Debugging EMAC OE SDK Applications Using Eclipse

The EMAC Eclipse distribution is pre-configured with remote debug support provided by the Debug Perspective. This eliminates the need to learn GDB's command-line when a program needs to be debugged. Remote debug support is facilitated by `gdbserver` on the remote machine. The following guide will provide an introduction to the use of the Eclipse Debug Perspective to step through code in a multi-threaded C/C++ application compiled for an EMAC OE product. For a general introduction to debugging and to gain a better understanding of GDB, refer to the EMAC OE SDK Debug Guide.

Introduction

This Eclipse Guide will use the `pthread_demo` EMAC OE SDK example project to provide instructions for the following debug tasks:

1. Debug Configuration Setup
2. Start a Debug Session
3. Add Breakpoints
4. Switch Between Threads
5. Enable Scheduler-Locking Using the GDB Console
6. Step Through Code
7. Other Debug Tools

These lessons assume that the user has just finished editing a C source file called `pthread_demo.c` from the EMAC OE SDK example projects. They expose the basic debugging capability provided by the Eclipse Debug perspective, but are not intended to be a comprehensive guide to debugging in Eclipse. More information is provided by the C/C++ Development Tools plugin through a help menu. To access CDT debugging help, do the following:

1. Click *Help → Help Contents*.
2. In the *Help → Eclipse* dialog window, select *C/C++ Development User Guide → Tasks → Running and debugging projects → Debugging*.

Debug Configuration Setup

Before switching to the Eclipse Debug Perspective, it is necessary to set up a Debug configuration that specifies the EMAC OE SDK version of GDB as the debugger. This is important because the remote target's CPU architecture must be taken into account by the debugger since each EMAC OE SDK is configured specifically for its particular target board.

The following procedure uses field values which are specific to this particular debug session. Because of this, it should be treated as an example of how to set up a Debug Configuration rather than a general guide on doing so.

1. Create a new launch configuration.
    1. In the Project Explorer View, right-click the project to be run.
    2. Select *Debug As → Debug Configurations...* to bring up the Debug Configuration Dialog.
    3. In the *Type List*, right-click *C/C++ Remote Application* and select *New...* from the context menu. A new configuration with the same name as the currently open project will appear. This name can be changed in the top leftmost text field labeled *Name:*. The Figures below use the name `pthread_demo`.
2. For instructions relating to fields in the main tab, refer to Figure 1.

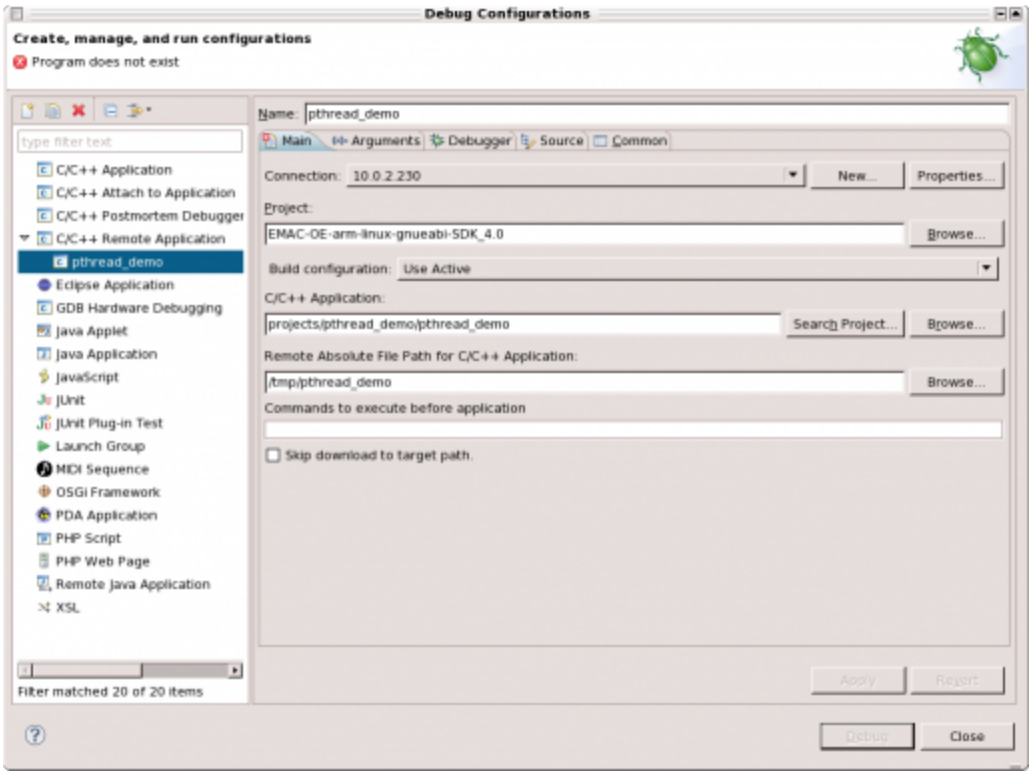

Figure 1. Debug Configuration Main Tab

1. Choose a connection from the *Connection:* drop-down or click *New...* to create a new one.
2. The default for the *Project:* field will be the currently-active project. If this is not the EMAC SDK project or if your source code is contained in another project, click *Browse...* to choose the correct one.
3. Choose a value for *C/C++ Application:* using one of the methods listed below:
   - Type `projects/pthread_demo/pthread_demo` in the text field provided. Note that this text field takes the path relative to a project's root directory.
   - Use *Search Project...* to initiate a search dialog. Type `pthread_demo` to search for the application binary.
   - Manually browse the file system to find `pthread_demo` using *Browse...*.
4. Choose a value for *Remote Absolute File Path for C/C++ Application:*. When uploading the application using the EMAC SDK Makefile, this location is `/tmp/pthread_demo`. This can be done one of two ways:
   - Type the location in the provided text field
   - Select *Browse...* to search the remote file system for the target binary.
     Note that this step requires the value of *Connection:* field to be set to a valid connection and the development machine to be connected to the target machine.

