White Paper

Bernth Andersson
Developer Products Division

Using the Intel® Application Debugger for Intel® Atom™ Processor

Document Number:

319696-002US

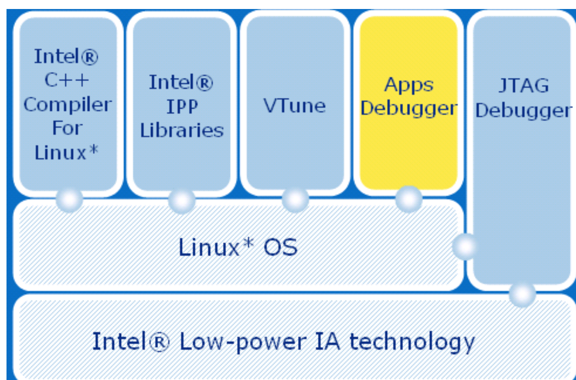
Introduction

This white paper gives a high-level overview of the Intel® Application Debugger for Intel® Atom™ Processor and is targeted primarily towards programmers writing applications for Intel® Atom™ processor based devices. It provides a description of the available features and benefits, while examining key features in more detail.

Overview

The Intel® Application Debugger for Intel® Atom™ Processor is part of the Intel® Application Software Development Tool Suite and Intel® Embedded Software Development Tool Suit for Intel® Atom™ Processor. It provides a full Eclipse RCP based GUI that helps to have better visibility of the application and system properties and thus have better control over the debugging process.

All of these tool suite components have been specifically optimized and enhanced in their features to enable you to get the best performance out of your Intel® Atom™ Processor based device targeted application and reduce the time-to-market.



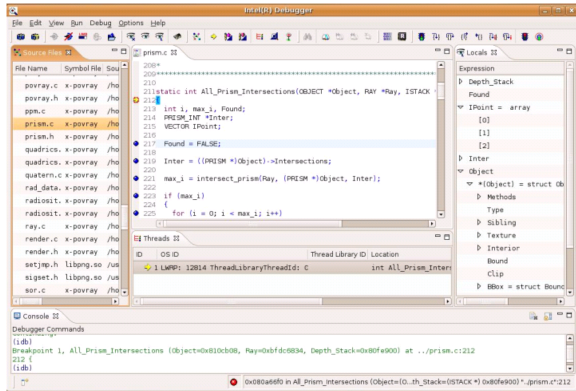
Rationale

Developing software applications and modifying and customizing device drivers for small form factor embedded systems running Linux requires a slightly different development paradigm than writing software for the native desktop PC or workstation. The main difference is two-fold. First the target device you are developing for may not have a keyboard but a small touch screen based user interface. This is very end-user friendly, but not very practical for development on the device itself. Secondly since your target device most likely is a dedicated system the Linux* OS image running on it may not even contain the GNU* GCC software development tools. These two factors mean that despite the platform similarities between a standard desktop PC and a low-power IA based Mobile Internet Device (MID) usually the most appropriate and efficient way of developing for a Mobile Internet Device is to do cross-development.

Debugging your application in such environment could be a challenge. The purpose of this white paper is to give some hints on how the Application Debugger can be used to shorten the debug process. It is assumed that the Intel® C++ Software Development Tool Suite for Linux* OS Supporting Mobile Internet Devices is installed and the system requirements specified in the documentation are met.

Intel® C++ Application Debugger

The Intel® Application Debugger for Intel® Atom™ Processor provides a powerful Eclipse RCP based GUI. It helps to have better visibility of the application and thus have better control over the debugging process.



Before you start with a debugging session we recommend that you make sure that you have a target system running one of the supported Operating Systems listed in the Release Notes.

Make sure that your target system is connected to a network. The Application Debugger requires a server application running on the target system to handle communication between development host and debug target. This program can be found at `/opt/intel/atom/idb/2.0.xxx/server` in the tool suite installation and simply copied over onto the target's file system. The location of the remote server on the target system is not important however it must be started before you can continue with the debug session. We recommend that you start the remote server as root to avoid any access problems if you would like to upload an application to debug. You also need to know the TCP/IP address of the target system.

Application Preparation

For the initial debugging of an application it is recommended that you compile it with debug information and no optimization. It is also convenient to rename the output to something different then 'a.out'. All this can be achieved in one go with the following compiler invocation line:

$ icpc –g –O0 –o *ouput_file_name input_file_name*

Other compiler switches may be used depending on your application. Let us assume that you have a file called *pi.cpp*. You can compile and verify that the program is working by executing the following commands:

```
$ icpc -g -O0 -o pi pi.cpp <cr>

$ ./pi <cr>

The value of PI is  3.141592653592
```

If you are using a stand alone target system you must make sure that your application is made available to the target OS. One method would be to establish a shared drive with access from the host and target or use the scp command.

