlockOps::generateFallingBlock()`**.

⇨ Click on the Continue button ( ) to continue the game.

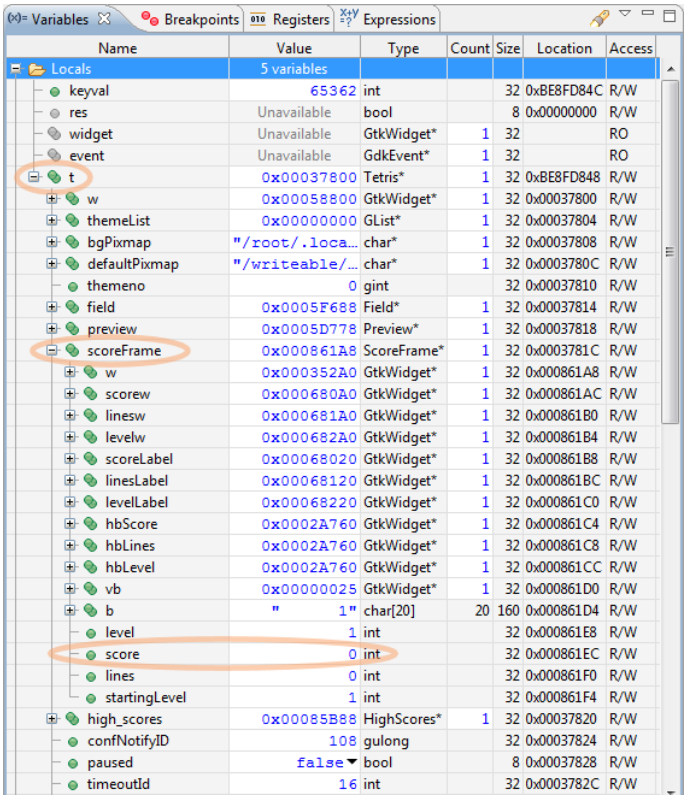
⇨ If your game has ended ("Game Over"), choose **Game** > **New Game** (Ctrl+N) in the Gnometris window to start a new one.

⇨ When a block is falling, type up-arrow to the game so that it stops at the **`BlockOps::rotateBlock()`** breakpoint.

Gaming the game

Let's give ourselves some points so that we can impress our non-debugging friends.

⇨ Click on the **`Tetris::keyPressHandler()`** frame in the call stack. This frame has a local variable, **`t`**, that is a pointer to the C++ object that represents the current game. Find **`t->scoreFrame->score`** in the **Variables** view

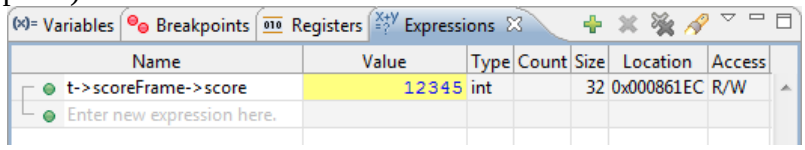


⇨ Click on the **`score`** value and type a new value. The new score will not show up on the Gnometris display immediately. It will be displayed the next time the score is redrawn. (The changes to the Gnometris code mentioned earlier cause the score to be redrawn every time a block moves down.)

Expressions view

The **Variables** view can get a bit crowded with variables that we're not interested in. We can use the **Expressions** view to display just the interesting expressions.

⇨ In the **Expressions** view enter the expression **`t->scoreFrame->score`**. There are a couple ways to do this. You can select **`score`** (the whole row) in the **Variables** view; then right-click and choose **Send to > Expressions View**. Alternately you can click where it says Enter new expression here then type the expression (or you can click the Add New Expression button ( ) to move the insertion point to the same place).



You can use registers in the expressions by using names like **`$r4`**. You can also create expressions by dragging a selection from other views such as source, **Variables**, **Registers** or **Memory** and dropping it into the **Expressions** view.

Like the **Variables** view, you can select one or more expressions and right-click on them to get a context menu that will let you change the format that the value is displayed in. Also like the **Variables** view, you

can also use **Show in Registers**, **Show in Memory** and **Show Dereference in Memory** commands using context menu.

Also like the **Variables** view, you can choose which columns are displayed in the **Expressions** view by right-clicking in the column headers. The **Expressions** view has a Search ( ) button, and a drop-down menu ( ) with **Refresh**, **Freeze Data** and **New Expressions View** commands like the **Variables**, **Registers**, **Disassembly** and **Memory** views do.

⇨ Click on the Continue button ( ) to continue the game.

The new score is displayed in the Gnometris window as soon as the block moves down one step.

⇨ Type up-arrow to the game so that it stops at the `BlockOps::rotateBlock()` breakpoint again.

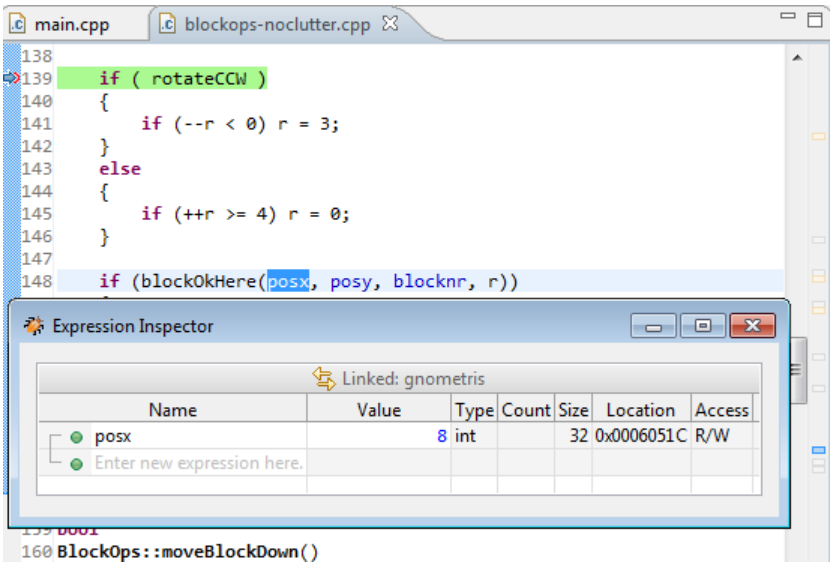
Since `BlockOps::rotateBlock()` does not have a variable named `t` in scope the value expression cannot be shown.

You can select expression rows and click the Remove Selected Expressions ( ) button to remove them or you can click the Remove All Expressions ( ) button to remove all of the expressions.

Expression Inspector view

There is a temporary version of **Expressions** view called the **Expression Inspector** view. You can use the **Expression Inspector** view from the source views while debugging to quickly inspect some values.

⇨ Select an expression in the source view, for example posx and right-click and choose **Inspect** from the context menu.

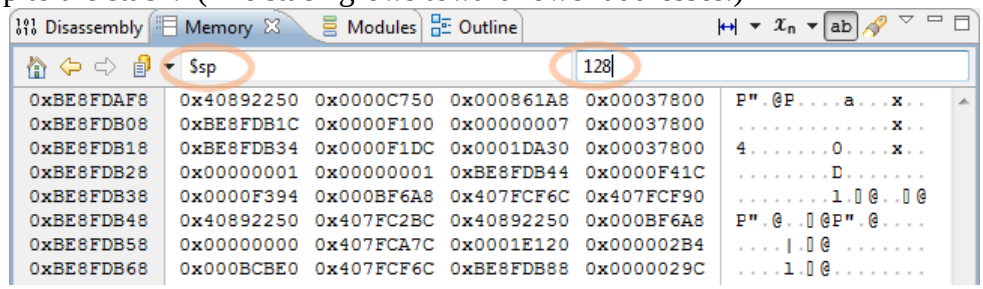


You can do most of the same operations in the **Expression Inspector** view that you can do to an **Expressions** view including adding more expressions, changing the columns and formatting, etc. If you want to save some expressions that you've added to an **Inspector** view, select them and right-click and choose **Send to** which will add them to an existing or new **Expressions** view.

Memory view

The **Memory** view displays memory in various formats. It has **Address** and **Size** fields. The **Address** field is an expression for the starting address to be displayed and **Size** is an expression for the number of bytes to be displayed.

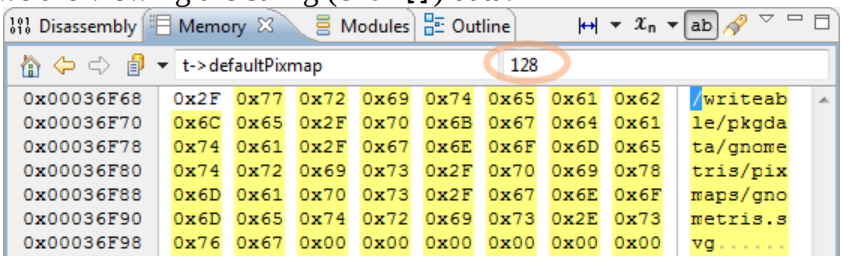
⇨ Put `$sp` in the **Address** field of the **Memory** view and put `128` in the **Size** field. Now we're viewing the top to the stack. (The stack grows toward lower addresses.)



The memory address of the first byte in each row is displayed in the left margin. The width of the view determines how many bytes are displayed on each row.

You can set the **Address** field by dragging a selection from another view such as source, **Variables** or **Expressions** and dropping it into a **Memory** view.

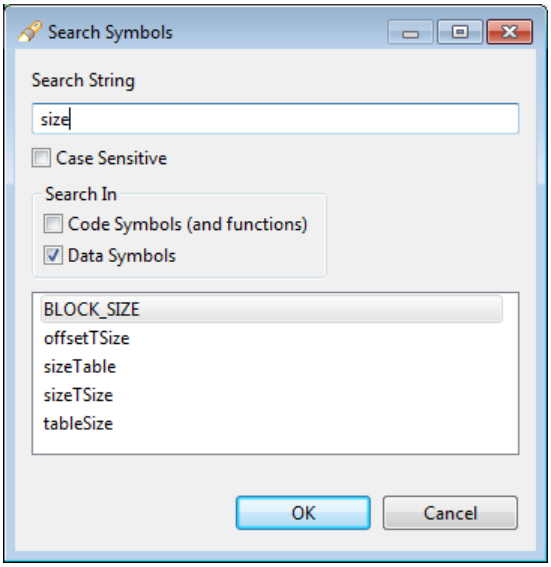
⇨ Use the call stack and the **Variables** view to find the `t->defaultPixmap` member of the Local variables in the `Tetris::keyPressHandler()` frame. Select the member's row in the **Variables** view, right-click and choose **Show Dereference in Memory** and change the **Memory** view's **Size** field to `128`. Now we are viewing the string (`char[]`) data.



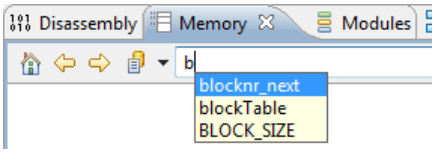
The memory values are displayed both numerically and as characters; either can be changed.

⇨ It won't hurt the game to change the beginning of this string. Select the first `/` character and type a few characters (for example "hello") to see how it works. Note that the changed values also show in the **Variables** view.

You can use the Search button () to open a search dialog like the **Variables Expressions**, **Registers** and **Disassembly** views that makes it easy to see the memory holding a global variable if you know part of its name.



You can type the first few characters (or none) into the **Address** field and then type Ctrl+Space to use Content Assist to see a list of variables whose names being with those characters which you can then pick from



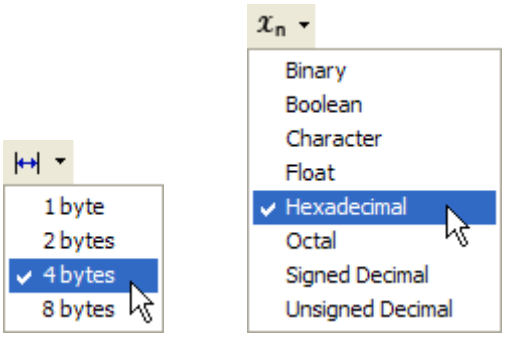
INFO: In Windows, if foreign language support is enabled, the *Ctrl+Space* key combination is used to change between languages inside a text box (for example English to Chinese). You can change the key combination to something else in Eclipse by going to **Window > Preferences > General > Keys > Content Assist** to change the key combination to something else.

The addresses (and sizes) you use are kept in a history list. You can use the Back ( ) and Forward ( ) buttons to walk through the history list or use the History ( ) drop-down menu to choose a recent address value. You can use the Home/Clear Data ( ) button to empty the **Address** field and show no data, which is the default. The **Memory** view has a drop-down menu ( ) that has a **Clear History** command that clears the history of the Address field. The **Disassembly** view also has **Address** and **Size** fields and a history list.

If you haven't already used them, you can try using the **Show in Memory, Show Memory Pointed to by *register*** commands from the context menus of the **Variables**, **Expressions** and **Registers** views to examine data in a **Memory** view.

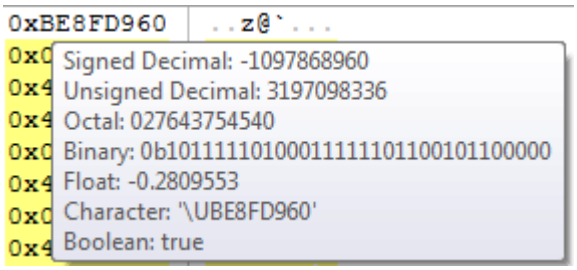
You can also select an expression in a source view, right-click and choose **Show in Memory** or **Show Dereference in Memory** from the context menu.

The **Memory** view has drop-down menus to change the number of bytes in each value (width) and the format of the numeric values.



It won't hurt anything if you try changing the width and the format to see what happens. The **Memory** view has a button ( ) to toggle the display of the characters on and off. You can try it, too.

If you hover the mouse pointer over a value in the **Memory** view a tooltip will show you the value in other formats.



You can toggle these tooltips off or on with the **Show Tooltips** command in the **Memory** view's drop-down menu ( ).

If the **Memory** view is displaying instruction memory where a breakpoint is set, the breakpoint symbol will be shown ( ). You can also right-click and toggle, enable and disable breakpoints on memory locations (these are execution breakpoints though, not data watchpoints).

The **Memory** view's drop-down menu ( ) has **Import Memory** and **Export Memory** command that you can use to read and write a region of memory to a file in various formats. The **Memory** view's drop-down

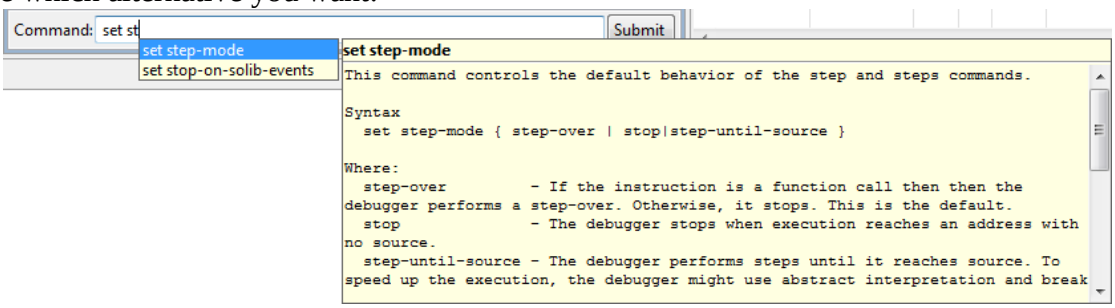
menu also has **Refresh**, **Freeze Data** and **New Memory View** commands like the **Variables**, **Expressions**, **Registers** and **Disassembly** views do.

Commands and History views

You have probably noticed that the debugging actions that you've been doing have been recorded in the **Commands** view along with responses. The commands are also recorded in the **History** view, without the responses. The debugger can be controlled by typing commands into the **Command** field of the **Commands** view and pressing Return or clicking the **Submit** button.

While you are typing a command you can get help with what you have typed so far by pressing Ctrl+Space to activate Content Assist

- Type **`set st`** into the **Command** field and then type Ctrl+Space. This activates the Content Assist which shows the possible completions and help for each. You can use the mouse and arrow keys to choose which alternative you want:



  In this case, we don't want any of them so just press **Esc** to dismiss the assist windows.

You can type the **`help`** command to find out about all the DS-5 Debugger commands. Pressing the **F1** key (or Shift+**F1** on Linux) will also display Help for whichever view you are using at the time.

Let's "improve" our Gnometris playing further, this time using the debugger command line.

- Reenable the breakpoint on **`BlockOps::generateFallingBlock()`** and run the game until it stops there.

This function uses a member, **`blocknr`**, that determines the shape the new block. I like the straight blocks which Gnometris represents by the value **`5`**

- Execute the command **`set var blocknr = 5`** (if you like the two-by-two blocks better you can use the value 6 instead).
- Click on the Continue button () to continue the game. Now the new block has the shape you forced.

You can re-execute the last command by typing Return or clicking the **Submit** button. You can also use up-arrow and down-arrow to easily access previous commands.

You can also drag selections from the other views such as source, **Variables**, **Memory**, **Project Explorer** and even Windows Explorer to the **Command** field to construct commands.

The **Commands** view has buttons that let you save, clear, and lock the scrolling of the view. You can use the Show History button () to open the **History** view. You can use the Run Script drop-down menu () to run your recent scripts, favorite scripts or any scripts.

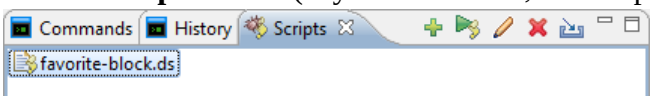
You can select lines of the **Commands** or **History** views and right-click on them to get a context menu containing **Save selected lines as a script...** and **Execute selected lines** commands.

The **History** view has an Export button () that creates a script from (exports) the selected commands. It also has buttons that clear the view () and lock the scrolling () of the view.

Scripts view

You can collect debugger commands into script files and easily re-execute them from the **Scripts** view.

- ⇨ Select the `set var blocknr = 5` command that you executed above in either the **Commands** view or the **History** view.
- ⇨ Right-click on the selected line and choose **Save selected lines as a script...**
- ⇨ A file save dialog will open. Save the file on the desktop with the name **favorite-block**. And choose **Yes** to the dialog asking if you want to save the script in the **Scripts** view. (If you choose **No**, the script file is still created on disk and you can use the Import Script button ( ) of the **Scripts** view to add it.)



You can run a script in the **Scripts** view by double clicking it or by selecting it and clicking the Execute button ( ) at the top of the view. You can also run scripts using the Run Script drop-down menu ( ) in the **Commands** view.

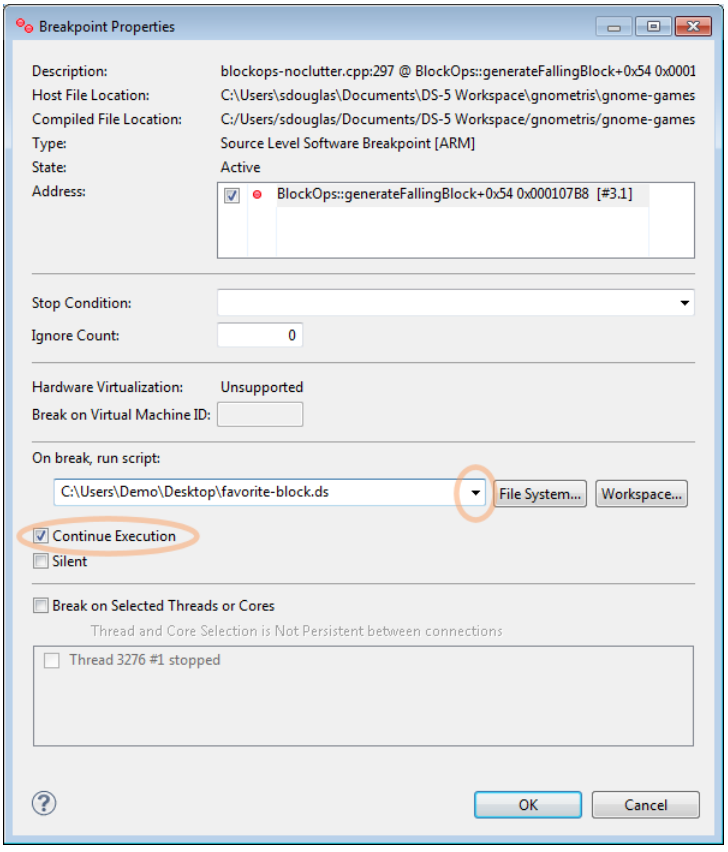
You can easily open a script to edit it by selecting it and clicking the Edit button ( ). You can create a new empty script and open it for editing by clicking the New script button ( ). If you already have a script file, you can add it to the **Scripts** view by clicking the Import Script button ( ) and finding it. You can delete scripts by selecting them and clicking the Delete button ( ).

You can also create scripts by pasting into the **Scripts** view or dragging selections into it from other views such as **History**, **Commands** and source.

Breakpoint scripts

We can do complex debugging by executing a script every time a breakpoint is hit. Let's automate forcing our favorite block.

- ⇨ Wait until the game hits a breakpoint, or type up-arrow to encourage it to hit a breakpoint.
- ⇨ Right-click on the breakpoint on `BlockOps::generateFallingBlock()` in the **Breakpoints** view and choose the **Properties...** context menu item. Use the drop-down menu to choose the **favorite-block.ds** script file (on the desktop, remember?).
- ⇨ Check **Continue Execution** and click the **OK** button.



Now whenever this breakpoint is hit, DS-5 will execute the favorite-block script and then continue execution. You can check **Silent** if you want the breakpoint to not display messages in the **Commands** view when it is hit.

- ⇨ Disable the breakpoint on `BlockOps::rotateBlock()` (the other one) so that it won't stop the game.
- ⇨ Click on the Continue button ( ) to continue the game. Now see how well you can play!