3. For instructions relating to the Arguments tab, refer to Figure 2.

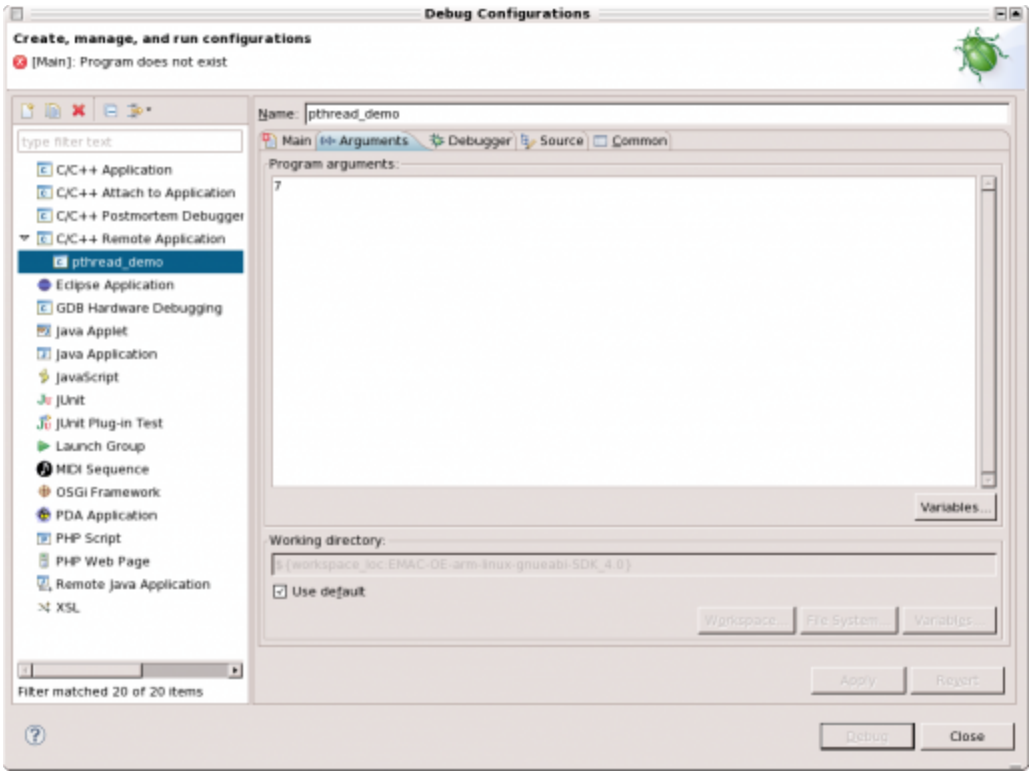

Figure 2. Debug Configuration Arguments Tab

If `app_name` takes any command-line arguments, this is where they must be entered. For this debug session the program takes a single integer argument as if the following command were run in a remote shell:

```
$ /tmp/pthread_demo 7
```

To achieve the same effect in the Debug Configuration, the *Program arguments:* field must contain the value '7'.

