Signal Processing and Power Management Workshop for Intel Embedded Platforms

Noah Clemons
Technical Consulting Engineer, Intel Corp.
noah.clemons@intel.com
Abstract

This paper details the full three-step workshop to code signal processing workloads on Embedded Intel® Architecture Platforms:

1. Know what Intel Embedded platforms exist along with their corresponding tools.

Learn the dos and don'ts of defining a custom development environment for your specific needs when targeting Intel® Architecture. See the available solutions to achieve a well-designed development setup.

Intel Embedded platforms come in many shapes and sizes. Many are based on heterogeneous System-on-Chip designs using a multitude of hardware architectures. System software stack customization and application development for varied configurations requires a flexible approach and development tools that can be adapted to the needs of cross-development for a variety of software stacks. Software development tools should also provide implementation approaches for dedicated throughput optimization and compute power of high priority processes.

Whether the embedded system you develop for is a low-power Intel® Atom™ Processor based design running an RTOS or an Intel® Xeon™ based multi-socket design with reserved processor cores for different tasks, the basic requirements of embedded software development apply. It is desirable to have deep hardware platform and device driver insight, flexible cross-build environment support for sysroot based solutions, configurability to adjust to unique requirements, and integration options for commonly used cross-build environments.

2. Use the optimized libraries and tools within Intel’s line of Embedded Software tools useful for signal processing

Learn about performance primitives, dedicated math libraries, and signal processing libraries that efficiently take advantage of data parallelism for a wide range of signal processing use cases. Identify which non-time critical portions of your signal processing application should be offloaded to accelerators, dedicated micro-engines, FPGAs, or GPUs, and also identify situations where using the application processor is advised. Single Instruction Multiple Data (SIMD) instruction sets will be used in the primitives and libraries for highly parallel and efficient data processing.

3. Tune your signal processing workloads for power

It is becoming increasingly important for software developers to become more power aware, not only to achieve extended battery life, but also for reduced thermal stress on embedded platform components. Consequently, it is critical for your signal processing workload software to not be wasteful in their power usage. We will present how you can use the power analysis features of Intel® VTune™ Amplifier 2013 to show when your system is asleep, what sleep state it was in when it woke up and what caused the system to wake up. We will also demonstrate how you can track the CPU Frequency along the execution time line of your application or system software stack component. We will show how you can optimize the use of CPU Sleep states and lower CPU Frequency by discovering where your software is making non-optimal use of resources and could therefore be wasting power.

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 virtual machine with hypervisor, 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.
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.

When first starting on an embedded software development project these requirements may seem confusing and it is easy to get distracted by the complexity of the different requirements instead of focusing on the simple basics of defining your setup. This is the starting point of this article.

**Table of Contents**

1. Intel® Architecture in Embedded Devices and Intelligent Systems................................................................. 4
2. Options for Cross-Development for Intelligent Systems.......................................................................................... 5
3. Intel® System Studio Components......................................................................................................................... 6
4. Reliability and Debug................................................................................................................................................ 7
   Application Debug..................................................................................................................................................... 7
   System Software Debug........................................................................................................................................... 8
   Event Trace Debug................................................................................................................................................... 9
5. Signal Processing Libraries for Intel® Embedded Architectures............................................................................. 10
6. Power and Performance Tuning.............................................................................................................................. 19
7. Summary.................................................................................................................................................................. 23
   Additional Resources................................................................................................................................................ 24
   References............................................................................................................................................................... 24
   Notices.................................................................................................................................................................... 24
   Optimization Notice.................................................................................................................................................. 25
1. Intel® Architecture in Embedded Devices and Intelligent Systems

The x86 architecture played an important role in embedded almost since its inception. Over the past 30+ years many industry players like AMD®, ST Microelectronics®, and General Dynamics® as well as Intel have had many highly integrated designs targeted at the embedded market place. Recently two developments accelerated, are changing the needs of software developers and are driving the requirement for a more flexible approach to software development environments. The traditional boundaries of embedded and non-embedded are breaking down in that highly integrated computational power is needed throughout the modern communication, M2M and cloud infrastructure, integrating the embedded domain ever more closely with the cloud and server infrastructure.

![Figure 1: From Embedded Devices to Intelligent Systems](image)

The Intel® Atom™ processors are 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 and home energy management. 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.

AMD® Opteron® series, Intel® Core™ and Intel® Xeon™ processor families are used in a wide variety of server, industrial, networking, communication and media infrastructure applications.

The embedded devices and intelligent systems operating inside the “Internet of Things” are based on the same architecture that powers many development machines and personal computers. Hence they benefit from the same software ecosystem that enables original equipment manufacturers (OEMs) of those devices to rapidly develop and upgrade with cost savings.

![Figure 2: Categories of Intelligent Systems](image)
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. Options for Cross--Development for Intelligent Systems

The first decision to be made when defining the build environment is the choice of a cross-build development tools set to base development on. If you are using a proprietary real-time operating system (RTOS), the basic development toolchain will come from the OS vendor. Examples for this are Integrity* from Green Hills Software, VxWorks* from Wind River*, Nucleus from MentorGraphics* or QNX*. All of these come with their defined set of development tools, which can only be augmented by utilities from other vendors.

If you are using a Linux* based target OS, whether with or without real-time scheduler, the choices get considerably more varied. You have the option to build your own cross build tool chain, create a full target OS image inside a buildroot/chroot wrapper or to use a variety of pre-defined cross build environments that come with specific Linux* flavors targeted for the embedded market.

Let us start with the roll-your-own approach to highlight what to consider when choosing a cross-development centric GNU* toolchain.

When talking about development tools usage in embedded, one must distinguish three different machines:
- the build machine, on which the tool set is built
- the host machine, on which the tool set is executed
- the target machine, for which the tool set generates code

Very rarely do you actually need 3 systems. In most scenarios the build machine and the host machine will be identical. This reduces the setup to a development host machine and a development target machine.

