



White Paper

**Robert Müller-Albrecht**  
Developer Products Division

# Device Driver Debugging on Intel® Atom™ processor based devices

Document Number: 319429-002US



# Introduction

One central aspect of developing a custom design platform based on the Intel® Atom™ processor or embedded applications is to enhance and customize the feature set of the underlying OS. This would be especially true for a closed system. Furthermore you may want to interface to specialized hardware platform extensions and write your own drivers for this purpose. Some of the multimedia codec optimizations may also be best applied on the device driver level.

The Intel® JTAG Debugger for Intel® Atom™ Processor offers the feature set to assist in OS adaptation and driver development as well as during the defect fix cycles and during validation and quality assurance. In this whitepaper we will focus on the methodology used to debug Linux OS bring up, Linux core components and runtime loaded kernel modules

## Overview

The Intel® JTAG Debugger for Intel® Atom™ Processor is included in the Intel® Embedded Software Development Tool Suite for Intel® Atom™ Processor only. It is a system debugger with a full graphical user interface. Its use currently requires the availability of an Intel® eXtended Debug Port (XDP) on the target device as well as an ITP-XDP3 Intel® In-Target-Probe. Support for additional 3<sup>rd</sup> party JTAG probes is planned for future releases.

The target audience for the Intel® JTAG Debugger are original equipment manufacturers and original design manufacturers (OxM), that require the capability of doing their own device driver development and low level OS kernel layer platform adaptations.

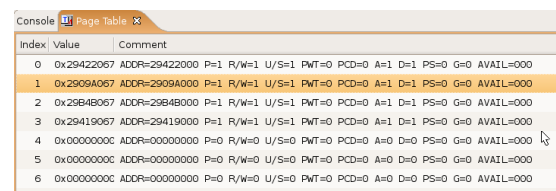
This specific class of software developers requires deep insight into the hardware the embedded OS is running on. At the same time developers should not have to give up the usability of a graphical user interface and the high level language support debug features they are used to.

Full Intel® Atom™ architecture support provides an in-depth view into the processor technology. It provides easy access to most Si specific features,

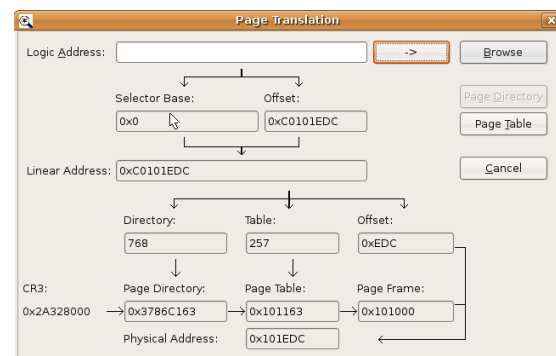
including architectural registers, Intel® SSE3 and graphics companion chip registers. Registers can be viewed and modified in Bitfield Editors that provide in depth and fully documented convenient access.

Bitfield Editors are not only available for standard registers, but also for descriptor table entries.

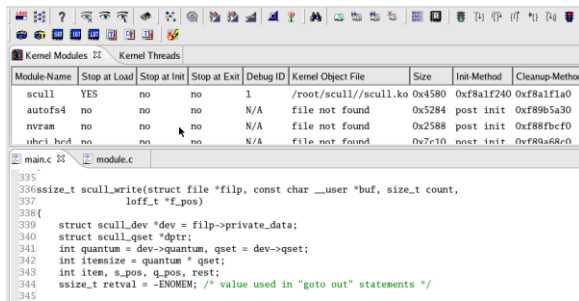
- Not only can the descriptor tables be easily viewed and modified for debugging purposes, but it is also possible to conveniently access the Page Translation Table and have the active memory mapping displayed in real time:



Index	Value	Comment
0	0x29422067	ADDR=29422000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
1	0x2909A067	ADDR=2909A000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
2	0x29B48067	ADDR=29B48000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
3	0x29419067	ADDR=29419000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
4	0x0000000C	ADDR=00000000 P=0 R/W=0 U/S=0 PWT=0 PCD=0 A=0 D=0 PS=0 G=0 AVAIL=000
5	0x0000000C	ADDR=00000000 P=0 R/W=0 U/S=0 PWT=0 PCD=0 A=0 D=0 PS=0 G=0 AVAIL=000
6	0x0000000C	ADDR=00000000 P=0 R/W=0 U/S=0 PWT=0 PCD=0 A=0 D=0 PS=0 G=0 AVAIL=000



- Execution Trace Support enhances the understanding of the flow of an executed program. It thus helps significantly with the isolation of memory leaks, data structure alignment and execution flow issues. Displaying execution trace for system debugging enables more effective debug cycles
- Linux OS Awareness for full understanding of the system behavior at all times. Display all relevant kernel information, active kernel threads and loaded kernel modules and debug them in the OS context.



