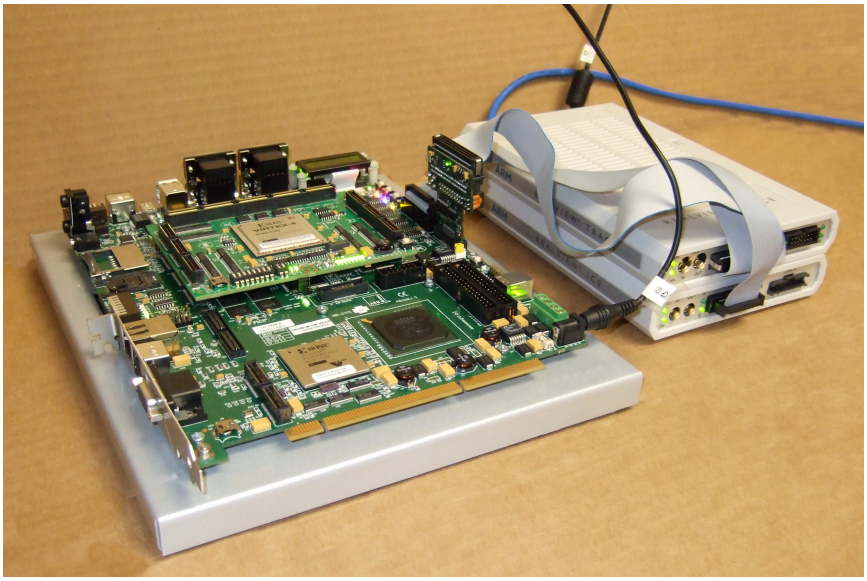
RVDS 3.1
RealView ICE
&
RealView Trace
Quickstart Tutorial



250v02

Contents

Introduction

Aim

This tutorial will get you started using RealView ICE (RVI) and RealView Trace (RVT) hardware with RealView Debugger (RVD) software. The versions used are:

- RVI host software v3.1
- RVI firmware v3.1
- RVD v3.1

It contains sections covering the essentials of installing and using:

- RealView ICE
- RealView Trace

Pre-requisites

The example code used in this tutorial will work with the memory map of an ARM926EJ-S Platform Baseboard (PB926EJ-S), but is designed in such a way that it can be ported to the memory map of your system with minimal effort. This platform has 128MB of RAM at address 0x0 once the Boot Monitor has run, but the examples should work with any target with at least 8KB of programmable memory at this location.

Additional information

This tutorial does not provide detailed documentation of RVD, RVT and RVI. Full documentation is provided with the products.

References in the subsequent sections of the tutorial are identified by these boxes.

Further help can be accessed by pressing *F1* from within RVD, or from the help menu.

Full documentation is available in PDF format. This can be found by going to *Start → Programs → ARM → RealView Development Suite v3.1 → RVDS v3.1 Documentation Suite*.

When working through this tutorial, of particular interest are:

- *RealView Debugger v3.1 User Guide*
- *RealView ICE and RealView Trace v3.1 User Guide*
- *RealView Debugger v3.1 Trace User Guide*
- *RealView Compiler Tools v3.1 Linker and Utilities Guide*

Other useful information can be found on the www.arm.com web site:

- *Application Note 168: Tracing with RVD* (http://www.arm.com/documentation/Application_Notes/index.html)
    - Contains a more advanced tutorial for RVD Trace
- FAQs (http://www.arm.com/support/devfaqsindex.html)

- Answer common RVD, RVI and trace questions

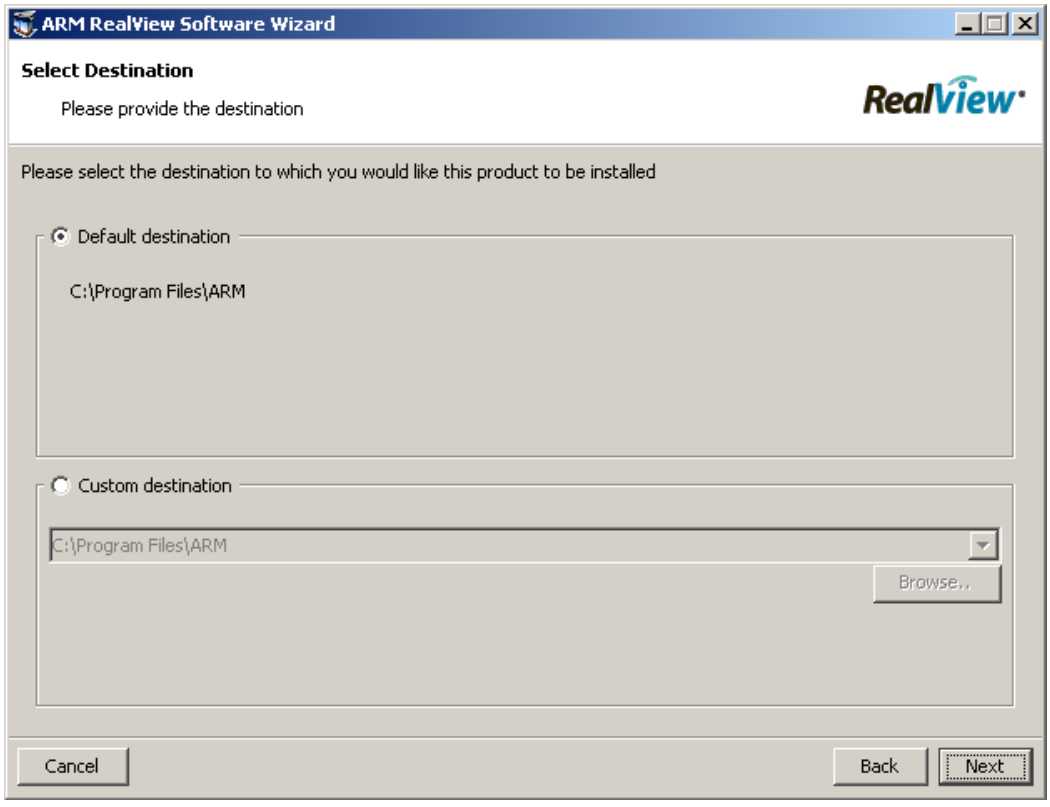
Section 1: RVDS and RVI Installation

This section describes how to install the required software on your host PC, and configure your RVI and RVT.

RedHat Enterprise Linux 4 is the only Linux platform supported by RVDS 3.1. References to *Linux* in this tutorial apply to RHE4.

Install the RealView Development Suite (RVDS) software

Follow the instructions on the RealView Development Suite (RVDS) v3.1 CD for installation. It is recommended that you install to the default location. If you are installing on RedHat Enterprise Linux 4, you should use the setuplinux.bin installer. On Linux you will need to edit your .bashrc file to include the line `source /<path>/RVDS31env.posh` where <path> is your installation path.



After installing RVDS, RVD can be launched by:

- On Windows, navigating to *Start* → *Programs* → *ARM* → *RealView Development Suite v3.1* → RealView Debugger v3.1
- On Linux, by typing **`rvdebug`** at a command shell prompt (note that this requires .bashrc to be updated as detailed above)

RVD configuration files are stored at:

- On Windows, `C:\Documents and Settings\<username>\Application Data\ARM\rvdebug\3.1\`
- On Linux, `/home/<username>/.rvdebug/3.1/`

You can revert to the default configuration by starting RVD with the **`--cleanstart`** switch. At a command prompt, type **`rvdebug --cleanstart`** and press enter.

You can revert to the default configuration by starting RVD with the **`--cleanstart`** switch (from a command line, type **`rvdebug --cleanstart`** and press enter).

Install the RealView ICE host software

Follow the instructions on the RealView ICE v3.1 CD for installation. Make sure that you install the software in the folder you are using for all of your ARM RealView tools. If you are installing on RedHat Enterprise Linux 4, you should use the `setuplinux.bin` installer.

Connecting to your RealView ICE unit

You can connect to your RealView ICE unit using one of 3 methods:

- via a USB cable (Windows only)
- via a Local Area Network (LAN) with or without DHCP
- via an Ethernet cross-over cable

By default, RVI is preconfigured to work correctly via USB or a DHCP-enabled network without additional configuration.

This document will briefly explain how to set up a connection via a USB cable.

If you are using Linux, you should connect to your RVI via Ethernet.

To set up an ethernet connection on a DHCP-enabled network:
- Follow step 1 below
- Connect the RVI to your network using the supplied Ethernet cable
- Follow on from step 5 below, noting that your RVI will appear under *TCP/IP*

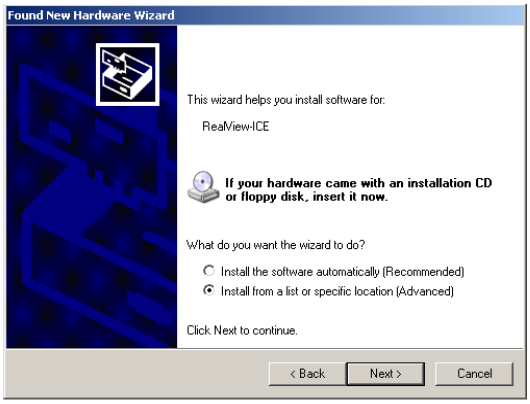
For a more detailed explanation, or for instructions for connecting via Ethernet, refer to Chapters 2 and 3 (‘*Getting Started*’ and ‘*Configuring RealView ICE Networking*’) of the *RealView ICE and RealView Trace v3.1 User Guide*.

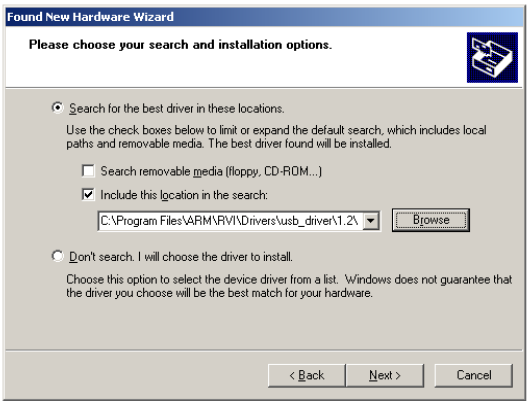
Connection via USB

1. Plug the power supply for the RealView ICE into the unit.

2. Connect the RealView ICE to your computer's USB port using the supplied cable.

3. Windows should automatically start the Found New Hardware Wizard. When prompted, browse to the driver stored in the `C:\Program Files\ARM\RVI\Drivers\usb_driver\1.2\6\win_32-pentium` folder.

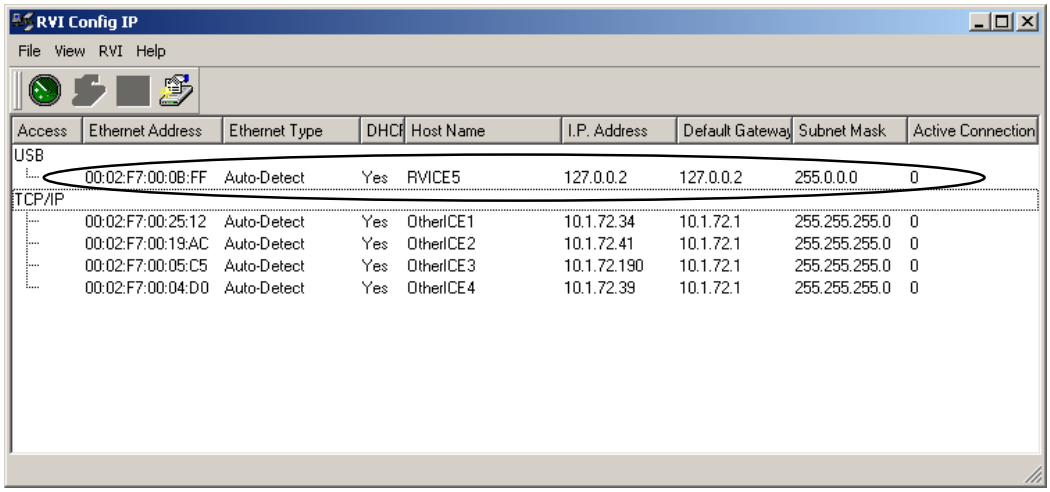




4. Restart your computer and the RealView ICE unit.

5. Confirm that your RVI can be detected using the RVI Config IP utility. This utility can be found at *Start → Programs → ARM → RealView ICE v3.1 → RealView ICE Config IP*.

In *RVI Config IP*, select *RVI → Start Scan* to scan for ICEs.



Your RVI should be listed under *USB*. There may be other ICEs on your network listed under *TCP/IP*.

Right-click on your ICE and select *Identify*.

This should cause the LEDs on the front of your RVI unit to flash.

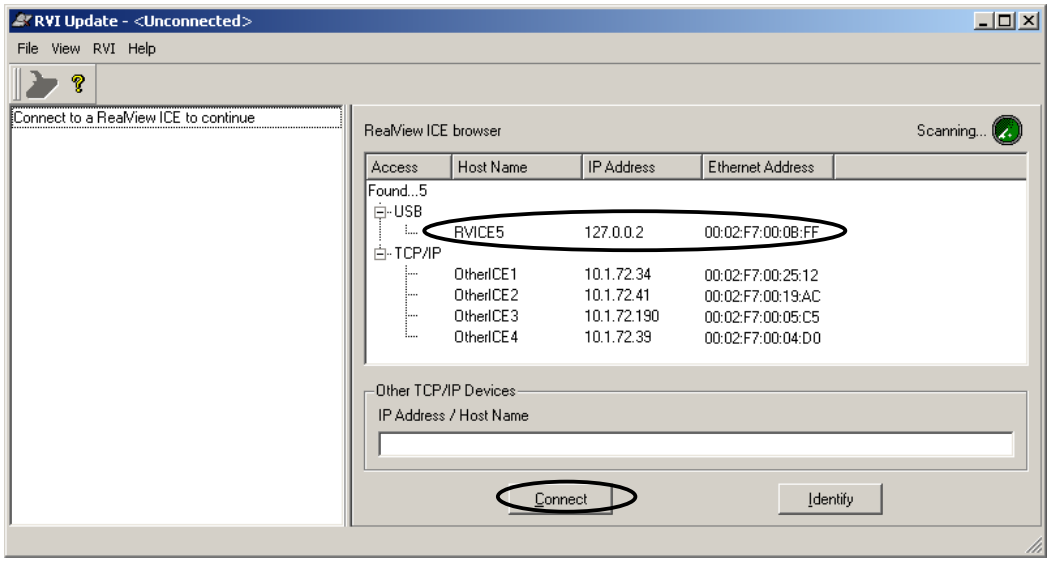
If you want to name your RVI, you can do this by right-clicking on the RVI, selecting *Configure*, and entering a *Host Name*. Click *Configure* to confirm.

Install the RealView ICE firmware

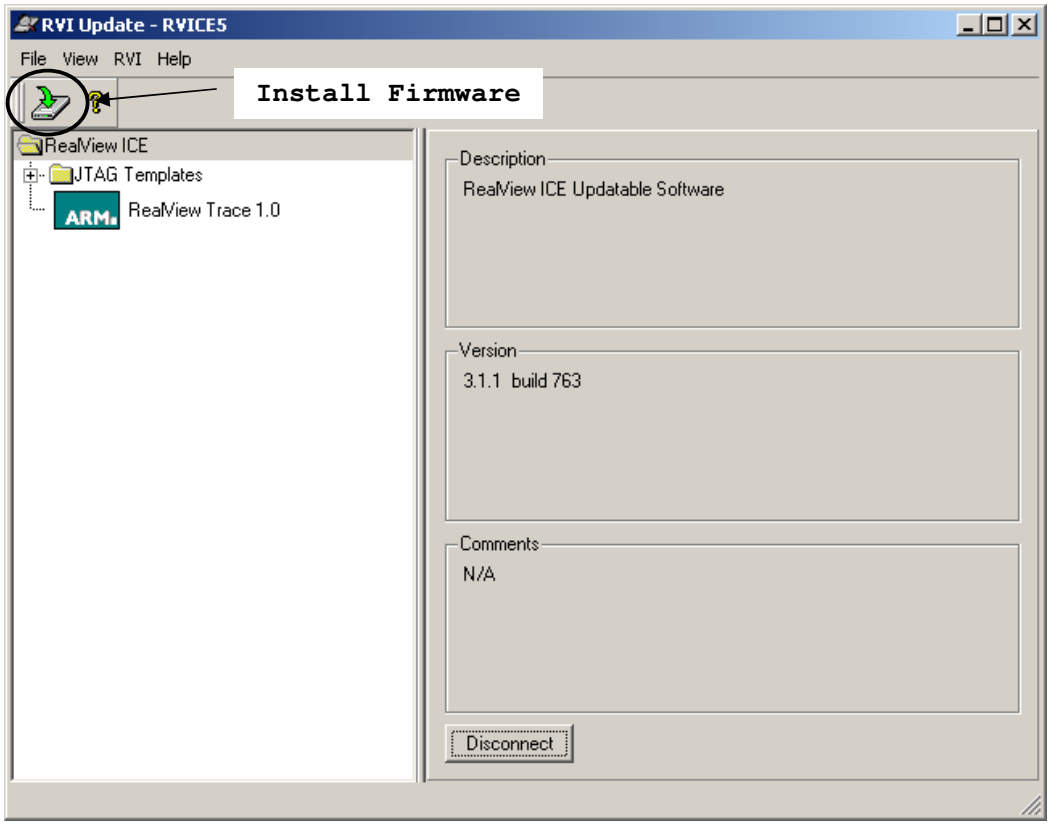
You should update your RVI to the latest firmware, which can be found in your RVI installation folder.

Open the RVI Update utility. This utility can be found at *Start → Programs → ARM → RealView ICE v3.1 → RealView ICE Update*.

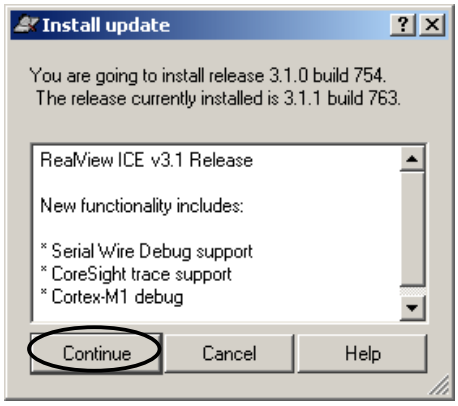
Click on your ICE and select *Connect.*



Click on the ICE (*Install Firmware*) button with a green arrow in the top left of the window.



Browse to the `C:\Program Files\ARM\RVI\Firmware\3.1\23` folder and select `ARM-RVI-3.1.0-754-base.rvi`. Click *Open*. Click *Continue* in the next dialog.