![Figure 3: Basic Cross-Development Setup](image)

A complete tool suite is made up of the following components:

- **Compiler**: This could be a range of different compilers. For Linux* targets the most common compiler to use are the GCC and G++ GNU* Project compilers. This could be augmented with for instance the Intel® C++ Compiler or in some cases it may even be useful to replace it with the Intel® C++ Compiler for an entire application build. The main question will be whether this compiler is going to be a native GCC build or a compiler build that is part of a tool suite specifically targeted at cross development.
- **Build Process Backend Components**: Looking beyond the compiler itself, there are the build backend tools like assembler, linker, and output format conversion utilities. In the Linux* world this is covered by the GNU Binutils.
- **Libraries**:
  - The main set of library files required are part of the C library, which implements the traditional POSIX API that can be used to develop userspace applications. It interfaces with the kernel through system calls, and provides higher-level services. GLIBC ([http://www.gnu.org/software/libc/](http://www.gnu.org/software/libc/)) is the C library from the GNU project. Embedded GLIBC ([http://eglibc.org](http://eglibc.org)) is a variant of the GNU C Library (GLIBC) optimized for embedded systems. Its goals include reduced footprint, support for cross-compiling and cross-testing, while maintaining source and binary compatibility with GLIBC. uClibc ([http://uclibc.org](http://uclibc.org)) is an alternate C library, which features a much smaller footprint. This library can be an interesting alternative if flash space and/or memory footprint is an issue. The C library has a special relation with the C compiler, so the choice of the C library must be done when the custom tool suite is generated. Once the toolchain has been built, it is usually no longer possible to switch to another library.
  - Additionally libraries for optimized and simplified implementation of signal processing, media processing and security encryption routines may be of interest. These are libraries like Vector Signal Image Processing Library VSIPL ([http://www.vsipl.org/](http://www.vsipl.org/)) or the Intel® Integrated Performance Libraries (Intel® IPP) or Intel® Math Kernel Library (Intel® MKL).
- **Performance and Power Analysis Tools**: With the typically embedded requirement of getting a maximum amount of performance for a dedicated task combined with reduced power consumption or battery usage out of a given highly integrated chipset, software
tuning and analysis tools become an ever more important part of your cross-development toolset. These tools come in very varied packages and with a wide range of use cases. They will be discussed in a bit more detail later.

- Debuggers: The tool whose importance is commonly most underestimated, by those not already used to the extreme reliability requirements of many embedded use cases, is the debugger. These fall into three categories. Memory and code correctness checking tools, high level language application debug tools and system software debuggers. The latter two are focused on real-time debug of software stack components. A forth type of debug tools that has found new live because of the complex interactions in heterogeneous multicore SoC chipsets is the instrumentation based event tracing utility.

Only the build tools configuration of a tool set like this would be considered for generating yourself.

The most straightforward approach that is especially easy in a scenario where the development host architecture as well as OS, are closely related to the target architecture and OS is something called buildroot (http://www.buildroot.net). The concept behind this is that you have a complete customized embedded Linux build environment and can create a custom OS image. If you include the build tools in this OS image, you automatically also get the build environment with it. If you then use chroot (http://www.linux.org/article/view/chroot-access-another-linux-from-your-current-linux-) to create a virtualized boot environment for your virtual OS image, you can run your new embedded Linux build environment in a protective virtual wrapper on top of the standard Linux host. As in the case of Intel® Architecture targets the development host and target are closely related, this is possible without the overhead of an actual virtual machine.

The alternative approach, which is valid for other architecture targets as well as Intel® architecture is the generation of a custom cross-build binutils and gcc tools set using the –sysroot option to tell the compiler which version of C libraries, headers, start objects and other shared objects to use and where they can be found. All common GNU based cross-build environments use some variation of this approach.

Creating your own cross-build GNU tool set is often tied into creating your own custom Linux* platform. OpenEmbedded (http://www.openembedded.org) with its Bitbake utility is one approach to roll a custom Linux*. Poky Linux (http://pokylinux.org) offers an approach to creating your own cross-build GNU toolchain.

Many complete Embedded Linux* frameworks like OpenEmbedded or Yocto Project rely on these as a foundation.

Instead of creating your own framework it usually makes more sense to rely on the frameworks that already exist. In addition to OpenEmbedded and Yocto Project, there are also a variety of offerings from Code Sourcery, Wind River, MontaVista and Timesys, that may be worth having a closer look at.

Now that we defined the base Embedded Linux* platform to use for our Intel® Architecture targeted development, let us explore the existing toolset for that architecture. We will take the Yocto Project* as an example in the context of which we talk more about the individual tools components for your embedded development environment.

### 3. Intel® System Studio Components

In order to give an overview of how signal processing code development works with Intel® Architecture, it is necessary to go over the components of Intel® System Studio, which is the preferred development suite for the hardware:

- **Intel® C++ Compiler**
  - The compiler is optimized for IA-based embedded systems and devices. It works within the cross-development environments described in Section 2 and contains Eclipse* Integrated Development Environment (IDE) integration.

- **Intel® Integrated Performance Primitives**
  - This library provides algorithmic building blocks for many types of applications with
    - a simple “primitive” C interface and data structures to enhance usability and portability
    - higher level APIs for more complex tasks like media processing, including related infrastructure around the performance critical operations supported by the primitives
    - primitives which evolve with Intel hardware, and samples provided as starting points.

- **Intel® Math Kernel Library**
  - This library offers highly optimized and extensively threaded routines which implement many types of operations in areas such as linear algebra, summary statistics, data fitting, vector math, vector random number generation, Poisson solvers, and optimization solvers. Additionally, Intel MKL supports signal processing applications by including Fourier transform routines for 1-D to 7-D data sets of size n as high as 2^64. The Intel MKL Fast Fourier Transform (FFT) functionality offers:
    - Optimization for large problem sizes: \(O(n \log n)\) algorithm implementation for all data sizes, including for values of \(n\) which are not a power of 2
    - Mixed radix 2,3,5,7 kernels
    - Real and complex
Single and double-precision
- Multiple packed formats: CCS, Pack, Perm, and CCE
- FFTW interface to Intel MKL

- Intel® VTune™ Amplifier XE
  - This tool provides the performance and power profiling for application development.

- Intel® JTAG Debugger
  - This tool offers the debug insight into the CPU, SoC, and chipset, along with source-level debugging of the operating system kernel software and its drivers.

- GDB-based Application Debugger
  - Largely self-explanatory, this tool provides application debug and traces, along with race detection when writing parallel code.

