eGui Eclipse™ User Guide

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	3.0.1
Filename:	eGui_Eclipse_User_Guide.doc
Project:	Imperas Eclipse Project eGui
Last Saved:	Tuesday, 23 March 2021
Keywords:	

Table of Contents

1 Preface

This document describes how to debug an application running on the OVP or Imperas Professional simulator using the Imperas eGui (based on Eclipse™) Integrated Development Environment.

The Imperas eGui is based upon Eclipse 2020.03 and provided as the installation package eGui_Eclipse. This provides a standalone version of Eclipse.

This package must be installed and used in conjunction with a standard OVP or Imperas product package installation.

The example in this document demonstrates debugging of ARM applications but the same approach is valid for applications running on processor models for any architecture supported by the OVP simulator except the OR1K, because the GDB provided with the OR1K processor is obsolete and does not support the MI interface needed to be used with Imperas eGui.

1.1 Notation

`Code` Code and command extracts

1.2 Related OVP Documents

The following documents are part of the OVP and Imperas installations and can be found in the directory:

$IMPERAS_HOME/doc/ovp

- Imperas Installation and Getting Started Guide
- OVPsim and CpuManager User Guide

1.3 Related Imperas Documents

The following documents for the Imperas Professional Tools are part of the Imperas installation and can be found in the directory:

$IMPERAS_HOME/doc/imperas

- Imperas Debugger User Guide

2 Introduction

This document describes how to install and use the Imperas eGui (based on Eclipse) debugger for interactive debugging of OVP virtual platforms.

The Imperas eGui debugger (hereafter referred to as eGui) is based on the Eclipse IDE, version 202003 with the additions of:

1. The Imperas eGui feature, which enhances Eclipse to connect to an Imperas simulation for debugging.
2. An enhanced version of the Eclipse C Development Tools (CDT) feature which adds support for debugging multiple processors simultaneously when using the Imperas Multi Processor Debugger (MPD).

eGui can be used as a standalone debugger for connecting to the Imperas simulator or it can be used as a full Eclipse Integrated Development Environment (IDE), allowing you to manage and build projects. The Eclipse IDE features in eGui are standard Eclipse functionality so consult Eclipse documentation for info on using eGui as an IDE.

If you already use another IDE, (such as VS Code or a different version of Eclipse), and wish to continue using your existing IDE, then eGui can still be used as a standalone debugger, independent of your IDE.

Note that the Imperas simulator provides a standard RSP interface usable with standard debuggers, such as gdb or lldb, and as such can be integrated into any debugging environment that supports the RSP interface. When using the standard RSP interface the Imperas-specific enhancements described above, such as MPD, are not available, and this document does not cover debugging an Imperas simulation in any other environment than eGui.

2.1 eGui Modes

The eGui feature supports connecting to Imperas simulations in either GDB or MPD mode. GDB mode is supported by all Imperas simulation environments. MPD mode requires an MPD license, which is part of the IMPERAS SDK product.

GDB Mode:

- Debug using the GNU debugger (GDB). For Imperas products that support multiple GDB connections, each GDB connection is independent and all must be put into a run state for simulation to proceed.

MPD Mode:

- Simultaneous debug of all processors and peripheral models. MPD handles the details of starting and stopping all processors in the simulation while debugging.
- Programmers View of registers and other model elements provided by the processor and peripheral models.
- VAP tools which provide additional powerful debugging capabilities.

3 Installing Imperas eGui

3.1 Prerequisites

The following OVP and/or Imperas packages must be installed to use eGui:

One of:

- OVPsim
- Imperas_DEV
- Imperas_SDK
- or riscvOVPsimPlus[1]

Plus:

- eGui_Eclipse

The *Imperas Installation and Getting Started Guide* provides a step-by-step guide to obtaining and installing the OVP and Imperas packages containing the simulator and other tools. It is strongly recommended that you follow that guide until you are able to build and simulate before attempting to debug using eGui.

In particular the Imperas environment must be setup as described in the Installation guide.

3.2 Installing the eGui_Eclipse package

eGui can be added to your Imperas installation by installing the eGui_Eclipse package. This package is provided in a self installing executable file available for download from the ovpworld.org or imperas.com websites (registration required).

The installer is a file named:

eGui_Eclipse.<version>.<arch>.exe

(Note, for consistency the executable file name includes the .exe suffix on both Linux and Windows.)

For Linux it may be necessary to make the file executable with the command:

chmod +x <fn>

To install, simply execute the installer and follow the directions. See the *Imperas Installation and Getting Started Guide* for additional information.

This installs into directory $IMPERAS_HOME/lib/$IMPERAS_ARCH/eGui.202003 the following:

[1] The installation of riscvOVPsimPlus does not require an Imperas environment to run. Therefore, if using this simulator, the Imperas environment must be setup separately to use eGui. Please use the setup scripts provided with this eGui installation and consult the Installation and Getting Started Guide from the OVPWorld website for more information.

1. The Imperas eGui (based on Eclipse) standalone product.
2. A Java Runtime Executable (JRE).

4 Starting a debug session

There are several ways to start a debug session depending on whether you want to just run a standalone debug session or are using Eclipse as your IDE to edit, build, run and debug your projects, and also whether you have a license for the Imperas Multi-Processor Debug (MPD) utility.

With Imperas MPD support all of the processors and peripherals in the platform may be debugged simultaneously. Without Imperas MPD support, GDB is used directly and only a single processor or peripheral may be debugged at a time.

4.1 Launching eGui from the simulation command line

This is the simplest way to start a debug session. The following simulator command line arguments are available for starting an eGui debug session connected to the simulation:

--mpdegui

This will connect to eGui in MPD mode,. MPD mode supports simultaneous debug of all processors and peripherals in the platform. This require an Imperas MPD license.

--gdbegui

This will connect to eGui in GDB mode. GDB mode supports debug of a single processor only. In platforms with multiple processors the --debugprocessor <platform>/<cpu> option will also be needed to specify the processor to be debugged, where <platform> and <cpu> are the respective platform and processor names.

eGui will be started if it is not already running. If eGui is already running then a new debug session will be started in it. (You should normally terminate any running debug session before starting a new one.)

4.2 Specifying platform options

Debugging is enabled when starting an OVP platform by specifying command line or control file options, or can be built into the platform source.

When using *iss.exe* or *platform.exe*, or when running a custom platform that includes the Imperas Command Line Parser (CLP), then command line options may be used to enable debugging without the need to make any changes to the platform source. The examples in this document assume the command line options are supported, which is true for all examples and demos provided by Imperas.

If using a custom platform that does not support the standard Imperas command line options see the *Simulation Control of Platforms and Modules User Guide* for information on adding the CLP to a platform. An alternative to the CLP is to use a control file to specify platform options at runtime. See the *OVP Control File User Guide* for

information on using control files. The platform source may also enable debugging using OP API calls. See the OP API documentation for information, although the CLP makes the use of these unnecessary in most cases.

4.3 Starting a standalone debug session

The following platform options are available for starting a standalone debug session:

--mpdegui

This will connect to eGui in MPD mode, starting eGui if it is not already running. MPD mode supports simultaneous debug of all processors and peripherals in the platform. This require an Imperas MPD license.

--gdbegui

This will connect to eGui in GDB mode, starting eGui if it is not already running. GDB mode supports debug of a single processor only. In platforms with multiple processors the `--debugprocessor <platform>/<cpu>` option will also be needed to specify the processor to be debugged, where platform and cpu are the respective platform and processor names.

4.3.1 The eGui port file

A standalone eGui debug session is initiated from the simulator. The simulator determines whether eGui is already running by looking for the eGui port file. This file is created when an Eclipse with the eGui feature starts up and it contains the TCP port number that eGui listens to for connections.

The file name for the eGui port file is determined by:

1. The value of the IMPERAS_EGUI_PORT_FILE environment variable, if specified.
2. If that environment variable is not set then the file name defaults to:
   $HOME/.egui.port on Linux
   %USERPROFILE%\.egui.port on Windows

If the eGui port file exists and a listener is found on the port, then a standalone debug session is started in the running eGui Eclipse which connects to the newly started platform.

If the eGui port file does not exist, or if no listener is found on the port, then eGui Eclipse is launched which then connects to the newly started platform.

Note:

- If the eGui feature has been added to an existing Eclipse installation, then using *--mpdegui* or *--gdbegui* will connect to it *if* it is already running, since the eGui feature in that Eclipse will have created an eGui port file.

- When no valid eGui port file exists the platform will only start the eGui product in the Imperas installation - it will not start an Eclipse that has added the eGui feature. Thus if you wish to use your own Eclipse installation with the eGui feature for standalone debugging it must already be running when the platform is started.
- If multiple instances of the eGui feature run simultaneously the eGui port file will contain the port for the one started most recently. The IMPERAS_EGUI_PORT_FILE environment variable may be used to control connecting to different instances running simultaneously.

4.4 Starting a debug session from Eclipse

For users who use Eclipse as their IDE, starting a debug session from within Eclipse may be more convenient, but requires setting up Eclipse launch configurations to both run the platform and start the debug session.

4.4.1 Starting the Imperas eGui Eclipse product

The eGui_Eclipse package includes a fully functional Eclipse, referred to here as eGui. Users may wish to use this as their IDE in addition to using it as a standalone debugger as described in section 4.3. This section contains information on running the eGui Eclipse installation provided by Imperas as an IDE.

eGui Eclipse may be started by executing the command *egui.exe* from any Linux or MSys shell or a Windows Command Prompt that has been setup with the Imperas environment. *egui.exe* is a wrapper program that verifies the environment and launches the eGui product found in:

$IMPERAS_HOME/lib/$IMPERAS_ARCH/eGui.202003/eguieclipse{.exe}

egui.exe arguments include:

--help
Print a help message listing all arguments

--version
Print the version and exit

--verbose
Print additional information that may be useful for troubleshooting

--open <fn>
Open the indicated file in Eclipse, using the default viewer for the file type. This is primarily intended to be used for viewing the VAP Tools functionprofile (*.iprof) and linecoverage (*.icov) results but can be used for any file type supported by eGui.

--eguioptions <string>
Specifies Eclipse command line options to be added to the eguieclipse command line. Consult the Eclipse documentation *Running Eclipse* section for the valid

options. Multiple *--eguioptions* may be specified and they will be combined. Option strings containing spaces may be specified by enclosing in quotes.

--eguicommands <string>

Specifies Eclipse commands to be sent to eguieclipse at startup. Command strings containing spaces may be specified by enclosing in quotes.

(There are several additional arguments intended for internal use by the simulator when launching standalone debugging sessions that are not documented here.)

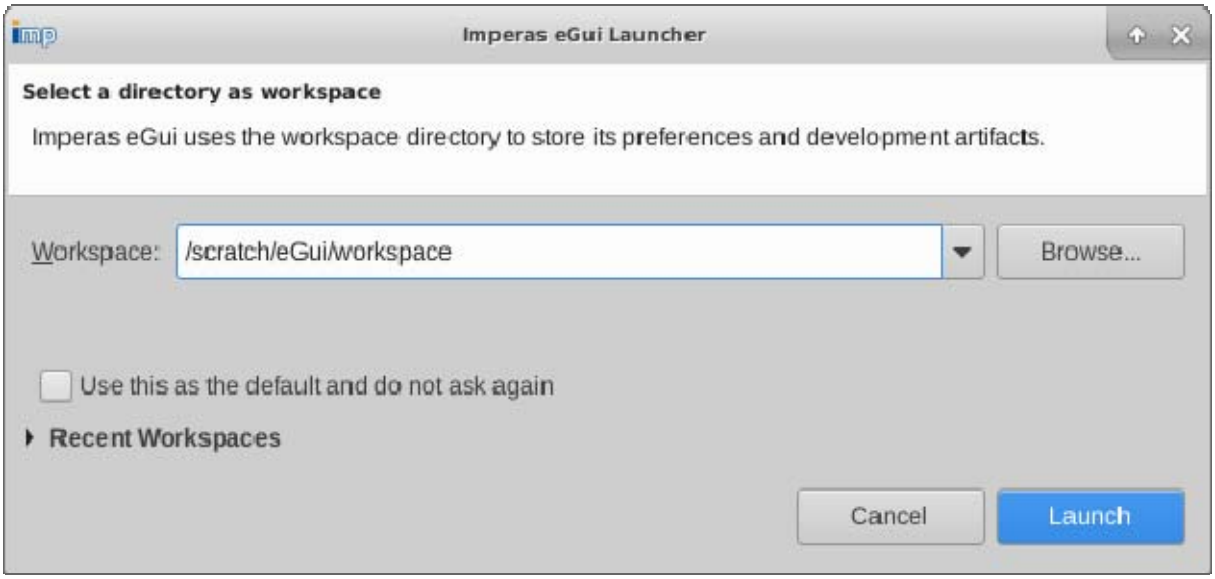
⇒ Note that Eclipse only supports options that start with '-' and will silently ignore options that start with '--'.

4.4.1.1 The Java Runtime Environment (JRE)

The eGui_Eclipse package includes a Java Runtime Environment (JRE) whose root is in the same directory as the eguieclipse executable and will be used by Eclipse by default. The Eclipse *-vm* command line option may be used to override this default JRE. See the Eclipse documentation for additional details.

4.4.1.2 Selecting a workspace

When eGui is started a workspace prompt will appear (unless the Eclipse *-data* option specifying the workspace directory is specified):



The workspace is where Eclipse stores settings for the session. Settings from a previous session may be reused by using the same workspace.

For standalone debugging you may want to just use a temporary workspace, perhaps in your working directory.

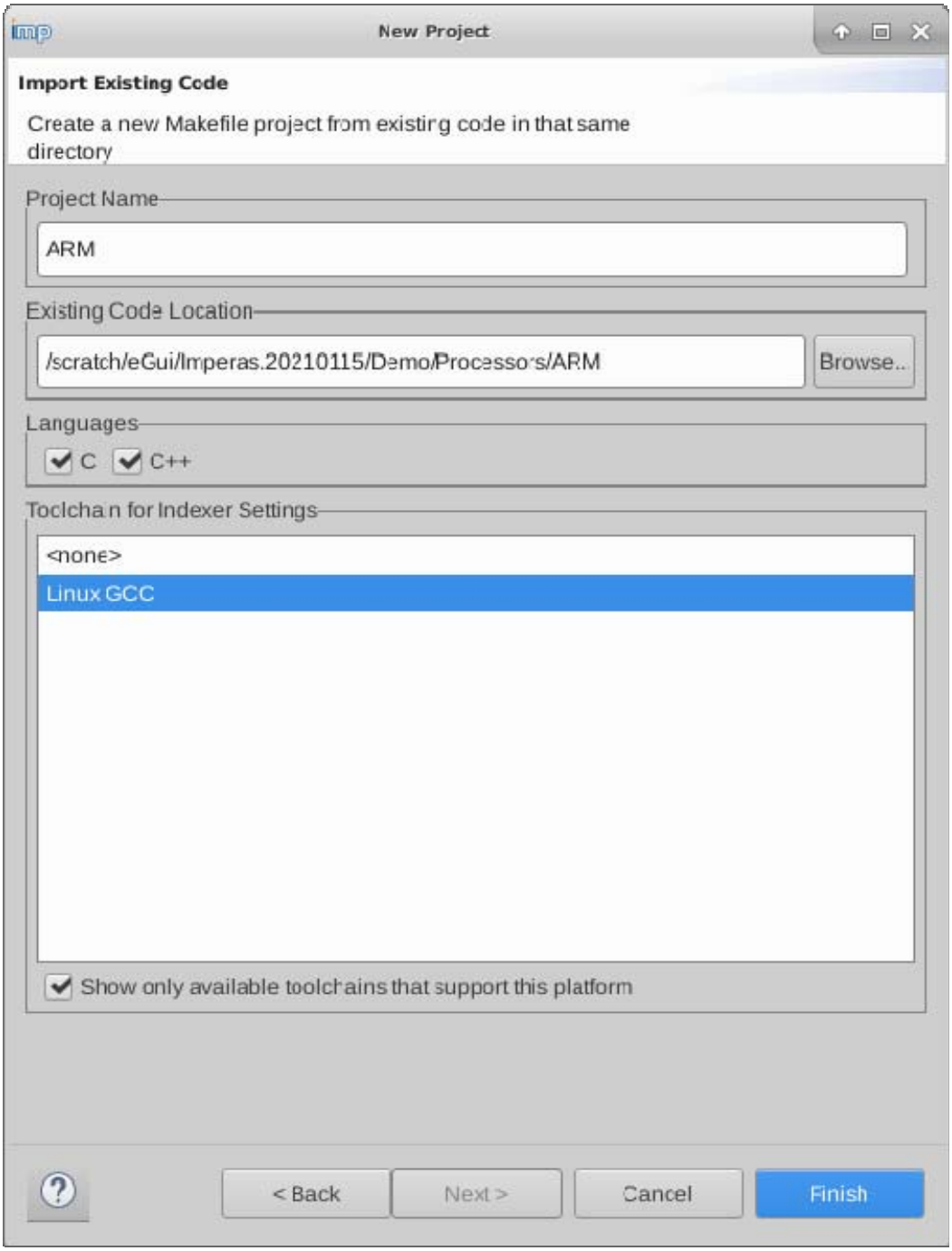
If you are using Eclipse as your IDE you probably want to use a permanent workspace that will save projects and other information that you configure, so select an appropriate

location for the workspace, perhaps in your home directory. Note that the workspace cannot be under a directory that will be imported as a project.