Show off your skills

You may have noticed that Gnometris's preview pane is still showing the next block that was chosen before our breakpoint script forced it.

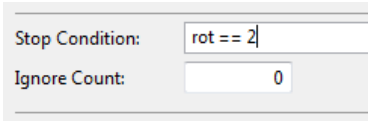
⇨ Prove that you understand what's going on by using the DS-5 Debugger to make the Gnometris preview pane also show your favorite block. Maybe you want to force the block to be your favorite color, too.

Advanced Breakpoints

There are some additional features available in the **Breakpoint Properties** dialog.

You can set a **Stop Condition** expression for a breakpoint. Every time the application reaches the breakpoint the condition expression is evaluated. If the expression's value is zero (false) the application is resumed without performing any other breakpoint actions; the ignore count is not updated and any breakpoint script is not executed. If the expression's value is non-zero (true) the breakpoint is processed normally, that is the same as if there were no condition expression.

⇨ Enable the breakpoint on `BlockOps::rotateBlock()` and in its **Breakpoint Properties** set the **Stop Condition** to `rot==2`. This means the breakpoint will be ignored except for one of the four possible rotations. (Some of the rotations look the same as other rotations for some shapes.)



⇨ Click on the Continue button () to continue the game.

⇨ Now type up-arrow repeatedly to rotate the block and see that the debugger stops only every fourth time.

You can also set an **Ignore Count** for a breakpoint. This causes the breakpoint to be ignored a given number of executions after which the breakpoint will begin stopping each time.

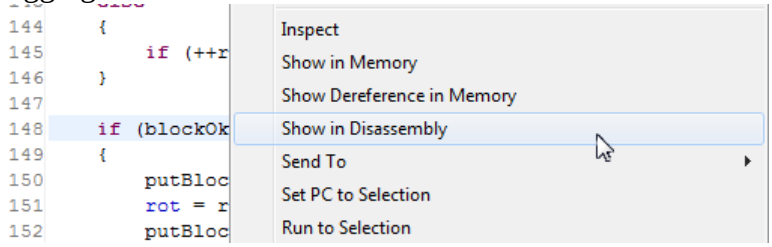
You can also access the **Breakpoint Properties** dialog by right-clicking on a breakpoint symbol (●) in a source view and choosing **DS-5 Breakpoints** > **Breakpoint Properties...** or right-clicking on a line with a breakpoint symbol (●) in **Disassembly** view and choosing the **Breakpoint Properties...** menu item.

You find out the purpose of a debugger internal breakpoint by right-clicking on it in the **Breakpoints** view and choosing the **Properties...** menu item.

More features in Source and Disassembly views

You can hover the mouse over the tab of a source view (e.g. **blockops-noclutter.cpp**) and a tooltip will show you the path to the file within the **Project Explorer** view.

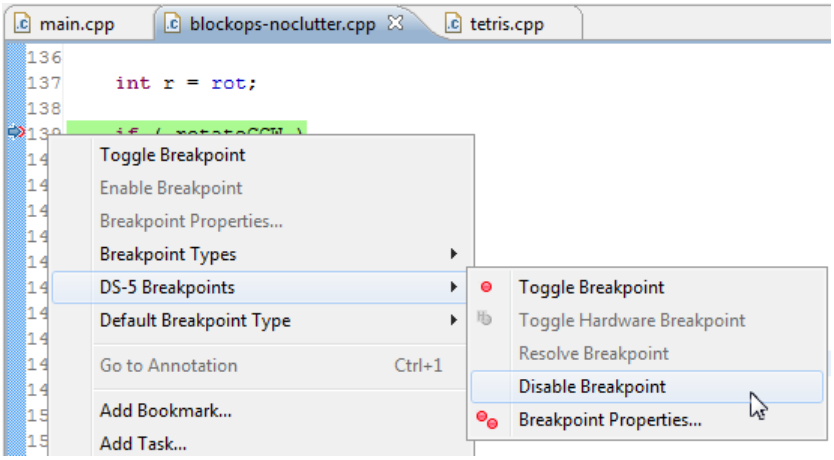
In a source view you can set the insertion point on a line of code and then right-click to get an enormous context menu. Most of the context menu items are for use when writing code but the following are used while debugging with DS-5:



- You can use the **Show in Disassembly** command to show the assembly instructions that correspond to the line with the insertion point in the **Disassembly** view. This also sets the **Address** field of the **Disassembly** view.
- Like in other views, you can use the **Send to** submenu to add the currently selected text to an existing or new **Expressions** view.

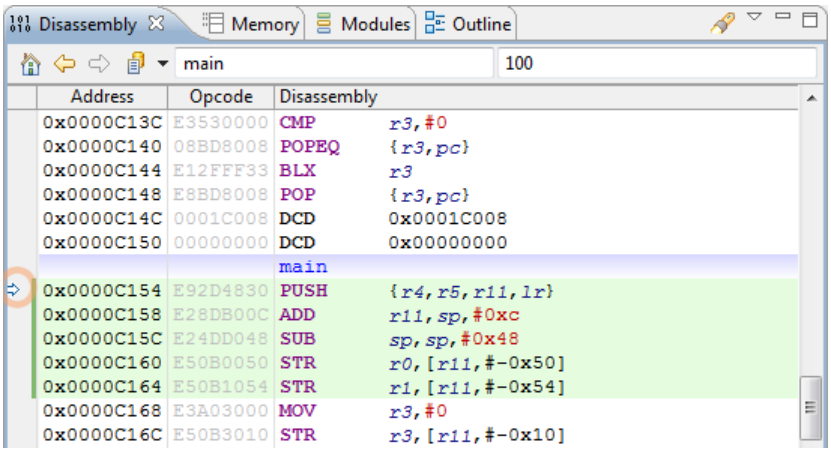
- You can use the **Set PC to Selection** command to set the PC register to the first assembly instruction for the line with the insertion point. This can be can be very handy for skipping some code or re-executing a bit of code that just did something unexpected—but you need to know what you are doing.
- You can use the **Run to Selection** command to place a temporary breakpoint on the line with the insertion point and then run until it is hit (or another breakpoint is hit).
- You can also use these commands which were discussed earlier: **Inspect**, **Show in Memory**, **Show Dereference in Memory**.

In addition to double-clicking on the left margin of a source line to toggle a breakpoint, which we've seen earlier, you can toggle, enable and disable breakpoints and access their properties by right-clicking on the left margin of a source line



In the source views when you hover the mouse pointer over an identifier, a tooltip will be displayed showing the value of a variable, the documentation for a library function or the source code of the declaration of the identifier. When you put the insertion point in a function, the left margin is highlighted to show the extent of the function (shown in the picture above). When you put the insertion point in an identifier, for example a function or variable name, all occurrences of the same identifier in the current file are highlighted. You can then right-click and choose **Find Declaration** (or type **F3**) to go to the definition or declaration of the identifier.

Like the **Memory** view, the **Disassembly** view has an **Address** (on the left) and **Size** (on the right) fields. You can type an expression (for example `$r0` or `0xC100`) or a symbol (for example `main`) in the **Address** field to disassemble around a specified location. A hollow arrow ( )indicates the line specified in the **Address** field:



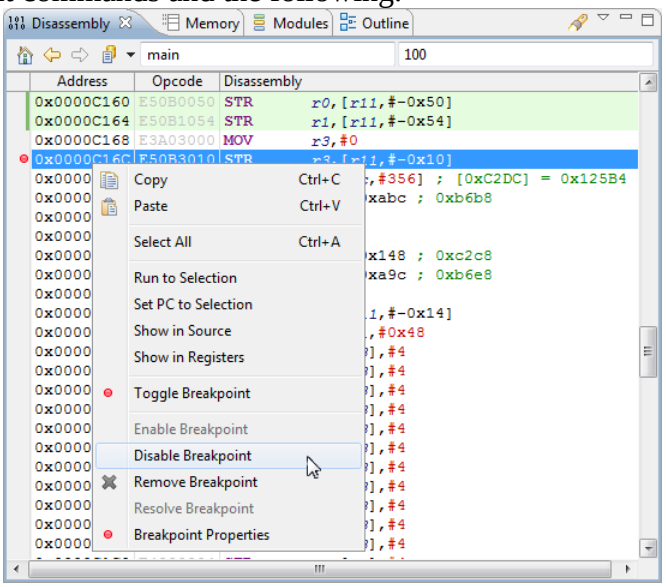
The green highlighting shows the instructions that belong to the same source line.

Like the **Memory** view, you can type the first few characters (or none) into the **Address** field and then type Ctrl+Space to use Content Assist to see a list of code symbols (functions) whose names being with those characters which you can then pick from. Also like the **Memory** view, the **Disassembly** view has a Search

button ( ) to open a search dialog that makes it easy to see the instructions of a function if you know part of its name.

The **Size** field, which defaults to `100`, specifies how many instructions before and after the address should be scrollable. You can use the Home ( ) button to set the **Address** field to the default **<Next Instruction>** which centers the **Disassembly** view around the PC. You can use the Back ( ) and Forward ( ) buttons to walk through the address history or use the History ( ) drop-down menu to choose a recent value.

In the **Disassembly** view you can right-click on an assembly instruction to get a context menu including the breakpoint commands and the following:



- The **Run to Selection** and **Set PC to Selection** commands do the same thing as they do in source views.
- You can use the **Show in Source** command to select the source code line that corresponds with the assembly instruction.
- You can use the **Show in Registers** command to highlight the registers accessed by an assembly instruction in the **Registers** view.

When you hover the mouse pointer over an assembly instruction in a **Disassembly** view, a tooltip will display if that address is pointed to by the PC, LR, or has a breakpoint.

The **Disassembly** view has a drop-down menu ( ) that contains an **Instruction Set** submenu that allows you to force the instructions to be disassembled as ARM or Thumb. Like the **Memory** view, the **Disassembly** view's drop-down menu also has a **Clear History** command that clears the history of the **Address** field.

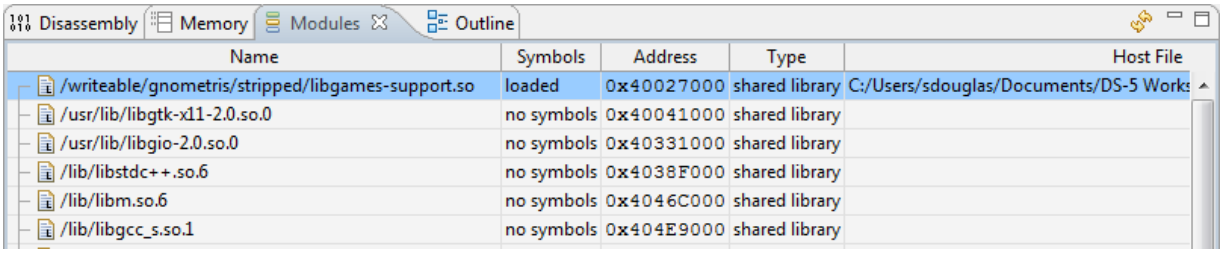
The **Disassembly** view's drop-down menu ( ) also has **Refresh**, **Freeze Data** and **New Disassembly View** commands like the **Variables**, **Expressions**, **Registers** and **Memory** views do.

Shared Libraries and Modules view

The DS-5 Debugger can debug shared libraries (`.so` files) in pretty much the same way as applications. You can step into and out functions in a shared library just as if they were in the main application and the call stack will show which frames are from which shared libraries.

Gnometris uses a shared library named `libgames-support.so`. If you want to try out the shared library support, you can place a breakpoint on the function `games_scores_add_score`, in `libgames-support/games-scores.c`, in the usual ways. You can see information about the shared libraries that are in

use in the **Modules** view:



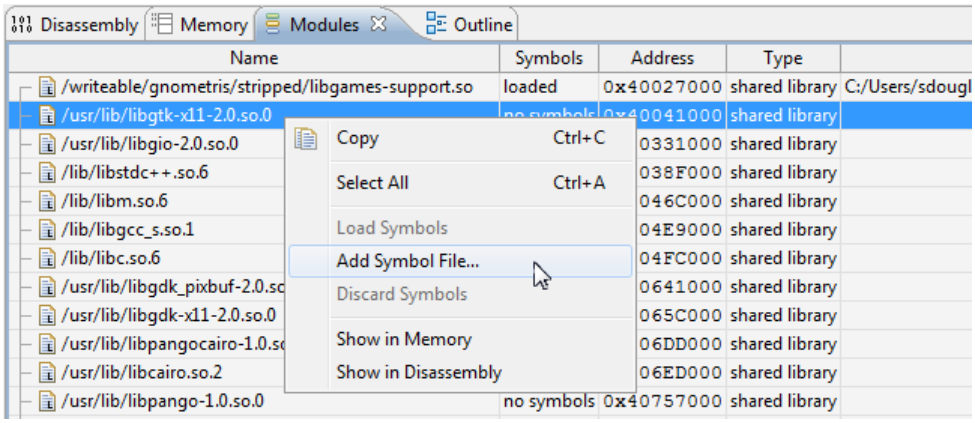
Like the **Variables**, **Expressions** and **Registers** views, you can control which columns of the **Modules** view are displayed by right-clicking on the column headings.

If you set a breakpoint in shared library code before the shared library has been loaded by the application then the breakpoint will be kept as "pending". When the library is loaded the debugger will search for the shared library's debug information according to the **`solib-search-path`** which can be set in the debug configuration or using the command line. When the debug information has been loaded the pending breakpoint will be resolved.

Because Gnometris loads **`libgames-support.so`** before **`main`** is reached you will not see breakpoints in **`libgames-support.so`** in the pending state unless you change the debug configuration to use **Connect only** or **Debug from entry point**.

You can also right-click on an entry in the **Modules** view to get a context menu with commands for loading and discarding symbols and displaying the module in a **Memory** or **Disassembly** view.

⇨ Right-click the entry for the C library, **`/lib/libc.so.6`** and choose **Add Symbol File...** then navigate to **`My Documents\DS-5 Workspace\distribution\filesystem\armv5t_mtx\lib`** and choose **`libc.so.6`**



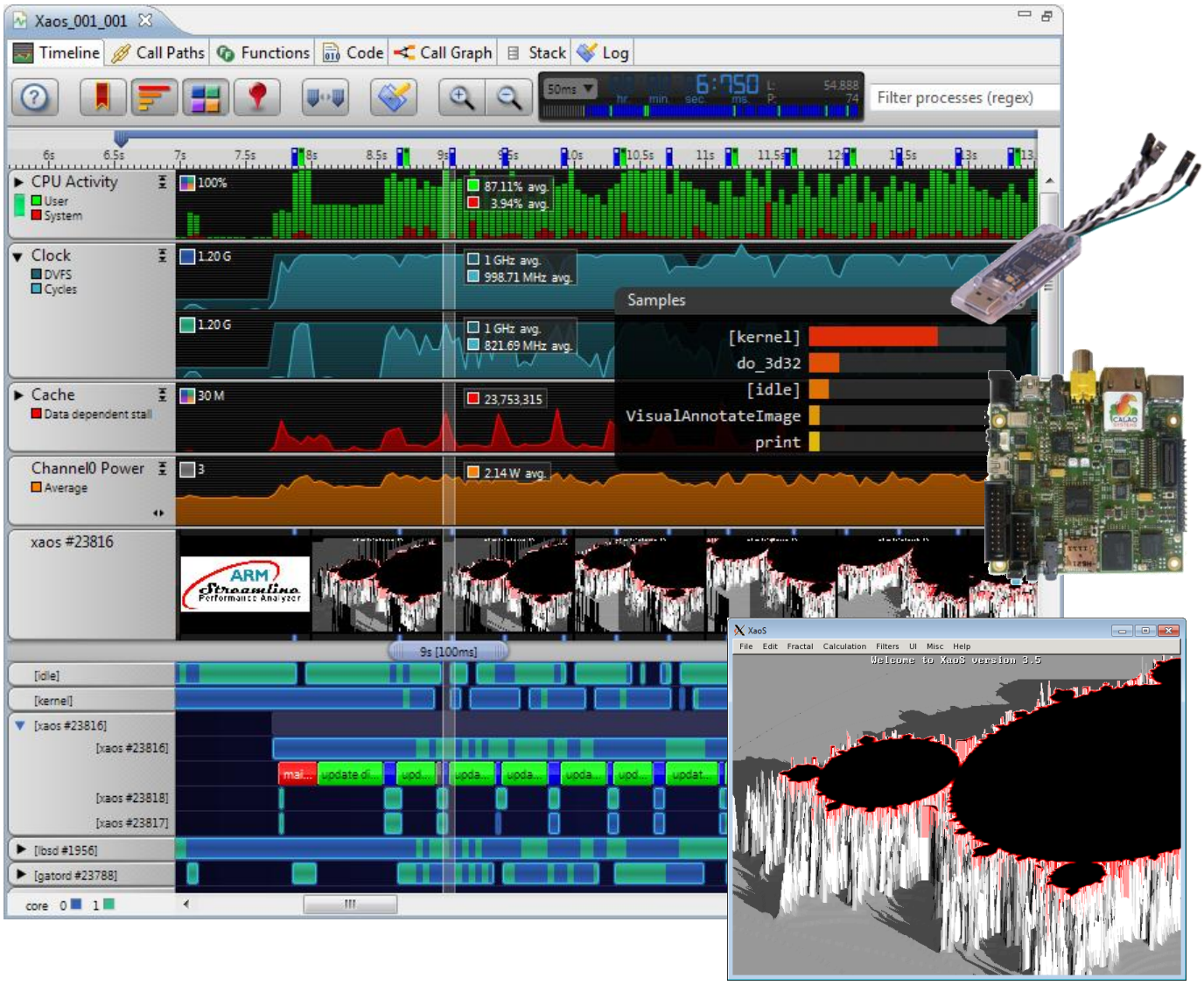
This copy of **`libc.so.6`** was not built with debug info, and the source is not in the example, but there is still enough information to use C library symbols, for example **`memcpy`**, and the call stack will now show C library symbol names instead of just addresses.

If you wanted, you could add **`My Documents\DS-5 Workspace\distribution\filesystem\armv5t_mtx\lib`** to the **Paths** in the **Debugger** pane of the debug configuration as a **Shared library search directory**. Then the debugger would find the symbols for the libraries automatically.

Change of topic

Let's change focus now from debugging to profiling and take a look at ARM Streamline. There are a few more debugging topics that we'll come back to after Streamline, if there's enough time and interest.

ARM Streamline Workshop using Xaos on Snowball

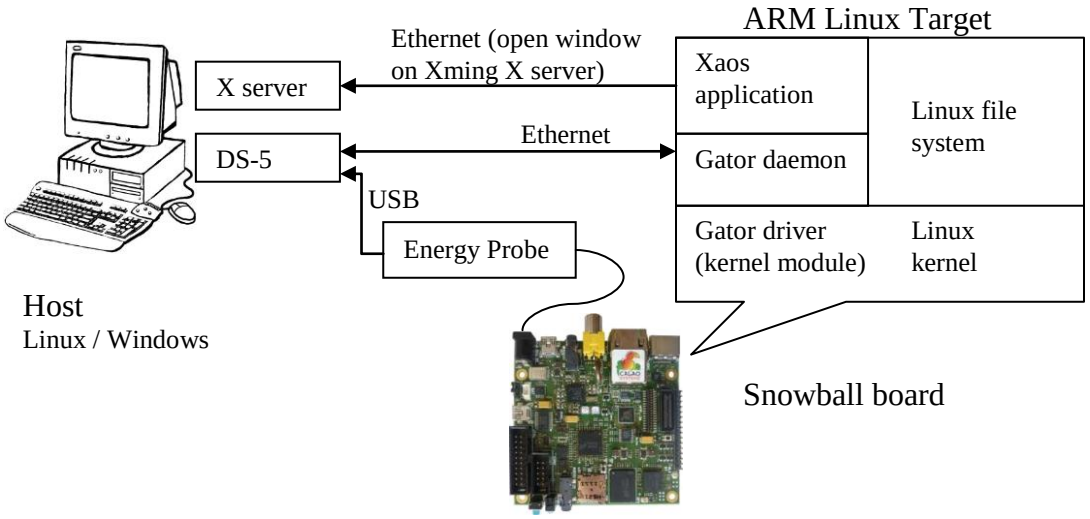


ARM® Streamline™ is a graphical performance analysis tool. Combining a Linux kernel driver, target daemon, and an Eclipse-based user interface, it transforms sampled data and system trace into reports that present the data in both visual and statistical forms. ARM Streamline uses hardware performance counters along with kernel metrics to provide an accurate representation of system resource use.

This part of the workshop demonstrates the use and features of the ARM Streamline performance analyser to inspect applications running on an ARM Linux target. The workshop details the Linux configuration and setup required to enable application profiling. It demonstrates how use the Streamline report to analyse and investigate application performance and power. All of the software you need, including the ARM Linux image for the target, is included with DS-5.

Introduction to ARM Streamline

The ARM Streamline product uses sample based techniques to capture system and application data for analysis. The data is then analyzed by the DS-5 Eclipse Streamline plug-in and presented as a report.



The kernel driver (called **`gator.ko`**) together with a daemon running on the target (called **`gatord`**) captures the data and sends it over the network to the host PC for analysis. The kernel module must be built against the Linux kernel in use on the target.

Preparation

For the purpose of this workshop we are going to use an application called Xaos. It is an interactive fractal zoomer included as an example in DS-5.

Host Setup

DS-5 can be used on Windows and Linux hosts. If your host has not already been setup for you then you will need to follow the instructions in the appendix on page 87

⇨ If it's not already started, start the Xming X server. (Refer to Starting the X server on the Host on page 33.)