The firmware update will take around 2-3 minutes to complete.

Browse to http://www.arm.com/support/downloads/rvi.html and download the latest RVI 3.1 patch. At the time of writing, the latest patch is RVI 3.1.1 (build 763). Extract the downloaded ZIP file to your hard drive.

As before, click on the ICE button with a green arrow, and browse to the `.rvi` file that you just extracted. Select *Open*, and then *Continue* in the following dialog.

Patching will again take around 2-3 minutes to complete. After completion, your RVI is now updated to the latest available RVI firmware revision.

Setting up RealView Trace

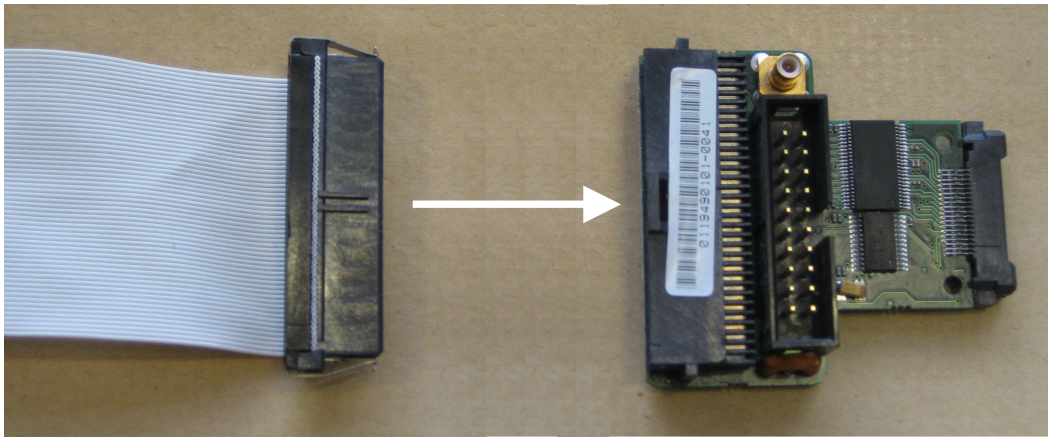
The RealView Trace unit should be securely mounted on the RVI unit. Note that the Trace unit requires no additional software in order to work.

Refer to section 6.4.2 (‘*Connection Instructions’)* of the *RealView ICE and RealView Trace v3.1 User Guide*. for more information on mounting the RVT unit onto the RVI unit.

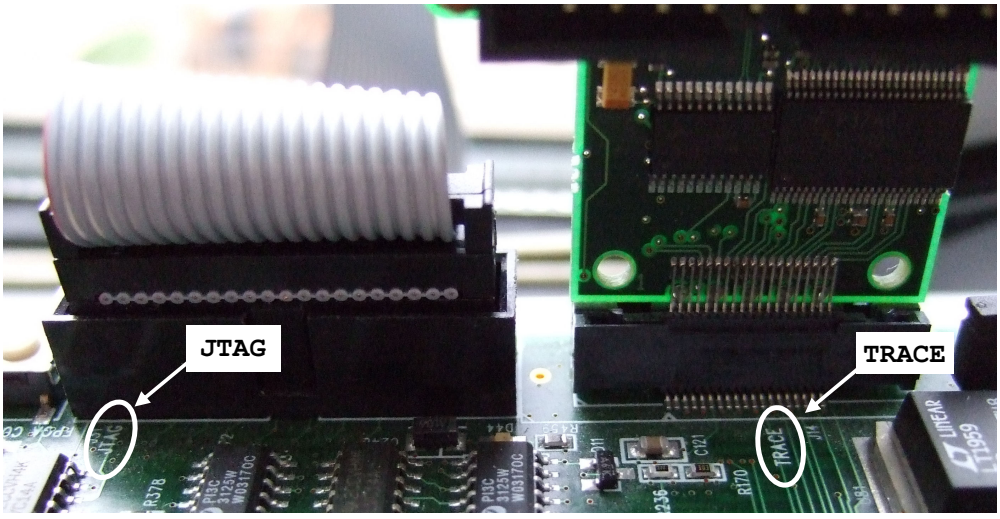
Connecting the RealView ICE and RealView Trace to your target hardware

Connect one end of the provided JTAG cable to the RealView ICE unit. Connect the other end of the cable to the socket marked *JTAG* on the target board.

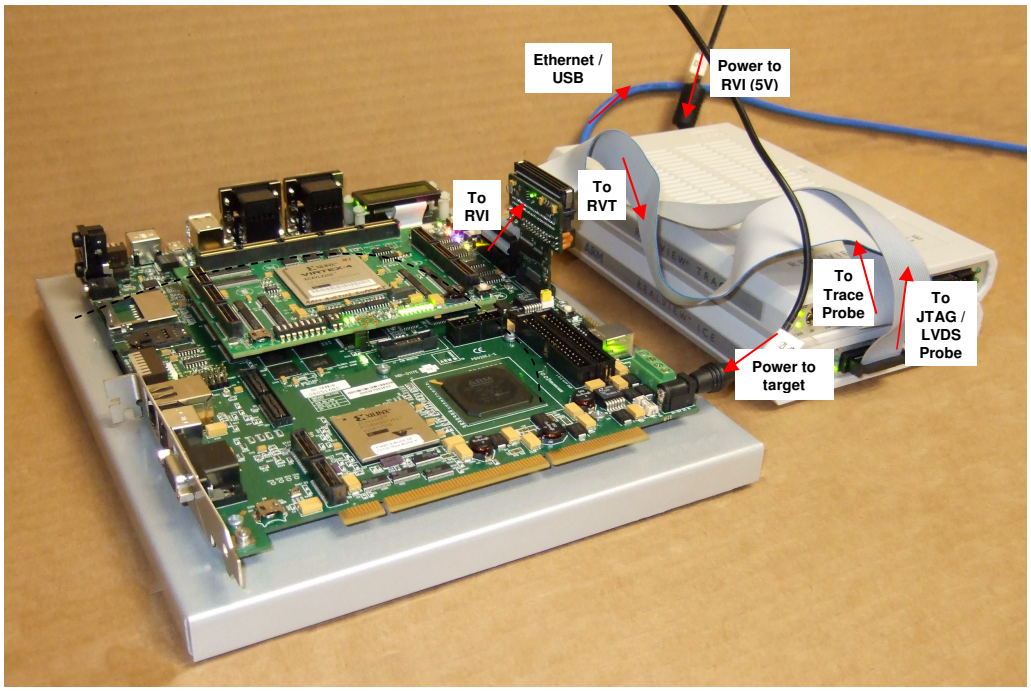
Connect one end of the provided Trace cable to the small Trace ‘T piece’ adapter.



Plug the adapter into the MICTOR connector marked *TRACE* on your target. Connect the other end of the cable to your RealView Trace (RVT).

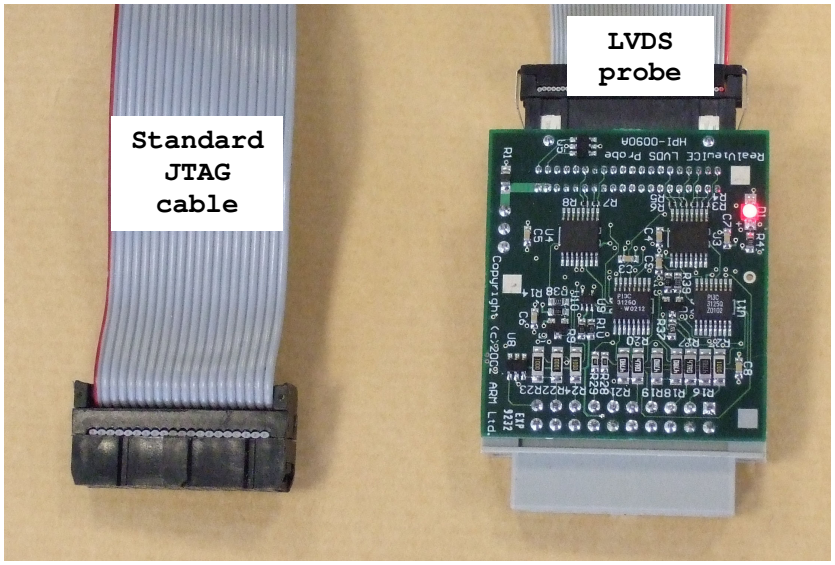


The RealView Trace unit does not need additional power; it can obtain power directly from the RealView ICE.



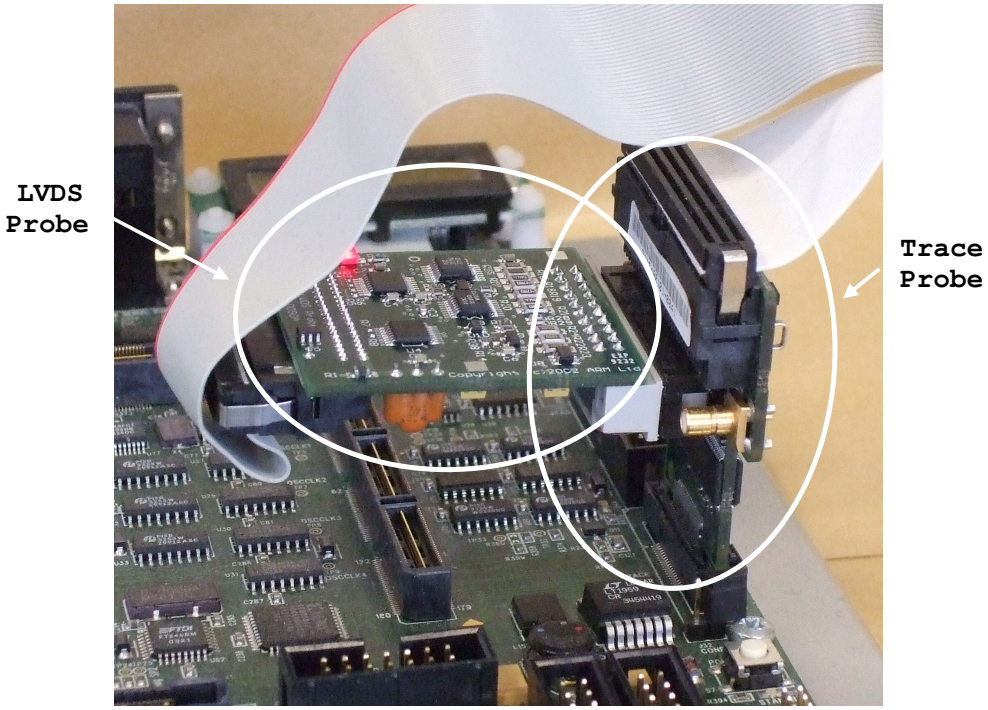
It is recommended that you use the LVDS JTAG probe in preference to the standard JTAG cable, as the LVDS probe:

- Lets you debug systems with a faster JTAG clock (as long as the target permits it). For TCK speeds of 20MHz or more you need to use the LVDS probe.
- Helps to avoid some issues related to weak JTAG signals, or JTAG signals with interference.
- Has a longer cable, enabling debugging when the ICE is further away from the target.



On some boards (including the PB926EJ-S), the JTAG and MICTOR connectors are too close together to plug both the LVDS JTAG and Trace probes into the board. In

this case, you can plug the LVDS JTAG connector into the side of the Trace connector, as shown below.



Section 2: Preparing for the Examples

This section prepares the debugger and the examples for the remainder of the tutorial.

2.1 – Building an Image (rebuilding the examples for your target)

The examples in this tutorial make use of 3 pieces of code:

- A Hello World example that outputs text, making use of RVD's *STDIO* tab
- A reset example, containing a vector table and some initialisation code, to demonstrate running a program at reset time
- A version of the classic Dhrystone benchmark modified to run indefinitely, to demonstrate trace capture

The code examples in this tutorial are provided with batch files that call the C compiler (*armcc*) and linker (*armlink*) to build the images. You will need to invoke the **`.bat`** batch files in each example folder in order to build an **`.axf`** image that can be loaded to your target. These examples will work without modification on the Versatile PB926EJ-S platform.

c:\rvds31_tutorial\PB926 — These are the files for use with the PB926EJ-S board

The examples require 8KB of memory at address 0x8000. If you are working with a PB926EJ-S, or another target board that meets this requirement, then you should skip to the next section.

If you do not have 8KB of memory at address 0x8000, then you should follow the remainder of this section, which gives more information on rebuilding the examples to work with your target.

The address that *armlink* will link the image to execute from is specified by a scatter file. The scatter files used for the examples all have the file extension *.scat*.

Scatterfiles describe where code and data are stored at load time and at run time. In the example below, the scatter file is also used to locate the stack and heap.

```
LOAD 0x8000 <-------- Modify this value
{
        RAM +0
        {
                *(+RO,+RW,+ZI)
        }

        ARM_LIB_STACKHEAP +0x1000 ALIGN 32 EMPTY 0x1000
```

```
        {
        }
}
```

This example scatter file creates one load region at 0x8000. Within this load region is an execution region called **RAM** at address **+0**, indicating that the address is at an offset of 0 from the load region address (i.e. 0x8000). This execution region contains all the code and data for the image.

A second execution region is called **ARM_LIB_STACKHEAP**, and is marked as **+0x1000 ALIGN 32** so that it will be placed on the next 32 byte boundary that is >= 0x1000 bytes from the end of the **RAM** region. It is **0x1000** bytes in size and is marked as **EMPTY** because it holds no code or data sections.

ARM_LIB_STACKHEAP is a key region name. In RVDS 3.0 and later, this execution region name tells the linker where you want to place the stack and heap, causing the linker to automatically link in all of the necessary code to set up the stack and heap accordingly.

In this example the heap will grow upwards from the beginning of the region **ARM_LIB_STACKHEAP** and the stack will grow downwards from the end of the region **ARM_LIB_STACKHEAP**. The absolute addresses that the heap and stack grow from will depend on the size of the **RAM** region.

In order to port this example to work with a target other than the PB926EJ-S, you would simply need to adjust the **LOAD** address (0x8000) to be an address in RAM on your target, allowing enough space above the chosen address to fit in the image including the stack/heap region (8KB is recommended).

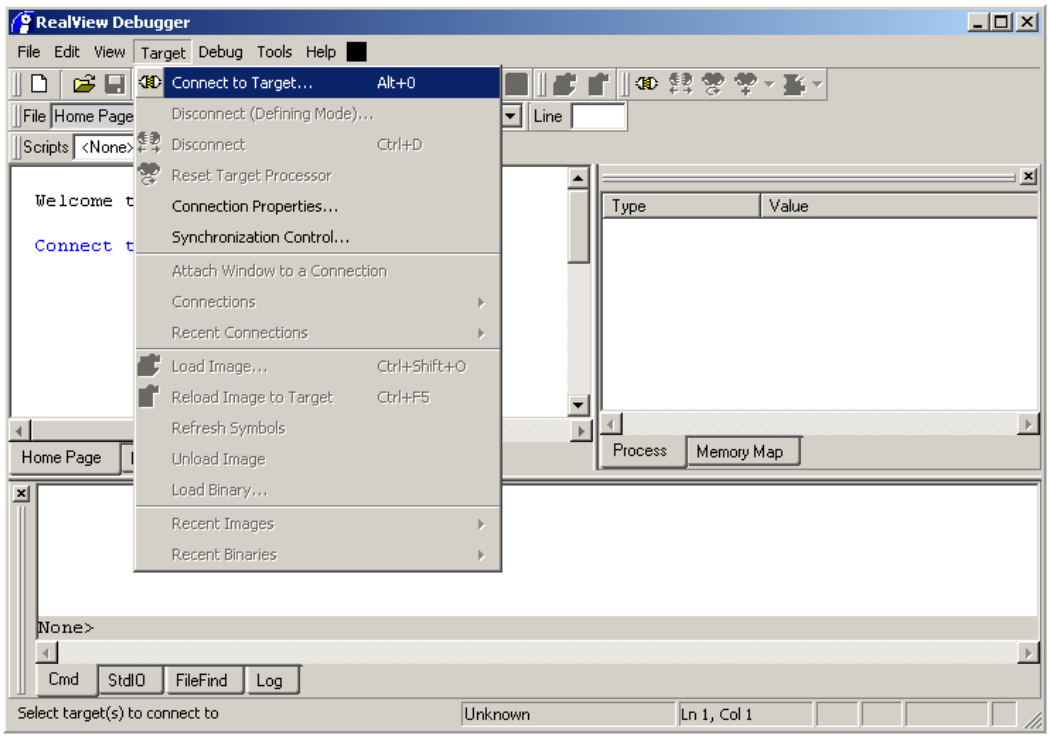
Refer to Chapter 5 of the RVCT 3.1 Linker and Utilities Guide ('*Using Scatter-loading Description Files*') for more information on using scatterfiles.

After making changes to the scatterfile or to the tools' command lines, you must rebuild the image to implement these changes. This can be done by invoking the **.bat** build script again.