This Debugger being based on the JTAG standard it provides direct hardware access and enables developers to access Si specific features independent of any software running on the target processor.

The extended hardware access and the extended OS awareness of this debugger also means that it is possible to debug dynamically loaded kernel modules (i.e. device drivers) remotely via JTAG, by simply launching a kernel module provided with the tool suite on that target. This dedicated kernel module exports all the module load events and memory locations so the JTAG debugger with it's OS awareness add-ons can pick this information up and allow for easy and convenient device driver debugging. Please read the release notes and debugger documentation closely for details.

## Rationale

Debugging device drivers and system services via JTAG poses additional challenges, because JTAG as such is target software agnostic. Since device drivers are commonly implemented as run-time loaded dynamic kernel modules, a mechanism needs to be implemented that exports the kernel module load and unload events to the debugger. The intent of this kernel level symbol export and event notification is to let the debugger monitor OS kernel symbols that contain memory load address information for the kernel module as well as information about the location of the kernel modules status, load and initialization methods.

There are multiple ways to achieve this. The traditional method is to require instrumentation of the kernel modules to be debugged. This is an invasive methodology and requires deactivation via define or removal of the debug instrumentation in the kernel module for release builds. This complicates analysis of defects or run-time problems that may only show up after the device driver has already been delivered to the end customer.

Another way is to require the application of a kernel patch to the OS kernel that triggers the Linux\* OS to export all the necessary information. The overhead to doing this is actually only a few Kbytes in memory and while no debugger is connected to query the export symbols, the runtime impact of these types of patches should literally be zero. There is however frequently concern about enabling debug hooks that make it potentially easier for competitors to analyze their code base. At least however this avoids having to do an instrumented build of your device driver code every time you want to debug it.

The best solution is to require neither kernel module instrumentation nor a kernel patch. The way to do this is to implement the kernel module load info export functionality as a kernel module itself. Thus you can take any target platform you have and without

recompiling the kernel or the debuggee you can start debugging your device drivers.

This whitepaper will provide a brief tutorial and usage scenario how this is done with the Intel JTAG Debugger for Intel® Atom™ Processor

## Building the kernel module for load info export.

To use the runtime loaded kernel module debugging feature that is part of the Linux\* OS awareness plugin in the Intel® JTAG Debugger you will need to have the kernel module **idbntf.ko** running and installed on the target device. The folder

**/opt/intel/atom/xdb/2.0.xxx/kernel-modules/idbntf**

contains code to generate the Linux\* kernel module that enables kernel module debugging. Simply run **make** in this folder to create the **idbntf.ko** kernel module using

**/opt/intel/atom/xdb/2.0.xxx/kernel-modules/idbntf/Makefile**

For generation simply transfer these files to your target system and invoke **make**. This will generate the kernel object **idbntf.ko**.

To enable module debugging this object has to be loaded prior to starting the debugger via the command **insmod idbntf.ko**. It will then export, initialization method and cleanup method load information from the target to the debugger on the host. After finishing the debug session, the module can be unloaded with **rmmod idbntf**.

