Blade Management Controller Rides FPGA Embedded Processor

CloudShield builds powerful, flexible solution around Xilinx PowerPC 440 running embedded Linux.

by Andy Norton
Distinguished Engineer, Office of the CTO
CloudShield Technologies, Inc.
ANorton@CloudShield.com

Jeff Mullins
Principal Engineer, Embedded Systems Lead
CloudShield Technologies, Inc.
JMullins@CloudShield.com

Dick Mincher
Embedded Systems Specialist
CloudShield Technologies, Inc.
DMincher@CloudShield.com

The trend toward converged networks for both telecom and enterprise simplifies network infrastructure and dramatically lowers costs while providing scalable, open platforms. Blade platforms are achieving network convergence using Ethernet, scaling from 1G/10G to 40G/100G, with blade-management controllers keeping pace by using standards-based Intelligent Platform Management Interfaces (IPMI).

To help drive network convergence into the mainstream market, our design team here at CloudShield Technologies architected a flexible blade-management controller by integrating the PowerPC® 440 found in the Virtex-®5 with standard and custom peripheral cores, creating multiple embedded-system configurations with a common hardware design. This FPGA-based design took advantage of a unique and flexible feature set that included system reconfiguration, powerful embedded processors, intellectual-property (IP) cores and embedded Linux firmware for chassis management services unique to the resident blade.

Xilinx Platform Studio cores allowed us to quickly instantiate UARTs, I²C buses, memory interfaces, Ethernet MAC/PHY combinations and a system monitor for voltage and temperature oversight. We easily created custom cores to provide novel system acceleration, offloading real-time tasks. Our application implemented standards-based IPMI and chassis management firmware running over embedded Linux, for a robust and open software infrastructure.

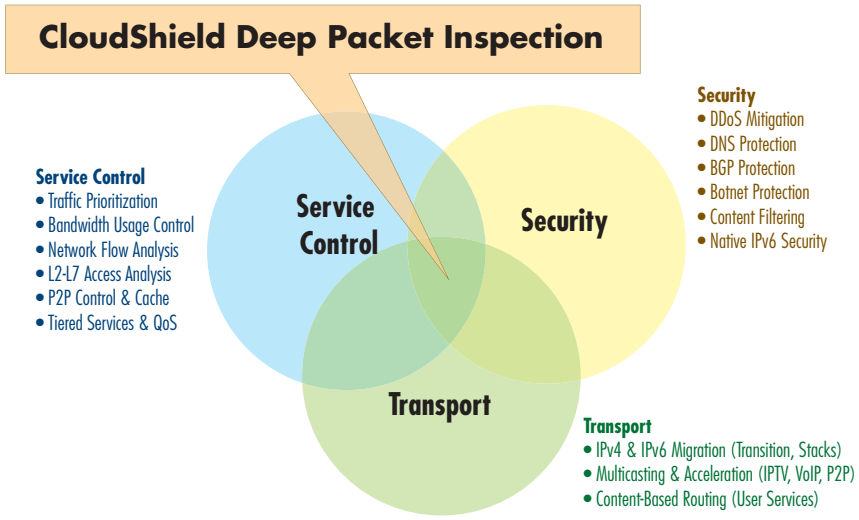

Figure 1 — Next-generation deep-packet inspection processing system enables convergence of service control, security and transport functions into integrated services and capabilities.

Deep-Packet Processing

Our next-generation content-processing platform is a highly programmable, modular, high-speed hardware system capable of handling a wide range of network applications and multiple simultaneous functions, as shown in Figure 1. The chassis integrates deep-packet processor blades, application-services processor blades, redundant chassis management modules and an array of interface modules that can be deployed with different network interface options and configurations.

Chassis and blade management are under the control of redundant chassis management modules using a variant of the Intelligent Chassis Management Bus (ICMB) that extends IPMI intelligent platform management to meet highly secure requirements. Each blade or module of this bus—except for the power supply module—contains an IPMI management controller to share environmental monitoring and status with the chassis management module.

Sharing common requirements, our family of management controllers consists of chassis management controllers (CMCs), processor management controllers (PMCs) on processor blades and interface management controllers (IMCs) on interface modules, as shown in Figure 2.

