System Development for embedded Linux* OS
Running on Intel® Atom™ Processor

Robert Mueller-Albrecht
Technical Consulting Engineer, Intel Corp.

Thorsten Moeller
Product Marketing Engineer, Intel Corp.
Abstract

Developing and validating embedded software stack stability and peripheral device interfaces for System-on-Chip (SoC) designs gets ever more challenging. Driving factor is the increased level of device integration and need of timely and synchronized messaging. Additionally driving this is a fragmented and specialized software and operating system ecosystem, as well as increased time-to-market pressure. In this article we will examine how software tools can help developer productivity by using specialized development and validation technologies. Furthermore we will examine key development, analysis and debug use cases as required with most common types of embedded Intel architecture platforms.

Introduction

Over the past several years we have seen the traditional embedded market segment experience a transformation from fixed function and isolated embedded systems to a new category of intelligent systems. The transformation is changing the way people engage with and experience computing. Implications from this shift are that devices are more capable and interact with each other and these usage models demand greater performance, an increasingly capable cloud of services, and software technology and tools that support these new classes of devices. These devices are secure, connected and managed. Several industry analyst firms such as IDC* and others are seeing this shift and have created a new embedded sub-category called “Intelligent Systems”. The increased usage of system-on-chip (SoC) designs in traditional embedded market segments as well as the new category of Intelligent Systems requires the developer as well as tools vendor a like to reassess the entire development process from design, code generation and debug to system performance analysis.

When developing the system software stack for embedded devices you are frequently confronted with the question how to set up the most appropriate development environment first. It helps to understand the options for approaching software development targeting embedded systems. In this article we will focus on SoC designs using an application processor based on x86 or Intel architecture. The target application processor architecture being similar to the architecture of the system most likely used by the developer for his or her development work has some unique advantages and widens the choices available.

These choices range from traditional native development of applications on the development host for later deployment on a target device to full blown traditional embedded cross development using a QEMU* based virtual machine, a remote debug connection to a physical target device and a sysroot or chroot based build environment. We will attempt in this article to shed some light on each of these options for Intel® Atom™ Processor targeted designs.

One of the most commonly used operating system for Intel architecture based embedded systems is a customized flavor of Linux*. These may range from something based mainstream Linux* distributions to custom builds from scratch. Linux distributions specialized on embedded and mobile computing applications like Wind River* Linux*, Yocto Project*, or Android*, are playing a more and more prominent role. Frequently embedded designs have a real-time requirement for at least parts of the software stack. This implies that isolating those components of the software stack that have this requirement and ensuring that they do have only minimal dependencies on the non real-time part of the software stack becomes one key challenge of developing and defining embedded systems software.

Furthermore, when talking about SoCs, understanding the interaction between all the platform components becomes vitally important. You may be developing device drivers that take advantage of special GPU features or use micro-engines and other hardware accelerators found on the platform. You may be designing the software interface for the many wireless radios found in today’s small form factor devices. In either scenario, being able to analyze the message timing – making sure the variable values that are exchanged between the different platform components contain the right value at the right time, becomes important for platform software stack stability. Being able to access device configuration registers easily and in a comprehensive manner also simplifies developing device drivers and controlling platform component interaction timings.
1. Intel® Architecture in Embedded

The x86 architecture played an important role in embedded almost since its inception. Over the past 30+ years many industry players like AMD®, ST Microelectronic®, and General Dynamics® as well as Intel have had many highly integrated designs targeted at the embedded market place. More recently the introduction of highly integrated low power footprint SoCs based on the Intel® Atom™ Processor added new platform capabilities as well as new development challenges.

Figure 1: Intel® Atom™ Processor N2xxx based Intelligent Connected Systems

The Intel® Atom™ Processor is used in designs ranging from print imaging, industrial control, digital signage, point of sales, medical tablet, in-vehicle infotainment, digital surveillance, IPTV, connected services gateways, home energy management, and many other intelligent embedded systems.