4. For instructions relating to the Debugger tab, refer to Figure 3.

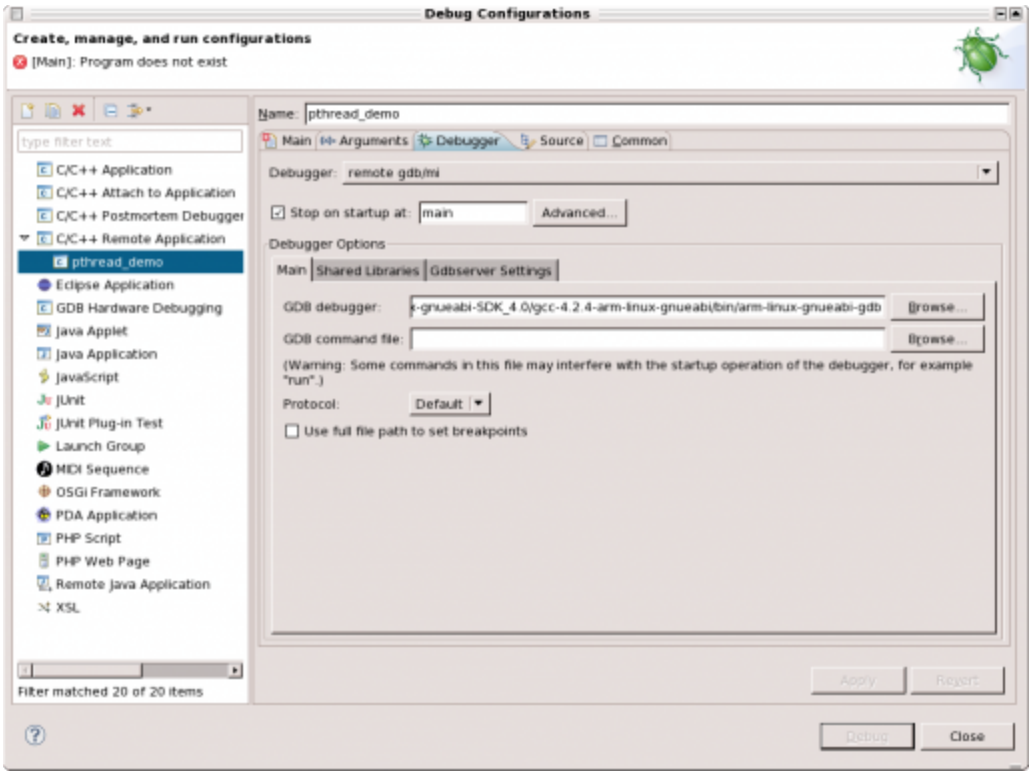

Figure 3. Debug Configuration Debugger Tab

1. Select `remote gdb/mi` from the *Debugger:* drop-down menu.
2. In the *Debugger Options* frame under the *Main* tab, change the *GDB Debugger:* to the following `/path/to/sdk/gcc-4.2.4-arm-linux-gnueabi/bin/arm-linux-gnueabi-gdb`.
   Note that this value may be different in SDKs created for different architectures.
3. Delete the default *GDB command file:* value as this will not be used during the debug session.

5. Click *Apply* when the above changes have been implemented successfully.

It is also recommend that line numbers are enabled for the default Eclipse text editor. This is helpful for Lesson 2 which will indicate breakpoints using line numbers. To enable this feature, perform the following procedure starting from an open Eclipse workspace:

1. Click *Window* → *Preferences*.
2. Select *General* → *Editors* → *Text Editors*. This will cause the Text Editors Preferences page to display on the right portion of the *Preferences* dialog window.
3. Check *Show line numbers*.

This setup prepares Eclipse for the debug session described in Lessons 1 through 5.

Walkthrough

The following lessons are intended to be run sequentially. Together they demonstrate one possible work flow for using the Eclipse Debug Perspective.

Lesson 1: Start a Debug Session

From the Eclipse Workspace:

1. Click *Run* → *Debug Configurations...*
2. Select *C/C++ Remote Application* → *pthread_demo* from the filter in the *Debug Configurations* dialog window.
3. Click *Debug*.

Depending on the project settings, Eclipse may attempt to perform a build as its first step. Otherwise, it will continue to the next step which is to upload the binary to the location in the remote file system specified in the Debug Configuration. It will then run that program attached to `gdbserver`. Eclipse will then run the SDK's architecture-specific version of GDB with the remote target specified by the hostname in the Remote Connection and the port specified in the Debug Configuration. Once all this is done, Eclipse will activate a prompt to determine whether or not it should automatically switch to the Debug Perspective. Answer *yes* to arrive at the workspace shown in Figure 4 below.

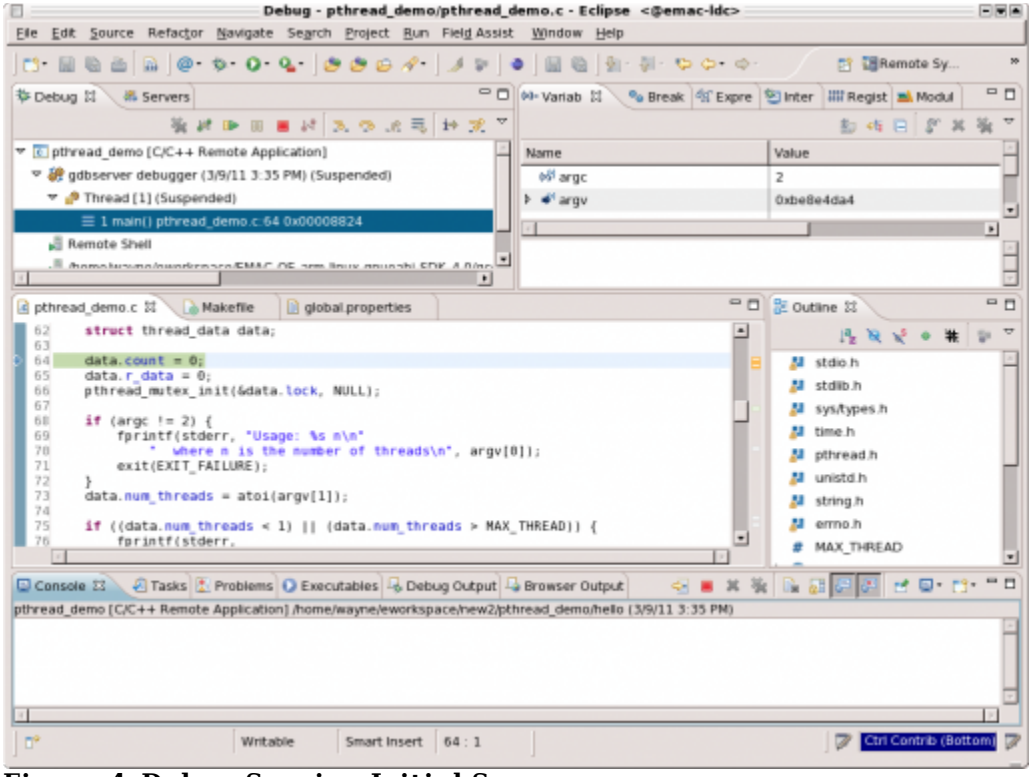

Figure 4. Debug Session Initial Screen

Lesson 2: Add Breakpoints

After starting the debug session, the next step is to add breakpoints. The procedure below covers this topic.

