Intel® JTAG Debugger
What you will learn from this slide deck

• Intel® JTAG Debugger for

  Linux*, Android* & Tizen™ Targets

• In-depth explanation of JTAG features for each development environment mentioned above

• Please see subsequent slide decks for in-depth technical training on other components
Intel® JTAG Debugger 2014

Accelerate system bring up and validation

Key Features

- New: JTAG debug for Intel® Core™, Xeon® & Quark SoC-based platforms
- New: Agent based UEFI debug
- JTAG system debug with event tracing (SVEN)
- Bitfield editor with full register description
- EFI/UEFI Firmware, bootloader debug, Linux* OS awareness
- Flashing and peripheral register support
- Access to page translation and descriptor tables
- Dynamically loaded kernel module debug
- LBR On-Chip instruction trace support, SMP run control support
Intel® JTAG Debugger 2014
Page Translation Table Insight into Memory Configuration

- Identify why memory access failed
- Complete register description & manipulation
- Instant and simple resolution & translation between virtual and physical address space

Fast issue tracking
- Bitfield Editor
- Edit registers
- Full register description

Memory address translation

Take control of memory configuration issues and memory leaks

Copyright © 2013, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Kernel & user mode

- Platform stability and start-up
- Firmware & bootloader
- Flash write & repair
- Operating system & driver
- Application debug
Where did things start to go wrong?

- Intel® Atom™ and 4th Gen. Intel® Core™ Processor supports Last Branch Record (LBR)
- Set breakpoint in OS signal event handler
- Unroll execution flow leading up to stack overflow or segmentation fault
- Follow execution backwards to where it deviated from expectation
- Re-run to that point and analyze memory accesses

Unroll past execution flow for fast issue resolution
SVEN - A Stethoscope for your System & SoC trace through JTAG

Trace Visualization
- Advanced navigation, search & filter
- Graphical and textual event display
- User controlled trace line grouping

Smart Event Triggers
- Live JTAG system debug with event tracing
- Smart breakpoints that interrupt execution on trace event calls
- Set smart breakpoints for in-depth analysis
  
  For example:
  - Break on any event from the USB driver
  - Break on any Debug String that starts with "ERROR"
  - Break if register X is accessed
  - Break if register X bits [7-9] have value 0b101

Enhance system stability through powerful JTAG & event tracing
Flash Memory Tool
- Broad flash type support - NOR and NAND
- Flash binary and hex files
- Mechanism to recover corrupted flash - Intel® Atom™ Processor CE5300
- Backup flash contents into binary file on host
- Erase/unlock/lock blocks
- Ideal for fast BIOS update

Easy to use flashing tool
- Multiple flash types
- NOR & NAND

Updating key system software stack components, made easy
<table>
<thead>
<tr>
<th>Phase</th>
<th>Component</th>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Verify, Debug &amp; Flash</strong></td>
<td>Intel® JTAG Debugger 2014¹</td>
<td>In-depth system and application debug</td>
<td>• In-depth debug insight into CPU, SoC and chipset for fast issue resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Leave trace instrumentation in production code for fast system-wide issue resolution</td>
</tr>
<tr>
<td></td>
<td>GDB* Debugger</td>
<td></td>
<td>• Detailed application debug and trace for fast issue resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Data race detection in parallel software</td>
</tr>
<tr>
<td></td>
<td>Intel® Inspector 2014 for Systems</td>
<td></td>
<td>• Increased productivity and code quality, and lowers cost, finds memory, threading, and security defects before they happen</td>
</tr>
<tr>
<td><strong>Tune</strong></td>
<td>Intel® VTune™ Amplifier 2014 for Systems</td>
<td>In-depth software analysis and tuning</td>
<td>• Fast in-depth analysis of SoC behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Remove guesswork, saves time, makes it easier to optimize for power efficiency and find performance optimization opportunities</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• In-depth analysis on resource limited targets</td>
</tr>
<tr>
<td><strong>Build</strong></td>
<td>Intel® C++ Compiler 14</td>
<td>Compiler and performance libraries</td>
<td>• Boost system performance for IA-based embedded designs and achieve scalability benefits of multicore and forward scale to many-core</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Cross platform development and integration into Eclipse for ease-of-use</td>
</tr>
<tr>
<td></td>
<td>Intel® Integrated Performance Primitives 8.1</td>
<td></td>
<td>• Speed up development &amp; performance with key software building blocks for signal, data, and media processing</td>
</tr>
<tr>
<td></td>
<td>Intel® Math Kernel Library 11.1</td>
<td></td>
<td>• GCC* compatibility</td>
</tr>
</tbody>
</table>