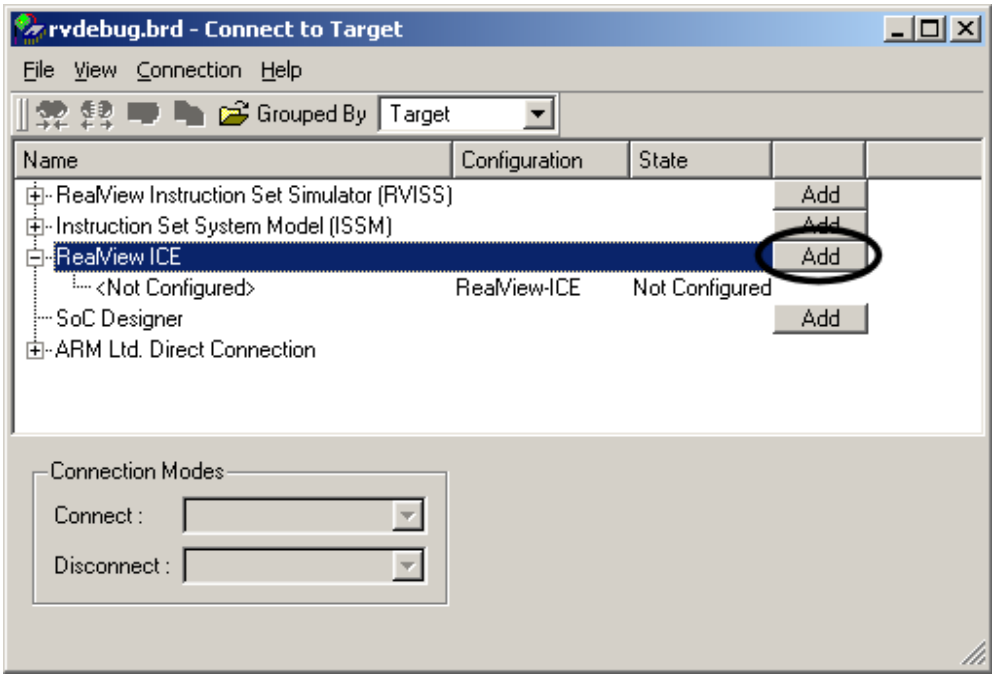
2.2 – Connecting to and Configuring your Target

Start RealView Debugger by going to *Start* → *Programs* → *ARM* → *RealView Development Suite v3.1* → *RealView Debugger v3.1*

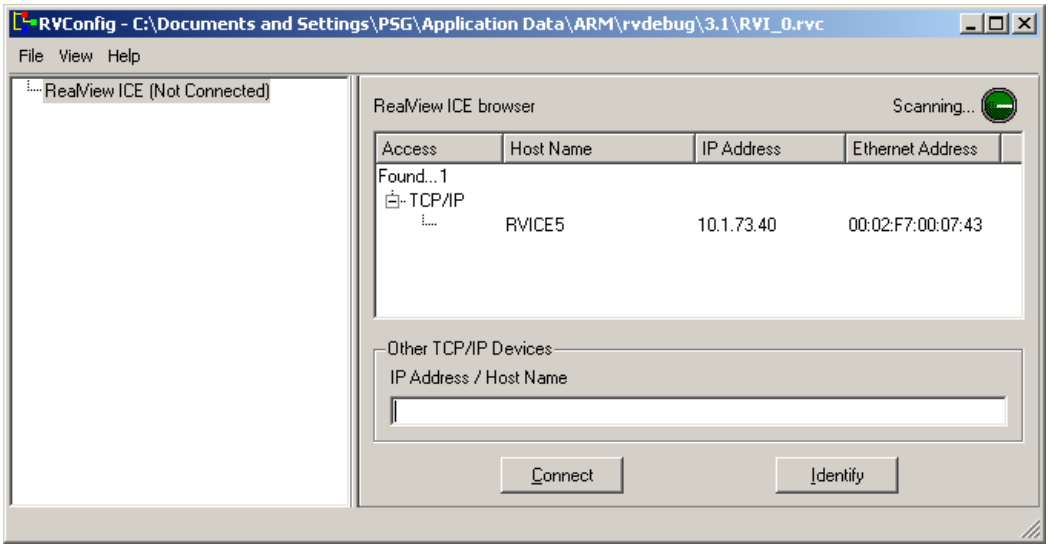
Select *Target* → *Connect to Target…*



Click on the *Add* button to the right of the *RealView-ICE* entry in the *Connect to Target* window.

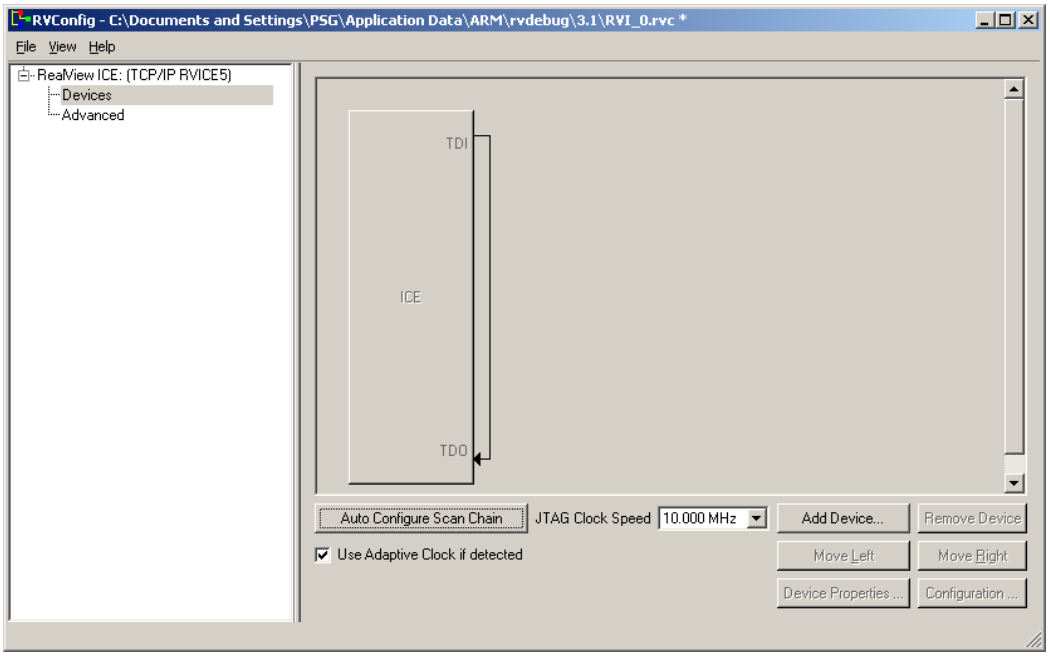


The *RVConfig* window appears. RVD should automatically detect your RealView ICE unit. If it does not, click on the green icon at the top-right hand corner of the *RVConfig* window to begin scanning.



Select your RealView ICE unit from the list and click *Connect*.

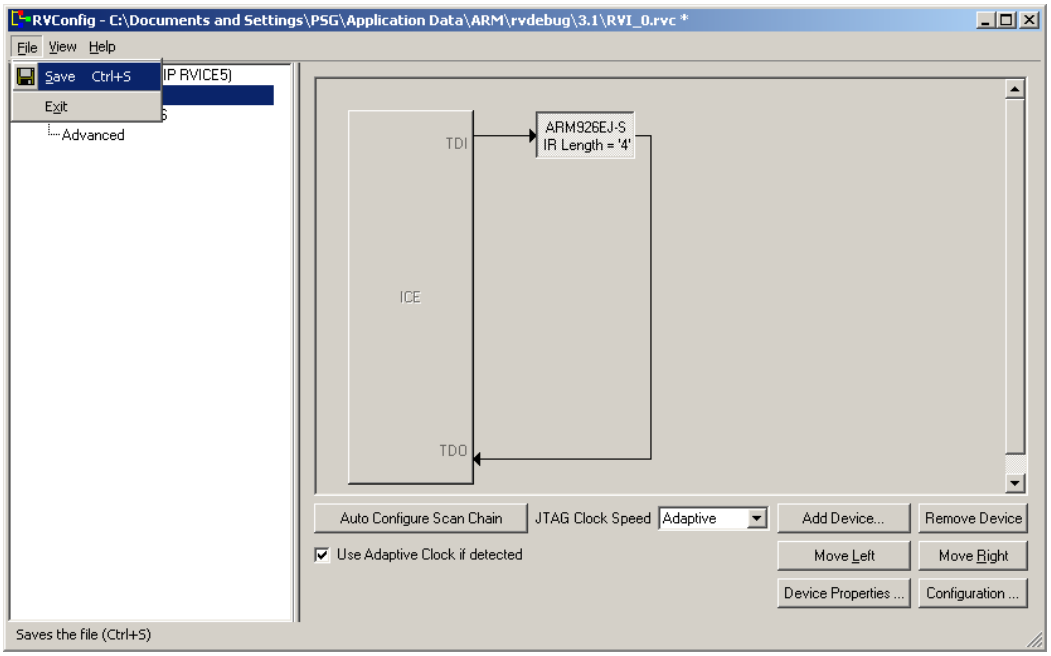
Additional options will appear:



Auto Configure Scan Chain causes the RVI to scan for devices in the target's scan chain. Each detected device is added to the tree diagram.

Click the *Auto Configure Scan Chain* button.

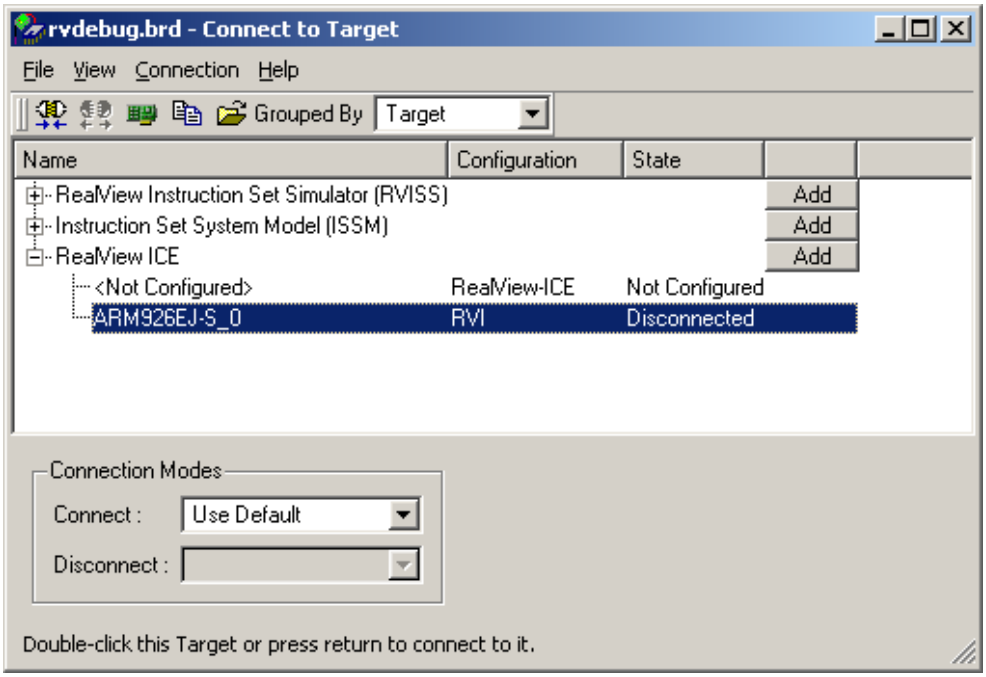
Your target should appear in the list:



If you are using a target that is not recognized by *Auto Configure*, you should select *Add Device…* and navigate to the ARM core on which your target is based. If you are manually configuring your target in this way, you will need to fully populate the scan chain with *Custom UNKNOWN* (in *Add Device…*) entries if there are items other than the core on the scan chain (e.g. DSPs, FPGAs etc).

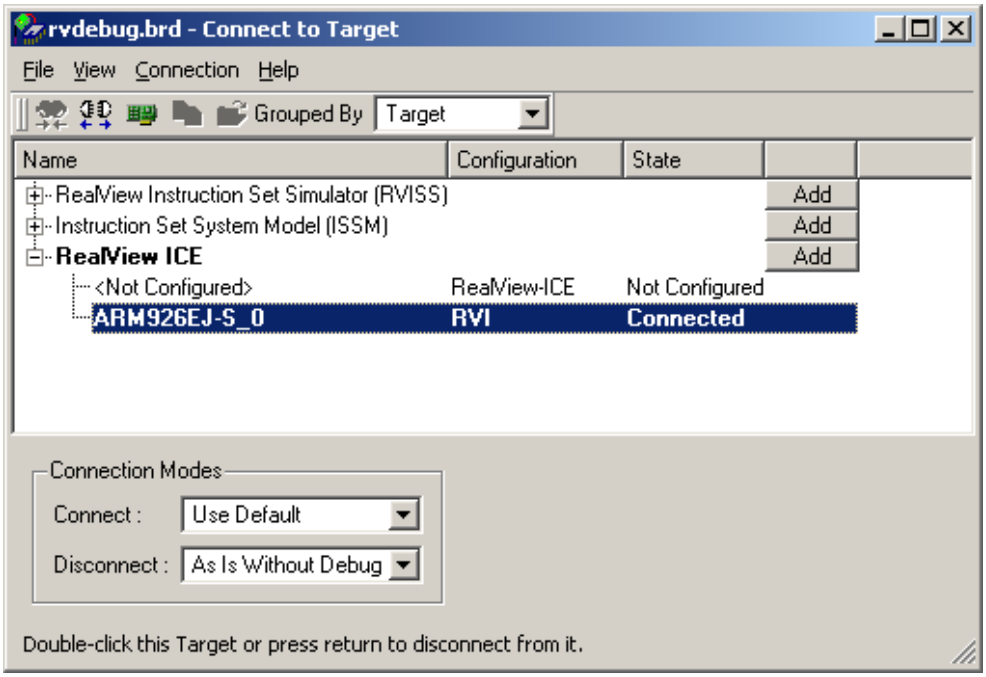
Select *File→Save*, then select *File →Exit* to save your configuration and close the *RVConfig* window.

Select your target underneath the *RealView ICE* entry in the *Connection Control* window.



Select *Connection →Connect* from the menu to connect to your target.

Double-clicking on the connection name (**`ARM926EJ-S_0`**) will also cause RVD to connect to your target.

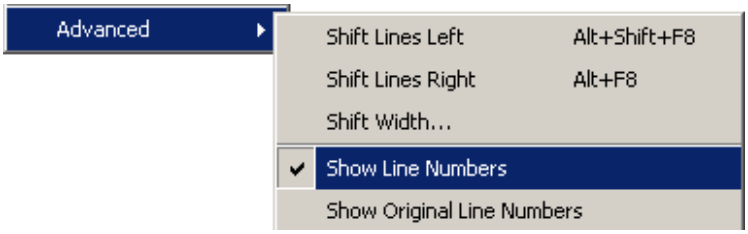

rvdebug.brd - Connect to Target
File View Connection Help
Grouped By Target
Name
Configuration
State
RealView Instruction Set Simulator (RVISS)
Instruction Set System Model (ISSM)
RealView ICE
<Not Configured>
RealView-ICE
Not Configured
ARM926EJ-S_0
RVI
Connected
Add
Add
Add
Connection Modes
Connect :
Use Default
Disconnect :
As Is Without Debug
Double-click this Target or press return to disconnect from it.

2.3 – Setting up RealView Debugger

Line Numbers

Line numbers are used through this tutorial to identify specific source lines.

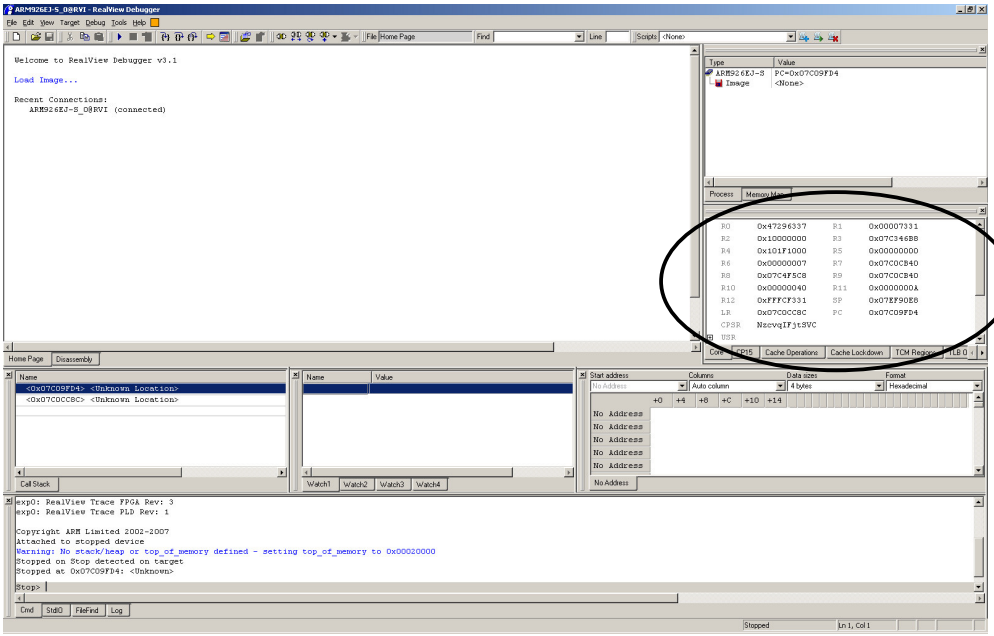
If you do not already have source line numbering enabled, select *Edit → Advanced → Show Line Numbers* to display the source file line numbers in the code window.



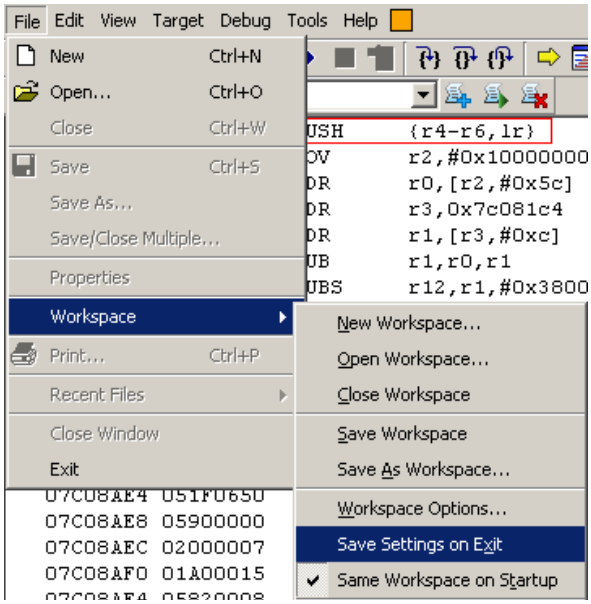
Workspaces

Workspaces are used to store personalized settings – for example the layout of individual windows within the main RVD window. You can dock/undock, resize and move these windows by dragging them. There are 'hot areas' to the left, right and bottom of the main RVD window. Dropping a window in one of these areas will cause the window to be docked.

- Select *View → Registers* to open a Registers window.
- Drag the *Registers* window to the right hand edge of the screen so that it becomes docked.



- Select *Target* → *Disconnect* to disconnect from the target.
- Select *File* → *Workspace* → *Save Workspace* to save the new layout.
- Select *File* → *Workspace* → *Save Settings on Exit* to remove the tick next to this option.



RVD will now remember the window layout that you have just created, and will start up without trying to connect to your target.

See the FAQ '*How can I use workspaces to control RVD's GUI?*' on the ARM web site for more information on using Workspaces.