1. In the upper-right frame of Eclipse Views, the Variable View should be showing. Click the Breakpoints View tab to bring it into focus in that frame.
2. Click the text editor showing the `pthread_demo.c` source file to put it in focus.
3. Scroll to line 129.
4. Right-click on the number 129 in the line number column.
5. Select *Toggle Breakpoint* to toggle a breakpoint for that line. This causes a check mark to appear to the left of the line number as shown in Figure 5. Note that the same effect can be achieved by double-clicking the line number. Also notice that the Debug Perspective C/C++ Editor is the same editor used within the C/C++ Perspective. Because the editors are the same, the functionality of toggling breakpoints is available in the C/C++ Perspective as well as the Debug Perspective.
6. Repeat steps 3-5 for lines 119 and 165. Notice that each breakpoint shows up in the Breakpoints view. Right-clicking on either of these breakpoints will pull up a context menu that provides control similar to right-clicking on the breakpoint's line number. These breakpoints are local to the project being debugged so they will persist across debug sessions until removed manually in the Breakpoints View.

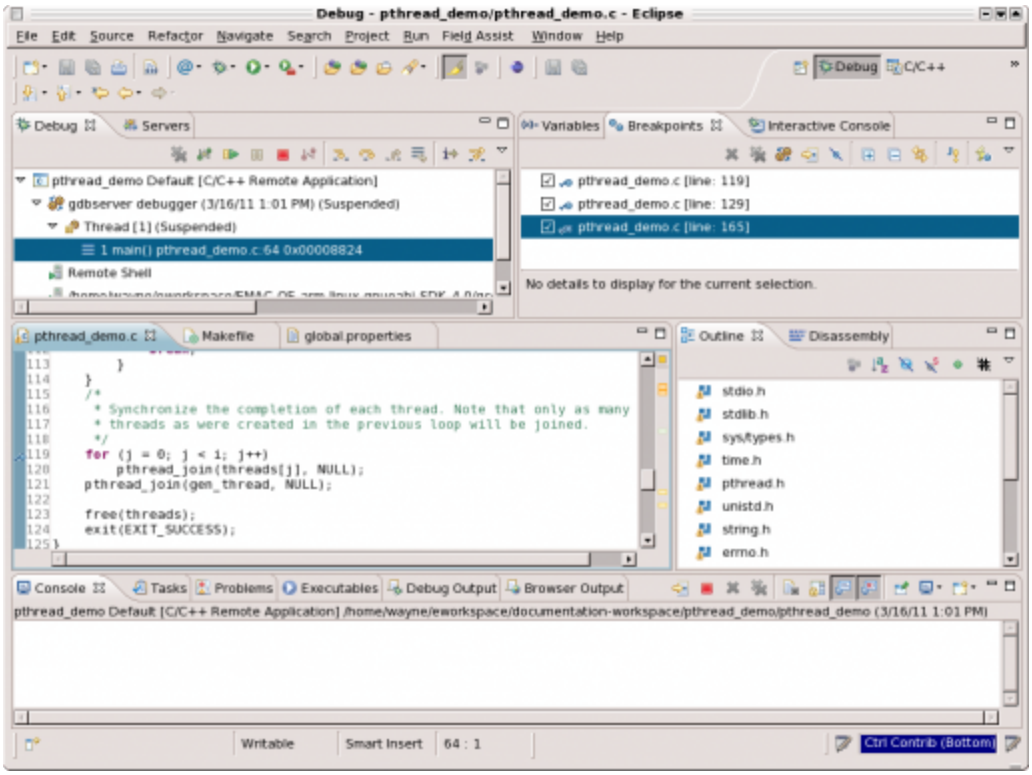

Figure 5. Add Breakpoints in Debug Perspective

7. After setting the breakpoints, click *Run → Resume*. The program will continue until the first breakpoint, which is line 165 in the `generator` thread. This will yield a workbench similar to what is shown in Figure 6 below. The `main` thread will be suspended somewhere within the call to `pthread_create()`. Note the thread number in brackets. This uniquely identifies threads within a program so that they may be distinguished from one another during the debug session.