## Launch Debug Session

1. First, in the shell environment, make sure that the XDB Debugger startup script located at

**/opt/intel/xdb/atom/2.0.xxx/bin/xdb.sh**

contains the following settings:

```
#!/bin/bash

# Intel(R) JTAG Debugger for Intel(R)
Atom(TM) Processor

# Copyright (C) 2000-2009 Intel
Corporation. All rights reserved.

INSTALLDIR=/opt/intel/atom/xdb/2.x.xx
x

export
LD_LIBRARY_PATH="$INSTALLDIR/lib:$INST
ALLDIR/gui:$INSTALLDIR/plugin/ia/lin
os:$INSTALLDIR/plugin/ia/trace:$INSTA
LLDIR/plugin/ia/flash":$LD_LIBRARY_PA
TH

export
PATH="$INSTALLDIR/gui:$INSTALLDIR/lib
":$PATH

export IDB_GUI_DEBUGGER="../lib/XDB"

dbggui -tgtype 'JTAG IA' -IUDGmode -
plg
'libguiialin.so,libiatrace.so,libplg-
flashw.so' -core 1 -target
device='XDP3'
scanchain='TargetPlatform' hotdebug
"$@"
```

If it doesn't contain these settings and the In-Target Probe eXtended Debug Port setting does not specify XDP3 or the designated target is not the processor you desire to debug or one of the shared object plug-ins for Linux OS awareness, or Flash Writing is missing please correct the settings in the **xdb.sh** script.

2. Check that the XDP3 JTAG probe is firmly connected to the eXtended Debug Port marked “JTAG” on the target board. Also check that the probe is connected to the USB cable and USB port on your lab computer.
3. Change to the debugger directory

```
chdir /opt/intel/stom/xdb/2.x.xxx
```
4. Launch the XDB Debugger

```
./xdb.sh
```
5. Check Linux Console Window on Connection Failure

## Debug the Linux Kernel

Once you are connected the JTAG interface will have taken over control and if you try to move the mouse on your target device or enter commands in a command shell it should not react – independent of whether you stopped directly after reset or whether your OS is already fully booted.

It is now time to load the symbol information for the Linux kernel.

You can do so by clicking on the “Load” button



in the debugger menu bar and navigate to the **vmlinux** kernel ELF/Dwarf2 file in the Linux kernel source tree that matches the Linux image running on the MID target.

Alternatively you may also type

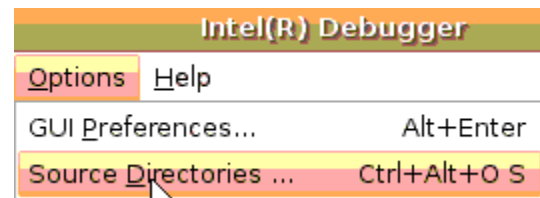
```
LOAD /SEGMENT /DEBUG /NOLOAD /GLOBAL OF  
"/usr/linux-2.6.31.i686/vmlinux"
```

into the console window. While the debugger is loading and interpreting this large OS kernel file it may seem unresponsive for a few moments.

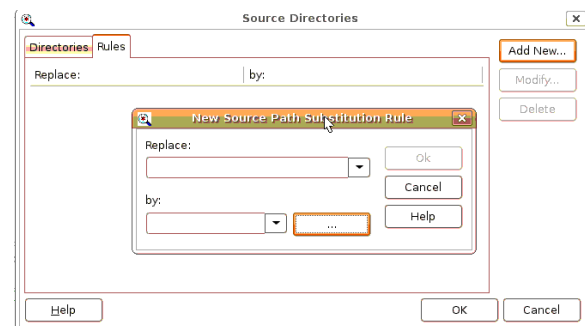
To map the paths for the location of the sources associated with the symbol info on the debug host system correctly, it is necessary to tell the debugger

where the top level directory of the source tree is located on the debug host system.

You may navigate to the the **Options** pulldown menu and select **Source Directories**.



After selecting the **Rules** tab and clicking on **Add New...** you will get to the dialog below, which allows you to enter the correct source mapping.



Alternatively in the debugger console window you could also enter a command of the form below, which equals what is outlined above as the GUI based source mapping approach.

```
SET DIRECTORY /SUBSTITUTE = ""/usr/linux-  
2.6.22.i686/"
```

Now we are done with making sure that we have full symbol debug capabilities for the Linux\* OS kernel enabled and can proceed debugging the kernel.

For the purpose of this whitepaper let us do so from the very beginning and start at the Linux kernel entry point.


To do so we shall first put the target processor back into reset. This can be done by issuing the command **restart** in the debugger console window.

Now we want to get to the point where the OS kernel is unpacked and the actual operating system boot process begins.