Semihosting

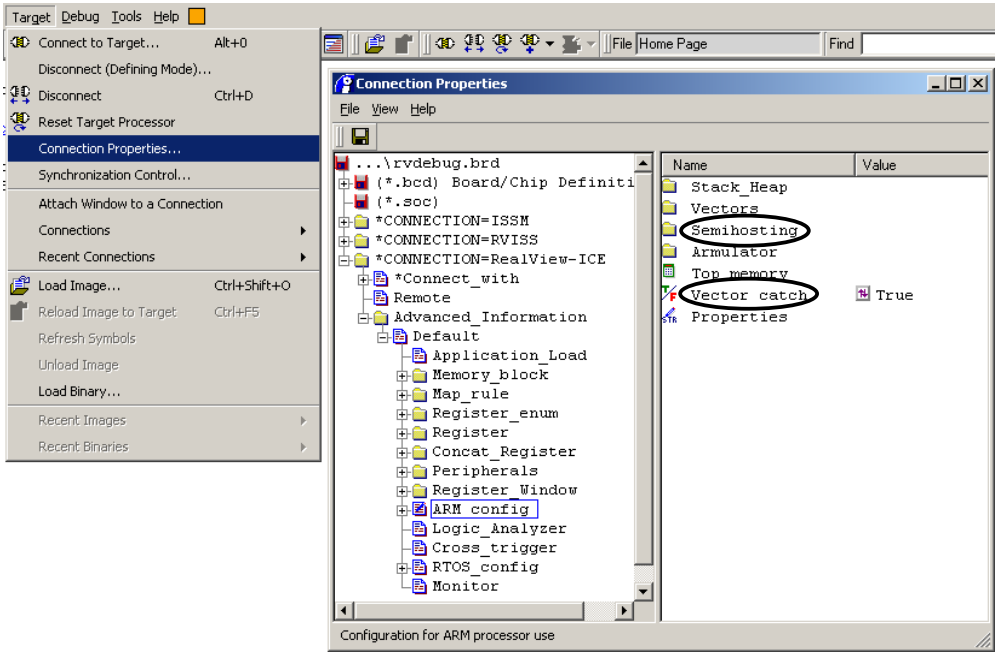
Semihosting is a mechanism that captures I/O requests made by code running on the target (typically library code), and communicates these to the host system for handling. For example, application *printf*s will by default appear within the debugger console window.

See Section 13.9 of the *RealView Debugger v3.1 User Guide* ('*Viewing semihosting controls for RealView ICE JTAG* connections') for more information on setting Semihosting options.

Vector catch is a mechanism that is used for catching exceptions that occur on the core. It is implemented using dedicated logic, or instruction breakpoints if the core that you are using does not implement this logic. This feature is particularly useful when debugging code for which you have not yet written exception handlers.

Semihosting and vector catch are controlled by RVD connection properties, and are enabled by default.

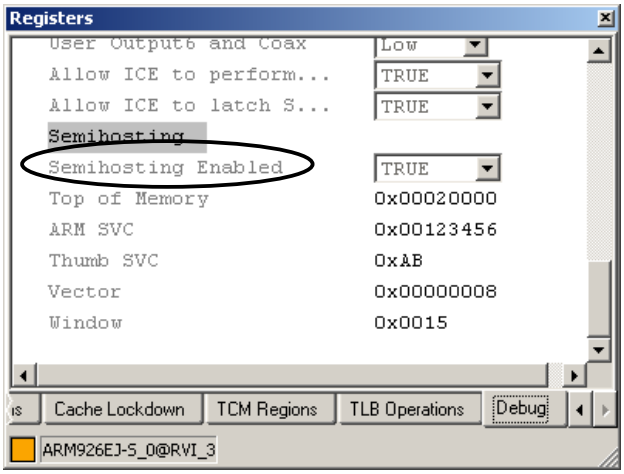
- Select *Target* → *Connection Properties* from the RVDmenu.
- Browse to *Advanced_information\Default\ARM_Config*
- Ensure that *Vector catch* is set to *True*
- In the *Semihosting* folder, ensure that *Enabled* is set to *True*
- Select *File* → *Save and Close* to save any changes.



Changing settings in *Connection Properties* causes those settings to be applied to all subsequent connections of that type.

Changes made in *Connection Properties* do not affect any currently active connections. You must disconnect from your target before making changes and then reconnect afterwards.

Alternatively, you can enable/disable Semihosting in the *Debug* tab of the *Registers* window. Settings made in this window are temporary and are lost when you disconnect from the target.

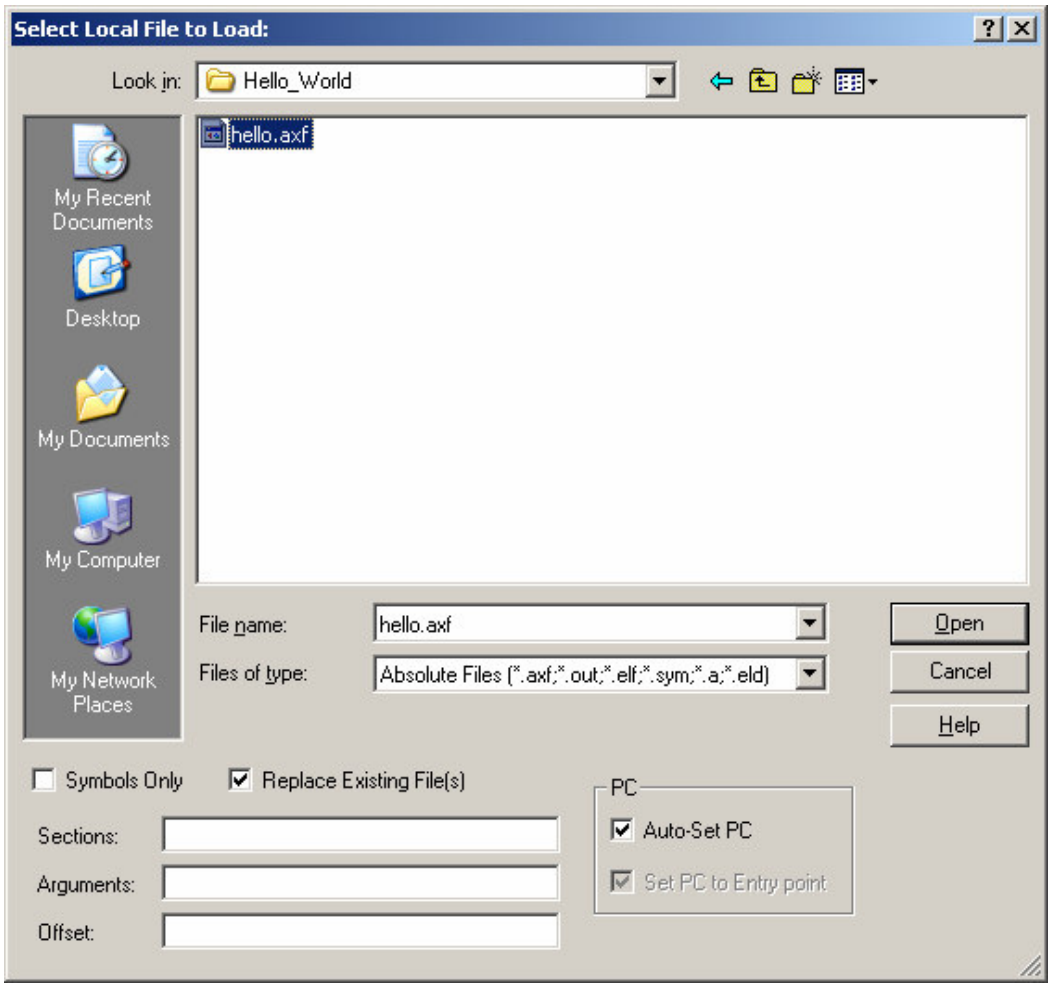

Registers
User Output6 and Coax
Low
Allow ICE to perform...
TRUE
Allow ICE to latch S...
TRUE
Semihosting
Semihosting Enabled
TRUE
Top of Memory
0x00020000
ARM SVC
0x00123456
Thumb SVC
0xAB
Vector
0x00000008
Window
0x0015
Cache Lockdown
TCM Regions
TLB Operations
Debug
ARM926EJ-S_0@RVI_3

Section 3: Using RealView ICE with RVD

This section provides an introduction to using the RealView ICE unit with RealView Debugger to debug an application running on your target hardware.

3.1 – Simple Hello World Project (RVD Basics)

Select *Target→Load Image…* from the RVD main menu. Browse to **hello.axf** in the **c:\rvds31_tutorial\PB926\Hello_World** directory and click *Open*.

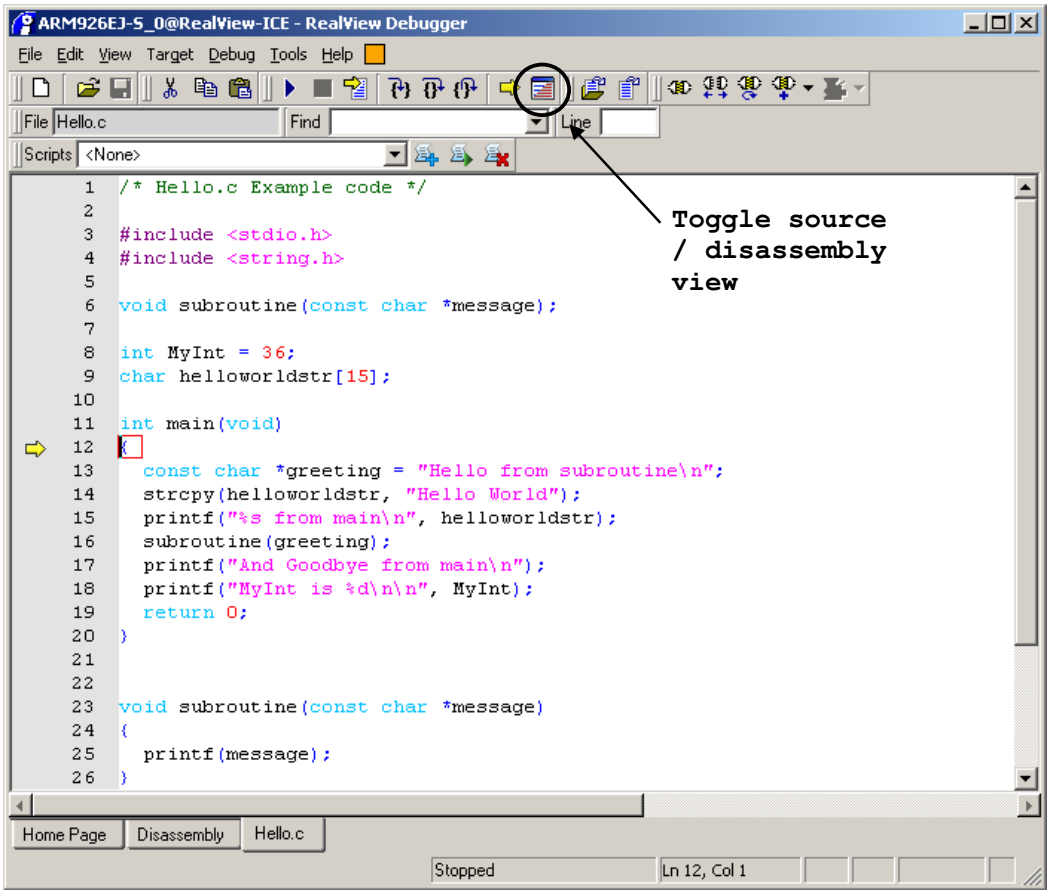


The main window now shows that the image is loaded. The red box indicates the current execution position.

Code Window Tabs

When RealView Debugger first loads an image, the main view contains a *Disassembly* tab that displays disassembled code with interleaved C/C++ source lines. The current location of the PC is shown (right click → *Interleave Source* to toggle source interleaving).

If source code exists for the current location of the PC, you can toggle between disassembly and source code views using the toolbar button below.



By default, RealView Debugger will show the scope relevant to the current context of the PC.

Select *Debug* → *Run* from the menu *(F5)*.

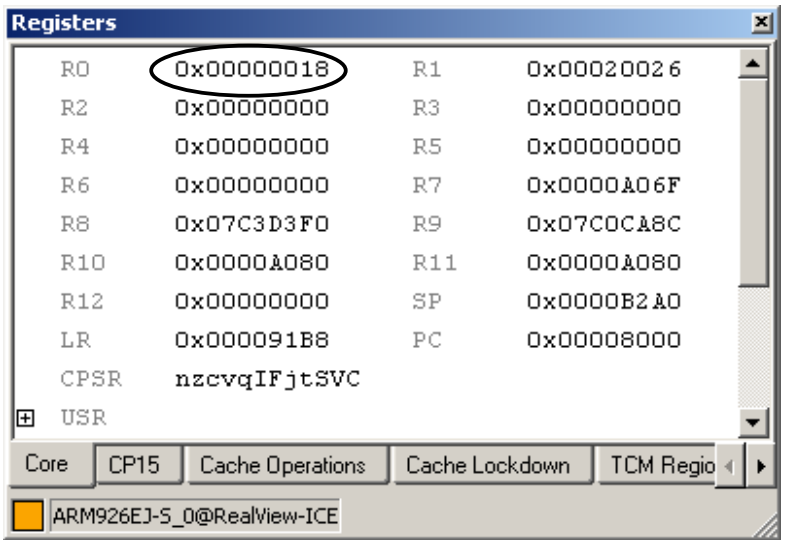
Execution begins. The *Output* pane at the bottom of the RVD window shows the *StdIO* tab which allows console I/O operations for the current image. The program prints some text to the output window and ends.

Select *Target* → *Reload Image to Target* from the menu.

RVD will reload the image ready for debugging. Again the current execution position is shown at `main()`.

Registers Window

The RVD *Registers* window allows you to view and modify the contents of ARM registers.



Of most interest are the tabs for the *Core* registers (r0 – r15 and the CPSR), the *CP15* registers (e.g. enabling/disabling caches, mmu, branch prediction etc), and the *Debug* registers (e.g. setting Top of Memory and enabling/disabling Semihosting).

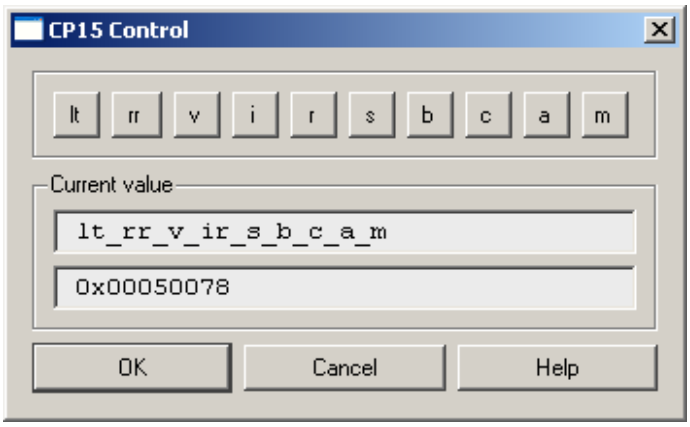
The displays of registers r0-r15 can be used to modify the value of live variables (variables that are currently in use) when the core is stopped, for example when you are stepping through code or hit a breakpoint.

You can switch between different value formats by right-clicking on a register and selecting *Format* from the context menu. The most common choices are *Hex* and *Decimal*.

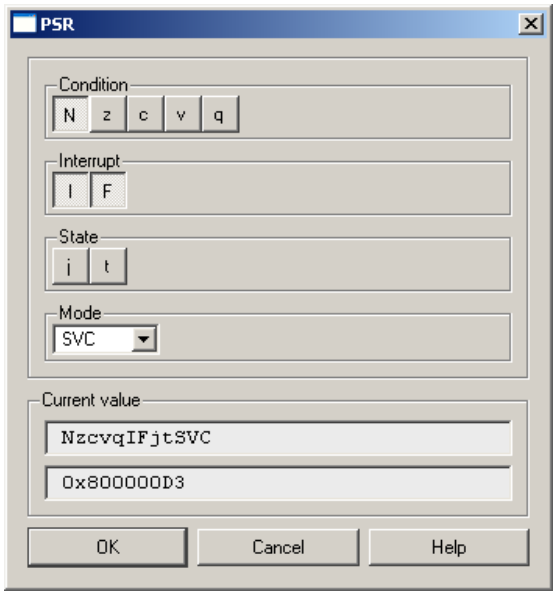
Right-click on *R0* and select *Format* → *Decimal* to change the format of *R0* to decimal. Select the value next to *R0* and enter a new value (e.g. **526**).

Some of the registers are enumerated – for example the CP15 *Control* register.

In the Registers window *CP15* tab, double click on *Control*. Notice that by clicking on m/M you can enable/disable the MMU. Hover the mouse over each button to see what can be enabled/disabled.



Another useful register that is enumerated is the *CPSR* (Current Program Status Register) in the *Core* tab. This register manages enabling/disabling of interrupts, core state (ARM or Thumb) and the current system mode.



Breakpoints

Breakpoints tell the debugger to stop the target when a particular event occurs. For example, you can tell the debugger to stop execution on the target before a particular instruction or C source code line is executed so that you can inspect and modify the current state of the core. This is implemented either by temporarily replacing the relevant instruction with a breakpoint instruction (a software breakpoint), or using watchpoint units within the core that monitor the address and data buses (a hardware breakpoint).

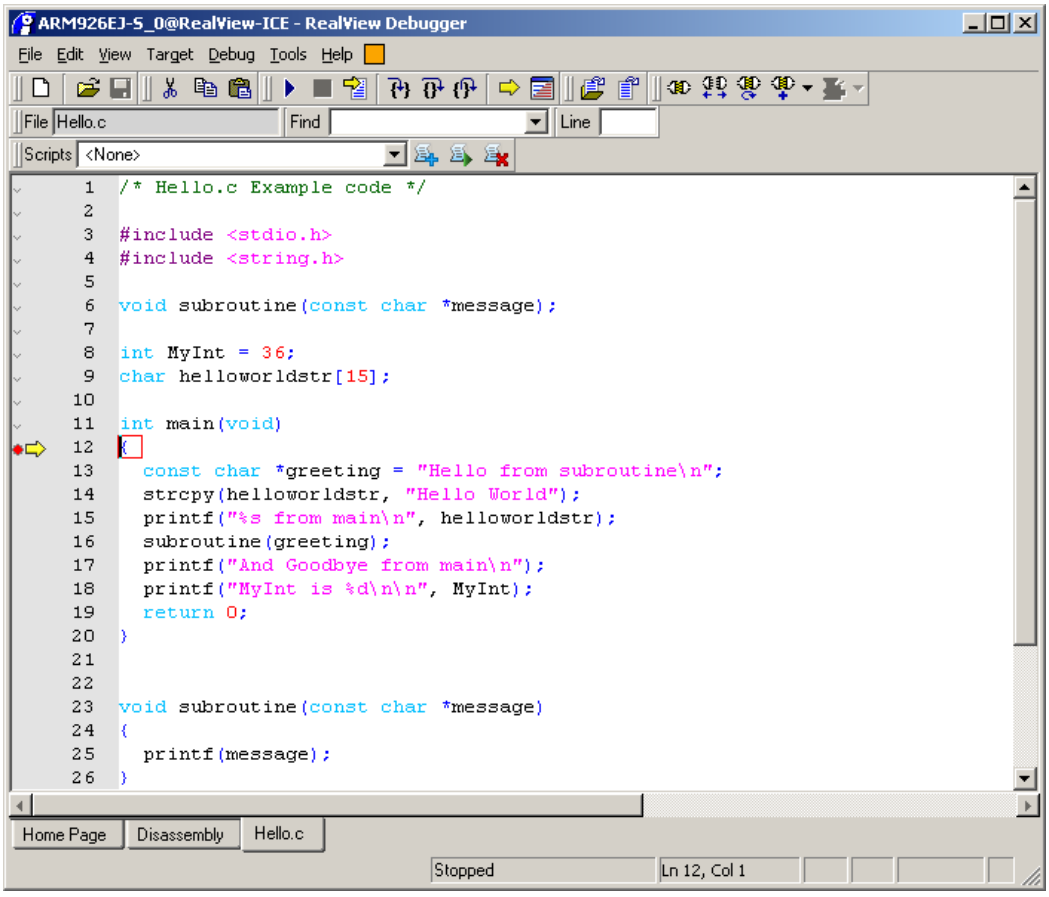
Software breakpoints can only be used on instructions that are in RAM, as they involve the temporary substitution of a breakpoint instruction for the original instruction. Hardware breakpoints can be used on any area of memory, including flash or ROM.