The chassis' extensive networking features support four control-plane Gigabit Ethernet links to each blade. The data plane has 16 high-speed lanes to each blade that provide four 10-Gbit Ethernet links using XAUI (4 x 3.125 Gbits/second). Future bandwidth scalability exists, with each lane capable of 10GBASE-KR (single lane, 10.3125 Gbps). We used Xilinx® Virtex-5 FPGAs to implement control-plane Gigabit Ethernet functions with hard trimode Ethernet media-access controllers (TEMACs) as well as data plane 10-Gbit Ethernet using the Xilinx 10GEMAC IP core with integrated XAUI (using RocketIO™ serial transceivers).

Blade Management Services

Our chassis management solution is a hierarchical structure, with the IMC and PMC entities operating in much the same way as a standard IPMI baseboard management controller (BMC) for their respective module or blade. For each blade, the controllers use IPMI commands to monitor voltage and temperature, provide a power control interface, enable blade network interface configuration and allow interrogation of manufacturing data.

As the center of chassis management, the CMC shares common IPMI BMC board-level features with the IMC and PMC, along with additional higher-level chassis management functionality. The dual redundant CMCs operate in an active/standby capacity with automatic health monitoring and failover. This chassis management capability allows the CMC to discover, inventory and provide power permissions for all blades and modules. To prevent oversubscription of the available power supplies, the active CMC regulates power demand for the entire chassis by allowing the IMCs and PMCs to power on only when capacity remains. The fan tray and power supply modules use a chassis I²C bus where the active CMC ensures sufficient power and cooling for the entire chassis.

Once the CMC detects a new blade in the chassis, it uses a discovery protocol to locate and identify each blade. The discovery protocol exchanges IPMI messages to retrieve serial number, model number and inventory for the blade or module, and then sends a command with power permissions if the blade can be correctly identified and power is available. The CMC monitors events from each IMC or PMC to detect warning and critical conditions, collecting and storing them in an event log.

To provide a rich user feature set, our primary user interface to the chassis management system is through a window management or command line interface (WMI or CLI) running on an application processor blade. The application processor blade either occupies a chassis slot or remotely communicates through a control-plane network connection to the chassis management controller. The CMC autonomously provides chassis oversight for health and