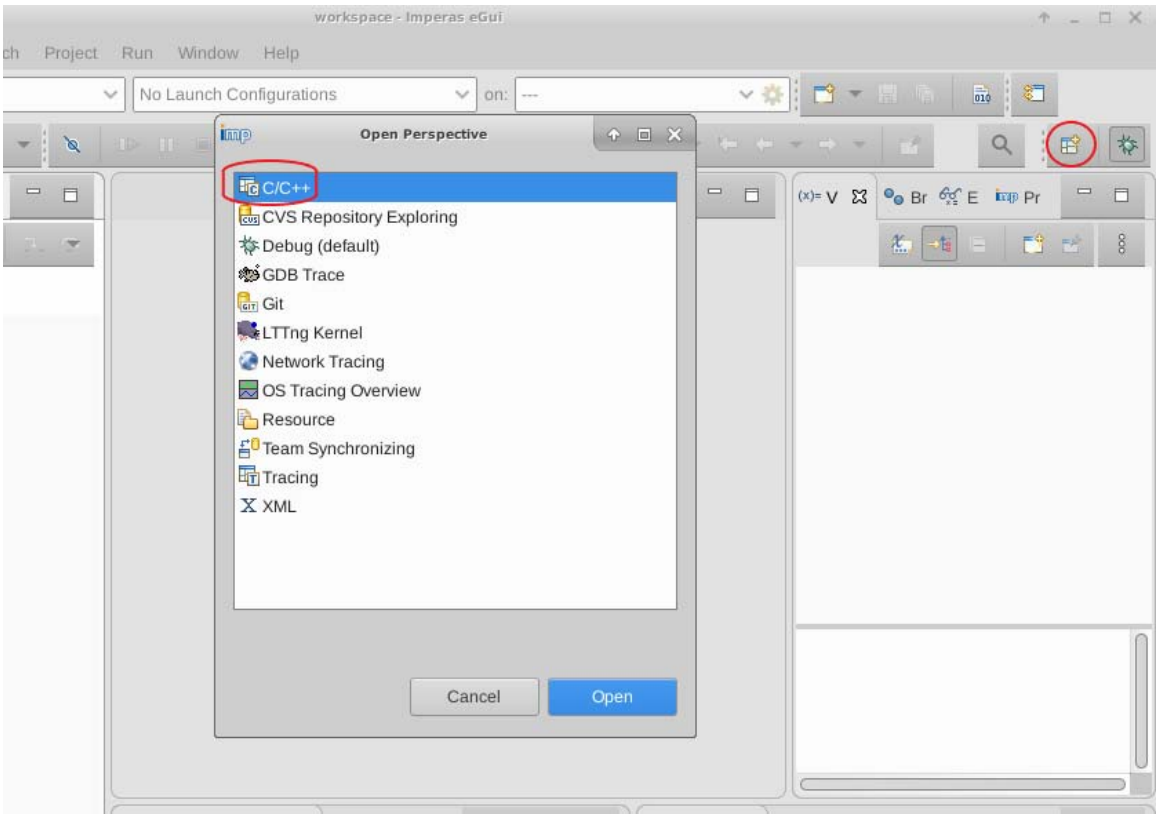
See the Eclipse documentation for additional information about workspaces.

4.4.2 Importing a project

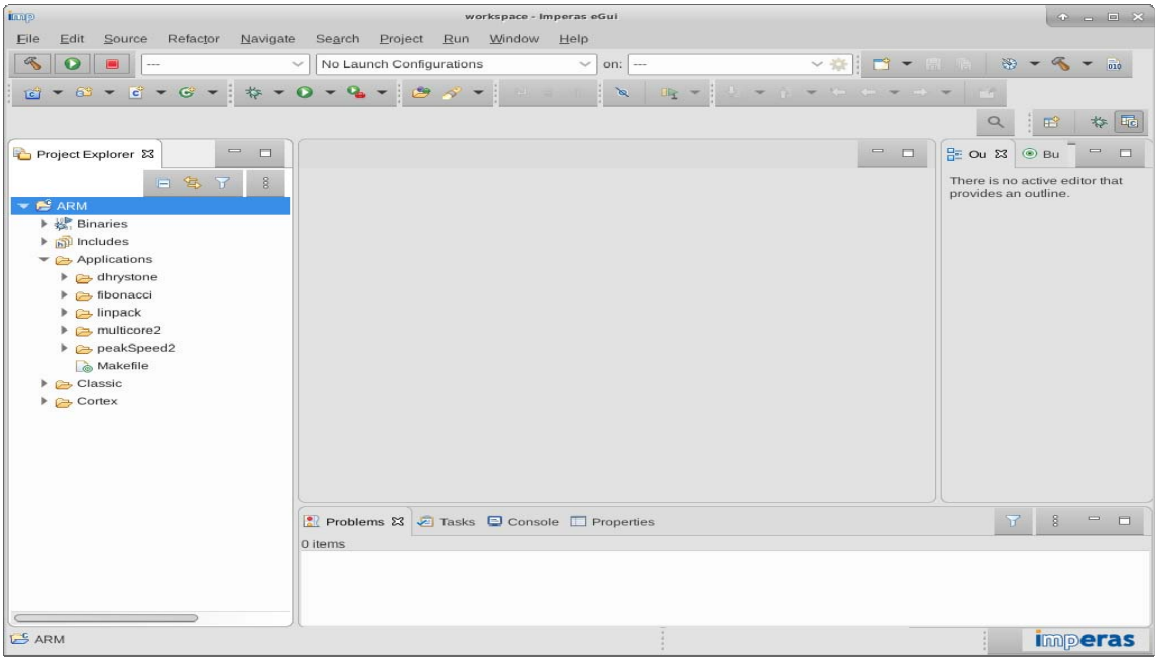
To use eGui as your IDE you need to create a project. We will walk through the process of importing an Imperas Demo directory as a project. First select from The Eclipse menu *File->Import->C/C++->Existing Code as Makefile Project* and select *Next*. In the dialog box that comes up select Browse and browse to the *$IMPERAS_HOME\demo\Processors\ARM* directory, for example, and then select *Finish*:



In order to see the project we must open the *C/C++* perspective by selecting the *Open Perspective* button in the top right corner and then choosing *C/C++* from the list of perspectives and selecting *Open* (alternatively, the Eclipse menu selection *Window->Perspective->Open Perspective->C/C++* may be used*):*



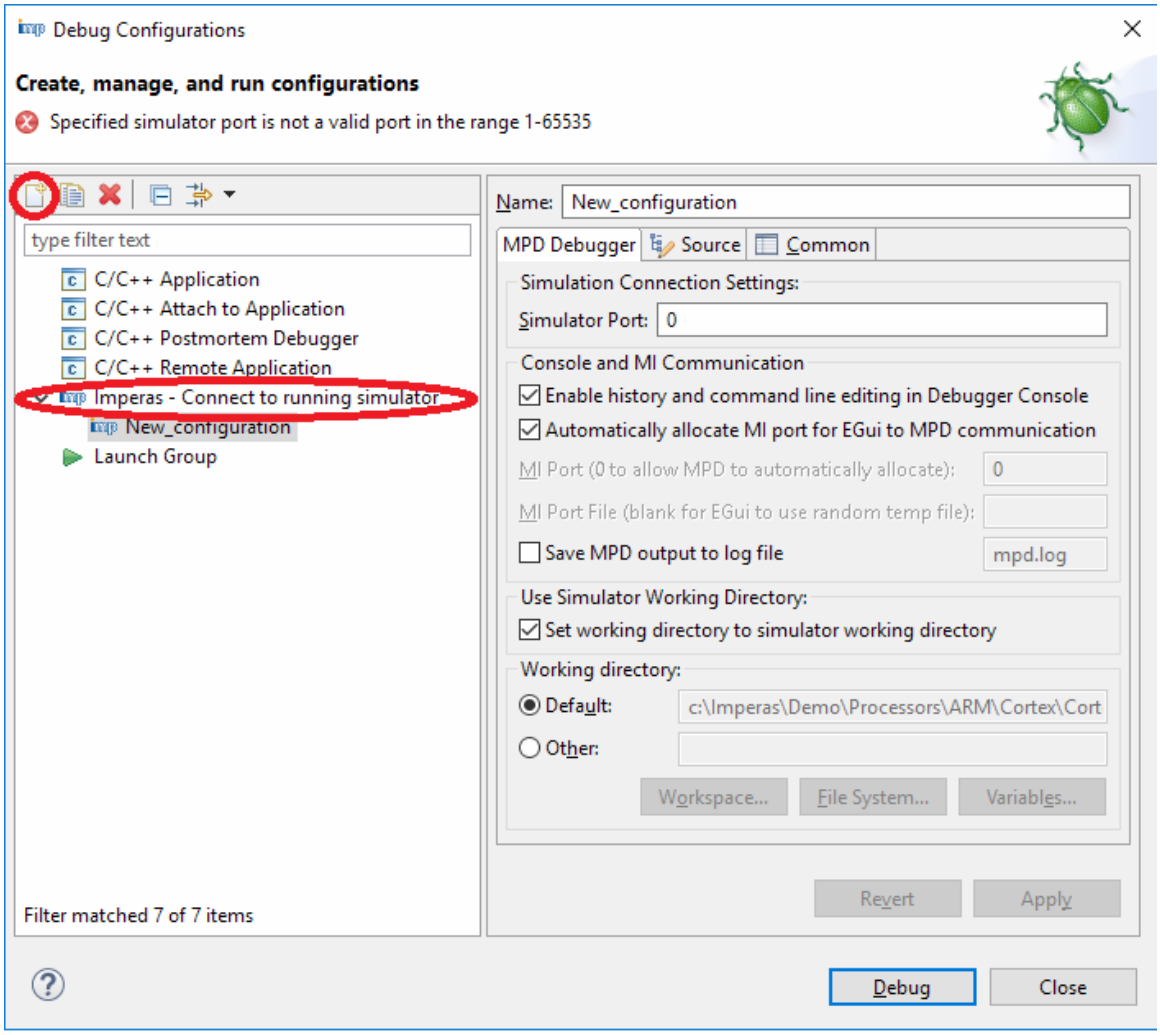
The imported project *ARM* will now be visible in the Project Explorer view of the C/C++ perspective:



4.4.3 Eclipse launch configurations

To start a debug session from within Eclipse you use launch configurations. eGui adds an Imperas-specific launch configuration specifically to launch Imperas debug sessions.

To create an Imperas Debug Launch Configuration select *Run->Debug Configurations...* Then select *Imperas - Connect* and then the *New launch configuration* (the rectangle with a + in the top left corner):



The fields are as follows:

Name

This is the name assigned to the debug configuration, similar to standard Eclipse configurations.

Port

This is the port that the platform is listening to, waiting for a connection. It should match the value specified when the platform was launched, or if a port of 0 was specified for the platform it should match the port number reported by the platform.

Console and MI Communication options

These are useful for debugging problems and should be left in the default state that is shown unless instructed to change them by Imperas support.

Set working directory to simulator working directory

When this is checked the debug session will run with the current directory set to the same working directory as the simulation that it connects to. This should normally be checked.

Working Directory

If the working directory is not obtained from the simulator it may be specified here.

The rest of the tabs function the same way as any other Eclipse debug configuration dialog and the Eclipse documentation should be consulted for information on them.

Examples of the use of this debug configuration may be found below.

⇒ To use the *Imperas - Connect* debug configuration you must have a license for the Imperas MPD.

4.5 Additional Features in eGui exclusively for Imperas MPD users

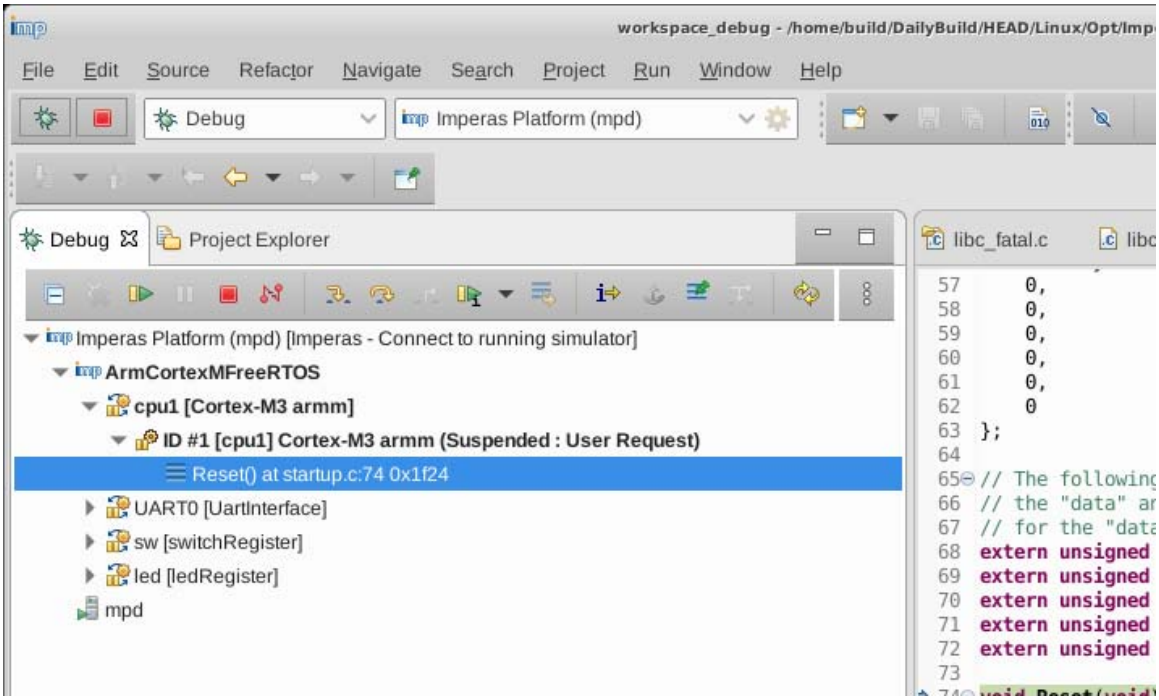
eGui provides the following additional features beyond those offered by the standard Eclipse CDT:

- Support for the launch of eGui under control of the simulator and automatically connecting in one of the following modes:
  - Debug with GDB of a single processor or Peripheral
  - Debug with independent GDB sessions of multiple processors and peripherals (Imperas DEV required)
  - Simultaneous debug of multiple processor and peripheral models (Imperas MPD required)
- Simultaneous debug of multiple processors (MPD required)
- Imperas Programmers View object display (MPD required)
- MPD Debugger Console including command history/editing for issuing MPD and VAP commands (MPD required)

A brief description of each of these features follows. More detailed information can be found through out the rest of this document.

4.5.1 Simultaneous debug with MPD

When using MPD the Eclipse Debug View in eGui has been enhanced to show all the processor and peripheral models that are available for debugging. A typical eGui Debug View is shown here:



Under the *Imperas Platform (mpd) [Imperas - Connect to running simulator]* entry all the simulated processors and peripherals are listed (here, they include *ArmCortexMFreeRTOS/cpu1*, *ArmCortexMFreeRTOS/UART0*, ...). The current call stack of any model may be viewed by opening the drop down list for the model.

When simulation returns control to the debugger (e.g. due to a breakpoint being hit or the debugger requesting an interrupt) all the simulated processors and peripherals are stopped and any of them may be debugged by selecting them in the Debug View.

By selecting different lines in the Debug View (in the above figure the *Reset()* function entry in the call stack for *ArmCortexMFreeRTOS/cpu0* is selected) the focus of the debugger is changed to that processor and all other views in the debugger will be updated to reflect the current selection.

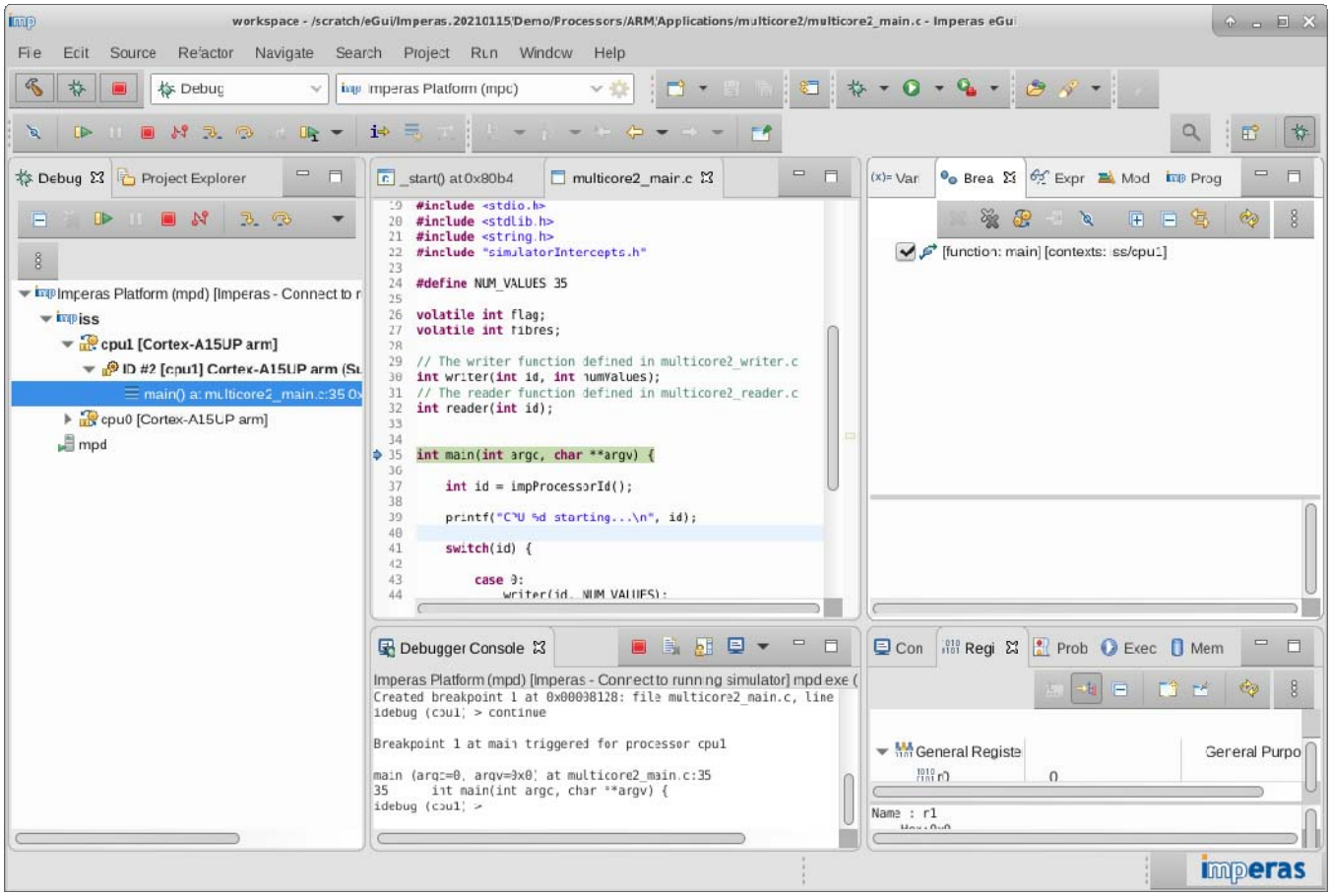
4.5.2 Multiple processor support in breakpoints

The Breakpoints view has been enhanced to support defining processor-specific breakpoints.