Starting the Debugger

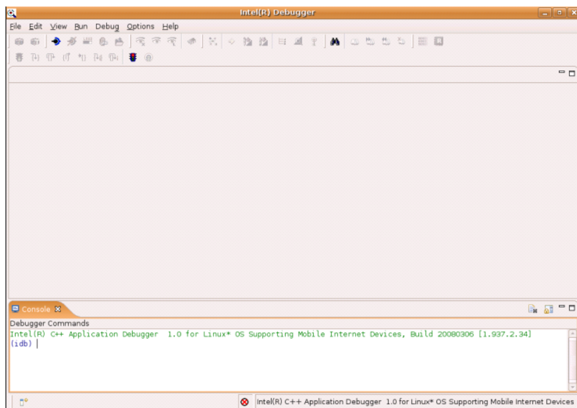
When you start the remote server on the target system it will respond with a sign-on message on the console of the target system:

```
Intel(R) Debugger Remote Server for IA32 Linux, Version 1.0.0
Copyright (C) 2006-2008 Intel Corporation. All rights reserved.

Waiting for connection with "tcpip:2000"...
```

You are now ready to start the Application Debugger on the host system. Look for the *idb.sh* file in `/opt/intel/atom/idb/2.0.xxx/bin` directory. You can start the Application Debugger with the following command:

```
$ cd /opt/atom/intel/idb/2.0.xxx/bin <cr>

$ ./idb.sh <cr>
```

The Application Debugger will open up for the first time with an empty window with a menu line with most buttons grayed out and below a Console window. If it has been used before it opens with the same window layout as when it was closed. Only the Console window will contain any data – the sign in message.



Below the Console window you find a status bar. In the middle you have an indication of what mode of operation the debugger is in:

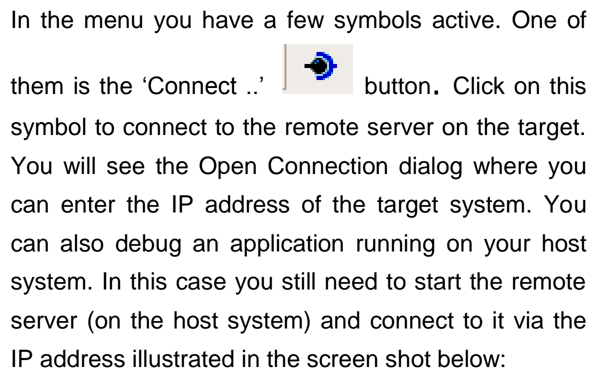
Disconnected from the target (also for start-up) -

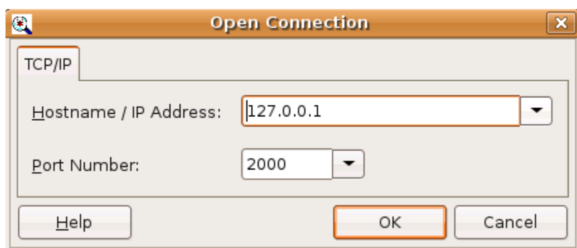
Connected to target ------------------------------------

Application loaded or attached ----------------------

Application active – running state -------------------

Application stopped ------------------------------------

In the menu you have a few symbols active. One of them is the 'Connect ..'  button. Click on this symbol to connect to the remote server on the target. You will see the Open Connection dialog where you can enter the IP address of the target system. You can also debug an application running on your host system. In this case you still need to start the remote server (on the host system) and connect to it via the IP address illustrated in the screen shot below:



The port number should be the same as the one used when the remover server was started. It is possible to have more than one Application Debugger running - each would be connected to a remote server with different port number. This allows you for example to debug an application both on the host and target at the same time.

If you get an error message like the one below:

```
connect refused by remote server
(idb)
```

most likely the IP address or port specified in the Open Connection dialog is wrong.