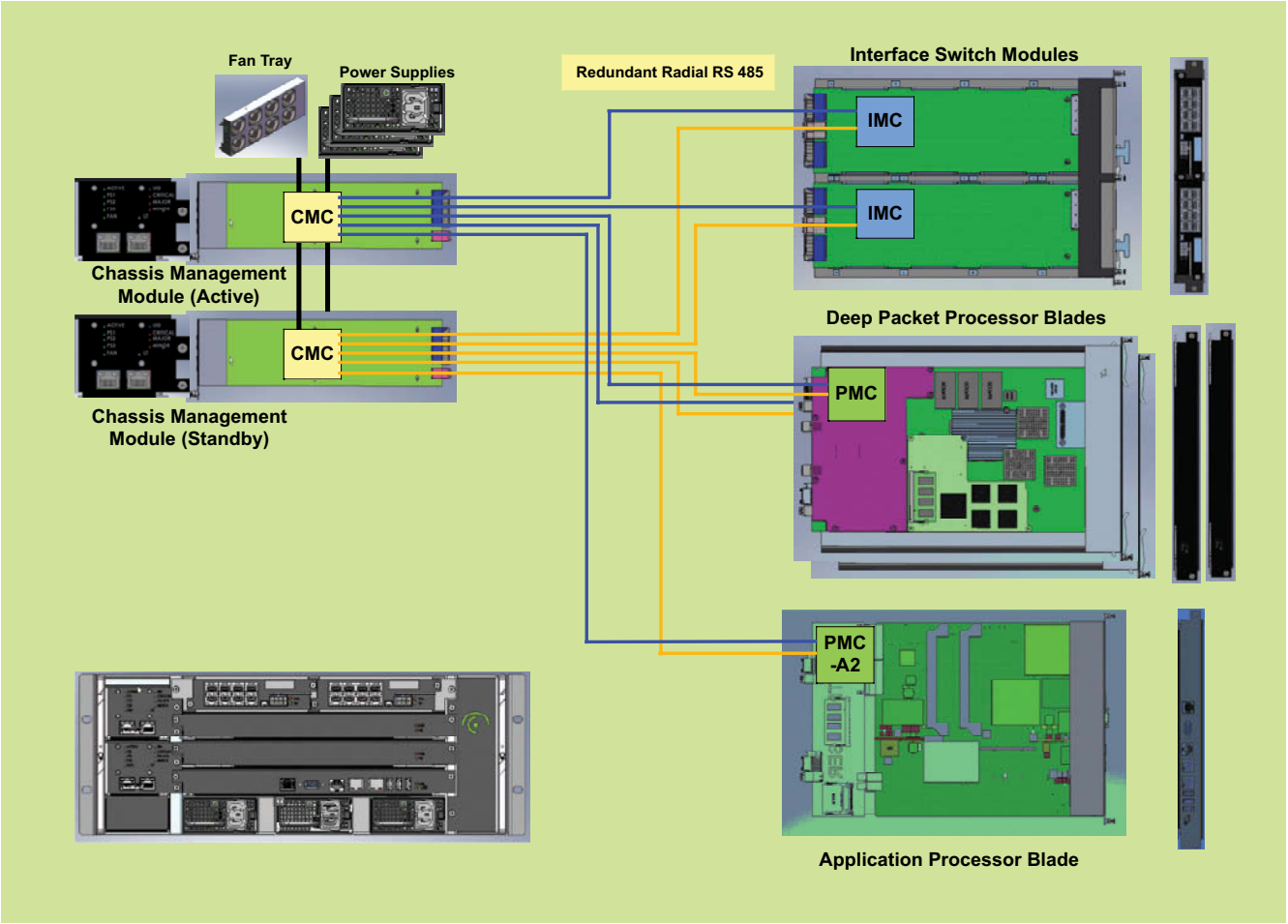

Figure 2 – Secure ICMB chassis management system

environmental monitoring, presenting this information through front-panel LEDs, while application-specific user control comes from the application processor blade. This same blade interrogates all blades and modules, including the CMC, to provide the user with environmental as well as operational health and status. The CMC contains an RS485 IPMI proxy service to pass application processor blade commands on the control plane through to the RS485 IPMI path to the target blade.

Each CMC has five independent point-to-point RS485 links to each blade. Unlike a bus topology, this architecture provides increased security from eavesdropping, minimizes contention and increases reliability, in that a failing blade cannot disrupt communication to other blades. Our redundancy implementation requires the standby CMC to constantly monitor the health of the active CMC. Upon detection of a failure, the standby CMC can take over, with its independent set of radial RS485 links, to keep the chassis operational.

Management Controller Hardware Architecture

Sharing common hardware functionality, our controllers include FPGA multiboot and fallback capability, PowerPC440 processor, SDRAM and NOR flash memory configuration, interrupt controller, serial console UART, ICMB RS485 custom core, Gigabit Ethernet links, system monitor sensors and local I^2C buses.

Our controller family is uniquely architected to share a common hardware platform using a single microprocessor hardware specification file that defines the embedded-processor core instantiations, connectivity and parameterization. Our mini boot loader resides in internal block random-access memory (BRAM), which copies the U-Boot boot loader from flash to DDR memory and then jumps to the U-Boot entry point.

Taking advantage of the features of the Virtex-5 FX30T FPGA, CloudShield's team easily assembled an advanced embedded system, integrating the PowerPC 440, DDR2 memory controller, multiport memory controller for flash, Ethernet MAC/PHYs, system monitor and other standard cores with our custom core.

By leveraging the Virtex-5's multiboot reconfiguration, the system achieves true in-system FPGA upgradability without a processor to manage bitstream loading. Our external flash supports both "golden" and "upgrade" bit files located at addresses the revision select pins determine. The FPGA's ability to fall back and reconfigure using the golden image in the event of an upgrade image failure was essential to our design objective of a configurable high-availability system.

Post-configuration, the PowerPC 440 core operates at 400 MHz, the crossbar and MPLB bus at 133 MHz, TEMACs at 125 MHz and DDR2 at 266 MHz. Our constructed hardware system covered the superset of functionality needed among the CMC, IMC and PMC variants as shown in Figure 3. The flash contains enough space for multiple application programs, including the CMC, IMC and PMC configurations.

