# ITP Source Debugger Users Guide

Revision: 2.0

April 2, 2014

# ITP Source Debugger User Guide for ITPII

Publication date March 20, 2013

# Document History

| Revision | Date | Description of changes |
|---|---|---|
| 0.1 | ww44'12 | Added use cases for: basic debugger utilization, loading source & symbols, Linux kernel debug. Added Appendix describing ITPII multi-client issues. |
| 0.3 | WW50'12 | Updates related to 2.2 API change, bug fixes. |
| 0.5 | WW09'13 | Added "Using Breakpoints" and "Troubleshooting Breakpoints" section. |
| 1.0 | WW11'13 | Edited/formatted to Intel document guidelines. |
| 1.5 | WW36'13 | DBSS and Intel PT updates, also misc. |
| 2.0 | WW14'14 | Added Execution Trace section. |

# Contents

## Table of Figures

# Introduction

This document presents an overview of the ITP Source Debugger. It is intended for users who are generally familiar with JTAG-based software debuggers and simply need help getting going with common use cases. This document is not an exhaustive list of all possible debugger features, nor is it an introduction to system software debug in general. An example of a typical user of this document would be a BIOS engineer who already has a debug task in mind, already knows generally what the debug strategy will be, and simply needs help getting this strategy underway.

In this document you will find sections for common usage scenarios such as BIOS debug, Linux Kernel debug, and within those sections you will find a short tutorial on how to perform common tasks. All possible usages are not covered, instead it is assumed that by becoming familiar with one or two tasks the general utilization of the debugger will become more clear and other features can be figured out intuitively.

## Disclaimer

This is a **living** document that is revised with each software release to provide up to date information on new capabilities and changes, as such it may contain slight inaccuracies, and the features described here may change iteratively over time.  If you experience any problems please email them to ITPSupport@intel.com with the subject “ITP Source Debugger” so we can continue to improve this document and the debugger.

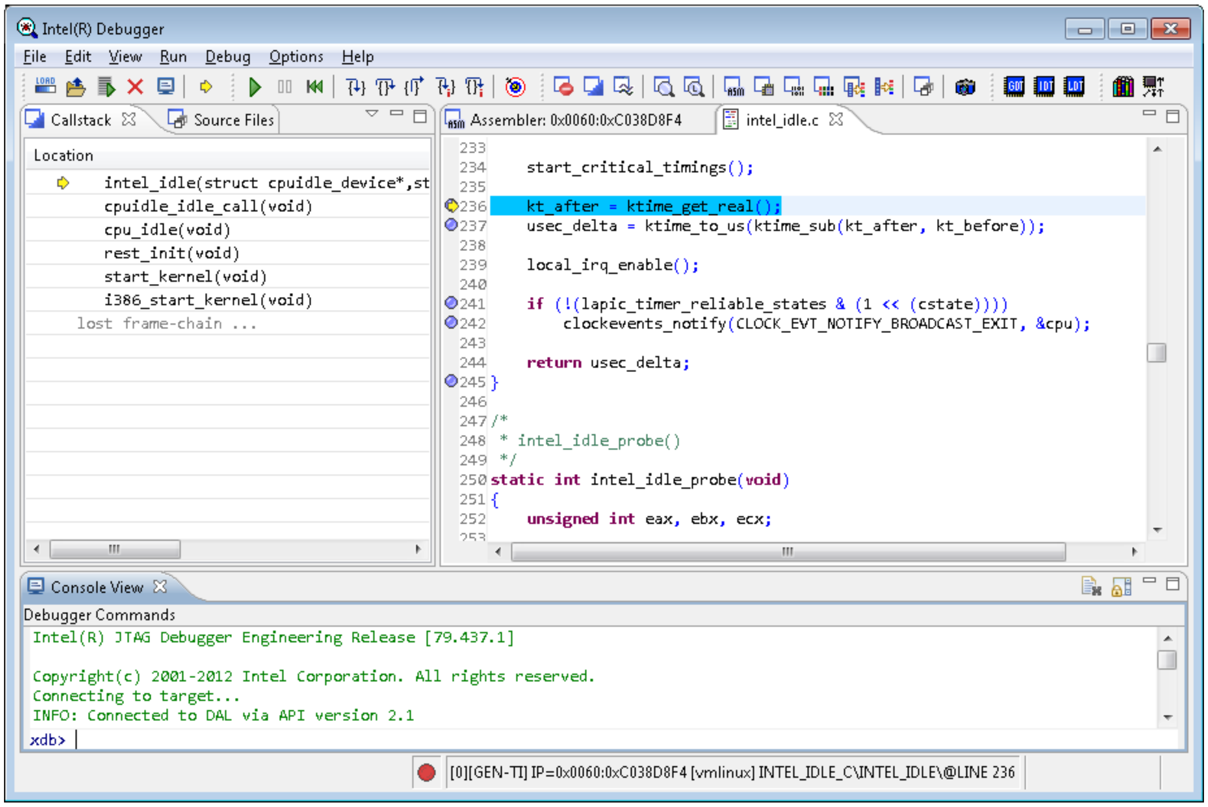
## Overview of the ITP Source Debugger

The ITP Source Debugger is an Architecture System Software debugger. This means it presents a view of the architecturally defined system state (processor data structures and registers) along with the software that is executing and interacting with that state (OS kernel, interrupt handlers).

As a **software debugge**r, most of the Source Debugger’s features are consistent with other GUI-based debuggers such as Eclipse or Visual Studio. You can run and stop the target, view instructions or source code, view and modify memory, registers, program variables, etc. The Source Debugger is a modern GUI-based program with functions available from the drop-down menus and many common functions mapped to toolbar buttons. Most sub-windows have context menus that allow you to access window-specific functions.

As an **architectural debugger**, the Source Debugger provides a set of features for exploring the state of the processor; for example, the CPU registers, Model-Specific Registers, and other hardware states can be directly accessed and modified. Low-level data structures such as the IDT and GDT can also be easily accessed.

In all cases, an attempt is made to present a human-readable view of data with inline documentation of bits, fields, etc., rather than just a simple data word. Figure 1 below shows a typical ITP Source Debugger main window.



```
Location
intel_idle(struct cpuidle_device*,st
cpuidle_idle_call(void)
cpu_idle(void)
rest_init(void)
start_kernel(void)
i386_start_kernel(void)
lost frame-chain ...
```

```

    start_critical_timings();

    kt_after = ktime_get_real();
    usec_delta = ktime_to_us(ktime_sub(kt_after, kt_before));

    local_irq_enable();

    if (!(lapic_timer_reliable_states & (1 << (cstate))))
        clockevents_notify(CLOCK_EVT_NOTIFY_BROADCAST_EXIT, &cpu);

    return usec_delta;
}

/*
 * intel_idle_probe()
 */
static int intel_idle_probe(void)
{
    unsigned int eax, ebx, ecx;
```

```
Intel(R) JTAG Debugger Engineering Release [79.437.1]

Copyright(c) 2001-2012 Intel Corporation. All rights reserved.
Connecting to target...
INFO: Connected to DAL via API version 2.1
xdb>
```

[0][GEN-TI] IP=0x0060:0xC038D8F4 [vmlinux] INTEL_IDLE_C\INTEL_IDLE\@LINE 236

Figure 1: ITP Source Debugger with the target halted.

## Processors supported by the Debugger

The Source Debugger is capable of debugging any CPU that conforms to the architecturally-defined behavior specified in the Intel Software Developers Manual (SDM). In some cases extended support may be provided for a particular CPU, SOC or platform in the form of features such as extended register or MSR awareness.

**Note**: There may be CPUs that are supported by ITPII but that are **architecturally divergent** in the sense that they do not conform to standard execution modes, addressing modes, or other characteristics. Unless otherwise noted, The Source Debugger is **not** guaranteed to function when targeting these processors with ITPII.

# Using the Debugger

## Installation

- The Intel Source Debugger is installed as part of the Intel DFx Abstaraction Layer installation.
- A functioning DAL installation is required. You should be able to launch the Python CLI and reliably connect to the target.
- Prerequisites (note: these should be installed automatically with the DAL):
  - A 32-bit Java Runtime Environment (JRE) is required.
    Note that this does not need to be installed but the Java binaries do need to be on the PATH environment variable.
  - .NET 4.0 (dotNetFx40_Full_x86_x64.exe)
  - Visual Studio 9 Redistributable (vcredist9)
  - Visual Studio 10 Redistributable (vcredist10)

## Launching the Debugger

- Launch the Source Debugger using the C:\Intel\DAL\Apps\XDB\**start_xdb.bat** script in the root of the distribution package or double clicking the "ITP Source Debugger" icon under Intel DFx Abstraction Layer
- A blue banner screen displays; then, the main application window displays.
- You will see status messages in the debugger console indicating it is connecting to the DAL
- Once the connection is complete you can click on the pause button to halt the processor. When the target processor is halted the assembly window will come up. Refer Figure 1

## Target Run-Control

To do anything with the debugger, you first need to understand how to control the execution of the target. This is similar to any other software debugger in that you can run, stop and reset the target. While stopped you can perform various step operations including both assembly-level and source-level steps.

Run Control commands are available from the **Run** menu and most of these functions

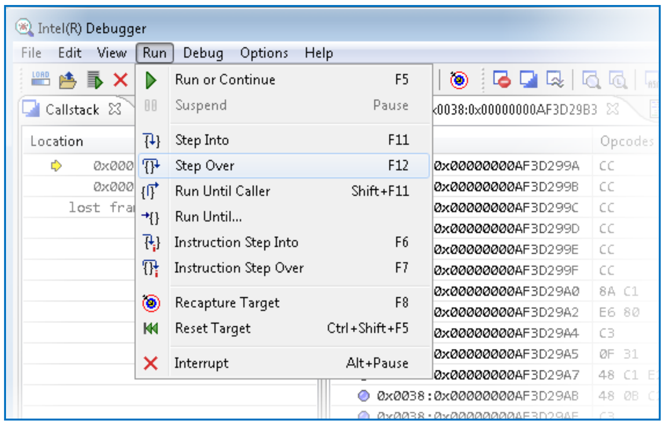


**Figure 2: Run Control commands on the "Run" menu**

are also replicated on the toolbar.

## Indication of target status

The debugger indicates the run state of the target in several ways:

- In the Source Debugger console the prompt indicates run state:
    - “xdb_D>” – **D**isconnected
    - “xdb_R>” - **R**unning
    - “xdb>” Target is halted.
- The Run/Halt buttons will become enabled/disabled depending on run state.
- A green round dot at the bottom right hand corner indicate running and red indicates halted

# Debugging Software using Symbols and Source Code

## The Basics

To debug your software using source code, you need to load debug symbols that are used to map the program in target memory to the original source files. To do this, the Source Debugger needs the following:

- Program loaded in target memory that has been compiled with debug information.
- The load address of the program in target memory.
- The program binary file (executable file).
- Debug information file for the program binary (aka, the **symbols**).
- Original program source code.

**Note**: With ELF/DWARF format files (e.g., Linux), the debug information is generally included in the original program binary. When using the Microsoft Compiler, the debug information is placed in a separate file with the **.pdb** extension; in this case, both the executable file and the PDB file are required and should be in the same directory.

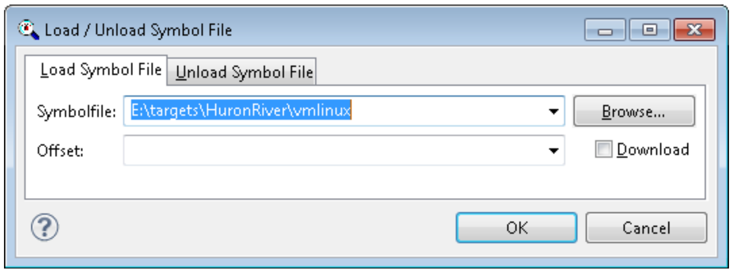
### User Case: Loading Symbols and Source Code

The target should be in "halt" state when the symbol files are being loaded.