- Intel® Inspector XE
  - This tool provides memory and threading analysis.

4. Reliability and Debug

The often most neglected components of a complete tool suite outside the embedded space are the debuggers. Embedded devices have a higher average deployment in the field with more difficult maintenance and higher stress on platform components compared to standard consumer PCs. This fact demands increased reliability of the software stack as well. This means that any development tool suite for embedded devices and intelligent systems is not complete without a well thought-out set of debug tools.

As mentioned in the introductory cross-development overview, debuggers fall into three categories: application debuggers, system debuggers, and event tracing tools.

Application Debug

For application debug in embedded cross-development scenarios, let us focus on GDB® as an example. It comes with a remote debug agent called gdbserver. This debug agent can be installed on the debug target to launch a debugger and attach to it remotely from the development host.

To do so, start your program using gdbserver on the target machine, gdbserver then automatically suspends the execution of your program at its entry point, waiting for a debugger to connect to it. The following command starts an application and tells gdbserver to wait for a connection with the debugger on localhost port 2000.

```
$ gdbserver localhost:2000 program
Process program created; pid = 5685
Listening on port 2000
```

Once gdbserver has started listening, we can tell the debugger to establish a connection with this gdbserver, and then start the same debugging session as if the program was being debugged on the same host, directly under the control of GDB.

```
$ gdb program
(gdb) target remote targethost:4444
Remote debugging using targethost:4444
0x00007f29936d0af0 in ?? () from /lib64/ld-linux-x86-64.so.
(gdb) b foo.adb:3
Breakpoint 1 at 0x401f0c: file foo.adb, line 3.
(gdb) continue
Continuing.
```

```
Breakpoint 1, foo () at foo.adb:4
 4    end foo;
```

It is also possible to use gdbserver to attach to an already running program, in which case the execution of that program is simply suspended until the connection between the debugger and gdbserver is established. The syntax would be

```
$ gdbserver localhost:2000 --attach 5685
```

to tell gdbserver to wait for GDB® to attempt a debug connection to the running process with process ID 5685.
Using GDB* for remotely debugging an application running inside a virtual machine follows the same principle as remote debug using the gdbserver debug agent.

The only additional step is to ensure TCP/IP communication forwarding from inside the virtual machine and making the ip address of the virtual machine along with the port used for debug communication visible to the network as a whole.

This requires in the case of QEMU that the bridge-utils package is installed. Inside the virtual machine in the guest OS the /etc/qemu-ifup file needs to be modified to include the correct setting for IP forwarding. Wikibooks is a good resource for the details of this setup (http://en.wikibooks.org/wiki/QEMU/Networking).

System Software Debug

For the system integrator and device manufacturer it may very well be necessary to work on the device driver and system software stack layer when determining their software stack to be as reliable and stable as feasible.

For true firmware, OS level system and device driver debug, using a JTAG interface is the most commonly used method in the embedded intelligent systems world. The Joint Test Action Group (JTAG) IEEE 1149.1 standard defines a “Standard Test Access Port and Boundary-Scan Architecture for test access ports used for testing printed circuit boards.” This standard is commonly simply referred to as the JTAG debug interface. From its beginnings as a standard for circuit board testing it has developed into the de facto interface standard for OS independent and OS system level platform debug.

More background information on JTAG and its usage in modern system software stack debugging is available at in the article “JTAG 101; IEEE 1149.x and Software Debug” by Randy Johnson and Stewart Christie (http://download.intel.com/design/intarch/papers/321095.pdf).

From the OEM’s perspective and that of their partner application and driver developers, understanding the interaction between the driver and software stack components running on the different parts of the system-on-chip (SoC) integrated intelligent system or smartphone form factor device is critical for determining platform stability. From a silicon validator’s perspective, the low level software stack provides the test environment that exposes the kind of stress factors the platform will be exposed to in real-world use cases. In short, modern SoCs require understanding the complete package and its complex real-world interactions, not just positive unit test results for individual hardware components. This is the level of insight a JTAG-based system software debug approach can provide. This can be achieved by merging the in-depth hardware awareness JTAG inherently provides with the ability to export state information of the Android OS running on the target.

Especially for device driver debug, it is important to understand both the exact state of the peripheral device on the chipset and the interaction of the device driver with the OS layer and the rest of the software stack.

Various JTAG vendors offer system debug solutions for embedded Intel® architecture including:

- American Arium (http://www.arium.com)
- Wind River (http://www.windriver.com/products/JTAG-debugging/)
- Macraigor Systems (http://www.macraigor.com)
- Lauterbach (http://www.lauterbach.com)
- Intel (http://software.intel.com)

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 and to 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.

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 bring-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.

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.

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. These registers and their contents can be simply listed with their register hex values or visualized as bitfields as shown in Figure 8. A bitwise visualization makes it easier to catch and understand changes to a device state during debug while the associated device driver is interacting with it.

![Figure 4: Device Register Bitfield View](image)

Additionally, if your debug solution provides access to Last Branch Storage (LBR) 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.

Last Branch Records can be used to trace code execution from target reset. Since discontinuities in code execution are stored in these MSRs, debuggers can reconstruct executed code by reading the ‘To’ and ‘From’ addresses, access memory between the specific locations, and disassemble the code. The disassembly is usually displayed in a trace GUI in the debugger interface. This may be useful for seeing what code was executed before a System Management Interrupt (SMI) or other exception if a breakpoint is set on the interrupt.

**Event Trace Debug**

Dozens of software components and hardware components interacting on SoCs increase the amount of time it takes to root-cause issues during debug. Interactions between the different software components are often timing sensitive. When trying to debug a code base with many interactions between components single-stepping through one specific component is usually not a viable option. Traditional printf debugging is also not effective in this context because the debugging changes can adversely affect timing behavior and cause even worse problems (also known as “Heisenbugs”).

There are a variety of static software instrumentation based data event tracing technologies that help address this issue. The common principle is that they utilize a small amount of DRAM buffer memory to capture event data as it is being created and then uses some kind of logging mechanism to write the trace result into a log file. Data trace monitoring can be real time by interfacing directly with the trace logging API or can be done offline by using a variety of trace viewers for analyzing more complex software stack component interactions.
LTTng*, Ftrace* and SVEN* are 3 of the most common such implementations.

