Embedded Video Parameterization over Ethernet on a Zynq SoC: Bare-Metal Versus Embedded Linux

Camille Lefort
UGA – CSUG
camille.lefort@etu.univ-grenoble-alpes.fr

***Abstract*—The objective is to design an embedded system able to transmit images in real time from camera to a computer through Ethernet. This allows a fast parameterization of the optical sensor, such as exposure time or analog gain. The system is based on a Zynq-7000 SoC, combining an FPGA and an ARM processor. Two approaches are studied: bare-metal programming using lwIP, and an embedded Linux solution using PetaLinux. The work focuses on Ethernet communication, debugging steps, and image transmission using UDP packets. The results show the advantages and limits of each approach.**

I. Introduction

The QlevErSat project aims to observe Earth surfaces and human activities from space. To reduce the data rate sentto ground, images are pre-processed directly on board usingembedded artificial intelligence. Before airborne test campaigns, such as stratospheric balloon flights, the camera parameters must be adjusted. The goal of this project is to allow real-time visualization of images on a PC, in order to modify the detector registers before the flight. The system (Fig.1) starts from an existing optical sensor and focuses on the communication between the acquisition electronics and the user application. The hardware platform used is a MicroZed 7010 board, based on a Zynq-7000 SoC. The FPGA part handles video data, while the ARM processor manages communication and control. Ethernet is chosen as the main interface, because it is standard, fast, and already available on the board. This report presents the design and implementation of the system. First, a bare-metal approach using the lwIP stack is studied. Then, an embedded Linux solution is implemented to simplify software development. Debugging issues, network behavior, and image transmission are analyzed.

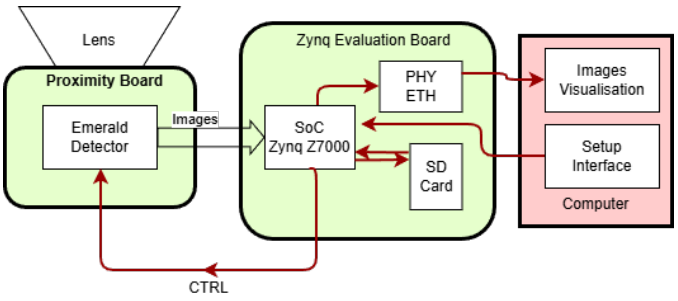

Fig. 1: System overview

II. Methods used

A. Hardware Design

The hardware design implements a video pipeline in the Zynq's programmable logic to handle image data streaming. We use a Test Pattern Generator (TPG) IP core to simulate a camera feed with synthetic video patterns (color bars, gradients, etc.) for testing. The TPG outputs a stream of pixel data (24 bits per pixel) which is passed through an AXI-Stream Subset Converter that pads it to 32 bits to match downstream components. The pixel stream then enters a FIFO (First-In-First-Out) buffer for decoupling and finally a DMA (Direct Memory Access) engine transfers the video frames into DDR memory via the Zynq's high-performance AXI interface. This offloads the heavy data movement from the CPU. The Zynq processing system IP core is configured in Vivado to enable the Gigabit Ethernet MAC (GEM0) and other required interfaces (e.g., UART for debugging, and HP port for DMA). The overall hardware pipeline allows continuous frame capture and buffering without CPU intervention, so the ARM processor can focus on processing and communication.

B. Bare-Metal Implementation (lwIP Stack)

For the bare-metal approach, we developed C firmware running on one ARM Cortex-A9 core without an operating system. We utilized lwIP, a lightweight TCP/IP stack designed for embedded systems. The firmware is compiled and deployed via Xilinx SDK/Vitis as a standalone application. Networking is initialized by configuring a static IP (e.g. 192.168.1.50/24) and gateway on the MicroZed. We assign a MAC address and add the Ethernet interface (GEM0 at base address 0xE000B000) to the lwIP stack. The lwIP initialization brings up the interface (equivalent to ifconfig up). A UART console is used for debugging prints.

In the main loop, because there is no OS, the program explicitly calls the lwIP input handling and timeout functions. On each iteration, we poll for incoming packets with `xemacif_input()` and call `sys_check_timeouts()` to allow lwIP to manage its timers (for ARP, ICMP, etc.). This super-loop architecture ensures the stack can respond to network events. Initially we verified basic connectivity: the board successfully replies to ping requests from a PC, confirming that IP and ARP are functioning.

UDP communication was implemented in bare-metal using lwIP's raw API. Without an operating system, the firmware

must explicitly handle lwIP input processing and timing functions. In particular, a dedicated time base is required to drive `sys_now()` and `sys_check_timeouts()` to ensure reliable UDP packet handling.

Hardware peripherals are controlled through memory-mapped registers via AXI-Lite, using either Xilinx drivers or direct register access in the absence of an operating system.