Every embedded use case that requires both compatibility to the wide range of the x86 based infrastructure and is power sensitive at the same time, is suitable for this processor generation.
It is based on the same architecture that powers development machines and personal computers. Hence the Intel® Atom™ Processor benefits from the same software ecosystem that enables original equipment manufacturers (OEMs) of those devices to rapidly develop and upgrade with cost savings.

However, the ever increasing integration level and increased use of heterogeneous multi-core systems of modern designs add some unique challenges to the established ecosystem of embedded software development solutions for Intel architecture. Here we are taking a look at these challenges and how they can be addressed.

2. System-On-Chip Challenges

A system-on-chip by definition consists of a large number of diverse devices, hardware accelerators, and micro-engines, all tightly packed on a single chipset. This extreme integration of an entire design on a single chip is especially prevalent when there are small form factor requirements like with warehousing and communications handhelds.

Another type of design that sees the need for a large number of specialized devices on a single chip that collaborate closely is the digital set-top box or IPTV. There typically the video and audio content streaming is strictly separated from the user interface and end-user interaction. This avoids anything that could impact the delivery of content streams and cause intermittent package loss due to diversion of system resources on the controlling application processor.

If we take the Intel® Atom™ Processor CE42xx or its successor the Intel® Atom™ Processor CE5xxx for instance, you will notice a wide array of hardware signal encoders and decoders as well as digital signal processors and security processors (Figure 2). However, since the Intel® Atom™ Processor at the center of this SoC needs to coordinate and orchestrate the interaction between all the other SoC components, it becomes important to be able to monitor the data flow across the entire SoC in an accurate manner that provides a good idea of the timeline of the data flow across the entire SoC.

![Figure 2: Intel® Atom™ Processor CE4200 SoC](image)

One of the most common categories of software problems encountered on SoCs is very hard to trace, because they are based on a breakdown in data exchange timing. Data is being accessed before the content that was expected in the accessed data structure has been updated with the desired values. These issues can be very difficult to reproduce and may not show up until the SoC is put into a stressor environment or in front of the end customer.

The desire of course is to find, identify and fix these kinds of issues as soon as possible in the development cycle.

Two commonly used approaches can be taken for this. You can

1. either use instrumentation in your code base and have the instrumented code log all the data transfers as well as alert you on inconsistencies
2. or have the hardware provide low level system wide clock signals that can be used to monitor data exchanges along with exact time stamps.
The second approach gets us to another challenge when it comes to debugging on a System-on-Chip based design: breakpoints. A breakpoint trigger cannot take the usual route along with all the other signals transferred on the SoC. There needs to be a dedicated trigger circuit on the SoC that allows for cross-triggering of breakpoints within only a few cpu clocks, a debug synchronization unit. Otherwise it would take a breakpoint on one SoC device hundreds of cycles before the second involved SoC components was stopped as well. This would render a cross-SoC data signal breakpoint difficult to take advantage of for debug purposes. The decision to break needs to happen within a few cycles of the break that occurred on the original SoC component, which first encountered the break condition.

As you can see from this brief discussion of the added difficulty in the data flow and thus in the debug and development environment for SoCs, understanding whole platform interactions for software design becomes an ever more important challenge with these designs. We will discuss how to address these challenges moving forward.

3. Cross-Development Environment

What is true for embedded software development targeting other architectures, also holds true for Intel architecture, although to a slightly lesser degree. Most of the time the software stack (OS layer, drivers, user interface) running on the target device differs significantly from the development host. This still holds true on Intel architecture. The only difference is that since host and target architecture are closely related, basic functionality testing where there is no strong dependency on target specific libraries and drivers can be done directly on the host development environment without any simulation layer.

Especially for small form factor devices with their function limited user interfaces and very specific signal processors and micro engines, the opportunities for traditional desktop style native software development are however limited.

Then you will end up with configurations similar to what you may have used when targeting other hardware architectures.

