Bare-Metal, RTOS, or Linux? Optimize Real-Time Performance with Altera SoCs

WP-01245-1.0

Abstract

This white paper examines various methods for optimizing real-time performance on Altera® SoCs, which integrate an FPGA and applications processor into a single chip. Standard software development models using high-level operating systems are compared to an ideal, hand-optimized, bare-metal solution running on the applications processor, while latency and interrupt jitter due to the interaction with the FPGA are explored. Given the system interaction complexities of a modern applications processor, it is shown that modern real-time high-level operating systems can provide the same level of real-time performance as a hand-optimized solution, but with the system stability and design reuse benefits of modern software development methodologies over hand-optimized bare-metal applications.

Introduction

Real-time systems may offer uncompromising hard real-time requirements where the jitter on deadline absolutely has to be within a certain bound. In some cases, failure to do so could result in serious injury or death. Others present soft real-time requirements, such as optimized energy efficiency, which will not introduce catastrophic failure, but are still very important over a long period of operation. Either way, it is important to understand the exact real-time response of a given system architecture in terms of real-time loop latency, jitter, and other requirements. Many system designers initially think that implementing a "no-OS" or "bare-metal" system will inherently be lighter and, therefore, faster and less intrusive than a full operating system (OS). With the advent of high-performance applications processors in today's systems, however, this is not necessarily true. Running an application on a very high speed and capable application processor running a rich RTOS may actually give better response time than a bare-metal implementation. Which one to choose? This white paper examines the response time of different OS implementations. A real-time "DataMover" application example is created to test the different implementations and yields some insightful results.

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Real-Time System Requirements and Challenges

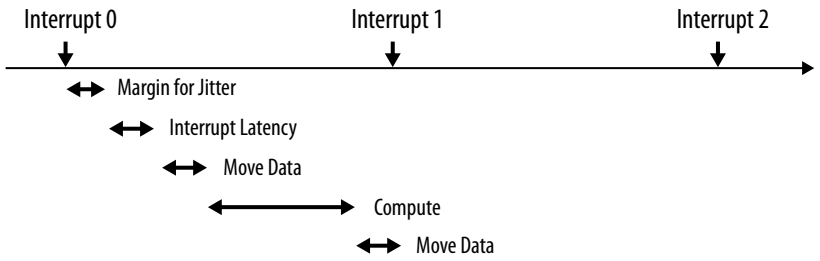
Industrial control applications offer an exemplary case study for real-time application requirements. A typical industrial control application contains both real-time tasks and non-real-time tasks. The real-time tasks handle external interrupts, either via register polling or interrupt servicing, that occur on the order of 10's of microseconds, i.e., to respond to the interrupt, to move the necessary data associated with each interrupt, to do computation and return the results before the next interrupt occurs. To ensure real-time response, the jitter cannot exceed a few microseconds. Often times, users wish to group all the real-time processing to one core for more direct control, and in the hope of obtaining higher performance. The non-real-time tasks typically include housekeeping tasks, networking and user interface. In most of these systems, there is little to no sharing of peripherals between the processor cores, but it is necessary to share some common memory buffers for synchronization, communication, or data to be displayed.

Implicitly, there are other requirements to ensure the success of any electronics product. These requirements begin with ease of programming. Being able to program a multicore processor using simple, documented, and proven solutions is key for productivity and the project schedule. Minimizing risk is also important. Risk can come in the form of known risks and unknown risks. It is important to adopt engineering practices that remove known risks and minimize unknown risks. Ecosystem support is essential to increase both ease of programming and for minimizing risks. The ecosystem implicitly enables users to benefit from the collective wisdom of the whole, in this case ARM, development community. Lastly, to make sure that a design can move to the next more powerful, more core processor quickly, and to benefit from new innovations in software from the worldwide developer community, it is critically important to have a design that is portable in hardware and software, which often means programming above an OS abstraction.

Measuring Real-Time Performance

Real-time response time and jitter tolerance requirements dominate most real-time design decisions. Real-time response time is typically expressed in terms of a real-time loop during which the system has to handle an interrupt and perform all the requisite computing before the next interrupt arrives. This is shown in **Figure 1**.