Target Setup

For the workshop you'll be supplied with a Snowball board that already has gator running on it. The gator driver and gator daemon are open source and are provided with DS-5. If you want to setup your own target please see *Appendix B: Snowball*: *Linaro Linux target setup* page 94. (If you want, you can connect to the target's serial port to see the target's console messages, but it is not necessary for this workshop. See the *Serial setup* in the appendix on page 92. We will, however use the different, but confusingly similar, **Terminals** view to make an ssh connection later.)

Starting Eclipse

If you have not started Eclipse for DS-5 yet, follow the steps below

⇨ **Start > All Programs > ARM DS-5 > Eclipse for DS-5** and choose a location for the workspace where Eclipse projects will be stored. The default workspace location is fine.

⇨ If this is the first time you've started DS-5 you will see the "Welcome to ARM DS-5" home page; click **Go to the workbench**. You can get the **Welcome** page back if you want it later by choosing **Help > Welcome**.

Installing a DS-5 License

In order to use DS-5 for this workshop you will need to have a license for DS-5 Basic Edition, either a full or evaluation license. (A license for Professional Edition will also work.) If you don't have a license a dialog will open with an explanation and a button to **Open License Manager...**. Without a license you will

see only some of the Streamline panes and information. You can get an evaluation or full license by opening the license manager using **Help > ARM License Manager...**, clicking the **Obtain License...** button and following the instructions. You will need to be connected to the internet. When you receive the license file, you can add it by using the **Add License...** button. After adding a license you will need to restart Eclipse in order to fully enable Streamline (sorry).

Importing Xaos

We use the two example projects **distribution** and **xaos**.

⇨ Look in the **Project Explorer** view and if **distribution** and **xaos** do not appear there, follow the instructions for importing them in the appendix on page 87.

The build produces an application `/xaos/xaos-3.5/bin/xaos` in the **Project Explorer** view. This file will contain debug information. The **stripped** subdirectory, `/xaos/xaos-3.5/bin/stripped`, has a copy of `xaos` with the debug information removed. The project contains pre-built copies of these two files which will be overwritten when you build the project.

Getting help for Streamline (digression)

You can press the **F1** key (or Shift+**F1** on Linux) at any time to get help about the current view. You can use **Help > Help Contents > ARM DS-5 Documentation > Using ARM Streamline** to view the documentation. You can access cheat sheets for various tasks using **Help > Cheat Sheets > ARM …**, for example importing the example projects. (But we won't be following those cheat sheets here.)

If you **don't have a real hardware target** you can analyze some pre-captured data to use for the rest of the workshop. If you do have a real target, skip this step and carry on with the workshop. To import a saved report go to the section Import, Export and Save Reports on page 82. Following the import you can skip to the section The Timeline view – a first look at the Streamline report on page 70.

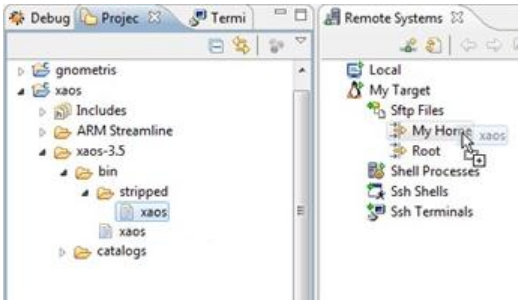
Install Xaos to the target

Unlike the DS-5 Debugger, ARM Streamline does not automatically copy applications to the target or start them. Instead, it profiles whatever is running on the target while it's capturing data. We'll copy Xaos onto the target ourselves and start it "by hand".

⇨ If it's not already configured, configure the RSE connection (Refer to Creating a Target Connection in the appendix on page 90.)

Now that we've established access to the target, we'll install the `xaos` application by copying it from the host;

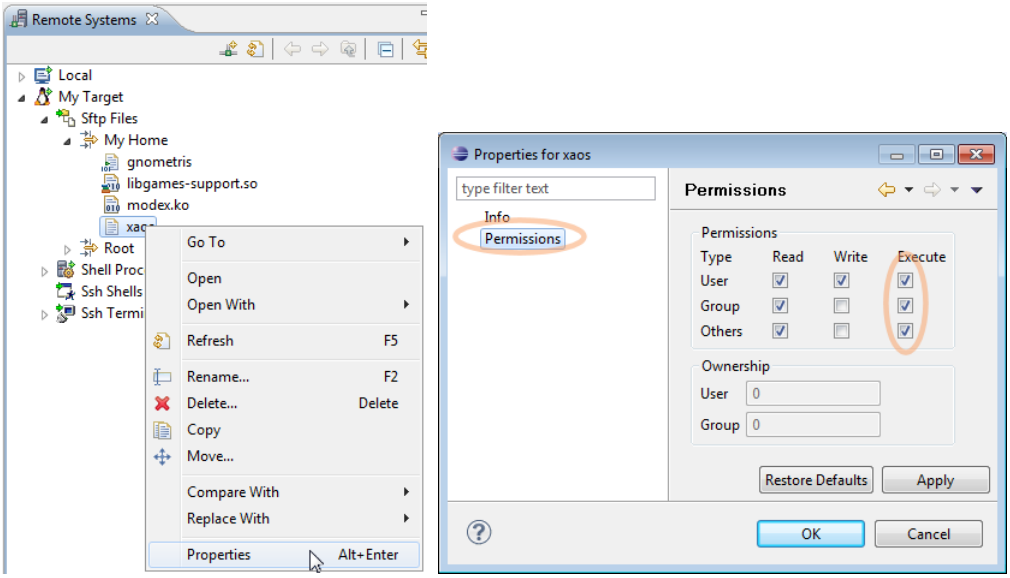
⇨ Drag the **Remote Systems** view by its tab to a different pane from the **Project Explorer** view so that you can see both views at the same time.



⇨ Copy the stripped `xaos` by dragging it from the `bin/stripped` folder in the project to the **My Target > Sftp Files > My Home** folder on the target. (If the connection has **Files** instead of **Sftp Files**, then the connection was not created correctly and you should **Disconnect** it, **Delete** it and recreate it.)

You can copy the unstripped version from the `bin` folder instead, but it's bigger and the debug information isn't needed on the target.

⇨ Expand the **My Home** folder and right-click on **`xaos`** on the target; choose **Properties** > **Permissions** and check the three **Execute** boxes:



Alternatively, you could set the execute bits by executing the following command in the **Terminals** view:

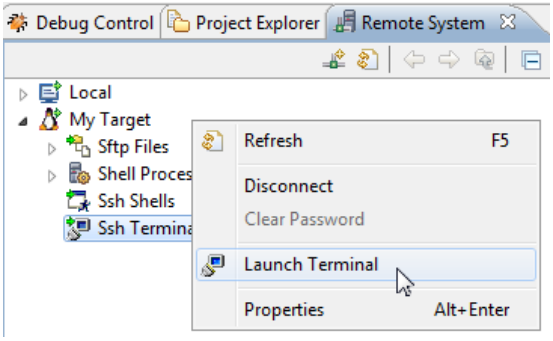
```
chmod +x /home/linaro/xaos
```

⇨ You should also copy **`splash.bmp`** by dragging it from the top-level **`xaos`** project folder to the **My Target** > **Sftp Files** > **My Home** folder on the target.

Error! Hyperlink reference not valid.

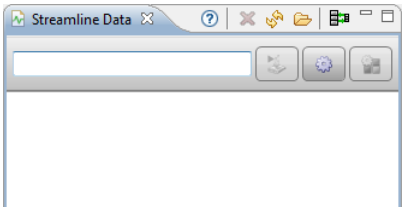
Now we'll create a terminal connection so that we can execute commands easily on the target. You can Collapse the **Sftp Files** to get them out of the way.

⇨ In the **Remote Systems** view, create a Terminal by right-clicking on **Ssh Terminals** and choosing **Launch Terminal**.



This will open a **Terminals** view in Eclipse that can be used to execute commands on the target. (This is different from, but confusingly similar to the **Serial** view).

Set the Capture Options



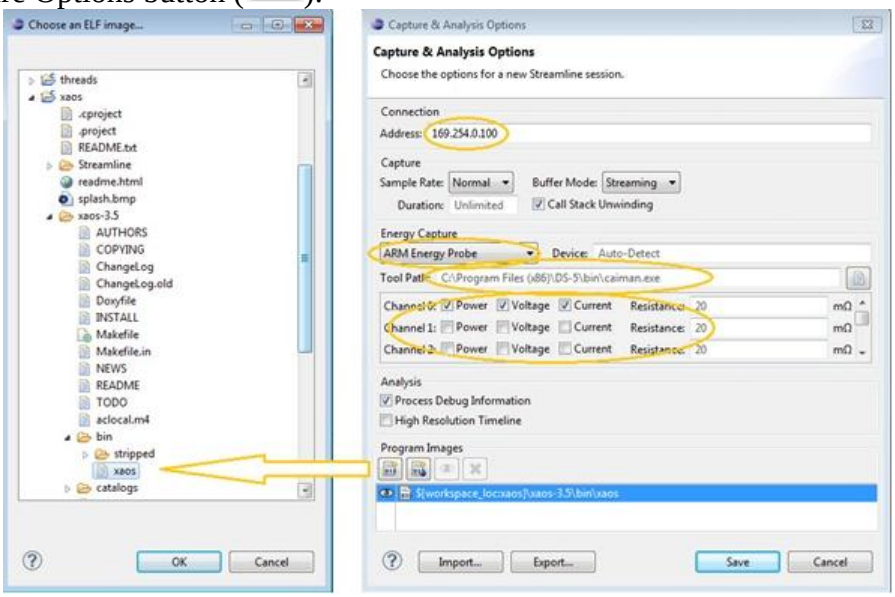
If it's not already open, open the **Streamline Data** view:

⇨ Select **Window > Show View > Streamline Data**.

Streamline data and reports are shown in the **Streamline Data** view. It will be empty, unless a previous profiling session has been run, in which case saved captures and reports may be present in the view. You can select any existing captures and reports and click the Delete button ( ) to get rid of them. You can get help by clicking the Show Help button ( ).

The **Streamline Data** view has an **Address** field for specifying the name or IP address of the target, a Start Capture button ( ), a Change Capture Options button ( ) and a Counter Configuration button ( ).

⇨ Click the Capture Options button ( ):



⇨ Set the **Address** field to the IP address of the target `169.254.0.100`.

- You can select **Sampling Rate** as **Normal** (1000 samples per second), **Low** (100 samples per second) or **None** (no sampling). We'll use **Normal** which is the default.
- You can set **Buffer Mode** to **Streaming**, **Large, Medium** or **Small**. In **Streaming** mode the captured data is regularly streamed from the target over the network to the host PC. It may skew the performance of any network critical applications. If **Buffer Mode** is **Large** (16MB), **Medium** (4MB) or **Small** (1MB), the data will be captured to a buffer allocated on the target. The profile session will be stopped when the buffer is filled up. Then data will then be passed to the host in one go. By default buffer mode is set to **Streaming**.

  Digression: If your target has limitations about capturing to the host PC (for example no networking), it is possible to capture to the target and then to manually copy the captured data to your host PC. See **Help > Help Contents > ARM DS-5 Documentation > ARM Streamline > Advanced Customizations > Capturing data on your target**

- You can use the **Duration** field to a maximum length of time to capture data in seconds or *minutes*:*seconds*. By default Streamline will capture data until you click the **Stop** button which we'll see later.
- If **Call Stack Unwinding** is checked and the program has been compiled with frame pointers in ARM state then Streamline will be able to capture information about call paths, call graphs and stack usage. Xaos is built this way, so leave **Call Stack Unwinding** checked.

⇨ If you are using an ARM Energy Probe, in the **Energy Capture** section you should choose **ARM Energy Probe**. Once the **ARM Energy Probe** is selected, the **Tool Path** will display the path for the caiman application if the DS-5 was installed in the default directory. (If DS-5 is installed in the non-default directory then, manually add the path to the **caiman** application. The **caiman** application can be found in the `. . .\DS-5\bin` directory.)

During a capture, Streamline will collect power information from the target using the energy probe and then display the power charts in the report. There are three channels available on energy probe and each channel gives us the information about the power consumption, voltage and current. Check the channels you have connected and if you wish to analyze the **Voltage** and/or **Current** then those need be checked as shown in the above figure. Power analysis is discussed in more detail later. See ARM Energy Probe setup in the appendix on page 99 for setup information.

- The **Analysis** section lets you specify whether Streamline should **Process Debug Information**. This will allow you to use Streamline to view source code. The **High Resolution Timeline** option enables you to zoom in more levels in the timeline (for example context switching). These checkboxes only control the defaults for the first report; they do not change the captured data.

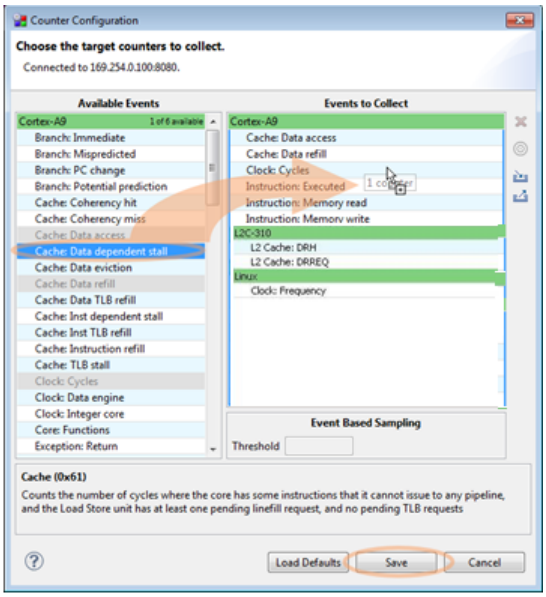
⇨ In the **Program Images** section, click the Add ELF image from Workspace... (second) button () and choose the `xaos` binary with debug information in the workspace from the `/xaos/xaos-3.5/bin` folder. The symbols button () next to binary name indicates that debug symbols will be loaded for the binary. You can also add more than one program to analyze at once. You can also add shared libraries, the kernel (`vmlinux`) and kernel modules (`.ko`) to the list.

⇨ Click **Save** to save the capture options.

You can use the **Export...** and **Import...** buttons to save and load favorite configurations to and from files.

Configure Counters

Advanced ARM processors have a performance monitor unit (PMU) in hardware. Exactly which events can be counted and how many events can be counted at once depends on which ARM processor is being used. The operating system allows access the performance counters for debug and profiling purposes. The Cortex-A9 in the Snowball board has a cycle counter and 6 configurable event counters.



Streamline defaults to capturing data from various counters in the PMU, kernel and L2 cache controller, depending on the target's hardware and kernel.

⇨ Click the Counter configuration button () to open the **Counter Configuration** dialog where you can examine and change the counters that you wish to capture.

You can hover your mouse over a counter to see a description of the event being counted.

You can add counters by dragging them from the **Available Events** list to the **Events to Collect** list. To see the Linux kernel and L2C-310 events you will need to scroll the **Available Events** list down. You can remove events from the **Events to Collect** list, so that you can count something else, by selecting them and clicking the Delete button (). The order in the **Events to Collect** list is fixed. We will see later how to reorder the display of the **Timeline** charts in the report.

⇨ If they are not already in the **Events to Collect** list, add these counters:

• **Cortex-A9/Cache: Data dependent stall**: to see when the processor is waiting on the L1 cache

• **L2C-310/L2 Cache: DRH** and **L2 Cache: DRREQ**: to see what the L2 data read hit ratio is. You will need to delete **L2 Cache: CO** first because the L2C-310 cache controller only allows two counters to be captured at once.

• **Linux/Clock: Frequency:** to the configuration to see how the kernel changes the maximum clock frequency that the core can run at (dynamic voltage and frequency scaling).

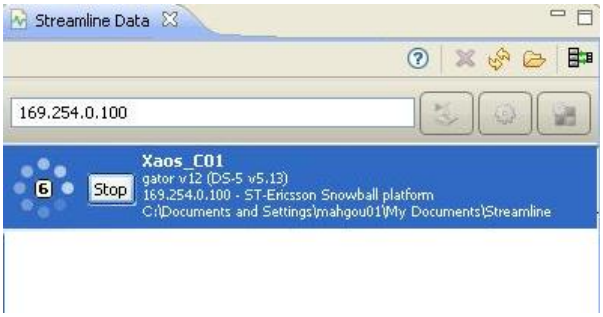
⇨ You can delete the **Disk IO:** and **Memory:** counters since they are not very interesting in this example. But leaving them in the list won't hurt either.

⇨ Click **Save** to save the counter configuration. The counter configuration is saved on the target.

You can use the Export ( ) and Import ( ) buttons to save and restore your favorite counter configurations.

If you have your own hardware or software counters (for example, number of packets processed) you can modify the gator software and use Streamline to capture and to display charts of your own custom counters. The counters of the L2C-310 cache controller are a good example to start from.

Capture some profile data

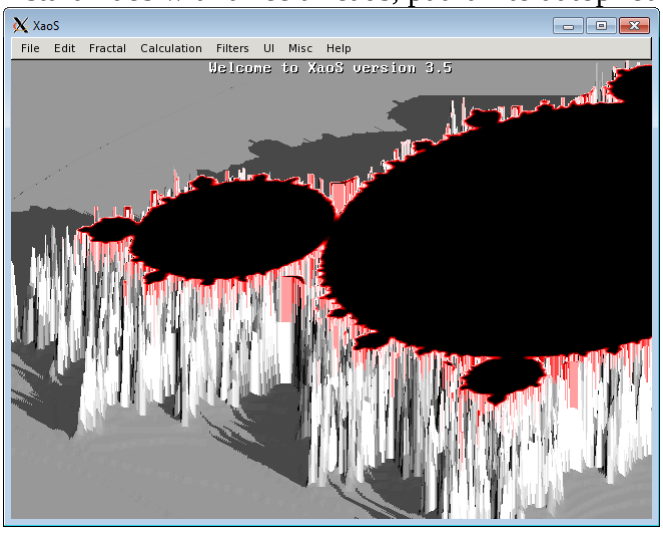


⇨ Click the start capture button ( ) to collect data from the target. You are now prompted to specify the name and location of the capture file where the profile data is stored on the host. By default it will be in the directory `...\My Documents\Streamline`. You can change the file name, for an instance we are profiling Xaos application, I used ***Xaos_C01.apc***. The **C01** indicates that it is the first capture. When you save the file, a new capture session will appear in **Streamline Data** view. The spinning wheel next to the button indicates that data is being captured from the target and the elaspsed time of the capture is shown.

⇨ Start the Xaos application. Xaos needs a `DISPLAY` environment variable to know which X server to open it's window on. Type these commands, without the `$`, in the **Terminals** view (`169.254.0.1` is the host's IP address):

```
$ export DISPLAY=169.254.0.1:0.0
$ ./xaos -autopilot -threed -threads 3 -streamlinevisualannotation
```

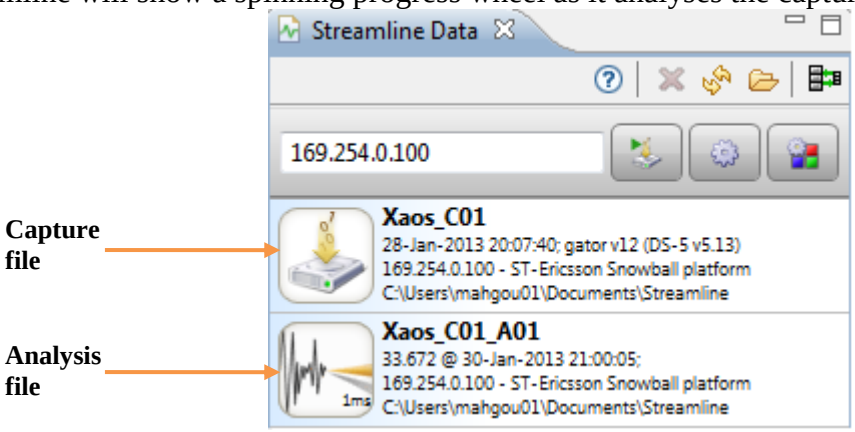
This will start Xaos with three threads, put it into autopilot mode and apply a Pseudo 3D filter.



You can also set the options from the Xaos GUI by selecting **UI > Autopilot** and **Filters > Pseudo 3d**. (When running with multiple threads the Xaos program has some display glitches and even hangs sometimes, but it's still useful as a demonstration. You can quit it and start it over if it gets stuck.)

The application has many filters and fractal formulae. You can play with the menu items a bit and if you disable Autopilot you can "drive" the application by clicking your left and right mouse buttons (but your profile results will vary a bit from what we have here). Once you are finished looking at the pretty pictures, continue.

⇨ Click the **Stop** button in the **Streamline Data** view to stop capturing data and generate a profile report. Streamline will show a spinning progress wheel as it analyses the captured data.



Only the capture has stopped, Xaos is still running. If you want, you can quit it by closing its window or typing Ctrl+C in the **Terminals** view where you started it or just leave it running.