Load the debug information using the following **Load** command in the debugger:

**File->Load/Unload Symbol File**

This function requires the program binary, and an optional **Offset** parameter. For normal debug the Download button should not be checked.



Symbolfile: E:\targets\HuronRiver\vmlinux
Offset:

Figure 3: Load/Unload Symbol File Dialog

#### Selecting the Correct Offset

**For fixed-offset modules** (e.g., most Linux kernels and ELF format test code) you leave the Offset parameter blank, this will cause the debugger to use the load offset encoded into the program binary at compile time.

**For dynamically located modules** (e.g., Kernel Modules/Drivers, and EFI BIOS modules) you need to specify the base address of the image in memory. Note that in some cases the Source Debugger provides special helper functions to accomplish this, see the sections on Debugging Linux Kernel Modules and Debugging EFI BIOS respectively.

## Downloading executable code to the target

*Note: if you are debugging code that is *already* loaded on the target and you simply want to display sources for that code then you must* ****not**** *select the Download option!*

The debugger has the ability to download a program binary to target memory via the JTAG connection. If the "Download" checkbox is selected during symbol load then the debugger will analyze the ELF or PE/COFF image and load the appropriate sections to target memory, the load address will be according to the link addresses specified in the file. If an offset is specified then the load address will be adjusted according to this value.

For more information about downloading code please contact customer support.

## Additional notes about loading symbols:

- The initial symbol load may take a little while, especially in the case of large images such as the Linux kernel.
- When the debugger first encounters a new program scope there may be a short delay while it loads additional symbolic debug information

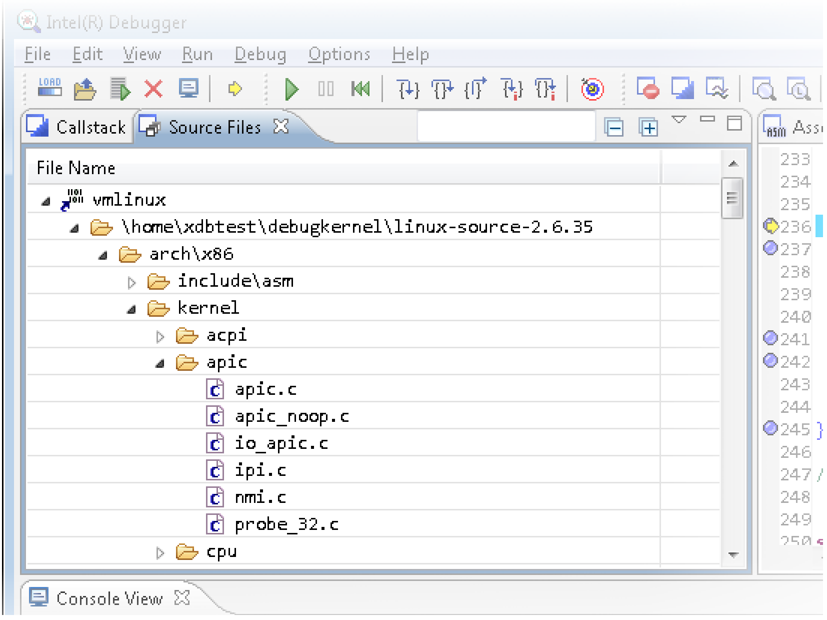
## Use Case: Navigating and Browsing Source Files

The ITP Source Debugger is a debugger, not a programmer's text editor.

This means that instead of choosing an arbitrary source file from the file system to view, you instead choose a file referenced by the loaded debug information.

The source files described by the debug information can be viewed in the **Source Files** window that is accessible from the **View** menu.

The files are displayed in a tree view that corresponds to the file system paths encoded into the debug information; double-click to open a file.

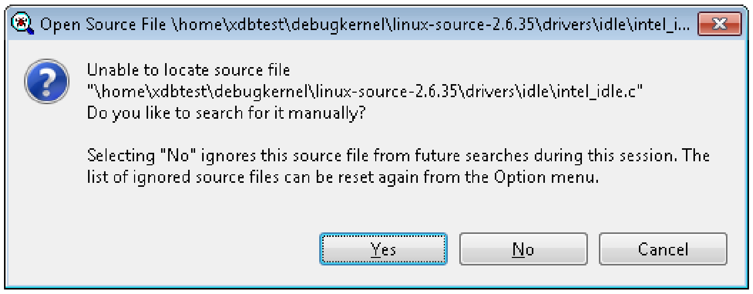


```
vmlinux
  \home\xdbtest\debugkernel\linux-source-2.6.35
    arch\x86
      include\asm
      kernel
        acpi
        apic
          apic.c
          apic_noop.c
          io_apic.c
          ipi.c
          nmi.c
          probe_32.c
        cpu
```

Figure 4: Source Files tab in the GUI

## Use Case: Specifying Source Directories and Replacement Rules

In the case where the source files on the host system do not match the paths encoded in the debug information, the Source Debugger will be unable to immediately load the source file. Instead, you will get a message similar to the one at the right:



Open Source File \home\xdbtest\debugkernel\linux-source-2.6.35\drivers\idle\intel_i...

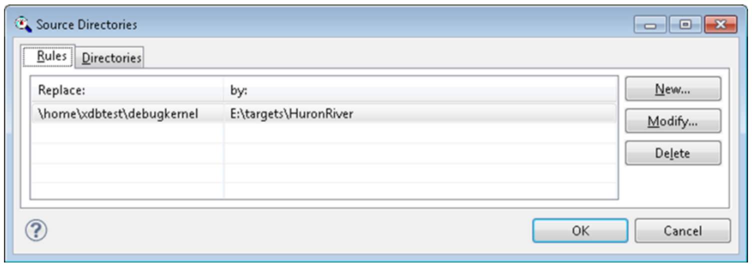
Unable to locate source file
"\home\xdbtest\debugkernel\linux-source-2.6.35\drivers\idle\intel_idle.c"
Do you like to search for it manually?

Selecting "No" ignores this source file from future searches during this session. The list of ignored source files can be reset again from the Option menu.

Click **Yes** to browse to the correct file, select it, and automatically create a **Replacement Rule** that allows the debugger to find all other files that have a similar base path.

You can also manually specify both Replacement Rules and folders that the debugger should search.

Use **Options→Source Directories…** to open the **Source Directories** window to specify either folders to search (non-recursive) or path replacement rules (i.e., change c:\mybuildfiles to e:\mydebugfiles).



| Replace: | by: |
|---|---|
| \home\xdbtest\debugkernel | E:\targets\HuronRiver |

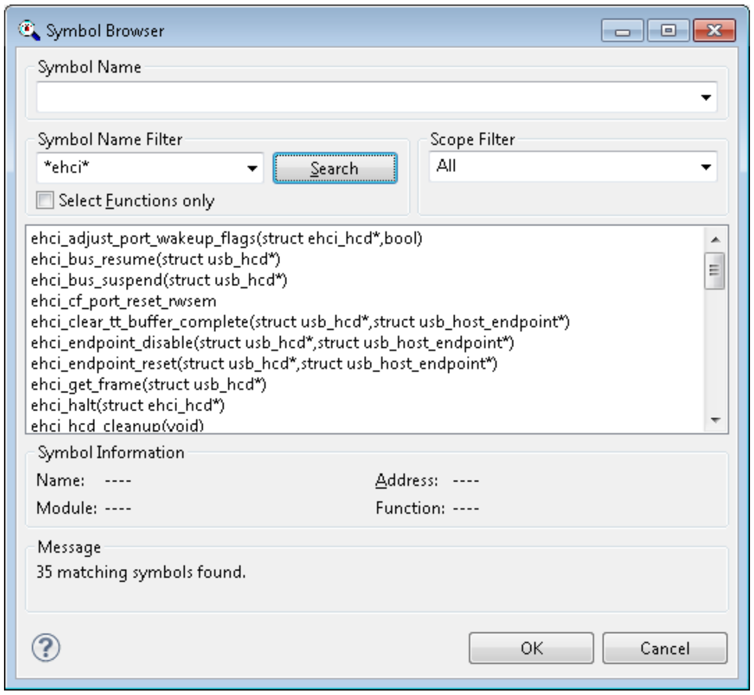
Figure 5: "Source Directories" Dialog

### Use Case: Navigating and Browsing Debug Symbols

The Source Debugger keeps an internal list of all **Symbols** such as function names, variable names and types.

This list of symbols can be searched using the **Symbol Browser**; for example, to look for a particular function or variable:

1. Launch the symbol browser using the **Debug→Symbol Browser…** menu item.



Symbol Name Filter: *ehci*
Scope Filter: All

```
ehci_adjust_port_wakeup_flags(struct ehci_hcd*,bool)
ehci_bus_resume(struct usb_hcd*)
ehci_bus_suspend(struct usb_hcd*)
ehci_cf_port_reset_rwsem
ehci_clear_tt_buffer_complete(struct usb_hcd*,struct usb_host_endpoint*)
ehci_endpoint_disable(struct usb_hcd*,struct usb_host_endpoint*)
ehci_endpoint_reset(struct usb_hcd*,struct usb_host_endpoint*)
ehci_get_frame(struct usb_hcd*)
ehci_halt(struct ehci_hcd*)
ehci_hcd_cleanup(void)
```

Name: ----
Address: ----
Module: ----
Function: ----

35 matching symbols found.

Figure 6: The Symbol Browser Dialog

## Using Breakpoints

Breakpoint utilization in a low-level debugger requires more understanding of the breakpoint mechanisms than when using a high-level application debugger. This section explains some of the subtleties of breakpoints in the Source Debugger and how to use them most effectively; then, explains how to access the breakpoint feature from the GUI.

For more information, see Troubleshooting Breakpoints in the Troubleshooting appendix.

### User-Specified Breakpoints vs. Implicit Breakpoints

The Source Debugger allows users to explicitly set **User-specified Breakpoints** in familiar ways, for example, right-clicking in the Source/Assembler GUI and selecting **Create Breakpoint**.

In addition, the Source Debugger may set breakpoints on behalf of the user for other functions, e.g., the **Step Out** function traverses the callstack, sets a breakpoint in the containing function, runs the target until it hits that breakpoint; then, removes the breakpoint. The user has no specific awareness of this breakpoint having been set and then deleted.  Even though this is not explicitly set by the user any debugger settings and restrictions on usage of breakpoints still apply.

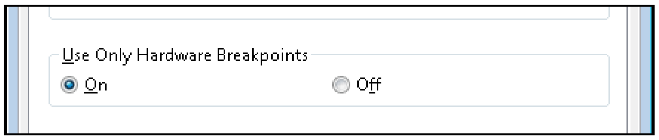
## Hardware vs. Software Breakpoints

By default, the Source Debugger attempts to use software breakpoints as they are more scalable in many-thread systems and they also are an unlimited resource.
**Software breakpoints** require inserting a special opcode at the desired break location in memory; therefore they can only be used to halt execution in a writable memory region. They also cannot be used for non-execution breaks (e.g. break on a memory access).
**Hardware breakpoints** are dependent on the debug registers (DRx) and hence limited to 4 on x86 architecture.

In some cases, the Source Debugger may elect to promote a software break to a hardware break; for example:

- When setting a software break in read-only memory (such as flash) and the operation fails, the Source Debugger detects this failure and automatically sets a hardware break instead. A warning to the user is printed.

The user can override the default usage of software breaks by selecting the **Options→Options…** menu item and clicking **On** for **Use Only Hardware Breakpoints**.