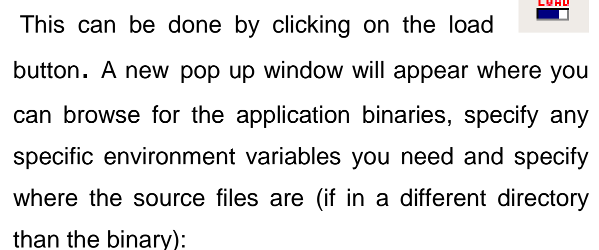
When the connection is established the remote server will provide a message to the target system console:

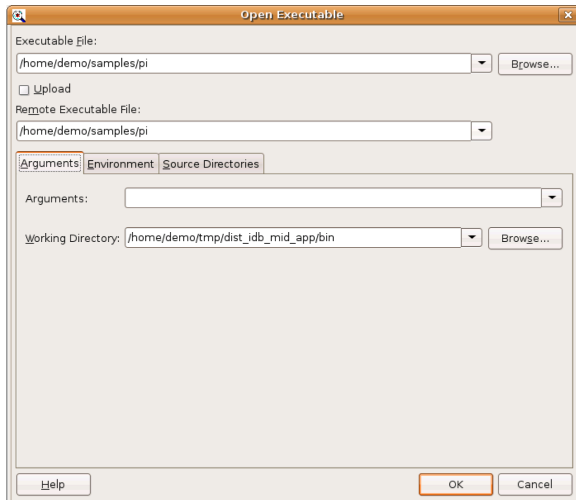
```
Intel(R) Debugger Remote Server for IA32 Linux, Version 1.0.0
Copyright (C) 2006-2008 Intel Corporation. All rights reserved.

Waiting for connection with "tcpip:2000"...
Connected
```

On the host system the debugger GUI will be updated to illustrate that you now are ready to attach to a running process or load a new application (those buttons are now enabled). You will find examples for each of both below.

Load and Debug a new Application

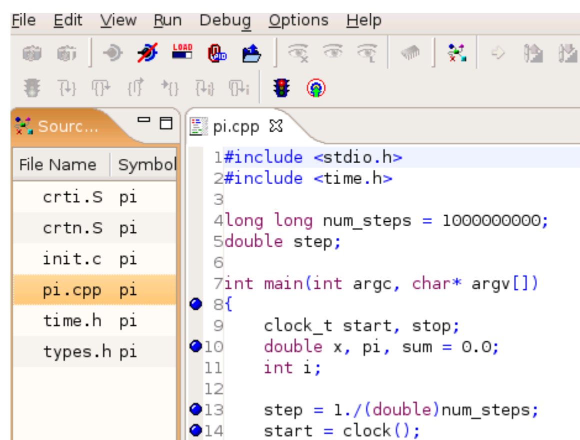
Let us use the pi application. You have compiled the application as recommended above and want to upload the application to the target (as mentioned this requires that the debug server on the target is launched with root privileges).

This can be done by clicking on the load  button. A new pop up window will appear where you can browse for the application binaries, specify any specific environment variables you need and specify where the source files are (if in a different directory than the binary):



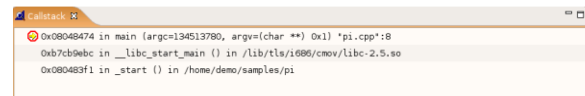
You need to check the 'Upload' box if you want the debugger to copy the file over to the target system. In the 'Target Executable File' field you enter the location on the target where you will store the executable. Make sure that all directories you specify already exist on the target system and that the file name is the same as specified in the 'Executable File' field. (The location on the host and target can be different but the file name should be the same). When you press the OK button it will take some time after which the status indicator will change to yellow. You are now ready to start debugging.

First let us look at what source files/modules you have available. Click View/Source Files (or Ctrl+Alt+F) and the source file window will open – here you can double click on a source file to get it displayed in the source window:

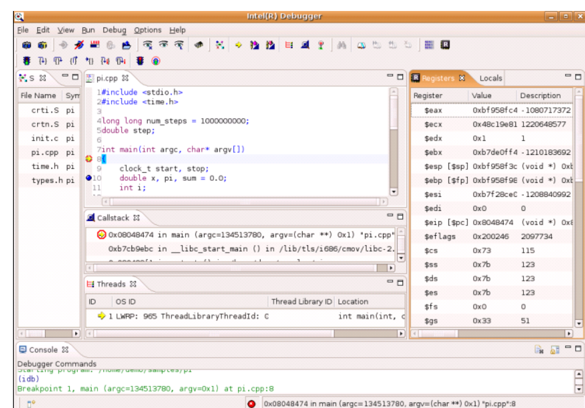


You can set a breakpoint by just double clicking on any of the blue dots to the left of the line numbers in the Source window. Note that if you select line 8 (in the screen shot above) you are outside of the stack frame from main and hence have no access to local variables.