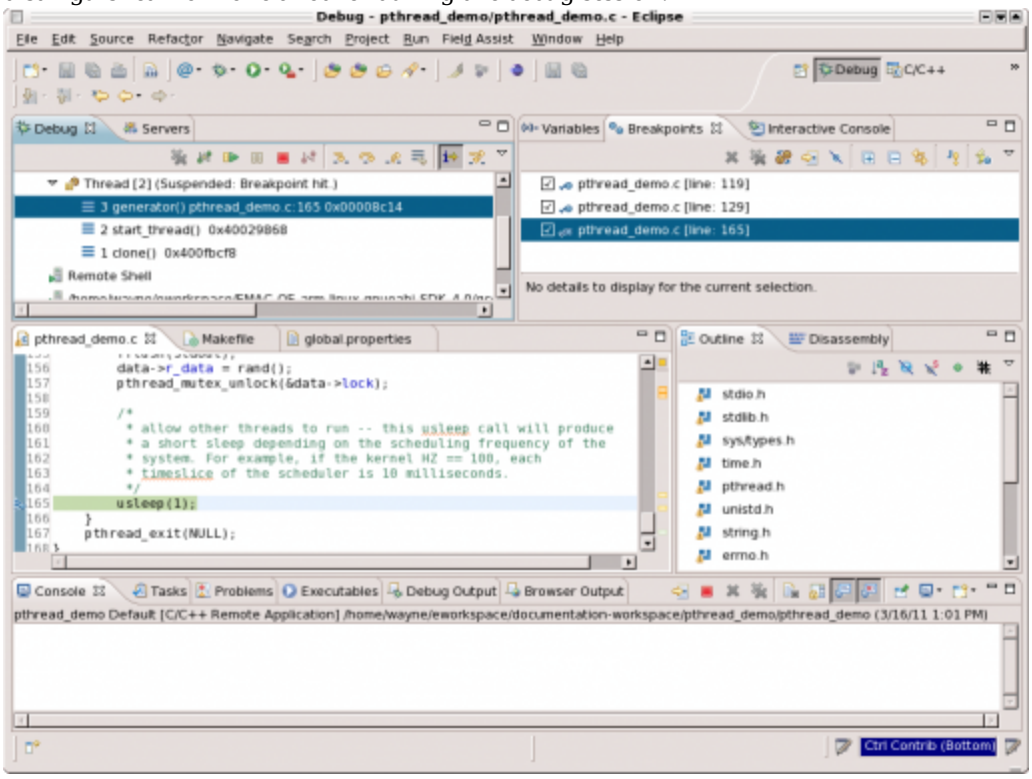

Figure 6. Newly Created `generator` Thread

Lesson 3: Switch Between Threads

After adding breakpoints, the next task is to learn how to switch between threads. The procedure below covers this topic.

1. In the Debug View, the following thread should currently be selected: *pthread_demo [C/C++ Remote Application] → gdbserver debugger (<date> <time>) (Suspended) → Thread [2] (Suspended: Breakpoint hit.) → 3 generator() pthread_demo.c:165 <hex location>*
   This is the third stack frame in the second thread of the `pthread_demo` application. It is currently selected in the Eclipse Debug view because it hit the breakpoint at line 165.
2. In the Eclipse window frame (not to be confused with a stack frame) which contains the Variables and Breakpoints Views, select the Variables View. Notice that the currently-selected stack frame will always be the one whose variables are shown in this view. In terms relevant to source code, a stack frame for a particular function is where its local variables are stored in the process's virtual memory space.
3. Expand *Thread [1] (Suspended)* as shown in Figure 7:
   - Notice the similarity between the hex location of the current instruction in this thread and that of the first stack frame in Thread 2. These memory locations are similar because this is where the new thread splits off from the main thread. See Figure 7 below for a demonstration of this similarity:

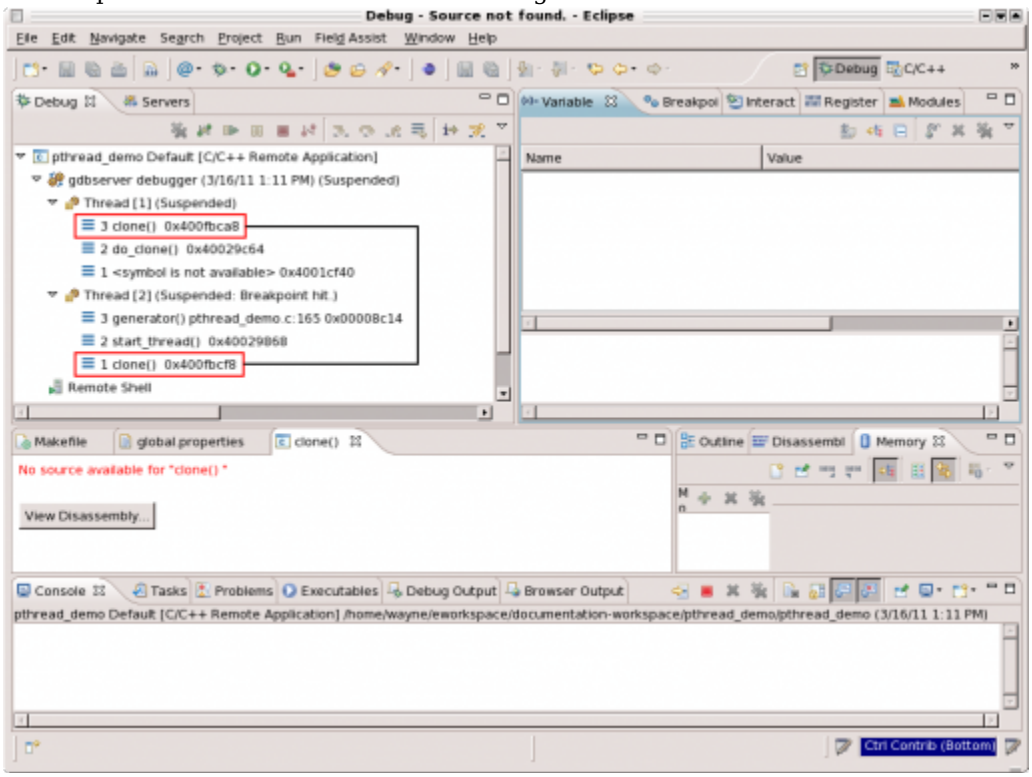

Figure 7. New Thread Branch Address

   - Notice also that the variables shown in the Variable View differ between these two stack frames as described in Step 2.