The build environment consists of a compiler and all the associated components: linker, assembler, and other utilities. There are two aspects about building for a target device other than your development host that may be different than traditional development:

1. The compiler and linker need to know which header files, shared objects and system specific libraries to reference during the build. This can be achieved by setting up a jailroot or chroot environment. The compiler runs inside its own protective wrapper and is installed inside a chroot based filesystem. In short, the build environment is not even aware of the host platform and OS it is running on. It only sees a command line filesystem that is identical to what will eventually run on the target device.

   Another less aggressive approach is to install a compiler tool suite that supports sysroot. The compiler and linker via a --sysroot command line option specify exactly which header files, libraries, startup objects, and other target system dependencies to use instead of the default host environment.

   For this purpose you can use something like the GNU* based MADDE* or Poky* infrastructure or solutions from Sourceforge* or Wind River*. Intel's C++ Compiler also supports the sysroot build model and can be plugged into the different sysroot based build environments available in the open source community.

2. The build process may very well require a separate link step with an explicitly defined link stage memory map for the application. This is especially true for drivers that may interface with signal processors or micro-engines on the SoC and require a specific memory layout and alignment to transfer data in dedicated shared memory. The likelihood of this requirement is bigger for the system software layer on the target platform than for the application and intermediate software layers. Modern build environments often implicitly link in system and standard C library components without explicitly mentioning them in the makefile. If the link step of the build is run separately this breaks down. Every dependency has to be cleanly listed for the linker. If you are not familiar with the sometimes complex embedded build environment you may encounter, identifying all these dependencies can be a challenge.

The Linux* Foundation's Yocto Project* for instants provides a framework for Linux* on embedded platforms and intelligent systems. Among that offering are board support packages (BSP) for various architectures, including the most current generation of Intel® Atom™ Processors. The Application Development Toolkit (ADT) and Build Environment for Yocto Project* is a typical example of how having a flexible framework for embedded Linux* builds on x86 can greatly improve time to market (TTM) for reliable and feature rich intelligent systems solutions.

Yocto Project* (Figure 3) for instants uses a sysroot based cross-build environment based on Poky* Linux* for both OS and application build. The build environment and open source repository structure is kept flexible enough to accommodate Debian* File Format (DEB) and RedHat* Package Manager (RPM) based build environments and repository access methodologies.
It includes a wide range of resources for key embedded functionality including real-time scheduler support. The build flow as shown in Figure 3 can be seen as representative of a commonly used development setup based on a highly configurable Embedded Linux framework.

Once we move past the build stage and look at software debug and validation, we see that on Intel® Architecture similar embedded cross-debug approaches apply as on other architectures.

For software functional testing a simulation based approach could be used. The code to be tested could be executed in a chroot environment used for building as well. In addition a Graphical User Interface (GUI) simulation layer could be used. This is the approach frequently used for user-mode application development or porting from a full-featured PC to a more embedded form factor. Developers targeting the currently popular mobile phone and tablet platforms may also be familiar with this approach. In the case of using an Intel® Atom™ Processor target the advantage is that the code being tested is identical to the code that will run on the final device, instead of it being a special compile just for functional testing.

To test in an environment that represents the entire target software stack correctly a virtual machine approach may make more sense. For Linux* target this would usually be something QEMU* based. This would also allow simulating the hardware I/O device response and getting a fuller picture for applications with stronger target hardware dependency. The runtime performance of this approach is a bit worse than the pure user interface simulation mentioned before, but it provides a better full simulation and for Linux* kernel image testing it allows to use the same Linux kernel image that may later be used for the full system build.

Lastly, once the target hardware is available you can execute the code base and test it on the actual hardware.

For debugging and controlling test scripts in all the scenarios described above a remote debug agent is used. In the GUI simulation and virtual machine case, a virtual USB or TCP/IP interface driver would be used to handle debug agent communication. The debugger runs outside the simulation environment with the debug agent handling run control and information exchange from inside the environment.