Another very useful window is the Callstack, where you can see the path through the function calls you have taken to reach your current position in the code. You can open the Callstack window by selecting View/Callstack from the debugger menu or typing Ctrl+Alt+R:



Additional windows which might be useful are the Register, Locals and Threads windows. The Register window is opened with View/Register or Ctrl+Alt+G and shows the current values of the segment registers, general purpose registers, floating point registers and the xmm registers. The Locals window shows the local variables and is opened with View/Locals or Ctrl+Alt+V. The Thread window is particularly useful if your application is multithreaded. It shows which thread you are currently debugging. You open this window with View/Threads or Ctrl+Alt+T. You may now have a debugger view looking like this:



Working with the Debugger

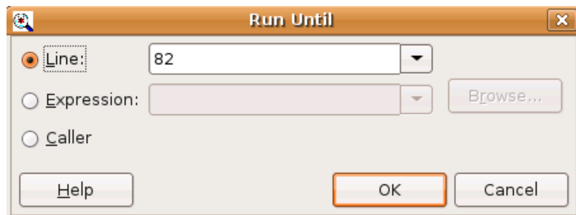
One of the most important functionalities you have at your disposal to find errors are the various ways you can step through the code. Following symbols can be used:

- step into; step one source line entering a function if required – same as F11

- step over; step one source line proceeding through function calls – same as F12

- run until caller – same as shift + F11

- run until – when you press this button a dialog will appear where you can specify the location:

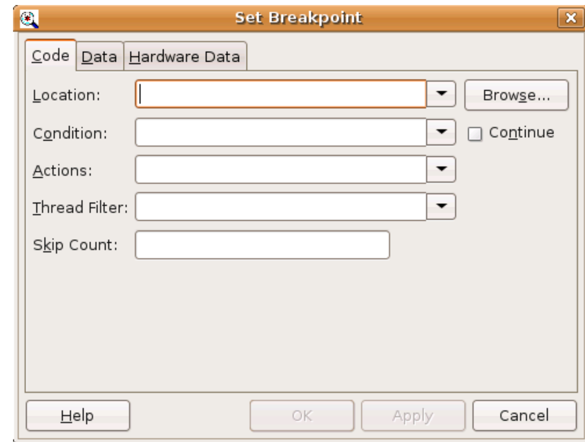


You can also make smaller steps aimed at assembly instructions:

- instruction step into – same as F6

- instruction step over – same as F7

As we have seen you can set a break point on a source line by just double clicking on the corresponding blue dot in the source code. You can also open up a dialog by selecting Debug/Set Breakpoint … from the debugger menu.



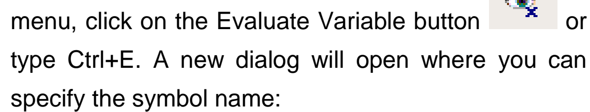
This dialog allows you to set breakpoints based on conditions. Furthermore you can specify a breakpoint which will only trigger after it has been reached a certain number of times (by using Skip Count). It is also useful for multithreaded applications as you can specify the thread id (by using Thread Filter) for which thread you would like to apply the breakpoint if more then one thread is executing the code on the location address. The differences between 'Data' and 'Hardware Data' breakpoints are basically: Hardware Data uses the processor debug registers directly (hence you can only have a limited number active at the same time). They are automatically thread specific. The Data breakpoint requires more internal work in the debugger (in most cases page protection is used, which gives you quite some flexibility) and hence they typically take longer time.

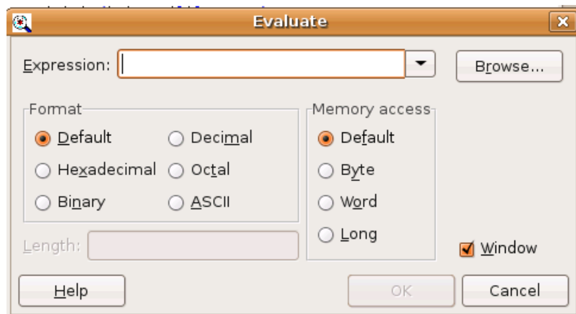
Once the break point is set you can use the 'Continue' and 'Stop' buttons as required.

Hint: If the debugger Console window indicates "Program exited with code 127" then look at the output from the target system. One possible reason for this error is that the application was built with a shared library which does not exist on the target system. If this is the case then copy the missing shared library over to the target system and set the environment variable LD_LIBRARY_PATH to point to the location of the shared library.