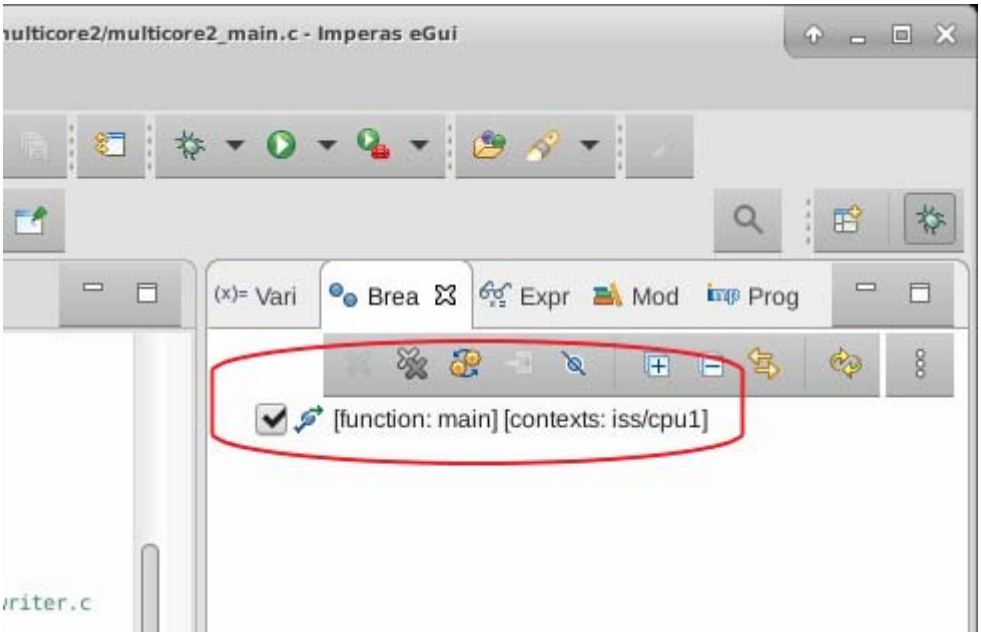
When setting a breakpoint in eGui it is set on only the current active processor by default. As an example, go to the ARM Cortex-A15UP multi_core demo and run it with the --*mpdegui* option:

```
> cd $IMPERAS_HOME/Demo/Processors/ARM/Cortex/Cortex-A15UP/multi_core
> ./Run_MultiCore2.sh --mpdegui
```

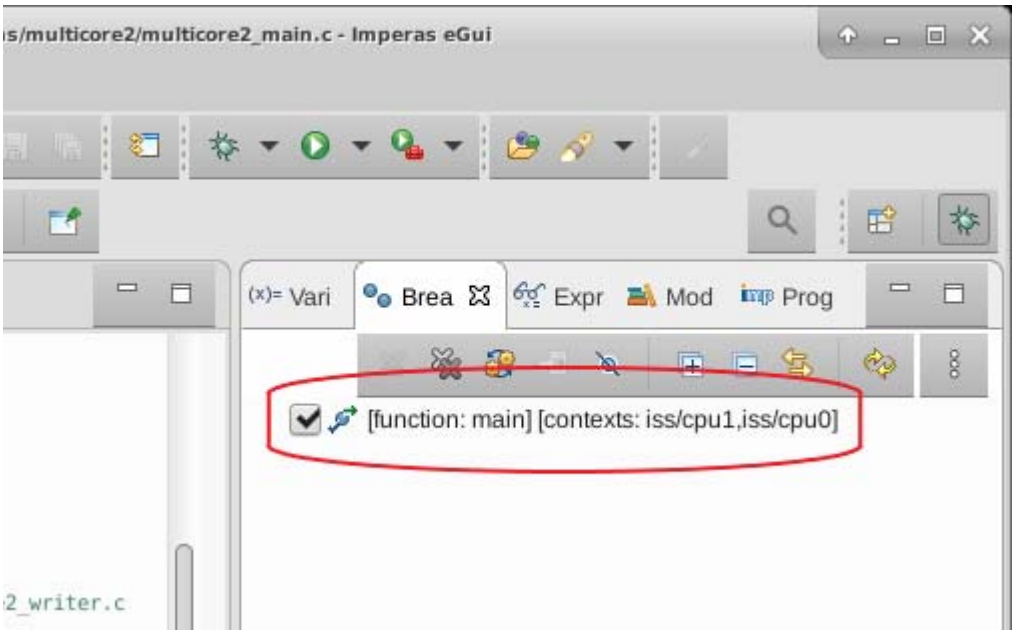
After starting eGui select *iss/cpu1* and enter the commands *b main* and *continue* in the Debugger Console window:



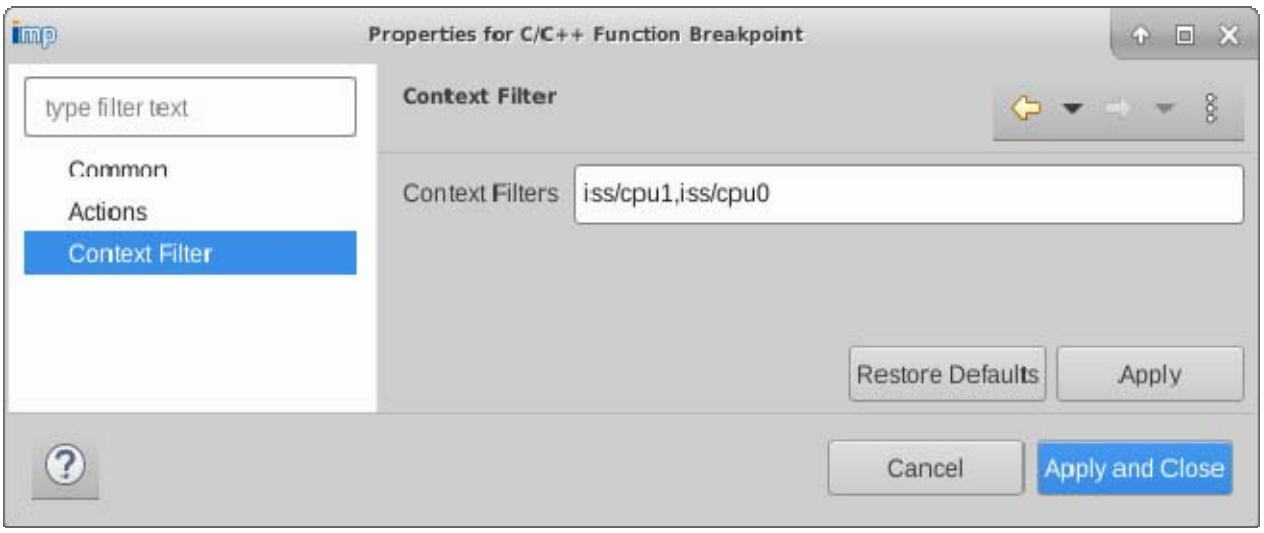
We can see in the Breakpoint view that the newly created breakpoint has a context of *iss/cpu1*:



If we subsequently select *iss/cpu0* in the Debug view, then enter the command *break main* in the Debugger Console again and look at the breakpoint we will see *iss/cpu0* added to it:



The breakpoint properties menu may also be used to change a breakpoint's context after the breakpoint has been added by right-clicking the breakpoint and selecting *Breakpoint Properties...* from the context menu, and modifying the *Context Filter* setting:

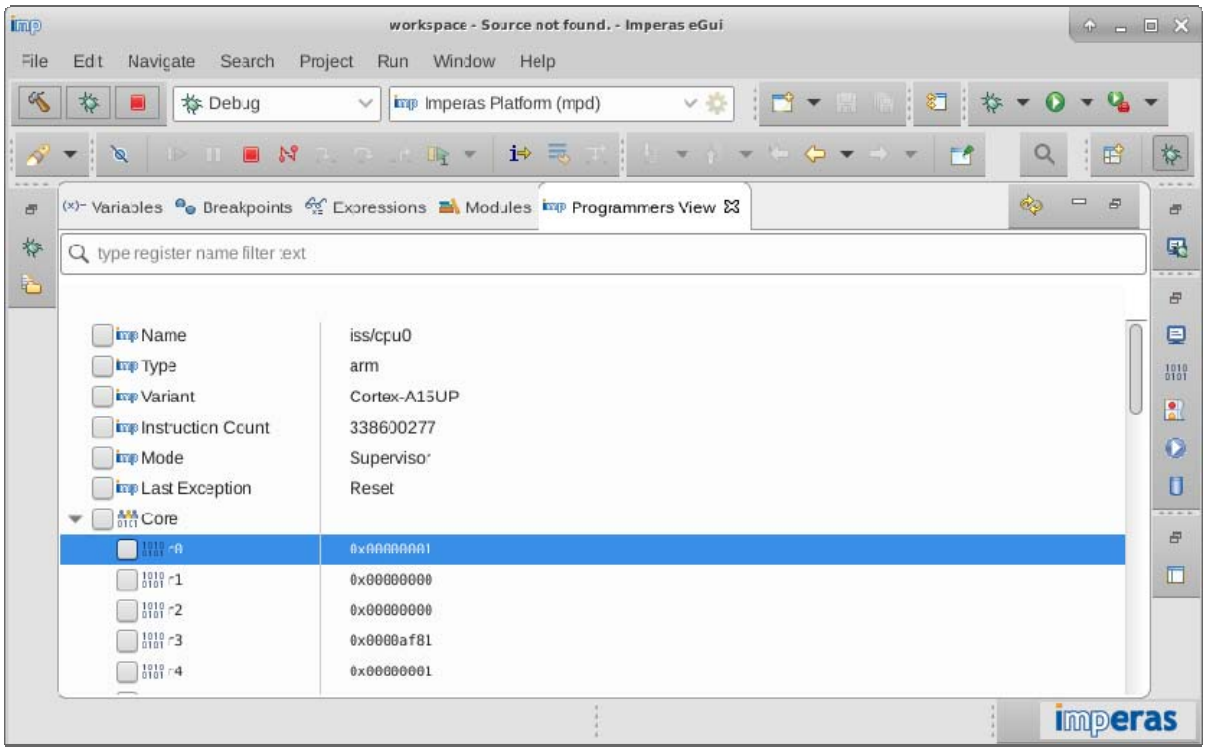


NOTE: Opening the breakpoint properties with ALT-Enter rather than right clicking and selecting from the context menu opens a properties menu for gdb that is not supported by MPD. Do not use this method to edit the breakpoint properties when using MPD!

4.5.3 Imperas Programmers View object display

eGui adds a new view that shows the Imperas Programmers View values for the selected processor. The Programmers View is a set of values selected by the model developer to be visible through the Programmers View interface. The values displayed vary depending on the specific model. The Programmers View is by default one of the tabs in the top right panel.

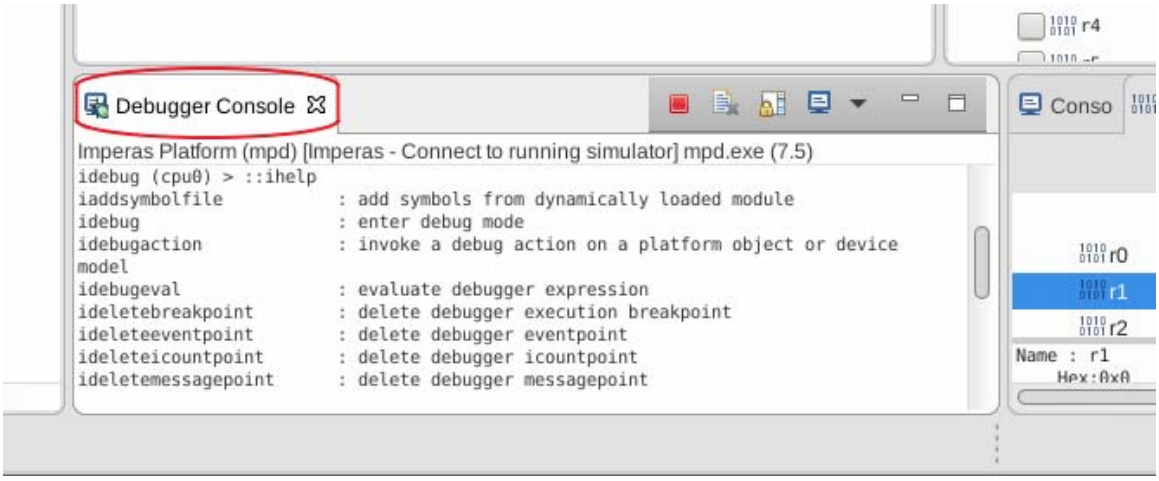
For example, below is the Programmers View for an ARM Cortex processor model (hint: double-clicking on the tab of a view will make it expand to fill the workspace. Double-clicking again will revert back):



Here we see there is various information about the processor including the name, type, variant, instruction count, mode and most recent exception type, as well as the current value of each register (including general purpose, banked and system registers).

4.5.4 MPD Debugger Console for issuing MPD and VAP commands

eGui opens a Debugger Console window that allows commands to be given directly to the Imperas Multi-Processor Debugger (MPD):



Note: If the Debugger Console tab is not displayed you can add the view to your Debug Perspective by selecting *Window->Show View->Debugger Console* from the main Eclipse menus.

The Debugger Console is where MPD commands (including VAP Tool commands) can be directly typed. For a full description of the commands available in this window see the *Imperas Debugger User Guide*.

Note that there are two shell modes described in this document: Debug mode and Tcl mode. Use the command *tcl* to enter Tcl mode from Debug mode, and the command *idebug* to enter Debug mode from Tcl mode. The current mode is shown as part of the command prompt.

In Debug mode use the command *help* to see online help messages for the debug mode. In Tcl mode use the command *ihelp* to see online help messages for that mode.

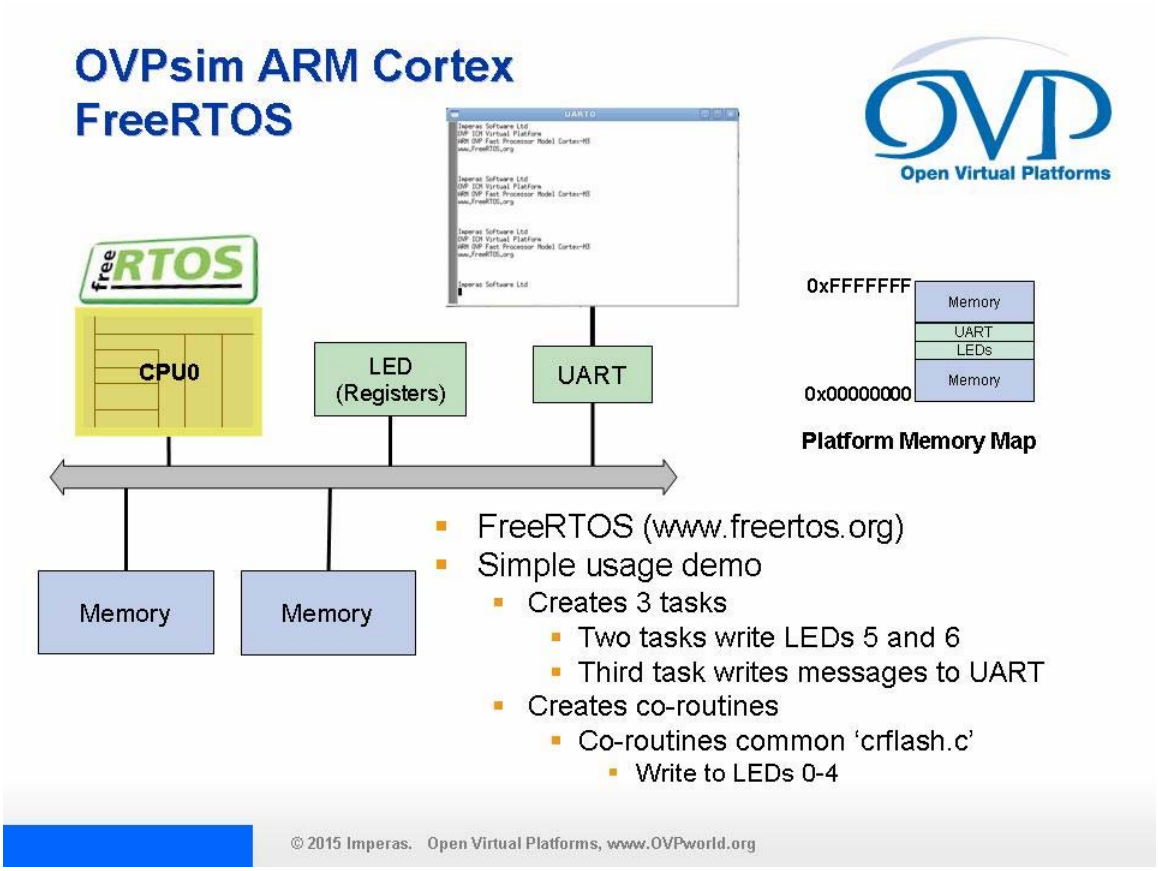
When in Debug mode a single Tcl command may be executed by prefixing it with '::'. See the figure above for an example of executing the Tcl *::ihelp* command in Debug mode.

5 A Sample Debug Session Using MPD

The following is a quick walk through of a debug session using MPD to simultaneously debug a processor and peripheral models.

This example requires the Imperas SDK product. Users of OVPsim or the Imperas DEV product should see the example using GDB instead.

The platform being debugged has a processor, memory, a UART and a LED registers peripheral and is running the FreeRTOS operating system:



5.1 Prerequisites

To follow this example you will need the following Imperas and OVP packages installed:

- Imperas_SDK
- eGui_Eclipse
- Demo_FreeRTOS_arm

The Imperas environment must be set up according to the directions in *Imperas Installation and Getting Started Guide.*

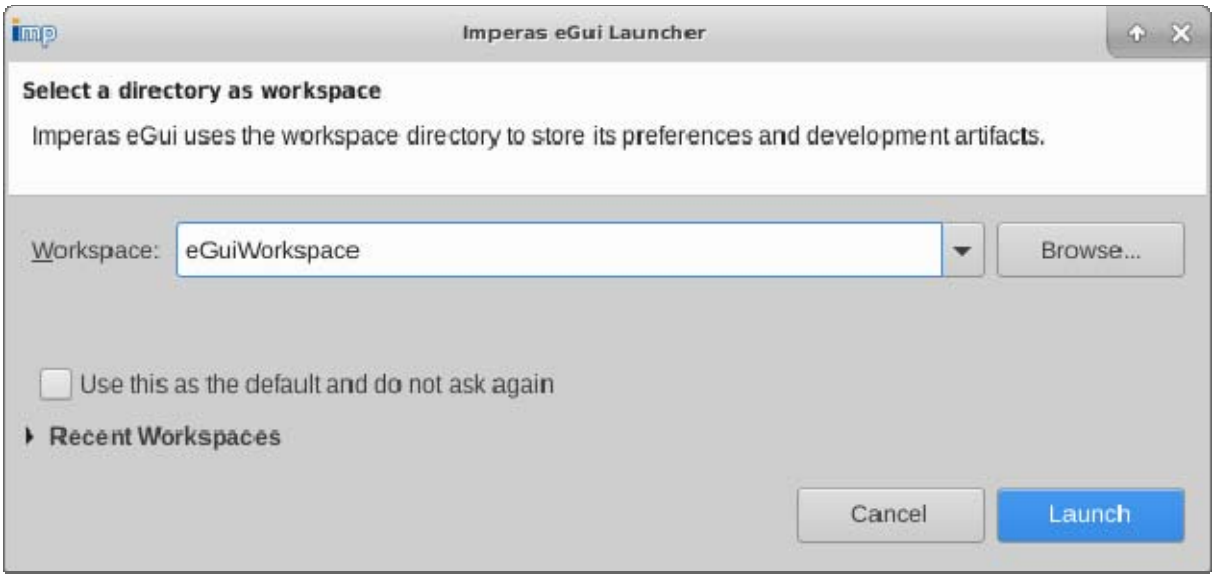
An Imperas MPD license is required. Users without an MPD license should skip this section and see section 6 which uses GDB rather than MPD.