C. Embedded Linux Implementation (PetaLinux)

The second approach uses an embedded Linux (PetaLinux) running on the ARM processors to leverage the OS services and drivers. We built a PetaLinux 2014 distribution for the MicroZed board. The hardware bitstream (with only Processing system design) is loaded at boot time along with the Linux kernel. An SD card is prepared with the bootloader (FSBL + U-Boot), the bitstream, Linux kernel image and device tree, and a root filesystem (initramfs). After powering on and setting the board to SD boot mode, Linux boots and we can access a Linux terminal on the UART (login as root/root).

Under Linux, the Ethernet interface appears as `eth0`. We configure it with a static IP (192.168.1.50) and netmask using the standard `ifconfig` tool, then bring the link up. The Linux kernel's network stack automatically handles ARP and TCP/IP according to the protocols (the kernel implementation of UDP/IP is robust and fully RFC-compliant). We utilize the Berkeley socket API to send and receive UDP packets in user space, which greatly simplifies network programming compared to writing raw lwIP code.

To demonstrate hardware control in Linux, we used the Linux GPIO driver via sysfs to toggle an LED (on the MicroZed, LED D3 is connected to MIO47). By exporting the GPIO (number 185, since base 138 + 47) and writing to its direction and value entries under `/sys/class/gpio`, we can turn the LED on or off. This confirmed that using Linux drivers, we could easily interact with hardware without writing our own register-level code.

We developed a simple user-space application in C that runs on Linux to handle UDP communications for parameter control. First, we created a UDP server that listens on a port (5000) for incoming commands. For example, sending the text "LedOn" or "LedOff" from the PC triggers the board to light or extinguish the LED by writing to the sysfs GPIO value. This UDP command server uses blocking `recvfrom()` calls to wait for packets and `sendto()` to respond (e.g., sending an acknowledgment or error message). In testing, this worked reliably, the board could be commanded remotely to control the hardware in real-time.

We also implemented an image transfer application on Linux to send a file (representing an image) from the board to the PC using UDP. Because Ethernet frames have a maximum payload of about 1500 bytes, large files must be divided into chunks. We chose a chunk size of 1400 bytes to leave room for headers. For reliable reassembly on the PC, we designed a simple custom packet header (added to each UDP payload) containing sequence information. The header fields are listed in Table udp_header. Each UDP packet carries this header followed by a portion of the file data. A Python script on the PC listens on the UDP port, receives the packets, checks the header, and reconstructs the image by ordering chunks using the IDs.

TABLE I: Custom UDP Packet Header Format for Image Transfer

Field	size (bits)	Description
magic	*16*	Signature to identify packets
version	*8*	Protocol version number
flags	*8*	Reserved (for future use)
frame_id	*32*	ID for the image/file being sent
chunk_id	*16*	Sequence number of this chunk
total_chunks	*16*	Total number of chunks for the file
payload_len	*16*	Bytes of actual image data in this packet

The image sender program on the board simply reads the file from the SD card into memory and splits it. It calculates the total number of chunks needed:

$$\text{total\_chunks} = \left\lceil \frac{\text{file\_size}+1399}{1400} \right\rceil$$

using integer arithmetic with a ceiling adjustment. It then loops to send each chunk in order, pre-pending the header information (with identical *magic* and *frame_id* for all packets, and incrementing *chunk_id*). The standard socket send function is used to transmit UDP packets to the PC's IP address. On the PC side, our Python reassembly script uses the header to ensure all chunks are received and writes the data back to a file. This approach takes advantage of Linux's ability to handle file I/O and networking seamlessly in user space. All timing, packet ordering (UDP provides no guarantee of order), and integrity checks must be managed by our application-level protocol since UDP is connectionless and unreliable. The simplicity of adding this logic in C on Linux (versus doing it in bare-metal) is one reason we explored the OS solution.

III. Results and Analysis

A. Bare-Metal Networking Results

Using the bare-metal lwIP approach, we achieved basic network connectivity. The board successfully obtained the configured IP and responded to ICMP ping requests. The lwIP stack performed ARP resolution automatically, sending out an ARP request for the PC's address and receiving the reply (the ARP protocol maps IP addresses to MAC addresses). This confirmed that our hardware (Ethernet MAC and PHY) and lwIP were correctly integrated. We observed the ARP packets in a network sniffer, verifying the sequence: the MicroZed announced its IP and asked "Who has 192.168.1.100?" and the PC responded with its MAC. Such low-level behavior demonstrated that even without an OS, the stack could handle background network tasks as long as we called the timer functions regularly.