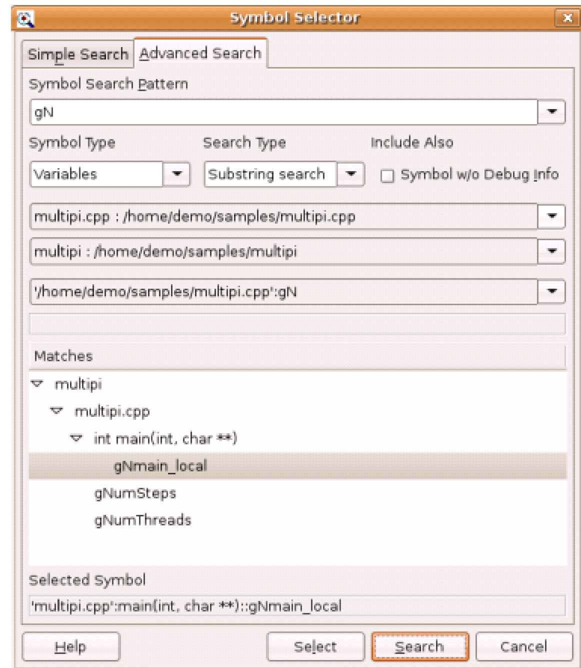
Easy access to symbol information is important for a High Level Language debugger. There are a number

of ways of displaying the current value of a symbol. One easy way is to move the cursor over the symbol in the source window. The value of the symbol will be shown in a window just under the selected symbol. Another way of displaying the value of a symbol is to open an Evaluate window. This can be done by either selecting Debug/Evaluate.. from the debugger menu, click on the Evaluate Variable button or type Ctrl+E. A new dialog will open where you can specify the symbol name:

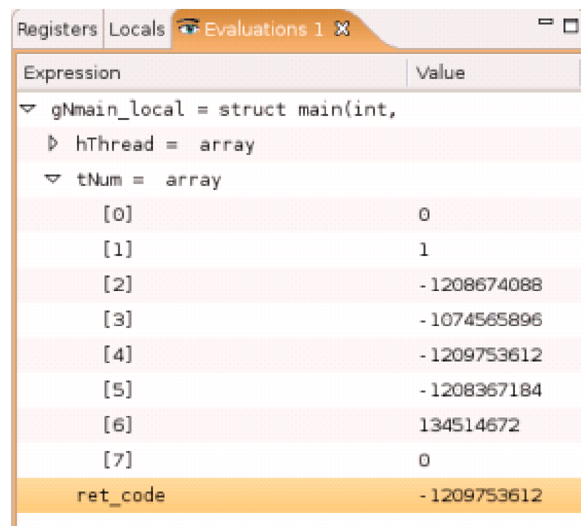




This is done in the Expression field. If the symbol name is not known you can click on the Browse button and the Symbol Selector dialog will open. Here you can search for a variable even if you do not know the exact name (Substring search). For example if you only know that the variable starts with gN you can enter these characters as Symbol Search Pattern and select the Search button. A list of all variables (you can decide if function names should also be included)I which matches the search criteria will be displayed. This is illustrated in the next screenshot:



You high light the variable you are looking for and hit Select. The variable name will now be transferred to the Evaluate dialog. If you click on OK you will see the following window:



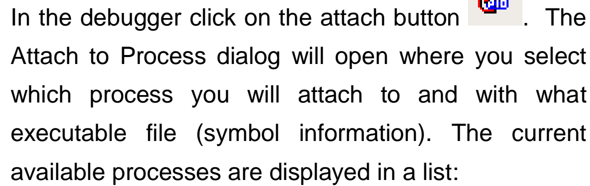
It is often desired to view C++ objects and other complex data structures. This can be done using the same technique as described above. If you right mouse click inside the Evaluation window you can change the value of a selected variable (select Set Value from the context menu).

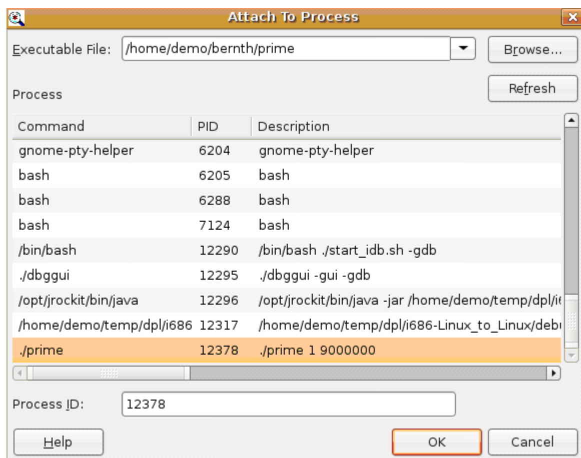
Additional Features

In this section we will look at how we can attach to a running process, debug a shared library and a multithreaded application.