Lesson 4: Enable Scheduler-Locking Using the GDB Console

This lesson will demonstrate how to enable the operating system's scheduler-locking for a multi-threaded application to ensure that only a single thread is running for a given Step Into or Step Over command. For debugging most applications, direct interaction with the GDB console is not necessary. It is documented here to demonstrate its applicability in special cases of multi-threaded programs with threads whose lifetimes are especially short.

1. In the Console View, select the console connected to the GDB console from the *Display Selected Console* icon as shown in Figure 8 below.

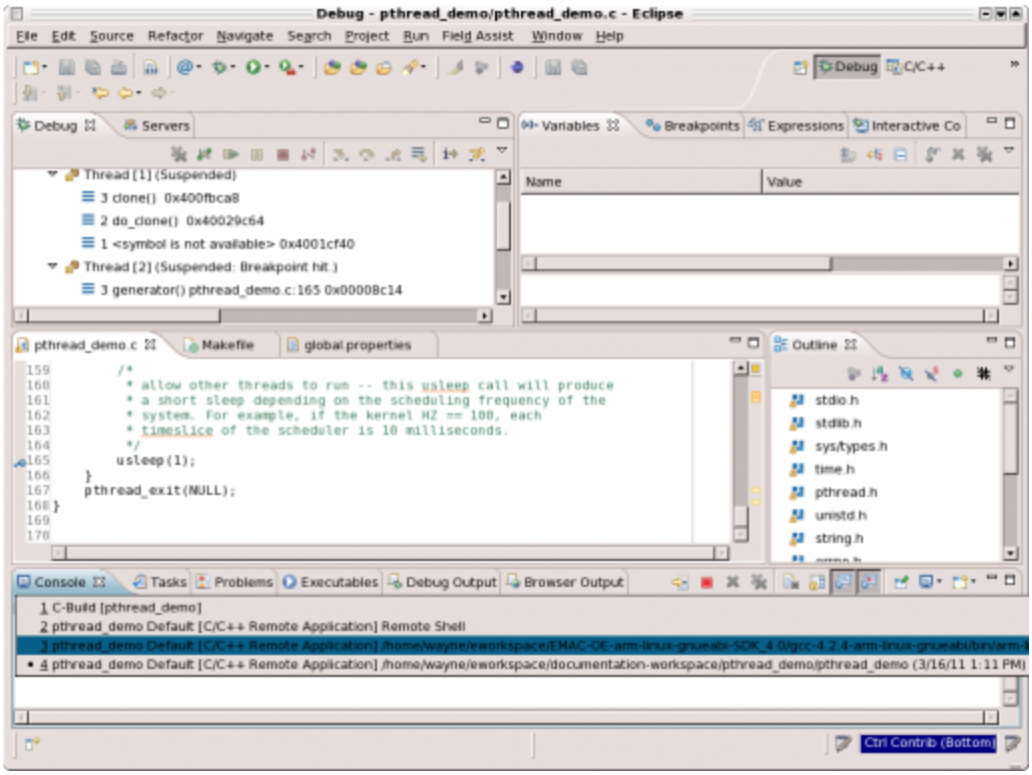

Figure 8: Switching Console Views

Note that the GDB console may or may not be the third console as seen in Figure 8. The GDB session can be identified by the correct EMAC OE SDK path to the GDB binary used by Eclipse to execute the program.

2. In the Console View, enter the following command: `set scheduler-locking on`. This command ensures that the Step Into (F5) and Step Over (F6) commands only apply to the thread currently selected in the Debug View.

3. Click the Resume button in the Workbench:  This will cause the `main` thread to iteratively create all the `reader` threads then suspend for the breakpoint at line 119. Each `reader` thread will suspend at line 129.

Lesson 5: Step Through Code

Once scheduler-locking is enabled, it is possible to step through the source code of one thread at a time. This lesson will demonstrate that functionality.

1. Select the `generator` thread.
2. Click *Window → View → Disassembly*. This will switch to the Disassembly View in the frame which contains the Outline View.