Examine the Report

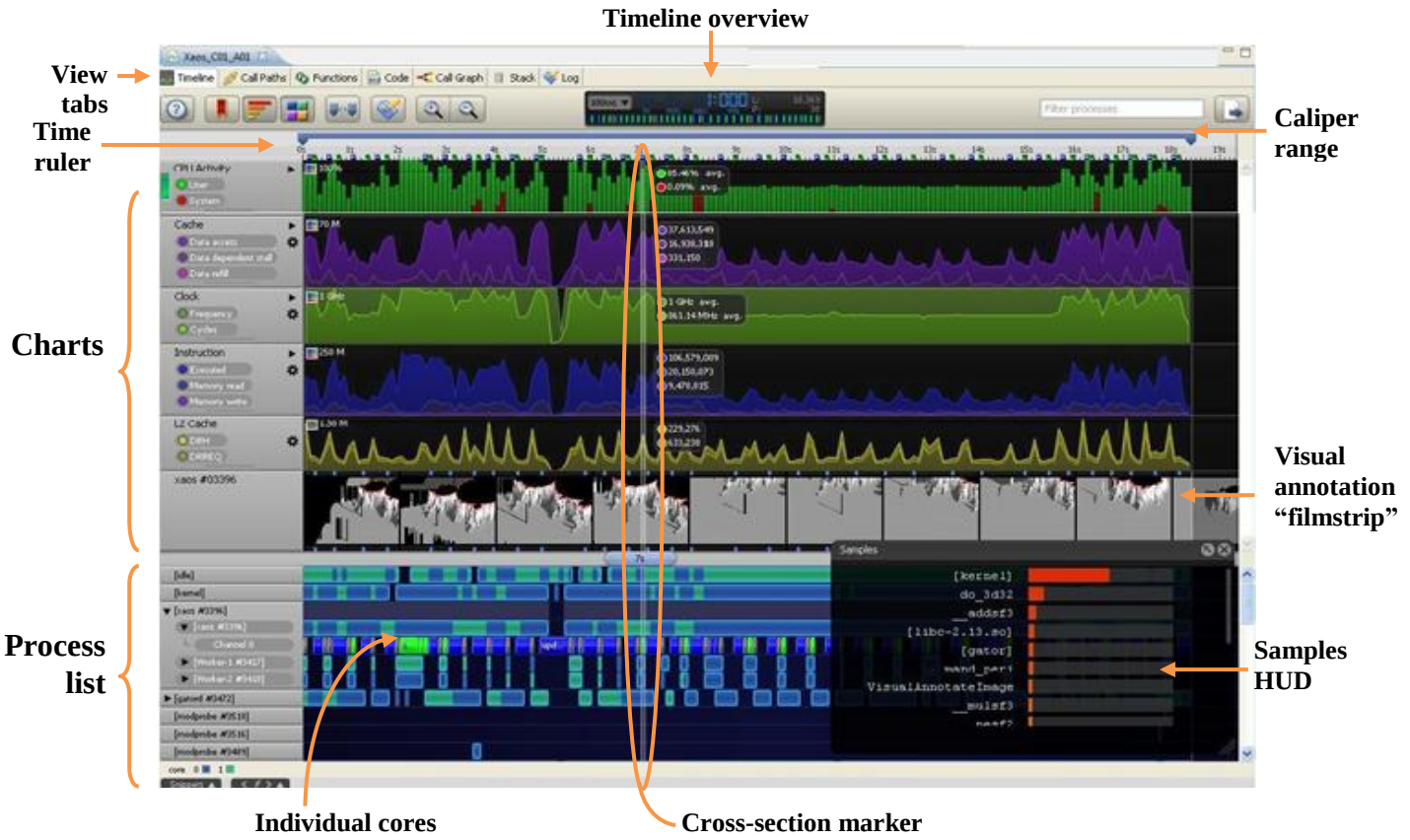
When the analysis is finished, the report view will open with a name determined by the capture's name, for example **Xaos_C01_A01** for the first analysis of the **Xaos_C01** capture.

⇨ Double click the **Xaos_C01_A01** tab of the report view to maximize it.

All of the Streamlines views can be exported to text files by using the Export button ( ) except for the Code and Call Graph views.

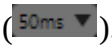
Timeline view – a first look at the Streamline report

The **Timeline** view is the first view presented when a report is opened. It displays information to give an overall view of the system performance. The graphs and percentages in your report may be different than the ones shown here depending on how long you captured data for.

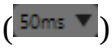


The top section of the **Timeline** view, below the time ruler, shows the system performance charts. Streamline captures performance data from the hardware and operating system, and displays the samples or "snapshots" of the chosen counters in the system performance charts. The order of the charts can be rearranged by dragging them by the gray legend area at the left end up or down. Just to the right of the legend is an indication of the full vertical scale of each chart (for example, 30 M). You can get help on any of the charts by clicking the **Help** button ( ) in the top left of the **Timeline** view and select **Timeline view charts**.

If the target has multiple cores then some of the charts will show combined values for all of the cores. You can use the expansion triangle to show the values for the individual cores as shown below.

Above the time ruler are the buttons and the Timeline overview. The Timeline overview shows the current zoom level as a drop-down menu (

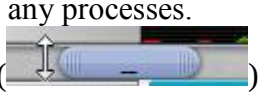
). The large time in the center shows the time of the current mouse position as you move the mouse. The Timeline overview also shows the length of the capture after the **L** and the number of processes after the **P**. Along the bottom of the Timeline overview is a bar that represents the entire capture. Clicking on the bottom bar will jump scroll the Timeline to that point.

⇨ Click the zoom buttons () to zoom in and out one level at a time (by a factor of 2 or 2.5). You can also use the drop-down menu (

) in the Timeline overview to change the zoom.

By default the finest resolution is 1 ms. If you use the **High Resolution Timeline** option when doing the analysis the finest resolution is 1 µs. You can also zoom in and out by typing **I** or **+** and **O** or **-**.

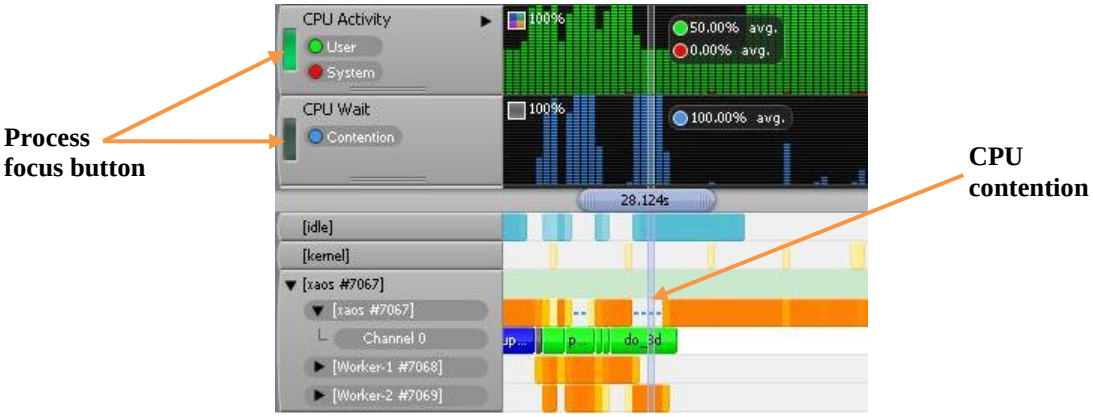
Between the charts and process list is the cross-section thumb bar (). If the cross-section thumb bar is at the bottom of the view, you won't be able to see any processes.

⇨ You can drag the cross-section thumb bar up and down () to adjust how many charts and processes are shown.

By default the process list shows a "heat map" colored by **CPU Activity** in red, orange and yellow. Red indicates that this is a "hot spot" i.e. an area where a lot of time was spent during the execution. These are typically the parts of the code which one would inspect and look to optimize first.

If a process has multiple threads it will have an expansion triangle to the left of the process name that will toggle between showing the individual threads and the aggregate of all the threads. You can see the context switches where one process or thread stops running and another starts. Xaos gives these threads names, **Worker-1**, **Worker-2** and so on depending upon the number of threads that you choose to run at the time of launching the application.

Process/thread contention is shown by in the process list. This is when a process or thread is runnable but other threads are using all of the cores, for example here is a case where all three threads of xaos want to run at the same time (but there are only two cores). Contention is also shown in the **CPU Wait/Contention** chart.



We can change to coloring the processes and threads to be "by contention":

⇨ Click on the button at the left end of the **CPU Wait/Contention** chart.

Now the process list has been recolored to show red when processes are waiting to run.

⇨ Click on the button at the left end of the **CPU Activity** chart to restore the normal coloring by CPU use.

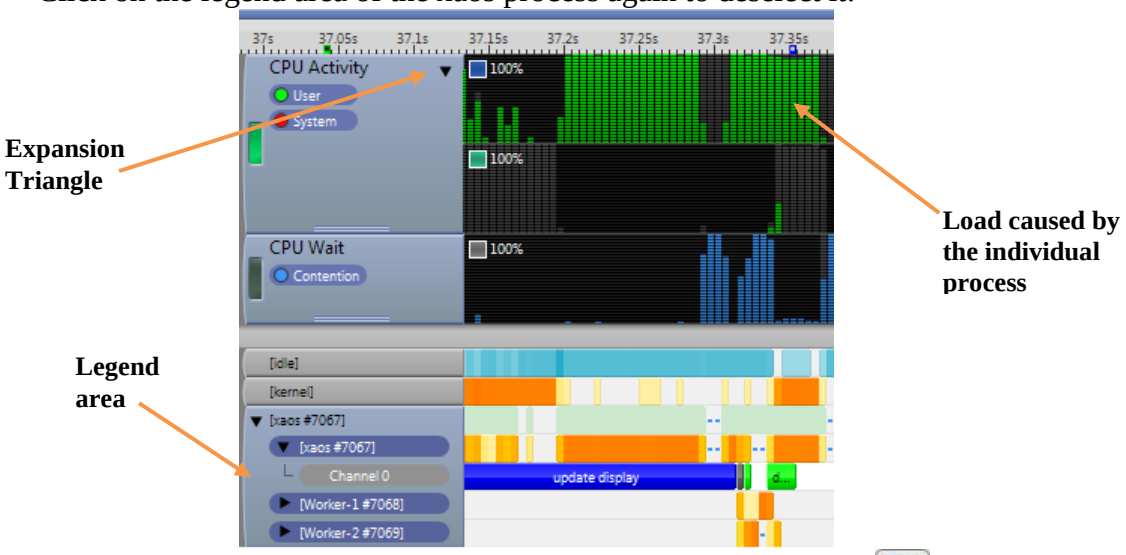
You can see how a particular process contributes to a chart by selecting it.

⇨ Click on the expansion triangle at the left end of the **CPU Activity** chart to show both cores.

⇨ Click on the legend area of the **xaos** process.

The xaos process will become selected and the **CPU Activity** chart will adjust to show the load caused by just the xaos process. You can see the original values of the **CPU Activity** chart as "ghosts".

⇨ Click on the legend area of the xaos process again to deselect it.



If the target has multiple cores, you can click the Toggle X-Ray button ( ) to turn on X-Ray mode which changes the coloring of the processes and threads to show which core they executed on. There is a legend at the bottom left of the view to show which core is which color (core 0 1). Hovering over a colored process bar will also show which core was active in a tooltip.

Let's find out more about where the time is being spent:

⇨ Zoom into an area where you can see switching between xaos, the [kernel] and [idle] similar to the pictures above and click to set the cross-section marker.

⇨ Show the Samples HUD (heads up display) by clicking the Samples HUD button ( ) or typing **S**.

The Samples HUD displays a colored histogram of where time was spent in the selected cross-section. If you have supplied debug symbols for the sampled functions then the function names will appear in the Samples HUD. For samples without symbols then the name of the process or shared library will be shown surrounded in square brackets `[]` to indicate that the time was spent inside some unknown function.

⇨ Click on the **Timeline** anywhere under the ruler and to the right of the labels to change the selected cross-section being inspected. The entries in the Samples HUD will change accordingly.

The initial width of the cross-section is determined by the current zoom level. The time of the beginning of the cross-section is shown in the "thumb" of the cross-section bar (2.3s). You can move the cross-section by dragging the thumb right and left and by typing the right and left arrow keys. You can grow the cross-section by dragging the right or left edge of the cross-section thumb. The duration of the cross-section is shown in square brackets (2.5s [0.7s]).

At the top of the **Timeline** view, above the time ruler, is the caliper range ( ). You can set the caliper range by dragging either end, or by right-clicking and choosing **Set left caliper**, **Set right caliper** or **Reset calipers**. All of the samples outside the caliper range are ignored in all of the other views. You can also reset the calipers to include all of the captured data by clicking the Reset calipers button ( ) at the top of the view.

Streamline has an **Annotate** feature that allows the application or kernel to add timestamped text, images and bookmarks to the Streamline report at key points in your code. These annotations are displayed in the **Timeline** and **Log** views. In the screenshot above text annotations are used to show when Xaos' update display and calculate fractal functions are executing. Additionally Xaos has also instrumented to capture the

frame buffer as a visual annotation which is displayed it as a "filmstrip" in the **Timeline**. The blue markers at the top and bottom of the filmstrip chart show the time at which the visual annotation was sent.

⇨ Zoom to about 200ms or coarser resolution and move the mouse right and left over the filmstrip to see the sequence replay.

There is more information about annotations later.

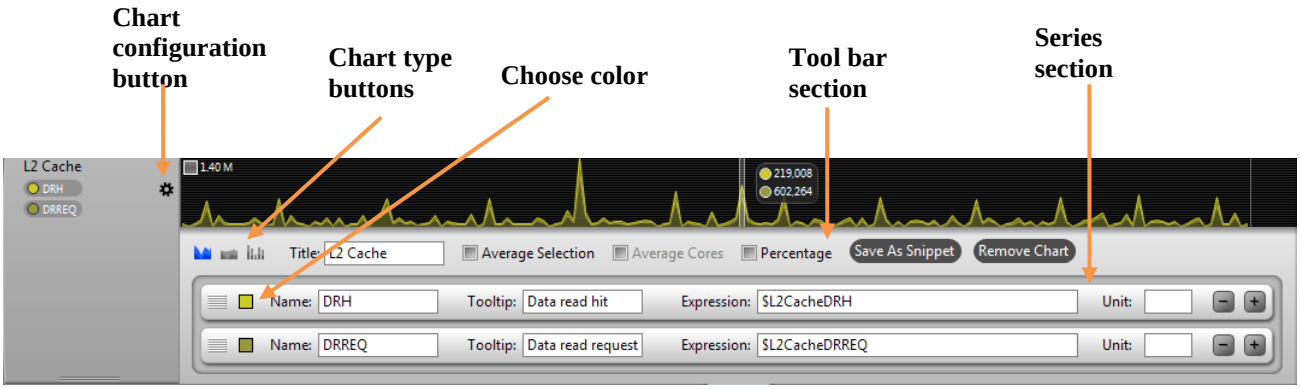
Configuring Charts

You can control many aspects of the display of the charts in the **Timeline** view by using each chart's configuration panel. You can control which counters are displayed and how they are grouped. This panel is used to change all aspects of the chart from color to the counters that the chart uses, giving the option to display the data in a more convenient way.

Each chart can display one of more series of captured data in a variety of ways. By default, the chart for the two L2 cache counters that we added (**L2C-310/L2 Cache: DRH** and **L2 Cache: DRREQ**) is a stacked chart. 

This is not really a good presentation because the data read requests counter (DRREQ) includes the accesses that were also counted by data read hits (DRH) so stacking the values is double-counting the hits. We will now see how to make this chart more user friendly.

⇨ Click on the chart configuration button ( ) to open the chart configuration panel for L2 Cache



⇨ Click on the chart color button ( ) to choose the color for differentiating the two series.

In the tool bar section, there are three chart type buttons on the left hand side,

⇨ As mentioned above, by default the chart is stacked on top of each other, to **click** on the **Overlay chart** button ( ) to see the area below the line is filled with the color defined in the series control for each data set. Now the series higher in the chart control will appear behind series that are lower in the chart control. This is the most appropriate way to view these series. 

⇨ To go back to the default view, click on the **Stacked chart** button ( ) to fill line charts to stack on top of each other.

⇨ Click on the **Stacked bar chart** button ( ) to see the bar style chart, each bin in the chart is represented by a colored bar.

Adding/Deleting series:

⇨ Click on the **add** sign button ( ) at the end of the series to add a new series. Fill the **Name** for the series. Fill the **Tooltip** field.

⇨ In the **Expression** field, press **Ctrl + Space** or the **$** symbol to activate a drop-menu that shows you a list of counters. You can select a counter in this Content Assist list to see its description. Double-click on a counter to add it to the **Source** field.

⇨ Enter the **Unit** type for the series. The value entered here will appear when the Cross Section Marker is used to select one or more bins.

⇨ To delete a series, click on the **minus** sign button ().

Check boxes on the tool bar:
Average Selection: When this option is checked, any selection made using the Cross Section Marker shows the average value of all bins included in the selection. If unchecked the overlay is a total value of all bins.

Average Cores: When this option is checked, values in a multi-core chart are the average of the individual cores when you have not used the multi-core disclosure control to show per core data. If unchecked this chart control option, the multi-core chart shows the totals from all cores.

Percentage: In a percentage style chart, values are plotted as percentages between 0 and 100 percent. The maximum value in the chart represents 100% in a percentage chart and all other values are compared to that number to calculate a percentage.

You can drag the series in the chart configuration panel up and down to reorder them using the handle () that is located on the left hand side of the series.

You can also control how counters from multiple cores are combined: average or sum.

You can also control how counters values are reported over a selection: average, sum, minimum, maximum or frequency.

You can control the descriptions (tool tips) and units that are displayed in the charts.

You can use the **Save as Snippet** button to save the current chart and its series as a "snippet". After it has been saved, you can use the **Snippets** menu, located in the bottom left of the **Timeline** view to add this chart as it is currently configured to any report.

You can use the **Remove Chart** button to remove the current chart from the **Timeline** view. If you have saved the chart as a Snippet, you can add it back to the **Timeline** view later using the **Snippets** menu.

The **Snippets** menu also allows you to add a new, blank chart which you can fill in from scratch and to import and export the snippets from/to a text file.

Power analysis

The ARM Energy Probe is designed to be a low impact, inexpensive solution to give you quick feedback on the impact of your code on the system energy footprint.

If you have the ARM Energy Probe connected to your target and if you setup the path to the caiman binary as instructed earlier, then you will also get a **Power** chart in the **Timeline** view. You will also see the **Current & Voltage charts** of "Channel 0" presuming that the Current & Voltage options were selected in the capture options. You can drag the **Power** chart upwards to get it closer to the top using a handle. If the Energy Probe is measuring the total power going into the target, then the changes in CPU power usage may get obscured by the power provided to other parts of the system.

The power data is correlated against the **CPU Activity** on the core. If the Energy Probe is measuring the total power going into the device, the CPU only power may get obscured by the power provided to other parts of the system. In this case the correlation may be slightly out by tens of milli-seconds. You can use the energy probe alignment menu( ) located in the bottom left of the **Timeline** view to manually move the power chart right and left on the x-axis if you think the correlation needs to be adjusted.

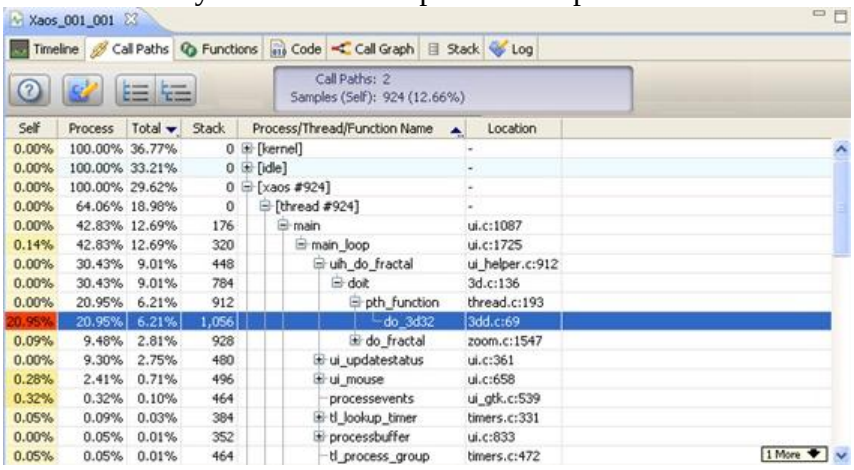
- ⇨ Drag the **Channel 0 Power** chart up or down so that it is next to the **Clock** chart so that we can correlate the power with frequency.
- ⇨ Click on **Clock** and compare the graph variations against the power and try to analyze the power consumption as per the frequency.
- ⇨ Click on an application thread in the process bar and try to correlate the energy & power consumption of that task.

To find out more about the energy probe setup see ARM Energy Probe setup in the appendix on page 99.

Call Paths view

- ⇨ In the Samples HUD in the **Timeline** view, you can right click on a function name to bring up navigation options. Find a cross-section where the **do_3d32** function is listed in the Samples HUD, then click on the function name to take you that function in the **Call Paths** view.

The **Call Paths** view displays the call paths taken by an application that were sampled. They are listed per process in the view. If you followed the previous steps it should look similar to the screenshot below.



There are five columns in the view which display the time spent as per the total number of the samples taken in that function / process.

- **Self** – The sampled time spent in that function only and not in its children as a total of the samples in that process.

- **% Self** – The sampled time spent in that function only and not in its children as a percentage of the samples in that process.
- **Process** – The sampled time spent in that function and its children as a total of the samples in that the process..
- **% Process** – The sampled time spent in that function and its children as a percentage of the samples in that the process.
- **Total** – The sampled time spent in that function as a total of the time of all processes.

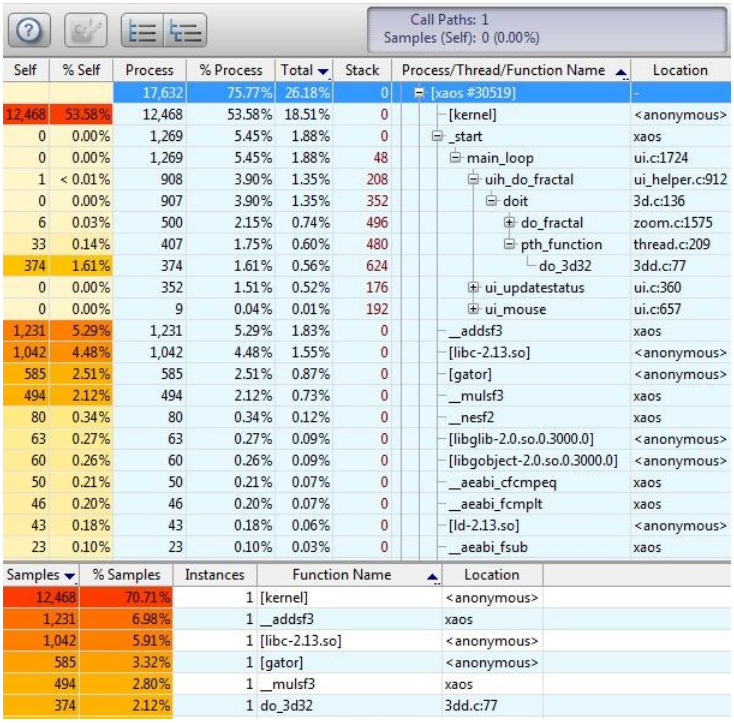
The percentages are color-coded based on their value. Functions where a lot of time was spent are highlighted in red.