5.2 Starting the debug session

After all the prerequisite packages have been installed and the Imperas environment configured, to start the session enter the following commands from a Linux shell or a Windows MSys shell:

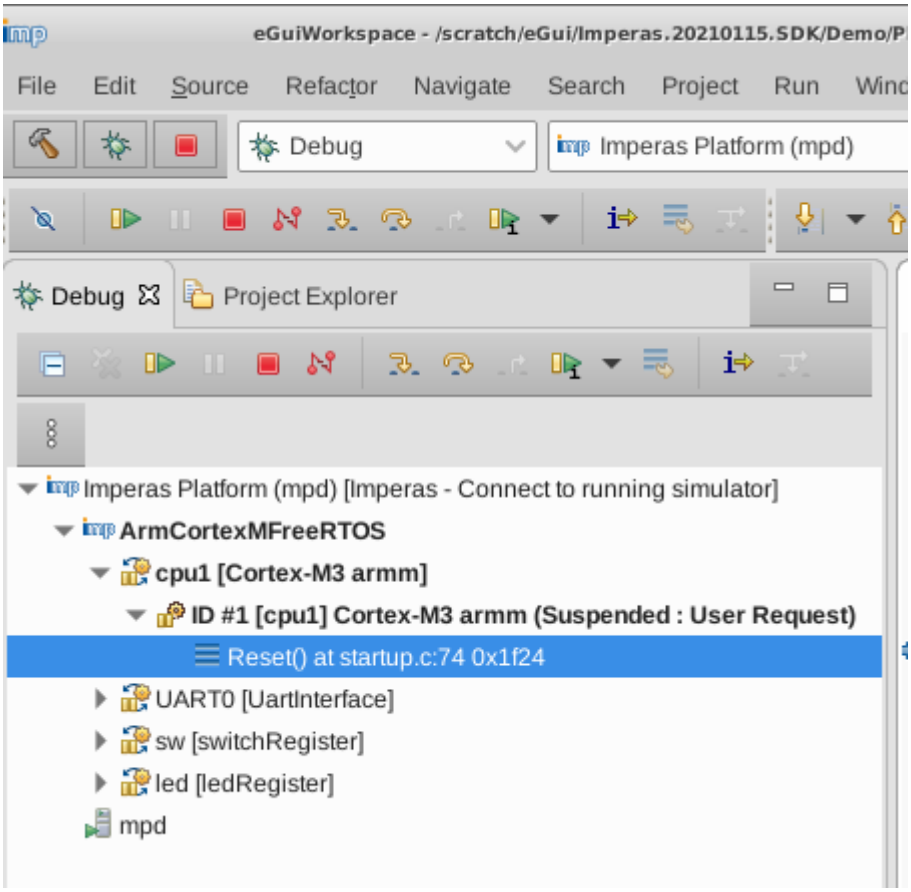
```
> cp -r $IMPERAS_HOME/Demo/Platforms/FreeRTOS_arm .
> cd FreeRTOS_arm/harness
> ./RUN_FreeRTOS.sh --mpdegui
```

When prompted *Do you want to connect browser for user visualization* just answer *No*. eGui will then prompt you to select a workspace. For standalone debugging just use a temporary workspace by selecting "workspace" which will create the directory *workspace* in the current directory:



5.3 The MPD Debug View

The Debug view shows all the debuggable processors in the simulation:



Here we see in the Debug view the following debuggable models listed:

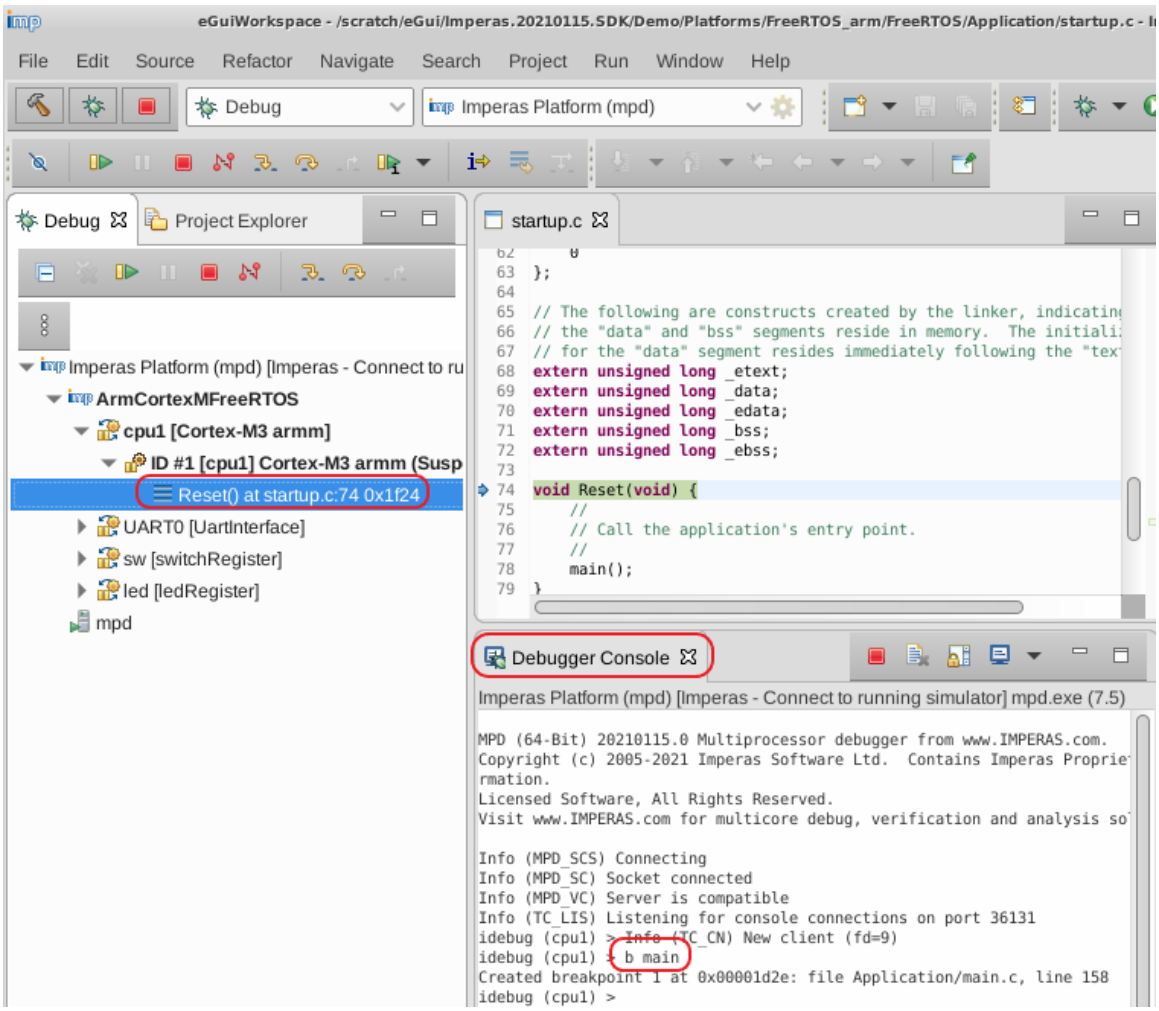
- ArmCortexMFreeRTOS/cpu1
- ArmCortexMFreeRTOS/UART0
- ArmCortexMFreeRTOS/LEDRegister

With MPD we can simultaneously debug all of these processors - when a breakpoint is hit they all will stop and when a resume command is issued they will all start.

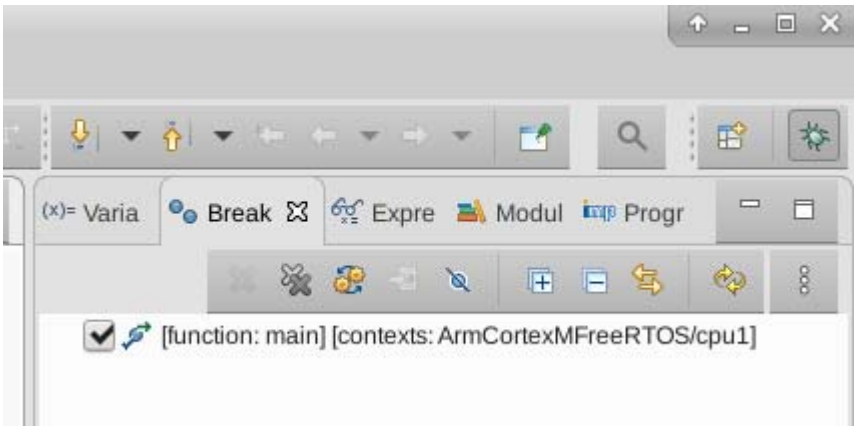
5.4 Setting a breakpoint from the console view

Once the eGui window is open, start the debug session by adding a breakpoint on the symbol main in the ARM processor.

First select *ArmCortexMFreeRTOS/cpu1* in the Debug view and then type the gdb command *b main* in the Debugger Console view:

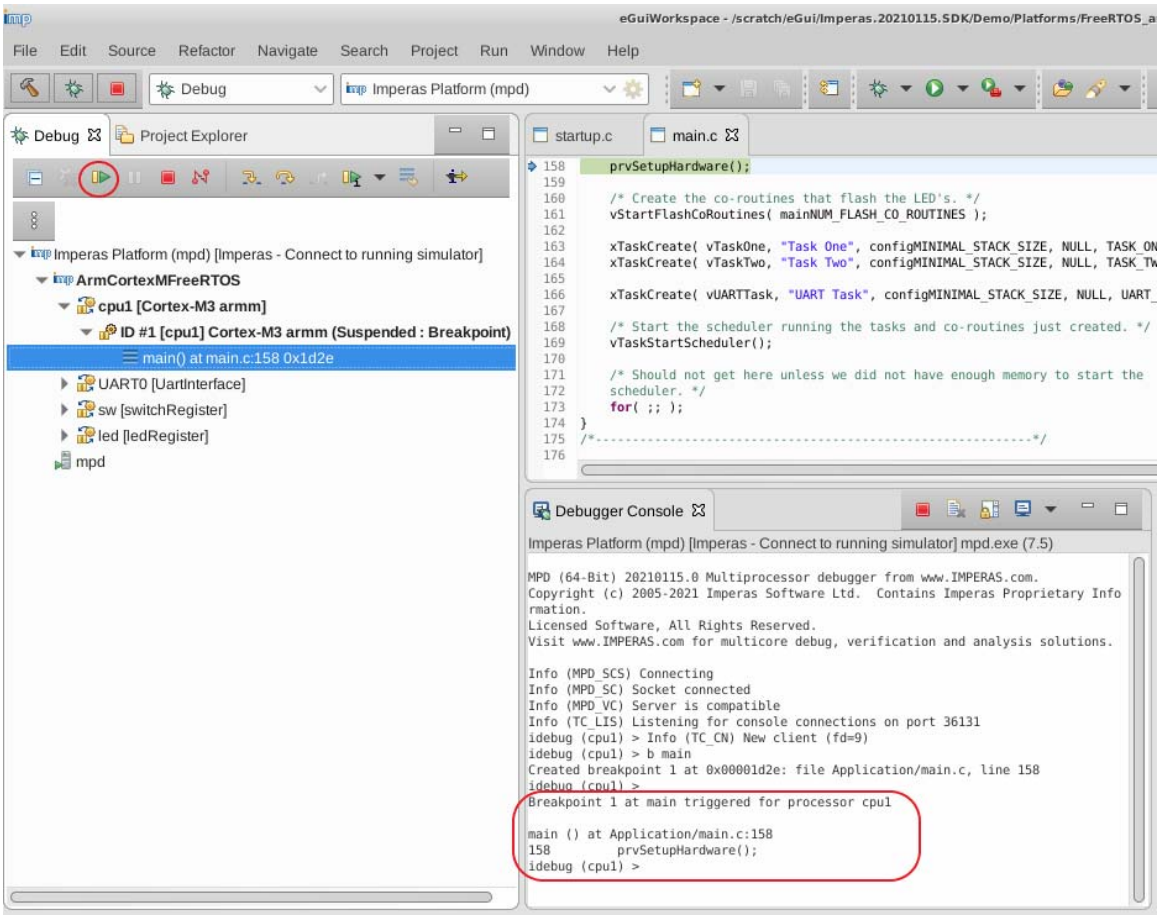


If we select the Breakpoints view we can see the new breakpoint listed:

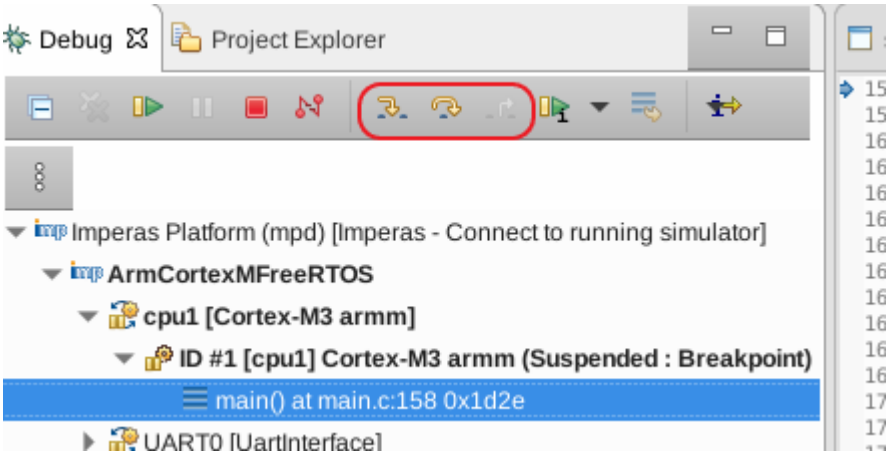


5.5 Running and stepping the simulation

We can now tell the simulation to start running by selecting the *Resume* icon (a yellow bar next to a green triangle) and it will run up to the breakpoint we have set in the function *main*, and display the source of the *main* function:

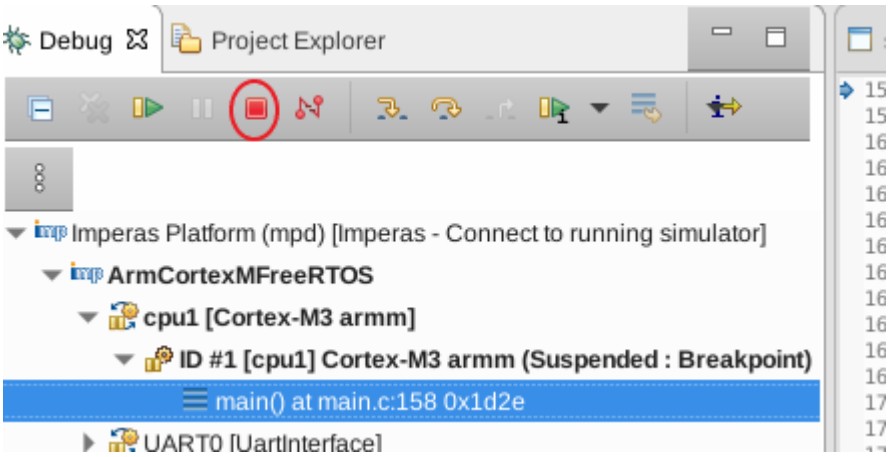


At this point we can use the *step into*, *step over* and *step return* buttons to step through the program (Note: f5, f6 and f7 respectively may be used in place of each button):



5.6 Terminating the simulation

If at any point we find we need to restart the simulation then simply select the Terminate icon (a red square) or press Ctrl-F2:



At this point we can re-run the simulation command without first shutting down eGui:

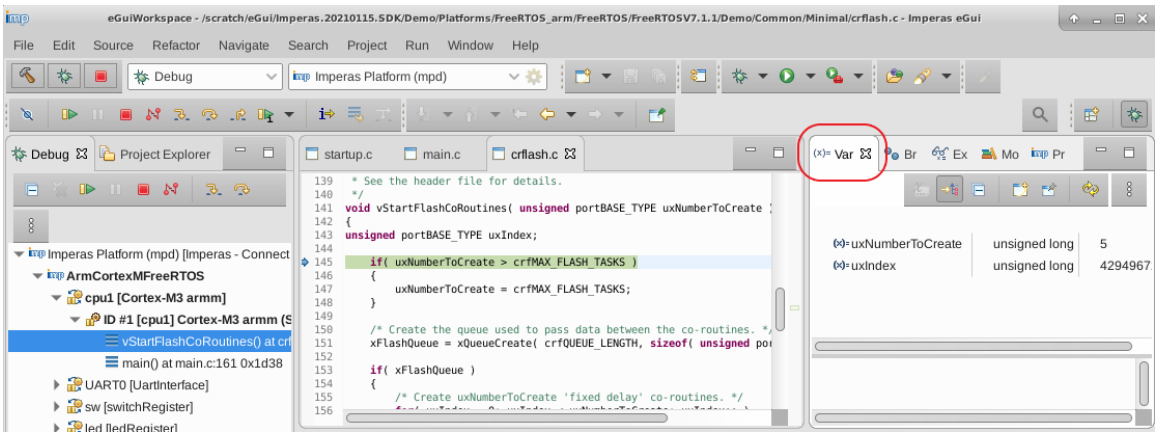
```
> ./RUN_FreeRTOS.sh --mpdegui
```

and a new debug session will be started without the overhead of restarting eGui. In addition any breakpoints we had are set again in the new simulation.

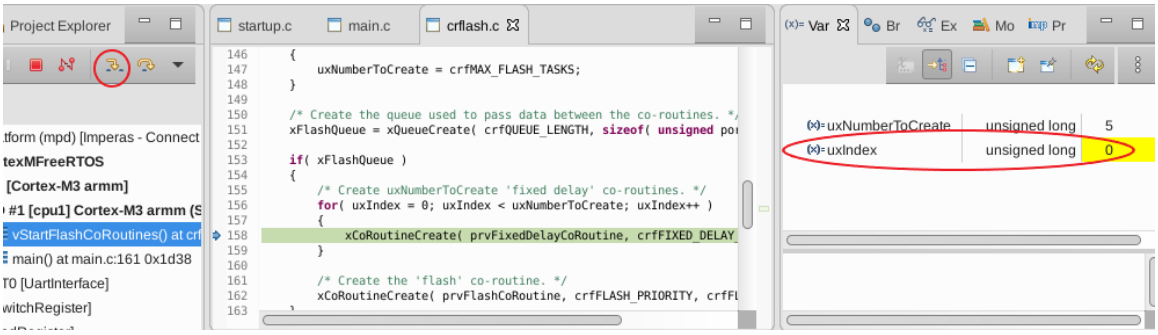
5.7 Viewing local variables in a function

Press *step over* (or f6) once and then *step into* (or f5) to step into the *vStartFlashCoRoutines* function.