**Enhance power efficiency, system reliability, and boost performance with Intel® System Studio 2014**

¹ Optional component
Windows* Host Build and Debug Support with Eclipse* Integration

Windows* Host Eclipse* integration for SVEN, GDB and Intel® C++ Compiler
Supported OSs

**Host:**

- Red Hat Enterprise* Linux* 5, 6
- Ubuntu* 10.04 LTS, 12.04 LTS, **13.04**
- Fedora* 17, 18
- **Wind River** Linux* 4, 5
- openSUSE 12.1
- SUSE LINUX Enterprise Server* 11 SP2
- **Microsoft** Windows* 7,8

**Target:**

- Yocto Project* 1.3, 1.4, and newer based environment
- CE Linux* PR32 based environment
- **Tizen** IVI **1.0, 2.0**
- Wind River* Linux* 4, 5 based environment
- Android 4.0.x – 4.4.x
Intel® JTAG Debugger (XDB)
System Software Debug Solution

IPT Support

- EFI/UEFI Firmware Debug
- ELF Dwarf / PDB symbol info support
- Complex Software Breakpoints
- Source File Bookmarks
- Memory Layout and Page Table views
- Descriptor Table Views (GDT, LDT, IDT)
- Linux* OS awareness
  - Kernel Thread Views, Kernel Module Debug
- Hardware/Platform register access
- Hardware Threads
- LBR based Hardware trace capability support
- Advanced Scripting
- Flash Writer

System Software Debug with in-depth register and memory configuration awareness – from Firmware to OS
Debug Firmware and Bootloaders

- Debug SEC, PEI, DXE phase EFI debug support
- Source level debug for modules in flash, RAM and shadowed mode
- Tree view of firmware modules
- Automatic load of source files
- Step and set breakpoints as in any high level language debugger

JTAG assisted firmware and bootloader debug made easy
Agenda

HW requirements

Installation

Start-up

GUI features
  - Run control
  - Symbol handling
  - Breakpoints
  - Display features

Command line

BIOS & OS/OS driver debugging

Additional features & Summary
Hardware requirements

Target:
1. A supported Atom processor CPU
2. An access port for the JTAG probe:
   - Macraigor Usb2Demon™
   - Intel® ITP-XDP3
3. Memory for debugee
Installation

• Different flavours of the debugger depending on what target OS support you require. Example:
  – Vanilla 2.6 & 3.0 Linux kernel
  – Yocto 1.1/1.2 Embedded Linux kernel

• Debugger part of different tool packages

• Depending on the tool suite there might be restrictions on what host OS is required [Windows (XP, 7), Linux (Ubuntu/Fedora) 32/64 bit]

• Run the installation script [part of the tool suite installation]
Start-up

1. Make sure the probe is connected to target [and host]
2. Probe driver installed and loaded
3. Target running
4. Select debugger start-up script and run
What can go wrong?

Check for any Debugger Error message!

Some typical issues:

- On Linux host:
  - Debugger GUI requires JRE [version 1.6 or later]
  - Initial probe communication require `fxload` to be installed on host

- Are the rules file correctly installed

- Probe driver installed
**Initial GUI**

When the debugger GUI comes up it will automatically establish contact with the target.