The **Stack** column is an indication of the amount of stack used by that function.

⇨ Selecting the name of a process, thread or function will display the top functions by time in the bottom pane that are lower in the call path:

⇨ All of the columns can be sorted by clicking on the column name. If you hold Shift while clicking on a column name the clicked column will become a secondary sort key (which is shown by two dots under the sort arrow). This works in all of the Streamline table views.

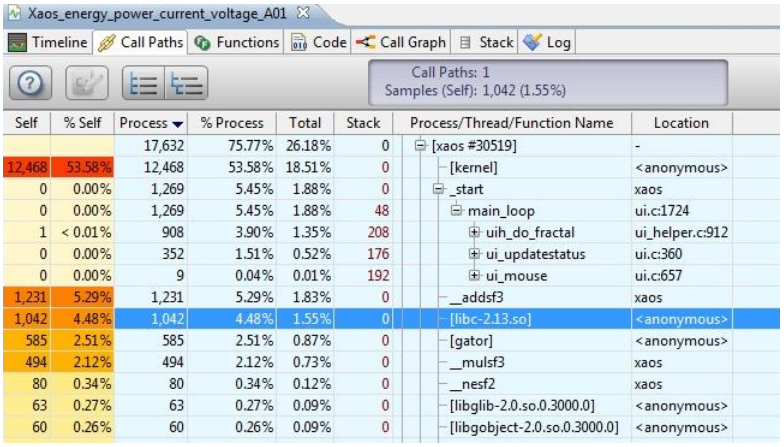
The **Call Paths** view will only display function names for code for which we have loaded the symbols. If the symbols were not loaded then the data for those binaries will appear in the **<anonymous>** location.



Self	% Self	Process	% Process	Total	Stack	Process/Thread/Function Name	Location
		17,632	75.77%	26.18%	0	[xaos #30519]	-
12,468	53.58%	12,468	53.58%	18.51%	0	[kernel]	<anonymous>
0	0.00%	1,269	5.45%	1.88%	0	_start	xaos
0	0.00%	1,269	5.45%	1.88%	48	main_loop	ui.c:1724
1	< 0.01%	908	3.90%	1.35%	208	uih_do_fractal	ui_helper.c:912
0	0.00%	907	3.90%	1.35%	352	doit	3d.c:136
6	0.03%	500	2.15%	0.74%	496	do_fractal	zoom.c:1575
33	0.14%	407	1.75%	0.60%	480	pth_function	thread.c:209
374	1.61%	374	1.61%	0.56%	624	do_3d32	3dd.c:77
0	0.00%	352	1.51%	0.52%	176	ui_updatestatus	ui.c:360
0	0.00%	9	0.04%	0.01%	192	ui_mouse	ui.c:657
1,231	5.29%	1,231	5.29%	1.83%	0	__addsf3	xaos
1,042	4.48%	1,042	4.48%	1.55%	0	[libc-2.13.so]	<anonymous>
585	2.51%	585	2.51%	0.87%	0	[gator]	<anonymous>
494	2.12%	494	2.12%	0.73%	0	__mulsf3	xaos
80	0.34%	80	0.34%	0.12%	0	__nesf2	xaos
63	0.27%	63	0.27%	0.09%	0	[libglib-2.0.so.0.3000.0]	<anonymous>
60	0.26%	60	0.26%	0.09%	0	[libgobject-2.0.so.0.3000.0]	<anonymous>
50	0.21%	50	0.21%	0.07%	0	__aeabi_cfcmpeq	xaos
46	0.20%	46	0.20%	0.07%	0	__aeabi_fcmplt	xaos
43	0.18%	43	0.18%	0.06%	0	[ld-2.13.so]	<anonymous>
23	0.10%	23	0.10%	0.03%	0	__aeabi_fsub	xaos

Samples	% Samples	Instances	Function Name	Location
12,468	70.71%	1	[kernel]	<anonymous>
1,231	6.98%	1	__addsf3	xaos
1,042	5.91%	1	[libc-2.13.so]	<anonymous>
585	3.32%	1	[gator]	<anonymous>
494	2.80%	1	__mulsf3	xaos
374	2.12%	1	do_3d32	3dd.c:77

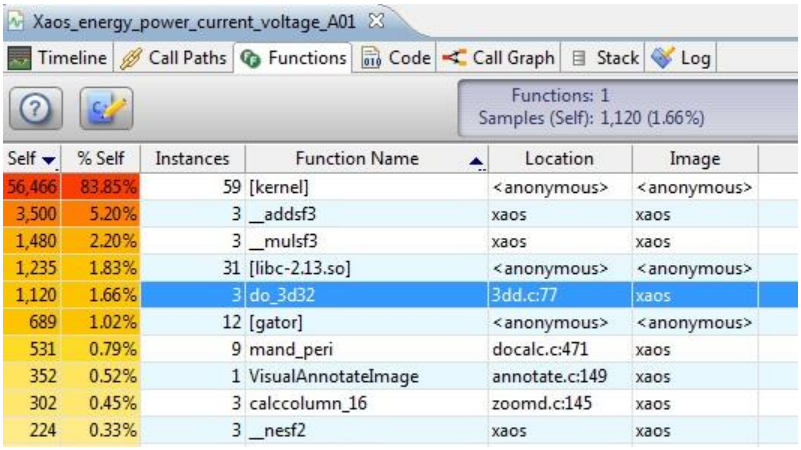
⇨ Here you can see how much time was spent inside shared libraries used by the application. You can re-analyze the profile by adding more symbols for shared libraries. We will do this as an exercise later.



Self	% Self	Process	% Process	Total	Stack	Process/Thread/Function Name	Location
		17,632	75.77%	26.18%	0	[xaos #30519]	-
12,468	53.58%	12,468	53.58%	18.51%	0	[kernel]	<anonymous>
0	0.00%	1,269	5.45%	1.88%	0	_start	xaos
0	0.00%	1,269	5.45%	1.88%	48	main_loop	ui.c:1724
1	< 0.01%	908	3.90%	1.35%	208	uih_do_fractal	ui_helper.c:912
0	0.00%	352	1.51%	0.52%	176	ui_updatestatus	ui.c:360
0	0.00%	9	0.04%	0.01%	192	ui_mouse	ui.c:657
1,231	5.29%	1,231	5.29%	1.83%	0	__addsf3	xaos
1,042	4.48%	1,042	4.48%	1.55%	0	[libc-2.13.so]	<anonymous>
585	2.51%	585	2.51%	0.87%	0	[gator]	<anonymous>
494	2.12%	494	2.12%	0.73%	0	__mulsf3	xaos
80	0.34%	80	0.34%	0.12%	0	__nesf2	xaos
63	0.27%	63	0.27%	0.09%	0	[libglib-2.0.so.0.3000.0]	<anonymous>
60	0.26%	60	0.26%	0.09%	0	[libgobject-2.0.so.0.3000.0]	<anonymous>

For the **Call Paths** to work the code needs to be built with frame pointers enabled. This allows the Streamline's gator daemon to reconstruct the call path taken by an application. In gcc the build argument is `-fno-omit-frame-pointer`. The **Call Graph** and **Stack** views also require frame pointers.

Functions view



Self	% Self	Instances	Function Name	Location	Image
56,466	83.85%	59	[kernel]	<anonymous>	<anonymous>
3,500	5.20%	3	__addsf3	xaos	xaos
1,480	2.20%	3	__mulsf3	xaos	xaos
1,235	1.83%	31	[libc-2.13.so]	<anonymous>	<anonymous>
1,120	1.66%	3	do_3d32	3dd.c:77	xaos
689	1.02%	12	[gator]	<anonymous>	<anonymous>
531	0.79%	9	mand_peri	docalc.c:471	xaos
352	0.52%	1	VisualAnnotateImage	annotate.c:149	xaos
302	0.45%	3	calccolumn_16	zoomd.c:145	xaos
224	0.33%	3	__nesf2	xaos	xaos

The **Functions** view displays a list of all the functions that were sampled during the profile session. The functions view is the quickest way to find which functions are your hot functions. If you've set the caliper range in the **Timeline** view then only the samples within the callipers will be reflected in the **Functions** view.

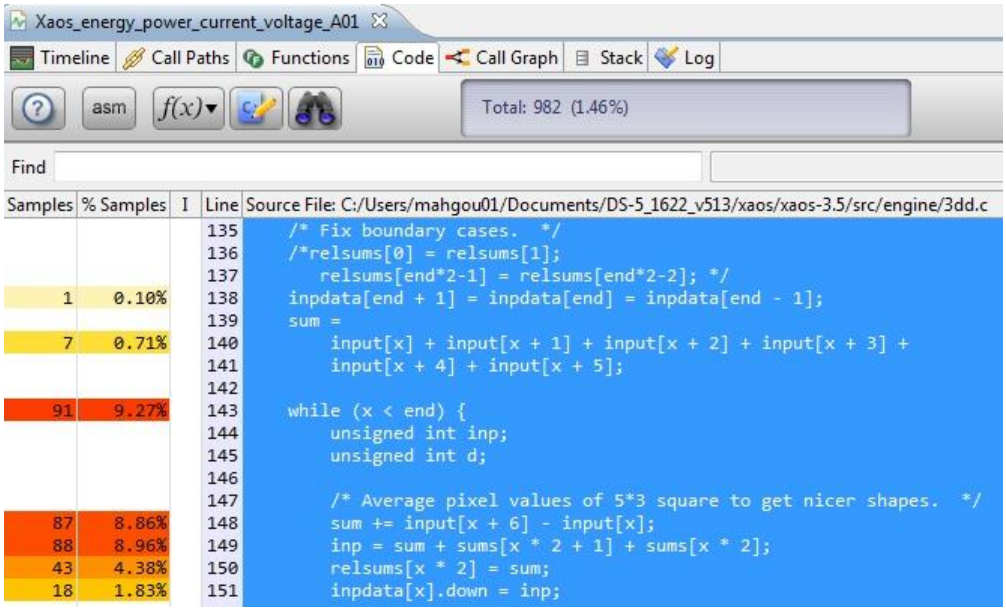
- ⇨ Select the **Functions** tab at the top of your report.
- ⇨ You can **Shift-** or **Ctrl-click** to select multiple functions to see a total count of the self time in the **Totals** panel in the top right of the view.

The **Instances** column shows the number of times a function appears in separate call paths.

Next let's go investigate the code inside one of our hot functions;

- ⇨ Right click on the `do_3d32` function and choose **Select in Code View** from the context menu.

Code view



Now the `do_3d32` function will be highlighted in blue in the **Code** view.

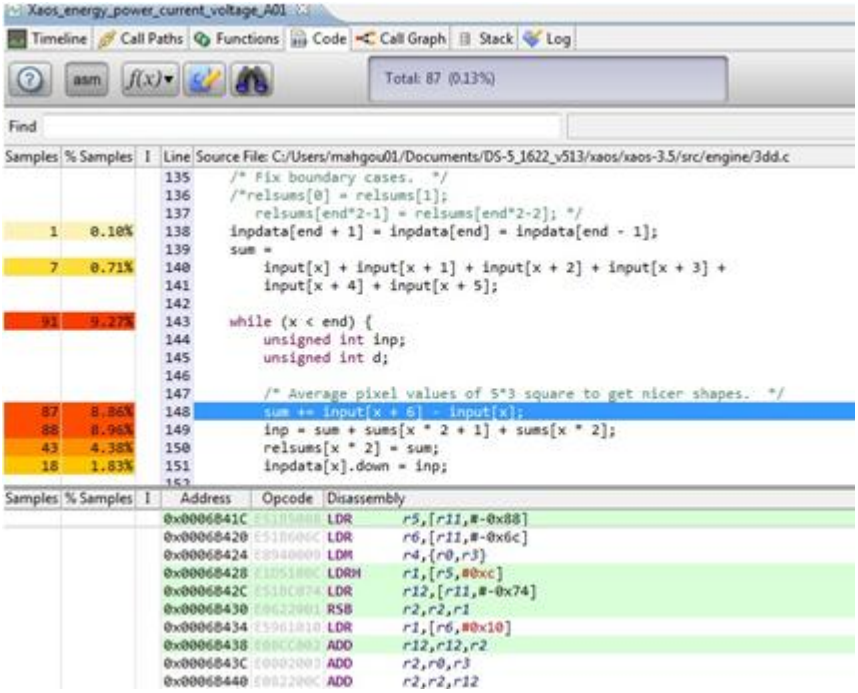
(If the source file has not been found by default you can locate it by pressing the (🔭) button or the "Click here to locate it." link to browse the file system. You can use the displayed path name to help you locate the file. This may happen if you used a pre-built copy of xaos instead of building it on your host or if the source files are

in a different location now compared to when you built the application.)

Lines in the source code where many samples were taken are highlighted in red. Lines of code that don't have any percentages next to were not sampled at all in this profiling run. Profiling for longer periods of time or at a higher sampling rate may give a better resolution in the source code.

You can also open up the disassembly for the code you are viewing.

- ⇨ Click the Toggle Asm button (asm) in the top right hand corner of the report. You can select a line or lines of source code and the corresponding disassembly lines will be highlighted in the bottom. Vice versa, you can also make a selection in the disassembly pane and the source above will be highlighted, along with the other disassembly lines for that source line.



Notice the **2 More** indicator in the screenshot. It indicates that there are two more disassembly instructions highlighted above. It's useful when compiler has moved the code out of order. You can click on the indicator to scroll them into view.

Call Graph view

The **Call Graph** view is a graphical display showing which functions called each other. Hover above a function name to see the number of samples taken on that function.

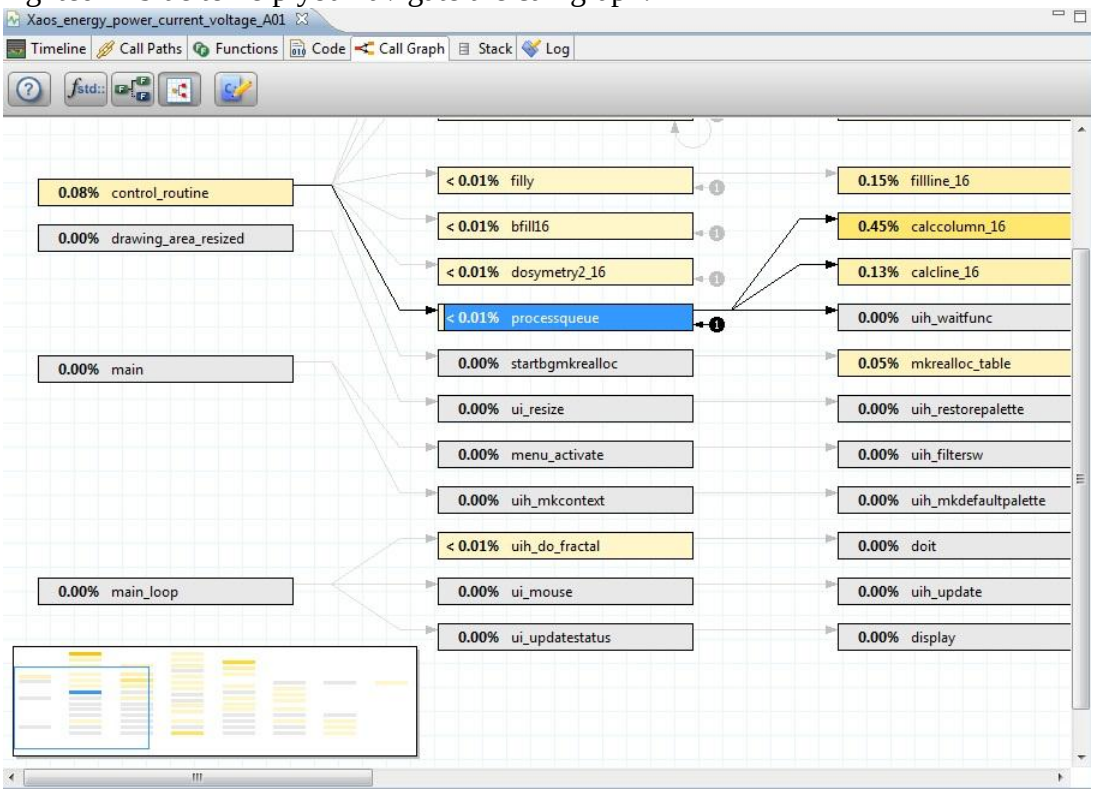
- ⇨ Use the mini-map in the bottom left hand corner to scroll around the map. If the mini-map is in your way you can disable it by selecting ( ) in the top left corner.

- ⇨ Enable **Uncalled** functions by clicking ( ).These functions were not in the call chain of any sample.

- ⇨ Enable **System** functions by clicking ( ). This displays runtime library functions and shared libraries without symbols.

To help navigate the call graph right you can right click on a function to find options to highlight callers and call paths.

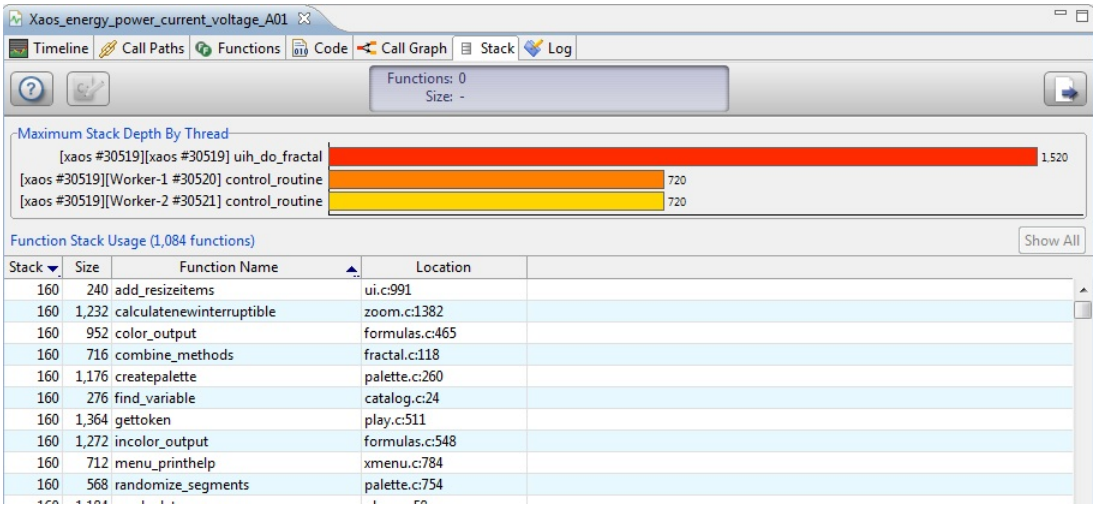
⇨ Right click on the `do_3d32` function. Select **Caller Tree**. The call path for the function will be highlighted in blue to help you navigate the call graph.



Stack view

The **Stack** view is designed to give the user an idea of the amount of maximum stack usage in a particular thread. In this view you can also see and sort the stack usage of all the functions used by the application.

⇨ Select the bar next to a particular thread to view the call chain and maximum stack usage for that thread



The total may be larger than sum of the individual entries. This is due to recursive functions.

⇨ Select the **Show All** button to view the stack usage of all the functions that were sampled.

Note : If all the functions that were sampled, then the **Show All** button will be greyed out.

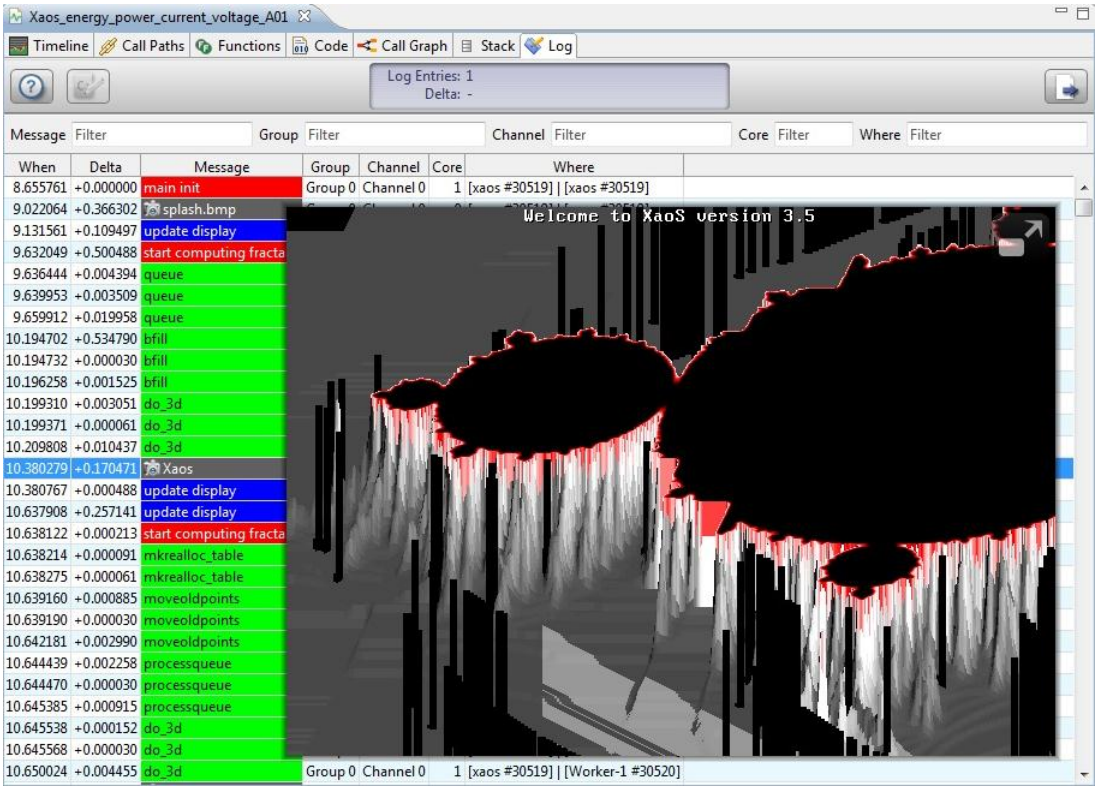
Log view and Annotations

The **Log** view displays a list of all the text and visual annotations that were generated by the application along with a time stamp for each and the time delta from the previous entry. You can filter the list on the various columns using regular expressions. When you select an entry for a visual annotation the image is displayed.