3. In the Debug View, click the Step Over button:  Watch the Editor containing `pthread_demo.c` and the Disassembly View while doing this. Notice that the currently-running line in the both the `pthread_demo.c` Editor and the Disassembly View are highlighted. The highlighted line in the Disassembly view advances each time the *Step Over* command is given while the highlighted line in `pthread_demo.c` remains on the statement to which each of the Disassembly View highlighted lines correspond.
4. Continue stepping over until the line 165 is reached once more in the `generator` thread. This will ensure that the `generator` thread has given up the mutex lock. For a detailed description of what a mutex variable is, please see the EMAC OE SDK Debug Walkthrough Lesson 3, Step 16.
5. Select one of the `reader` threads. These are the threads whose thread number is 3 or greater in the Debug View.. Step from line 129 to line 138. Notice that the `reader` locks the mutex variable before reading or accessing it. It also unlocks the mutex variable before exiting. To see the output of `generator` and `reader` threads, switch the Console View entry with “Remote Shell” specified at the end as shown above in Figure

8.
6. To generate a new random number, select the `generator` thread once more. Then click the *Resume* button or press F8. This will cause the generator thread to iterate through its while loop until it reaches the `usleep()` call at line 165 once more.
7. Repeat steps 5 and 6 until each of the `reader` threads have made a call to `pthread_exit()`.
8. To finish the debug session gracefully, perform the following steps:
    1. Switch to the GDB Console View once more and enter `set scheduler-locking off`.
    2. In the Breakpoints View, uncheck the check boxes next to each of the currently-set breakpoints.
    3. Select the `main` thread. Then select *Run* → *Resume* or press F8 to allow each of the threads to gracefully join the `main` thread. Once this has happened, the program will exit.

Other Debug Tools

The walkthrough above demonstrates a typical but very basic usage of the Eclipse Debug Perspective. There are additional tools available through the Eclipe Debug Perspective, some of which are detailed below.

The Variable View

The Variable View allows the developer to inspect and alter the values of variables while the program is paused, including navigating data structures and dereferencing pointer locations. In addition, watchpoints can be set to pause execution when certain conditions are met. It is recommended that the Variable View be used from within the Debug Perspective. The variables are only shown during an active debug session.

Change Variable Values

To change the value of a variable:

1. Right-click on the variable in the Variable View.
2. Select *Change Value...*
3. Enter the new value in the *Set Value* dialog then click *OK*.

Set Watchpoints

A watchpoint is similar to a breakpoint in that it causes program execution to pause. It differs in that instead of causing the program to suspend on a specific line, it causes the program to suspend when a specified variable is read or written during program execution. To set a watchpoint on a variable:

1. Right-click on the variable in the Variable View.
2. Select *Add Watchpoint (C/C++)*.
3. In the *Add Watchpoint* dialog, choose the *Access* types for which GDB should watch. This can be either or both of the values *Read* and *Write*. This tells GDB to watch for reads from or writes to the variable.

Hardware watchpoints will need to be disabled on most EMAC products using the GDB console as described in Lesson 4 above. In place of `set scheduler-locking on`, use the following command: `set can-use-hw-watchpoints 0`

Using the Memory Monitor

The Memory Monitor is a tool that allows the developer to see changes to adjacent portions of memory while stepping through source code. The changes show up as memory values highlighted in red as seen below in Figure 9.

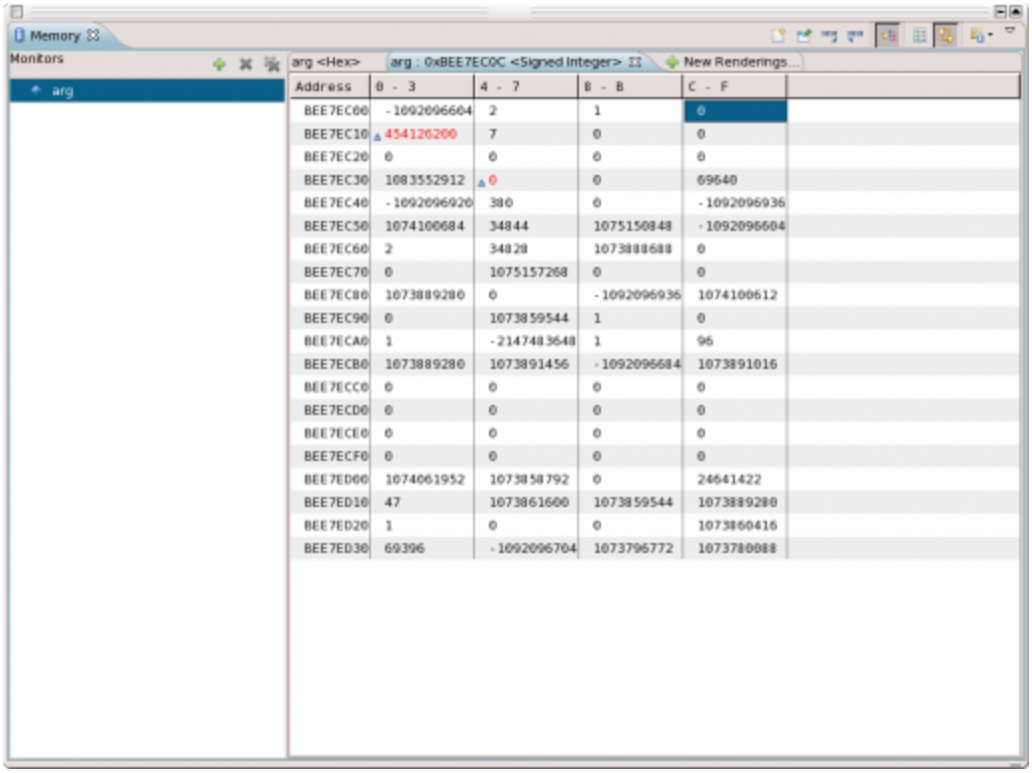

Figure 9. Memory Monitor View

Note that the Memory Monitor View normally appears in the same Eclipse workbench frame as the Console view but was enlarged and singled out for demonstration in this guide.