We were able to rapidly craft the base system thanks to the wide variety of standard peripheral cores available in the IP library. We selected cores as complex as Gigabit Ethernet (with choices of 1000BASEX, SGMII and GMII PHY interfaces) and as simple as traditional 16550 UARTs to meet our system requirements.

Key requirements for a blade controller involved chip- and board-level monitoring and alarms for voltages and temperatures. We took advantage of an integrated system monitor, simply instantiating the SYSMON_ADC pcore from the standard IP library. The system monitor is a hard macro on Virtex-5 devices that contains a 10-bit, 200-kilosample/s analog-to-digital converter, supporting both on-chip and off-chip monitoring of supply voltages and temperatures.

Off-the-shelf IP and the availability of an Avnet Virtex-5 FXT evaluation kit that used the XC5VFX30T FPGA allowed us to rapidly prototype our design and kick-start the software development effort right out of the box.

With the software effort under way, we then created custom IP cores to satisfy a number of needs. While off-the-shelf pcores might meet functional requirements, multiple instances of simple pcores such as GPIO can result in excessive Processor Local Bus (PLB) loading as well as higher slice utilization. Rather than incur the expense of multiple IP interfaces, we built a single super-GPIO module with the help of the Create Import Peripheral (CIP) wizard integrated into XPS. Amortizing a single IP interface over multiple GPIO registers, we collapsed large numbers of internal and external registers into a single custom pcore including revision, scratch, and internal and external GPIO registers.

We also used the CIP wizard to build our special-purpose custom IP. The wizard created the necessary files (MPD, PAO, HDL templates) and directory structure, and offered easy selection of IP interface services and user logic interfaces. It also generated the bus functional model needed for simulation. Our novel PicoBlaze™-based coprocessor pcore is described later in greater detail.

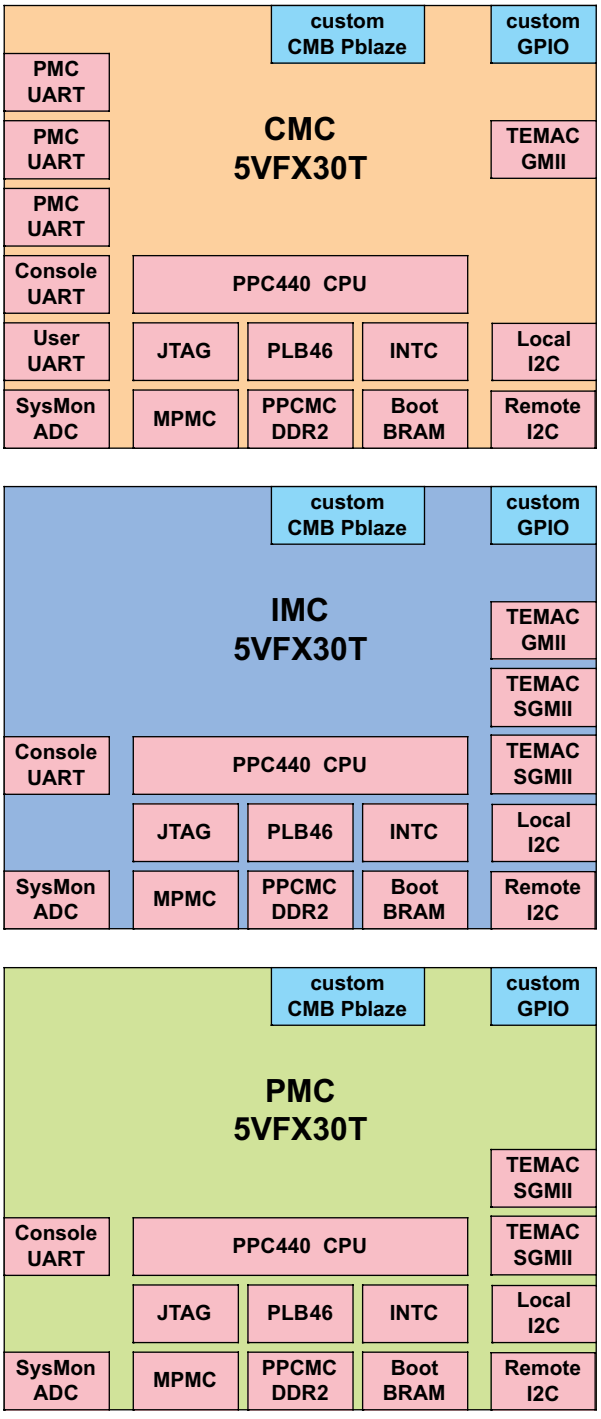

Figure 3 – Common hardware architecture for CMC (top), IMC (center) and PMC versions.