However, we encountered significant challenges during development. One blocking issue was unreliable UART output in early bare-metal tests. Debug prints were garbled or incomplete, making it hard to troubleshoot other parts. After extensive debugging, we discovered a hardware configuration mistake: the UART TX/RX pins were configured for the

wrong voltage standard. The MicroZed's serial port pins use LVCMOS 1.8V, but our FPGA I/O bank was set to 3.3V. This electrical mismatch meant the UART signals were not recognized properly, hence the strange behavior (characters dropped or corrupted). We resolved this by correcting the voltage setting in Vivado and also disabling an unnecessary pull-up on the UART RX line. Once fixed, the UART communication became stable, and we could reliably use `xil_printf` to output messages for debugging.

Another issue was initializing the Ethernet MAC in lwIP. At first, our lwIP `xemac_add()` call failed with an error indicating it "unable to determine type of EMAC at address 0xE000B000". This turned out to be a problem with our Board Support Package (BSP) or hardware description. We realized that the BSP settings must match the hardware design exported from Vivado. Regenerating the hardware platform and ensuring the GEM0 interface was correctly defined resolved the EMAC initialization error. After fixing this, lwIP was able to set up the network interface without errors.

Once the network was up, we attempted to send UDP packets in bare-metal. We found that while the ARP handshake took place (as mentioned), the UDP packet we tried to send did not show up on Wireshark. This was traced to the timing loop: we had not properly handled the lwIP timeout mechanism. Our initial code did not call `sys_check_timeouts()` frequently, and we left the `sys_now()` function as a stub. As a result, certain lwIP processes (possibly ARP refresh or UDP packet dispatch scheduling) were not running. We learned that in standalone mode, the developer must implement a timebase for lwIP. After adding a simple millisecond counter and ensuring `sys_check_timeouts()` was invoked, the UDP transmission succeeded. We managed to get the UDP echo server working in bare-metal: when the PC sent a UDP packet to the board, the same payload was returned to the PC. This confirmed that the bare-metal implementation could handle bi-directional UDP communication, although the effort to reach this point was considerable due to the need to manually address hardware and timing issues.

B. Embedded Linux Results

Moving to the embedded Linux approach, many low-level concerns were mitigated by the operating system. The Linux kernel took care of initializing the Ethernet interface (once we configured the IP with `ifconfig`). ARP and ICMP were handled behind the scenes by the kernel's network stack. For instance, the board responded to pings immediately once configured, without any special user code. This immediate network functionality showed the advantage of using a mature OS stack.

We implemented the UDP LED control server on Linux and found it to be straightforward. The server continuously listens on port 5000. When we sent a text command from a PC (using a simple UDP client script), the board parsed the command and toggled the LED via sysfs. The response (either an acknowledgment or an error message if the command was unknown) was sent back to the PC. In testing, the round-trip for these command packets was very quick (a few milliseconds over a local network). There were no packet losses or order issues at the scale of this test, which is expected in a LAN environment for such small traffic. The Linux UDP socket API proved reliable and easier to debug (we could print to the Linux console or use standard tools like `tcpdump` on the board).

For the image transfer, the Linux implementation successfully sent an entire image file in UDP fragments. For example, using a test image of moderate size, the program split it into dozens of 1400-byte chunks. All packets were received by the Python script on the PC, which reassembled them by sorting on the *chunk_id* field. Because UDP does not guarantee order or delivery, in a real scenario some chunks might arrive out-of-order or be missing. In our tests on a local network, all packets arrived and in order. Nonetheless, our custom header (Table headertab:udp_headerld allow detection of missing chunks (the PC can see if a sequence number is skipped and request a resend, though our simple implementation did not include a resend mechanism). The successful image transfer demonstrates that even fairly large data (larger than a single Ethernet frame) can be transmitted over UDP in the Linux environment with relative ease. We simply leveraged file I/O to read the image from disk and the OS network stack to send packets; the heavy lifting of TCP/IP, error checking, and so on is handled by the kernel. The main bug we encountered during development of the sender was an incorrect calculation of total chunks due to integer division (as discussed in the Methods section, e.g., 2500/1400 = 1 instead of 2, which we solved by rounding up). Once fixed, the file transmission worked as expected.

C. Comparison of Bare-Metal vs Linux Approaches

Table II summarizes key differences between the two implementations. The bare-metal approach required more effort in terms of low-level debugging. We had to manually ensure the hardware was configured correctly (voltage standards, core placement, etc.) and that the software handled all aspects of the network stack (from ARP timers to packet buffering). The absence of an operating system meant we had to write or configure everything by ourselves, including device initialization and drivers. This resulted in a deeper understanding of the system (for instance, we learned exactly how UART and Ethernet were set up on the Zynq and how lwIP works internally). On the downside, development was slower and prone to subtle bugs (one line in BSP or one hardware setting could break the whole communication).