Use Only Hardware Breakpoints
[✓ On] Off

**Figure 7: Configuring the Debugger to always use Hardware Breakpoints**

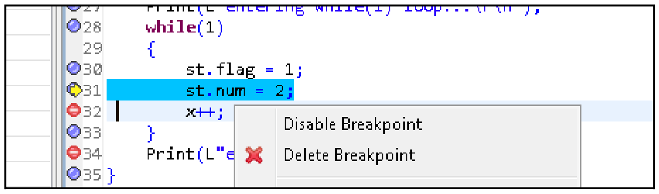
**Note**: Hardware breakpoints are a limited resource and using all available resources may degrade high-level language features such as **Step Out**, **Go Here**, etc. (but always with an appropriate warning to the user on the console).

## Setting Breakpoints from the Assembly or Source Window

Both the Assembly and Source Code Windows allow you to set breakpoints directly.

However, due to code optimizations and debug symbol limitations not all locations in your high-level code are *breakpointable*.

To add/disable/delete Breakpoints directly in the Source or Assembly window, right-click the desired line to display a context menu, as shown at the right:

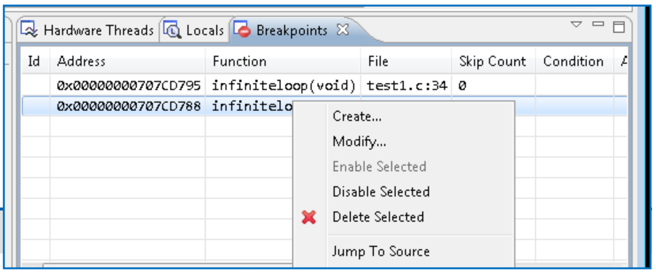


```
    while(1)
    {
        st.flag = 1;
        st.num = 2;
        x++;
    }
    Print(L"e
}
```

**Figure 8: Adding a breakpoint using the Source or Assembler Window**

## Setting Breakpoints from the Breakpoints Window

The **Breakpoints Window** allows you to create, remove and modify breakpoints independent of the Source or Assembly window.



| Id | Address | Function | File | Skip Count | Condition |
|---|---|---|---|---|---|
| | 0x000000000707CD795 | infiniteloop(void) | test1.c:34 | 0 | |
| | 0x000000000707CD788 | infinitelo | | | |

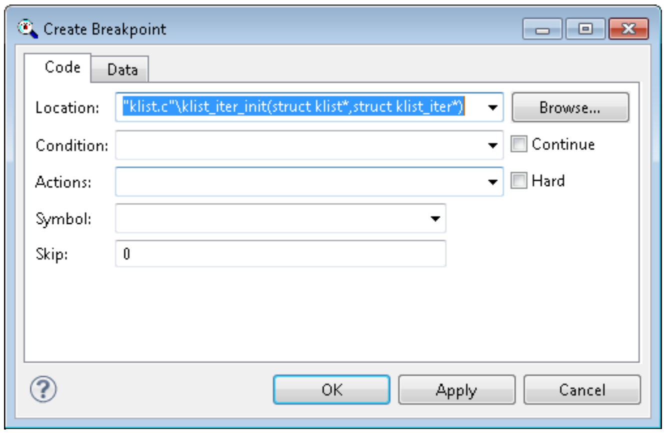
**Figure 9: Accessing Breakpoints via the Breakpoints Window**

To access these features from the window's context menu, right-click the desired line as shown at the right:

Right-click and select **Create…** to launch the **Create Breakpoint** dialog box to enter a breakpoint address or a debug symbol that evaluates to an address.

This dialog allows specification of additional parameters such as skip count, conditions, etc.

The **Data** tab allows you to specify Data Breakpoints, including Port IO breakpoints.



Location: "klist.c"\klist_iter_init(struct klist*,struct klist_iter*)
Condition:
Actions:
Symbol:
Skip: 0

**Figure 10: The Create Breakpoint Dialog**

## Debugging EFI BIOS

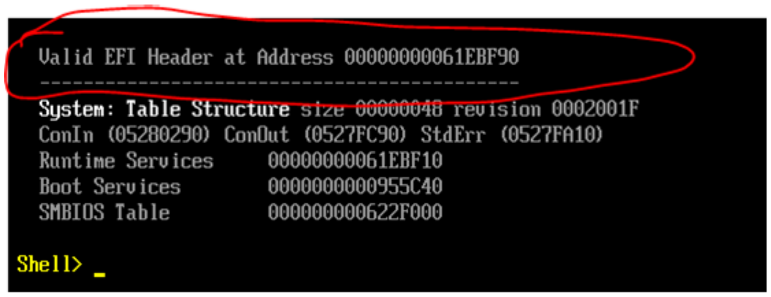
Because the EFI environment uses re-locatable code modules, the address of a code module must be supplied to the debugger when executing the **load** command. Because this address is usually not known to the end user, some method must be used to locate the modules in memory.

The Source Debugger provides two general methods for locating code modules in memory:

- Identify the specific module located at a certain memory address, e.g., at the instruction pointer.
- List all modules that are known by the EFI runtime and allow the user to choose a specific module to load symbols for.

The first class of methods relies only on the module of interest. The debugger can scan memory near the given address and attempt to locate the module header which contains sufficient data to load symbols.

The second class of methods involves locating an EFI data structure in memory (or flash) and enumerating the list of loaded modules described there. Therefore, it is dependent on a-priori knowledge of certain data structures such as the EFI System Table Pointer or the Flash Volume.



```
Valid EFI Header at Address 00000000061EBF90
---------------------------------------------
System: Table Structure size 00000048 revision 0002001F
ConIn (05280290) ConOut (0527FC90) StdErr (0527FA10)
Runtime Services      00000000061EBF10
Boot Services         00000000000955C40
SMBIOS Table          000000000622F000

Shell> _
```

Normally the EFI Table address can be obtained by booting to the EFI shell and typing the **mem** command. Normally for a given system and given BIOS and no change in memory configuration one can safely assume this address to be constant between boots.

In this case, 0x61EBF90 is the EFI Table address.

### Use Case: Identify/load symbols for a module at a certain address

To load symbols for a specific address, the easiest method is to use the **loadthis** command. This command tries to identify the module at the indicated address and loads symbols for it. If no address is provided, the current instruction pointer is used.

This method is not really EFI-specific. It detects any PE/COFF image in target memory and is only dependent on the debug information compiled into the PE/COFF header and not on any other EFI data structures.

```
xdb> efi "loadthis"
INFO: Software debugger set to: efi64 - EFI/PI compliant BIOS (64-bit mode)
INFO: Using DRAM search semantics, align=0x00001000 range=0x00100000
INFO: Searching backwards from 0x00000000809FB6C3 to 0x00000000808FB6C3 for PE/COFF header
INFO: Found PE/COFF module at 0x00000000809FB000 - 0x00000000809FF1C0 (size: 16832 bytes)
INFO: Loading debug symbols found at:
```

```
e:\dev.efi\work\Build\MdeModule\NOOPT_VS2008x86\X64\MdeModulePkg\Application\xdbefiutil\xd
befiutil\DEBUG\xdbefiutil.efi
```

The above example used the current instruction pointer as the search address; alternatively, you can supply a known address:

```
xdb> efi "loadthis 0xFFF98765"
INFO: Software debugger set to: efi32 - EFI/PI compliant BIOS (32-bit mode)
INFO: Using FLASH search semantics, align=0x00000004 range=0x00100000
INFO: Searching backwards from 0x00000000FFF98765 to 0x00000000FFE98765 for PE/COFF header
INFO: Found PE/COFF module at 0x00000000FFF97B04 - 0x00000000FFFA1A64 Entrypoint:
0x00000000FFF99E89 (size: 40800 bytes)
```

In the two examples above, we searched for both a module in RAM and a module in FLASH. The debugger used slightly different search semantics in each case. These can be configured, see efi help for more information.

## Use Case: List all loaded modules (DXE phase)

Listing all modules in the DXE phase requires that the debugger know the system table pointer. This can be specified manually using the **efi setsystab <addr>** command; then, verified by displaying the system table with **efi showsystab** which displays output similar to the following. Note that earlier we have shown how to get the system table address:

```
xdb> efi "setsystab 0xaf536f18"
xdb> efi showsystab
INFO: Software debugger set to: efi64 - EFI/PI compliant BIOS (64-bit mode)
INFO: Reading EFI_DEBUG_IMAGE_INFO table, this could take a little while...
EFI System table at 0x00000000AF536F18

Configuration Tables:
____________________________________________________________
GUID:                              Pointer:    Name:
GUID 05ad34ba, 6f02, 4214, {...}   0xae72bdb0  DXE_SERVICES_TABLE
GUID 7739f24c, 93d7, 11d4, {...}   0xaef17018  HOB_LIST
GUID 4c19049f, 4137, 4dd3, {...}   0xae72c7b0
GUID 49152e77, 1ada, 4764, {...}   0xae72d150  EFI_DEBUG_IMAGE_INFO_TABLE
GUID 8868e871, e4f1, 11d3, {...}   0xaf7fef98  EFI_ACPI_20_TABLE
GUID eb9d2d31, 2d88, 11d3, {...}   0xaf533218  SMBIOS_TABLE
<snip>
```

Alternatively, the debugger can search for a valid system table in a user-specified range:

```
xdb> efi "setsearchrange 0x78500000 0x786f0000"
INFO: Software debugger set to: efi32 - EFI/PI compliant BIOS (32-bit mode)
xdb> efi showsystab
INFO: Software debugger set to: efi32 - EFI/PI compliant BIOS (32-bit mode)
INFO: Searching backwards from 0x00000000786F0000 to 0x0000000078500000 for EFI System
Table...
INFO: Found valid EFI System Table Pointer at address: 0x00000000786DBF90
```

Once a valid system table is located, a list of all EFI modules can be displayed using **efi showmodules**:

```
xdb> efi "load Shell.efi"
INFO: Software debugger set to: efi64 - EFI/PI compliant BIOS (64-bit mode)
INFO: Using cached EFI State Information
INFO: Loading debug symbols found at:
```

```
e:\efi\work\Build\Shell\DEBUG_VS2008x86\X64\ShellPkg\Application\Shell\Shell\DEBUG\Shell.efi
```

## Additional EFI-specific Functions

Typing **efi help** displays a list of commands as shown below:

```
xdb> efi help
EFI debugger extension
Enter any of the following on the command line:
efi showconfig              - show debugger configuration
efi "setswmode <mode>"     - set the software mode of the debugger (see also showconfig)
efi showmodules             - print a list of all known modules
efi "load <modulename>"    - try to load symbols for the supplied module name (e.g.
mymodule.efi)
<snip>
```

**Note**: Multi-part commands must be quoted, e.g., efi “setsystab 0x1234” has quotes around the last two tokens.

## Debugging the Linux Kernel

Debugging the Linux Kernel using the Source Debugger is relatively straightforward as the kernel is a single monolithic executable and is always loaded at a fixed address.

### Use Case: Loading symbols for the Linux Kernel

To load kernel symbols, use the **File→Load/Unload Symbol File…** menu item to open the **Load** dialog and specify the **vmlinux** kernel image as the symbol file to load.

**Note**: The **Offset** parameter is left blank. This causes the debugger to use the offset that is encoded into the image to locate the executable code in target memory.

Once the symbol information is loaded, the debugger refreshes and attempts to display source code for the current execution point (if it is halted in kernel code).