Figure 1. Real-Time Loop



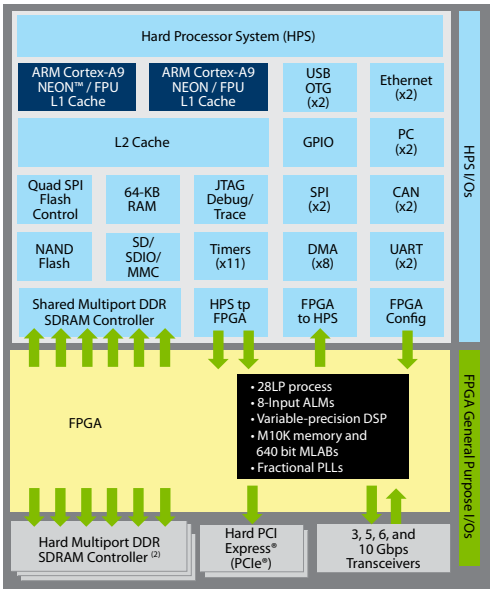
The real-time loop time can vary from ~1 microsecond (ms) in a software real-time system, to 10's of microseconds in a real-time system or in single microseconds in a very high-performance hard real-time system. Because real-time requirements dictate most design decisions, real-time loop time is employed as the metric for performance in evaluating different system architectures.

Introducing the Altera SoC

Semiconductor integration capabilities have reached the point where high-performance application processors, such as the ARM® Cortex®-A9 processor, are cost-effectively integrated with varying FPGA sizes.

The Altera® Cyclone® V SoC brings together an integrated high-performance application processor with integrated FPGA fabric. See **Figure 2**.

Figure 2. Altera Cyclone V SoC Block Diagram

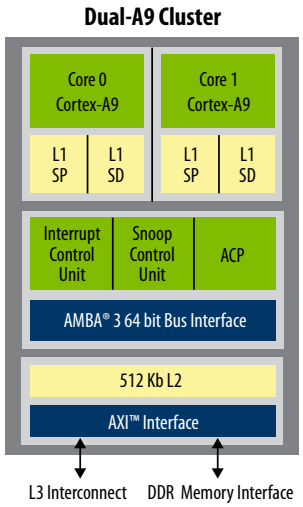


This combination provides users with a wide range of flexibility. Functions can be run on the integrated application processor, an instantiated soft processor in the FPGA array, or a state machine in the FPGA fabric. However, this increased flexibility comes with a challeng—how to take advantage of this increased capability while managing the increased complexity—putting more pressure on system partitioning to achieve the maximum capability of the silicon, in both a time and engineering resource efficient manner.

SMP and AMP on the Altera SoC

The dual-core ARM Cortex-A9 processor present in the SoC hard processor system (HPS) shown in **Figure 3** is tightly coupled using the ARM MPCore™ technology, in a classical symmetric multiprocessing (SMP) hardware configuration. Mature, proven software solutions exist today from many ARM ecosystem software providers to enable asymmetric multiprocessing (AMP) on this platform, including methods such as SMP with Core Affinity or Core Reservation, making programming an AMP system as simple as programming a single-core CPU. Empirical data for symmetric multicore systems presented later in this paper (**Figures 5, 6, 7, and 8**) shows that employing well-defined programming methods, such as Core-Affinity and Thread-Lock, generally outperforms programming each core independently. By adopting standard, proven solutions, users can get the best combination of performance, productivity, system reliability, and future scalability.

Figure 3. SMP Dual-A9 Cluster



At the chip-level, an SoC presents a heterogeneous multicore system that is AMP in hardware configuration by definition. Implementations of one or more soft core processors, hardware accelerators or other custom computing units in the FPGA makes asymmetric multiprocessing "at will", thereby enabling greater flexibility with Altera's SoCs compared to fixed SoCs. A very practical use of the FPGA is to augment the dual-A9 cluster in terms of real-time processing.

Real-Time Application Example

To have the data for an objective evaluation of different real-time OS configurations, an example of real-time application was constructed. This benchmark application is designed to embody the characteristics of real-time applications. The system runs on an Altera Cyclone V SoC development board, utilizing both the dual-A9 cluster and the FPGA. There is a small direct memory access (DMA) design that runs on the FPGA and works in tandem with the A9 cluster to move the data to and from the FPGA. This design is referred to as the "DataMover" design.