Once contact is established the target is halted and memory read around the current instruction pointer.
GUI – target run control

Run/continue, single step, stop/halt and reset are basic features. Three ways of specifying any command:

- Using a control/function key combination on your keyboard (ex. F5 for run/continue)

- Writing the command in console window

- Using the GUI menu icons
Run control icons (& key-code)

- Run or Continue (F5)
- Suspend execution (Pause)
- Target reset (Shift + Ctrl + F5)
- Step program until next source line (F11)
- Step one source line proceeding through function calls (F12)
- Run until caller (Shift + F11)
- Instruction (asm) step into (F6)
- Instruction (asm) step over (F7)
Symbol handling

The debugger allows source level debugging (using symbols) provided that we have symbol information for the binary loaded in the target.
Loading the symbols

Browse to find the identical binary file as loaded on the target

Download = download to target. **Do not** tick this if You will debug SW already on the target.
How to find the sources

The debugger will try to find the source file which matches the current instruction pointer. If it cannot find the file it will ask you to do a manual search.

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.

[Image showing a debug window and a file browser]
Select source files to view

With the ‘Source Files’ icon you can open a window which contains the source tree as found in the binary file loaded in the debugger. Just double click on any of the source file names to open the file in the source window.
“You know the name of the source file”

Click on Open Source File icon and in the dialog which open:
- scroll to the file
- type the name of the file in the filter field
Evaluate symbols

Hover the cursor over a variable and the debugger will show you its value.

Highlight a variable and use the ‘right mouse click’ – additional options are now available.
### Local variables

The example above shows part of the ‘rq’ structure

- In expression field indicate that this is a compound element and that it has been Expanded (for not expanded)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>*rq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>raw_lock</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>spinlock</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>cpu_running</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>array [0 ... 4]</td>
<td></td>
</tr>
<tr>
<td>last_load_update_tick</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>skip_clock_update</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>inv_weight</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>nr_load_updates</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>nr_switches</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>cfs</td>
<td>cfs_rq</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>load_weight</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Local variable window will display all variables which are accessible from within current scope.
How did I reach the current location?

Function tree – bottom up
Only address presented when no debug information available

Source file: line number

Current file
Breakpoints

There are basically two types of breakpoints:
- Code **Breakpoints**
- **Watchpoints** or data breakpoints; also include I/O accesses

As Breakpoints are an essential feature for a debugger there are a number of ways of defining a breakpoint:
- Via menu entry dialog box
- In the source window
- From the Breakpoint window
Create a breakpoint

Minimum - code location is required.

**Hard** option allow you to set a breakpoint even if the memory location is not accessible.

For Watchpoints the location, access type and length are important.
Breakpoints cont.

Extract from source code:

```
tick nohz restart sched tick();
    preempt_enable_no_resched();
schedule();
    preempt_disable();
}
```

- Code breakpoint
- Data breakpoint
Display features - registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>EBX</td>
<td>0xC16BFD8</td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>EDX</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td>0xF50B9F84</td>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td>0xF50B9F8C</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>0x0060</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>0x007B</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td>0x006B</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td>0x007B</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td>0x00DB</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td>0xC100164B</td>
<td>cpu_idle(void) + 0x005B</td>
</tr>
<tr>
<td>EFL</td>
<td>0x0010202</td>
<td>EFLAGS Register</td>
</tr>
</tbody>
</table>

Ensure that the processor specific registers are enabled:
- Press Shift+Alt+F5

Model specific registers:
- Press Shift+Alt+F6

System controller registers:
- Press Shift+Alt+F7

Next slide
Example: System Controller Registers entry.

With many of the registers having individual bits with unique interpretations the programmer can use a bit field editor to inspect and change those values.
Vector registers

<table>
<thead>
<tr>
<th>xmm7</th>
<th>00000000051513630</th>
<th>xmm8</th>
<th>67e3e450c804b400c</th>
<th>xmm9</th>
<th>00000ee3f0c0c</th>
<th>xmm10</th>
<th>00000ee3f0c0c</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmm11</td>
<td>00000ee3f0c0c0c</td>
<td>xmm12</td>
<td>16347c2c36c11330101</td>
<td>xmm13</td>
<td>00000ee3f0c0c0c</td>
<td>xmm14</td>
<td>00000ee3f0c0c0c</td>
</tr>
<tr>
<td>xmm15</td>
<td>00000ee3f0c0c0c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector registers:
- xmm0: 5.014490e+37 1.626603e+00
- xmm1: -3.081979e-16 3.452680e-01
- xmm2: 0.0 1.87500e+00
- xmm3: 0.0 1.09035e+01
- xmm4: -3.596465e-07 -7.513350e-01
- xmm5: 8.648240e+13 2.407279e+02
- xmm6: 4.691425e-21 -1.988215e-04
- xmm7: 1.924282e+06 -3.248629e-01

Size options:
- Hexadecimal
- Signed Decimal
- Unsigned Decimal
- Octal
- Binary

Format options:
- INT8
- INT16
- INT32
- INT64
- FLOAT32
- FLOAT64

Copy options:
- Copy
- Copy All
- Select All
More display features - memory

As address you can specify:
- logical address
- linear address
- browse for a symbol
- physical address:
  phys(0xnnnnnnnnnnn)
What about paging?