The approach when dealing with a real target device is identical. A debug agent like the gdbserver that comes with GDB* or the idbserver that comes with the Intel® Debugger runs on the target platform. It then communicates with the development host via a standard I/O application interface like USB or TCP/IP. For kernel space cross-debug this type of debug agent could even be provided as a kernel module or linked into the kernel itself.

Another approach, especially useful when doing testing on a still unstable software framework running on-top of early hardware revisions, is to use a JTAG* based debug solution. JTAG is defined by the IEEE 1149.x standard for test access ports and boundary scan architecture for testing circuit boards. It provides the de-facto standard method for low level hardware platform debug and testing. It allows full run and memory access control of the target application processor through a serial send/receive signal line together with reference voltage lines and control signal lines.

This approach is essentially software stack agnostic. Any Linux* OS awareness will need to be based on structures exported by the OS itself and read by the JTAG Debugger from target memory.
As those familiar with cross-development targeting architectures other than Intel® Architecture, the development methods and approaches are very similar. The one central difference is the degree to which one may have to rely on cross-development versus being able to use native development on an actual target device or chroot based approaches.

4. BIOS, Firmware & Bootloader

Let us look at system software development on Intel® Atom™ Processor based platforms from the ground up to get a better idea of how this cross-development concept works for an Intel® architecture based embedded platform running Linux®. The first step is to build statically linked bare metal code. To build statically linked code as used in platform tests and OS bootloaders, you can use either the GNU* compiler or the Intel® C++ Compiler. The main difference to standard application code is twofold.

First, the compilation step and link step tend to be separate. In the link step you will explicitly specify the link and load address where the application will be loaded in target memory. This ensures that the debugger knows the download location for the code and that the debugger knows how to map the symbol info to the application running on the target. This will allow the sources shown in the source window to match with the target code.

Then there is an additional build step at the end, which is to use the GNU* utility objcopy or the GNU* linker to generate a hex file or binary in addition to the ELF Dwarf2 linker output. This hex file or plain binary is what will be downloaded to and executed on the target.

For debugging you will want to make sure to have symbol information (-g) available. You also want to really have the link stage create the ELF Dwarf2 file since it contains the symbol info passed on from the compiler when the –g option is used. To avoid a lot of excitement and strange effects during debugging you should always make sure that link base address for the application and the address that you actually load the application to are identical, although a good debugger can add an offset to the symbol info location to compensate for address mismatches.

To debug a bootloader application, which runs on the target device even before any OS is present, the debugger needs to be able to first of all download this relatively simple, but usually highly footprint optimized bare-metal application into pre-initialized target RAM for testing.

The debugger needs to be able to map and associate the symbol info and sources for statically linked code and to also download the application directly into target RAM for easy debug ability and testing. Once testing, developing and debugging is complete the bootloader would then go into target solid state memory. This is usually flash, where it can do its job for preparing of target initialization and of loading the OS image from NAND, unpacking it into RAM and launching the OS.

All these steps of bootloader development are actively supported by the Intel® JTAG Debugger as well as Wind River® On-Chip Debugging and similar offerings from Green Hills® Software, Macraigor Systems*, American Arium*, Lauterbach* and others. If you are used to one of the aforementioned system debugger and JTAG solution vendors from another architecture, the transition to debugging and developing bootloaders on Intel® Atom™ Processor is going to be quite straightforward.

5. OS Kernel & Device Drivers

The next step after having the bootloader working is to develop and debug the Linux* OS kernel itself. The Linux* kernel build environment, whether it is inside a configuration IDE like Wind River® Workbench, the Eclipse® IDE or it is simply kernel makefile config based, is similar to what a Linux* system developer would expect.

Even at the OS layer it pays however to use relevant optimization switches to achieve the best possible user experience and performance. If you are targeting primarily Intel® Atom™ Processor based devices, the compiler used should generate code that is optimized for this hardware platform.

This implies the use of the GNU* compiler GCC* in version 4.5.1 or newer with the option switch –mtune=atom. When using the Intel® C++ Compiler in version11.x or newer you should use the option –xSSE3_ATOM, or with the latest Intel® C++ Compiler 12.1 the option –xSSE3_ATOM.