The system receives interrupts from the FPGA. With each interrupt, some data is sent from the FPGA to the HPS for handling, which involves a small amount of computation, after which some results are written back to the FPGA. The tasks that simulate the interrupt handling, the data moving, and the return of the data to the FPGA are collectively referred to as the "real-time tasks". The round-trip loop time and jitter are measured as an indication of the system's real-time responsiveness. Interrupts are handled both as interrupt service routines, and via interrupt polling. The system also has a number of non-real-time tasks which are simulated by an OS continuously generating Fibonacci series.

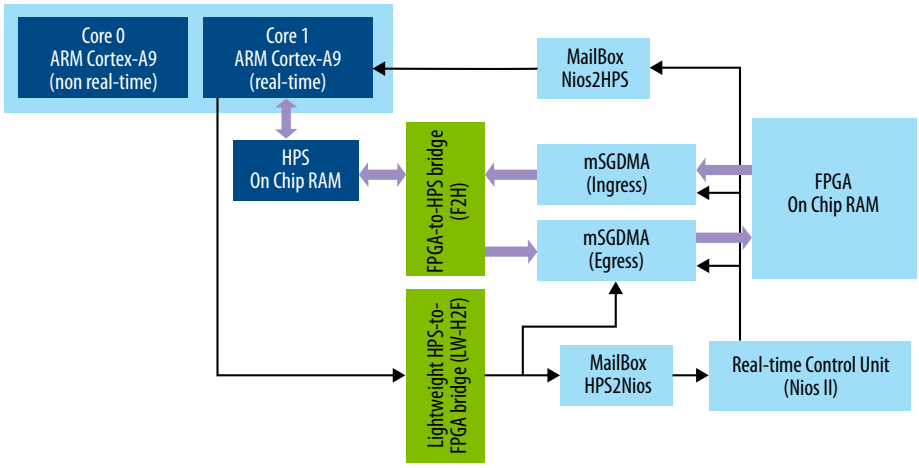
This application software is implemented in the following ways to compare results between three different software configurations:

- Software Architecture 1: Linux SMP
  - Running over Linux in SMP with core affinity mode, on both cores of the dual-A9 cluster
  - Core 1: Utilizes the DataMover for data movement and interrupt handling
  - Core 0: Idle or busy running a continuous Fibonacci series to simulate non-real-time tasks
  - Interrupts are handled in polling or interrupt service routine
- Software Architecture 2: VxWorks SMP
  - Running over VxWorks in SMP core affinity mode, on both cores of the dual-A9 cluster
  - Core 1: Utilizes the DataMover for data movement and interrupt handling
  - Core 0: Idle or busy running a continuous Fibonacci series to simulate non-real-time tasks
  - Interrupts are handled in polling or interrupt service routine
- Software Architecture 3: Bare-Metal Single-Core
  - Running in bare-metal mode on one core only
  - Core 1: Utilizes the DataMover for data movement and interrupt handling
  - Core 0: Not utilized to represent a "best case" environment (no overhead or buffer management)
  - Interrupts are handled in polling or interrupt service routine

System Architecture

Figure 4 shows the reference design hardware design.

Figure 4. Real-Time Application System Architecture Example



In this system, external interrupts are simulated by data stored on an on-chip RAM implemented in the FPGA. The size of the data can be changed easily to study the relationship between data size and efficiency of DMA operations.

Simulated interrupt handling begins with a DMA operation, which can be initiated by the DMA controller itself, the Cortex-A9 processor, or the real-time control unit (in this case, utilizing a Nios® II soft core processor). Data is moved by the DMA from FPGA to the HPS on-chip RAM.

Core 1 of the dual-A9 cluster is configured to poll the on-chip RAM to see if new data has appeared and if so, it will copy the data to a private storage, and copy the result back to memory. Polling mode operation, in a system that requires microsecond real-time response, is both appropriate and optimal because it is more deterministic and takes a fraction of the time of interrupt handing. Data is also handled via an interrupt service routine (ISR) as a comparison as well.

The real-time control unit can be as simple as a custom DMA dispatcher intellectual property (IP), or as sophisticated as a soft processor-based system. This can be a fixed-function hardware state machine that simply knows how to sequence and orchestrate the DMA logic that captures and generates the data in the FPGA. This could also be something like a soft core processor that is implemented in the FPGA to run software algorithms that determine how the data flow to and from the FPGA must be orchestrated. The data collected in this paper makes use of a soft-core Nios II processor-based design.