Attach to a Running Process

Once you connected the debugger to the remote server you can attach a debug session to any running process you have the sources for.

In the debugger click on the attach button . The Attach to Process dialog will open where you select which process you will attach to and with what executable file (symbol information). The current available processes are displayed in a list:



Just double click on the process in the list and the debugger will stop this process and fill the open windows with information from this process. You can now continue debugging this process in the same way as the one downloaded to the target via the debugger. Just bear in mind that any output from the application will be re-directed to the shell from where it was started.

Multithreaded Applications

The Application Debugger is aware of multi threaded applications and uses a simplified method of 'stop all' debugging. Basically when any thread is hitting a breakpoint the debugger will freeze all threads at their current location. If you single step in one thread the others will enter run mode. If you single step assembly instructions (F6/F7) you will stay in the same thread as the other do not enter run mode. The following screen shots are using a multithreaded version of the pi application:

```
multipi.cpp
33int main(int argc, char **argv)
34{
35    pthread_t hThread[gNumThreads];
36    int tNum[gNumThreads];
37    int ret_code;
38
39        printf("Computed value of Pi: ");
40
41    pthread_mutex_init(&MyLock, NULL);
42    gStep = 1.0 / gNumSteps;
43    for ( int i=0; i<gNumThreads; ++i )
44    {
45        tNum[i] = i;
46        pthread_create( &hThread[i],  // thread handle
47                        NULL,         // thread attributes
48                        threadFunction,  // Thread function
49                        &tNum[i]);  // Data for thread func()
50
51    }
52    for (int j = 0; j < gNumThreads; j++)
53            pthread_join(hThread[j], NULL);
54
55    pthread_mutex_destroy(&MyLock);
56
57    printf("%12.9f\n", gPi );
58    return 0;
59}
```

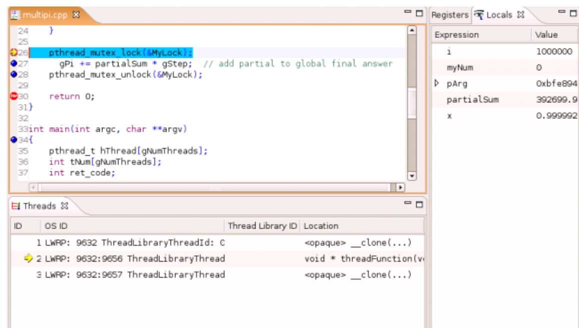
```
11void *threadFunction(void *pArg)
12{
13    int myNum = *((int *)pArg);
14    double partialSum = 0.0, x;  // local to each thread
15
16    printf(" Thread %d is now starting its work\n", myNum);
17
18    for ( int i=myNum; i<gNumSteps; i+=gNumThreads ) // use every
19                                                   // gNumThreads-th step
20    {
21        x = (i + 0.5f) / gNumSteps;
22        partialSum += 4.0f / (1.0f + x*x);       // compute partial
23                                                   // sums at each thread
24    }
25
26    pthread_mutex_lock(&MyLock);
27      gPi += partialSum * gStep;  // add partial to global final answer
28    pthread_mutex_unlock(&MyLock);
29
30    return 0;
31}
```

In the main function you find that 8 threads are created and each of them are passed an unique number (0-7) [line 46-49]. The worker threads will execute the thread function (void *threadFunction(void *pArg)). In this function the different partial sums are calculated [line 27]. For 'thread 0' the loop index (i) will have a value of 0, 8, 16 etc. For 'thread 1' the loop index (i) the corresponding values are 1, 9, 17 etc. [line 18]. With this split the worker threads will have an approximate even workload.

A mutex is used to make sure that each thread can update the global variable gPi without a race condition.

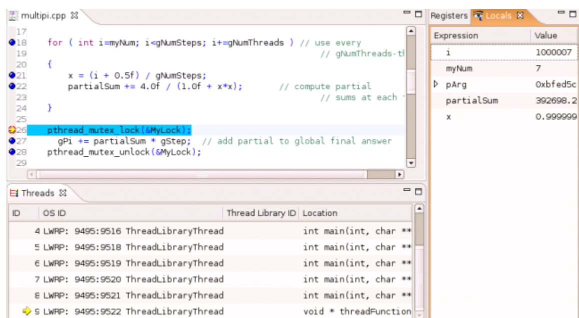
As all threads will execute the same code a breakpoint in threadFunction() will be hit be all threads. In such situations it is important to be able to restrict a break-point to only one thread. The next screen shot shows the first worker thread hitting the

breakpoint while the master thread is in the middle of the fork process for the second worker thread:



The yellow arrow indicates which thread you are currently debugging. The local variables to the right contain the values from worker thread 0.