### Use Case: Setting a breakpoint at the beginning of kernel initialization

In many cases you may want to stop target execution early during kernel initialization to walk up to some point of failure. The easiest way to do this is to set a hardware breakpoint on the **start_kernel** function in the Linux kernel. However, since the breakpoint needs to be installed before the kernel is loaded, use the following special steps:

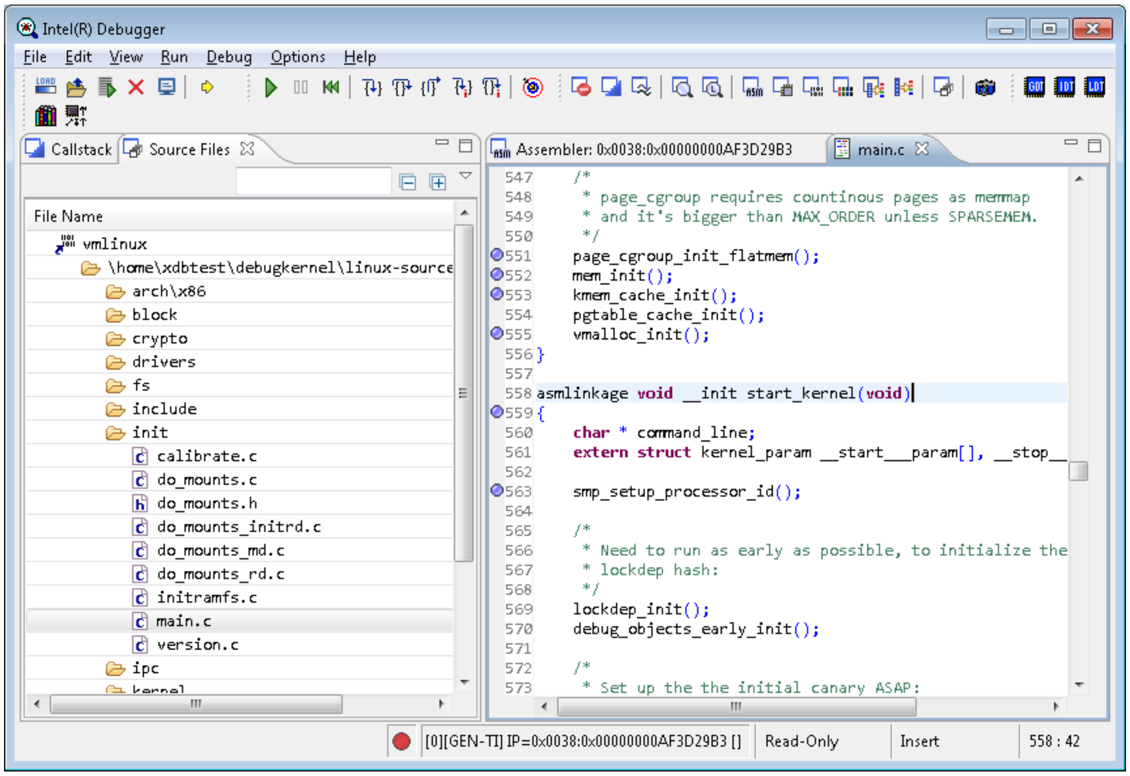
1. Stop the target prior to kernel execution.
2. Load the Linux Kernel symbols; then, locate the **start_kernel** function in the source.
3. Create a **hardware** breakpoint at the appropriate location.
4. Resume target execution; hit your breakpoint.

#### Stopping the target prior to kernel execution

To stop at the kernel initialization function, we must stop the target prior to that point so that we can install a breakpoint. The simplest way to do this is to reset the target and stop at the Reset Vector. Alternatively, you could boot the target to the OS boot loader or BIOS boot selection screen; then, stop target execution in the debugger.

#### Loading Linux Kernel symbols; then, locating the start_kernel function in the source

Use the **Load** command as described previously to load your kernel symbols; then, use the **Source Files** tab in the Debugger window to locate the **start_kernel** function in file **main.c**.


Intel(R) Debugger
File Edit View Run Debug Options Help
Callstack
Source Files
Assembler: 0x0038:0x00000000AF3D29B3
main.c

```
File Name
vmlinux
  \home\xdbtest\debugkernel\linux-source
    arch\x86
    block
    crypto
    drivers
    fs
    include
    init
      calibrate.c
      do_mounts.c
      do_mounts.h
      do_mounts_initrd.c
      do_mounts_md.c
      do_mounts_rd.c
      initramfs.c
      main.c
      version.c
    ipc
    kernel
```

```
	/*
	 * page_cgroup requires countinous pages as memmap
	 * and it's bigger than MAX_ORDER unless SPARSEMEM.
	 */
	page_cgroup_init_flatmem();
	mem_init();
	kmem_cache_init();
	pgtable_cache_init();
	vmalloc_init();
}

asmlinkage void __init start_kernel(void)
{
	char * command_line;
	extern struct kernel_param __start___param[], __stop__

	smp_setup_processor_id();

	/*
	 * Need to run as early as possible, to initialize the
	 * lockdep hash:
	 */
	lockdep_init();
	debug_objects_early_init();

	/*
	 * Set up the the initial canary ASAP:
```

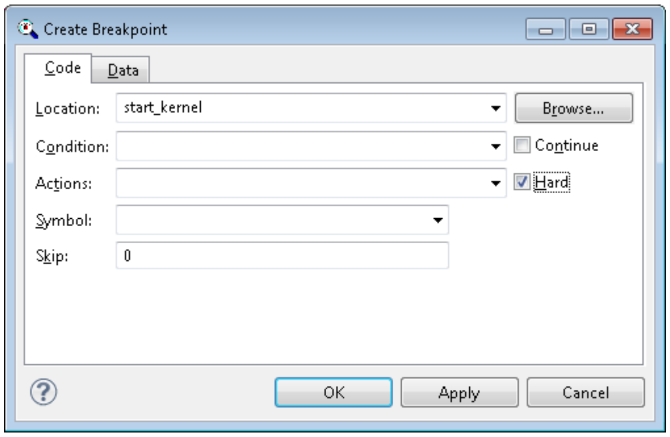
```
[0][GEN-TI] IP=0x0038:0x00000000AF3D29B3 []    Read-Only    Insert    558 : 42
```

Figure 11: “start_kernel” source code displayed in the debugger

## Creating a Hardware Breakpoint on the start_kernel function

Because we want to set a breakpoint on a function that has not yet been loaded into memory, we must use a Hardware Breakpoint.

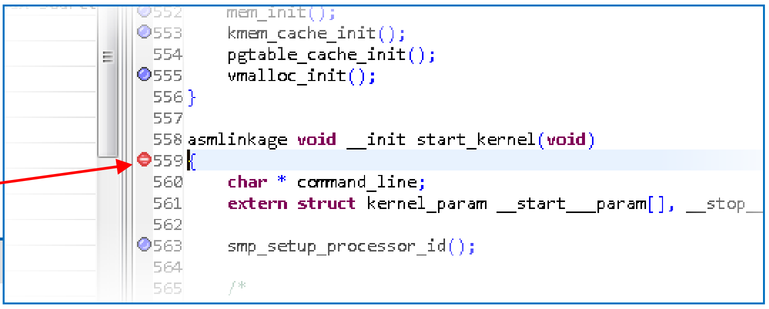
One way to do this is to use the **Debug→Create Breakpoint…** menu to open the **Create Breakpoint** dialog; then, specify the function name for the **Location** parameter, and select the **Hard** checkbox, as shown at the right:



| | |
|---|---|
| Location: | start_kernel |
| Condition: | |
| Actions: | |
| Symbol: | |
| Skip: | 0 |
| Continue | ☐ |
| Hard | ☑ |

Figure 12: Creating a hardware breakpoint on the start_kernel function

Click **OK**; then, confirm your breakpoint is set by verifying that a red circle has been placed in the margin of the **Source Window** at the start of your function, as



```
	mem_init();
	kmem_cache_init();
	pgtable_cache_init();
	vmalloc_init();
}

asmlinkage void __init start_kernel(void)
{
	char * command_line;
	extern struct kernel_param __start___param[], __stop_

	smp_setup_processor_id();

	/*
```

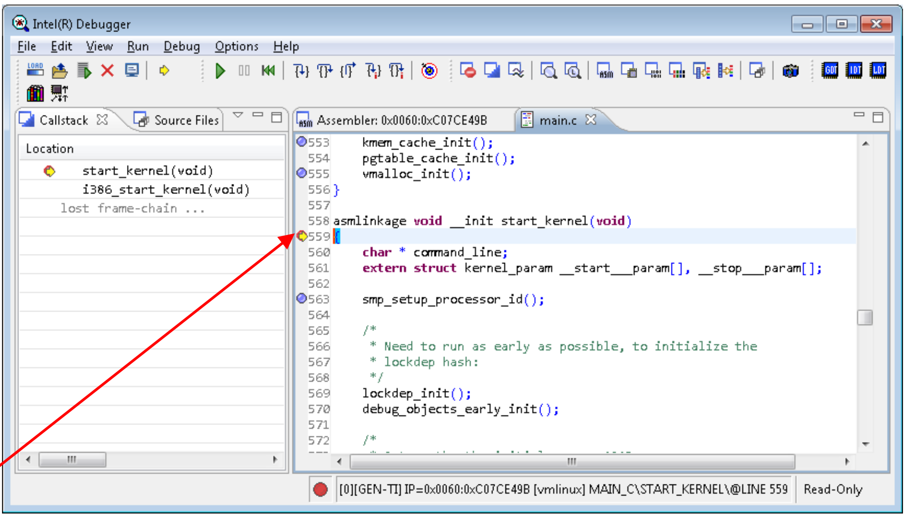
Figure 13: Breakpoint indicator in Source Window margin

shown at the right:

## Resuming target execution; hitting the breakpoint

After your breakpoint is set, you can resume normal target execution. The target should stop at the breakpoint you set.

To confirm this, view the callstack, or check for a yellow pointer in the Source Window. The pointer indicates the current execution point is at your breakpoint, as shown at the right:


Intel(R) Debugger
File Edit View Run Debug Options Help
Callstack
Source Files
Location
Assembler: 0x0060:0xC07CE49B
main.c
Read-Only

```
start_kernel(void)
i386_start_kernel(void)
lost frame-chain ...
```

```
    kmem_cache_init();
    pgtable_cache_init();
    vmalloc_init();
}

asmlinkage void __init start_kernel(void)
{
    char * command_line;
    extern struct kernel_param __start___param[], __stop___param[];

    smp_setup_processor_id();

    /*
     * Need to run as early as possible, to initialize the
     * lockdep hash:
     */
    lockdep_init();
    debug_objects_early_init();

    /*
```

```
[0][GEN-TI] IP=0x0060:0xC07CE49B [vmlinux] MAIN_C\START_KERNEL\@LINE 559
```

Figure 14: Execution stopped at the "start_kernel" function

## Debugging Linux Kernel Modules

Because Linux Kernel Modules are both relocatable and have an unconventional object file format, the standard Source Debugger **Load** command cannot be easily used to load debug information. Instead, it provides a feature for dynamically detecting the loading/unloading of kernel modules and performs the necessary logic to load debug symbols. This feature is dependent on a **Kernel Awareness** module; this is a Linux Kernel Module (xdbntf.ko) that must be compiled and loaded into the target system.

### XDBNTF Compilation

The source code for **xdbntf** is located in the debugger installation directory in the **\kernel-modules\xdbntf** folder. This folder should be copied to the build environment for the target system's kernel; then, built using the supplied **Makefile**. The resulting **xdbntf.ko** module should then be copied to the target system and loaded using the **insmod** command.

<Work in progress>

## Retrieving Execution Trace

### Introduction to Execution Trace

In order to debug issues, most engineers usually follow one of two workflows:

1- Backwards traversal:
   a. Determine the symptoms of the issue
   b. Place a breakpoint at the place where the symptoms become apparent
   c. Retrace the execution backwards by unwinding the call stack and checking callers

2- Forward traversal:
   a. Determine the hierarchy of function calls that lead to the symptom of the issue
   b. Place a breakpoint at a function that leads to the issue
   c. Single-step the execution until the symptoms are apparent

In some cases, however, looking at the call stack might not provide us with all the answers as the bug could have originated in a separate execution branch.