On Linux\* this entry point is usually called `start_kernel()`. Thus let us set a hardware breakpoint at the Linux kernel start-up. It has to be a hardware breakpoint because RAM memory is not fully configured this early in the boot process. To be on the safe side when it comes to single stepping through the boot code we should even go one step further and tell the debugger to treat all breakpoints as hardware breakpoints for now. To do this you would go to the debugger console window and enter

set option `/hard=on`

You can set the breakpoint using the breakpoint

dialog  or by issuing the following command in the console window:

set break at `start_kernel` hard

To run to the kernel entry point you may then simply

click on the  button or enter

run

in the debugger console window.

After running through the BIOS and the OS bootloader the target execution will halt at the function entry point for `start_kernel()`. You will also see the source window and can track exactly where in the Linux OS kernel sources you are.

Issuing the command

```
run until sched_init
```

in the debugger console window

will get you to the scheduler initialization for the OS.

```
run until mwait_idle
```

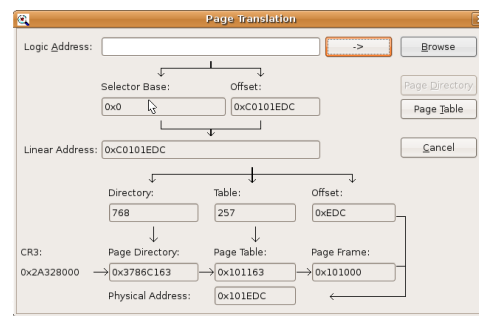
will get you to the central OS idle loop.

To understand the OS memory configuration you could either take a closer look at the descriptor tables



or you could directly go for a graphical representation of the physical-to-virtual memory translation by taking a look at the page translation

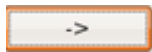
table  as seen below.



program counter location check the value of the EIP register



and copy & paste the value into the Logic

Address field. Clicking on the  button will then give you a complete representation of the page translation.

If you would like to check the actual page table entries

you can then click on  and the below table will appear.

Index	Value	Comment
0	0x29422067	ADDR=29422000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
1	0x2909A067	ADDR=2909A000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
2	0x29B4B067	ADDR=29B4B000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
3	0x29419067	ADDR=29419000 P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 G=0 AVAIL=000
4	0x0000000C	ADDR=00000000 P=0 R/W=0 U/S=0 PWT=0 PCD=0 A=0 D=0 PS=0 G=0 AVAIL=000
5	0x0000000C	ADDR=00000000 P=0 R/W=0 U/S=0 PWT=0 PCD=0 A=0 D=0 PS=0 G=0 AVAIL=000
6	0x0000000C	ADDR=00000000 P=0 R/W=0 U/S=0 PWT=0 PCD=0 A=0 D=0 PS=0 G=0 AVAIL=000

Clicking on any of the index entries will bring up a bitfield editor providing you with a detailed bit for bit view of the respective page table configuration register and explanation of the current settings.

## Debugging Device Driver

Let us assume the `idbntf.ko` module has already been loaded on the target and kernel module load information is being exported to the debugger on the host. Let us use the example of the ethernet driver as an example for debugging kernel modules automatically loaded during the OS boot process. First we need to make sure the kernel modules are built with symbol info generation enabled so the


debugger has a chance to map the sources once it knows the load memory load location. In the debugger itself we now need to tell the Linux OS awareness plug-in where to find the kernel module sources. You do this with the following commands:

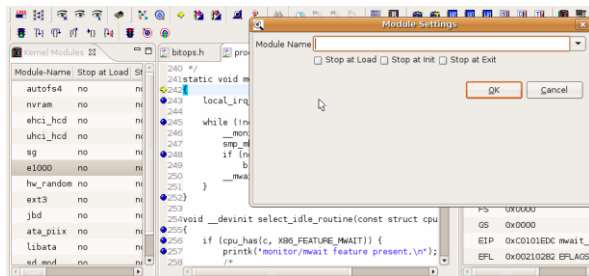
```
set directory "/usr/linux-2.6.31.i686/drivers/net/"
OS "setdir \"/usr/linux-2.6.31.i686/drivers/net/"
```

We will want to have the debugger take control and halt the boot process as soon as the ethernet driver is loaded by the operating system and ethernet connection is being established. To achieve this we need to modify the OS boot sequence so `idbntf.ko` is the first kernel module loaded or at least it is loaded way before the device driver that you would like to debug. Next we simply in the debugger tell the Linux kernel OS awareness plugin to stop at the driver initialization method.