To view a specific location in memory corresponding to a program variable:

1. Activate the *Variable View* in the Eclipse workbench.
2. Right-click on the variable whose location is to be examined then Select *View Memory* in the context menu.

The Memory Monitor can support multiple "Renderings" of a particular memory location. Each type of rendering is a different way to represent the values of the memory locations. Memory locations can be viewed as Hex, ASCII, Signed Integer, Unsigned Integer, Hex Integer, or Traditional. To view a different rendering of the same memory location, click the *New Renderings...* tab in the Memory Monitor View. Figure 10 below shows the "Traditional" Rendering which shows three panes: Address, Binary, and Text.

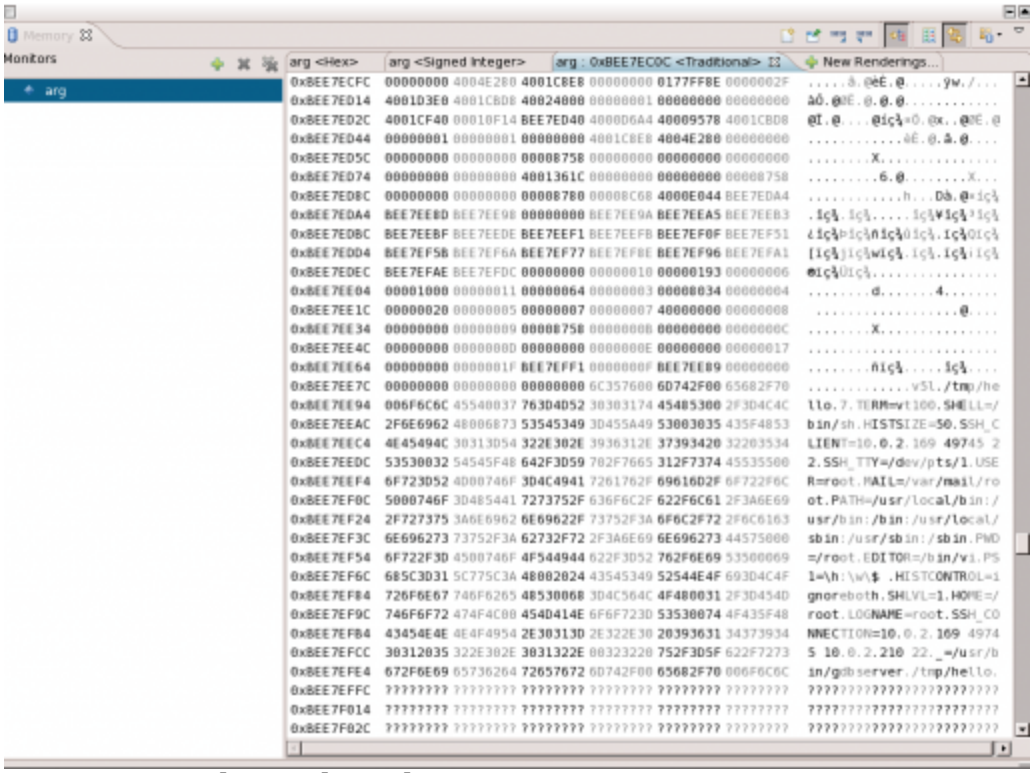

Figure 10. Traditional Rendering

The Disassembly View

In some cases, it is necessary to go deeper into a program than the source code to understand what is going on. In these cases, it is useful to set the Eclipse debugger to Instruction Stepping Mode and observe the location of the instruction pointer while the program is suspended. This can provide an insight into the way a line of source code is compiled into machine instructions, for example on a line with complex logic operations inside a conditional block.

Instruction Stepping

Instruction stepping alters the functionality of the Step Into and Step Over GUI commands so that they step line-by-line through the program's disassembled code rather than through its source code. To enable instruction stepping, click the Instruction Stepping Mode button in the Debug View: 

The Register View

The Register View provides about as deep an insight into what the program is *really* doing as possible. This view shows what values the various registers in the target machine's processor currently hold at this point in the program's execution. These can be viewed or altered just like source code variables.

Be careful when changing register values as this can drastically alter the functioning of the system. In particular, care should be taken with instruction and data addresses. Modifying these values requires advanced knowledge of the relationship between CPU architecture, physical memory, and the operating system's virtual memory.

Next Steps

If you have not already done so, learning to remotely debug using the command line interface for GDB and its remote companion `gdbserver` may improve your understanding of the debug process. The EMAC OE SDK Debugging Guide provides useful introductory material for this purpose.

See Also

- Eclipse IDE
  - Install
  - Development System Configuration
  - First Time Using Eclipse
  - Import EMAC OE SDK
  - Eclipse Terminal View
  - Using the EMAC OE SDK Examples Projects
  - Create New EMAC OE SDK Projects
  - Using the EMAC OE SDK Eclipse Plugin
  - Remote System Explorer Configuration
    - RSE Setup
    - RSE SFTP Setup
    - Remote Shell/Terminal Setup
  - Execute Remote Applications
  - Debug Remote Applications

» import » terminal » example » newproject » remote » rse » stfp » execute » linux_start » debug

---