Forward traversal, on the other hand, might end up being a time-waster in the cases where you might end up stepping through dozens of executions of that patch of code until the issue manifests itself. Not to mention that forward traversal can prove destructive to issues like race conditions, deadlocks, etc…

This is where execution trace comes in. It allows you to place a breakpoint at the location where the symptoms become apparent and be able to see thousands, even millions, of code lines that executed up to this point.

## Different Types of Trace Methods

The Source Debugger supports multiple execution trace collection methods:

- LBR (Last Branch Record): Records the last 16 (8 on older processors) conditional branches. Limited, but widely supported.
- RTIT (Real Time Instruction Trace): Records up to several megabytes of conditional branch information. Able to provide millions of lines of instruction history. Only supported on a limited set of new processors (including, but not limited to , the Silvermont architecture)
- PT (Processor Trace): Successor to RTIT. Records up to gigabytes of conditional branch information. Able to provide millions of lines of instruction history. Still new, so only supported on a limited set of new processors (including, but not limited to, the Broadwell architecture)

## Opening the Trace Window

You can open the trace window by clicking on the "Execution Trace" toolbar item:

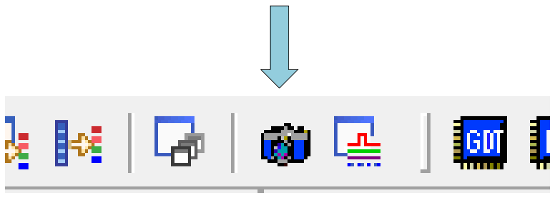
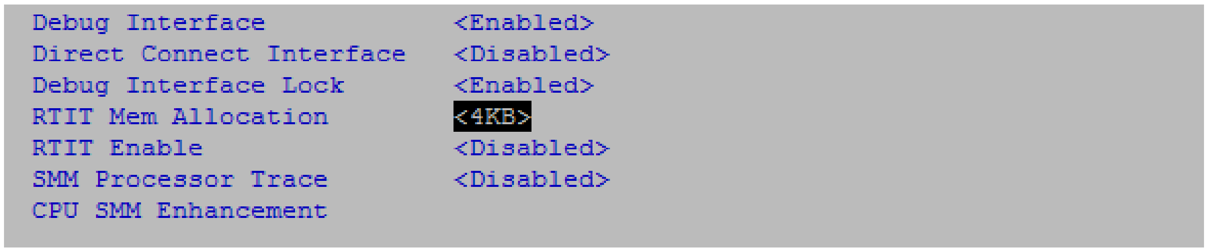


Figure 15: Execution Trace Toolbar Item

## Configuring RTIT/PT Trace Collection

Unlike LBR, PT and RTIT record trace data inside one or more buffers in the system memory. This allows for very large amounts of trace data. However it requires that some regions of memory be allocated specifically for tracing.

In most cases, the EFI BIOS is able to configure the trace parameters for you. Just locate the PT or RTIT memory allocation options, and tell the BIOS to automatically allocate some memory. The options should look like the following:



```
Debug Interface                <Enabled>
Direct Connect Interface       <Disabled>
Debug Interface Lock           <Enabled>
RTIT Mem Allocation            <4KB>
RTIT Enable                    <Disabled>
SMM Processor Trace            <Disabled>
CPU SMM Enhancement
```

**Figure 16: Example RTIT Memory Allocation Options**

However, in some rare cases, the BIOS might fail to allocate this memory, or the option might not be available. In those cases, you can easily edit the trace configuration using the **Trace Configuration** dialog. To open the dialog, click on the **Configure Trace** toolbar item in the trace window:

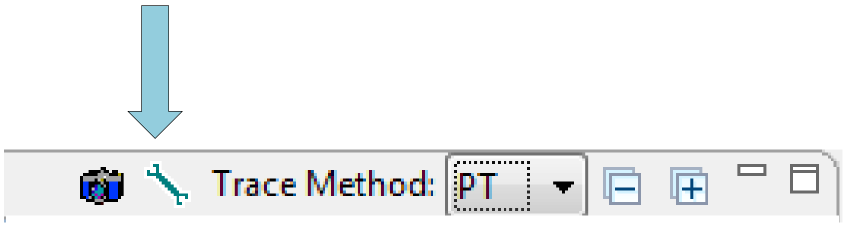
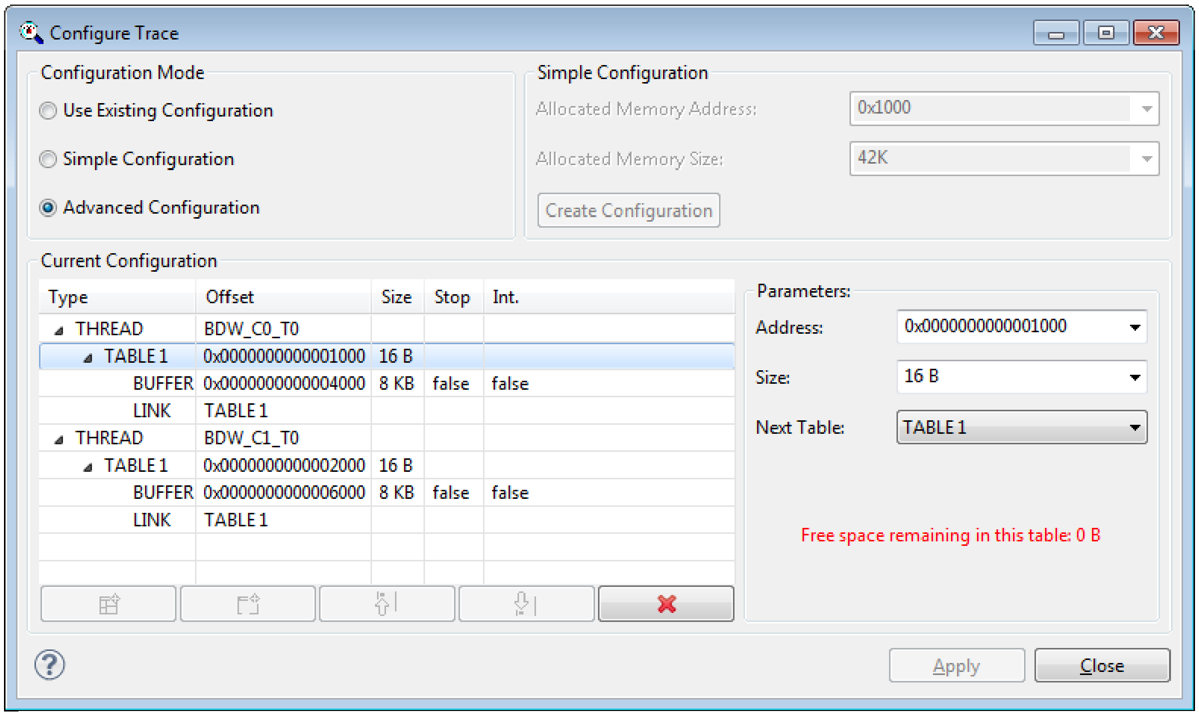


**Figure 17: Configure Trace Toolbar Item**

This should open up a dialog similar to Figure 18. Upon launching, the dialog will display the existing trace configuration. You are then able to easily investigate and edit the configuration.



Configure Trace

Configuration Mode
- Use Existing Configuration
- Simple Configuration
- [✓ Advanced Configuration]

Simple Configuration
Allocated Memory Address: 0x1000
Allocated Memory Size: 42K
Create Configuration

Current Configuration

| Type | Offset | Size | Stop | Int. |
|---|---|---|---|---|
| THREAD | BDW_C0_T0 | | | |
| TABLE 1 | 0x0000000000001000 | 16 B | | |
| BUFFER | 0x0000000000004000 | 8 KB | false | false |
| LINK | TABLE 1 | | | |
| THREAD | BDW_C1_T0 | | | |
| TABLE 1 | 0x0000000000002000 | 16 B | | |
| BUFFER | 0x0000000000006000 | 8 KB | false | false |
| LINK | TABLE 1 | | | |

Parameters:
Address: 0x0000000000001000
Size: 16 B
Next Table: TABLE 1

Free space remaining in this table: 0 B

Apply Close

**Figure 18: Trace Configuration Dialog**

The dialog offers two modes for editing the trace configuration: A simple mode, and an advanced mode. The simple mode should satisfy the needs of most users as all they need is to set up some buffers for tracing and get going.

In order to set up trace buffers the simple way, all you need is a region of memory that you know to be unused (or that you've allocated yourself):

1. Select the **Simple Configuration** mode
2. Input the address of the memory region (the dialog expects addresses to be provided in hexadecimal)
3. Input the size of the memory region (the size can be expressed in human readable format. E.g.: 1.5MB)
4. Press the **Create Configuration** button
5. The dialog calculates an appropriate distribution of buffers and displays the generated configuration
6. If you are satisfied with the generated configuration, you can hit **Apply** and start debugging
7. If not, you can use the **Advanced Configuration** mode to fine-tune the parameters

For advanced users, the **Advanced Configuration** mode allows them to fine-tune the trace collection parameters:

1. Select the **Advanced Configuration** mode
2. Select the processor thread that is of interest to you
3. Depending on the processor that you are debugging, you might or might not be able to add **TABLE**s to a thread. Tables allow you to easily group multiple trace collection buffers. Each table contains a **LINK** to the next table that the processor should start using once this table's buffers are consumed. If a table links back to itself, the processor will keep overwriting this table's buffers. This behavior can be used to create some neat tricks in advanced cases (e.g.: if a user is only interested in the beginning and the end of a long trace and does not want those sections to be overwritten, they can create two tables: Table1 and Table2, where Table1 links to Table2 but Table2 links back to itself. This forces the processor to only overwrite the second table, leaving the first one intact. Therefore the contents of Table1 would have the beginning of the trace, while Table2 would have the end of it)
4. Depending on the processor that you are debugging, you might or might not be able to add multiple **BUFFER**s to a table. The availability of options for buffers is also dependent on the CPU type. If an option is not available/disabled, that means that your processor does not support it. Buffers can offer two parameters: **Stop**, and **Interrupt**. If checked, the former instructs the processor to stop tracing once this buffer is full. The latter informs the processor to send a PMI (Power Management

Interrupt) once the buffer is full. This interrupt can be used as a trigger to halt the target.

5. Once you are satisfied with the configuration, you can hit **Apply** and start [ab]using other parts of the debugger

## Tracing Code

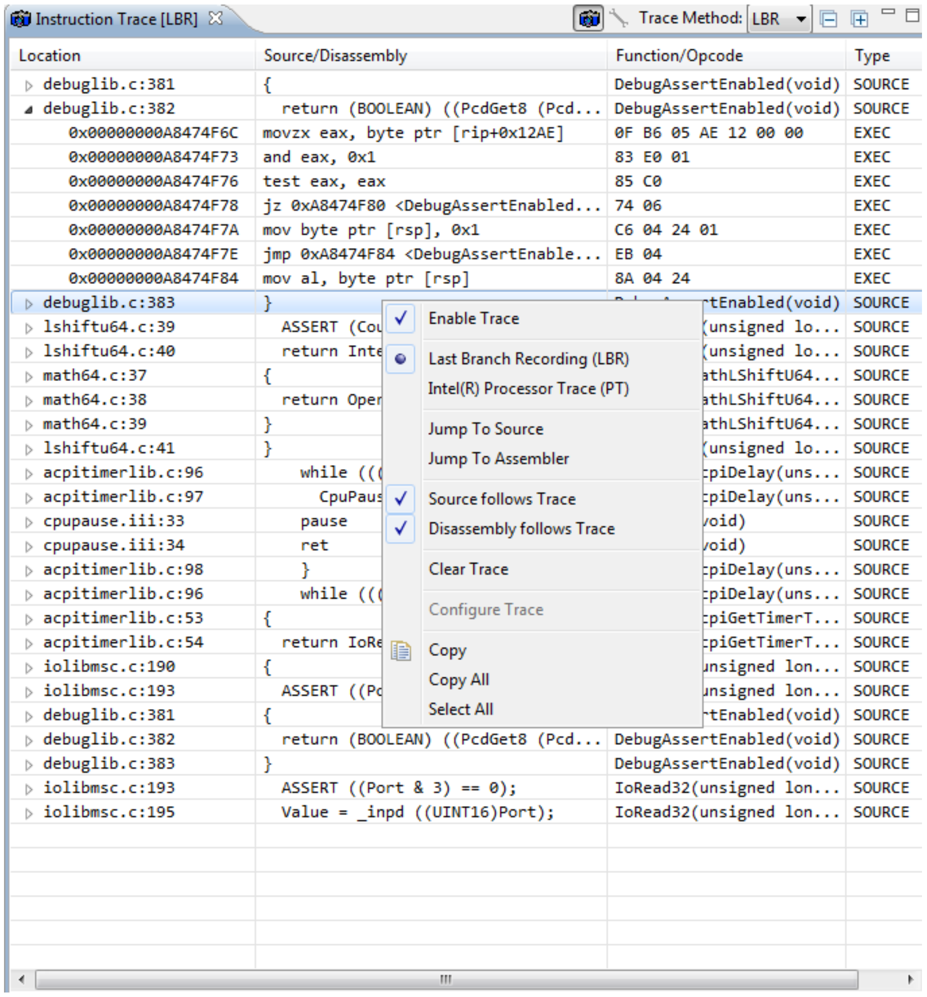
Now that you have selected a tracing method and (optionally) configured it, you are ready to start tracing code. Run to the place where you want to start your trace, and enable tracing by clicking on the **Enable Trace** button in the trace window toolbar.