Double click on a page table/directory entry and You will have the ability to modify the attribute bits with the bit field editor.
Virtual -> physical address mapping

Select an address and press Translate to find the address mapping. When you press OK the page directory/table involved will be shown in the Paging window.
For each entry in the descriptor tables you can view the important bits - excluding the address information (base + limit)
### Console View

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20h Range</td>
<td>INTGATE32</td>
<td>sel=6x0000 0ff=0xC14899A4 ti=GDT rpl=0 P=1 DPL=0</td>
</tr>
<tr>
<td>6</td>
<td>Invalid Opcode</td>
<td>INTGATE32</td>
<td>sel=6x0000 0ff=0xC14899C8 ti=GDT rpl=0 P=1 DPL=0</td>
</tr>
<tr>
<td>7</td>
<td>Device Unavailable</td>
<td>INTGATE32</td>
<td>sel=6x0000 0ff=0xC14899A4 ti=GDT rpl=0 P=1 DPL=0</td>
</tr>
<tr>
<td>8</td>
<td>Double Fault</td>
<td>TASKGATE</td>
<td>sel=6x08F8 P=1 DPL=0</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
<td>INTGATE32</td>
<td>sel=6x0000 0ff=0xC14899D4 ti=GDT rpl=0 P=1 DPL=0</td>
</tr>
<tr>
<td>10</td>
<td>Invalid TSS</td>
<td>INTGATE32</td>
<td>sel=6x0000 0ff=0xC14899E0 ti=GDT rpl=0 P=1 DPL=0</td>
</tr>
</tbody>
</table>

#### Modify Descriptor

- **New Values:**
  - Descriptor Type: 32-Bit Interrupt Gate
  - Segment Offset: 0xC14899A4
  - Segment Selector: 0x0060
  - Descriptor Layout (Inner 32 Bits):

- **Original Values:**
  - 32-Bit Interrupt Gate
  - 0xC14899A4
  - 0x0060

- **Use the GDT descriptor table for the destination selector.**

**Description:**

- IDT[7] at 0x00000000C167A038 32-bit Interrupt Gate
- Selector=0x0060 (GDT[12] RPL 0) offset=0xC14899A4
- P=1 DPL=0
- raw data=4996000008E48C1
Intel® Atom™ Processor supports Last Branch Record (LBR) using a set of internal registers.

Debugger can display the collected trace data – reconstructed execution flow.
Enable & Display trace

open the trace window with the icon 🎨
Enable / Disable trace by right mouse click in the trace window and select from the options.

--------------------- Run to collect data ---------------------

The collected data will be displayed in the open trace window
Multiple HW threads

The debugger can display how the multiple logical cores are used and indicate which logical core is used by the current code displayed. You can select a logical core and continue debugging the SW running there.
Command line usage

You can specify all the commands which we have looked at so far from command line. In the console window you find often a lot of text – here the debugger output error/warning messages and the equivalent command line for the GUI selections done. The bottom line in the console window is ‘reserved’ for your input [commands].