- LTTng* Project: https://lttng.org/
- Ftrace*: http://elinux.org/Ftrace
- SVEN*: http://software.intel.com

5. Signal Processing Libraries for Intel® Embedded Architectures

Employing performance libraries can be a great way to streamline and unify the computational execution flow for data intensive tasks, thus minimizing the risk of data stream timing issues and heisenbugs. Here we will describe the two libraries that can be used for signal processing within Intel® System Studio.

Intel® Integrated Performance Primitives (Intel® IPP)

Performance libraries such as the Intel IPP contain highly optimized algorithms and code for common functions including as signal processing, image processing, video/audio encode/decode, cryptography, data compression, speech coding, and computer vision. Advanced instruction sets help the developer take advantage of new processor features that are specifically tailored for certain applications. One calls Intel IPP as if it is a black box pocket of computation for their low-power or embedded device – ‘in’ flows the data and ‘out’ receives the result. In this fashion, using the Intel IPP can take the place of many processing units created for specific computational tasks. Intel IPP excels in a wide variety of domains where intelligent systems are utilized:
Without the benefit of highly optimized performance libraries, developers would need to optimize computationally intensive functions themselves carefully to obtain adequate performance. This optimization process is complicated, time consuming, and must be updated with each new processor generation. Intelligent systems often have a long lifetime in the field and there is a high maintenance effort to hand-optimize functions.

Signal processing and advanced vector math are the two function domains that are most in demand across the different types of intelligent systems. Frequently, a digital signal processor (DSP) is employed to assist the general purpose processor with these types of computational tasks. A DSP may come with its own well-defined application interface and library function set. However, it is usually poorly suited for general purpose tasks; DSPs are designed to quickly execute basic mathematical operations (add, subtract, multiply, and divide). The DSP repertoire includes a set of very fast multiply and accumulate (MAC) instructions to address matrix math evaluations that appear frequently in convolution, dot product and other multi-operand math operations. The MAC instructions that comprise much of the code in a DSP application are the equivalent of SIMD instruction sets. Like the MAC instructions on a DSP, these instruction sets perform mathematical operations very efficiently on vectors and arrays of data. Unlike a DSP, the Single Instruction Multiple Data (SIMD) instructions are easier to integrate into applications using complex vector and array mathematical algorithms since all computations execute on the same processor and are part of a unified logical execution stream.

For example, an algorithm that changes image brightness by adding (or subtracting) a constant value to each pixel of that image must read the RGB values from memory, add (or subtract) the offset, and write the new pixel values back to memory. When using a DSP coprocessor, that image data must be packaged for the DSP (placed in a memory area that is accessible by the DSP), signaled to execute the transformation algorithm, and finally returned to the general-purpose processor. Using a general-purpose processor with SIMD instructions simplifies this process of packaging, signaling, and returning the data set. Intel IPP primitives are optimized to match each SIMD instruction set architecture so that multiple versions of each primitive exist in the library.

Intel IPP can be reused over a wide range of Intel® Architecture based processors, and due to automatic dispatching, the developer’s code base will always pick the execution flow optimized for the architecture in question without having to change the underlying function call (Figure 2). This is especially helpful if an embedded system employs both an Intel® Core™ processor for data analysis/aggregation as well as a series of Intel® Atom™ Processor based SoCs for data pre-processing/collection. In that scenario, the same code base may be used in part on both the Intel® Atom™ Processor based SoCs in the field and the Intel® Core™ processor in the central data aggregation point.

Both concepts often coexist in the same SoC. Code with failsafe real-time requirements is protected within its own wrapper managed by a modified round-robin real-time scheduler, while the rest of the operating system (OS) and application layers are managed using standard SMP multi-processing concepts. Intel Atom Processors contain two Intel Hyper-Threading Technology based cores and may contain an additional two physical cores resulting in a quad-core system. In addition Intel Atom Processors support the Intel SSSE3 instruction set. A wide variety of Intel IPP functions found at http://software.intel.com/en-us/articles/new-atom-support are tuned to take advantage of Intel Atom Processor architecture specifics as well as Intel SSSE3.
Figure 8: Intel IPP is tuned to take advantage of the Intel Atom Processor and the Intel SSSE3 instruction set.

Throughput intensive applications can benefit from the use of Intel SSSE3 vector instructions and parallel execution of multiple data streams through the use of extra-wide vector registers for SIMD processing. As just mentioned, modern Intel Atom Processor designs provide up to four virtual processor cores. This fact makes threading an interesting proposition. While there is no universal threading solution that is best for all applications, the Intel IPP has been designed to be thread-safe:

- Primitives within the library can be called simultaneously from multiple threads within your application.
- The threading model you choose may have varying granularity.
- Intel IPP functions can directly take advantage of the available processor cores via OpenMP*.
- Intel IPP functions can be combined with OS-level threading using native threads or Intel® Cilk™ Plus.

The quickest way to multithread an application that uses the Intel IPP extensively is to take advantage of the OpenMP* threading built into the library. No significant code rework is required. However, only about 15-20 percent of Intel IPP functions are threaded. In most scenarios it is therefore preferable to also look to higher level threading to achieve optimum results. Since the library primitives are thread safe, the threads can be implemented directly in the application, and the performance primitives can be called directly from within the application threads. This approach provides additional threading control and allows meeting the exact threading needs of the application.

Figure 9: Function level threading and application level threading using the Intel IPP

When choosing applying threading at the application level, it is generally recommended to disable the library’s built-in threading. Doing so eliminates competition for hardware thread resources between the two threading models, and thus avoids oversubscription of software threads for the available hardware threads.
Intel IPP provides flexibility in linkage models to strike the right balance between portability and footprint management.