The embedded Linux operating system could not natively guarantee interrupt latency. Our solution leveraged the capabilities of the PicoBlaze 8-bit sequencer.

IPMI Messaging

Our chassis management controller with its corresponding system interfaces is shown in Figure 4. We use the custom ICMB coprocessor pcore for IPMI messaging to the processor-management and interface-management blade controllers. GPIOs monitor and control the front panel, backplane, redundancy and interlocks, as well as internal registers, while TEMACs are used for control-plane Ethernet switch management. Our I^2Cs read the local module EEPROM, and handle remote monitoring of chassis fans and power supplies. UARTs take care of front-panel user connectivity, debug consoles and proxy UARTs to CPUs on processor blades.

The real-time protocol requirements of the Intelligent Chassis Management Bus presented us with a problem in that the embedded Linux operating system could not natively guarantee interrupt latency.

Our solution uses a custom ICMB pcore with the PicoBlaze 8-bit sequencer as an ICMB data-link and physical-layer coprocessor offloading the PowerPC. This underappreciated resource (see *Xcell Journal*, issue 67, page 53, "A Hidden Gem: PicoBlaze IP") is shown in Figure 5, coupled with BRAM instruction store, registers, communication FIFOs, UARTs and timers. After initialization, the PowerPC holds the PicoBlaze in reset, loads code into the PicoBlaze program BRAM and then removes reset, activating the coprocessor.

We wrote our PicoBlaze code in the form of a state machine to monitor the transmit FIFO and control registers, implement the collision-avoidance and back-off algorithms, and manage five UARTs.

The TX FIFO passes 8-bit data from the PowerPC to the PicoBlaze for transmission on the ICMB bus. To transmit a packet, the PowerPC writes the packet into the TX