See the FAQ '*What is the difference between HW and SW breakpoints?*' on the ARM web site for more information on hardware and software breakpoints.

Hardware and software breakpoints can be conditional - the debugger will only stop the core under certain conditions. For example, a breakpoint could be set so that the debugger stops the core before a particular line of code is executed. A condition could be added to this breakpoint such that the breakpoint is only hit when a particular value is stored at a particular location in memory.

The simplest way to set a breakpoint is to double click in the grey margin to the left of a line in the *Disassembly* tab or within a source file. This will set an instruction breakpoint so that when the core is about to execute that assembly instruction or C source line the debugger will halt execution. RVD will use a software breakpoint where possible, otherwise a hardware breakpoint.

Set a breakpoint on **`main()`** by double clicking in the margin to the left hand side of the *Code* window. From the RVD main menu select *View→ Break/Tracepoints* to view information about the breakpoints you have created.

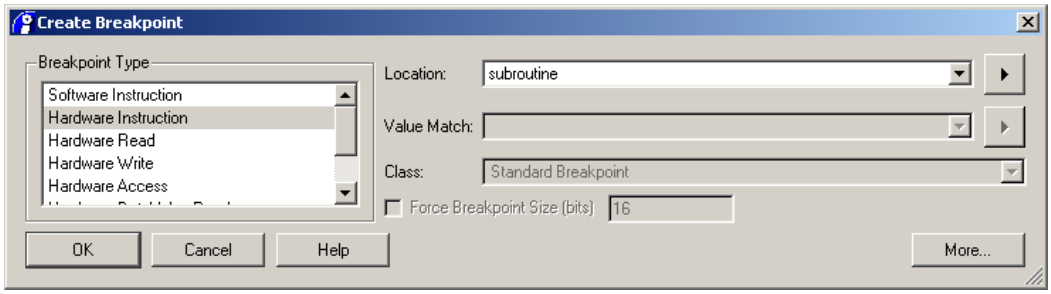


Select *Debug→Run* from the menu (F5).

Execution now halts at **`main()`** after initialisation of the C library.

If the location where you want to set a breakpoint is in non-volatile memory (e.g. flash), you can explicitly set a hardware breakpoint.

Set a new hardware breakpoint on **`subroutine()`** by selecting *Debug→Breakpoints→Create Breakpoint…* from the main menu.
- Select *Hardware Intsruction*
- Type **`subroutine`** for the location and click *OK*.



Select *Debug→Run* from the menu *(F5)*.

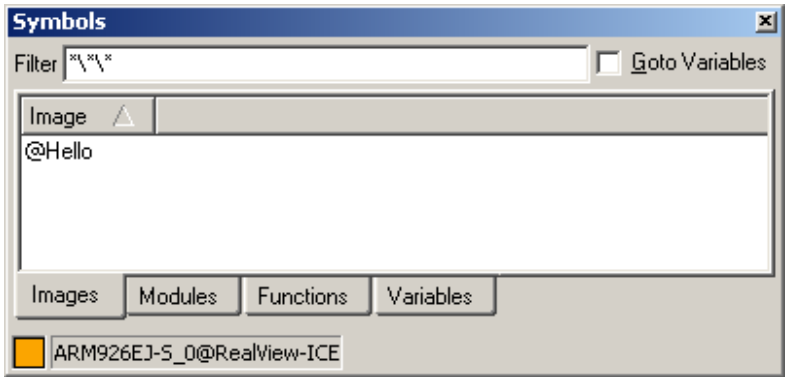
Execution will halt at **`subroutine()`**.

Tracepoints

Tracepoints are similar to breakpoints but instead of halting program execution will start/stop trace capture. Note that you need to have the Trace Analyzer connected to be able to create a tracepoint. Using the Trace Analyzer and the setting of tracepoints are discussed in more detail in the *Trace* section of this tutorial.

Symbols Window

The RVD *Symbols* window allows you to browse through symbols contained in the images currently loaded into the debugger.



From the *Symbols* window you can conveniently locate a function, run to a function, add a variable to the watch window or set a breakpoint.

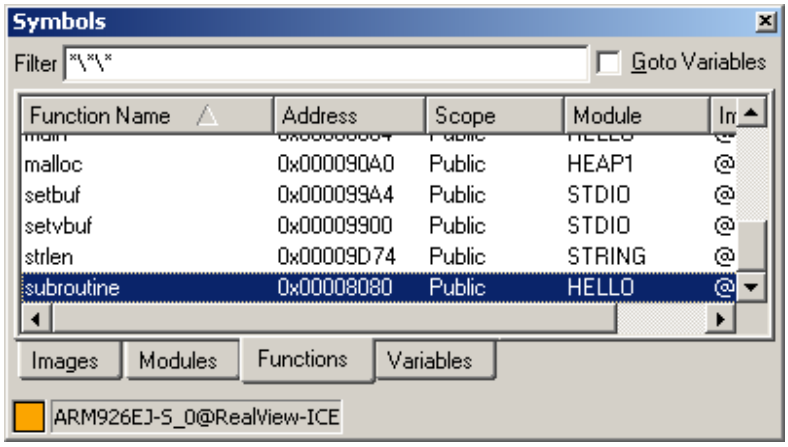
If the *Symbols* window is not already open, select *View* → *Symbols* from the RVD menu to display the *Symbols* window.

The format of the *Filter* at the top of the window is *Image\Module\Function*. You can manually edit the filter, or double click on an item in the *Images* or *Modules* tab to filter by that selection. Note that * is a wildcard to avoid filtering.

- The *Images* tab displays currently loaded images;
- The *Modules* tab displays modules contained in the currently filtered images;
- The *Functions* tab displays functions contained in the currently filtered modules;
- The *Variables* tab displays variables contained in the currently filtered functions.

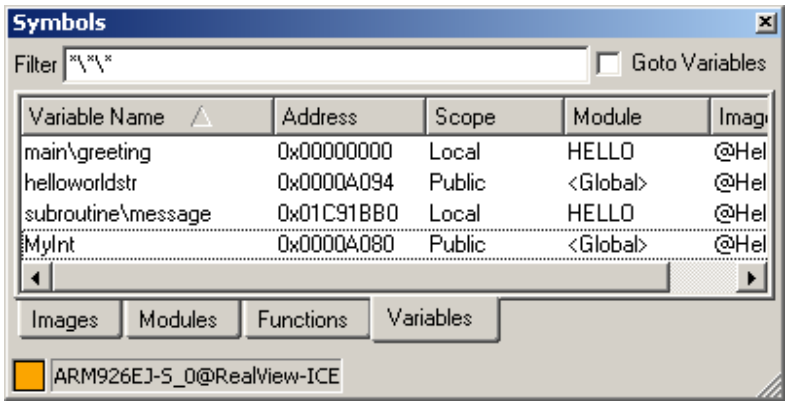
Double clicking on a function name jumps to that function in the *Disassembly* tab.

Select the *Functions* tab in the *Symbols* window and double-click on `subroutine()`. Notice that the main view changes to the *Disassembly* tab and shows the assembly code for the selected function.

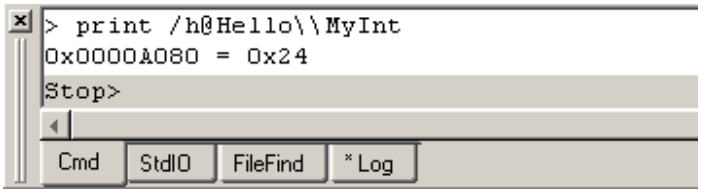


Double clicking on a variable name displays the address and value of that variable in the *Cmd* tab. By default, the *Variables* tab does not display locals.

- Select the *Variables* tab in the *Symbols* window
- Right-click in the white space in the window and select *Show Locals*



Double-click on **`MyInt`** to display its address and value in the *Cmd* tab.

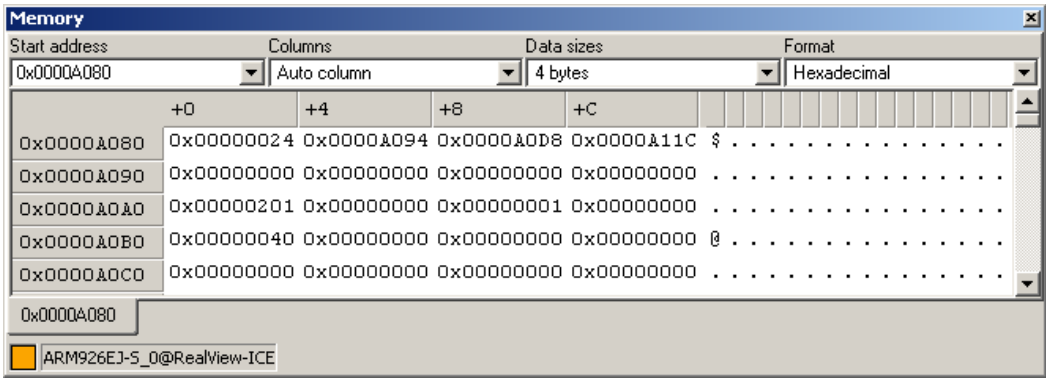


See the FAQ '*How do I access the symbols in my image using RVD?*' on the ARM web site for more information on the *Symbols* window and its tabs.

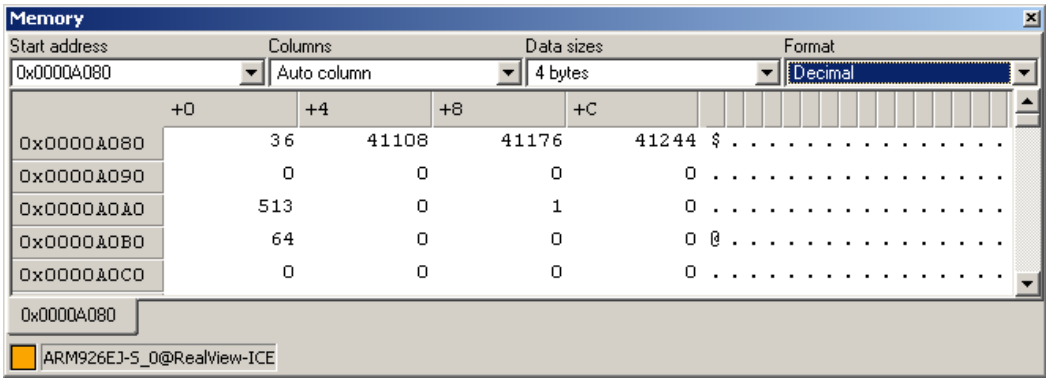
Memory Window

The *Memory* window allows you to view the data in a particular area of memory, in a configurable layout and format.

- Find the address of **`MyInt`** from the *Address* column of the *Symbols* window *Variables* tab (0xA080 in the screenshots above in the *Symbols Window* section)
- Enter the address into the *Start address* field in the *Memory* window and press *enter*.



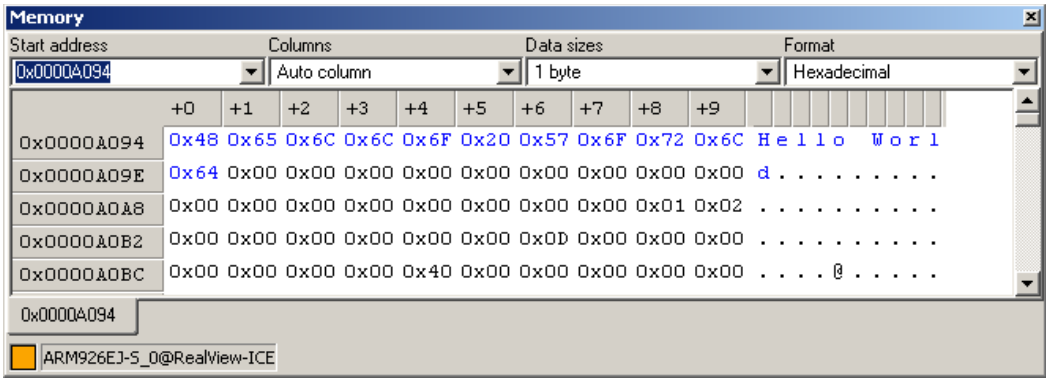
- Change the *Format* to *Decimal*



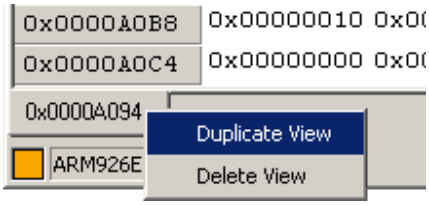
- Step forward (press F11) until the yellow arrow points at line **15**, (**`printf("%s from main\n", helloworldstr);`**).

- Find the address of **`helloworldstr`** from the *Address* column of the *Symbols* window *Variables* tab.
- Enter the address into the *Start address* field in the memory window and press *enter*.

- Change the *Format* to *Hexadecimal* and *Data sizes* to *1 byte.*
- Notice that the text that makes up the string `helloworldstr` is displayed on the right-hand side of the *Memory* window.



You can add new tabs to the *Memory* window by right-clicking on an existing tab and selecting *Duplicate View*. Tabs can be closed by selecting *Delete View*.

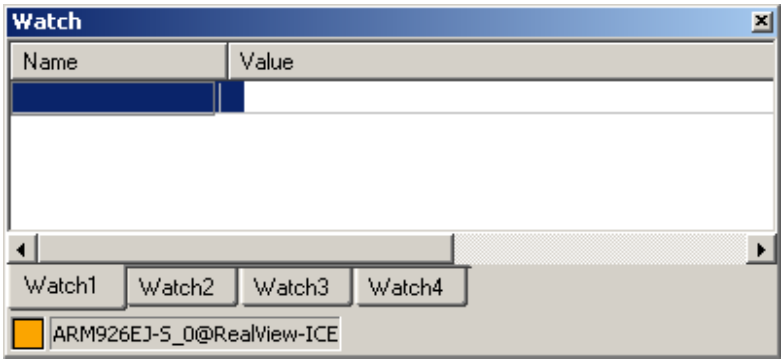


Select *Debug* → *Set PC to Entry Point* from the main menu, then press F5 to run to the breakpoint set earlier on `main()`.

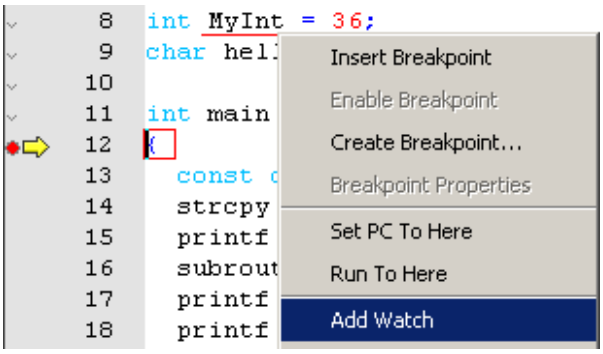
Watch Window

The *Watch* window allows you to keep track of specific variables.

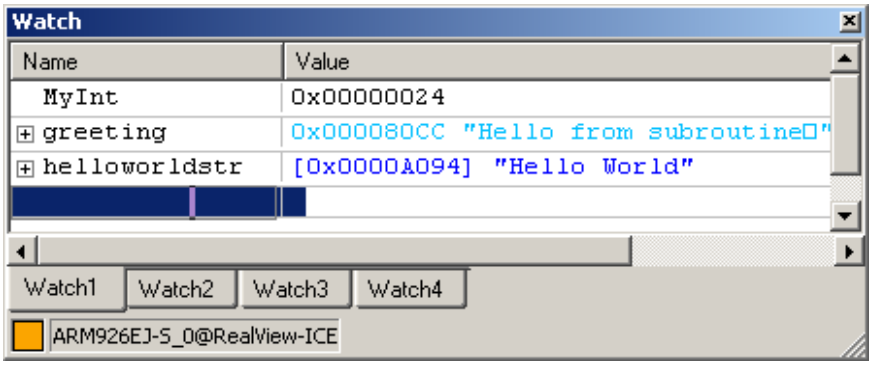
If the *Watch* window is not already open, select *View* → *Watch* from the RVD menu to display the *Watch* window.



Right-click on **`MyInt`** (on line **8** of **Hello.c**) in the main view and select *Add Watch*. Repeat for **`greeting`** and **`helloworldstr`** (lines **13** and **9** respectively).



- Press F11 (Step Into) 3 times so that the yellow arrow points to line **15** (**`printf("%s from main\n", helloworldstr);`**).
- Observe that as **`greeting`** and **`helloworldstr`** are initialised in the code, their values update in the *Watch* window.