<table>
<thead>
<tr>
<th></th>
<th>Standard Dynamic</th>
<th>Custom Dynamic</th>
<th>Dispatched Static</th>
<th>Non-dispatched Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizations</td>
<td>All SIMD sets</td>
<td>All SIMD sets</td>
<td>All SIMD sets</td>
<td>Single SIMD set</td>
</tr>
<tr>
<td>Distribution</td>
<td>Executable(s) and standard Intel IPP DLLs</td>
<td>Executable(s) and custom DLLs</td>
<td>Executable(s) only</td>
<td>Executable(s) only</td>
</tr>
<tr>
<td>Library Updates</td>
<td>Redistribute as-is</td>
<td>Rebuild and redistribute</td>
<td>Recompile application and redistribute</td>
<td>Rebuild custom library, recompile application, and redistribute</td>
</tr>
<tr>
<td>Executable Only Size</td>
<td>Small</td>
<td>Small</td>
<td>Large</td>
<td>Medium</td>
</tr>
<tr>
<td>Total Binary Size</td>
<td>Large</td>
<td>Medium</td>
<td>Medium</td>
<td>Small</td>
</tr>
<tr>
<td>Kernel Mode</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Intel IPP Linkage Model Comparison.

The standard dynamic and dispatched static models are the simplest options to use in building applications with the Intel IPP. The standard dynamic library includes the full set of processor optimizations and provides the benefit of runtime code sharing between multiple Intel IPP-based applications. Detection of the runtime processor and dispatching to the appropriate optimization layer is automatic.

If the number of Intel IPP functions used in your application is small, and the standard shared library objects are too large, using a custom dynamic library may be an alternative.

To optimize for minimal total binary footprint, linking against a non-dispatched static version of the library may be the approach to take. This approach yields an executable containing only the optimization layer required for your target processor. This model achieves the smallest footprint at the expense of restricting your optimization to one specific processor type and one SIMD instruction set. This linkage model is useful when a self-contained application running on only one processor type is the intended goal. It is also the recommended linkage model for use in kernel mode (ring 0) or device driver applications.
Currently, the Intel IPP is delivered for and validated against five different operating systems: Microsoft Windows*, Linux*, Mac OS* X, QNX Neutrino*, and Wind River VxWorks*. QNX* and VxWorks* are limited to single-threaded static variants of Intel IPP.

Application of the Intel IPP to a “non-supported” operating system requires that the OS is compatible with the Application Binary Interface (ABI) defined by one of the aforementioned five operating systems, and that memory allocation routines can be accessed through a standard C library or mapped via glue code using a special i_malloc interface.

The atomic nature (no locks or semaphores) of the Intel IPP function implementation means that it is safe to use in the deterministic environment of a RTOS. An example of the value of applying the Intel IPP library to an RTOS would be the TenAsys INtime* RTOS for Windows*. ([http://www.tenasys.com](http://www.tenasys.com)). The INtime* RTOS is an OS designed to run alongside Windows, handling real-time requirements on a Windows* based platform. The ABI used by INtime RTOS* is compatible with the Windows* ABI and employs Windows* compatible function calling conventions. Using the Intel IPP in conjunction with such an RTOS expands its appeal by providing the performance of SIMD-based data throughput intensive processing, with determinism usually only characteristic of DSPs.

Intel IPP addresses both the needs of the native application developer found in the personal computing world and the intelligent system developer who must satisfy real-time system requirements with the interaction between the application layer and the software stack underneath. By taking the Intel IPP into the world of middleware, drivers and OS interaction, it can be used for embedded devices with real-time requirements and dominant execution models as well. The limited dependency on OS libraries and its support for flexible linkage models makes it simple to add to embedded cross-build environments, whether they are RTOS specific or follow one of the popular GNU* based cross-build setups like Poky-Linux* or MADDE*.

Developing for intelligent systems and small form factor devices frequently means that native development is not a feasible option. Intel IPP can be easily integrated with a cross-build environment and be used with cross-build toolchains that accommodate the flow requirements of many of these real-time systems. Use of the Intel IPP allows embedded intelligent systems to take advantage of vector instructions and extra-wide vector registers on the Intel Atom Processor. Developers can also meet determinism requirements without increasing the risks associated with cross-architecture data handshakes of complex SoC architectures.
Developing for embedded small form factor devices also means that applications with deterministic execution flow requirements have to interface more directly with the system software layer and the OS (or RTOS) scheduler. Software development utilities and libraries for this space need to be able to work with the various layers of the software stack, whether it is the end-user application or the driver that assists with a particular data stream or I/O interface. The Intel IPP has minimal OS dependencies and a well-defined ABI to work with the various modes. One can apply highly optimized functions for embedded signal and multimedia processing across the platform software stack while taking advantage of the underlying application processor architecture and its strengths, all without redesigning and returning the critical functions with successive hardware platform upgrades.

**Intel® Math Kernel Library (Intel® MKL)**

Intel MKL includes routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. Intel MKL includes a C-language interface for the Discrete Fourier transform functions, as well as for the Vector Mathematical Library and Vector Statistical Library functions.

The Intel® Math Kernel Library includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
  - Vector operations
  - Matrix-vector operations
  - Matrix-matrix operations

- Sparse BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices)

- LAPACK routines for solving systems of linear equations

- LAPACK routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations

- Auxiliary and utility LAPACK routines

- ScaLAPACK computational, driver and auxiliary routines

- PBLAS routines for distributed vector, matrix-vector, and matrix-matrix operation

- Direct and Iterative Sparse Solver routines

- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces)

- Vector Statistical Library (VSL) functions for generating vectors of pseudorandom numbers with different types of statistical distributions and for performing convolution and correlation computations

- General Fast Fourier Transform (FFT) Functions, providing fast computation of Discrete Fourier Transform via the FFT algorithms and having Fortran and C interfaces

- Cluster FFT functions (only in Intel MKL for Linux* and Windows* operating systems)

- Tools for solving partial differential equations - trigonometric transform routines and Poisson solver

- Optimization Solver routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms and computing Jacobi matrix by central differences

*Figure 11: The Intel IPP can be used over a full range of development setups and software stack targets.*
- Basic Linear Algebra Communication Subprograms (BLACS) that are used to support a linear algebra oriented message passing interface
- Data Fitting functions for spline-based approximation of functions, derivatives and integrals of functions, and search

**Intel IPP and Intel MKL for Signal Processing**

Since this workshop is focused on signal processing, the next question is when to use one Fourier Transform over another with respect to Intel IPP and Intel MKL.

DFT processing time can dominate a software application. Using a fast algorithm, Fast Fourier transform (FFT), reduces the number of arithmetic operations from $O(N^2)$ to $O(N \log_2 N)$ operations. Intel MKL and Intel IPP are highly optimized for Intel architecture-based multi-core processors using the latest instruction sets, parallelism, and algorithms.