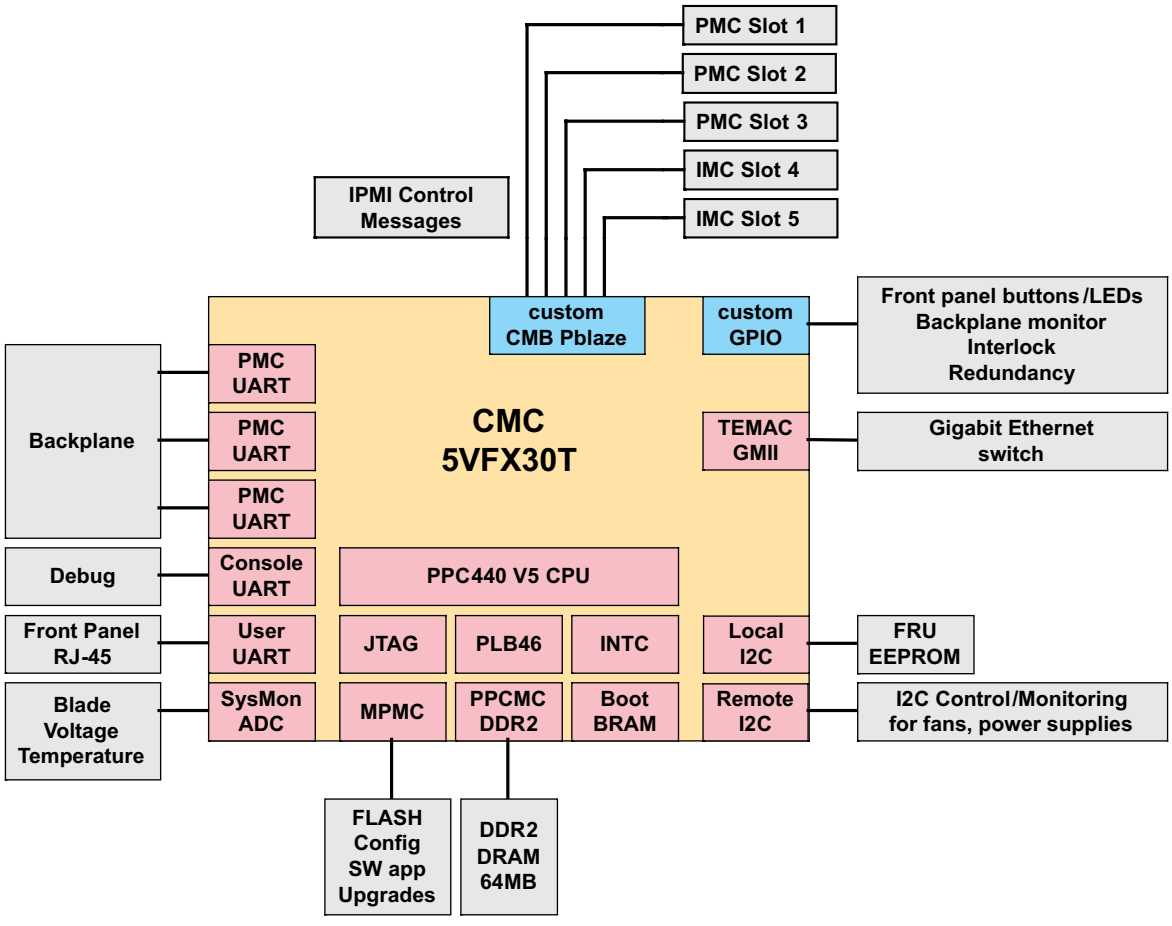

Figure 4 – CMC system interfaces

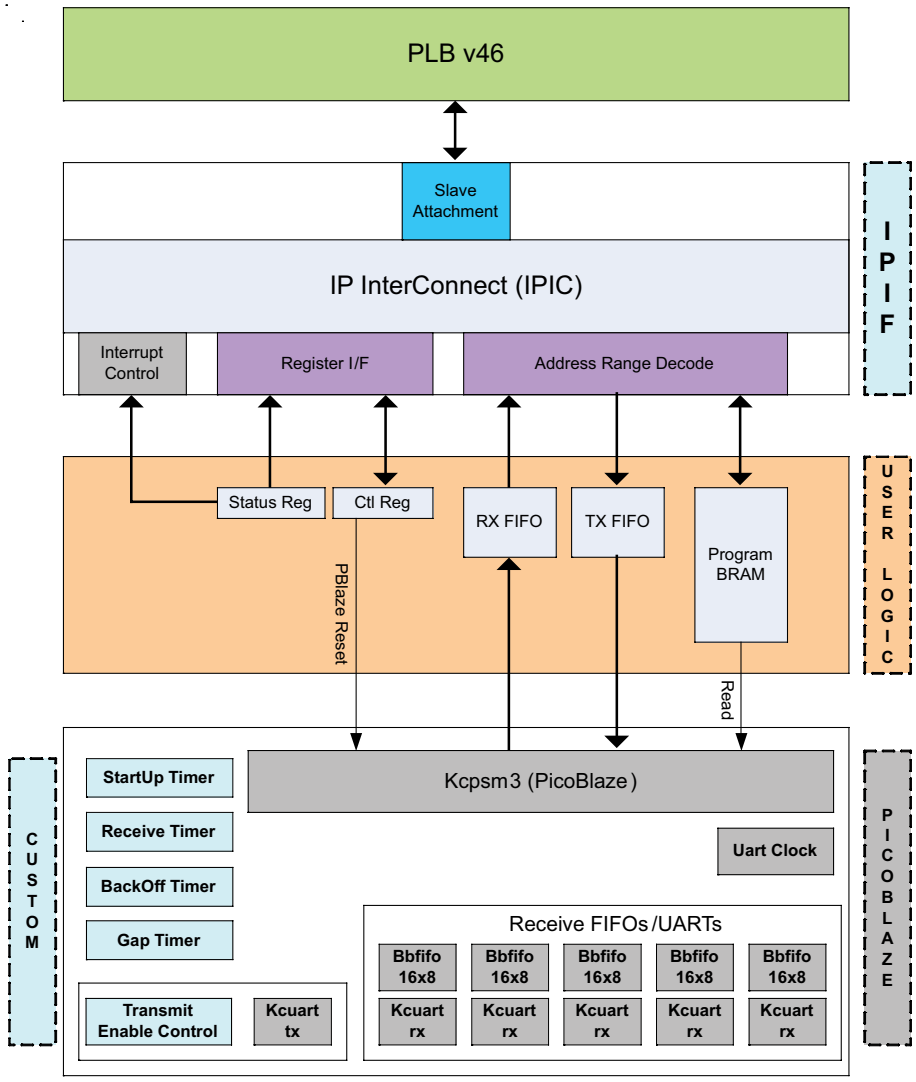

Figure 5 – Custom ICMB PicoBlaze pcore

FIFO, followed by a write to the control register with a "start" bit and channel number. Using a common transmit UART and individual transmit enables, the PicoBlaze enables a single RS-485 transceiver for data transmission. Upon completion, the PicoBlaze indicates success or failure by writing to the status register and generating an interrupt to the PowerPC.

Packets come in on multiple UARTs simultaneously. The PicoBlaze maintains individual state information for each channel. Indications of start/end of packet and data bytes are placed in the RX FIFO along with the channel number, and the PowerPC gets an interrupt when data is available. The RX FIFO passes 8-bit data, 5-bit status (start/end of packet, error indications, debug information) and 3-bit channel number information from the PicoBlaze to the PowerPC. The PowerPC then demultiplexes packet data and, when valid, sends it up to the servicing agent. We included additional trace and state transition capability in the receive channel in order for the PowerPC to print debug messages to the console.

We used the CIP wizard to create a bus functional model, enabling us to generate bus stimulus and verify the operation of the peripheral core without needing to create an extensive test bench or full-system simulation. The wizard made it easy to generate, respond to and analyze bus transactions. As a result, we created and modified our simulation for the custom pcore and got it running in a matter of hours.

We used the output of the PicoBlaze assembler to initialize the program BRAM. Next, using the Bus Functional Language to describe PLB bus transactions, we generated a PicoBlaze reset, sent data to the TX FIFO and wrote to the control register to transmit data on a channel. Looping back transmit and receive ports was a simple way to verify the operation of the receive channels and RX FIFO.

Firmware Architecture