⇨ Type **X** in the **Message** regex filter.

Now only the visual annotation log entries are shown (because only they contain an **X** in their messages) and the values in the **Delta** column have been recalculated.

⇨ Select one of the visual annotations and click the grow button in the top right corner of the visual annotation thumbnail



⇨ You can use the up and down arrow keys to replay through the images.

You can double-click an annotation entry and you will be taken to the **Timeline** view and the cross-section marker will be set to the time of that annotation. You can right-click on an entry and choose **Select Process/Thread in Timeline** or **Select in Call Paths**, which can be handy if you have many processes and threads creating annotations.

In order to create annotations the application, libraries, kernel or kernel module code must be modified. The Annotation features uses standard file IO to write to a virtual annotation file, **`/dev/gator/annotate`**. Header files containing C style macros and Java classes are provided to annotate from both Java, C/C++ user space and also from kernel space. You can find the annotate header files by going to **Help > ARM Extras… > gator > annotate**. To find out more about annotation see **Help > Help Contents > ARM DS-5 Documentation > ARM Streamline > Annotate and the Log View**.

Advanced Streamline

Reanalyze Streamline data

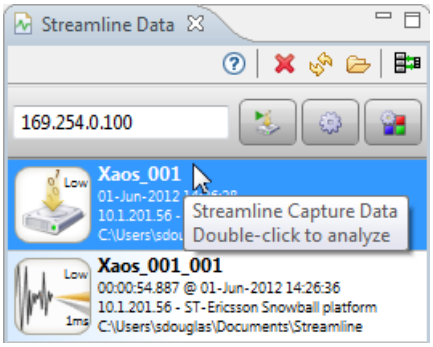
Because we did not include the C library in the initial analysis, all samples in it are shown together as **`[libc-2.13.so]`**. We can add the C library and reanalyze the same capture to get a new report, showing the C library function names.

Because the version of libc on the target is different than the copy we used to build against in the example distribution, we need to get a copy of the target's libc onto the host.

⇨ Use **My Target** in the **Remote.Systems** view to copy **Sftp files** > **Root** > **`/lib/arm-linux-gnueabi/libc-2.13.so`** to the host by dragging it to the **xaos** project in the **Project Explorer** view. (You could also copy it by dragging it to a **Windows Explorer** window or use **Local** in the **Remote Systems** view.)

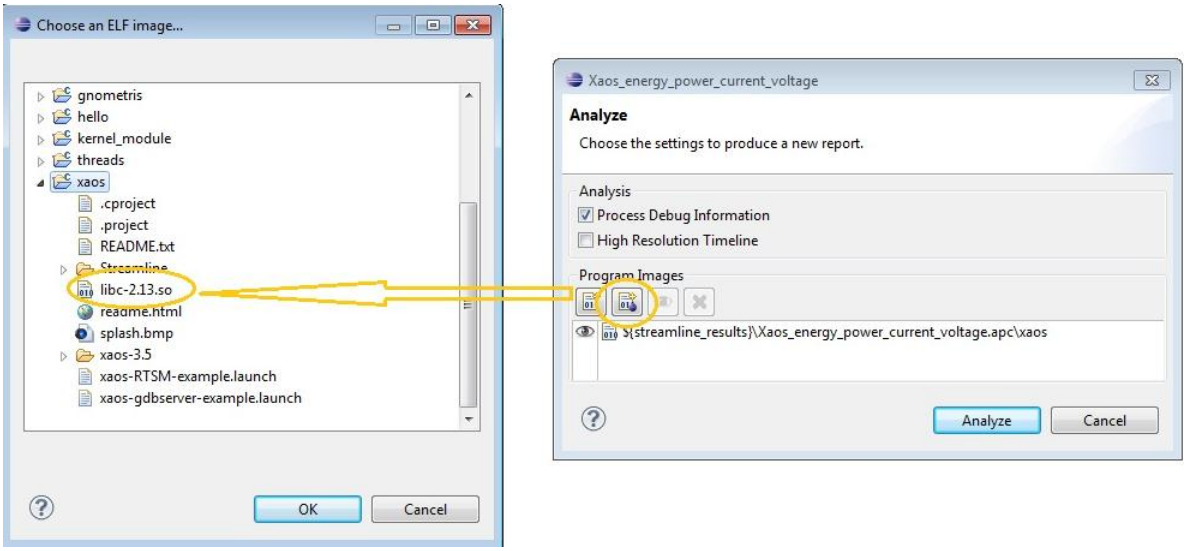
This copy of **`libc`** does not have debug information so Streamline won't be able to show the source code of the C library (which would also require the source files), but there are still enough symbols to identify the functions.

Captured streamline data and generated reports are shown in the **Streamline Data** view. Captured data is identified by ( ) and has a name like **Xaos_C01**, whilst a generated report is identified by ( ) and has a name like **Xaos_C01_A01**.

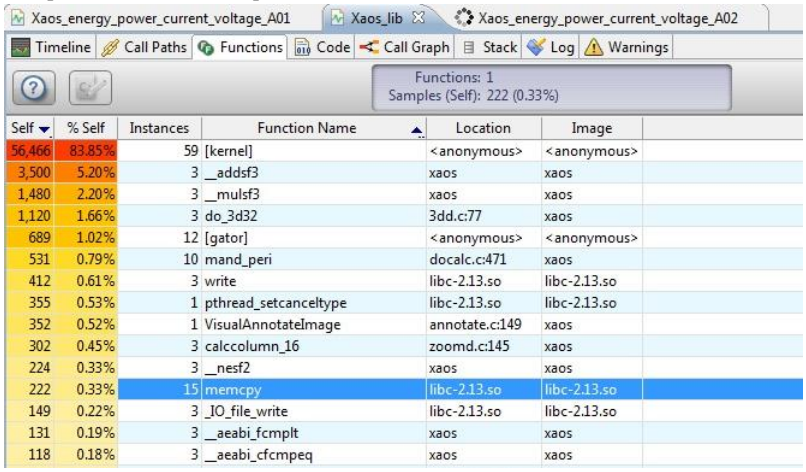


⇨ Double-click the **Xaos_C01** capture data in the **Streamline Data** view.

⇨ Click the Add ELF image from Workspace... (second) button ( ), choose the copy of **`libc-2.13.so`** that you copied from the target and click the **OK** and **Analyze** buttons; then click the **OK** button in the save report dialog:



A new report, **Xaos_C01_A02**, will be generated which will show the C library functions like **`memcpy`** instead of **`[libc-2.13.so]`** as shown:



Self	% Self	Instances	Function Name	Location	Image
56,466	83.85%	59	[kernel]	<anonymous>	<anonymous>
3,500	5.20%	3	__addsf3	xaos	xaos
1,480	2.20%	3	__mulsf3	xaos	xaos
1,120	1.66%	3	do_3d32	3dd.c:77	xaos
689	1.02%	12	[gator]	<anonymous>	<anonymous>
531	0.79%	10	mand_peri	docalc.c:471	xaos
412	0.61%	3	write	libc-2.13.so	libc-2.13.so
355	0.53%	1	pthread_setcanceltype	libc-2.13.so	libc-2.13.so
352	0.52%	1	VisualAnnotateImage	annotate.c:149	xaos
302	0.45%	3	calccolumn_16	zoomd.c:145	xaos
224	0.33%	3	__nesf2	xaos	xaos
222	0.33%	15	memcpy	libc-2.13.so	libc-2.13.so
149	0.22%	3	_IO_file_write	libc-2.13.so	libc-2.13.so
131	0.19%	3	__aeabi_fcmplt	xaos	xaos
118	0.18%	3	__aeabi_cfcmpeq	xaos	xaos

You can also change the **High Resolution Timeline** option when you reanalyze.

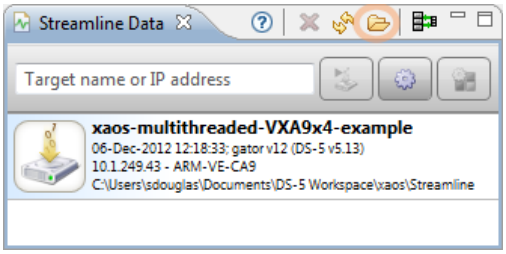
Because we did not include the kernel debug symbols in the analysis, the report shows all the kernel functions lumped together as **`[kernel]`**. You can add the kernel debug symbols and re-analyze by double clicking on the captured data **Xaos_C01**; clicking the Add ELF Image from File System (first) button( ) and finding **`kernel\vmlinux-3.3.0-1000-ux500`** and then clicking **Analyze** and then **OK** in the save report dialog. Now the kernel functions are shown individually.

Streamline captures and analysis can also be scripted from the command line. Please refer to the Using Streamline on the Command Line section in the documentation.

Import Captures

Captured data (a **`.apc`** directory) can be copied to a different location or host and imported into Streamline. If you don't have a target, there is a pre-captured report in the Xaos example project **`/xaos/Streamline`** that you can use instead of capturing your own.

⇨ Click the Edit Locations... button ( ) to add the location **`...\My Documents\DS-5 Workspace\xaos\Streamline`** to the **Streamline Data** view.



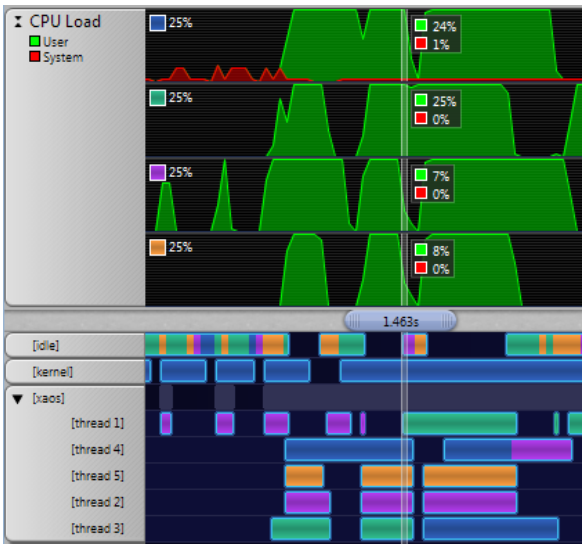
⇨ Create a report by double clicking the capture **xaos-multithreaded-VXquadA9-example** and clicking **Analyze**. and then **OK** in the save report dialog.

This report was generated on a Versatile Express Quad-Core Cortex A9 board. In the **Timeline** view you will be able to expand the chart for each core and multithreaded process.

⇨ Expand the **xaos** process so that you can see the threads.

⇨ Click the Toggle X-Ray button ( ) to turn on X-Ray mode.

Now by zooming in and hovering over the activity bars, you can see which core the processes and threads were running on at any given time.



You can export the report views from Streamline as text by selecting ( ). This allows you analyse the captured data outside of the Eclipse environment.

Troubleshooting Streamline

My target and host can't communicate

There are a number of things that could to check here

1) Make sure that your IP Address is in the same range as the target i.e. for the workshop set it to `169.254.0.1`. The target defaults to `169.254.0.100`.
2) If you're on a laptop disable WiFi and any other network adapters that you are not using.
3) Try to ping the target from the host and the host from the target.
4) Check your firewall settings. If you have trouble with the target and host communicating and your host is running a firewall you may need to configure it to allow network traffic from the target (for example, make IP address `169.254.0.100` a "friend" or allow Xming in Inbound Rules). You may need to quit and restart Xming after the change.

Xaos does not appear even though I can ping the target

1) Make sure that you selected **No access control** when you ran XLaunch.

Back to Debugging

If you have enough time, here a a few more topics about debugging you might be interested in.

Debugging an application that is already running

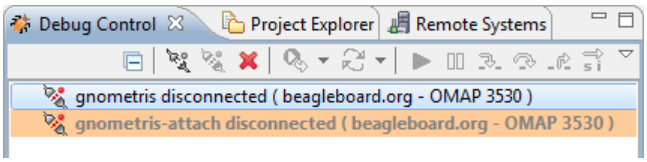
In the examples above, when we started debugging the application it was not running yet and we started a new copy of the application to debug. It's also possible to attach the debugger to an application that is already running.

- First disconnect ( ) any connected debug configuration in the **Debug Control** view. You can remove ( ) it from the **Debug Control** view if you want. The DS-5 Debugger can have multiple connected debug configurations at the same time , so it's not strictly necessary to disconnect any existing debug configurations, but it can be confusing, especially if you're debugging multiple copies of the same application.
- If the **Terminals** view is not already open, use **Window > Show View… > Other… > Remote Systems > Terminals** to open it.
- Use the **Terminals** view to start `gnometris` on the target without gdbserver (as one command):
  ```
  /home/linaro/gnometris --display=169.254.0.1:0 &
  ```
- Start a **New Game** and play it for a short while.

We need to change the debug configuration so that it uses **Connect only** instead of trying to debug from `main`. We will do this by copying the existing configuration so that we can have both configurations handy. You can just leave the game running.

- Choose **Run > Debug Configurations…** to open the **Debug Configurations** dialog.
- Select the **gnometris** configuration that was created earlier and click the duplicate button ( ).
- Change the name from **gnometris (1)** to **gnometris-attach**.
- In the **Connection** pane, set the **Debug operation** to **Connect to already running gdbserver** and put the target IP address, `169.254.0.100`, in the **Address** field.
- In the **Files** pane, create a **Load symbols from file** entry in the **Files** list by clicking the **Workspace...** button and choosing the `gnometris` file at the top-level of the **gnometris** project.
- In the **Debugger** pane, choose **Connect only** instead of **Debug from symbol**.
- Click the **Apply** button to save the changes and click the **Close** button to close the dialog.

The **gnometris-attach** debug configuration appears in the **Debug Control** view. It is currently disconnected.



Now we'll start gdbserver and attach it to the running game:

- On the target, execute the command:
  ```
  gdbserver --attach :5000 $! &
  ```
  The `$!` represents the process id of last command that was started in background, with a trailing `&`. In our case that was the `/home/linaro/gnometris` command. The output should be something like:
  ```
  Attached; pid = 1280
  Listening on port 5000
  ```
  The game stops when gdbserver attaches to it. gdbserver is waiting for the debugger to connect.
- Double-click the **gnometris-attach** debug configuration in the **Debug Control** view to connect to the gdbserver. The Terminals view will show:
  ```
  Remote debugging from host 169.254.0.1
  ```
  You can also connect by selecting the **gnometris-attach** configuration and clicking the Connect ( ) button or by or by selecting the **gnometris-attach** configuration in the **Debug Configurations** dialog and clicking the **Debug** button.

At this point the game will be stopped at some arbitrary point, you can now set breakpoints, run and debug it. When attaching to an already running application, the debugger won't initially know about shared libraries that the application has already loaded or threads that were created before gdbserver was attached. You can execute the **sharedlibrary** command in the **Commands** view to tell the debugger to learn about all currently loaded shared libraries.

Advanced Debug Configurations

The initial temporary breakpoint can be set in the debug configuration with **Debug from entry point** or **Debug from symbol**. It can also be set using the command line or scripts. Execution of the application typically starts in the dynamic loader, **ld-linux.so**, which loads and initializes the shared libraries and then jumps to the application's entry point which then executes various library initialization routines before arriving at the `main` function.

We can change where the initial temporary breakpoint is placed by changing the setting in the debug configuration, for example if we use `Tetris::gameNew` instead of `main` then when debugging starts Gnometris would not stop at `main` but would open its window and run until we executed the **New Game** (Ctrl+**N**) command. The initial temporary breakpoint is removed after it is hit.

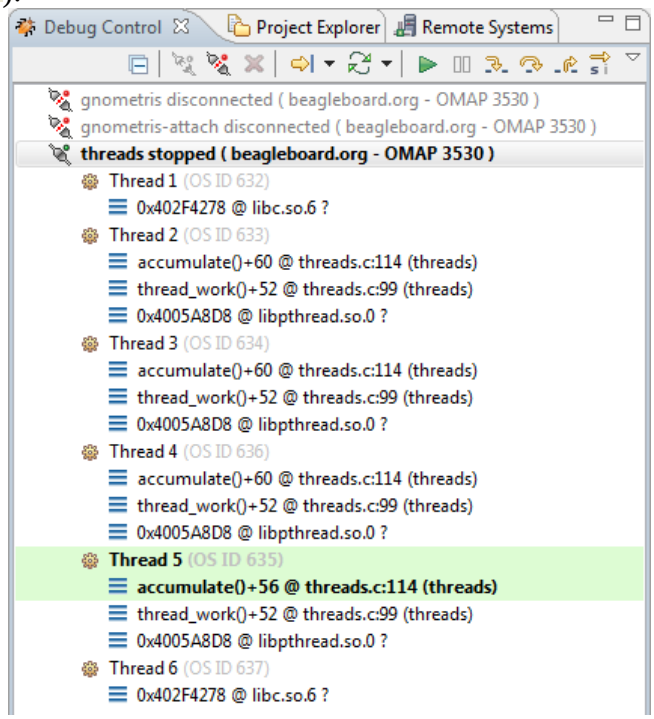
If we want the application to start running and not stop anywhere in particular we can choose **Connect only** and add `run` as a command to execute in the **Debugger** tab of the debug configuration. In this case we'll need to use the Interrupt button ( ), a breakpoint or perhaps a signal to stop the application.

Debugging Threads (“extra credit”)

Unfortunately Gnometris only has a single thread so it’s hard to show the thread features of the debugger. DS-5 contains a *threads* example. If you have some time left, you can import it, build it and debug it according to the instructions its `readme.html` file. To run the threads example on the target, you'll need to create a debug configuration for it, like we did for Gnometris; but threads doesn’t need use shared libraries or need any arguments. There will be a **threads-RTSM-example** debug configuration that uses the RTSM. You can put a breakpoint in the `accumulate` function which is run in multiple threads

The threads are displayed under the debug configuration in the **Debug Control** view. You can control the way threads are displayed.

⇨ Choose **Flat** from the **Thread Presentation** submenu of the **Debug Control** view’s drop-down menu ( ):



Below each thread will be the frames of its call stack. You can change the debugger's focus from one thread to another by clicking on the different threads. You can collapse all of the threads by clicking the Collapse All button ( ).

And you probably noticed that you can make a breakpoint apply only to one or more specific threads by using the **Breakpoint Properties** dialog.

Debugging on the RTSM

If you've still got time left, you can try using the **gnometris-RTSM-example** debug configuration to debug the game running on the Real Time Sysrtem Model (RTSM). The same binaries (application and shared libraries) will work on the RTSM. Check below first if your host is running Windows Vista or Windows 7 (or later?). You can find the **gnometris-RTSM-example** debug configuration in the **Debug Configurations** dialog. You can select it, inspect its various settings (don't change them) and click the **Debug** button. Another way to start the **gnometris-RTSM-example** debug configuration is to choose **Add Configuration (without connecting) …** from the **Debug Control** view's drop-down menu ( ) and then select **gnometris-RTSM-example** and click **OK**. Now **gnometris-RTSM-example** appears in the **Debug Control** view and you can double-click it to start it as usual.

Using the RTSM on Windows Vista or Windows 7

The RTSM uses a telnet client for its simulated serial ports. On Windows Vista and Windows 7 (and later?), the telnet client is disabled by default. Before you use the RTSM you will need to enable it by choosing **Start > Control Panel > Programs > Programs and Features > Turn Windows features on or off** and checking **Telnet Client** and clicking **OK**.

Other DS-5 features you didn't see

Due to time, hardware or other constraints there are some DS-5 features that we haven't had a chance to show in this workshop. We'll give a brief description here to let you know about them.

- **Screen** view: There is a **Screen** view that can display target memory as a picture. For example, displaying an LCD RGB565 data buffer.
- **Target** view: There is a **Target** view that displays the properties of the target.
- Command-line debugging: the debugger can be driven from the command line and/or scripts instead of using Eclipse.

Finished!

Thanks for your time. Please give us feedback on either this workshop or the tools.

Appendix A: Setup

Host Setup

DS-5 can be used on Windows and Linux hosts. If your host has not already been setup for you then you will need to:

- Download and install an evaluation copy DS-5 by following the **Download Now** link from http://www.arm.com/ds5. Also download the **DS-5 Linux Distribution Example** found in a file named **`DS500-BN-00009-r5p0-13rel0.zip`** which is a separate download located "next to" the DS-5 installer.
- Obtain and install an evaluation license (covered below)
- Adjust the host's networking so that it can communicate with the target. TCP/IP (for example Ethernet) is used to communicate with the target when doing application level debug and Streamline profiling. The rest of this workshop assumes that the host's IP address is **169.254.0.1** and that the target's IP address is **169.254.0.100**. If your addresses are different you will need to make the appropriate adjustments to the instructions in this workshop.
- If you are using a Windows host you will need to install and start an X server such as Xming (see below). The Gnometris and Xaos applications use the X server to display a window. The Public domain version of Xming can be downloaded from http://www.straightrunning.com/XmingNotes.
- If your host is running firewall software it may need special configuration to allow full network communication with the target.