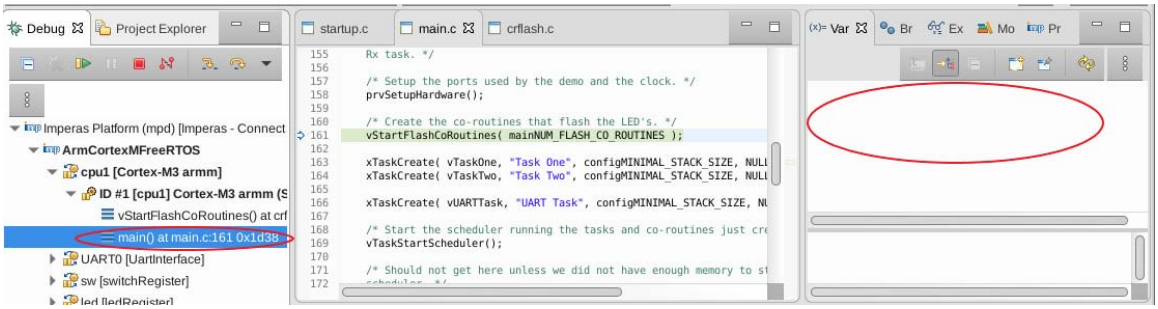
Then select the Variables view to see the values of the local variables defined in the function:



Note that the *uxIndex* value has a random value in it, as it has not been set yet in the function - press *step over* (or f6) several times until we see *uxIndex* initialized to 0 (when a variable changes it is highlighted in yellow in the Variables view):



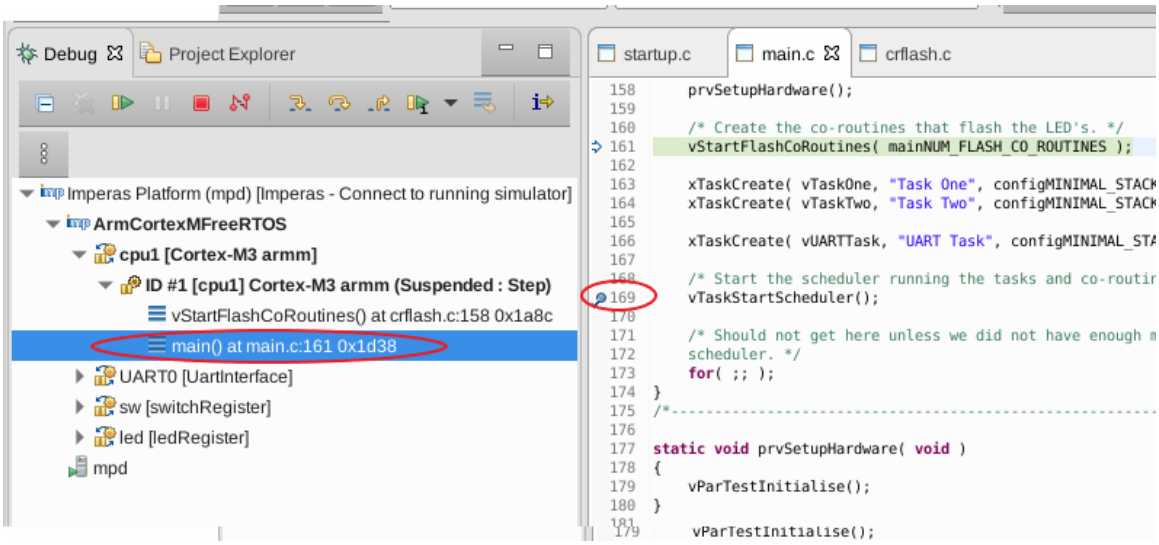
The Variables view shows the local variables of the stack entry *currently selected* in the Debug view. If we select a different stack element in the debug view the Variables view will be updated. For example if we select the *main()* function in the stack for cpu1 the Variables view gets updated (and in fact is empty, as there are no local variables defined in *main()*:



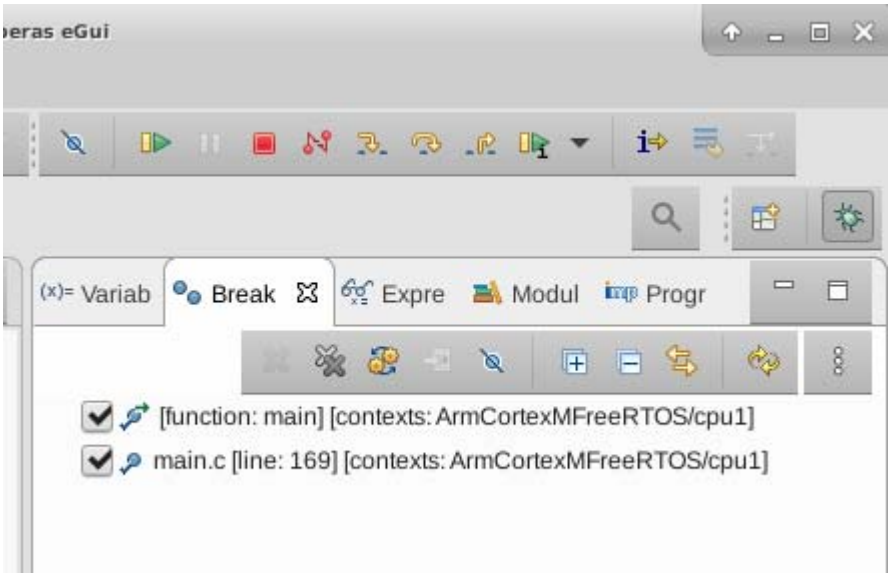
5.8 Adding breakpoints from the source window

Breakpoints may be added by double clicking on the left edge of the source line where the breakpoint is to be set.

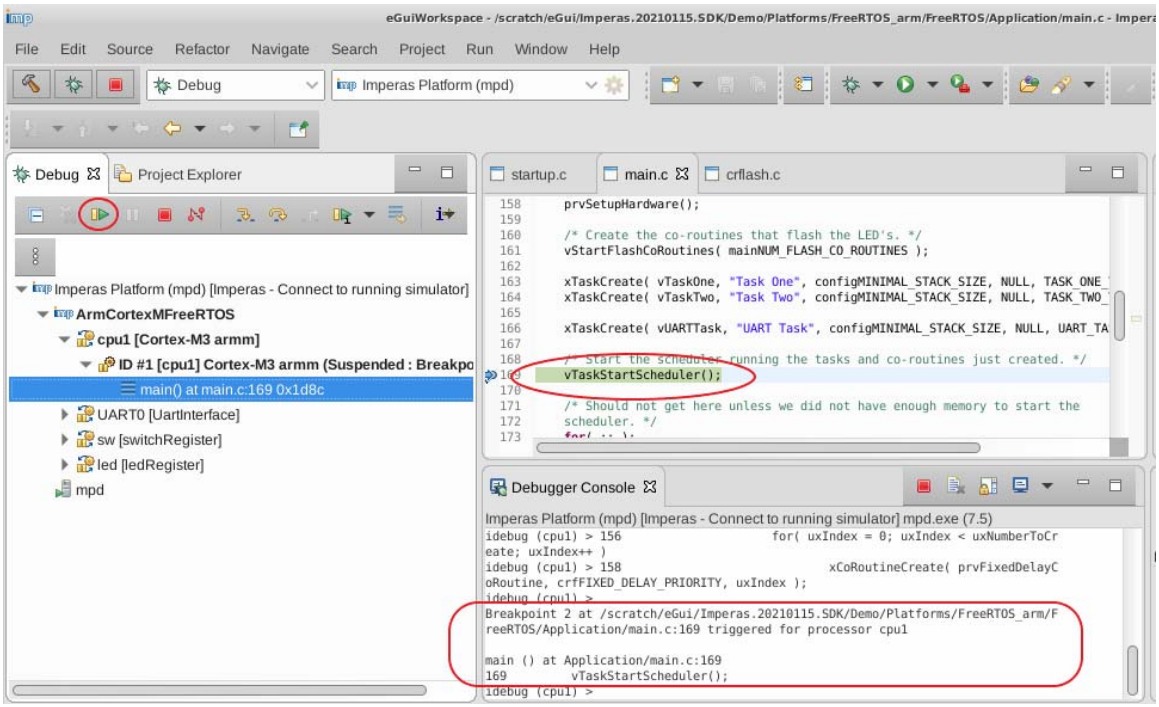
Here we will set a breakpoint in the main function on cpu1. First we change the focus to the *main.c* source file by selecting the appropriate line in the Debug view and then double click in the main.c source view to the left of line 169 to create the breakpoint:



If we now select the Breakpoints view we can see the new breakpoint listed, along with the breakpoint on function main that we set from the Debugger Console view:

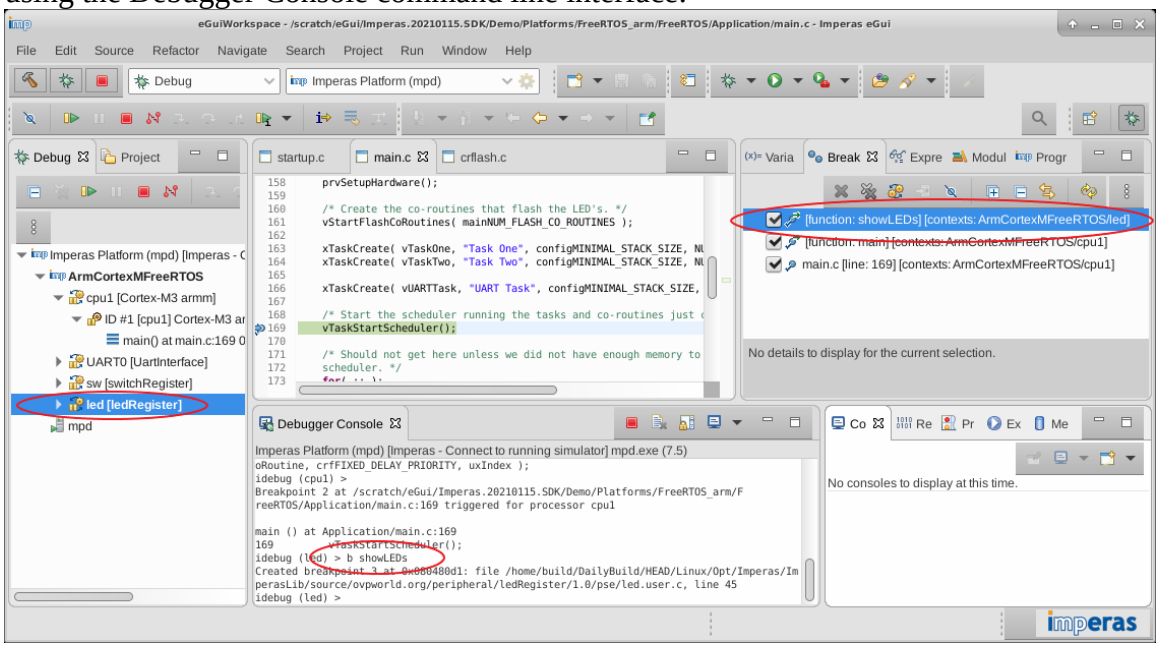


Now we press resume (or F8) to run the simulator to the breakpoint we just set and we will stop just before executing the line:

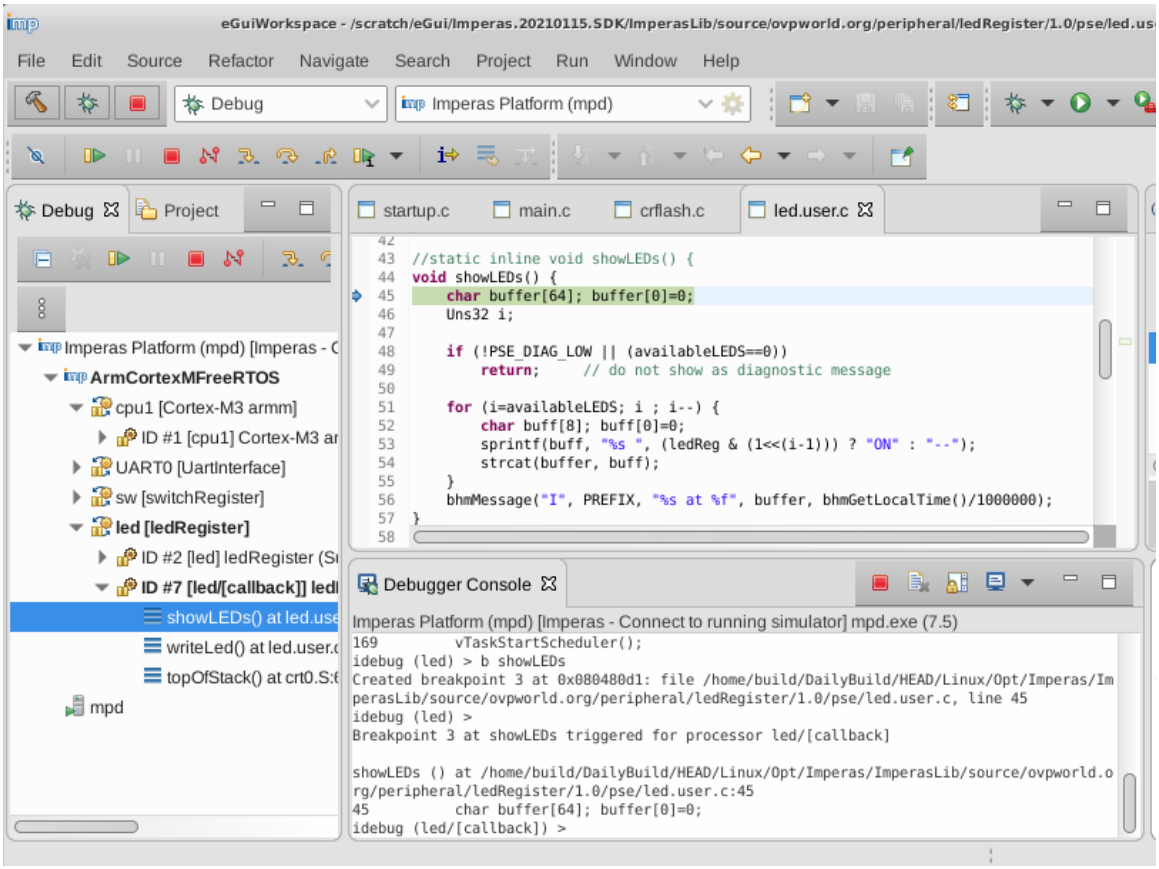


5.9 Examining context across the platform

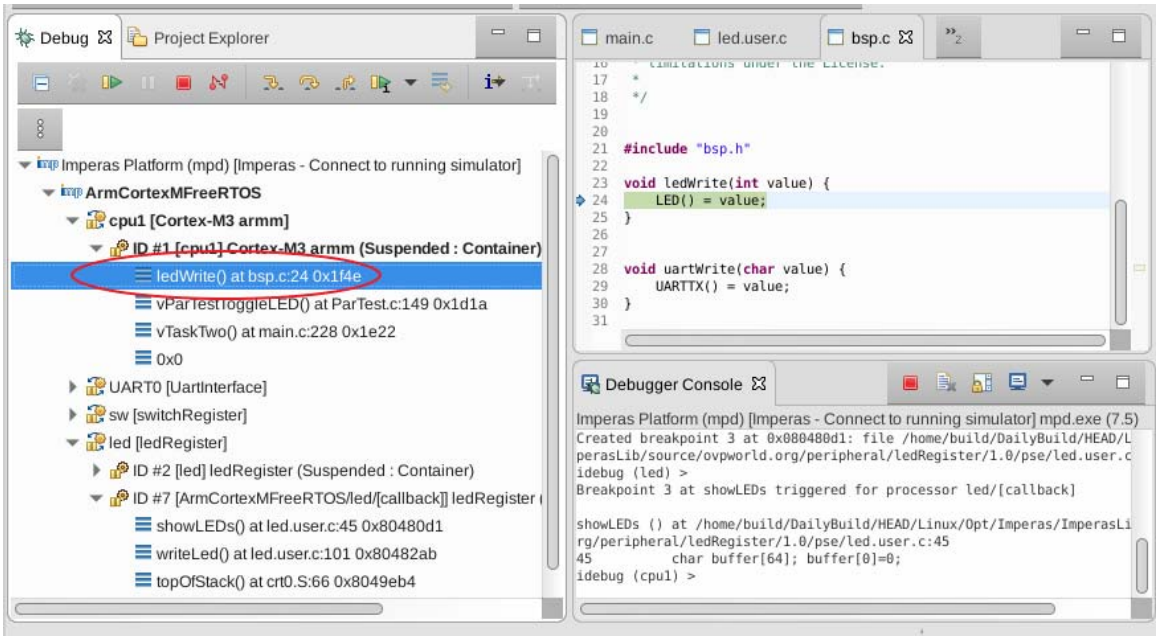
With MPD we can simultaneously debug all processor and peripheral models at the same time. To illustrate this we will set a breakpoint in the led peripheral, by selecting the led in the Debug view to change the focus and adding a breakpoint in function *showLEDs* using the Debugger Console command line interface:



Now if we resume execution we will hit the breakpoint when the program updates the LED registers:



At this point we can also examine the call stack in the application processor at the time the write to the peripheral register was made by opening the drop down list for *ArmCortexMFreeRTOS/cpu1* in the debug view:

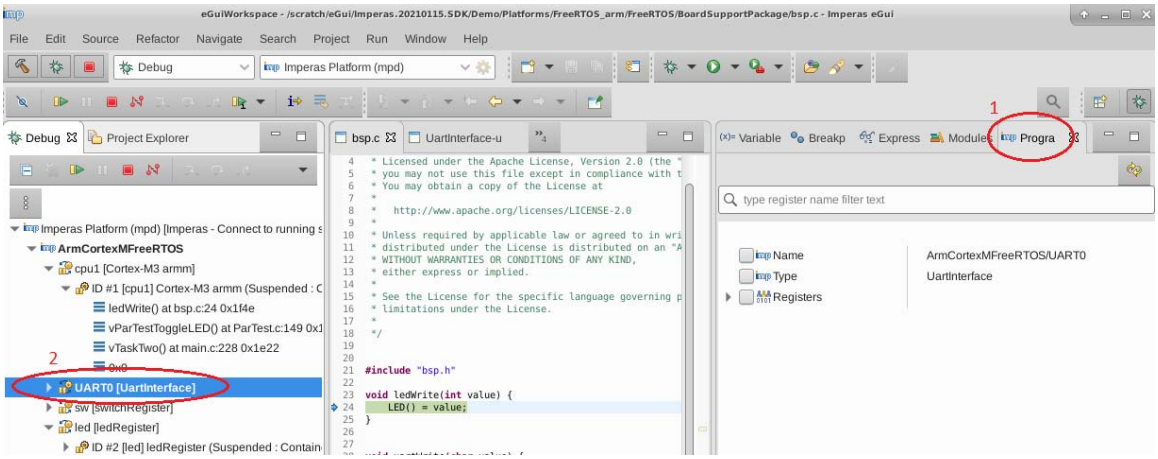


This allows us to identify the point in the application code where the peripheral register was accessed. This can be very useful when debugging complicated operating systems to determine when peripherals are being accessed.