If you initially set a breakpoint in main i.e. master thread you may not be able to catch all of the worker threads. Carefully selecting the breakpoint locations is import when debugging multithreaded application. The next screen shot illustrates how a breakpoint was hit in the last worker thread while all the others were in the 'join phase':



Debugging Shared Libraries

Another important aspect of application debugging is the ability to debug a shared library. The next two command lines illustrate how you can build a shared library (libshb.so) from the source file shbernth.cpp and build an application using this shared library:

- icpc –g –O0 –o libshb.so –shared *shbernth.cpp* <cr>
- icpc –g –O0 –o *shprime* –I. –L. –lshb *shprime.cpp* <cr>

The executable *shprime* is dependent on the shared library *libshb.so.* The test application will call function shbernie() in the shared library.

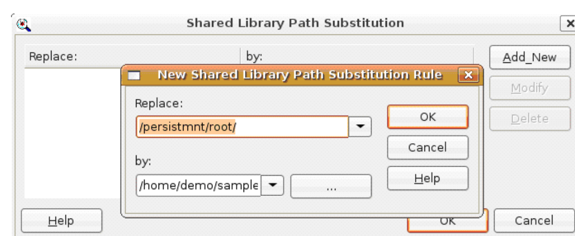
First you need to make sure that the shared library and the test application is available on your target system. This can be done by using the scp command if you know the ip address and password for the target system. Example command:

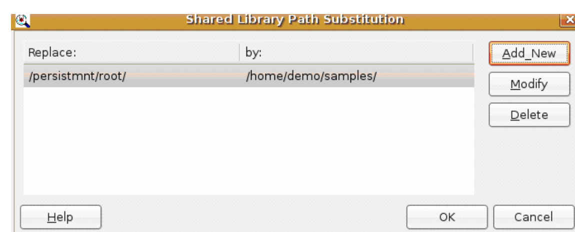
- scp ./libshb.so root@192.168.10.50:/root

This will copy the file libshb.so from you local directory to the /root directory on the target system. For the test application you can either copy it over or use the upload button on the load (Open Executable) dialog.

As your application can be dependent on several shared libraries and these are located (typically) in different places the debugger provides a 'Shared Library Settings.' dialog from the Options menu. Here you specify where on the host you have your shared library which you copied over to the target under /root.

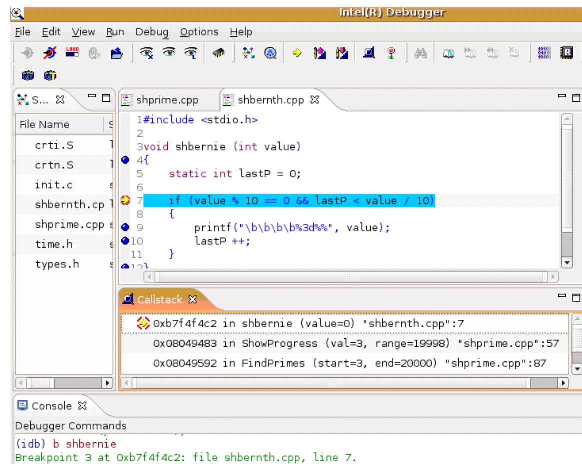
Specifically for the moblin system your /root on the target is mounted to /**persistmnt/root**. This you have to replace with the full path to your shared library on the host:





After this you can set a breakpoint in the shared library either via 'b functionname' in the Console

window or using the Set Breakpoint dialog from the Debug menu. It is not recommended that you try to single step into the shared library function. This will take some time as you will go through the target OS loader.



Application Trace Facility

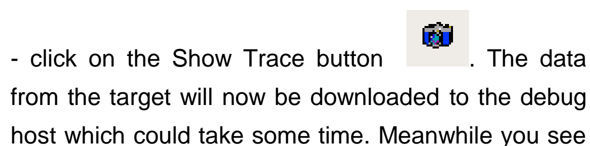
A feature which is very useful during debugging is the trace facility (does require a kernel patch – for more information please view the read.me file in `/opt/intel/atom/idb/2.0.xxx/kernel-patches` directory). For example you can use the trace facility to verify what instructions you actually have been executing. This is very useful if for example you are getting an incorrect result or the application terminates unexpectedly.