Importing projects

If you already have an Eclipse project with the same name, for example if you've already imported an older version, you won't be able to import it again. You can either rename or delete the existing Eclipse projects if you want to import them again. If you delete them, you should check the **Delete project contents on disk** checkbox. If you have deleted the Eclipse project, but the project folder is still in the workspace directory, for example if you didn't check **Delete project contents on disk**, then you will need to delete the project folder "by hand" before you can import it again.

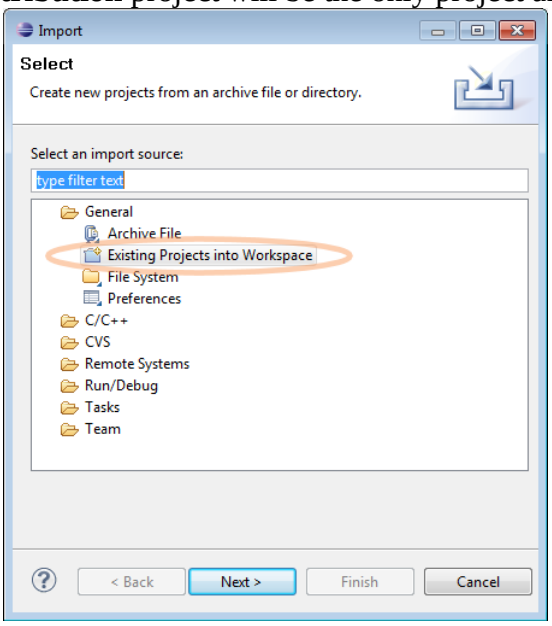
Importing distribution

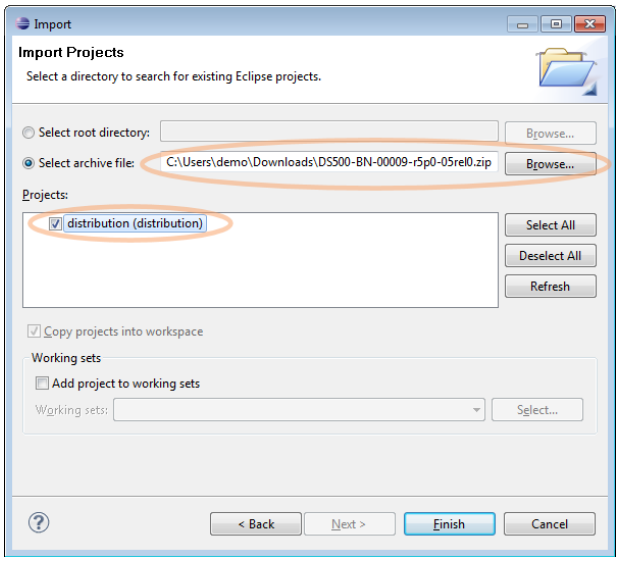
The **`distribution`** project doesn't build anything itself. It contains header files and copies of shared libraries from the target filesystem that are needed to compile and link the **`gnometris`** project. Collecting these shared libraries and headers into **`distribution`** project is a handy way for the **`gnometris`** project to be able to find and use them. The **`distribution`** project is located in a separate **`.zip`** file that is not installed by the DS-5 installation and needs to be downloaded separately. It is listed as **DS-5 Linux Distribution Example** at https://silver.arm.com/browse/DS500 and the resulting file name is **`DS500-BN-00009-r5p0-13rel0.zip`**.

⇨ Choose **File > Import... > General > Existing Projects into Workspace > Next >**

⇨ Choose **Select archive file** and use the **Browse...** button to select **`DS500-BN-00009-r5p0-13rel0.zip`**

The **distribution** project will be the only project and it will be checked.





⇨ Click the **Finish** button. If the **Finish** button is disabled then Eclipse will put a message explaining why at the top of the **Import** dialog.

Importing Gnometris

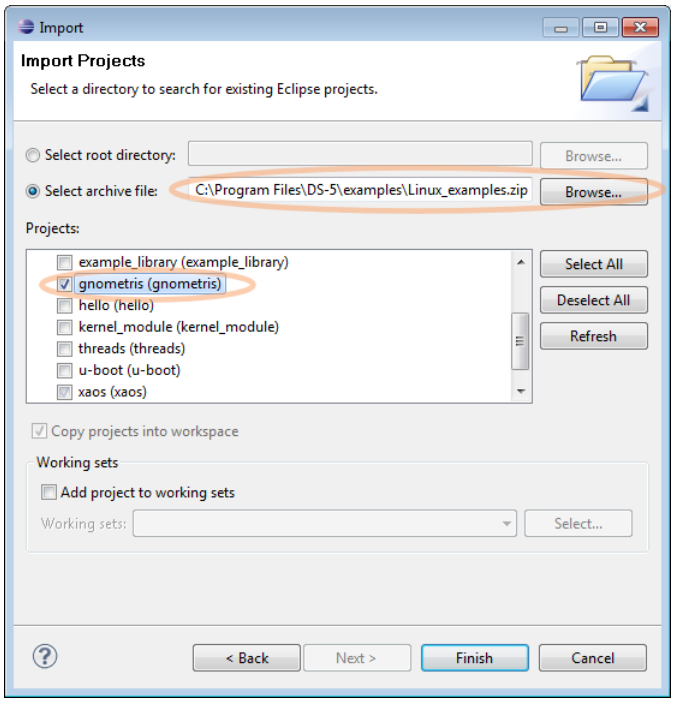
Now import the **gnometris** project which is in a different `.zip` file:

⇨ Choose **File > Import... > General > Existing Projects into Workspace > Next >**

⇨ Choose **Select archive file** and use the **Browse...** button to select `C:\Program Files\DS-5\examples\Linux_examples.zip`

⇨ Click the **Deselect All** button; then check only **gnometris**. If you're doing the Streamline section of the workshop, you can also check **xaos** to import it now or wait until later.

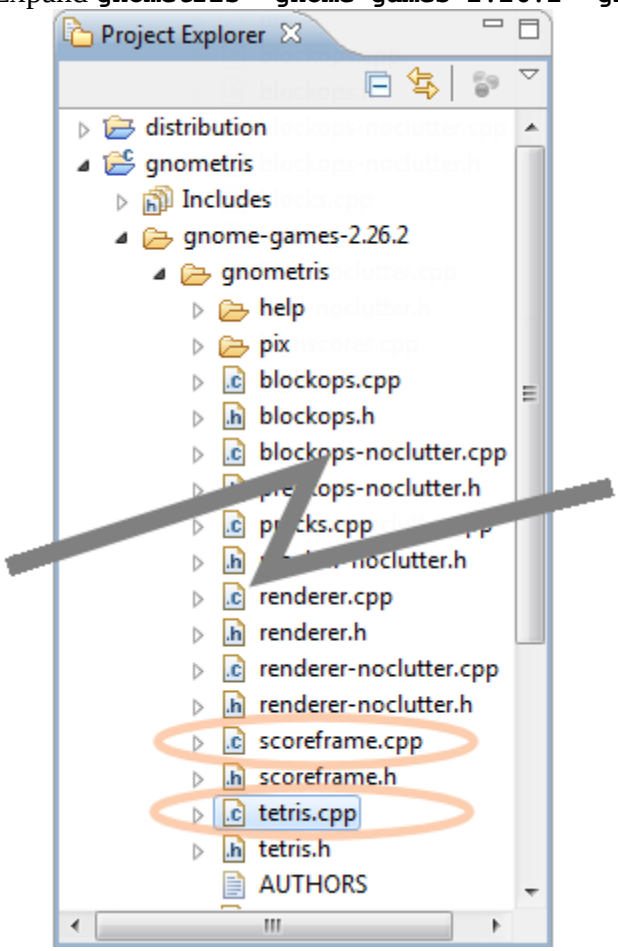
⇨ Click the **Finish** button.



Changing Gnometris

The Gnometris example project is supplied pre-built, but we are going to make some changes to the Gnometris code so that Gnometris will update its display more often so that we can see the effects of some of our debugging actions. If this workshop was setup on your host for you, this may already have been done.

⇨ Expand **`gnometris`** > **`gnome-games-2.26.2`** > **`gnometris`** in the **Project Explorer** view



⇨ Double-click on **tetris.cpp** and **scoreframe.cpp** to open them

For the next step you can copy the text from **readme.html** located in the top level of the **`gnometris`** project folder. (Click on the link to *Debugging the Gnometris application executable with gdbserver* and scroll to under point number 13.)

⇨ In **tetris.cpp**, insert the three lines marked **`//_ARM_ refresh ...`** below into both **`Tetris::timeoutHandler()`** near line 933 and **`Tetris::keypressHandler()`** near line 1000. You can use the scrollbar and watch the line numbers or use **Navigate** > **Go to line ...** (**Ctrl+L**).

```
#ifndef HAVE_CLUTTER
        t->field->redraw();
        t->scoreFrame->scoreLines(0);  //_ARM_ refresh score
        t->preview->previewBlock(blocknr_next, rot_next, color_next);  //_ARM_ refresh next block
        gtk_widget_queue_draw(t->preview->getWidget());  //_ARM_ refresh display
#endif
```

Note: the Gnometris application is built with the macro **`HAVE_CLUTTER`** *not* defined.

⇨ In **scoreframe.cpp** and comment out the lines 123 and 124 in **`ScoreFrame::scoreLines()`**:

```
        //case 0:
                //return level;
```

⇨ Save the changes and choose **Project** > **Build Project**. You can see any messages or errors from building in the **Console** view.

Now you can resume where you left off.

Importing Xaos

The Xaos example project is in a different **.zip** file from the distribution project:

- ⇨ Choose **File > Import... > General > Existing Projects into Workspace > Next >**
- ⇨ Choose **Select archive file** and use the **Browse...** button to select **C:\Program Files\DS-5\examples\Linux_examples.zip**
- ⇨ Click the **Deselect All** button; then check **xaos**.
- ⇨ Click the **Finish** button. If the **Finish** button is disabled then Eclipse will put a message explaining why at the top of the **Import** dialog.

Now **xaos** appears in the **Project Explorer** view. The Xaos example project is supplied pre-built, but we'll clean and rebuild it to show that we can and so that debug information will refer to the correct pathnames on your host (instead of some host in Cambridge). You won't be able to rebuild Xaos unless you've imported the **distribution** project as described above.

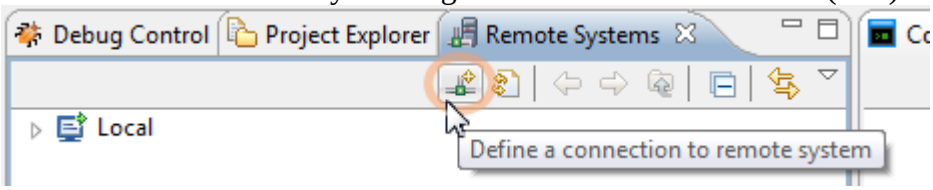
- ⇨ Right-click on **xaos** in the **Project Explorer** view and choose **Clean Project**; then right-click it again a choose **Build Project**. You can see any messages or errors from building in the **Console** view.

Now you can resume where you left off.

Creating a Target Connection

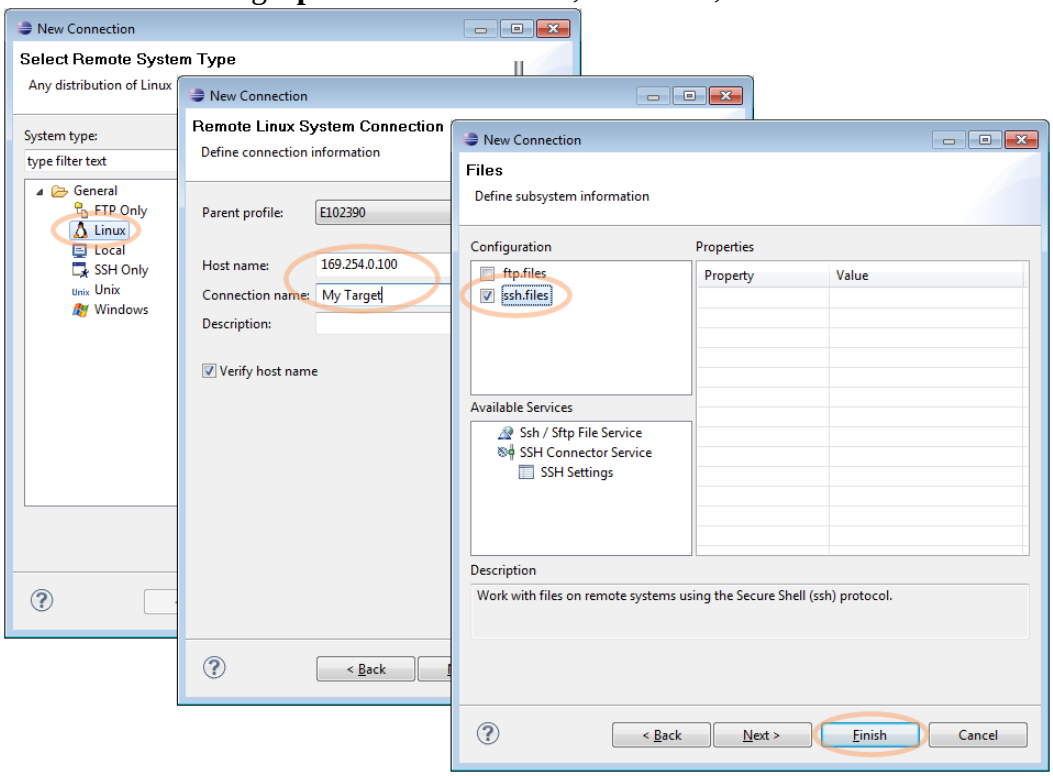
We will establish a connection to the target using Eclipse's Remote Systems Explorer (RSE) so that we can browse its file system and create a terminal connection.

- ⇨ If the **Remote Systems** view is not open, you can open it by choosing **Window > Show View > Other... > Remote Systems > Remote Systems** in any perspective.
- ⇨ Click on the tab of the **Remote Systems** view to bring it to the front.
- ⇨ Create a new connection by clicking the New Connection button ( )



- ⇨ Choose **General > Linux >** "**Next >**"; put the target's IP address, `169.254.0.100` in the **Host name** field and use **My Target** as the **Connection name**; click "**Next >**". (If target is using a different IP address, for example, if your target is connected to a network with a DHCP server, you'll need to determine its IP address by using the serial console or a monitor, keyboard and mouse.)

⇨ Check **ssh.files** then click the **Finish** button. If you click "**Next >**" instead of **Finish**, you want the rest of the default settings: **processes.shell.linux**, **ssh.shells**, and **ssh.terminals**.



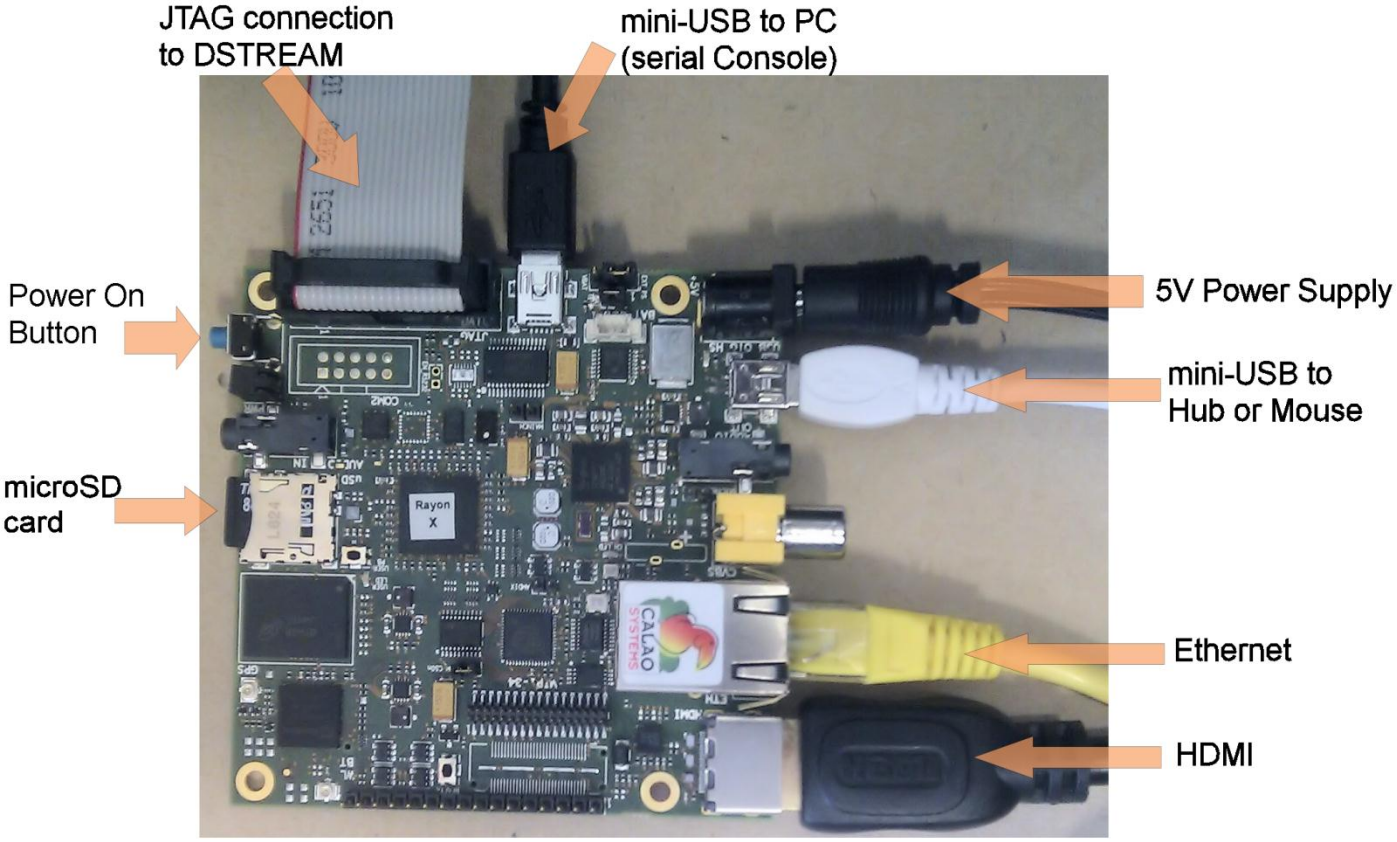
Now you can resume where you left off.

Appendix B: Snowball

Board details and connections

The Snowball board contains a ST-Ericsson Nova A9500 processor (Dual Cortex A9 + Mali 400 GPU). It has 1GB DDR and a 4 or 8 GB eMMC. ("eMMC" stands for "embedded MMC" and is Flash memory built into the Snowball which is accessed as if it were an MMC or SD card.) The board is powered by a 5 Volt power supply. For more information please see www.igloocommunity.org

The following instructions are based on using the 12.03 (that is March 2012) Oneiric Linaro release on the target (see www.linaro.org). There may be differences, for example different file or directory names, in later Linaro releases.



Note that the builtin eMMC can be used instead of the microSD card.

Write a target image to a microSD card

We will write a microSD card (4GB or larger) with an image for the target. The image will contain a bootloader (U-Boot), Linux kernel and root filesystem.

Skip these steps if the target is already setup for you

- ⇨ Using your host, download an image for your target from releases.linaro.org. For a Snowball target, download http://releases.linaro.org/images/12.03.1/oneiric/ubuntu-desktop/snowball_sd-ubuntu-2012.03.1.img.gz to your host and decompress it. (When decompressed it's 3GB).
- ⇨ Now write the decompressed image to a microSD card. On Windows you can use a program like Win32DiskImager https://launchpad.net/win32-image-writer.
- ⇨ With the target power disconnected, install the microSD card in the target.

Serial setup

The Snowball uses a serial console over USB, which we will communicate with using an Eclipse **Terminal** view. This is different, but confusingly similar to the **Terminals** view in Remote Systems Explorer which use ssh over TCP/IP instead of serial.

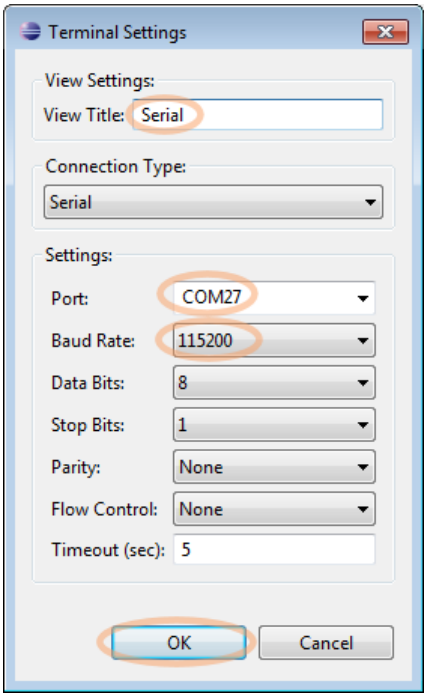
⇨ Connect a USB mini-B cable from your host to the mini USB port next to the 20-pin JTAG connector (shown at the top in the picture above).

⇨ Connect the 5V power supply to the target, but don't push the blue power button yet.

⇨ On a Windows host, you may need to install driver software for the FTDI USB-to-serial chip; see www.ftdichip.com/Drivers/VCP.htm

⇨ Open the **Terminal** view by choosing **Window > Show view > Other… > Terminal > Terminal**.

⇨ Click on the Settings button ( ) to open the **Terminal Settings** dialog.



⇨ Change the **View Title** to **Serial** to make it easier to keep track of.