Example:

```
xdb> set break at start_kernel hard
BREAKPOINT 0 AT start_kernel (addr=0xC16C3420) : enabled (S=0,CS=0,HW=0)
xdb>
```
Generate & Run script files

You can record a debug session and later 'replay' it.
Useful for generating 'over-night-tests'

There are two options to generate a script file:
- 'Start Command Recording' will store all commands you make in the file [specified in the dialog which opens] until you select Stop Command Recording
- 'Start IO Recording' will also, in addition, collect any outputs from the debugger

To run any script file click on the icon and provide the data needed (min name of the script file) in the dialog which opens
EFI BIOS debugging

EFI environment uses relocatable code modules. To debug any of those modules you need to provide the address of the code to the debugger. There are a set of console commands to identify an efi-module and load corresponding symbols.

```
xdb> efi "loadthis"
INFO: Software debugger set to: efi64 - EFI/PI compliant BIOS (64-bit mode)
INFO: Using DRAM search semantics, align=0x0001000 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:
\dev\work\Build\MdeModule\NOOPT_VS2008x86\Y64\MdeModulePkg\Application\xdbefiutil\xdbefiutil
```

```
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
```

```
xdb> efi showmodules
INFO: Software debugger set to: efi64 - EFI/PI compliant BIOS (64-bit mode)
INFO: Using cached EFI State Information

<table>
<thead>
<tr>
<th>ModuleID</th>
<th>Base</th>
<th>Size</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0x00000000AE71C000</td>
<td>0x00013F51</td>
<td>DxeMain.efi</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
xdb> efi "load Shell.efi"
```
Debugging BIOS & Linux kernel

The Debugger is OS agnostic* – you can source level debug any code provided that you have the symbol information.

* With the exception that it is aware of Linux kernel threads which can be displayed by selecting the icon.
Debugging loadable modules

There is a Linux kernel module [idbntf] provided with the debugger which you can build and install – you then can load & debug Linux kernel modules [for example: loadable drivers]

Before driver loaded:

![Before driver loaded](image1)

After driver loaded – insmod sum3.ko:

![After driver loaded](image2)
Additional features: Modify target code on the fly

Powerful tool for on-the-fly testing and rerun of test and firmware code
• Open memory window or disassembly window
• The hex values can be modified – memory window

• Opcodes or mnemonics can be overwritten – disassembly window
Additional features cont.
Flashing Target Images and Bootloader

Intel® JTAG Debugger Flash Memory Tool
- Flash binary and hex files
- Erase/unlock/lock blocks
- Backup flash contents into binary file on host
- Ideal for BIOS update

Flash "select /board= 'target board'
Flash "change offset 0x00"
Flash "change data file '/home/qa/BIOS50.bin'
Flash "burn flash false true true"
SoC Trace – low overhead technology for static instrumentation of key SoC components

- Instrumented Software Modules of Devices & Systems
  - User
  - Device
  - Kernel
  - JTAG Device

- Event tracing with JTAG
- Smart breakpoints
- Interrupt execution on key events

- Event Recorder

- SVEN Event:
  - Timestamp
  - Payload

- Visualize, Analyze and Debug system event traces
  - Command line interface
  - Graphical interface
Additional features cont. SoC Trace support

This feature requires System Software support on target!

Trace Visualization
- Advanced navigation and search capabilities
- Graphical and textual event displays

Smart Event Triggers
- Live JTAG system debug with event tracing
- Smart breakpoints that interrupt execution on trace event calls
Summary

Intel® JTAG Debugger is a powerful tool which allows you to:

- Use GUI based source level debugging of BIOS, OS kernels and drivers
  - Bitfield editor supporting Atom™ based CPUs and SoCs
  - Flashing and peripheral register support
  - Access to page translation and descriptor tables
  - Dynamically loaded kernel module debug
  - On-Chip trace support
  - SMP run control support
  - Linux* awareness support
Summary / Next Steps
Intel® System Studio 2014

1. Windows* Host and Linux* Host Support with Eclipse* Integration
2. Support for Intel® Atom™ Processor E3xxx (Baytrail SoC)
3. Support for Tizen* IVI
4. Intel® C++ Compiler Improvements
   • Optimizations for latest Intel processors
   • improved sysroot and GNU cross-build integration support
   • Cross-build support for Windows* host and Wind River* Linux* target
5. Extended Intel® VTune™ Amplifier System-Wide Analysis
6. Yocto Project* Compatible
7. Intel® JTAG Debugger support for next generation processors of all sizes
8. Graphical installer for both Windows* host and Linux* host

http://intel.ly/system-studio
The next step in Intelligent Systems Software Development
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY
ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS
DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR
IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES
RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY
PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on
Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using
specific computer systems, components, software, operations and functions. Any change to any of
those factors may cause the results to vary. You should consult other information and performance
tests to assist you in fully evaluating your contemplated purchases, including the performance of that
product when combined with other products.