Intel® Atom™ Processor specific compiler optimizations cover a wide range of low level optimizations. These include the extensive use of LEA (Load Effective Address) instructions to reduce memory pressure and the penalty for memory accesses. They include a wide range of optimizations to most efficiently use the processors floating point unit. They include modeling of the processors instruction pipeline to feed instructions to the processors instruction pipeline in the order best for fast processing. Beyond this the efficient use of vector registers can have a major performance impact, but let us revisit that in the context of device driver and application development, where those types of optimizations have a greater impact.

A system debugger, whether debug agent based or using a JTAG device interface is a very useful tool to help satisfy several of the key objectives of OS development.

The debugger can be used to validate the boot process, analyze and correct stability issues like runtime errors, segmentation faults, or services not being started correctly during boot.
It can also be used to identify and correct OS configuration issues by providing detailed access and representations of page tables, descriptor tables and also instruction trace. The combination of instruction trace and memory table access can be a very powerful tool to identify the root causes for stack overflow, memory leak or even data abort scenarios.

Figure 4 shows the detailed access to page translation attributes and descriptor tables as provided by the Intel® JTAG Debugger. With the high level of flexibility that is available on x86 in defining the depth of translation tables and granularity of the addressed memory blocks, this level of easy access and visibility of the memory layout becomes even more important for system development on the OS level.

This highlights two key differences between developing and configuring the embedded OS software stack on Intel architecture and many other architectures. The selector base and offset addressing model, combined with the local descriptor table (LDT) and global descriptor table (GDT) allow for deep, multi-layered address translation from physical to virtual memory with also variable address chunk granularity. This is a powerful capability for custom memory configuration in a compartmentalized environment with protected isolated memory spaces. If used incorrectly it can however also increase memory access times. Thus the good visibility of memory page translation is desirable.

One other difference between Intel architecture and others is the handling of system interrupts. On ARM® for instance you have a predefined set of hardware interrupts in the reserved address space from 0x0 through 0x20. These locations then contain jump instructions to the interrupt handler. On Intel architecture a dedicated hardware interrupt controller is employed. The hardware interrupts are not accessed directly through memory space, but by accessing the Intel® 8529 interrupt controller. The advantage of this approach is that the interrupt handler already allows for direct handling for I/O interrupts for attached devices. In architectures that don’t use a dedicated interrupt controller usually the IRQ interrupt had be overloaded with a more complex interrupt handler routine to accomplish this.

A good JTAG Debugger solution for OS level debug should furthermore provide visibility of kernel threads and active kernel modules along with other information exported by the kernel. To allow for debugging dynamically loaded services and device drivers a kernel patch or a kernel module that exports the memory location of a driver’s initialization method and destruction method may be used.

These capabilities combined with the memory layout awareness become especially helpful when dealing with an OS kernel that uses a preemptive real-time scheduler for parts of its software stack and requires for security reasons that those real-time components be executed in dedicated memory space.

6. Code Correctness

This leads us from the question of OS level build and design to the question of code-correctness. In the scenario of developing device drivers that interact with SoC devices and Digital Signal Processors (DSPs), it is not just important to understand memory layout and the correctness or memory accesses. For this purpose you have thread correctness and memory checking tools. Valgrind* and the Intel® Inspector XE come to mind. Some high-end debugger’s like Roguewave’s Totalview* and Alinea’s DDT* incorporate memory checking features into their debug solution.
Especially for system configuration and device driver debugging, it is also important to be able to directly access and check the contents of device configuration registers. The concept of bitfield editors as shown in Figure 5 can be very useful for this. A bitfield editor is a bitwise visualization of SoC device registers, that allows monitoring changes to a device state in real-time while the associated device driver is interacting with it.

Figure 5: Device Register Bitfield Editor View

If you are connected with your JTAG Debugger to the target platform early during the boot process, to find e.g. issues in the OS boot-up, it is first and foremost important to configure your debugger to only use hardware breakpoints. Attempting to write software breakpoints into memory, when target memory is not fully initialized, can corrupt the entire boot process.