Read further to decide which FFT is best for your application.

**Table 2: Comparison of Intel MKL and Intel IPP Functionality**

<table>
<thead>
<tr>
<th></th>
<th>Intel MKL</th>
<th>Intel IPP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target Applications</strong></td>
<td>Mathematical applications for engineering, scientific and financial applications</td>
<td>Media and communications applications for audio, video, imaging, speech recognition and signal processing</td>
</tr>
</tbody>
</table>
| **Library Structure** | • Linear algebra  
• BLAS  
• LAPACK  
• ScaLAPACK  
• Fast Fourier transforms  
• Vector math  
• Vector statistics  
• Random number generators  
• Convolution and correlation  
• Partial differential equations  
• Optimization solvers | • Audio coding  
• Image processing, compression and color conversion  
• String processing  
• Cryptography  
• Computer vision  
• Data compression  
• Matrix math  
• Signal processing  
• Speech coding and recognition  
• Video coding  
• Vector math  
• Rendering |
| **Linkage Models** | Static, dynamic, custom dynamic | Static, dynamic, custom dynamic |
| **Operating Systems** | Windows*, Linux*, Mac OS X* | Windows, Linux, Mac OS X, QNX* |
| **Processor Support** | IA-32 and Intel® 64 architecture-based and compatible platforms, IA-64 | IA-32 and Intel® 64 architecture-based and compatible platforms, IA-64, Intel IXP4xx Processors |

**Intel MKL and Intel IPP Fourier Transform Features**

The Fourier Transforms provided by MKL and IPP are respectively targeted for the types of applications targeted by MKL (engineering and scientific) and IPP (media and communications). In the table below, we highlight specifics of the MKL and IPP Fourier Transforms that will help you decide which may be best for your application.
### Table 3: Comparison of Intel MKL and Intel IPP DFT Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel MKL</th>
<th>Intel IPP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API</strong></td>
<td>DFT&lt;br&gt;Cluster FFT&lt;br&gt;FFTW 2.x and 3.x</td>
<td>FFT&lt;br&gt;DFT</td>
</tr>
<tr>
<td><strong>Interfaces</strong></td>
<td>C and Fortran&lt;br&gt;LP64 (64-bit long and pointer)&lt;br&gt;ILP64 (64-bit int, long, and pointer)</td>
<td>C</td>
</tr>
<tr>
<td><strong>Dimensions</strong></td>
<td>1-D up to 7-D</td>
<td>1-D (Signal Processing)&lt;br&gt;2-D (Image Processing)</td>
</tr>
<tr>
<td><strong>Transform Sizes</strong></td>
<td>32-bit platforms - maximum size is $2^{31}-1$&lt;br&gt;64-bit platforms - $2^{64}$ maximum size</td>
<td>FFT - Powers of 2 only&lt;br&gt;DFT - $2^{32}$ maximum size (*)</td>
</tr>
<tr>
<td><strong>Mixed Radix Support</strong></td>
<td>2,3,5,7 kernels (**)</td>
<td>DFT - 2,3,5,7 kernels (**)</td>
</tr>
<tr>
<td><strong>Data Types</strong></td>
<td>Real &amp; Complex&lt;br&gt;Single- &amp; Double-Precision</td>
<td>Real &amp; Complex&lt;br&gt;Single- &amp; Double-Precision</td>
</tr>
<tr>
<td><strong>Scaling</strong></td>
<td>Transforms can be scaled by an arbitrary floating point number (with precision the same as input data)</td>
<td>Integer (&quot;fixed&quot;) scaling&lt;br&gt;- Forward 1/N&lt;br&gt;- Inverse 1/N&lt;br&gt;- Forward + Inverse $\sqrt{1/N}$</td>
</tr>
<tr>
<td><strong>Threading</strong></td>
<td>Platform dependent&lt;br&gt;- IA-32: All (except 1D when performing a single transform and sizes are not power of two)&lt;br&gt;- Intel® 64: All (except in-place power of two)&lt;br&gt;- IA-64: All</td>
<td>1D and 2D&lt;br&gt;- Can use as many threads as needed on MP systems.</td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>High accuracy/p&gt;</td>
<td>High Accurate</td>
</tr>
</tbody>
</table>

### Data Types and Formats

The Intel MKL and Intel IPP Fourier transform functions support a variety of data types and formats for storing signal values. Mixed types interfaces are also supported. Please see the product documentation for details.
Table 4: Comparison of Intel MKL and Intel IPP Data Types and Formats

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel MKL</th>
<th>Intel IPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real FFTs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>Single, Double</td>
<td>Single, Double</td>
</tr>
<tr>
<td>1D Data Types</td>
<td>Real for all dimensions</td>
<td>Signed short, signed int, float, double</td>
</tr>
<tr>
<td>2D Data Types</td>
<td>Real for all dimensions</td>
<td>Unsigned char, signed int, float</td>
</tr>
<tr>
<td>1D Packed Formats</td>
<td>CCS, Pack, Perm, CCE</td>
<td>CCS, Pack, Perm</td>
</tr>
<tr>
<td>2D Packed Formats</td>
<td>CCS, Pack, Perm, CCE</td>
<td>RCPack2D</td>
</tr>
<tr>
<td>3D Packed Formats</td>
<td>CCE</td>
<td>N/A</td>
</tr>
<tr>
<td>Format Conversion Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex FFTs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>Single, Double</td>
<td>Single, Double</td>
</tr>
<tr>
<td>1D Data Types</td>
<td>Complex for all dimensions</td>
<td>Signed short, complex short, signed int, complex integer, complex float, complex double</td>
</tr>
<tr>
<td>2D Data Types</td>
<td>Complex for all dimensions</td>
<td>Complex float</td>
</tr>
</tbody>
</table>

Formats Legend
CCE - stores the values of the first half of the output complex conjugate-even signal
CCS - same format as CCE format for 1D, is slightly different for multi-dimensional real transforms
For 2D transforms, CCS, Pack, Perm are not supported for 3D and higher rank
Pack - compact representation of a complex conjugate-symmetric sequence
Perm - same as Pack format for odd lengths, arbitrary permutation of the Pack format for even lengths
RCPack2D - exploits the complex conjugate symmetry of the transformed data to store only half of the resulting Fourier coefficients

Performance

The Intel MKL and Intel IPP are optimized for current and future Intel® processors, and they are specifically tuned for two different usage areas:

- Intel MKL is suitable for large problem sizes
- Intel IPP is specifically designed for smaller problem sizes including those used in multimedia, data processing, communications, and embedded C/C++ applications.