There are many embedded-OS choices available for the PowerPC 440 and, specifically, the Xilinx hard PPC440 core. All of the management controllers required real-time performance, a robust network stack and extensibility for our custom peripherals. Our choice of embedded Linux allowed us to take advantage of existing open-source resources, utilities, optimizations and support.

After reset, the PowerPC boot sequence begins from reset vector space within a small internal BRAM. Mapped into this area is a small mini boot loader (12 kbytes), which looks for a valid U-Boot image at an "upgrade" or "golden" location within the NOR flash. Since the golden image is programmed only at the factory during the manufacturing process, this choice eliminates the possibility of a failure during flash programming of a new U-Boot version. Once it detects and validates the proper U-Boot image, the PowerPC loads it into SDRAM and executes it. U-Boot then loads the proper Linux image files and passes control for the final time. At this point, a fully configured Linux kernel loads and begins our application-specific processes.

We considered several options for the boot sequence during the design process. For example, the mini boot loader could have mapped the NOR flash into the reset vector area and eliminated the BRAM. This would have saved some BRAM, but the processor would then be dependent on a valid NOR flash part and image, which could be corrupted during a

failed flash attempt. Similarly, a case could be made for eliminating U-Boot by loading and executing Linux directly. While this could be a much more direct boot sequence, we found the features of U-Boot during the development phase to be extremely valuable for prototype bring-up and debugging.

Our initial design challenge was to locate and choose the correct open-source trees. DENX Software Engineering (*www.denx.de*) has extensively developed the Das U-Boot boot loader for a variety of embedded-processor boards. But while it directly supports the MicroBlaze, PowerPC 405 and PowerPC 440 processor cores, the DENX version does not include the latest patches, board support or drivers for some of the XPS IP cores that come with the EDK.