Analyzing the code after the Linux* compressed zImage kernel image has been unpacked into memory, is possible by simply releasing run control in the debugger until start_kernel is reached. This implies of course that the vmlinux file that contains the kernel symbol info has been loaded. At this point the use of software breakpoints can be re-enabled. The operating system is then successfully booted once the idle loop mwait_idle has been reached.

Additionally, if your debug solution provides access to Branch Trace Store (BTS) based instruction trace, this capability can in conjunction with all the regular run control features of a JTAG Debugger be used to force execution stop at an exception and analyze the execution flow in reverse identifying the root cause for runtime issues.

7. Performance

As a final step in the development process performance tuning may be necessary. In this chapter we will briefly discuss how to optimize performance on Intel® Atom™ Processor based platforms. We already mentioned previously how important it is to use a compiler that fully takes advantage of the platform and produces optimized code. There are multiple opportunities for compiler based and manual optimizations.

Compiler based vectorization is one area that can yield performance improvements that is worth considering. The Intel® Atom Processor has a set of 128bit wide vector registers, which can be accessed using intrinsics. Intrinsics are C level function APIs that call specific assembly instructions, while handling the register save & restore overhead. In addition to using these registers and the Streaming Single Instruction Multiple Data (SIMD) Instructions (SSE) explicitly, a compiler targeting Intel® Atom™ Processor based architecture should actively identify opportunities to use up to and including Intel® Supplemental Streaming SIMD Instructions 3 (SSSE3) for loop optimizations. These loop optimizations are called auto-vectorization.

The example in Figure 6 shows the process of auto-vectorization. The compiler does loop unrolling and feeds the loop contents into vector registers. This allows executing multiple iterations of a loop in parallel.
The limitations for this optimization are that it requires countable loops with regular strides and is fairly sensitive to inlined function calls and heavily nested loops. The compiler provides a detailed vectorization report in the build output, when compiler option --vec_report[N] is used. It is advisable to follow the compiler’s suggestions provided in this report, on how to restructure loops. Apart from bootloader code, this is the one area where customizing source level code for the target architecture may produce significant payoff. Taking advantage of auto-vectorization is most desirable for data processing intensive code sequences and applications.

Media and signal processing function libraries like VSIPL* or the Intel® Integrated Performance Primitives may also directly take advantage of the data parallelism provided through vector registers.

Finally, let us talk about identifying optimization opportunities using performance analysis tools. Using the OS timer or the CPU clock to identify where most time is spent inside a specified application or OS component is the most straightforward approach. This can be done using standard OS interfaces or using architectural features. The architectural feature that the Intel® Atom™ Processor provides for performance analysis is called the Performance Monitoring Unit (PMU). In addition to monitoring the amount of cpu_clock time spent in a specific piece of code, you can also monitor things like cache misses, branch mispredictions, and memory accesses.

This data can then be correlated to program counter memory locations for these events. Tools like PerfMon*, Intel® VTune™ Amplifier XE, and many others take advantage of these capabilities to varying degrees. Additionally many of these events can be programmed directly for writing your own analysis utility.
Figure 7: Key Intel® Atom™ Processor Profiling Events

Figure 7 shows some of the most commonly used PMU events for the Intel® Atom™ Processor. This highlights the kind of resource bottleneck information that can be obtained using PMU based performance analysis in addition to just relying on OS counters. Most importantly you can get information about which piece of code generates the largest number of cache misses and may thus lead to very expensive memory loads or memory stores. You can also get similar information about costly branch mispredictions. If information about these types of system events correlates to where a lot of time is spent, you not only know where additional code optimization is warranted, but also what the reason for performance issues may be.

The strength of relying on PMU events instead of correlating OS timers is that sampling can be instrumentation free and can look at system level events and the impact that interactions between a device driver and the OS kernel have on performance. This is difficult to achieve with methods that rely on dynamic code instrumentation at run-time instead of architectural features like those provided by the Intel architecture.