Copyright ©, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk
are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that
are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and
other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on
microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended
for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for
Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information
regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Intel® System Studio
JTAG Debugger
Agenda

- HW requirements
- GUI features
- OS/OS Driver debugging
- Summary
Hardware requirements

Target:
1. An Atom-based Phone/Tablet (Merrifield/Cherry Trail...)
2. Intel® ITP-XDP3
3. Debug card to connect between the phone and ITP
Start-up

1. Make sure the probe is connected to target [and host]
2. Probe driver installed and loaded (libusb & fxload)
3. Target running
4. Select debugger’s start-up script of your target and run

```
start_xdb_XDP3_Merrifield.sh
start_xdb_XDP3_CherryTrail.sh
```
Symbol handling

- The debugger allows source level debugging (using symbols) provided that we have symbol information for the binary loaded in the target.
With many of the registers having individual bits with unique interpretations the programmer can use a bit field editor to inspect and change those values.
Enable & Display trace

open the trace window with the icon

Enable / Disable trace by right mouse click in the trace window and select from the options.

------------- Run to collect data -----------------

The collected data will be displayed in the open trace window

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Function/Opcode</th>
<th>Source/Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCE</td>
<td>bitops.h:312</td>
<td>intel_idle(struct cpu) (addr[nr / BITS_PER_LONG]) != 0;</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>intel_idle.c:444</td>
<td>intel_idle(struct cpu) if (!need_resched()) {</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>processor.h:735</td>
<td>intel_idle(struct cpu) asm volatile(&quot;.byte 0x0f, 0x01, 0xc8;&quot;</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>intel_idle.c:446</td>
<td>intel_idle(struct cpu) smp_mb();</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>thread_info.h:84</td>
<td>intel_idle(struct cpu) return test_bit(flag, (unsigned long *</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>bitops.h:312</td>
<td>intel_idle(struct cpu) (addr[nr / BITS_PER_LONG]) != 0;</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>intel_idle.c:447</td>
<td>intel_idle(struct cpu) if (!need_resched())</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>processor.h:742</td>
<td>intel_idle(struct cpu) asm volatile(&quot;.byte 0x0f, 0x01, 0xc9;&quot;</td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>0xC153099F</td>
<td>B1 01 mov cl, 0x1</td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>0xC15309A1</td>
<td>8B 45 E0 mov eax, dword ptr [ebp-0x20]</td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>0xC15309A4</td>
<td>0F 01 C9 mwait</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>intel_idle.c:451</td>
<td>intel_idle(struct cpu) start_critical_timings();</td>
<td></td>
</tr>
<tr>
<td>BRANCH</td>
<td>0xC15309A7</td>
<td>E8 44 A9 D9 FF call 0xC12CB2F0 &lt;start_critical_timings&gt;</td>
<td></td>
</tr>
</tbody>
</table>
```
Debugging Android* kernel

The Debugger is OS agnostic* – you can source level debug any code provided that you have the symbol information.

* With the exception that it is aware of Android kernel threads which can be displayed by selecting an icon
Summary

- Intel® JTAG Debugger is a powerful tool which allows you to:
  - Use GUI based source level debugging of OS kernels and drivers
    - Bitfield editor supporting Atom™ based CPUs and SoCs
    - Access to page translation and descriptor tables
    - Dynamically loaded kernel module debug
    - On-Chip trace support
    - Android* awareness support
Installation

- **Windows***: Just use the exe installer!
- **Linux*** (Ubuntu***):

### Host Software Requirements

1. For installation of the Intel(R) JTAG Debugger, root or sudo root rights are required.
2. Linux* system running Ubuntu* 10.04 64bit.
3. libusb 0.1.12 or higher
4. fxload 0.0.20020411 or higher
5. Java runtime environment (JRE) 1.5 or newer to use the Eclipse* framework. In a web browser, access [www.java.com](http://www.java.com), and download and install JRE 1.6. Make sure that the $PATH environment variable contains the path to the JRE bin-directory.