Therefore, we chose to use the Xilinx development branch of the U-Boot tree (*xilinx.wikidot.com*), which contains the latest changes and the drivers for the I^{2}C, ENET and LLTEMAC, providing additional functionality without starting from scratch. Xilinx frequently updates its development branch from the original DENX master tree, submitting changes upstream for acceptance into the DENX tree.

Having selected the Xilinx U-Boot tree, we looked at the main Linux tree (*www.kernel.org*) and its support for the Xilinx drivers. We similarly concluded that we obtained more working functionality when choosing the Xilinx-maintained Linux development branch (also at *xilinx.wikidot.com*), which included the latest changes that may not necessarily be in the main Linux tree.

A potential drawback of using the Xilinx-maintained trees is that there is lag time in terms of synchronization with the main trees, and they are officially development-quality branches. Nevertheless, we selected the Xilinx trees to support the LLTEMAC cores and newer features in our design.

In order to build U-Boot and Linux, we needed a Linux distribution and machine to run the scripts and tools that cross-compile, link and build the PowerPC executable. While this could be a barrier if only Windows-based machines were available, there are now a few Windows virtual machines that run Linux within the Windows environment. The added benefit is that once we set up a build machine, we could back up the virtual-machine disk image or replicate it for use on other development machines. We selected Sun xVM VirtualBox as a lightweight, reliable and easy-to-use virtual machine that will run many different OS types, including Linux.

While the Linux kernel provided an excellent foundation, the actual drivers and applications handled the entire workload specific to our application. The Xilinx I^{2}C and LLTEMAC drivers connect the Linux IP stack to the hardware, and the CloudShield ICMB driver hooks the custom PicoBlaze IP core into the Linux driver subsystem. We used Busybox as a standard Linux application, and added three new CloudShield applications for chassis management by the CMC, as shown in Figure 6.

Our RS485 proxy application connects to a standard IP port and communicates over the control-plane Ethernet, decoding each packet. Packets received from our WMI/CLI on the application processor blade may be addressed to three different targets in the system: BMC functions on the chassis management module board, CMM blade management for the chassis or other blades within the chassis. All baseboard management controller functionality runs over the I^{2}C buses, handling CMC and chassis health requests. The RS485 proxy application translates packets destined for other blades in the chassis into ICMB packets and sends them over the RS485 link to the appropriate destination. Blade management packets allow the user to obtain chassis status, such as power subscription, as viewed by the CMC. Some additional minor applications for redundancy and front-panel serial console interaction complete the CMC.

Our IMC and PMC were very similar to the CMC firmware, except that we replaced the blade-management application with a switch- or processor-management application appropriate for that blade.

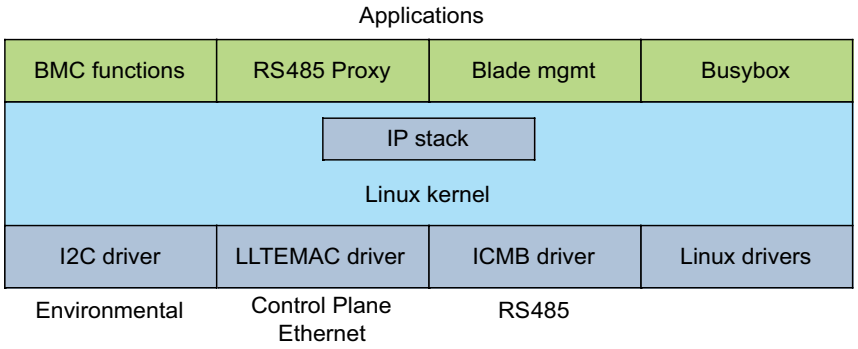

Figure 6 – CMC firmware architecture

Architect, Build, Verify, Deliver

The key to rapidly architecting, developing and delivering complex embedded-system solutions is "right-sizing" the choices, efforts and methodology. The combination of powerful embedded processors in FPGAs, off-the-shelf IP and accelerated design and verification of custom IP enabled us to create multiple embedded-system configurations with a common hardware design. The benefits of using Linux and an integrated development environment freed us to focus on the target application while utilizing the available open-source services, drivers and libraries.

Indeed, the ability to rapidly alter internal FPGA embedded architecture, easily deploy in-system upgrades and exploit new hardware-software trade-off boundaries is transforming embedded-system design.