The simulated real-time loop is completed by the DMA moving the data from HPS memory back to the FPGA.

Software Application Example

The reference hardware design uses the following OS versions:

- Linux LTSI v3.10
- VxWorks v6.9
- Bare-metal using hardware libraries in SoC Embedded Design Suite 14.0 as the foundation

The example application basically runs a loop of continuous real-time data request, processing and return. The input data is sent from the FPGA via the FPGA-to-HPS bridge with cache coherent access to the A9 processor. The pseudo code of the application is shown below.

```
While(1){

    Real_input = Request_a_data(); // Request a new data from FPGA

    While(keytoken(real_input) != expected_token); // Wait for the new data to be valid.
    We embed a token in every data to know the validity of the new data

    Request_data_back(real_out); // Request to get the result sent back to the FPGA
```

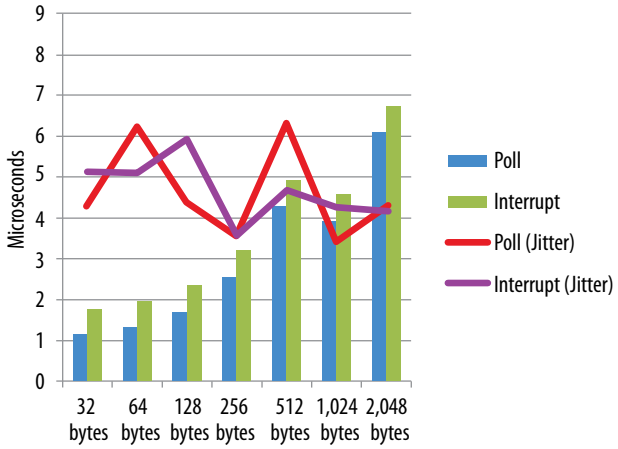
Test Results

Standard performance counters were used to capture the total loop time. Due to nanosecond-level measurements being taken, in order to account for the variance in the counter granularity, each result is an average of 100 measurements taken. No special IP was required to be built in order to achieve this level of measurement.

Software Architecture 1: Linux SMP

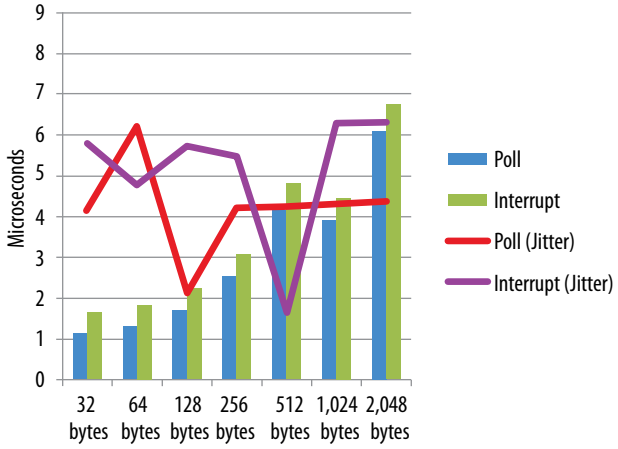
The results for an idle Linux system, where Core 1 is handling the real-time tasks in a polling or ISR method, while Core 0 is idle, are shown in Figure 5.

Figure 5. Linux SMP with Core 0 Idle



The results for a busy Linux system, where Core 1 is handling the real-time tasks in a polling or ISR method, while Core 0 is calculating a Fibonacci series, are shown in Figure 6.

Figure 6. Linux SMP with Core 0 Busy

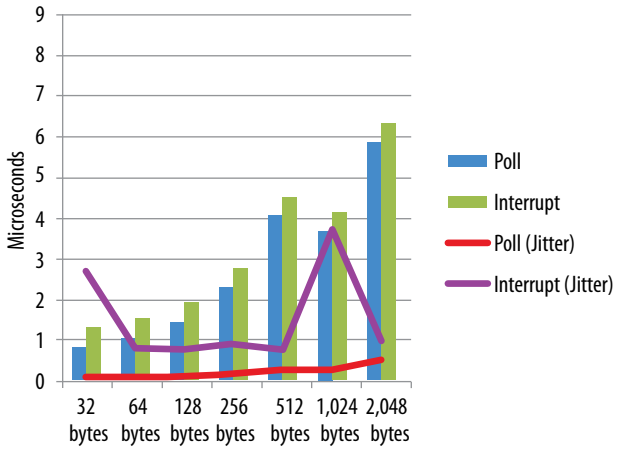


The results are predictable, with no discernible difference in a busy or idle Linux system. Polling does yield a slightly better overall response, while jitter in both polling and ISR are relatively high (< 10 microseconds). In a system where interrupt jitter of 10 microseconds or higher is acceptable, a standard Linux SMP system is sufficient.