Choosing the Best FFT for Your Application

Before making a decision, developers must understand the specific requirements and constraints of the application. Developers should consider these questions:

- What are the performance requirements for the application? How is performance measured, and what is the measurement criteria? Is a specific benchmark being used? What are the known performance bottlenecks?
- What type of application is being developed? What are the main operations being performed and on what kind of data?
- What API is currently being used in the application for transforms? What programming language(s) is the application code written in?
- Does the FFT output data need to be scaled (normalized)? What type of scaling is required?
- What kind of input and output data does the transform process? What are the valid and invalid values? What type of precision is required?
6. **Power and Performance Tuning**

Power Tuning is one of the historically most neglected areas of embedded software development. Mostly it was seen as the realm of platform hardware design. With the need to squeeze ever more battery life out of our devices as well as meeting challenging thermal requirements, software level power tuning has however come into its own.

As in our previous discussions let us start with the available open source solutions. A good resource is the Less Watts initiative (http://www.lesswatts.org). It provides best known methods for reducing power consumption on an OS as well as application level. In addition you can download a range of utilities like PowerTop, the Battery Life Toolkit, and Power QoS. Have a look around and find the approach that is right for you.

Granola* from MiserWare* provides an integrated power management and power footprint solution that is primarily targeted at server type applications, but can be scaled for embedded use cases as well.

Intel recently added power analysis capabilities to its VTune™ Amplifier product as well, combining basic principles of tools like PowerTop with its background in mapping events (in this case frequency changes and power mode changes) to application and system software source locations in memory.

![Figure 12: Wake-Up Event and Sleep Mode Analysis](image)

In addition to direct power tuning, tuning for performance and thus having critical frequently used applications finish their work quickly and then allow the OS to revert to a lower power state quickly also has considerable impact on battery life of small form factor devices. Application and software stack performance is however also a desirable goal in its own right. Whether it is only to create a smoother user experience or in the case of dedicated signal processing to reduce the stress on the high performance embedded server processors doing all the work.

In the Linux* world OProfile* is probably the most used utility for advanced performance tuning. It comes with command-line sampling capabilities as well as Eclipse* Integrated Development Environment (IDE) integration.
The Intel® VTune™ Amplifier provides similar baseline functionality with its own graphical user interface (GUI). The key feature for performance analysis as well as power analysis in Embedded is the ability to support remote data collection and having a small footprint remote sampling collector. In addition being able to do system-wide sampling across the kernel space / user space barrier is important as many embedded applications have strong dependencies on device drivers that interact with dedicated micro-engines or other I/O devices. The solution to this is to have a processor Performance Monitoring Unit (PMU) driver implemented as a kernel module. This gives broader access to more performance relevant platform events like cache misses, branch mispredictions, memory writes and TLB lookups. In addition it allows looking at these from a whole software stack perspective and not just from the viewpoint of a single application as you would have with dynamic binary instrumentation based solutions. It also limits the memory impact of the performance sampling, an important consideration for many throughput optimized embedded use cases with memory intensive application workloads.

Figure 7 shows how a remote command line sampling collector interacting with a driver stub on the target allows for remote performance and power sampling without having to manually do result file transfers.

How to use power collection on Intel® VTune™ Amplifier XE 2013 for Intel® System Studio

The embedded OS we will be focused on is Yocto Project* version 1.2. This platform supports many Intel BSP’s and it also allows you to work without running any physical embedded hardware by letting you develop via an emulator that they provide. Here are the steps we will take to run our collection:

1. Setting up a Yocto 1.2 environment.
   a. Setting up your Linux host
b. Setting up a cross compilation environment
c. Setting up git inside the Intel firewall
d. Setup a full build of Yocto for your BSP
e. Building a Yocto kernel

2. Install VTune Amplifier XE 2013
   a. Cross build the sampling driver s(sep, vtss)
   b. Load drivers onto your device.
   c. Cross build the power driver

3. Setup target to not require password. (necessary for the remote collection script)
4. Run a remote collection using amplxe-runss.py
5. On your Linux host view the results in the Intel®VTune™ Amplifier XE 2013 GUI

Note: steps 1, 2 and 3 are one time steps. Once you have the sampling driver built and loaded on your system you should be enabled to collect performance data.

Setting up a Yocto 1.2 environment

1. Download the pre-built toolchain, which includes the runqemu script and support files
download from: http://downloads.yoctoproject.org/releases/yocto/yocto-1.2/toolchain/
   a. The following tool chain tarball is for a 32-bit development host system and a 32-bit target architecture: poky-eglbc-i686-i586-toolchain-gmae-1.2.tar.bz2
   b. You need to install this tar ball on your Linux host in the root "/" directory. This will create an installation area "/opt/poky/1.2"

2. Setup your Linux host system:
   a. For my Ubuntu x64 12.04 system I ran the following command to setup my system.
      $ sudo apt-get install sed wget cvs subversion git-core coreutils \
       unzip texi2html texinfo libsd1.2-dev docbook-utils gawk \
       python-pysqlite2 diffstat help2man make gcc build-essential \
       g++ desktop-file-utils chrpath libg1-mesa-dev libglu1-mesa-dev \
       mercurial autoconf automake groff
   b. See the Yocto getting started guide for more information on the setup required for various Linux distros:
      http://www.yoctoproject.org/docs/1.0/yocto-quick-start/yocto-project-qs.html

3. Build Yocto
   a. Download the Latest stable Yocto build system.
      i. wget http://downloads.yoctoproject.org/releases/yocto/yocto-1.2.1/poky-denzil-7.0.1.tar.bz
   b. tar xjf poky-denzil-7.0.1.tar.bz2
   c. source poky-denzil-7.0.1/oe-init-build-env poky-denzil-7.0.1-build
   d. Edit poky-denzil-7.0.1/build/conf/local.conf
      i. Tailor MACHINE for the bsp you want to build.
      ii. In my case I am building fri2-noemgd
   e. Edit poky-denzil-7.0.1/build/conf/bblayers.conf
      i. Specify the meta-intel you checked out.
      ii. Specify the specific BSP meta directory. (meta-intel/meta-fri2)
   f. Build Yocto
      i. bitbake core-image-sato
      ii. This will create a kernel that is sufficient to build sep and vtss.