⇨ Set the **Port** to the name of the serial port on the host (see below).

⇨ Set the **Baud Rate** to **115200**.

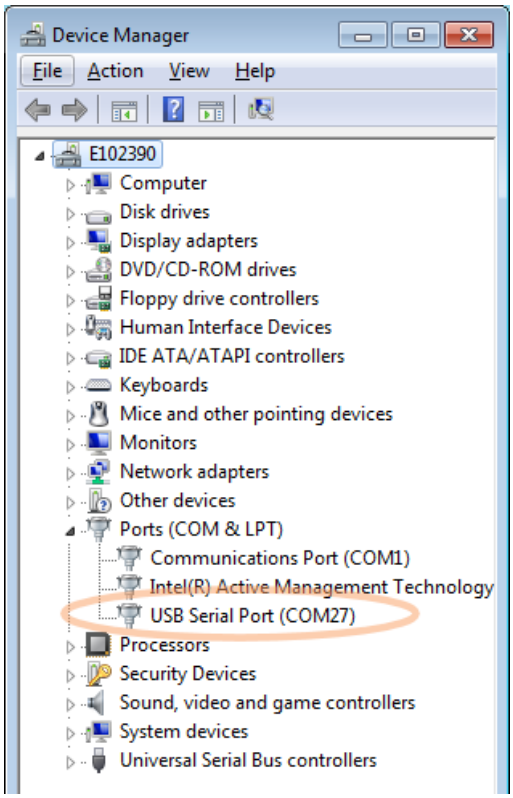
⇨ Use the defaults of 8 data bits, 1 stop bits, no parity and no flow control.

⇨Click **OK** the **Serial** view will connect to the port but there won't be any output yet.

⇨You can drag the tab of the **Serial** view to the right edge of the window to dock it there so that it it's easier to see what it's doing and switch to it.

To determine the COM port on a Windows host:

⇨ Go to the Windows **Control Panel > Device Manager**, expand the 'Ports (COM & LPT)' section to find the **USB Serial Port** that you need to connect to. Your COM port might be different than "COM27":



For example on an Ubuntu Linux host:

⇨ Verify that the serial port is recognized for example run `dmesg | grep ttyUSB`

⇨ Use the device name found, for example `/dev/ttyUSB0`, to connect to the serial port.

The Snowball's serial console is a root shell, but there is no root password on Ubuntu systems. When you're not using the serial console (for instance when you use a **Remote Systems Explorer** terminal), you can login as user=linaro, password=linaro and use sudo.

Flash an eMMC image to the board

You need to do this if your snowball contains firmware or start-up files which are fairly old or incompatible with later Linux releases. The firmware or start-up files can be security aware and may disable Linux kernel JTAG debug. You can turn it off the security checks in the Linux kernel, but you will need to rebuild the kernel to do this. The eMMC images provided from www.igloocommunity.org contains the correct firmware which will allow you to debug the Linux kernel.

To flash an eMMC image you will need a Linux host and the riff tool installed. Please see
http://www.igloocommunity.org/support/Flashing_howto
The eMMC image also contains a "magic" configuration partition
(http://www.igloocommunity.org/support/ConfigPartitionOverview).
More information about the Snowball boot process can be found here
http://www.igloocommunity.org/support/Booting_Guide

Linaro Linux target setup

Now we will use the target itself to download some required software packages from the Internet.

Skip these steps if the target is already setup for you

- Use the **Serial** view to connect to the target (see *Serial setup* above on page 92).
- Connect the target to an Ethernet network with Internet access before powering up/booting the target.
- Press the blue power button. Messages should appear on the serial console as the target boots.

When the target has booted it will prompt `root@linaro-ubuntu-desktop:~#`. You can press return in case the prompt got lost in the messages.

- Install the Streamline gator daemon and driver by executing these commands on the target

```
apt-add-repository -y ppa:linaro-maintainers/arm-ds5
apt-get -y update
apt-get -y install gator
```

- Install ssh by executing this command on the target's serial console:

```
apt-get -y install ssh
```

If you need to use a newer version of gator than is available from the Linaro Package Respository via `apt-get` then you can build it yourself on the target by following these steps:

- Install ssh and g++:

```
apt-get -y update
apt-get -y install ssh g++
```

- Setup an RSE connection and copy `.../DS-5/arm/gator/driver-src` and `.../DS-5/arm/gator/daemon-src` from the host to `/home/linaro` on the target.
- Build the gator driver and daemon:

```
cd /home/linaro
tar -xzf driver-src/gator-driver.tar.gz
make -C /usr/src/linux-headers-`uname -r` ARCH=arm modules M=`pwd`/gator-driver
tar -xzf daemon-src/gator-daemon.tar.gz
make -C gator-daemon CFLAGS='-O3 -Wall -Werror -mthumb-interwork'
```

- Install the just-built gator driver and daemon:

```
stop gator-daemon
rmmod gator.ko
mv /lib/modules/3.3.0-1000-ux500/extra/gator.ko{,.orig}
cp gator-driver/gator.ko /lib/modules/3.3.0-1000-ux500/extra/gator.ko
mv /usr/sbin/gatord{,.orig}
cp gator-daemon/gatord /usr/sbin/gatord
start gator-daemon
```

U-Boot download and debug setup

You need to do this setup if you are going to do the sections on bare-metal debugging of U-Boot, unless it's been done for you.

The following instructions are based on the 12.03 Oneiric Linaro release and may differ slightly (for example directory names) in later Linaro releases. It also assumes that you've completed the previous section to install Linaro 12.03 to your target.

In this section we will download the U-Boot and kernel sources for the Snowball board using the Ubuntu package manager. We'll also build the U-Boot image on the target. Once this is completed we will create projects for these inside DS-5 and copy the sources from the target to our host so that we can use it for debug.

To debug U-Boot on the Snowball board we need to get the sources and build U-Boot. You can do this natively on your target by following these steps:

- ⇨ Use the **Serial** view to connect to the target (see *Serial setup* above on page 92).
- ⇨ Connect the target to an Ethernet network with Internet access before powering up/booting the target.
- ⇨ Turn on the device and boot the Linaro Linux image

Skip these steps if the target is already setup for you

- ⇨ In the serial port change directory to /home/linaro

```
cd /home/linaro
```

- ⇨ Download U-Boot

```
apt-get -y source u-boot-ux500
```

- ⇨ Build U-Boot

```
cd u-boot-ux500-2009.11
```

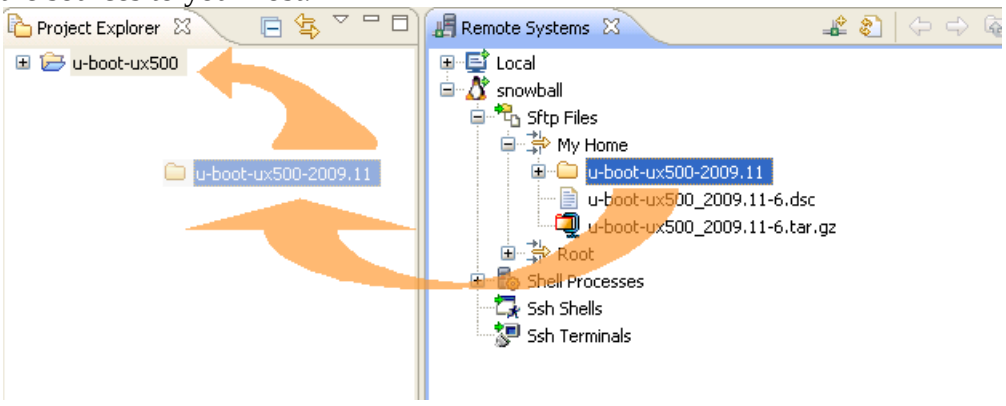
@@@ The next line is to work around a bug

```
sed -e 's/icache_enable/\/\/icache_enable/' \
      cpu/arm_cortexa9/db8500/cpu.c > cpu.c && \
      mv cpu.c cpu/arm_cortexa9/db8500/cpu.c
make u8500_snowball_config
make -j2
make clean
```

The make clean is just to remove some object and library files that we don't need for debugging.

⇨ In DS-5 Eclipse create a new General project **File** > **New Project** > **General** > **Project** and name the project *u-boot-ux500*

⇨ Open the **Remote Systems** view and copy the folder **`/home/linaro/u-boot-ux500-2009.11`** into the **`u-boot-ux500`** project in the **Project Explorer** view. (If you have not yet connected to your target please see *Creating a target connection* in the appendix on page 90) It will take a few minutes to copy all the sources to your host.



Linux kernel download and debug setup

You need to do this setup if you are going to do the sections on kernel and module debugging, unless it's been done for you.

To debug the Linux kernel we need to use the target to fetch the debug symbols (vmlinux) and the corresponding sources:

- ⇨ Use the **Serial** view to connect to the target (see *Serial setup* above on page 92).
- ⇨ Connect the target to an Ethernet network with Internet access before powering up/booting the target.
- ⇨ Turn on the device and boot the Linaro Linux image

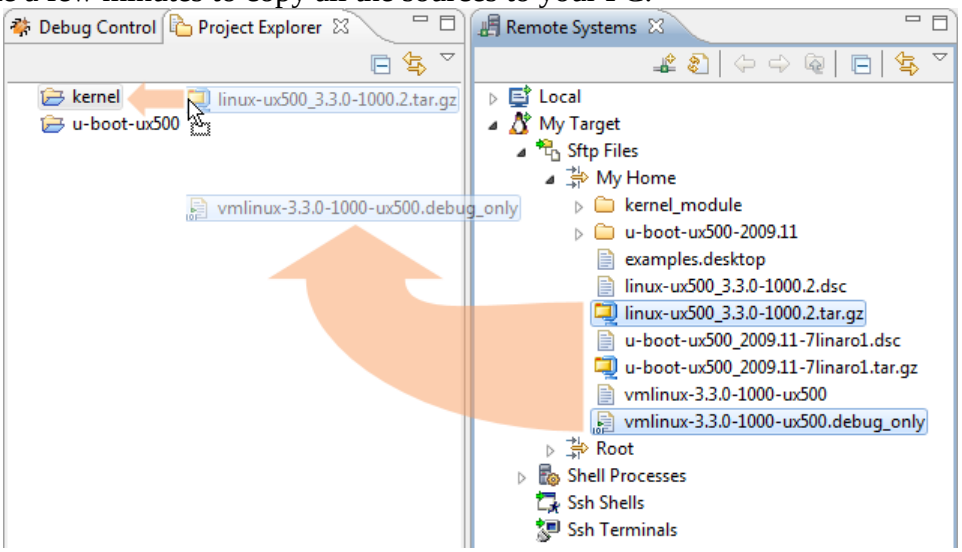
Skip these steps if the target is already setup for you

- ⇨ In the serial port change directory to
  `cd /home/linaro`
- ⇨ In order to find the kernel sources, duplicate the `deb` line in `/etc/apt/sources.list.d/hwpack.linaro-landing-team-ste.list` and replace `deb` with `deb-src`
- ⇨ In the serial port download the kernel debug symbols (`vmlinux`)
  `apt-get -y --force-yes install linux-image-$(uname -r)-dbgsym`
- ⇨ Find the `vmlinux` file containing the kernel image and debug symbols and copy it to the current directory. Then create a copy with only the debug symbols:

```
cp $(dpkg -L linux-image-$(uname -r)-dbgsym | grep vmlinux) /home/linaro
objcopy --only-keep-debug vmlinux-3.3.0-1000-ux500 \
                          vmlinux-3.3.0-1000-ux500.debug_only
```

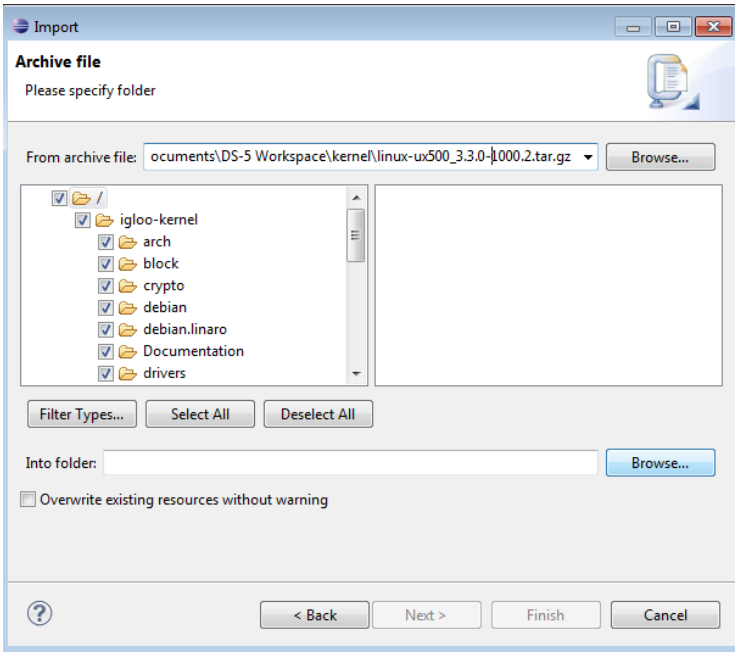
- ⇨ Download the kernel sources
  `apt-get --force-yes -d source linux-image-$(uname -r)`

- ⇨ In DS-5 Eclipse create a new General project **File** > **New Project** > **General** > **Project** and call the project *kernel*
- ⇨ Open the **Remote Systems** view and copy `vmlinx-3.3.0-1000-ux500.debug_only` and `linux-ux500_3.3.0-1000.2.tar.gz` to the **kernel** project in the **Project Explorer** view (If you have not yet connected to your target please see *Creating a target connection* in the appendix on page 90) It will take a few minutes to copy all the sources to your PC.



- ⇨ Import the kernel sources by right clicking on the **kernel** project folder and choosing **Import** > **General** > **Archive File** > **Next**; click on the **Browse…** button and choose the kernel source archive `linux-ux500_3_3_0.1000.2.tar.gz` in your workspace. Click **Finish** to start the import, then go

have some tea – this will take some time:



You will get a message that says you encountered errors during the import; ignore it and click **Cancel** to not import the source again. This is due to Windows not understanding symbolic links in the archive nor file names that differ only in upper/lower case.

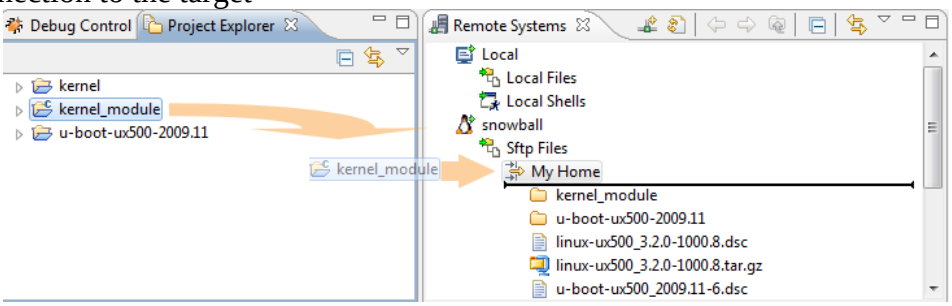
Kernel module debug (modex) build and setup

DS-5 ships with a kernel module debug example called **modex**. To run the example the kernel module needs to be built against the kernel on your target. We can do this by copying the source to the target and building the example natively.

- ⇨ Choose **File > Import... > General > Existing Projects into Workspace > Next >**
- ⇨ Choose **Select archive file** and use the **Browse...** button to select **C:\Program Files\DS-5\examples\Linux_examples.zip**
- ⇨ Click the **Deselect All** button; then check **kernel_module**.
- ⇨ Click the **Finish** button. If the **Finish** button is disabled then Eclipse will put a message explaining why at the top of the **Import** dialog.

Now **kernel_module** appears in the **Project Explorer** view. The kernel module example project is supplied pre-built for the RSTM and Beagleboard images provided in DS-5 but not for Snowball. You won't be able to rebuild the kernel module on a windows machine and you also need the kernel headers to build it. So we will rebuild it on the target.

- ⇨ Copy the **kernel_module** project over to your target by dragging it into the **Remote Systems** connection to the target

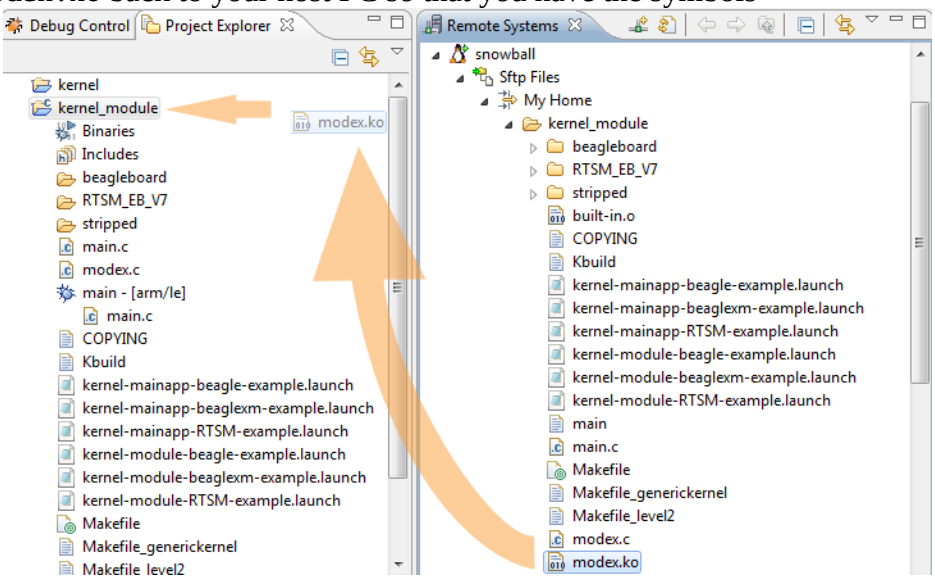


Skip these download steps if the target is already setup for you

Next we'll build the kernel module natively on the target and then copy it back onto our PC. In your **Serial** view to the target run the following commands

- Change directory to the kernel module
  `cd /home/linaro/kernel_module`
- Download the kernel headers from the internet to the target
  `apt-get -y --force-yes install linux-headers-$(uname -r)`
- Re-configure the kernel
  `make -C /usr/src/linux-headers-$(uname -r) oldconfig`
- Build the kernel module
  `make -C /usr/src/linux-headers-$(uname -r) M=$(pwd) modules`

- Copy `modex.ko` back to your host PC so that you have the symbols



You are now ready to debug the kernel module example.

Setting a static IP address

You can set a static IP address on the target by editing `/etc/network/interfaces` on the target and adding the following lines to the file. You can change `169.254.0.100` to the address that you want the target to use.

```
auto eth0
iface eth0 inet static
      address 169.254.0.100
      netmask 255.255.0.0
```

You'll need to reboot the target for the changes to take effect. To make sure the changes are written to the disk (microSD card) execute the commands `sync;sync` on the console before you reboot it.

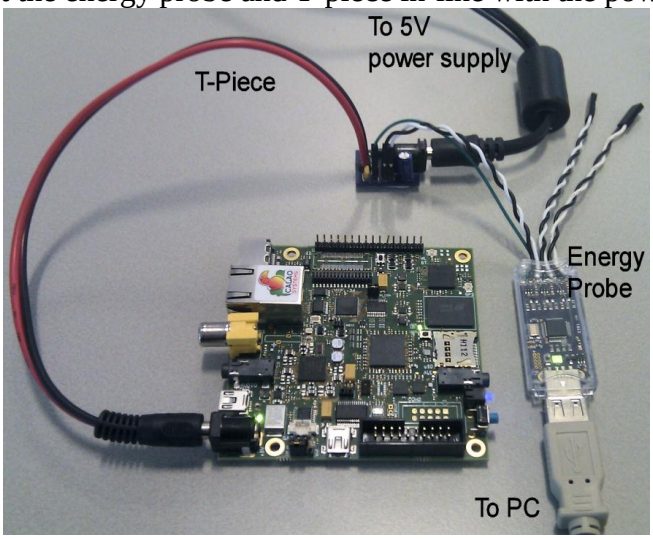
With the `eth0` entry in `/etc/network/interfaces` NetworkManager will no longer attempt to manage the interface. For use on a cross-over cable, you can now use `ifup eth0` to enable the interface with the static IP address and `ifdown eth0` to disable it. If you attach the Ethernet port to a network with a DHCP server, you can use `dhclient -1 eth0` to get an address from the DHCP server.

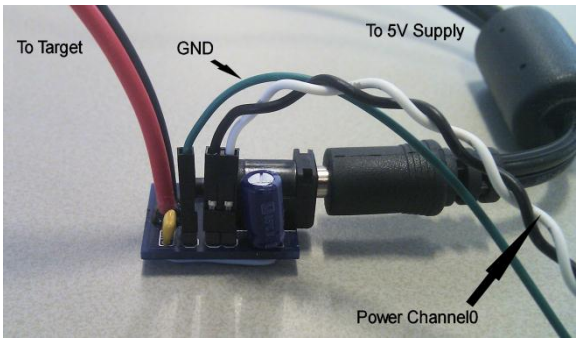
Note: For the Ethernet to be setup automatically you need to have the Ethernet cable connected when you boot the target.

ARM Energy Probe setup

The ARM Energy probe can measure the power of up the three channels or measurement points at a time. If your board has power measurement points you can connect the probe to those points and specify the resistor values in the Capture & Analysis Options dialog. The snowball board does not contain any soldered on measurement points for us to use. Instead, we will use the T-Piece (which contains a 20mOhm shunt resistor) to measure the total power consumed by the board.

Connect the energy probe and T-piece in-line with the power supply as shown below:





When you first connect the probe to your PC you will need to install a driver from the DS-5 installation. On windows this is typically **`C:\Program Files\DS-5\sw\energy_probe`**. Once the driver is installed, the probe connected and the target powered on the Energy Probe will display a green LED.