In contrast, the embedded Linux approach abstracted away many of these details. The OS provided ready-to-use drivers for UART, Ethernet, and filesystem, so we did not need to worry about register addresses or low-level protocol details for ARP/ICMP. This dramatically sped up development of features like the image transfer. We could focus on application logic (how to split and label the data) rather than reinventing reliable data transfer. Linux also allowed using high-level languages or scripts on the board for quick testing (e.g., one could even use Python on the board for networking, though we used C for

performance). The trade-off is that Linux introduces overhead: the system needs to boot (tens of seconds for PetaLinux boot vs near-instant startup for bare-metal), and it consumes more memory (the Linux kernel and processes occupy a significant portion of DDR). For a resource-constrained satellite, this could be a limitation. Also, nondeterministic latency might be introduced by Linux's scheduling if precise real-time operation is required (the standard Linux kernel is not real-time). In our testing, the Linux network performance was excellent for our needs and we did not encounter noticeable latency issues for the UDP traffic, but this might matter for high-frame-rate video streaming or time-critical control signals.

Another difference is maintainability and scalability. Developing on bare-metal with lwIP is adequate for a small application, but adding complex features (for example, implementing a filesystem or handling multi-threaded tasks) would be difficult. On Linux, those capabilities are readily available. For instance, writing the image to SD card and reading it was trivial under Linux, whereas in bare-metal we would need a FAT filesystem library or a custom data source. Linux also makes it easier to integrate with existing tools (we could run a web server or use SSH for remote control if needed, leveraging open-source software).

Security and reliability considerations can also guide the choice. A bare-metal firmware has a smaller attack surface (no OS services running) and can be more predictable, which might be important in a satellite context. Linux, while more complex, benefits from well-tested networking code and can handle errors (memory management, process isolation).

TABLE II: Comparison of Bare-Metal vs. Embedded Linux Implementations

Aspect	**Bare-Metal (lwIP)**	**Embedded Linux (PetaLinux)**
OS / Environment	None (Standalone C program)	Linux OS (kernel 3.17, userland)
Network Stack	lwIP library (manual integration)	Linux TCP/IP stack (kernel built-in)
Development Effort	High: low-level driver code, custom timing needed	Lower: ready drivers, use of sockets and high-level APIs
Boot Time	Very fast	Longer (tens of seconds, OS boot sequence)
Memory Footprint	Small (app + lwIP)	Larger (kernel, OS services in DDR)
Hardware Control	Direct register access (Xilinx SDK drivers)	Abstracted (sysfs, standard Linux drivers)
Real-Time Behavior	More deterministic (no scheduler preemption)	Non-deterministic scheduling (not real-time by default)
Networking Features	Basic (limited by what is implemented via lwIP)	Full Linux networking (TCP/UDP, sockets, security features)

Overall, both approaches succeeded in establishing the core functionality: sending and receiving data over Ethernet, and enabling remote control of hardware parameters. The bare-metal implementation is lightweight and gave us fine control over the hardware, but required significant debugging (e.g., figuring out electrical issues and writing infrastructure like timing routines). The embedded Linux implementation, on the other hand, allowed rapid development and rich functionality using existing OS infrastructure, at the cost of higher resource usage. In our project, the Linux approach ultimately allowed us to demonstrate the full video parameterization use-case (including file transfer) more quickly.

IV. Conclusion

We have developed a system on the Zynq SoC for real-time video transmission and parameter control, exploring two distinct approaches. A bare-metal firmware with lwIP, and an embedded Linux platform. Both implementations were able to utilize the custom hardware pipeline (FIFO, DMA) to send image data over Ethernet and support basic UDP communication. The bare-metal approach proved that a lightweight stack can run on the Zynq with minimal overhead, but it demanded careful handling of low-level details (timers, hardware setup). The embedded Linux approach greatly simplified application development by providing standard drivers and networking utilities, enabling features like remote LED control and image file transfer with less effort.

Each approach has advantages: the bare-metal firmware might be preferred for its simplicity and real-time predictability in a flight model with strict resource limits. The Linux solution is advantageous for prototyping and feature-rich applications due to its ease of use and robustness. By comparing them, this project highlights the classic trade-off in embedded system design between low-level control and high-level convenience. Future work will involve integrating the actual image sensor via SPI and assessing whether the bare-metal or Linux environment. Better serves the final satellite deployment, potentially considering a hybrid approach (Linux with real-time patches or a separate real-time microcontroller for critical tasks).

References

[1] A. Dunkels, *Design and Implementation of the lwIP TCP/IP Stack*. Swedish Institute of Computer Science, Tech. Report, 2001.

[2] Xilinx Inc., *Zynq-7000 SoC Technical Reference Manual*, UG585 (v1.11.1), 2018.

[3] Xilinx Inc., *PetaLinux 2014.2 Board Bring-up Guide*, UG980 (v2014.2), 2014.

[4] D.C. Plummer, "An Ethernet Address Resolution Protocol," IETF RFC826, Nov. 1982.