Install Intel® VTune™ Amplifier XE 2013

1. Install Intel® VTune™ Amplifier XE 2013 on your Linux host. VTune™ Amplifier XE is a part of the Emberson suite of tools.
2. You will need to build the sampling driver and load it on your target in order to collect performance data.
   a. cd $SEP_LOCATION/sepdk
   b. For example, if you are building a Yocto 1.2 build of the fri2-noemgd board support package then your build command would be similar to the following:
      ./build.driver -ni --compiler=i586-poky-linux-gcc \
      --kernel-src-dir=/yocto/poky-denzil-7.0/build/tmp/work/fri2_noemgd-poky-linux/linuxyocto3.2.11+git1+5b4c9dc78b5ae607173cc3ddab9bce1b5f78129b_r1/linux-fri2-noemgd-standard-build

21
Note: To find the kernel version, see $kernel-src-dir/include/linux/utsrelease.h

3. Load sampling driver on your target
   a. scp -r $SEP_LOCATION root@target_ip:/home/root
   b. Login to your target
   c. cd /home/root/sep/sepdk
   d. ./insmod-sep3 -re
   e. You may need to also install the vtsspp collector separately. On some Yocto platforms the shell does not provide the capabilities to load the drivers via a script.
      i. cd /home/root/sep/sepdk/vtsspp
      ii. insmod vtsspp.ko
   f. Verify that the sep and vtsspp drivers loaded.
      i. lsmod | grep sep
      ii. lsmod | grep vtsspp

4. Install remote collection binaries on target
   a. Copy the contents of /opt/intel/vtune_amplifier_xe_2013/target/linux32 or linux64 to your target. Note: if your target is 32 bit you will copy the linux32 directory and for 64 bit targets you will copy linux64.
      i. scp linux32 root@ip_target:/tmp/linux32

5. Cross build the power driver
   a. cd /opt/intel/vtune_amplifier_xe_2013/powerdk
   b. source /opt/poky/1.2.1/environment-setup-i586-poky-linux
   c. ./build-driver --kernel-src-dir=./yocto/poky-denziel-7.0/build/tmp/work/fri2_noemgd-poky-linux/linuxyocto3.2.11+git1-5b4c9dc78b5ae607173cc3ddab9bce1b5f78129b_1+76dc683eccc4680729a76b9d2fd425ba540a483-r1/linux-fri2-noemgd-standard-build --kernel-version=3.2.18-yocto-standard

6. Load the power driver on your target
   a. scp /opt/intel/vtune_amplifier_xe_2013/powerdk/src/apwr_3.ko root@target_ip:/tmp
   b. On target
      1. Load power driver
         i. insmod /tmp/apwr_3.ko
      2. Verify power driver loaded
         i. lsmod | grep apwr

Setup target to not require a password

1. Make sure your root directory has a .ssh directory and only the owner had read/write/execute access on it.
2. cat ~/.ssh/id_dsa.pub | ssh root@ip_target " cat >> /home/root/.ssh/authorized_keys"
3. On the target chmod 600 /home/root/.ssh/authorized_keys
4. Verify you can now login without password
   a. ssh root@ip_target

Run Intel® VTune™ Amplifier XE 2013 remote collector on your application

1. Set up some target environment variables
   a. export AMPLXE_TARGET_PRODUCT_DIR=/tmp/linux32
   i. From the location you copied the remote collection binaries.
   b. export AMPLXE_TARGET_TMP_DIR=/tmp
2. Run collection
   a. /opt/intel/vtune_amplifier_xw_2013/target/amplxe-runss.py --target=root@ip_target -pwr-config=sleep,frequency --result-dir r000 @@@ -- /tmp/hello
      i. This will run a remote power collection on your target, 10 seconds. It will also create a result directory r000 that you can open on your host.

On your Linux host: View the VTune™ Amplifier XE results

1. amplxe-gui r000
Run power collection using wuwatch

1. Verify the version of wuwatch you are running uses a compatible power driver.
2. Assuming wuwatch is installed on your system
   a. wuwatch --cs --ps -t 60
      i. This command will collect sleep C states and frequency P states. For 60 seconds.
      ii. After the command is run you will have a wuwatch_output.txt and wuwatch_output.ww1
   b. Copy the wuwatch data back to your host
   c. Run the command
      i. amlx-runss --import-wuwatch-data /host/path/wuwatch_output.ww1
      ii. This will create a result directory that you can view using the GUI.

7. Summary

When first starting on an Intel® Embedded software development project based on signal processing, the requirements may seem confusing. It is easy to get distracted by the complexity of the different requirements instead of focusing on the simple basics of defining your setup. This paper and complementing workshop gives a step-by-step guide of the following

- Intel® Architecture in Embedded Devices and Intelligent Systems
- Options for Cross–Development for Intelligent Systems
- Intel® System Studio
- Reliability and Debug
  o Application Debug
  o System Software Debug
  o Event Trace Debug
- Signal Processing Libraries for Intel® Embedded Architectures
- Power and Performance Tuning

The key to being successful when developing for Intel® Architecture 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.nasoftware.co.uk/home/attachments/019_Atom_benchmarks.pdf

Intel® Embedded Design Center http://edc.intel.com


References


Yocto Project*: http://www.yoctoproject.org

IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5412866

About the Author

Noah Clemons is a Technical Consulting Engineer for the Embedded Compute, Debugger, and Libraries Division at Intel Corporation. In 2011 he earned the Intel Developer Zone Black Belt award for excellence in teaching programing models across the entire spectrum of Intel designs – from low-power SoCs all the way up to large clusters. His expertise lies in performance libraries and parallel models to solve computational problems in image/signal processing, scientific computing, and three-dimensional visualization. He previously worked for two successful startups, the latter as a Solutions Architect for RapidMind which was acquired by Intel in 2009.

Notices

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