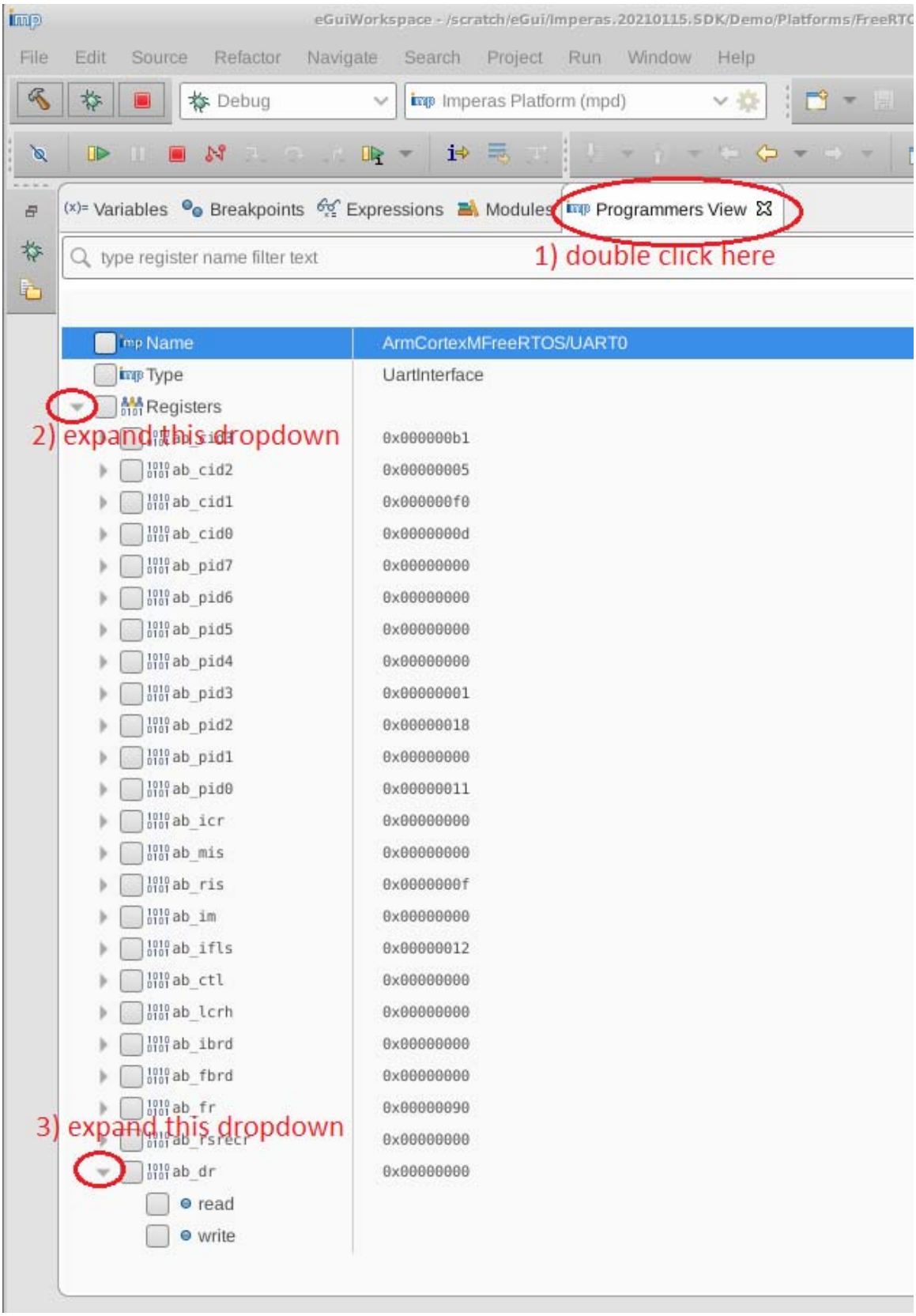
5.10 Eventpoints on reads/writes of peripheral registers

The Programmers View may be used to easily insert Eventpoints (these are like breakpoints but for Programmers View events). The *read* and/or *write* boxes on each register may be used to add or remove an eventpoint on the read or write of a register.

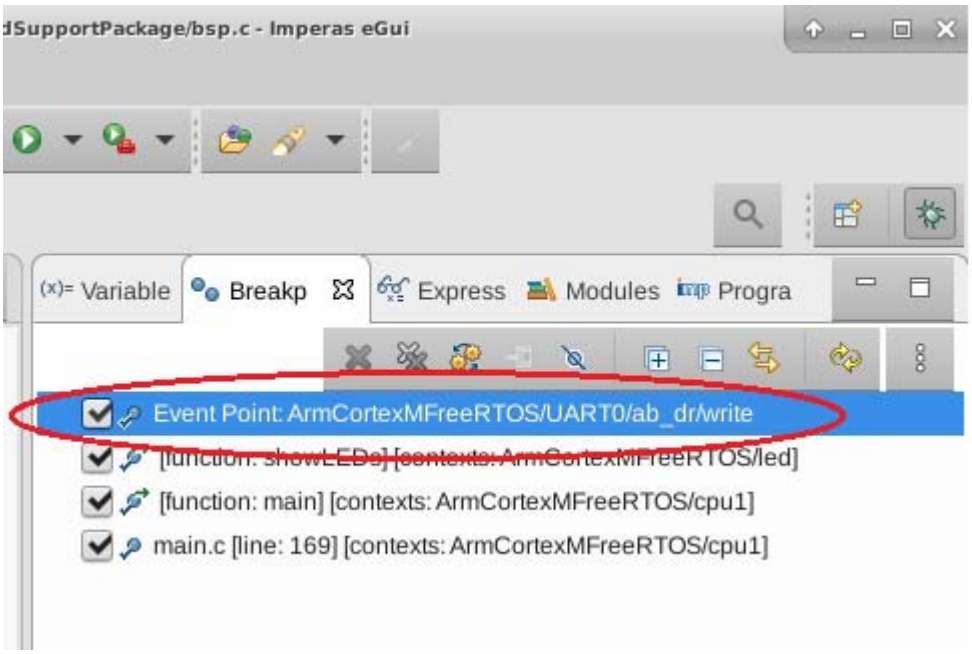
If we first select the Programmers View and then select the *ArmCortexMFreeRTOS/UART0* peripheral in the Debug View:



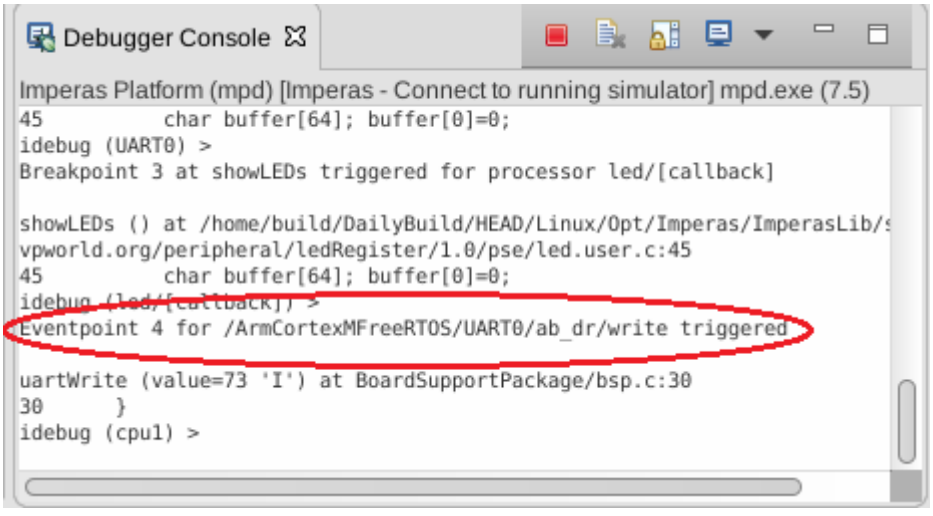
Next we want to expand the Programmers View to full size by double clicking on its tab and then expand the Registers drop down list, and also expand the drop down list for the *ab_dr* register:


eGuiWorkspace - /scratch/eGui/Imperas.20210115.SDK/Demo/Platforms/FreeRTO
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Imperas Platform (mpd)
Variables
Breakpoints
Expressions
Modules
Programmers View
type register name filter text
1) double click here
Name
ArmCortexMFreeRTOS/UART0
Type
UartInterface
Registers
2) expand this dropdown
0x000000b1
ab_cid2
0x00000005
ab_cid1
0x000000f0
ab_cid0
0x0000000d
ab_pid7
0x00000000
ab_pid6
0x00000000
ab_pid5
0x00000000
ab_pid4
0x00000000
ab_pid3
0x00000001
ab_pid2
0x00000018
ab_pid1
0x00000000
ab_pid0
0x00000011
ab_icr
0x00000000
ab_mis
0x00000000
ab_ris
0x0000000f
ab_im
0x00000000
ab_ifls
0x00000012
ab_ctl
0x00000000
ab_lcrh
0x00000000
ab_ibrd
0x00000000
ab_fbrd
0x00000000
ab_fr
0x00000090
0x00000000
3) expand this dropdown
ab_dr
0x00000000
read
write

If we click on the *write* box for the *ab_dr* register we will create an eventpoint on writes to the *ab_dr* register of UART0. (Note: you can double-click the Programmers View tab again to shrink it down from full size.) We can see this by selecting the Breakpoint View again:

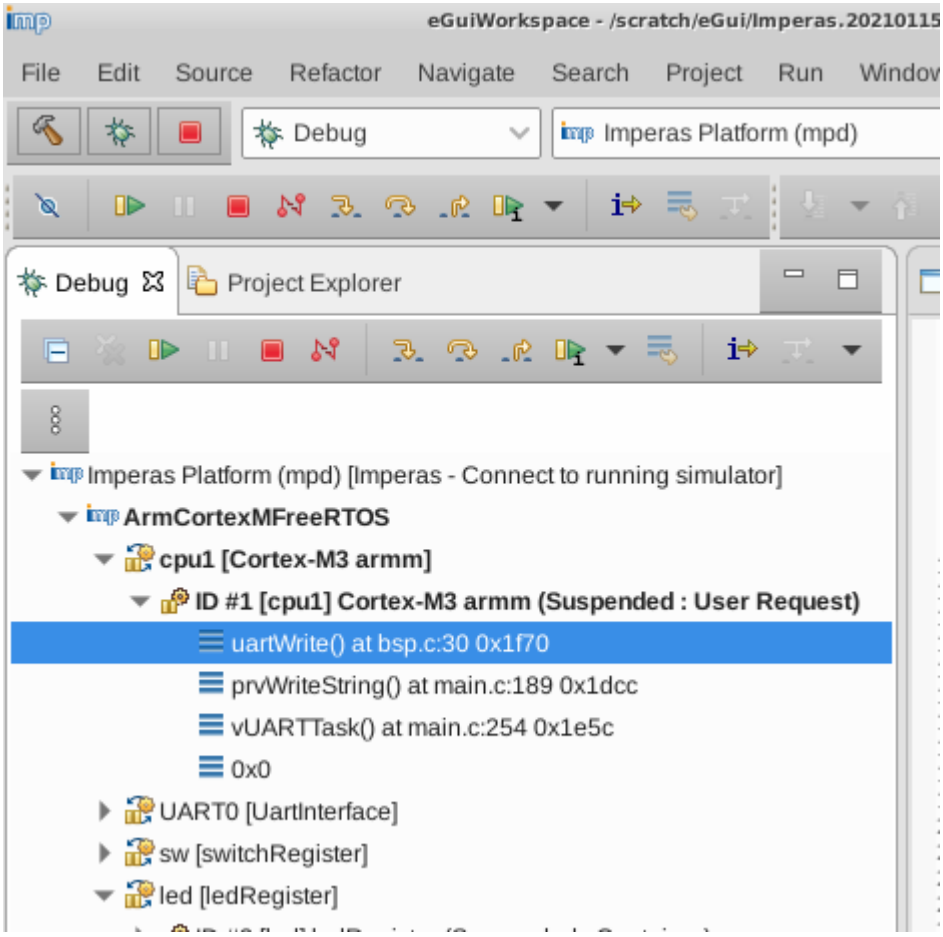


Now we can select Resume (or f8) until the Debugger Console window shows that the eventpoint is hit:



(You may hit the led.user.c breakpoint a few times before the eventpoint is hit - disabling the breakpoint by unchecking its box in the Breakpoints view can avoid this.)

Now if we look at the call stack for *ArmCortexMFreeRTOS/cpu1* again we can again see where the write to that register has occurred:



6 A Sample Debug Session Using GDB

6.1 Prerequisites

To follow this example you will need the following Imperas and OVP packages installed:

- OVPsim Imperas_DEV
- eGui_Eclipse

The Imperas environment must be set up according to the directions in *Imperas Installation and Getting Started Guide*.

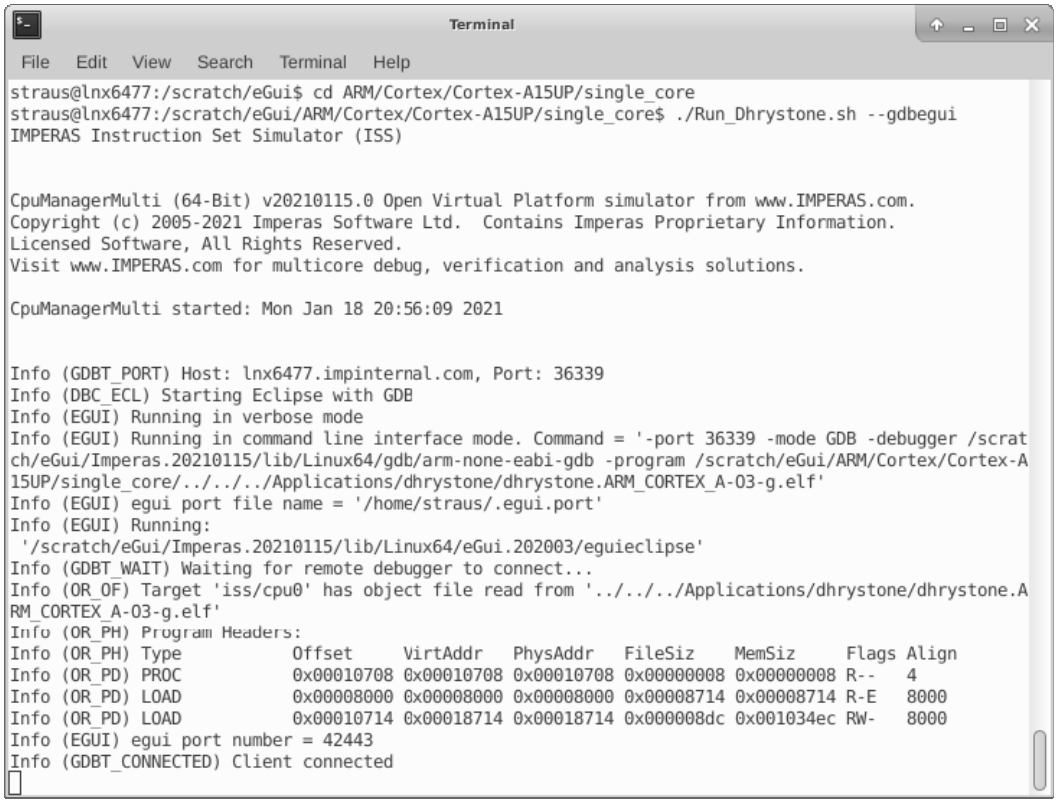
The installation includes a number of demos of processors running simple applications. These demos may be easily run in standalone debugging sessions by running with additional debug options specified.