At this point, you can start executing code. The trace window will be updated every time execution stops.

## Trace Window Perks

The trace window offers some nice functionalities that aid in analyzing the collected trace information. These features can be accessed by right-clicking on a trace entry (see Figure 19). Using this context menu, the user can:

- Enable/Disable trace
- Switch the trace method
- Jump to the source line that corresponds to this specific trace entry
- Jump to the assembler instruction that corresponds to this specific trace entry
- Access the **Trace Configuration** dialog
- Clear the trace buffers
- Source/Disassembly Follows Trace: These two options can be used to keep the source and/or disassembler views in sync with what is currently selected in the trace window. This allows the user to walk through the trace as it goes through source lines

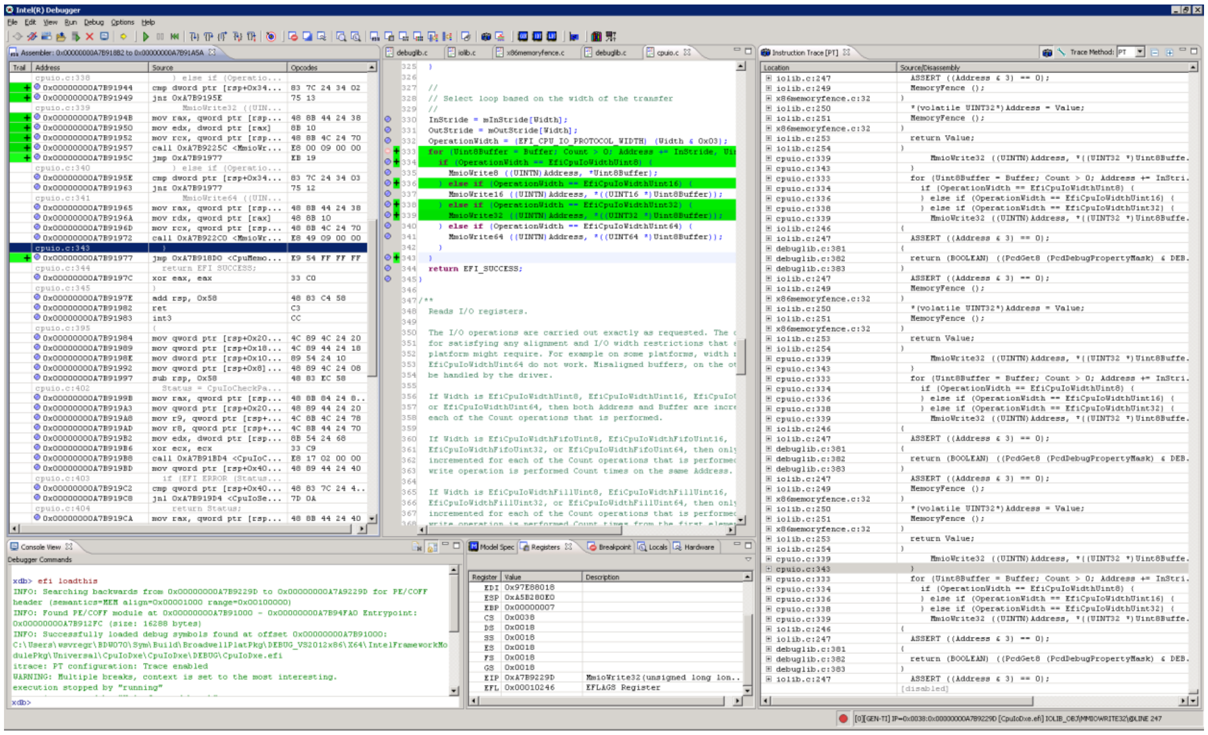


| Location | Source/Disassembly | Function/Opcode | Type |
|---|---|---|---|
| debuglib.c:381 | { | DebugAssertEnabled(void) | SOURCE |
| debuglib.c:382 | return (BOOLEAN) ((PcdGet8 (Pcd... | DebugAssertEnabled(void) | SOURCE |
| 0x00000000A8474F6C | movzx eax, byte ptr [rip+0x12AE] | 0F B6 05 AE 12 00 00 | EXEC |
| 0x00000000A8474F73 | and eax, 0x1 | 83 E0 01 | EXEC |
| 0x00000000A8474F76 | test eax, eax | 85 C0 | EXEC |
| 0x00000000A8474F78 | jz 0xA8474F80 <DebugAssertEnabled... | 74 06 | EXEC |
| 0x00000000A8474F7A | mov byte ptr [rsp], 0x1 | C6 04 24 01 | EXEC |
| 0x00000000A8474F7E | jmp 0xA8474F84 <DebugAssertEnable... | EB 04 | EXEC |
| 0x00000000A8474F84 | mov al, byte ptr [rsp] | 8A 04 24 | EXEC |
| debuglib.c:383 | } | tEnabled(void) | SOURCE |
| lshiftu64.c:39 | ASSERT (Cou | (unsigned lo... | SOURCE |
| lshiftu64.c:40 | return Inte | (unsigned lo... | SOURCE |
| math64.c:37 | { | athLShiftU64... | SOURCE |
| math64.c:38 | return Oper | athLShiftU64... | SOURCE |
| math64.c:39 | } | athLShiftU64... | SOURCE |
| lshiftu64.c:41 | } | (unsigned lo... | SOURCE |
| acpitimerlib.c:96 | while ((( | :piDelay(uns... | SOURCE |
| acpitimerlib.c:97 | CpuPaus | :piDelay(uns... | SOURCE |
| cpupause.iii:33 | pause | void) | SOURCE |
| cpupause.iii:34 | ret | void) | SOURCE |
| acpitimerlib.c:98 | } | :piDelay(uns... | SOURCE |
| acpitimerlib.c:96 | while ((( | :piDelay(uns... | SOURCE |
| acpitimerlib.c:53 | { | :piGetTimerT... | SOURCE |
| acpitimerlib.c:54 | return IoRe | :piGetTimerT... | SOURCE |
| iolibmsc.c:190 | { | unsigned lon... | SOURCE |
| iolibmsc.c:193 | ASSERT ((Po | unsigned lon... | SOURCE |
| debuglib.c:381 | { | tEnabled(void) | SOURCE |
| debuglib.c:382 | return (BOOLEAN) ((PcdGet8 (Pcd... | DebugAssertEnabled(void) | SOURCE |
| debuglib.c:383 | } | DebugAssertEnabled(void) | SOURCE |
| iolibmsc.c:193 | ASSERT ((Port & 3) == 0); | IoRead32(unsigned lon... | SOURCE |
| iolibmsc.c:195 | Value = _inpd ((UINT16)Port); | IoRead32(unsigned lon... | SOURCE |