Software Architecture 2: VxWorks (RTOS) SMP

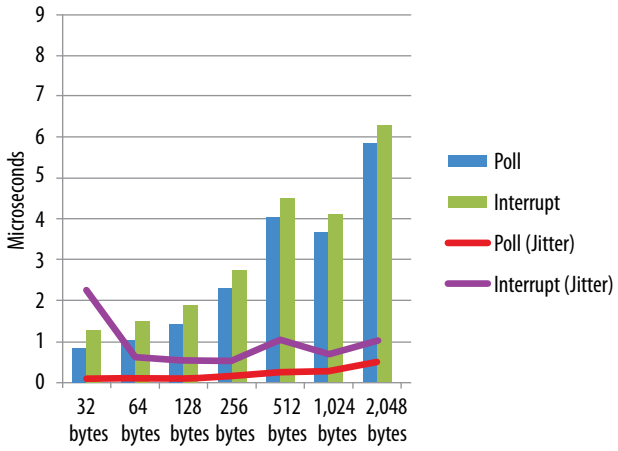
The results for an idle VxWorks system, where Core 1 is handling the real-time tasks in a polling or ISR method, while Core 0 is idle, are shown in **Figure 7**.

Figure 7. VxWorks SMP with Core 0 Idle



The results for a busy VxWorks system, where Core 1 is handling the real-time tasks in a polling or interrupt service routine method, while Core 0 is calculating a Fibonacci series, are shown in **Figure 8**.

Figure 8. VxWorks SMP with Core 0 Busy

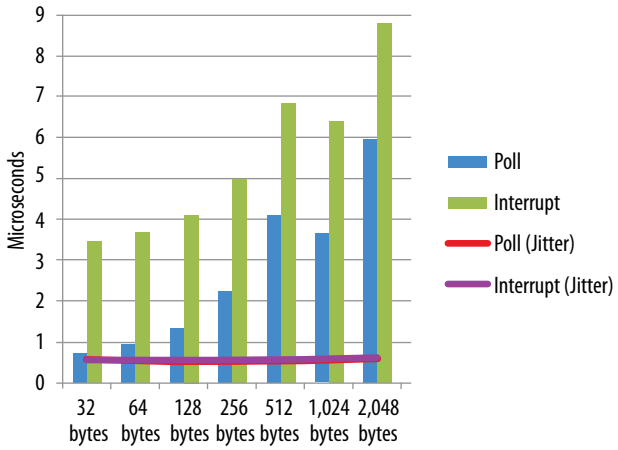


The results here are also predictable. There is no significant difference in a busy or idle VxWorks system in terms of loop time in a busy or idle system. There is less variation in overall jitter however, in a busy VxWorks system at a 1k data size.

Software Architecture 3: Bare-Metal Single-Core

A bare-metal system was created to run on one core; this core does nothing else except handling interrupts, in a polling or ISR method. Since the other core is not used, there is no idle or busy system comparison. This would represent a "best case" scenario in terms of response, as any overhead or managing of the common buffers between CPUs is not taken into account. It would be expected that such a system could see significant increases in jitter based on how active the communication between cores would be. The results are shown in **Figure 9**.

Figure 9. Bare-Metal Single Core



As expected, jitter is less than a microsecond in both the polling and ISR methods, as nothing else is occurring in the system. Overall loop time is on-par with both Linux and VxWorks for a polling method, however, the ISR response is noticeably higher. This can be attributed to the fact that no scheduling or pre-emption of events is present in a bare-metal system. Thus, there is no performance advantage from programming in bare-metal.

Besides the difficulties in creating a bare-metal application for an A9-class multicore processor, the resulting product is hardware specific, and therefore not readily portable to future, more-core or different-core processors. Bare-metal applications are therefore not future proof. By comparison, applications that run on top of an operating system are abstracted away from hardware differences and therefore are readily portable to future devices.