6.2 Starting the debug session

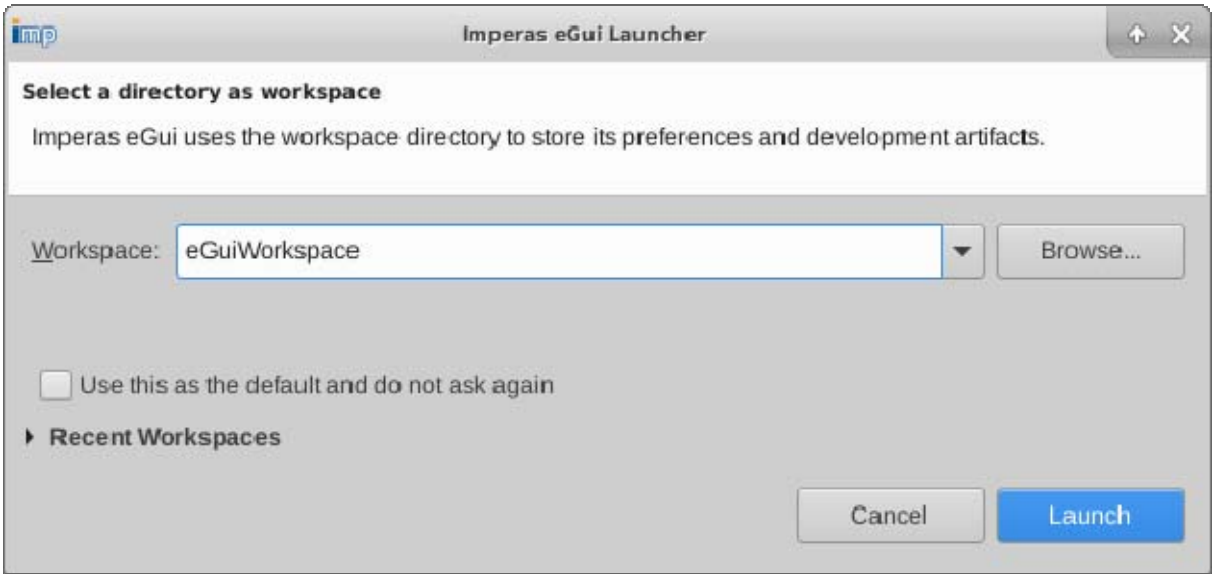
First we must change to the demo directory and start the demo in GDB mode. From a Linux shell or a Windows MSys shell:

```
cp -r $IMPERAS_HOME/Demo/Processors/ARM .
cd ARM/Cortex/Cortex-A15UP/single_core
./Run_Dhrystone.sh --gdbegui
```

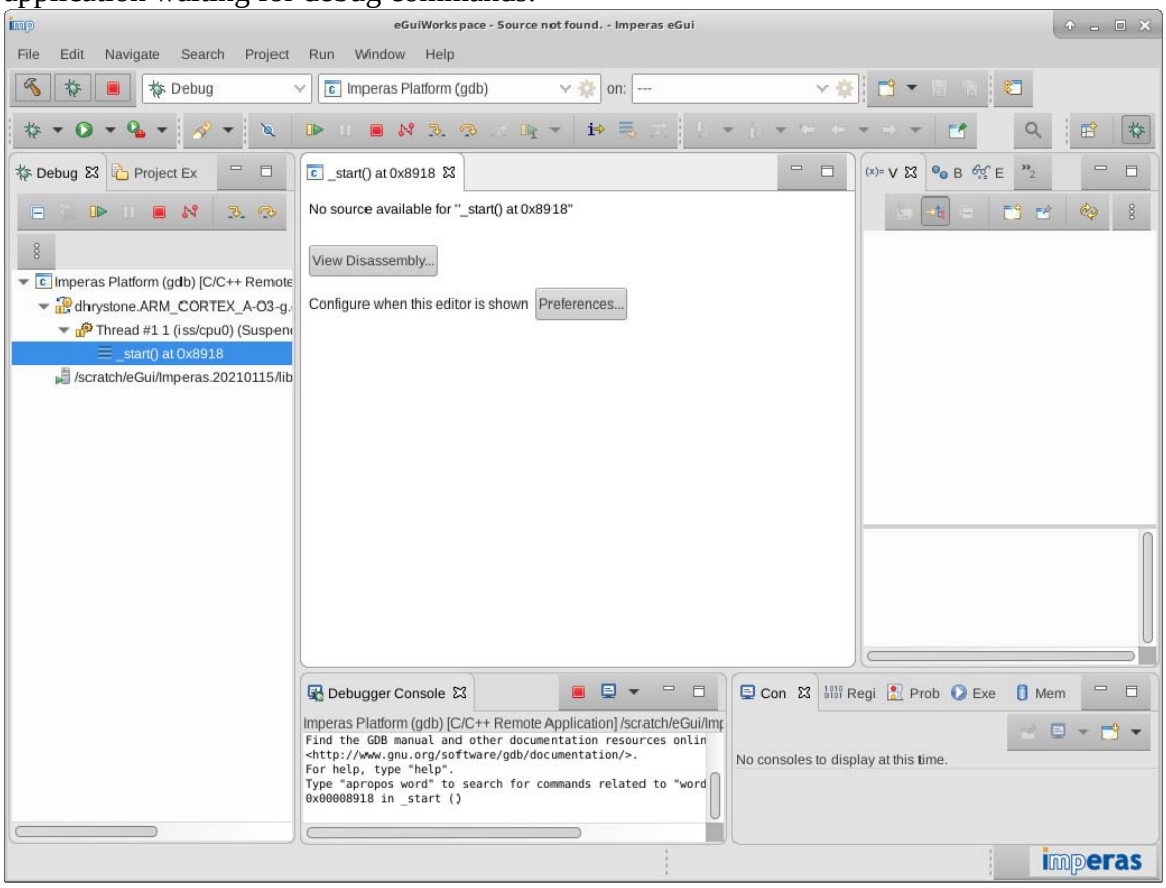
The shell from which the script was invoked will show the connection messages:



Eclipse will prompt you for a workspace. Since we are just running a standalone debugging session we can create a new workspace in the current directory:



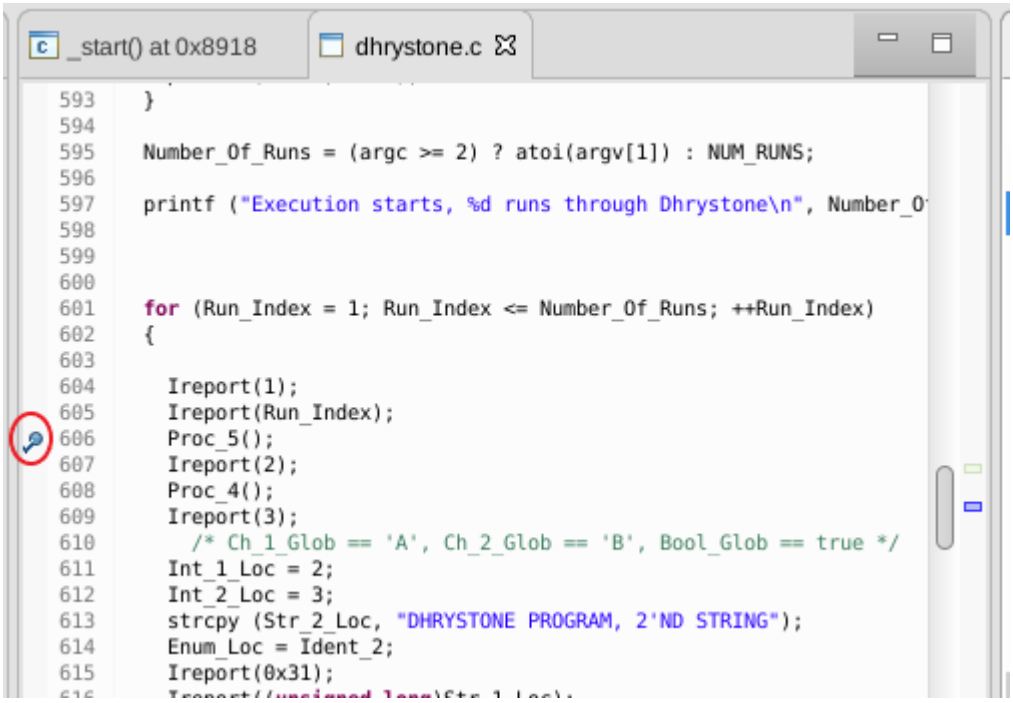
Once started Eclipse will be in the debug perspective stopped at main in the Dhrystone application waiting for debug commands:



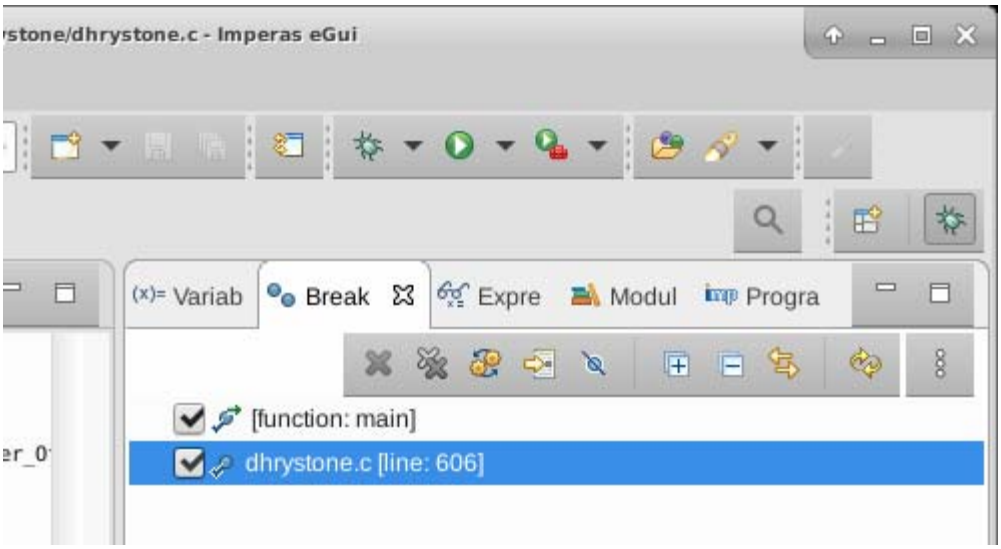
Now we can debug the program normally using the standard Eclipse features as described in the following section.

6.3 Example GDB Debug Session for Dhrystone Benchmark application

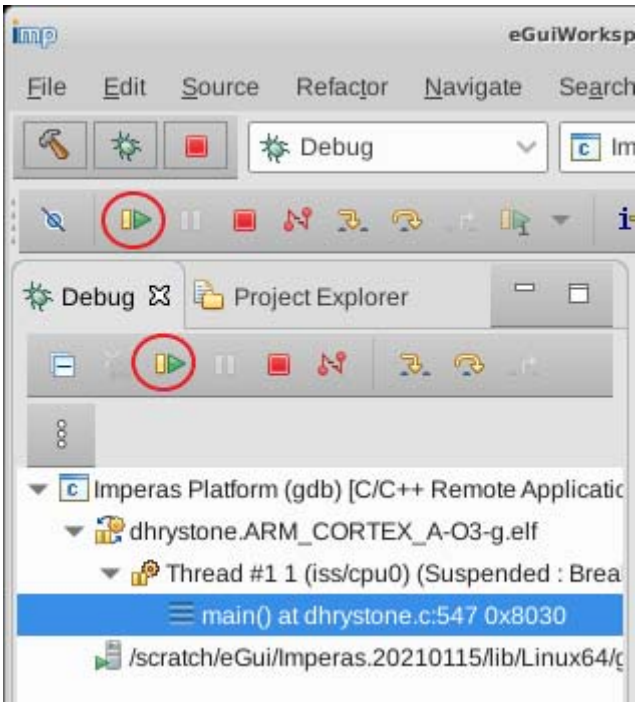
1. In the Debugger Console view (bottom middle) enter the gdb command *break main* and *continue* to advance to the entry of function *main()*.
2. Scroll down in the source code window to the line containing the call to 'Proc_5();' and set a breakpoint on that line by double clicking beside it:



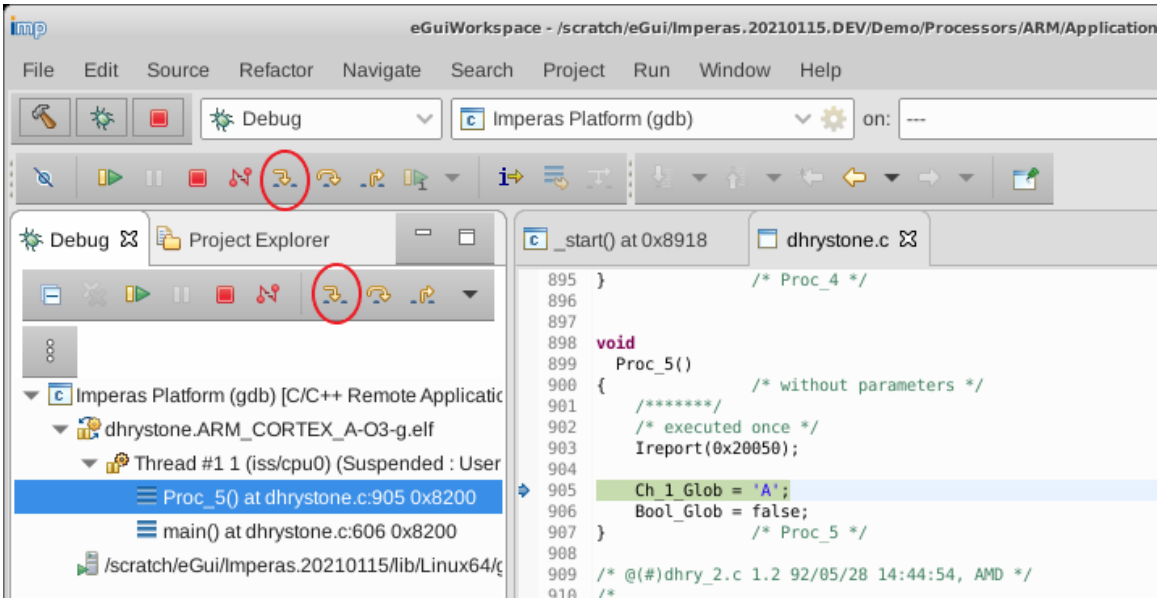
Once set the breakpoint appears in the Breakpoints window:



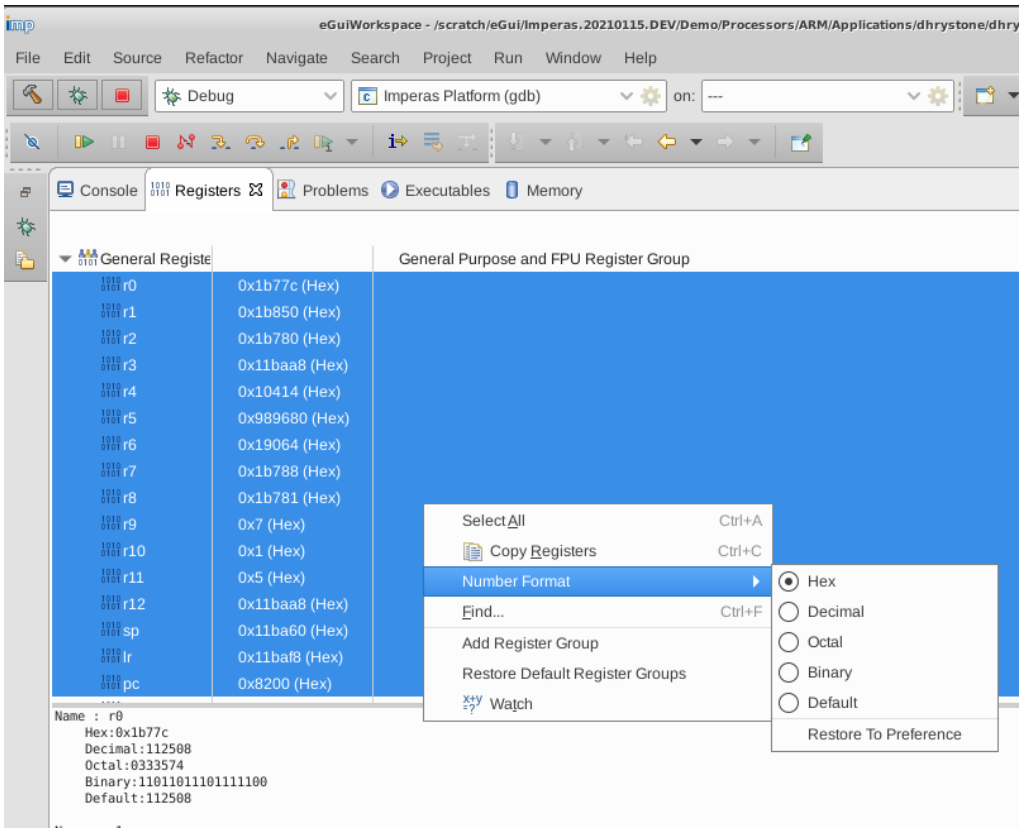
3. Continue the simulation by selecting either of the green 'Resume (F8)' (one in the main toolbar and one in the Debug View icons)



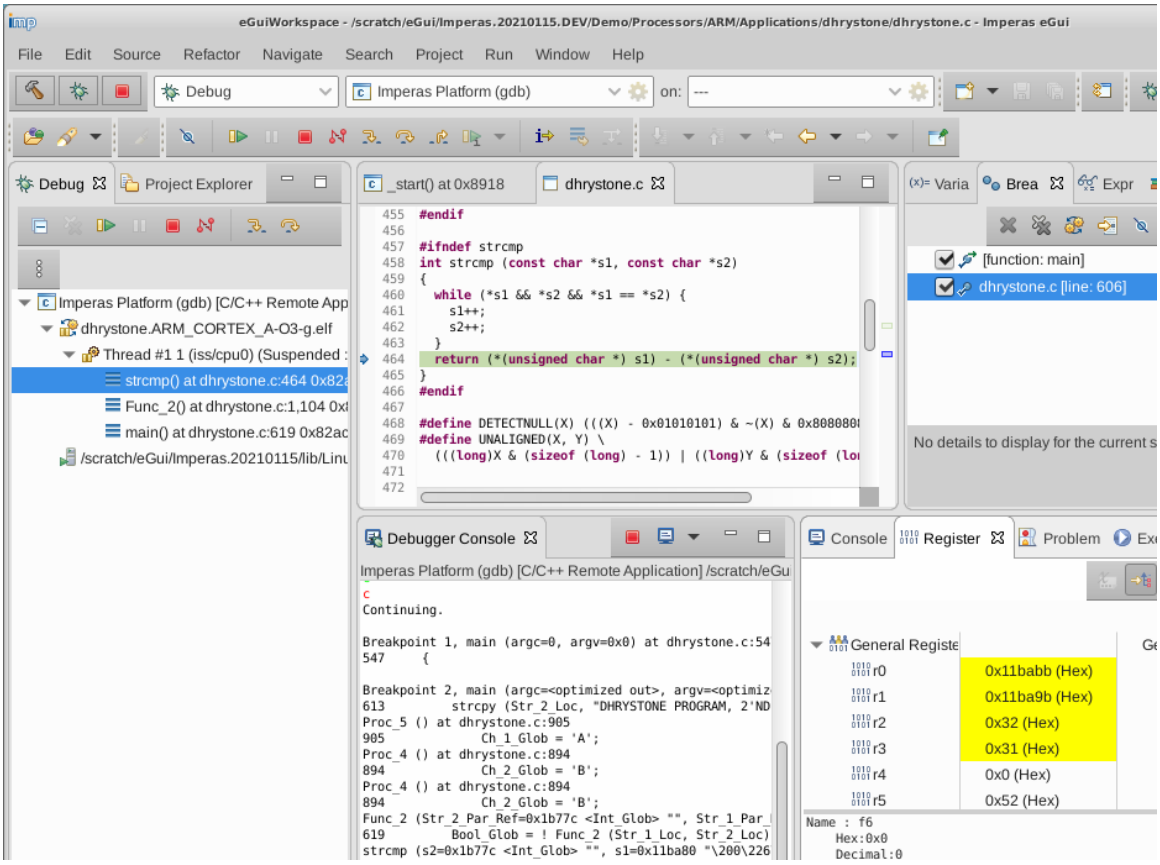
4. After the simulation hits the breakpoint, step the application forward by selecting either of the 'Step Into' icons:



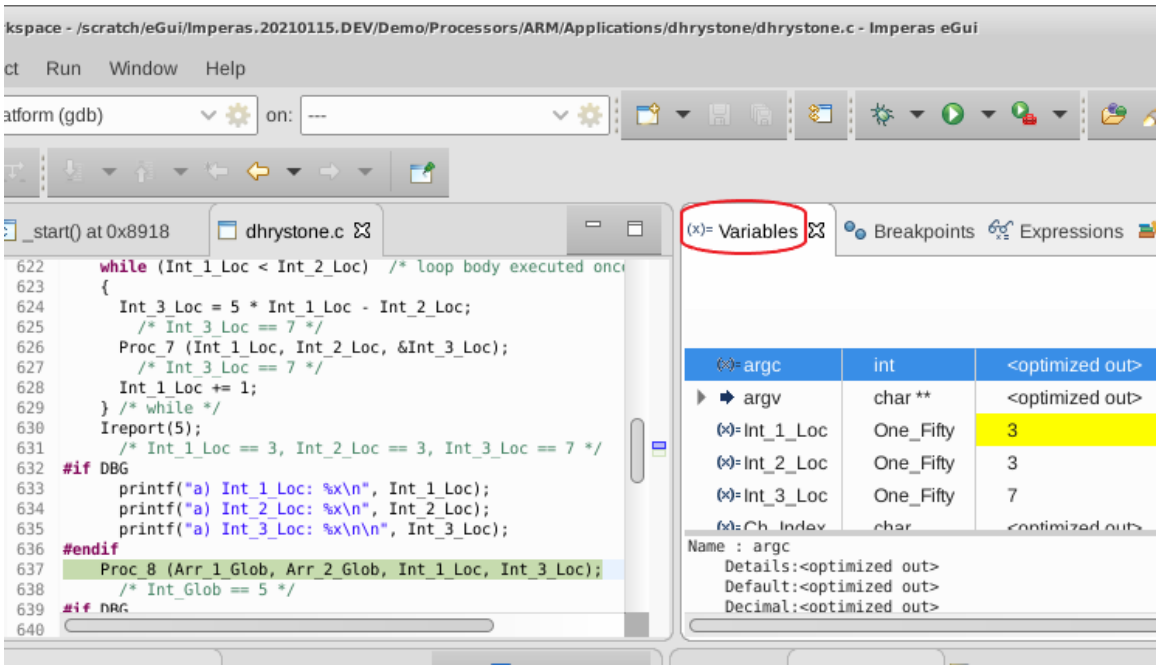
5. Select the register view; maximize the Registers window by double clicking on the Registers tab; expand the 'General Registers' drop down list; select registers r0 through r12; Right click on the selected registers and set the format to hexadecimal



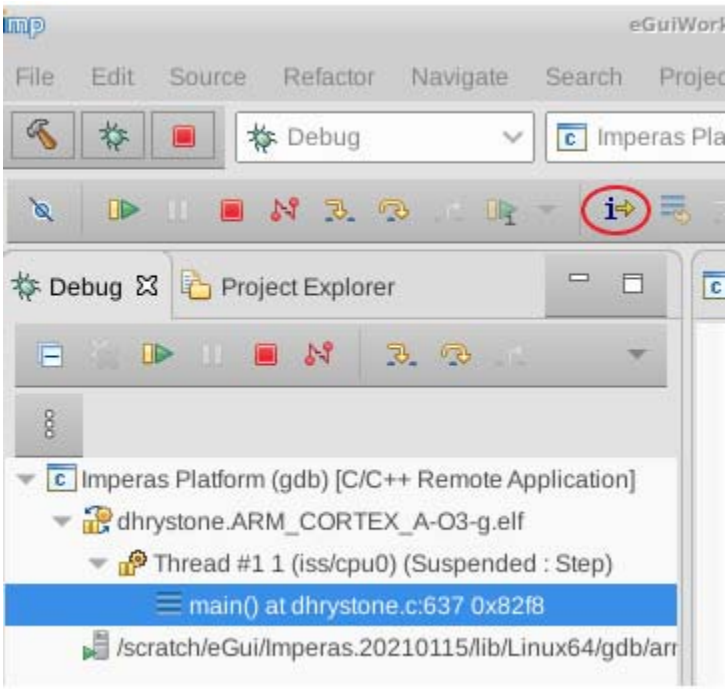
6. Minimize the Registers window by double clicking on the Registers tab; step the application forward using the 'Step Into' button a few times. This will show you the colored highlighting of the registers that change



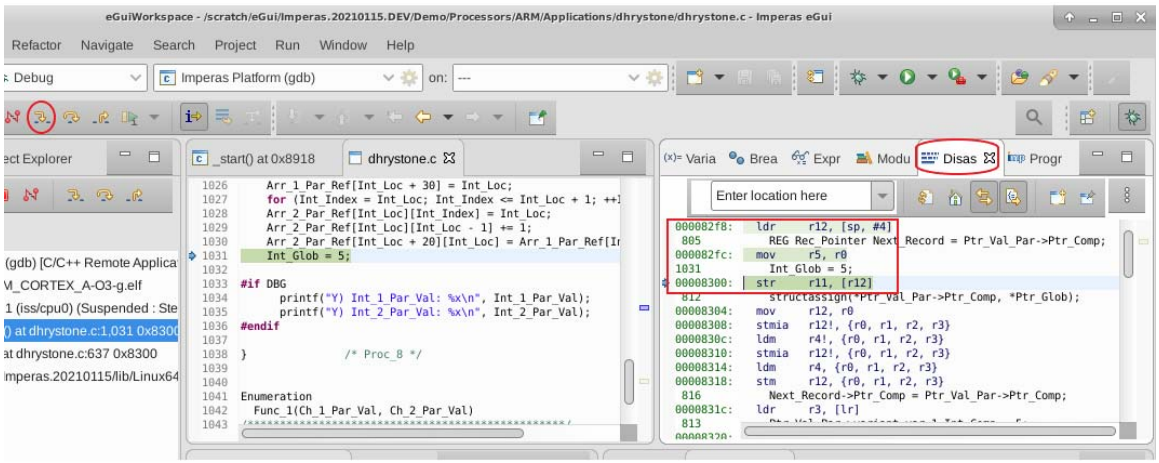
7. Open the Variables view; showing the current local variables. Note some variables are shown as '<optimized out>'. This is because we are debugging an executable with compiler optimizations enabled. Compiling with optimizations turned off, e.g. with the gcc option -O0, will allow all the variable values to be visible.



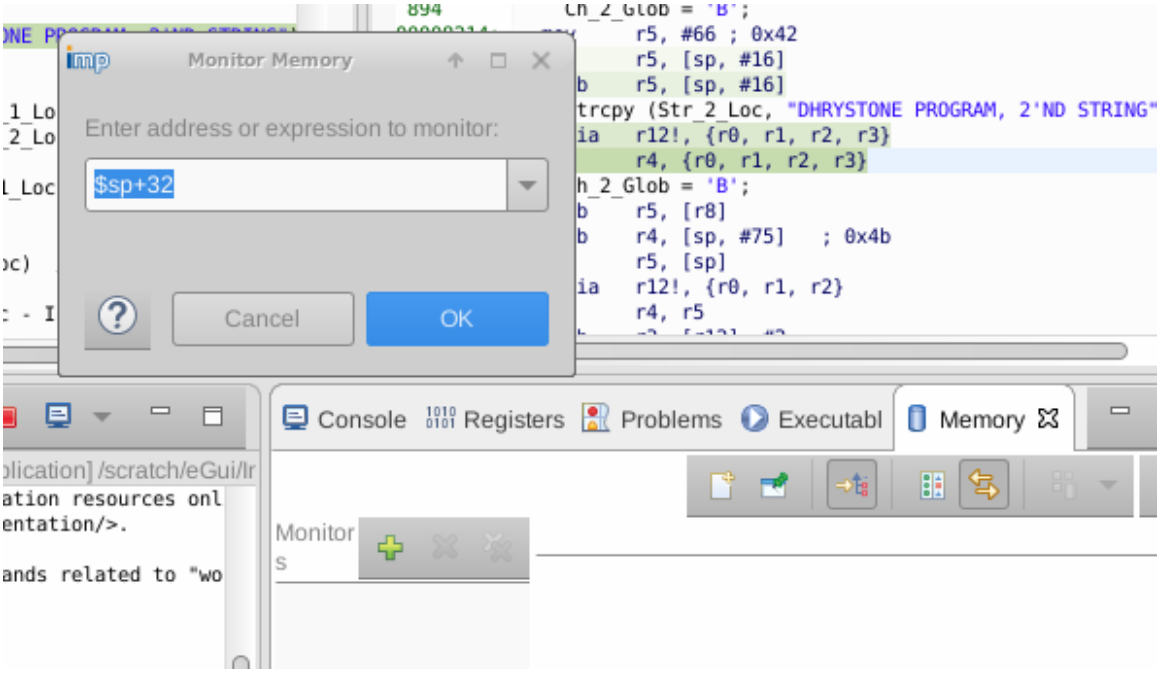
8. Select 'Instruction Stepping Mode' using the button.



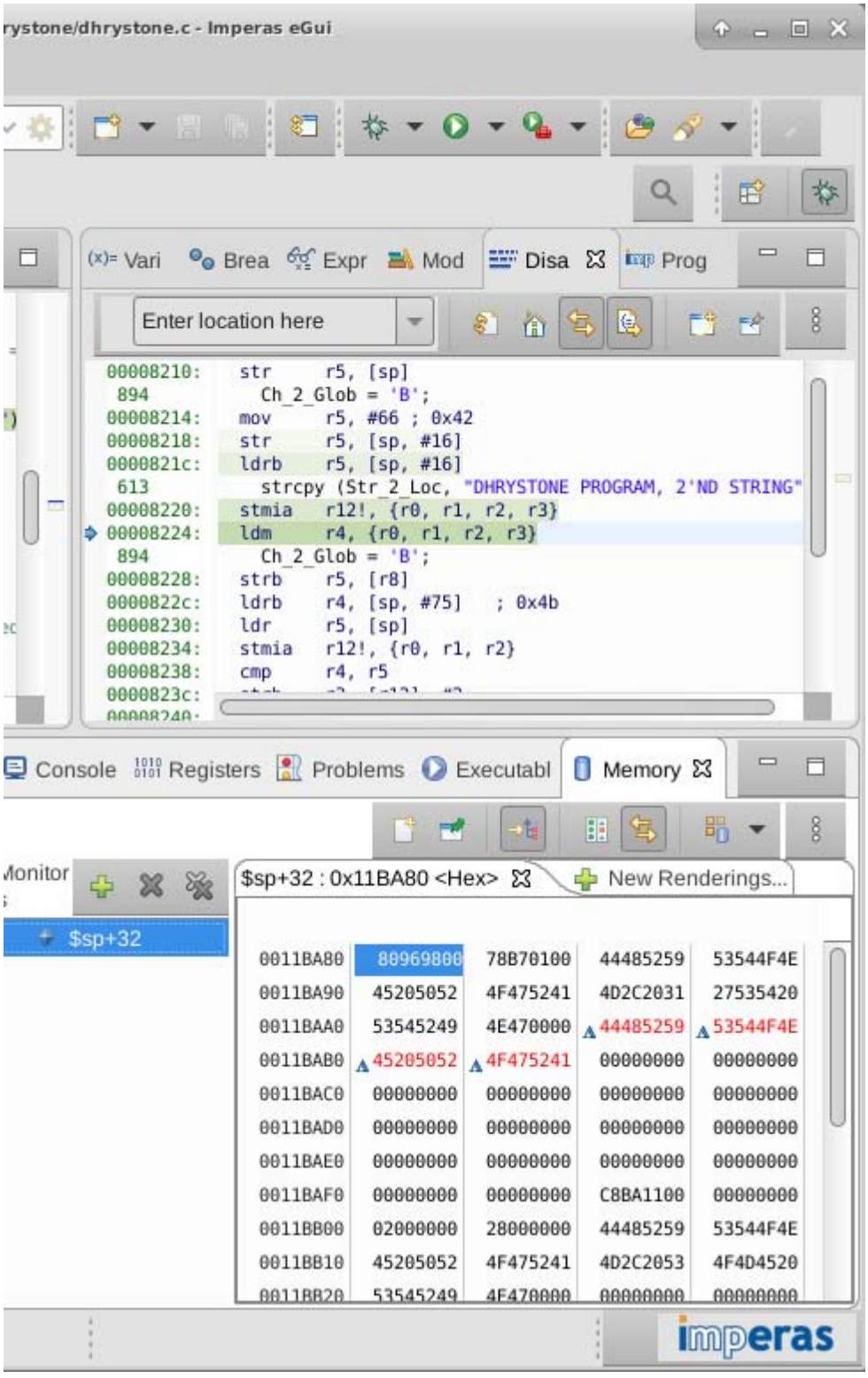
9. Step the application forward using the 'Step Into' button. This will open the disassembly view and highlight the instructions as they are stepped.



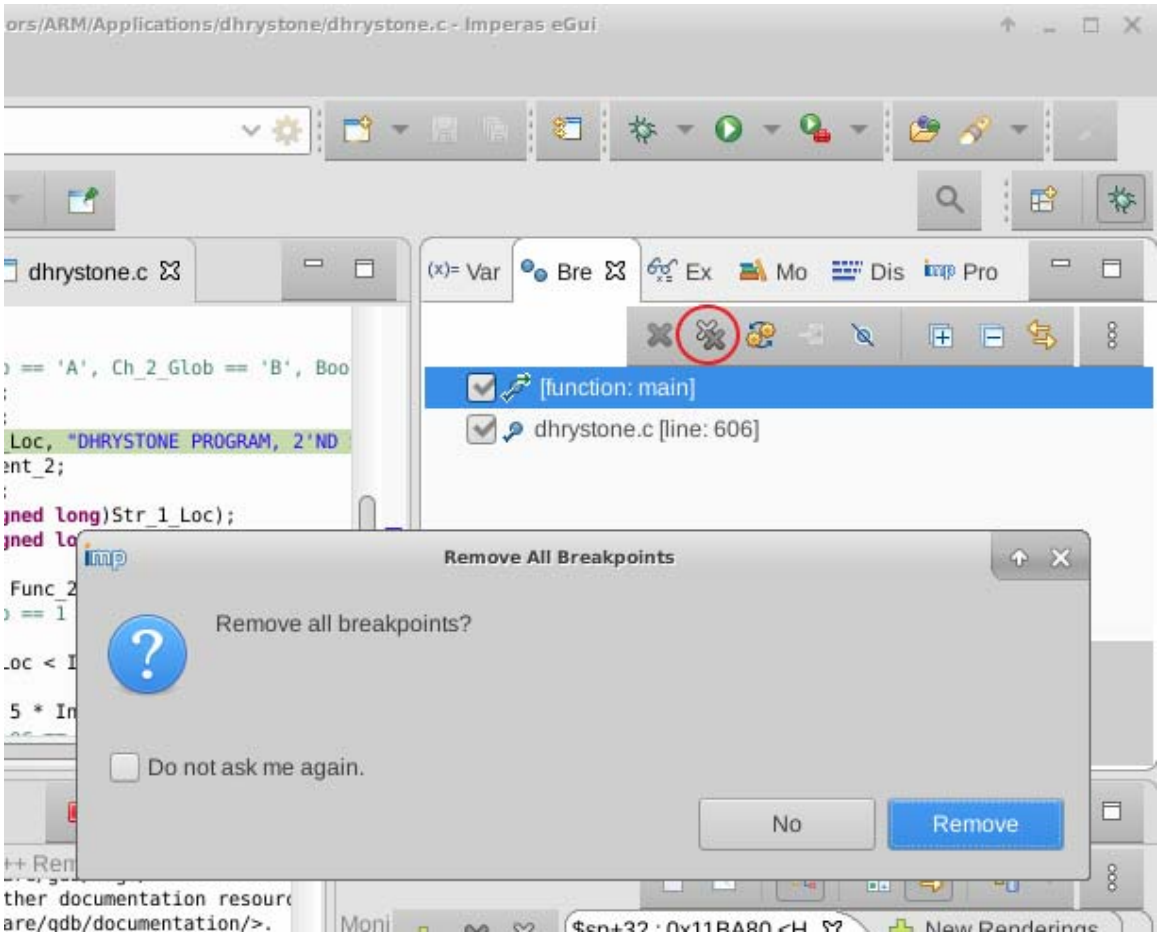
10. In the 'Memory' window, select 'Add Memory Monitor' (The plus symbol) and enter $sp+32 to select the value of the stack memory to display in the memory window:



11. Move the application forward using the 'Step Into' button. Memory locations that are modified will be highlighted.

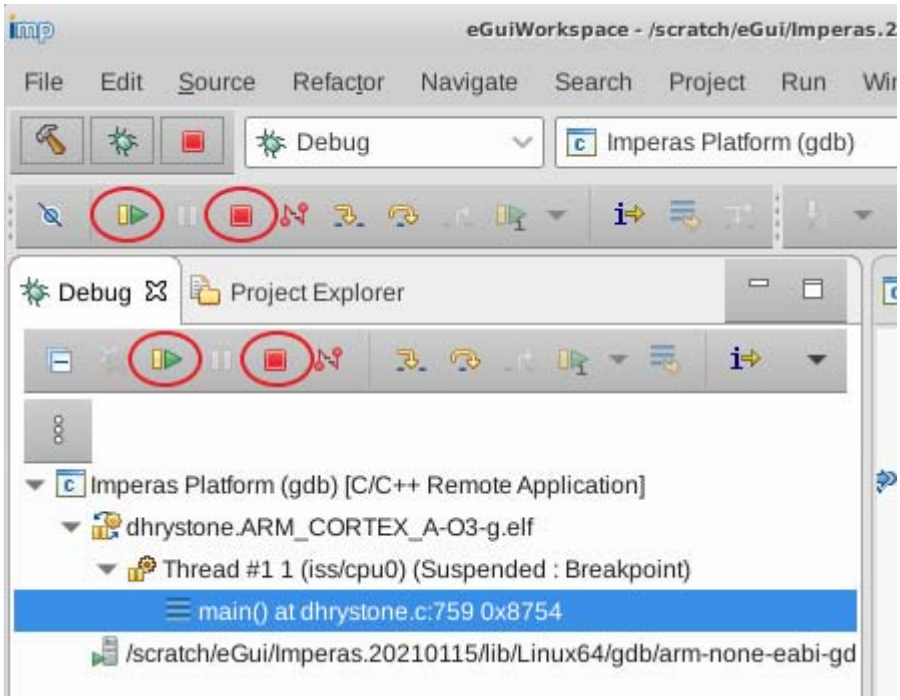


12. Remove all breakpoints by selecting the double gray X icon in the Breakpoints view and selecting Remove to confirm:



13. To finish the simulation you may either:
    a. Use either of the 'Resume' icons (the green play arrows) to continue to next breakpoint or the end of simulation.
    b. Use either of the 'Terminate' icons (the red square icon) to terminate the simulation without continuing

    After either of these the debug perspective can be cleaned with the 'Remove All terminated Launches' button (the two gray x's in the debug view)


eGuiWorkspace - /scratch/eGui/Imperas.2
File
Edit
Source
Refactor
Navigate
Search
Project
Run
Debug
Imperas Platform (gdb)
Debug
Project Explorer
Imperas Platform (gdb) [C/C++ Remote Application]
dhrystone.ARM_CORTEX_A-O3-g.elf
Thread #1 1 (iss/cpu0) (Suspended : Breakpoint)
main() at dhrystone.c:759 0x8754
/scratch/eGui/Imperas.20210115/lib/Linux64/gdb/arm-none-eabi-gd

###