**Figure 19: Trace Window Context Menu**

## But Wait, There's More

The latest release of the Source Debugger can highlight source lines and instructions that were executed recently according to trace information. This allows you to visually distinguish recent code from "old" code. It also allows you to detect which branches of an "if" statement or a "switch" were most recently taken (See Figure 20)



```
Trail  Address              Source                         Opcodes
       cpuio.c:338              } else if (Operatio...
    +  0x00000000A7B91944   cmp dword ptr [rsp+0x34...     83 7C 24 34 02
    +  0x00000000A7B91949   jnz 0xA7B9195E                 75 13
       cpuio.c:339                MmioWrite32 ((UIN...
    +  0x00000000A7B9194B   mov rax, qword ptr [rsp...     48 8B 44 24 38
    +  0x00000000A7B91950   mov edx, dword ptr [rax]       8B 10
    +  0x00000000A7B91952   mov rcx, qword ptr [rsp...     48 8B 4C 24 70
    +  0x00000000A7B91957   call 0xA7B9225C <MmioWr...     E8 00 09 00 00
    +  0x00000000A7B9195C   jmp 0xA7B91977                 EB 19
       cpuio.c:340              } else if (Operatio...
       0x00000000A7B9195E   cmp dword ptr [rsp+0x34...     83 7C 24 34 03
       0x00000000A7B91963   jnz 0xA7B91977                 75 12
       cpuio.c:341                MmioWrite64 ((UIN...
       0x00000000A7B91965   mov rax, qword ptr [rsp...     48 8B 44 24 38
       0x00000000A7B9196A   mov rdx, qword ptr [rax]       48 8B 10
       0x00000000A7B9196D   mov rcx, qword ptr [rsp...     48 8B 4C 24 70
       0x00000000A7B91972   call 0xA7B922C0 <MmioWr...     E8 49 09 00 00
       cpuio.c:343            }
    +  0x00000000A7B91977   jmp 0xA7B918D0 <CpuMemo...     E9 54 FF FF FF
       cpuio.c:344            return EFI_SUCCESS;
       0x00000000A7B9197C   xor eax, eax                   33 C0
       cpuio.c:345          }
       0x00000000A7B9197E   add rsp, 0x58                  48 83 C4 58
       0x00000000A7B91982   ret                            C3
       0x00000000A7B91983   int3                           CC
       cpuio.c:395          {
       0x00000000A7B91984   mov qword ptr [rsp+0x20...     4C 89 4C 24 20
       0x00000000A7B91989   mov qword ptr [rsp+0x18...     4C 89 44 24 18
       0x00000000A7B9198E   mov dword ptr [rsp+0x10...     89 54 24 10
       0x00000000A7B91992   mov qword ptr [rsp+0x8]...     48 89 4C 24 08
       0x00000000A7B91997   sub rsp, 0x58                  48 83 EC 58
       cpuio.c:402            Status = CpuIoCheckPa...
       0x00000000A7B9199B   mov rax, qword ptr [rsp...     48 8B 84 24 8..
       0x00000000A7B919A3   mov qword ptr [rsp+0x20...     48 89 44 24 20
       0x00000000A7B919A8   mov r9, qword ptr [rsp+...     4C 8B 4C 24 78
       0x00000000A7B919AD   mov r8, qword ptr [rsp+...     4C 8B 44 24 70
       0x00000000A7B919B2   mov edx, dword ptr [rsp...     8B 54 24 68
       0x00000000A7B919B6   xor ecx, ecx                   33 C9
       0x00000000A7B919B8   call 0xA7B91BD4 <CpuIoC...     E8 17 02 00 00
       0x00000000A7B919BD   mov qword ptr [rsp+0x40...     48 89 44 24 40
       cpuio.c:403            if (EFI_ERROR (Status...
       0x00000000A7B919C2   cmp qword ptr [rsp+0x40...     48 83 7C 24 4..
       0x00000000A7B919C8   jnl 0xA7B919D4 <CpuIoSe...     7D 0A
       cpuio.c:404              return Status;
       0x00000000A7B919CA   mov rax, qword ptr [rsp...     48 8B 44 24 40
```

```
  }

  //
  // Select loop based on the width of the transfer
  //
  InStride = mInStride[Width];
  OutStride = mOutStride[Width];
  OperationWidth = (EFI_CPU_IO_PROTOCOL_WIDTH) (Width & 0x03);
+ for (Uint8Buffer = Buffer; Count > 0; Address += InStride, Ui
+   if (OperationWidth == EfiCpuIoWidthUint8) {
      MmioWrite8 ((UINTN)Address, *Uint8Buffer);
+   } else if (OperationWidth == EfiCpuIoWidthUint16) {
      MmioWrite16 ((UINTN)Address, *((UINT16 *)Uint8Buffer));
+   } else if (OperationWidth == EfiCpuIoWidthUint32) {
+     MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffer));
    } else if (OperationWidth == EfiCpuIoWidthUint64) {
      MmioWrite64 ((UINTN)Address, *((UINT64 *)Uint8Buffer));
    }
+ }
  return EFI_SUCCESS;
}

/**
  Reads I/O registers.

  The I/O operations are carried out exactly as requested. The c
  for satisfying any alignment and I/O width restrictions that a
  platform might require. For example on some platforms, width r
  EfiCpuIoWidthUint64 do not work. Misaligned buffers, on the ot
  be handled by the driver.

  If Width is EfiCpuIoWidthUint8, EfiCpuIoWidthUint16, EfiCpuIoW
  or EfiCpuIoWidthUint64, then both Address and Buffer are incre
  each of the Count operations that is performed.

  If Width is EfiCpuIoWidthFifoUint8, EfiCpuIoWidthFifoUint16,
  EfiCpuIoWidthFifoUint32, or EfiCpuIoWidthFifoUint64, then only
  incremented for each of the Count operations that is performed
  write operation is performed Count times on the same Address.

  If Width is EfiCpuIoWidthFillUint8, EfiCpuIoWidthFillUint16,
  EfiCpuIoWidthFillUint32, or EfiCpuIoWidthFillUint64, then only
  incremented for each of the Count operations that is performed
```

```
Location                 Source/Disassembly
iolib.c:247                ASSERT ((Address & 3) == 0);
iolib.c:249                MemoryFence ();
x86memoryfence.c:32      }
iolib.c:250                *(volatile UINT32*)Address = Value;
iolib.c:251                MemoryFence ();
x86memoryfence.c:32      }
iolib.c:253                return Value;
iolib.c:254              }
cpuio.c:339                    MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffe.
cpuio.c:343                }
cpuio.c:333                for (Uint8Buffer = Buffer; Count > 0; Address += InStri.
cpuio.c:334                  if (OperationWidth == EfiCpuIoWidthUint8) {
cpuio.c:336                  } else if (OperationWidth == EfiCpuIoWidthUint16) {
cpuio.c:338                  } else if (OperationWidth == EfiCpuIoWidthUint32) {
cpuio.c:339                    MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffe.
iolib.c:246              {
iolib.c:247                ASSERT ((Address & 3) == 0);
debuglib.c:381           {
debuglib.c:382             return (BOOLEAN) ((PcdGet8 (PcdDebugPropertyMask) & DEB.
debuglib.c:383           }
iolib.c:247                ASSERT ((Address & 3) == 0);
iolib.c:249                MemoryFence ();
x86memoryfence.c:32      }
iolib.c:250                *(volatile UINT32*)Address = Value;
iolib.c:251                MemoryFence ();
x86memoryfence.c:32      }
iolib.c:253                return Value;
iolib.c:254              }
cpuio.c:339                    MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffe.
cpuio.c:343                }
cpuio.c:333                for (Uint8Buffer = Buffer; Count > 0; Address += InStri.
cpuio.c:334                  if (OperationWidth == EfiCpuIoWidthUint8) {
cpuio.c:336                  } else if (OperationWidth == EfiCpuIoWidthUint16) {
cpuio.c:338                  } else if (OperationWidth == EfiCpuIoWidthUint32) {
cpuio.c:339                    MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffe.
iolib.c:246              {
iolib.c:247                ASSERT ((Address & 3) == 0);
debuglib.c:381           {
debuglib.c:382             return (BOOLEAN) ((PcdGet8 (PcdDebugPropertyMask) & DEB.
debuglib.c:383           }
iolib.c:247                ASSERT ((Address & 3) == 0);
iolib.c:249                MemoryFence ();
x86memoryfence.c:32      }
iolib.c:250                *(volatile UINT32*)Address = Value;
iolib.c:251                MemoryFence ();
x86memoryfence.c:32      }
iolib.c:253                return Value;
iolib.c:254              }
cpuio.c:339                    MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffe.
cpuio.c:343                }
cpuio.c:333                for (Uint8Buffer = Buffer; Count > 0; Address += InStri.
cpuio.c:334                  if (OperationWidth == EfiCpuIoWidthUint8) {
cpuio.c:336                  } else if (OperationWidth == EfiCpuIoWidthUint16) {
cpuio.c:338                  } else if (OperationWidth == EfiCpuIoWidthUint32) {
cpuio.c:339                    MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffe.
iolib.c:246              {
iolib.c:247                ASSERT ((Address & 3) == 0);
debuglib.c:381           {
debuglib.c:382             return (BOOLEAN) ((PcdGet8 (PcdDebugPropertyMask) & DEB.
debuglib.c:383           }
iolib.c:247                ASSERT ((Address & 3) == 0);
                         [disabled]
```

```
xdb> efi loadthis
INFO: Searching backwards from 0x00000000A7B9229D to 0x00000000A7A9229D for PE/COFF
header (semantics=MEM align=0x00001000 range=0x00100000)
INFO: Found PE/COFF module at 0x00000000A7B91000 - 0x00000000A7B94FA0 Entrypoint:
0x00000000A7B912FC (size: 16288 bytes)
INFO: Successfully loaded debug symbols found at offset 0x00000000A7B91000:
C:\Users\wsvregr\BDW070\Sym\Build\BroadwellPlatPkg\DEBUG_VS2012x86\X64\IntelFrameworkMo
dulePkg\Universal\CpuIoDxe\CpuIoDxe\DEBUG\CpuIoDxe.efi
itrace: PT configuration: Trace enabled
WARNING: Multiple breaks, context is set to the most interesting.
execution stopped by "running"
xdb>
```

| Register | Value | Description |
|---|---|---|
| EDI | 0x97E88018 | |
| ESP | 0xA5B280E0 | |
| EBP | 0x00000007 | |
| CS | 0x0038 | |
| DS | 0x0018 | |
| SS | 0x0018 | |
| ES | 0x0018 | |
| FS | 0x0018 | |
| GS | 0x0018 | |
| EIP | 0xA7B9229D | MmioWrite32(unsigned long lon.. |
| EFL | 0x00010246 | EFLAGS Register |

```
[0][GEN-TI] IP=0x0038:0x00000000A7B9229D [CpuIoDxe.efi] IOLIB_OBJ\MMIOWRITE32\@LINE 247
```

Figure 20: Source Highlighting Example

The way this works is that whenever new execution trace data is available (either via LBR or RTIT/PT), the source viewer and assembly window automatically highlight the execution of the last 128 source lines or instructions. Lines that have been executed more than once in the last 128 lines are marked with a black "plus" sign.

The coloring can be disabled, simple, or full. The screenshot shows the assembler window using the "simple" version, where the highlighting is only shown next to the line, and the source window using the "full" mode where the color covers the whole line (the options can be accessed via **Options** -> **GUI Preferences**)

The colors range from dark green (very recent execution) to very faded and light green (old execution).

Future releases of XDB will allow the user to customize the coloring in even greater detail.

# Appendix A: Interoperability with other ITPII client applications

The ITPII DAL service architecture is designed to manage the target state while providing multiple concurrent clients access to that state. The Source Debugger is only one potential client; other clients may be attached to the DAL and may access the same target state as the Source Debugger. This can lead to a condition where another client has modified some state (e.g., a memory region or register contents) and the Source Debugger has no knowledge of this, leading to stale data in the Source Debugger.

Note that “other client” in this case includes the ITP Python Console!

The Source Debugger has been designed to detect external state modifications and at minimum warn the user via prints to the console window. However, due to API limitations there are many scenarios where this detection is not possible. There are long-range plans to extend DAL APIs to provide the Source Debugger with additional notifications and many of these issues will be resolved at that point.

## Known Issues

Following is a list of known issues with regard to multi-client interoperability.

### Manualscans Control Variable

The Source Debugger utilizes the **manualscans** control variable to improve performance of batch operations when requesting target state such as registers and memory. XDB asserts **manualscans** on logical operations such as Halt, Step, memory access, register access and possibly others. Manualscans remain asserted until the Source Debugger has been idle for approximately 1 second; then, it is deasserted. If no user interaction occurs and no target run-control or power events occur then manualscans remains deasserted.

### Resetbreak Control Variable

The Source Debugger utilizes the **resetbreak** control variable when processing a Restart command that is issued from the Source Debugger GUI. If the user changes the state of Resetbreak to anything other than “break” then the operation of the “Restart” command is not guaranteed, and the user is also responsible for manually clearing and re-installing all breakpoints after a target reset.

### Breakpoint Management

The Source Debugger implements breakpoints using the DAL breakpoint mechanism. This is a shared pool of breakpoints for all clients.

- If other clients modify or delete breakpoints that have been set by the Source Debugger, then, the Source Debugger is unable to update its internal state to reflect the new breakpoint condition. Source Debugger prints a warning to the console in this case.

- If other clients set or modify new unique breakpoints, the Source Debugger is unaware of these breakpoints. If the execution stops at one of these breakpoints, Source Debugger reports **unknown breakpoint** to the console.

## Run Control Events

In scenarios where an external client causes a fast sequence of run control events, the Source Debugger may lose track of the target state. An example would be using the CLI to set a breakpoint on the next instruction; then, issuing a **go** command. In this case, the Source Debugger may not detect that the target ran and then **very quickly** stopped again. Consequently, the information displayed in the Source Debugger may be stale.

**Note**: This will **NOT** occur if all run-control commands are performed from within the Source Debugger.

## Register State Modification

The Source Debugger has limited ability to monitor modifications to registers. For now, only external writes to the general purpose registers are detected (e.g., RIP, RAX, etc.). If an external program modifies most other registers, then the information displayed in the Source Debugger may be stale.

# Appendix B: Optimizing Debugger Performance

The Source Debugger is a high-level C-language debugger, and it provides many of the features that are expected of such a debugger, including watch windows, memory windows, conditional breakpoints, etc.

Implementing these features requires significantly more target access than in a purely silicon-oriented debugger such as ITP. In turn, there is a performance impact since transporting this data across the JTAG interface, as well as the API between the Source Debugger and ITP, has a cost.

This appendix gives guidelines on how to improve the performance of the Source Debugger.

## Common Problem Areas

The most significant performance impact in the Source Debugger is from having many GUI panels open and visible during your debug session. Each element in each panel likely requires multiple target accesses to populate. This can quickly add up to hundreds of accesses and significantly slow the debugger.

### Reducing the number of open windows

The Source Debugger is intelligent in its management of GUI windows. Only the **currently visible windows** are updated; therefore, if you make use of the “tabbing” feature in the GUI and keep most windows in the background until you need them, the impact is greatly reduced while not completely disabling them in the GUI.