Key Findings from Measuring Real-Time Loop Time

The top most contributing factors to lowering interrupt response time and minimizing jitter are:

- Configuring the system to run in SMP mode with core affinity and dedicate Core 1 to real-time processing. This conclusion holds true for both VxWorks and Linux. Configuring the processor to run in core affinity mode dramatically increased the real-time processing performance on Core 1. While it is also true that this approach decreases the processor's ability to load balance between the two cores, and therefore reduces Core 0's ability to do its work, if real-time responsiveness is the single most important requirement to meet, then this is the easiest and safest way to accomplish that goal.

- Core 1 is used to poll for interrupt instead of using the processor's native interrupt handling mechanism. Detractors can point out that polling causes a system to run all the time, consuming more power, but if microsecond response time is needed, then the processor is already running all the time. Therefore, polling is a very good way to remove the non-determinism of the A9-MPCore interrupt handling process.

- Using the DataMover DMA design in the FPGA to deliver data to the on-chip RAM where Core 1 can find and use with least overhead. Depending on the actual requirement, defines how sophisticated you make the DMA design. A simple DMA was employed, using Core 1 to start and stop it. Alternatively, you can design a DMA or DMA with a Nios II processor to take more load off Core 1.

Design Best Practices for Programming a Multicore SoC

Taking into consideration all the design requirements of real-time responsiveness at the microsecond level, the desire to separate real-time task and non-real-time tasks and to balance those requirements with the need to minimize design risk and increase design portability, a summary of all the best design practices to consider are:

- Use all available hardware resources to increase system performance. This is illustrated by the DataMover application system architecture (figure 4) that utilized a FPGA-based DMA to move data in real-time and in parallel with the processing of the data. To overcome the non-deterministic characters of the Cortex-A9 processor interrupt polling was employed. Used in aggregate, this simple system produced microsecond level real-time responsiveness with minimal jitter.

- Pair the default SMP hardware architecture with an SMP operating system for best balance between performance and development time. This point is supported by the examples where standard configurations of VxWorks and Linux were employed to create two similarly high-performance systems, with the only variance being in interrupt response jitter.

- Choose the right RTOS as the easiest way to ensure real-time performance and determinism – VxWorks was employed to prove that getting deterministic performance is very simple, in fact, right out of the box.

- Adopt multicore programming techniques to achieve concurrency, load balancing, and future-proof scalability. The DataMover design (figure 4) shows how the FPGA can be used to increase overall system concurrency, in a way that a fixed SoC cannot do. The system uses standard off-the-shelf SMP operating systems, this design can scale easily to future more-core processors.
- Use SMP/Core Affinity to separate real-time and non real-time tasks on different cores to achieve application level AMP when necessary. Even though a SMP operating system was paired with a SMP processor core, core affinity feature was employed to effectively accomplish asymmetric processing of the application, giving users what is desired – being able to separate real-time tasks from non-real-time tasks, and most importantly, to reduce jitter in order to produce deterministic real-time responsiveness.
- Benefit from ARM software ecosystem by adopting proven, widely accepted solutions – employing Linux and VxWorks in the most standard configurations, the design is easily supported by software and software partners. As these operating systems evolve and improve, the system will take advantage of it, right out of the box.

Conclusion

The most significant driver for hand-creating run-time software is for its perceived performance benefit and cost. Given the complexities of modern applications processors, it is very difficult to create a stable, hand-optimized solution without the use of a modern OS. As shown by the test results in this white paper, even if such a system can be created, it performs no better than an OS-based solution given the system interactions both at the processor level, and with the FPGA. There are numerous free or low-cost OS solutions available. In terms of performance, with the scheduling demands of modern applications processors, most OS have already been fully tuned to take advantage of the processor architecture, which would have to be re-developed in a hand-crafted bare-metal solution. By using a proven OS as the run-time software, application developers can focus on system-level optimizations. When true hard real-time performance is required, the FPGA provides an excellent target. If a soft processor-based hard real-time solution is desired, a Nios II soft processor core instantiated within the FPGA provides the most optimal path.

Acknowledgements

Chee Nouk Phoon, Embedded Software Engineer, Altera Corporation

Chei Siang Ng, Embedded Applications Engineer, Altera Corporation

Steve Jahnke, Embedded Software Product Planning, Altera Corporation

Findlay Shearer, Linux Marketing Manager, Altera Corporation