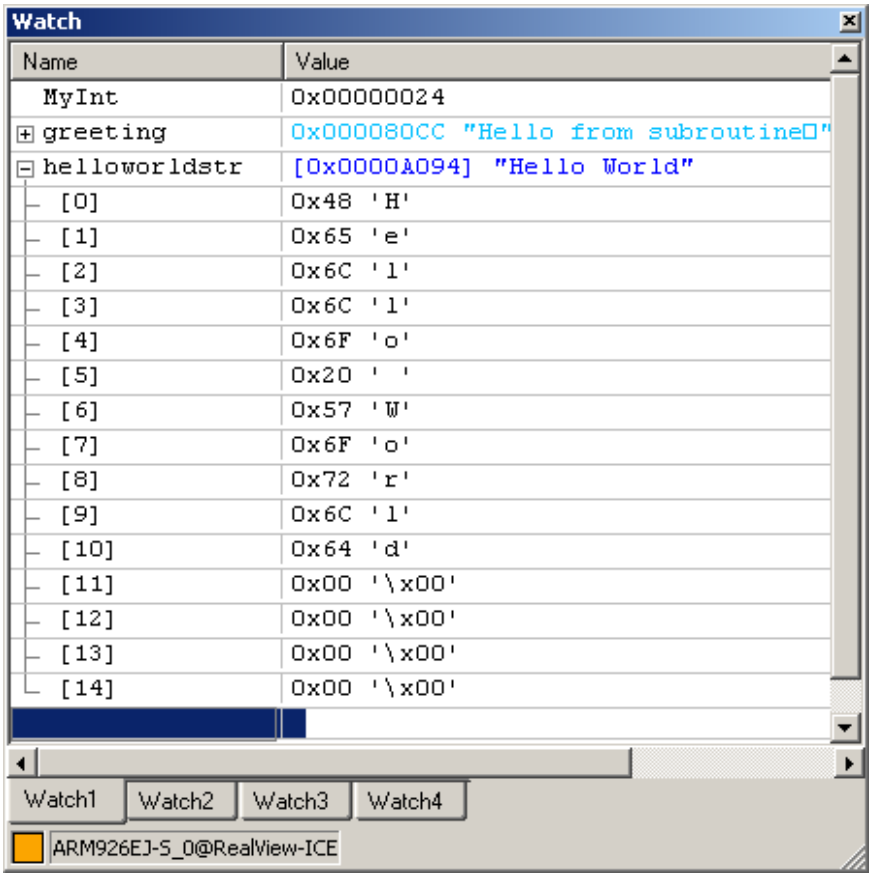


As well as adding a variable to the *Watch* window from the variable's context menu, you can manually enter a variable name into the *Watch* window *Name* column. Note that the core must be stopped for the values displayed to be updated.

When watching a pointer (including a string/array), a + symbol will be displayed to the left of the pointer name. Click on this to expand the display to show the contents of an array, or, for a pointer, to show the value pointed to by that pointer.

Click on the + next to the pointer **`greeting`**. Observe that the value pointed to by **`greeting`** is shown (the first character of the string, *H*).

Click on the + next to the array name `helloworldstr`. Observe that the characters that make up the array are shown. The text string ends with the first *NULL* character (0x0) at **`helloworldstr[11]`**.



The *Watch* window contains 4 tabs so that you can have a different set of variables that you want to keep track of depending on which part of your image you are currently working with. This is particularly useful when moving up / down the Call Stack (see *Call Stack* below).

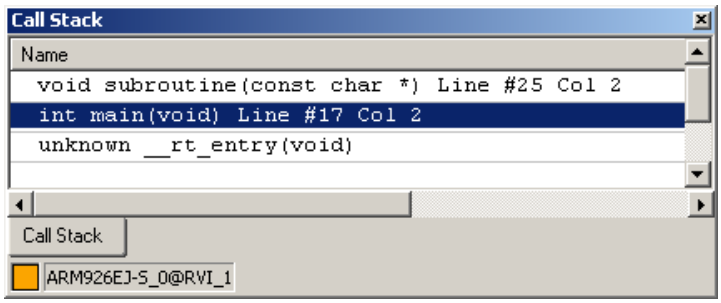
Call Stack

The execution scope or context determines the visibility of variables and functions. A variable or function is referred to as in scope if the name can be accessed at the current point of execution. The scope of a variable or function can be:

- the current source file, for global variables and functions;
- the current function, for local variables

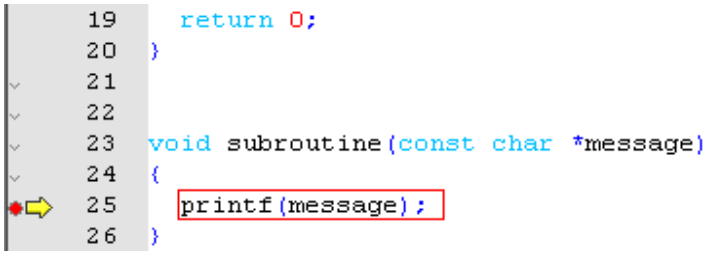
When you load an image, scope is initially set to the value of the PC, which is usually the entry point of the image. As you step through the image and move into child functions, the scope updates to continue to show the current context.

- Select *View* → *Call Stack* from the RVD main menu.



The Call Stack window shows the hierarchical flow of a program and enables you to trace back to the program's status at an earlier point. By moving up to a previous entry in the call stack, you can change the scope to be at the point where the child function will return to. This works by retrieving from the stack local variables that were active immediately before the function call took place.

Double-click in the margin to the left of the **`subroutine()`** function definition (line **23**) to set a breakpoint. Press F5 to run to this point in the program.



- Double-click on the **`int main(void) Line #17 Col 2`** entry in the *Call Stack* window.
- Observe the output **`Scoped at level 1: (0x000080AC): HELLO\main Line 17:2`** in the *Cmd* tab.

A blue arrow and a blue box show the new scope:

```
10
11  int main(void)
12  {
13    const char *greeting = "Hello from subroutine\n";
14    strcpy(helloworldstr, "Hello World");
15    printf("%s from main\n", helloworldstr);
16    subroutine(greeting);
17    printf("And Goodbye from main\n");
18    printf("MyInt is %d\n\n", MyInt);
19    return 0;
20  }
```

- A blue arrow and blue box indicates that the current scope is different from the location of the PC.
- A yellow arrow and red box indicates that the current scope is the location of the PC.

You can also move up (to parent functions) and down (to child functions) the call stack by entering the commands *up* and *down* in the RVD Command Line (see *Command Line* below).

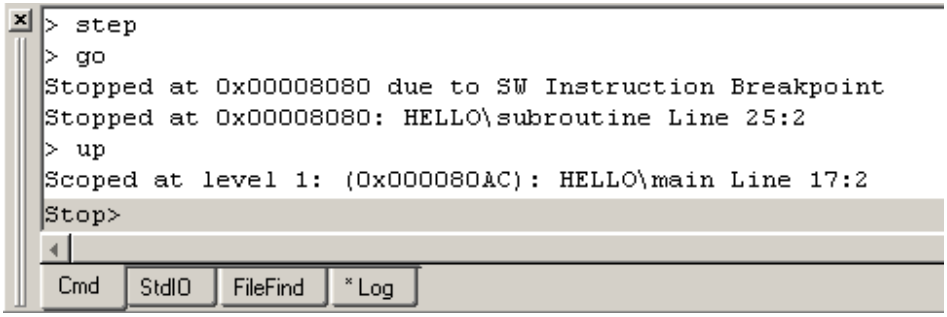
As you move up and down the call stack, observe that the variables displayed in the *Locals* window change (select *View* → *Locals* if this window is not already open).

Variables in the *Watch* window that are local to a particular function also become active / inactive as you move up / down the stack. You can make use of multiple *Watch* windows, or the multiple tabs within an individual *Watch* window to isolate variables that are relevant to a particular scope.

If the scope is at a location that corresponds to a source file then RVD automatically opens that source file if the *Home Page* tab or another source file currently has the focus.

Command Line

The *Cmd* tab in RVD displays the current status of the debugger. When you carry out a GUI action, you will usually see a command-line equivalent echoed to this tab. You can use these textual equivalents on the command-line (the grey bar at the bottom of the *Cmd* tab). This is particularly useful when creating a script (see below).



Use the *up* and *down* commands to move up and down the call stack (see *Call Stack* above).

Simple Scripting

You can gather together a sequence of command line instructions into a script. This is a plain text file that can be called automatically when you connect to a target, or on demand. An example of automatically calling a script at connect time is given in the *Configuring Trace* section of this tutorial.

To create and run a simple script:

- Carry out the GUI actions that you want to script
- After each GUI action, copy and paste the command echoed to the *Cmd* tab into a text file
- Save the text file (e.g. *MyScript.inc*)
- Click on the *Add Script* toolbar button and browse to the script that you created
- Click on the *Run Script* toolbar button to run the script

Scripts are most commonly used to perform target configuration on connection to a target, or to perform the connection itself followed by some subsequent steps. The example below shows the generation of a simple connection script.

- Disconnect from your target (*Target → Disconnect*)
- Right-click in the *Cmd* tab and select *Clear* to clear the existing output.
- Connect to your target by double-clicking on the target in the *Target → Connect to Target…* dialog
- Load the previous **Hello.axf** image via the *Target → Load Image…* dialog.

- Set a breakpoint on `subroutine()` by double-clicking in the margin to the left of the first line of the function definition (line **23**)

- Press F5 to run to the breakpoint
- Observe that the above steps caused the following commands to appear in the *Cmd* tab, along with some additional information:

```
connect "@ARM926EJ-S_0@RealView-ICE"
load/r C:\rvds31_tutorial\PB926\Hello_World\Hello.axf
binstr \HELLO\#25:0
go
```

- Paste these commands into a plain-text editor (not including the initial **>** character) and save the file as **SimpleScript.inc** at **c:\rvds31_tutorial\PB926**
- Disconnect from your target (*Target* → *Disconnect*)
- Add the script that you just created using the *Add Script* toolbar button.
- Run the script using the *Run Script* toolbar button.

See the FAQ '*RVD Scripting & Automation*' on the ARM web site for more information on scripting with RVD, including some example scripts.

Section 4: Using RealView Trace with RVD

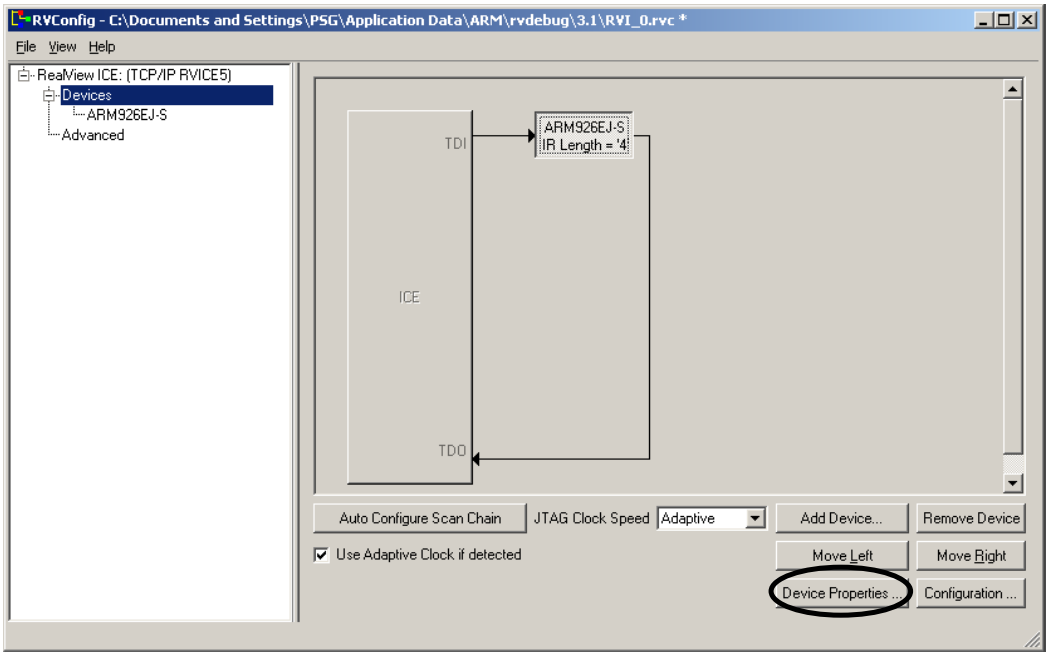
This section provides an introduction to using the RealView Trace unit with RealView Debugger to perform trace capture.

Application Note 168 'Tracing with RVD' provides a more comprehensive walkthrough guide to tracing your target.

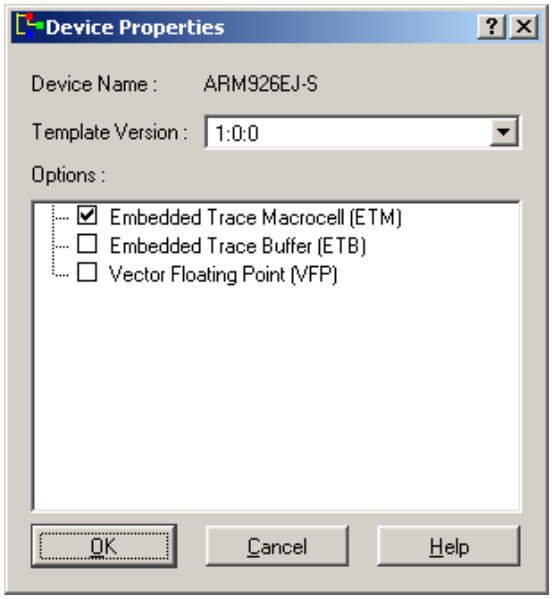
4.1 – Configuring Trace

When you have auto-configured (or manually configured) your target in RVD, you can configure whether or not your target has an ETM (Embedded Trace Macrocell) or an ETB (Embedded Trace Buffer). The PB926EJ-S contains an ETM, but no ETB.

Click on *Device Properties…* in *RVConfig*.



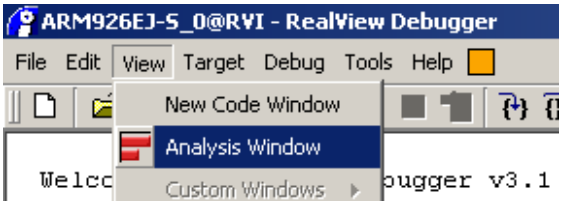
Ensure that *ETM* is selected, and *ETB* is deselected. Click *OK*.

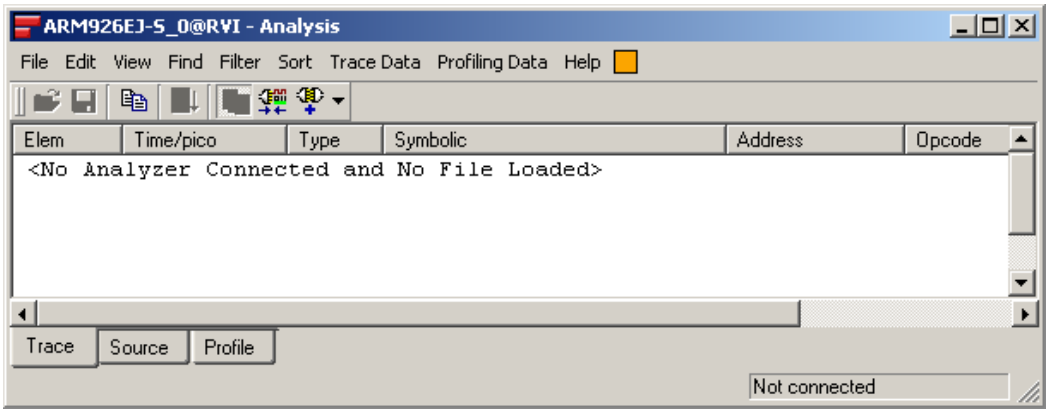


If your target contains an ETB (for example, the CM1136JF-S), then you can also select *ETB* to use the on-chip buffer rather than the external RealView Trace unit.

Connect to your target, then open the RVD *Analysis* window.

Select *View* →*Analysis Window*

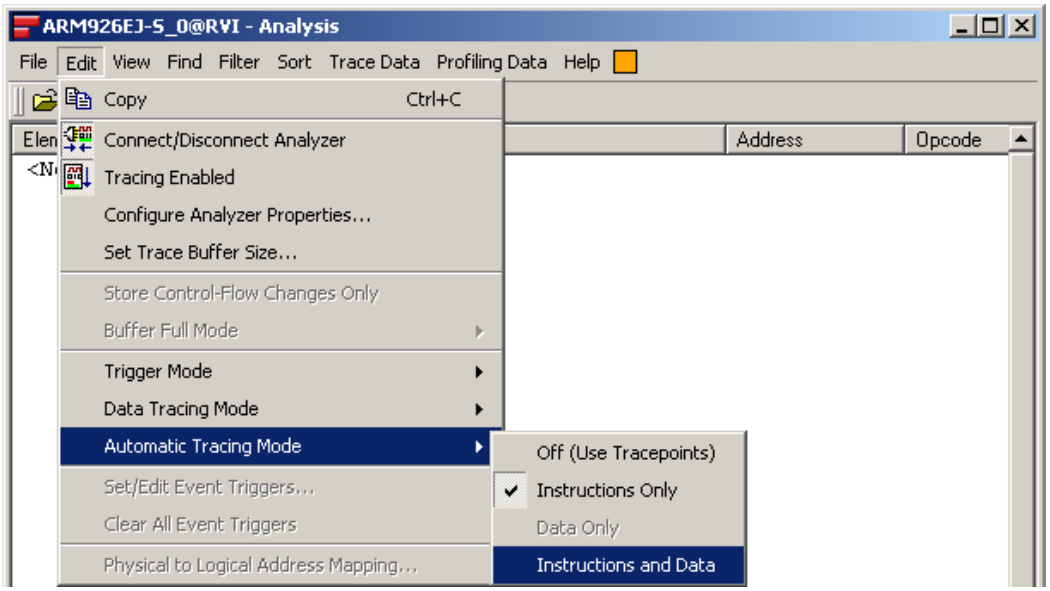




From the Analysis window you can configure trace settings and view collected trace data.

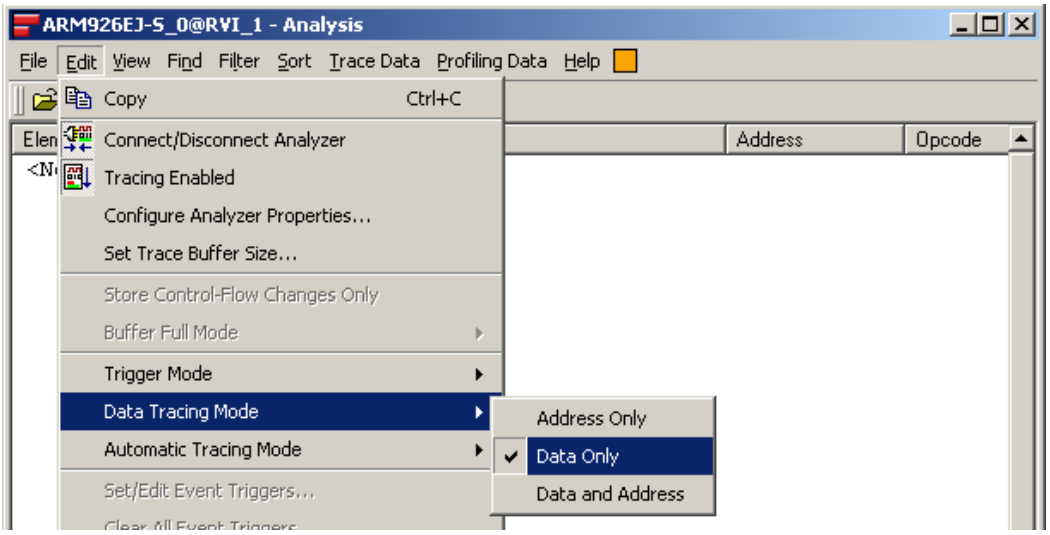
Connect the analyzer to your target by selecting *Edit → Connect / Disconnect Analyzer*

Select *Edit → Automatic Tracing Mode → Instructions and Data* from the menu in the *Analysis* window.