### Using evaluations rather than locals for the registers window

The **Locals** window is a handy way to automatically display the state of any program variables that are in-scope; however, some programs may have many variables in a given scope, or there may be compound types (e.g., structs) that require multiple target accesses to resolve.

If you do not need access to all scope variables, replacing the **Locals** window with an **Evaluation** window that only contains items that you actually want to monitor may improve performance.

Similarly, the various **Register** windows refresh all visible content. If you only want to monitor a single register, use the **sysreg(<registername>)** syntax to add this single register to an Evaluation window; for example:

1. Right-click in the **Evaluations** window.
2. Add **Evaluation…**
3. Enter **sysreg(EIP)** in the **Expression:** field.
4. Click **OK**.

## Many-thread Systems

Targets with many threads (e.g. enterprise servers and massively parallel coprocessors) present a challenge to a JTAG debugger in that the scaling of access to many logical threads is not cheap, and debugger operations that traditionally were applied symmetrically to all threads can now be measured in tens of seconds when applied to a system with hundreds of threads. For example on a typical Haswell-class server the simple act of halting all threads can take from 10-20 seconds. For interactive software debug this has terrible implications to usability. Fortunately there are ways to improve performance, but it requires some care on the part of the debugger user.

### Demand-based Save State

To improve performance on a many-thread system the debugger utilizes a technique called "Demand-Based Save State", this is enabled/disabled by the ITP control variable "savestateondemand". This flag causes the ITP software to only interrogate the CPU for a minimal amount of data, this improves interactive performance at the expense of "cheap" data availability. If the user (or an application such as a source debugger) requests data beyond what has already been "saved" then the deferred access will be performed to complete the request. This means that the performance impact is shifted from the initial "halt" to the point of actually requesting the data or performing a data-dependent operation such as setting a breakpoint.

**Recommendation:** No action is needed, the Debugger will enable this mode automatically when the thread count of the target warrants it; in this case the debugger also reduces the amount of data that is immediately visible to the user, and some GUI elements (such as the hardware-threads window) will display "click to refresh".

### Hardware/Software breakpoint impact on performance

Hardware breakpoints utilize per-thread resources that must be configured symmetrically across all threads. Software breakpoints, however, are inherently **global** in the sense that the memory modification to insert the breakpoint must only be done once and it will work for all threads.

Because of this difference, there is a severe performance penalty for using hardware breakpoints as it requires traversing all processor threads to install the breakpoint; on many-thread systems, this can take a significant amount of time. The problem is compounded if high-level run control operations are being performed (e.g., **Step Out**, **Step Over** or **Go Here**) since these operations implicitly set one or more breakpoints.

**Recommendation:** When possible software breakpoints should be used on many-thread systems to improve overall system performance.

## Symmetrical Run Control vs. Single-threaded Run Control

By default the ITP software is configured to go/halt on all threads, this is controlled by the "breakall" control variable. This has a significant performance cost on a many-thread system, both because entering/exiting probemode is expensive, and also because hardware breakpoints must be enabled on all threads prior to probemode exit.

**Recommendation:** In situations where the user knows there are no inter-thread dependencies (e.g. BIOS/EFI code) the ITP "breakall" variable can be set to "0", this will have a significant performance boost, but the user must understand the implications of this setting:

- The debugger will only go/halt on the currently selected thread.
- Other threads will remain in/out of probemode independent of actions on the current thread.
- Hardware breakpoints (both explicit and implicit) will only be set on the currently selected thread.

In practical terms this means the following:

- 90% of EFI/BIOS code can be debugged with better interactive performance by setting breakall=0
- MP scenarios in BIOS/EFI can also be debugged with breakall=0, but the user must manually install a breakpoint on the individual threads when necessary
- OS bootloaders and *early* kernel initialization can be debugged with breakall=0
- OS kernel *after* MP init and OS driver code should be debugged with breakall=1

In *all* cases setting breakall=1 should provide 100% correct operation, at the expense of performance.

# Appendix C: Troubleshooting Guide

## Module Load Issues

If the module successfully loads, you can see it in the **Source Files** window. If the current instruction pointer is within the module, then the source code window opens to the current line of code.

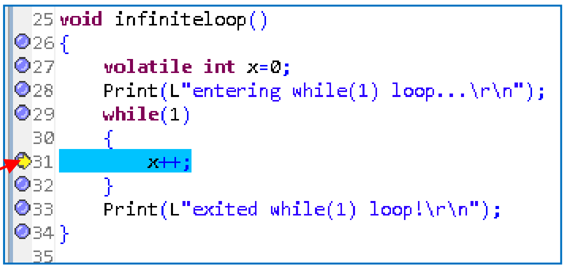
If your local file system paths do not match those included in the debug information (i.e., you are building and debugging on different systems), then you need to set source folders and/or replacement rules in order for the debugger to find the debug information and source files.

**When using the Microsoft Compiler**: Your executable (.efi or .exe) and the corresponding debug information (.pdb) must be in the same folder.

## Verifying the Module Offset Parameter

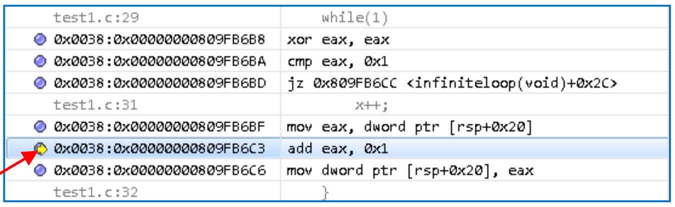
In general, you can verify that you have supplied the correct offset in the following ways:

- With a hard-coded breakpoint or while (1) loop at some known location, the debugger should automatically load the correct source file and you should see a yellow arrow pointing to the correct line of code, as shown at the right:



```
void infiniteloop()
{
    volatile int x=0;
    Print(L"entering while(1) loop...\r\n");
    while(1)
    {
        x++;
    }
    Print(L"exited while(1) loop!\r\n");
}
```

- Right-click on a line; then, select **Jump To Assembler** from the context menu to show assembly code that clearly correlates to your function and/or execution point, as shown at the right:



```
test1.c:29                        while(1)
0x0038:0x00000000809FB6B8   xor eax, eax
0x0038:0x00000000809FB6BA   cmp eax, 0x1
0x0038:0x00000000809FB6BD   jz 0x809FB6CC <infiniteloop(void)+0x2C>
test1.c:31                          x++;
0x0038:0x00000000809FB6BF   mov eax, dword ptr [rsp+0x20]
0x0038:0x00000000809FB6C3   add eax, 0x1
0x0038:0x00000000809FB6C6   mov dword ptr [rsp+0x20], eax
test1.c:32                        }
```

## Debugging/Troubleshooting Target Stability issues caused by the Source Debugger

The Source Debugger requires a great deal of data to construct a full source-level view of the target state. For a simple operation such as **step**, there can be tens to hundreds of individual accesses to target state. In the case where one of these accesses crashes the target, it can be difficult to know exactly what the cause was.

### Techniques for narrowing/isolating causes of target stability issues

- **Close as many GUI panels as possible**
  Each panel in the debugger (e.g., the registers panel, MSR panel, etc.) is self-updating. If many panels are open, then there are many target accesses. Closing panels is one way to isolate which functional group is causing the problem.

- **Use the TCI_LOG flag to turn on API logging**
  The debugger can log all transactions made to the ITP software. Although this information is primarily intended for developers, it may also be useful for a sophisticated user. Enable logging by setting the environment variable **TCI_LOG=1**.

  For example, you can edit the start_xdb.bat startup script to include the command set TCI_LOG=1.

  This technique causes a log file to be created in the **tcilog.txt** startup folder.

## Troubleshooting Breakpoints

The Source Debugger does not set breakpoints in the target directly; instead, it uses an abstract Breakpoint API provided by ITP. This means that the Source Debugger breakpoints co-exist with breakpoints set by other ITP clients (such as the PDT and the Python CLI). It also means that you can use the Python CLI to troubleshoot issues.

### Known Problem areas

Source Debugger breakpoints may not function correctly in the following cases. The Breakpoint is set in the Source Debugger and:

- A different ITP client application modifies or removes that breakpoint.
- An “external” reset such as power-cycle or a target-software-initiated reset occurs.
- A different ITP client application initiates a target reset.
- A different ITP client application modifies the “resetbreak” ITP control variable.

In most cases, as long as the user only interacts with the Source Debugger with regard to breakpoints, and the user does not change the “resetbreak” control variable, **and** the target platform has a reliable resetbreak implementation, the Source Debugger functions correctly.

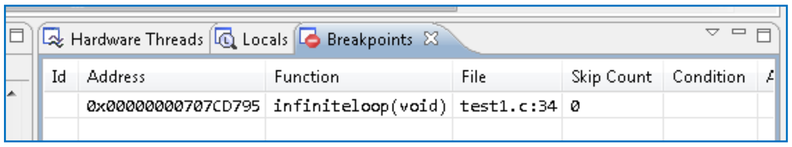
In all cases any problems should be recoverable by halting the target, disabling/removing all breakpoints using the Source Debugger GUI, and then re-applying the necessary breakpoints.

## Troubleshooting

If you believe a breakpoint is set in the Source Debugger (is shown as enabled in the Source Debugger GUI) and you are not hitting it when expected, the first step to verify that the breakpoint shown in the GUI matches what is shown in ITP.

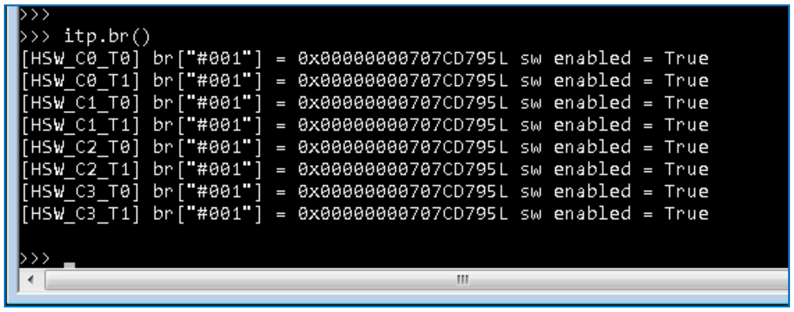
You do this using the ITP CLI **br()** command:

Breakpoint in the **Source Debugger GUI**:



| Id | Address | Function | File | Skip Count | Condition |
|---|---|---|---|---|---|
| | 0x00000000707CD795 | infiniteloop(void) | test1.c:34 | 0 | |

Equivalent breakpoint in **ITP CLI**:



```
>>>
>>> itp.br()
[HSW_C0_T0] br["#001"] = 0x00000000707CD795L sw enabled = True
[HSW_C0_T1] br["#001"] = 0x00000000707CD795L sw enabled = True
[HSW_C1_T0] br["#001"] = 0x00000000707CD795L sw enabled = True
[HSW_C1_T1] br["#001"] = 0x00000000707CD795L sw enabled = True
[HSW_C2_T0] br["#001"] = 0x00000000707CD795L sw enabled = True
[HSW_C2_T1] br["#001"] = 0x00000000707CD795L sw enabled = True
[HSW_C3_T0] br["#001"] = 0x00000000707CD795L sw enabled = True
[HSW_C3_T1] br["#001"] = 0x00000000707CD795L sw enabled = True

>>>
```

**Note**: A single logical breakpoint in the Source Debugger is replicated across all hardware threads in ITP.

If these two windows do not contain equivalent data, there is an issue with the Source Debugger. Please report this as a bug.

If both windows show equivalent data then there is either a bug in ITP, or there is a misunderstanding on the part of the user with regard to application of breakpoints.

# Appendix D: Miscellaneous useful hints

- The debugger console  does not support cut and paste (CTRL-C and CTRL-V) however it does support CTRL-C and then left click in front of the “xdb>” prompt and a right click will paste it.