8. Summary

Developing the Linux* system software stack for complex SoCs provides some unique challenges. These challenges can however be easily managed with the right set of software development tools and by using a well thought out development setup. Embedded system software development targeting Intel® Atom™ Processor based designs is as straight forward and frequently even provides more options than you find for other architectures. The usual development methodologies used for other embedded architectures also apply for the Intel® Atom™ Processor. A rich and established development tools ecosystem for the embedded space exists, especially when talking about embedded Linux* targets. Open source projects like Yocto Project* and SourceForge* provide a rich framework and set of utilities. This is augmented by commercial offerings from Mentor Graphics*, Wind River*, Green Hills* Software and others. The architecture has been used and proven itself in the embedded space over many decades. One advantage of having the same basic embedded architecture on development host and target is that it reduces the need for using cross-development, although especially for system development it is frequently still recommendable.

In addition, even if not required, it is frequently still recommendable to employ cross-development to ensure a clean build and validation environment. Linux* provides a wide array of customizable build solutions targeting the Intel® Atom™ Processor for exactly that purpose.
The rich set of software development tools from the open source community as well as tools vendors and Intel itself makes development easier, especially for those moving from personal computer centric native development to developing for intelligent systems for the first time.

In this article we provided a wide overview of the challenges of developing for Embedded Linux* and the types of development tools solutions, that are available. Furthermore we also highlighted the importance of taking advantage of the analysis and debug capabilities available when developing for Intel® Atom™ Processor.

The key to being successful when developing for Intel® Atom™ Processor is to be aware of the rich ecosystem and to first define the build environment that meets your needs. Keep the build environment simple, but also ensure that target environment dependencies are not broken. Relying on printf debugging can cost valuable time, when serious issues arise. Taking advantage of advanced cross-debuggers and performance analyzers, will increase software stack stability and performance. Take a look at embedded Linux* frameworks like Yocto Project* targeting Intel® architecture to have a good start into defining your custom Linux* software environment.

This is only a starting point, indicating the things to request and look for when setting up a project. Please check out the additional resources and references for more in-depth guidance.

**Additional Resources**


Intel® Atom™ Performance for DSP Applications, Tim Freeman and Dave Murray: Birkenhead, UK, N.A. Software Ltd. 2009: [http://www.nassoftware.co.uk/home/attachments/019_Atom_benchmarks.pdf](http://www.nassoftware.co.uk/home/attachments/019_Atom_benchmarks.pdf)

Intel® Embedded Design Center [http://edc.intel.com](http://edc.intel.com)


**References**


Yocto Project*: [http://www.yoctoproject.org](http://www.yoctoproject.org)


**About the Authors**

Thorsten Moeller, Product Marketing Engineer for embedded software development tools with Intel Corporation has more than 10 years of experience in the high technology industry with extensive product strategy, product management, and technology marketing expertise in the areas of embedded, mobile and enterprise computing. Before Intel, Thorsten was a senior systems engineer at Sarvega Inc., a technology provider who pioneered high-performance XML and XML security processing solutions. For the past 6 years, Thorsten has focused almost exclusively on bringing software solutions to market in the areas of embedded system software and enterprise computing. Thorsten has an MSc in Computer Science from the University of Applied Sciences Frankfurt.

Robert Mueller-Albrecht, Technical Consulting Engineer for embedded software development tools and debug solutions with Intel Corporation has more than 10 years of experience in the embedded and mobile industry. After completing his MSc in Physics / Computer Science at the University of Kaiserslautern, Robert started his career in software engineering in 2000 with CAD-UL. At Intel since 2001 he continued in this focus as a software engineer with development tools for the mobile and embedded space. Recently his focus has been primarily on supporting embedded development tools and debug environment requirements targeting Intel® Core and Intel® Atom processor based designs.
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel, the Intel logo, VTune, Cilk and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright© 2012 Intel Corporation. All rights reserved.

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