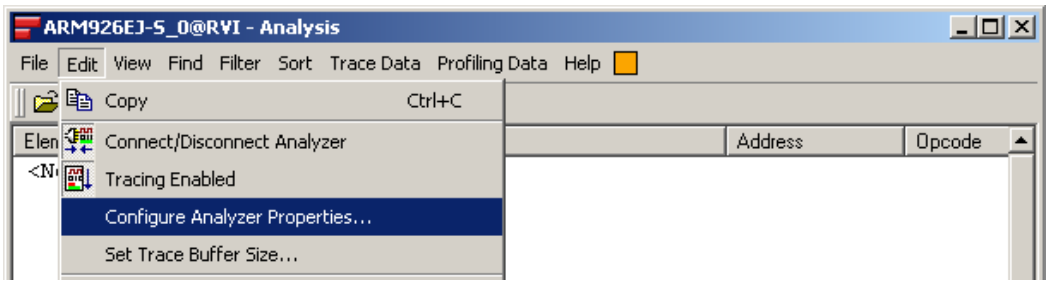
The debugger is now configured to automatically capture trace for both instructions & data.

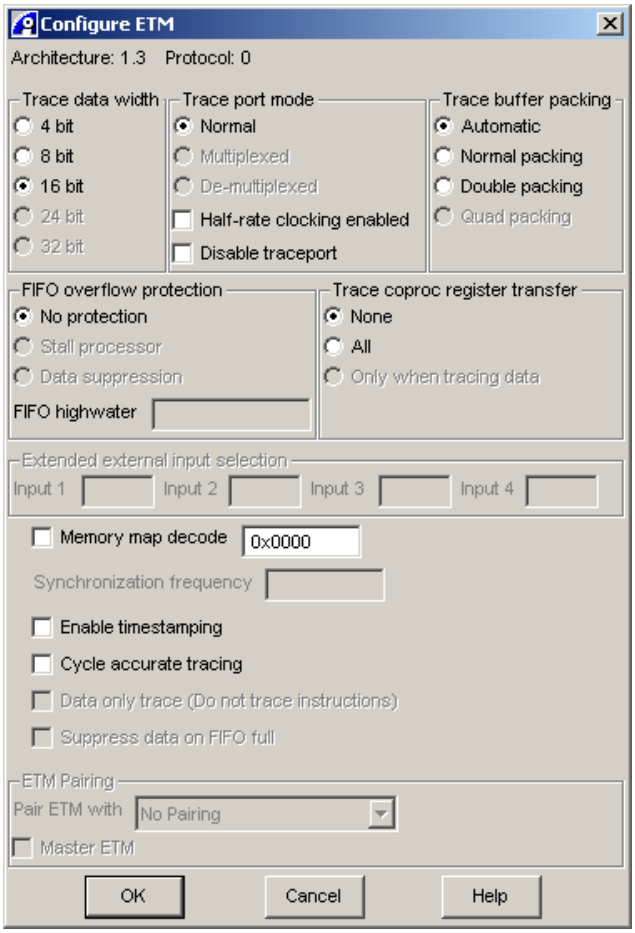
Select *Edit → Data Tracing Mode → Data Only* from the menu in the *Analysis* window.



The debugger is now configured to capture both data values and addresses for data trace capture.

Trace settings can be configured via the *Configure ETM* dialog, by selecting *Edit* → *Configure Analyzer Properties…* from the Analysis window menu. For this tutorial the default settings do not need to be changed.

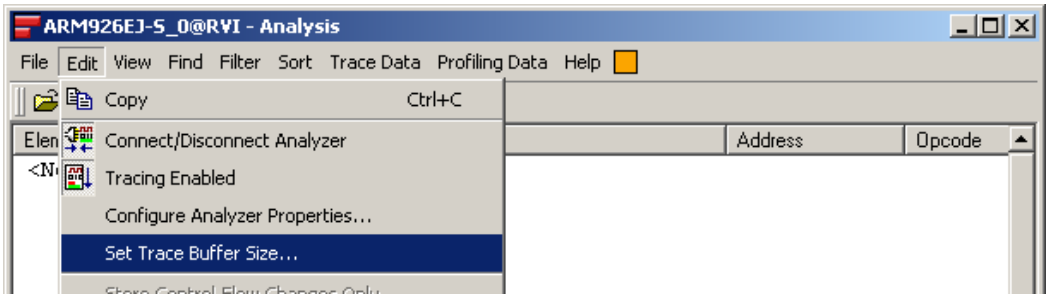


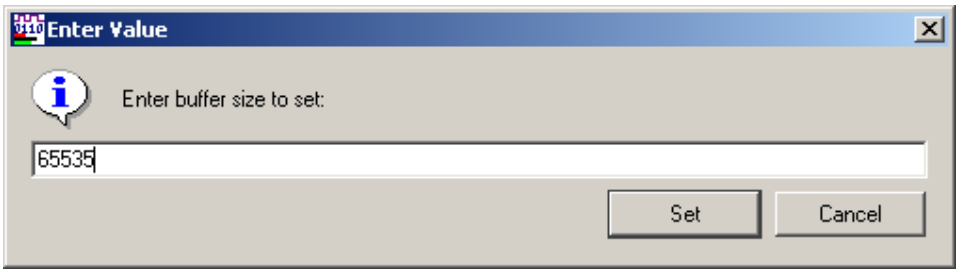


See Section 4.3 of the *RealView Debugger v3.1 Trace User Guide* (‘*Configuring the ETM parameters*’) for more information on ETM configuration options.

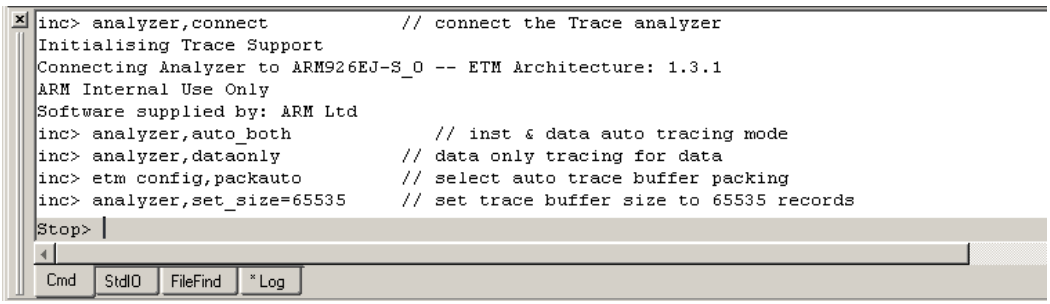
A reduced trace buffer size will reduce the time taken for the debugger to retrieve the trace data from the trace unit, but will limit the amount of data that can be captured. If you are performing lots of single steps or will be stopping your target regularly, you may want to set a small buffer size – perhaps as small as 1024 records. If you need to capture trace data for the entire execution of a larger piece of code, you may want to use the maximum buffer size. For this example we will set the buffer size to 65535 records. The maximum available buffer size for your trace hardware will be set automatically when you first connect to your RealView Trace unit (at least 1 million records).

Select *Edit → Set Trace Buffer Size…* from the menu, and enter a buffer size of *65535* records. Click *Set*.





Trace analyzer connection and configuration can be carried out using a connection script that can be associated with a connection. When you perform the instructions above, notice that a command appears in the RVD *Cmd* tab for each step. You can copy these instructions into a text editor and save the resulting script as a text file.



Open Notepad and paste in the following commands:

```
analyzer,connect        // connect the Trace analyzer
```

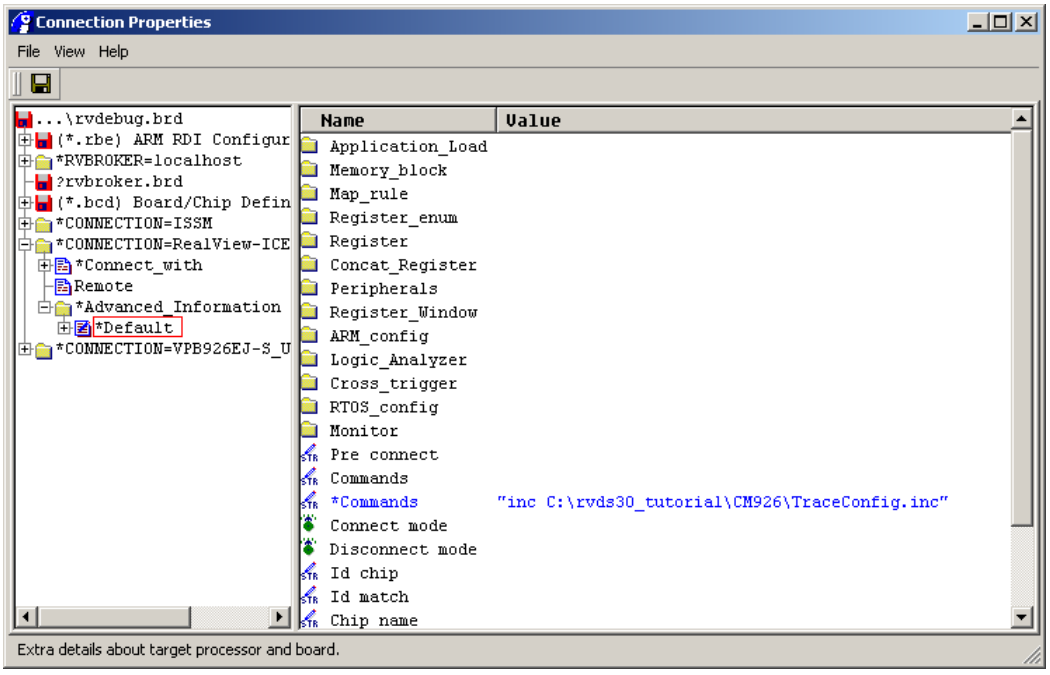
```
analyzer,auto_both       // inst & data auto tracing mode
analyzer,dataonly        // data only tracing for data
etm_config,packauto      // select auto trace buffer
                         //  packing
analyzer,set_size=65535 // set trace buffer size to 65535
                         //  records
```

Save the file as **TraceConfig.inc** at **c:\rvds31_tutorial\PB926**

If you are using a target other than the PB926EJ-S, you may need to modify the above commands.

You can tell RVD to run the script that you have created whenever you connect to a target via RVI.

- Go to *Target → Connection Properties*
- Click on *CONNECTION=RealView-ICE* in the left hand pane
- Browse to *Advanced_Information\Default*
- Right click on Command in the right-hand pane and select *Edit as Filename…*
- Browse to the script file that you just created and click Save
- Right click on the *Commands* entry you just created and select *Edit Value…*
- Insert the command **`inc`** in front of the path to the script file
- Select *File → Save and Close*



If you now disconnect from and reconnect to your target, you should find that the analyzer is automatically connected and configured for you.

When you no longer want to run this script at connect time you will need to remove the command you just created from *Connection Properties*. If you want to run the script a single time, you can type **`inc '<full_path_to_script>'`** on the RVD command line.

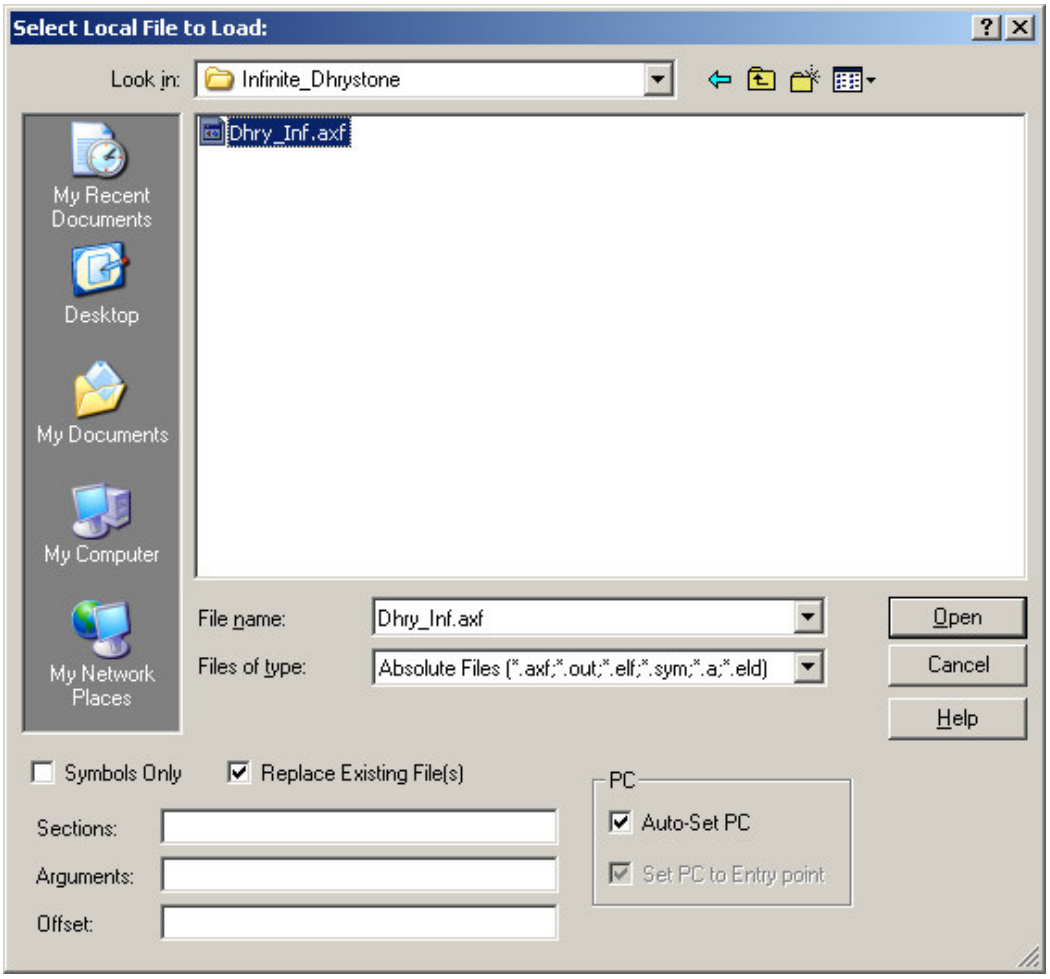
NB: If you are working with a Versatile PB926EJ-S or AB926EJ-S development board, you will need to reduce the core clock speed from 210MHz (default) to 140MHz in order to perform trace capture in both normal and half-rate tracing modes. This can be done by running the following RVD script commands before you begin your trace, either manually from the RVD command line, or by adding them to the beginning of your trace configuration script (**TraceConfig.inc**):

```
// unlock system registers
setmem /32 0x10000020 =0x0000A05F
// modify SYS_OSC0
setmem /32 0x1000000C =0x00002C6C
// lock system registers
setmem /32 0x10000020 =0x00000000
```

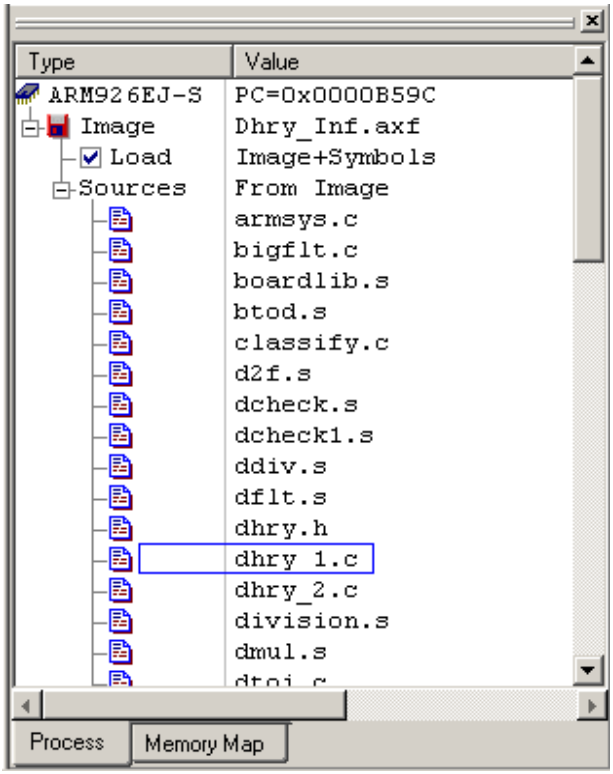
Your system is now configured to the point where you can perform *Auto Trace* using the ETM without setting any tracepoints. Trace capture will begin immediately when your program begins execution and will continue until the target is stopped (e.g. at a breakpoint). In the following examples tracepoints are used to specify a specific region where tracing should be carried out, reducing the total amount of trace data captured.

4.2 – Performing Simple Trace Capture

Select *Target→Load Image…* from the RVD main menu. Browse to **Dhry_Inf.axf** in the **c:\rvds31_tutorial\PB926\Infinite_Dhrystone** directory.



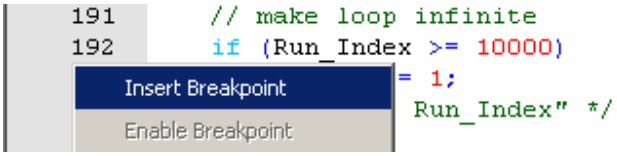
In the *Process* tab of the *Process Control* window (*View →Process Control)*, expand *Sources* and double click on **dhry_1.c** to open that source file.



In this example we will run an *Automatic Trace*. This uses no trace points for specifying the trace range and so causes trace to be captured for the entire program execution. It is possible to specify *Instruction Only* or *Instruction and Data* trace (see previous section). If tracing data in addition to instructions, you may need to consider the size of your trace buffer and the bandwidth of your trace port.

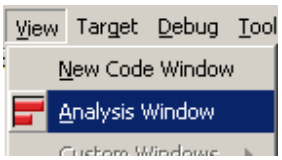
Population of the trace buffer will automatically wrap back to the beginning of the buffer when it is filled.

Right-click in the margin to the left of line **192** (`if (Run_Index >= 10000)`). Select *Insert Breakpoint* to set a breakpoint.

