Accelerating Bare Metal Driver Development with Linux Drivers and System Verilog DPI-C.

Abstract – Developing bare metal device drivers is critical yet time-consuming for IP verification, system-level validation, and embedded development. Despite mature Linux drivers existing for virtually all peripheral hardware, their tight OS integration prevents direct bare metal use. Our systematic framework leverages System Verilog DPI-C interfaces to convert Linux drivers into functionally equivalent bare metal drivers. Our approach introduces innovative hardware abstraction wrappers with DPI-C interfaces replacing kernel services, expert-guided driver transformation preserving core device logic, and comprehensive validation ensuring zero functional regressions. We implemented this process for a USB4 bare metal driver, demonstrating 83% development time reduction from 6-8 months to 1 month, reusing 39,000 lines of verified code while maintaining full functional equivalence. This approach can be applied to convert any Linux device driver to bare metal driver, significantly accelerating development across various hardware platforms. The converted drivers can be used in hardware/software co-verification, pre-silicon debug, SoC bring-up, and virtual prototyping environments, making them highly relevant for emulation and system-level validation.

I. Introduction

Bare metal drivers are crucial in IP verification environments where Linux kernel overhead is prohibitive and direct hardware control is essential. Traditional bare metal driver development suffers from long development cycles, limited code reuse, high expertise requirements, and extensive testing for IP specification compliance. Conversely, the Linux kernel contains mature, thoroughly tested drivers for virtually every hardware peripheral, embodying years of development, optimization, and bug fixes. However, these drivers are architected for Linux's specific infrastructure and cannot be used directly in bare metal environments. Our innovative framework systematically decouples device-specific logic from OS dependencies, enabling proven Linux driver logic reuse in bare metal environments. By leveraging the mature Linux driver ecosystem, our framework ensures functional equivalence and maintains performance characteristics required for verification environments. This approach applies to various hardware platforms, significantly accelerating bare metal driver development.

II. Current State And Challenges In Bare Metal Driver Development

Understanding Linux driver interactions with peripheral hardware is crucial for appreciating the complexity of converting these drivers to bare metal environments. In the Linux ecosystem, driver-device interaction involves multiple system components:

1. **Device Discovery:** Linux kernel scans PCIe bus during boot, discovers devices by vendor/device ID
2. **Driver Binding:** Kernel matches devices discovered with registered drivers, calls probe() function.
3. **Resource Management:** Driver calls pci_enable_device(), pci_request_regions() to allocate hardware resources
4. **Memory Mapping:** ioremap() creates virtual memory mappings enabling register access via ioread32()/iowrite32()
5. **Interrupt Setup:** request_irq() registers interrupt handlers with kernel's IRQ subsystem
6. **Runtime Operation:** Device driver performs register read/write through kernel's memory management, with interrupts routed through kernel IRQ handling.

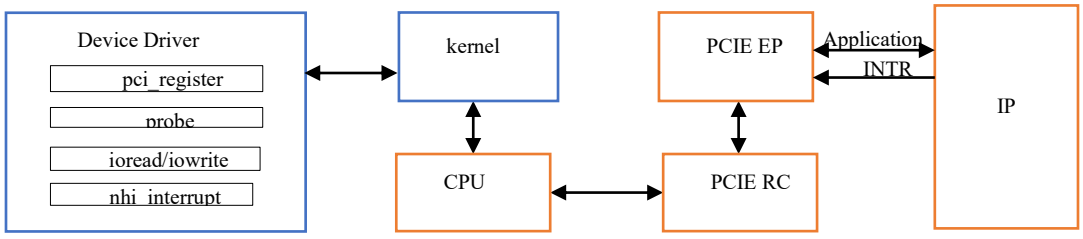

Fig1. Driver interaction with Device in Linux eco system.

A. Challenges in Bare Metal Driver Development:

These interactions underscore Linux driver complexity, and it's tight coupling with the kernel, making direct bare metal reuse challenging. In addition, the following critical components are missing in bare metal environments (simulation, emulation, embedded systems without OS),:

- **No Linux Kernel:** No device enumeration, memory management, or interrupt routing
- **No PCIe Infrastructure:** No PCIe root complex, endpoint discovery, or configuration space access
- **No System Services:** No virtual memory management, interruption handling, or resource allocation

This infrastructure gap explains bare metal driver development's time-consuming nature and significance of our DPI-C bridge approach.

III. Framework and Methodology for Linux to Bare Metal Driver Conversion

A. System Architecture:

Our framework replaces the missing Linux infrastructure with three interconnected components that recreate the Linux driver environment in bare metal:

1. **Kernel Proxy:** This component is a C++ class that "pretends" to be the Linux kernel, providing the same services (memory allocation, device mapping, interrupt handling) but implemented for bare metal.
2. **DPI-C Interface:** This "magic bridge" bridges software calls to hardware registers and routes hardware interrupts back to software
3. **Hardware Wrapper:** This module translates DPI-C calls into actual AXI/APB transactions to the USB4 IP.