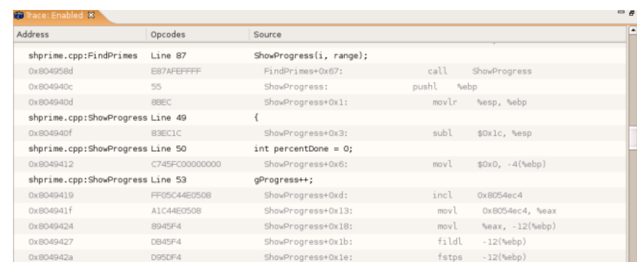
The following steps will display the application trace:

- stop at a suitable location before you enter the interesting area. For example stop at the function call 'ShowProgress(I, range);'

- click on the Toggle Trace button

- continue until you passed the function or area of interest.

- click on the Show Trace button . The data from the target will now be downloaded to the debug host which could take some time. Meanwhile you see an empty window. When the data is collected the execution path is regenerated with source code intermixed with the assembly instructions:



Standard Debugger Features

The Application Debugger does not only uniquely provide the cross debug capabilities and the execution trace capability to Mobile Internet Devices, but also supports the standard capabilities most users expect from a modern symbolic source-code application debugger:

• Attach to (and detach from) a running process and debug the corresponding program

• Load a program into (and unload a program from) the debugger, automatically creating and deleting corresponding processes as necessary

• Support multiple-process debugging, where the processes may be associated with the same program or with multiple other programs:

- Actively run one process at a time
- Switch focus between processes
- See processes and examine detailed process state
- Set breakpoints for a specific process

• Debug programs with shared libraries

• Provide language-specific evaluations of command expressions

• Provide ability to "call" functions in a target process from within a command expression

• Catch/ignore unaligned access

• Display the source listing of a program

• Set breakpoints to stop program execution when specified sections of program code are executed

• Set watchpoints to stop program execution when a specified area of memory or specified program variable is written

• Add conditions to breakpoints and watchpoints so that program execution will only stop at the specified break or watch event when the condition is true

• Support setting of pending breakpoints if a breakpoint location specified cannot be resolved to an address in the current debuggee at the time it is being set.

• Step into or over calls to routines

• Step through the execution of a program one source line or one machine instruction at a time

• Examine the stack of currently active functions

• Examine and change program variables and data structure values in same or in different scopes

• Examine and change the contents of memory in various formats (including international character strings)

• Disassemble and examine machine code

• Examine and change machine register values

• Support mixed-language applications, C++ templates, C++ user-defined operators, and Fortran modules

• Provide a customizable debugging environment by using environment variables, initialization files, sourced scripts, aliases (i.e., parameterized macros), and debugger variables for commands and command sequences

• Support in-place edit of assembly code in RAM for instant fix and execution replay.

- Regular expression searches of the symbol table
- Debug optimized code:
  - In-lined instances of functions (show in backtrace and selectable for current focus)
  - Registerized variables
  - Semantic stepping
  - PC-to-source column mapping (for multi-statement lines)

These key debugging features allow programmers to debug at both the source and machine levels in a single session.

Conclusion

The Intel® Application Debugger for Intel® Atom™ Processor is part of the complete tools solution set to address Intel® Atom™ Processor and embedded development specific software requirements. You can debug your application on your target (cross debugging) or on a KVM* virtual machine running on the on the development host.

The Application Debugger provides a full featured GUI interface to allow you full control over your debugging session. The trace facility allows you to follow your application execution and not being disturbed by the rest of the system. In addition to its GUI the debugger provide a powerful command interface allowing for batch testing.

Where to find it

The Intel® Application Software Development Tool Suite for Intel® Atom™ Processor and Intel® Embedded Software Development Tool Suite for Intel® Atom™ Processor can be purchased or a 30 day evaluation license can be obtained the the tool suites can be downloaded from the Intel® Software Development Products Web pages (http://software.intel.com/en-us/intel-compilers/). Customers have access to product updates and product support through the following Web pages:

• Intel Software Development Products Support:
http://software.intel.com/sites/support/

• Intel® Software Network Discussion Forums:
http://software.intel.com/en-us/forums/software-development-toolsuite-atom/

• Intel Premier Support:
https://premier.intel.com/

Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.
Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.
The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.
Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.
Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright (C) 2009, Intel Corporation. All rights reserved.