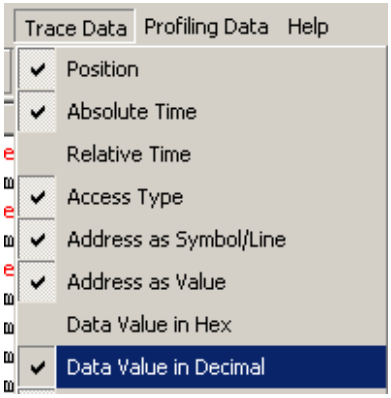


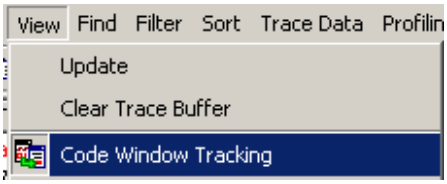
Start executing the image by selecting *Debug → Run (F5)* from the menu.

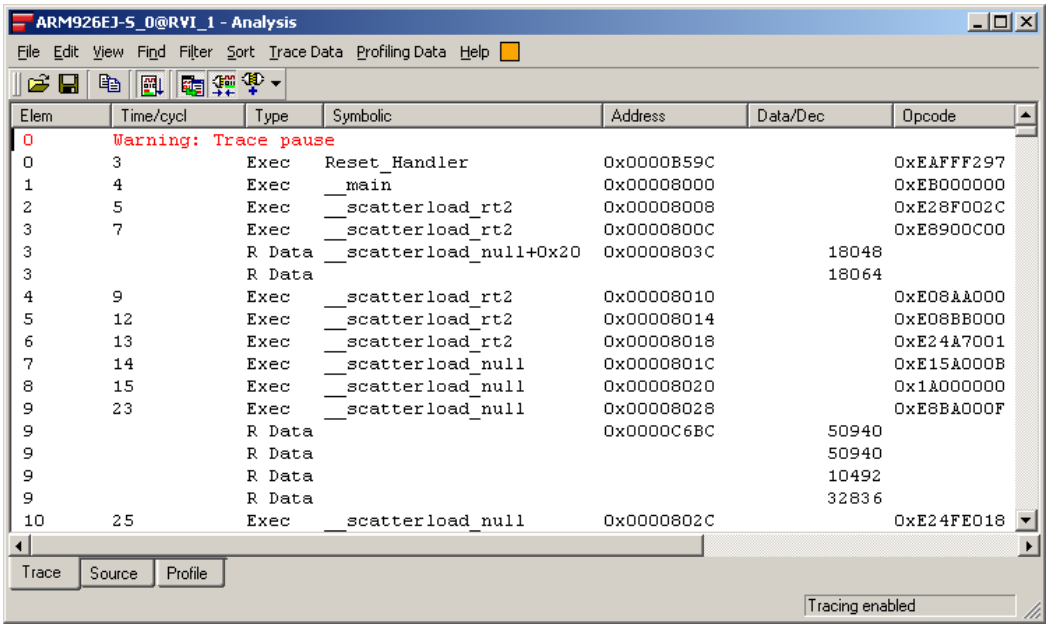
Execution halts on the breakpoint. Select *View → Analysis Window* from the RVD menu to view the captured trace.



Ensure that *Data Value in Decimal* is selected in the *Trace Data* menu, and *Code Window Tracking* is selected in the *View* menu.







The columns can be interpreted as follows:

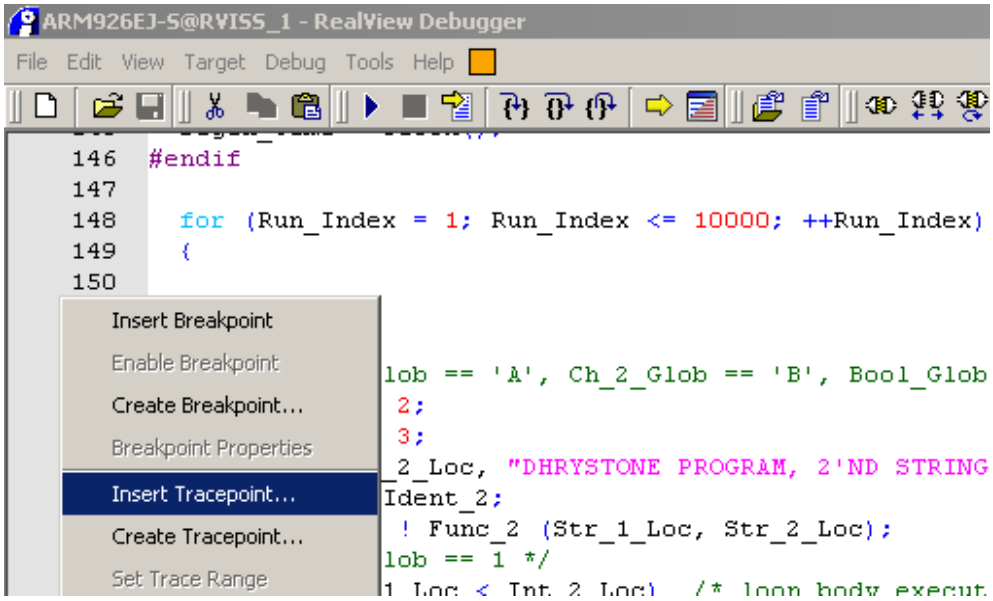
Elem	The element number in the current trace buffer. If a trace trigger has been set then element 0 will appear at the trigger point. Otherwise element 0 will appear as the last element.
Time/cycl	The relative cycle number on which an element began execution.
Type	**`Exec`**: An instruction that was executed **`NoExec`**: A conditional instruction that was not executed

	`R Data`: A data read **`W Data`**: A data write
Symbolic	Gives the module name and line number for the corresponding source code.
Address	Indicates the address an instruction was fetched from, or the address data was read from or written to.

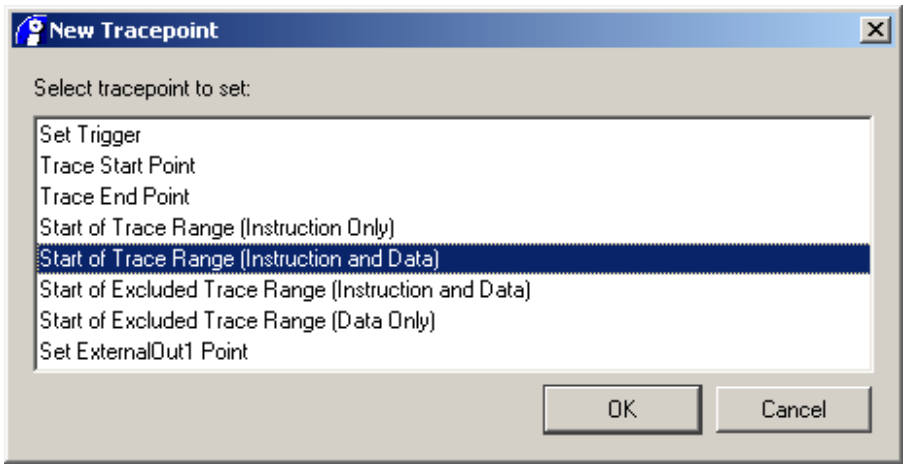
Scroll through the captured trace data to see the source code and disassembly synchronized with each line of trace data.

The same example can be run with a defined trace range where tracing will take place, by placing markers, or tracepoints, in the code. The advantage of using tracepoints is that you can capture trace data just for specific areas of interest in your code, and avoid having that data overwritten by unwanted data when the trace buffer wraps.

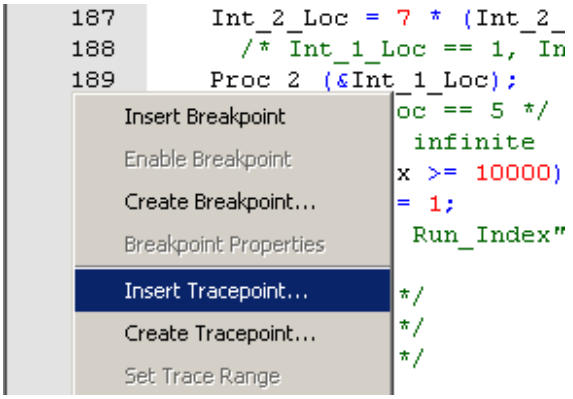
Right-click in the margin to the left of line **150** in **dhry1.c** (within the main **`for()`** loop in the body of **`main()`**). Select *Insert Tracepoint…*to display the *New Tracepoint* dialog.



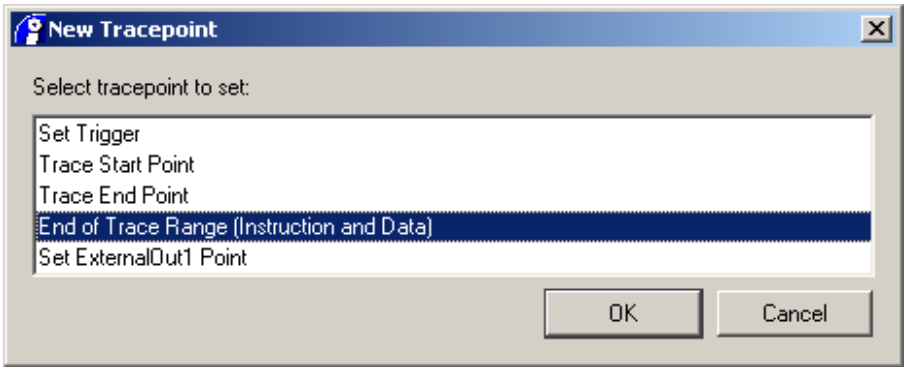
Select *Start of Trace Range (Instruction and Data)* from the list and click *OK* to set the tracepoint.



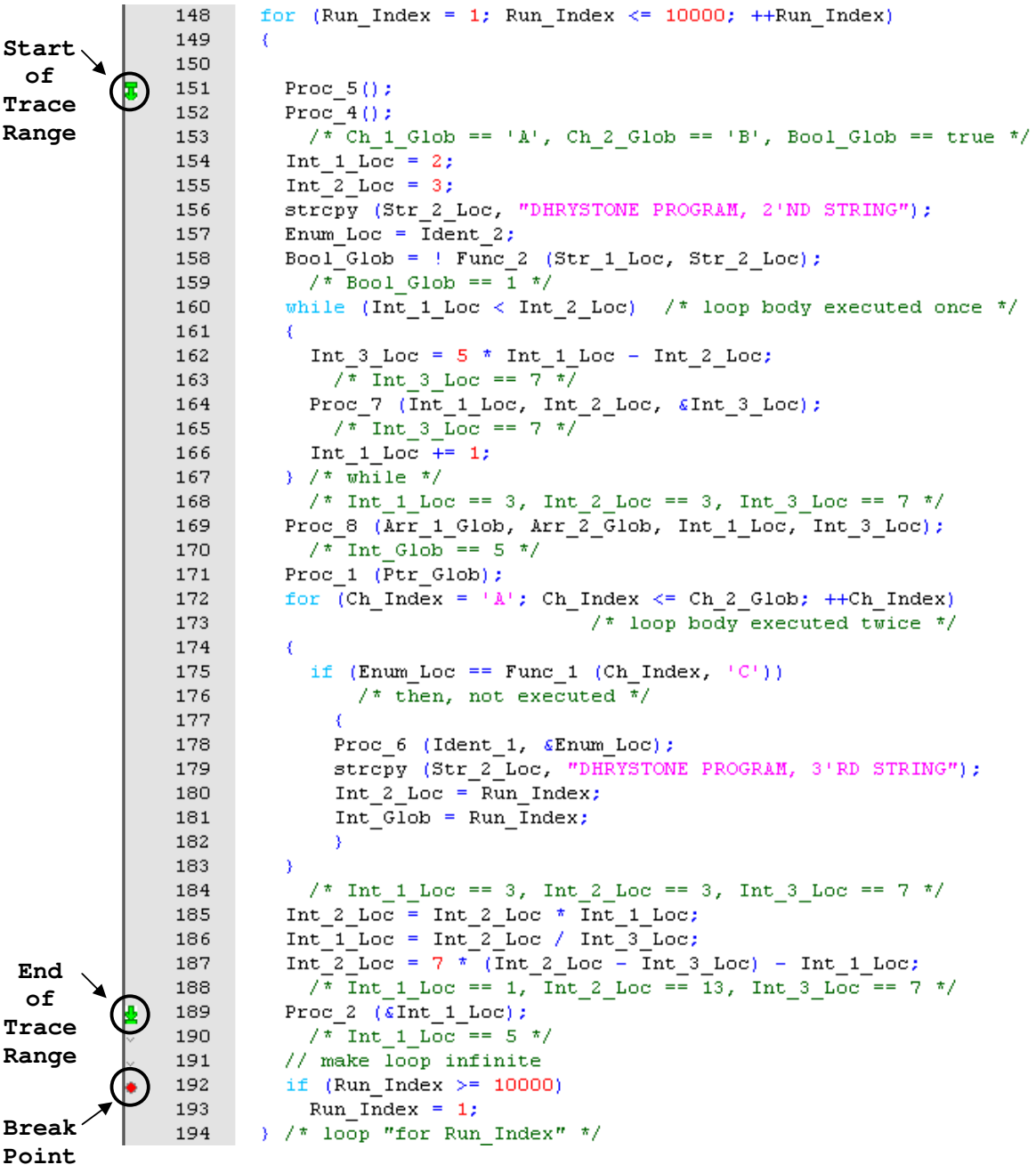
Right-click in the margin to the left of line **189** (**`Proc_2 (&Int_1_Loc);`**). Select *Insert Tracepoint…*to display the Tracepoint *List Selection* dialog again.



Select *End of Trace Range (Instruction and Data)* and click *OK* to set the trace stop point.



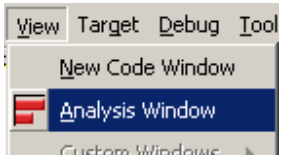
The source code display should now show the recently set trace and break points:

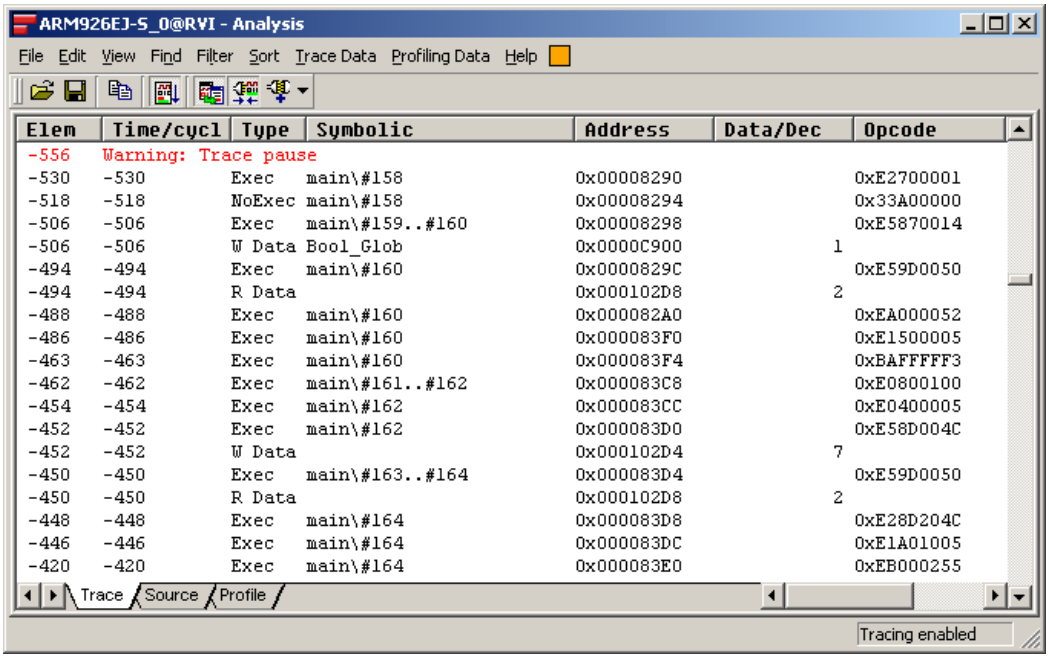



```
148  for (Run_Index = 1; Run_Index <= 10000; ++Run_Index)
149  {
150
151    Proc_5();
152    Proc_4();
153      /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
154    Int_1_Loc = 2;
155    Int_2_Loc = 3;
156    strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
157    Enum_Loc = Ident_2;
158    Bool_Glob = ! Func_2 (Str_1_Loc, Str_2_Loc);
159      /* Bool_Glob == 1 */
160    while (Int_1_Loc < Int_2_Loc)  /* loop body executed once */
161    {
162      Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
163        /* Int_3_Loc == 7 */
164      Proc_7 (Int_1_Loc, Int_2_Loc, &Int_3_Loc);
165        /* Int_3_Loc == 7 */
166      Int_1_Loc += 1;
167    } /* while */
168      /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
169    Proc_8 (Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
170      /* Int_Glob == 5 */
171    Proc_1 (Ptr_Glob);
172    for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
173                                /* loop body executed twice */
174    {
175      if (Enum_Loc == Func_1 (Ch_Index, 'C'))
176          /* then, not executed */
177        {
178        Proc_6 (Ident_1, &Enum_Loc);
179        strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 3'RD STRING");
180        Int_2_Loc = Run_Index;
181        Int_Glob = Run_Index;
182        }
183    }
184      /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
185    Int_2_Loc = Int_2_Loc * Int_1_Loc;
186    Int_1_Loc = Int_2_Loc / Int_3_Loc;
187    Int_2_Loc = 7 * (Int_2_Loc - Int_3_Loc) - Int_1_Loc;
188      /* Int_1_Loc == 1, Int_2_Loc == 13, Int_3_Loc == 7 */
189    Proc_2 (&Int_1_Loc);
190      /* Int_1_Loc == 5 */
191    // make loop infinite
192    if (Run_Index >= 10000)
193      Run_Index = 1;
194  } /* loop "for Run_Index" */
```

Select *Debug → Set PC to Entry Point* from the main menu, then press F5 to run to the breakpoint set earlier on line **192**.

Execution halts on the breakpoint. Select *View → Analysis Window* from the RVD menu to view the captured trace.





Scroll through the captured trace data to see the source code and disassembly synchronized with each line of trace data.

For a more in-depth trace tutorial, refer to *Application Note 168 'Tracing with RVD'* on the ARM web site.

Summary

In this tutorial, you have:

- Installed RVDS and the RVI software
- Connected your RVI and RVT to your computer and target board
- Updated your RVI's firmware
- Configured RVD

You have learnt:

- How to rebuild the example images
- Basic Scatterfile usage
- How to connect to your target
- How to debug a program using RVD
- How to use automatic trace
- How to define a trace range