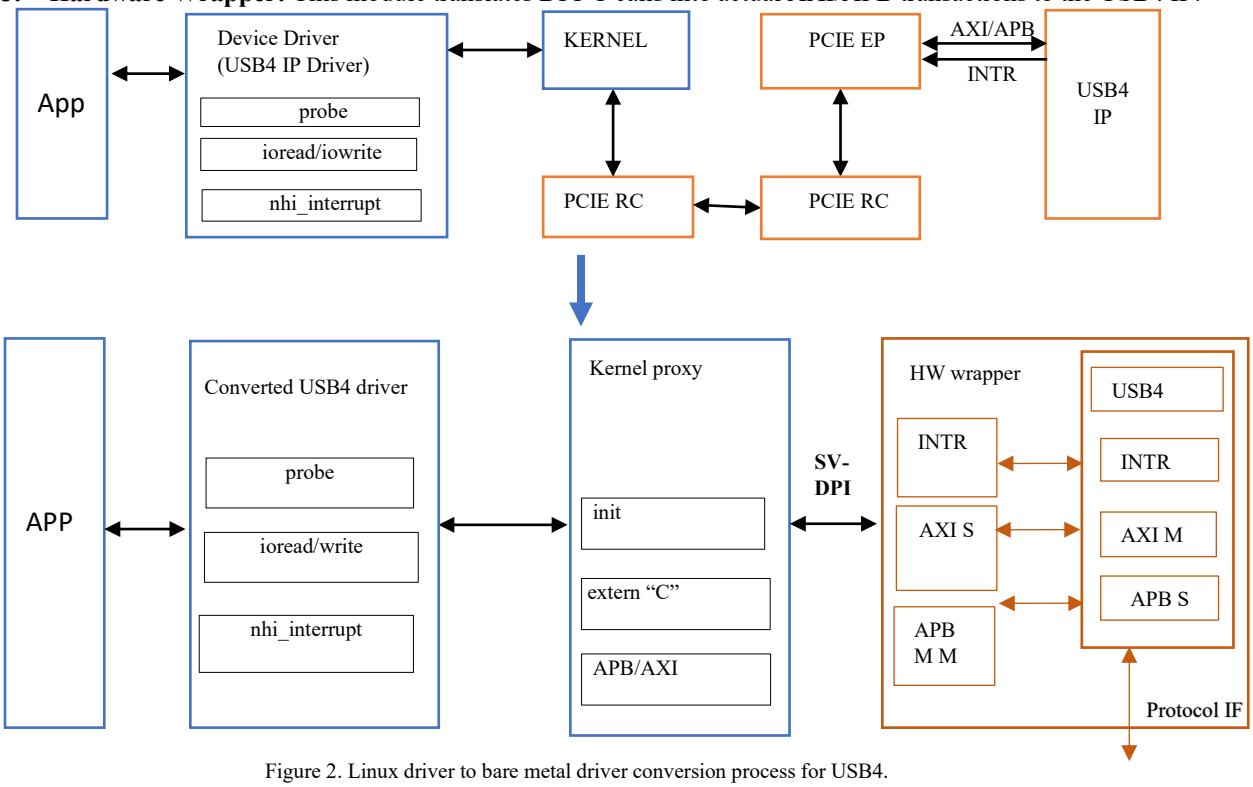

Figure 2. Linux driver to bare metal driver conversion process for USB4.

B. Methodology for Transforming Linux Drivers for Bare Metal Environments:

1. **Device driver and Kernel Proxy:**
   a. Identify various calls related to memory, PCI, interrupts, device management, and synchronization within the device driver. Since PCIe and Linux kernel code are unnecessary for a bare metal environment, comment them out or create empty functions within a kernel proxy class. Substitute memory read/write operations, interrupt requests, and other hardware interaction requests with alternate definitions suited for bare metal usage.
   b. Develop extern "C" functions facilitating efficient communication between driver and the hardware wrappers.
2. **H/W Wrapper:**

a. Since bare metal software needs direct hardware register access and interrupt handling without Linux kernel and PCIe infrastructure, develop a System Verilog module. This module will wrap the USB4 IP and provide software-accessible interfaces, ensuring seamless interaction between the hardware and the bare metal software.
This approach ensures drivers are efficiently adapted for bare metal operation, eliminating Linux kernel and PCIe infrastructure dependencies, providing direct, efficient hardware resource access.

```
//Kernel macro redefinition -- devm_kcalloc
#define devm_kcalloc(x,n, size, type)
calloc(n,size)

// Kernel function redefinition -- iowrite32
void iowrite32(uint32_t v,uint32_t add)
  mstr_w_data(value, addr);

//Make kernel function empty
static bool device_can_wakeup (struct
device *dev){}

//Connect S/W with H/W through DPI-C
extern "C" void mstr_w_data(int addr,
int value);

//Providing HW wrapper interrupt to USB4 driver.
extern "C" void __interrupt()
  run_nhi_interrupt(linux_driver);
```

```
module hw_wrapper (
);

// DPI-C functions determined by dependency
analysis
import "DPI-C" task __interrupt;
export "DPI-C" task mstr_w_data(addr,
value);

function void mstr_w_data(reg[31:0]
addr, reg [1:0]data);
…
endfunction

// Notify software on H/W interrupt.
always @(posedge clk) begin
  if (|intr)
    __interrupt();
end
xtor_usb4_svs usb (
  .intr (intr)
  .axi* (axi*)
  .usb4* (usb*)
);
endmodule
```

Figure 3. Code snippets depicting Kernel Proxy and H/W wrapper.

IV. Case Study Results And Conclusion

Our framework successfully converted a Linux driver to a bare metal driver for USB4. This conversion significantly reduced the development time from the typical 6-8 months to approximately 1 month. By leveraging our methodology, we were able to reuse around 39,000 lines of verified code, ensuring both efficiency and reliability.

The same systematic approach can be applied to convert any Linux device driver to a bare metal driver, making it highly versatile and adaptable to various hardware platforms. The developed bare metal driver can be used to configure and manage IP in both simulation and emulation environments, providing robust support for pre-silicon validation and system-level verification.

Furthermore, this approach extends beyond traditional hardware setups, effectively utilized in virtual prototyping environments, enabling early software development and testing. This broad applicability underscores our framework's significant impact in accelerating the bare metal driver development across different use cases and platforms.

References

[1] IEEE Standard Association, "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," IEEE Standard 1800-2017, Dec. 2017 Available: https://standards.ieee.org

[2] Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). Linux Device Drivers, Third Edition. O'Reilly Media.