This can be done with the following command to be entered at the debugger console window:



```
OS "stopinit e1000 /on"
```

Alternatively you could also do this via the kernel module window . By right clicking on the `e1000` kernel module in this window you will then get to a context menu that allows you to set the debugger to stop at the initialization method.



As soon as the ethernet driver is initialized we will be stopping at the `init_method` and the symbol info will be loaded, so we can actually see the source.

Scroll down in the sources and set a breakpoint at a specific function (i.e. IP address acquisition).

Hit the Restart button  and the Run button , wait for the OS to reboot and see how it stops at ethernet driver load later in the desired routine.

## Kernel Modules Loaded after Bootup

The process for debugging a device driver or kernel module loaded after the boot process is complete is very much analogous to what we just discussed. Let us take an arbitrary kernel module named `scull.ko` as an example

In the debugger you again tell the Linux OS awareness plug-in where to find the kernel module sources using the debugger console window:

```
set directory "/home/lab/scull/"
OS "setdir \"/home/lab/scull/"
```

Then you open the Linux Modules window and do a right mouse click and select "add" from the pulldown menu. You will notice all the other kernel modules already loaded during the boot process being listed. You will also notice that for most of them it states file not found. Referring to the location of the kernel module with symbol info. In the dialog that pops up you type in "scull" and you again select "Stop at Init". This time additionally you select "Stop at Exit" as well

Now the target can be released with the run command or button.

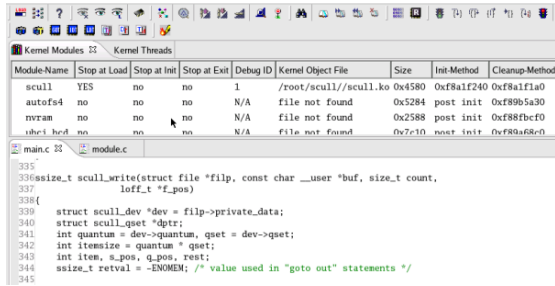
You PuTTY/SSH or Telnet onto the released and running target. After changing to the `scull` directory you initialize the kernel module by typing

```
./scull.init start
```

The debugger as expected stops at the `scull` init method.

You set a breakpoint at the `scull_read()` function and release the target once more.

Then you send an echo to the /dev/scull0 device. The debugger halts and you can step through the function.



## Conclusion

The Intel® Embedded Software Development Tool Suite for Intel® Atom™ Processor is a complete tools solution set to address Intel® Atom™ Processor specific software performance requirements, and to enhance the productivity and experience of the Linux-based system and application development process.

The Tool Suite covers the entire cycle of software development: coding, compiling, debugging, and analyzing performance. All included tools are Linux hosted and compatible with GNU tools.

As such they are an ideal supplement to your GNU GCC development environment to make it capable of efficient and optimized application development and deployment for your Mobile Internet Device.

## Where to find it

The Intel® Embedded Software Development Tool Suite for Intel® Atom™ Processor can be purchased or a 30 day evaluation license can be obtained the the tool suites can be downloaded from the Intel® Software Development Products Web pages (<http://software.intel.com/en-us/intel-compilers/>).

Customers have access to product updates and product support through the following Web pages:

- Intel Software Development Products Support: <http://software.intel.com/sites/support/>
- Intel® Software Network Discussion Forums: <http://software.intel.com/en-us/forums/software-development-toolsuite-atom/>
- Intel Premier Support: <https://premier.intel.com/>

For product and purchase information visit:

[www.intel.com/software/products](http://www.intel.com/software/products)

Intel, the Intel logo, Intel. Leap ahead. and Intel. Leap ahead. logo, Pentium, Intel Core, and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © 2009, Intel Corporation. All Rights Reserved.

0506/DAM/ITF/PP/500 319332-002

<sup>1</sup> <http://developer.intel.com/software/products/compilers/clin/docs/manuals.htm>

<sup>2</sup> <http://developer.intel.com/software/products/compilers/cwin/docs/manuals.htm>

<sup>3</sup> <http://www.streamline-computing.com/>

<sup>4</sup> <http://www.etnus.com/>

Document Number: 319332-002US