Journal of
Instrumentation

OPEN ACCESS

Building complex SoC images made simple with SoCks

To cite this article: L.E. Ardila-Perez *et al* 2026 *JINST* **21** C03011

View the article online for updates and enhancements.

You may also like

- Using containers with ATLAS offline software
J Elmsheuser, L Heinrich, G Stewart et al.
- Making containers lazy with Docker and CernVM-FS
N Hardi, J Blomer, G Ganis et al.
- Micro-CernVM: slashing the cost of building and deploying virtual machines
J Blomer, D Berzano, P Buncic et al.

This content was downloaded from IP address 141.52.96.24 on 22/10/2026 at 15:01

Published by IOP Publishing for Sissa Medialab

Received: *November 6, 2025*
Accepted: *January 19, 2026*
Published: *March 6, 2026*

Topical Workshop on Electronics for Particle Physics
Rethymno, Crete, Greece
6–10 October 2025

Building complex SoC images made simple with SoCks

L.E. Ardila-Perez[a,*] **M. Fuchs**[a] **T. Muscheid**[a] **O. Sander**[a] **and L. Scheller**[a]

[a] *Institute for Data Processing and Electronics, Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1, Eggenstein-Leopoldshafen, Germany*

E-mail: luis.ardila@kit.edu, marvin.fuchs@kit.edu

Abstract: Modern heterogeneous System-on-Chip (SoC) devices combine CPUs, GPUs, FPGAs, and AI accelerators into compact and energy-efficient platforms. However, the construction of bootable images for these systems remains complex because of fragmented tools, opaque dependencies, and inflexible workflows. SoCks (System-on-Chip blocks) introduces a lightweight and modular build framework that partitions an SoC image into encapsulated blocks, each built independently with standardized interfaces. This modularization simplifies development, encourages component reuse, enables distributed collaboration, and integrates seamlessly with modern Continuous Integration and Continuous Delivery (CI/CD) pipelines. Within back-end development for the Phase-2 upgrade of the Compact Muon Solenoid (CMS) experiment, SoCks has been used to build the bootable image for the Kria K26 System-on-Module (SoM) on the Serenity-S data processing board. The image utilizes an AlmaLinux-based operating system that employs a layered root file system, combining a read-only network-mounted lower layer with a writable local upper layer, to ensure consistency across boards. Furthermore, the Serenity-S image is optimized for a fully network-based boot process, allowing synchronized deployment across multiple Serenity-S boards and remote updates.

Keywords: Detector control systems (detector and experiment monitoring and slow-control systems, architecture, hardware, algorithms, databases); Data acquisition concepts; Large detector-systems performance; Trigger concepts and systems (hardware and software)

*Corresponding author.

© 2026 The Author(s). Published by IOP Publishing Ltd on behalf of Sissa Medialab. Original content from this work may be used under the terms of the Creative Commons Attribution 4.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.

https://doi.org/10.1088/1748-0221/21/03/C03011

Contents

1 **Introduction** 1

2 **Related work** 2

3 **SoCks concept and implementation** 2

4 **Building the Zynq UltraScale+ image for the Serenity-S board with SoCks** 3

5 **Conclusion** 5

2026 JINST 21 C03011

1 Introduction

Modern System-on-Chip (SoC) devices tightly integrate advanced components such as multi-core processors, AI accelerators, Field-Programmable Gate Array (FPGA) reconfigurable fabric, and high-speed interfaces into a single package, thereby addressing computational challenges across a broad range of domains. For the Phase-2 upgrade of the Large Hadron Collider (LHC), thousands of such devices will be deployed in the ATLAS and CMS experiments as part of the off-detector electronics back-end readout systems [1]. However, the growing complexity of these heterogeneous devices also increases the difficulty of developing, integrating, and maintaining their firmware and software environments. Existing development tools, while powerful, have become increasingly intricate, often becoming a barrier to effectively managing these complex devices.

To address these challenges, we introduce *System-on-Chip blocks* (SoCks) — a lightweight, modular, and scalable build framework for creating complete SoC images. SoCks introduces an abstraction layer that divides an SoC image into a small set of independent components, enabling reproducible builds, simplified maintanance, and seamless integration with modern Continuous Integration and Continuous Delivery (CI/CD) pipelines.

A prominent example of SoC-based deployment in high-energy physics is the Serenity-S data processing card, shown in figure 1 (Left), developed for the CMS Phase-2 Upgrade and used in several CMS sub-detector systems [2]. The Serenity-S is a versatile ATCA-based platform featuring

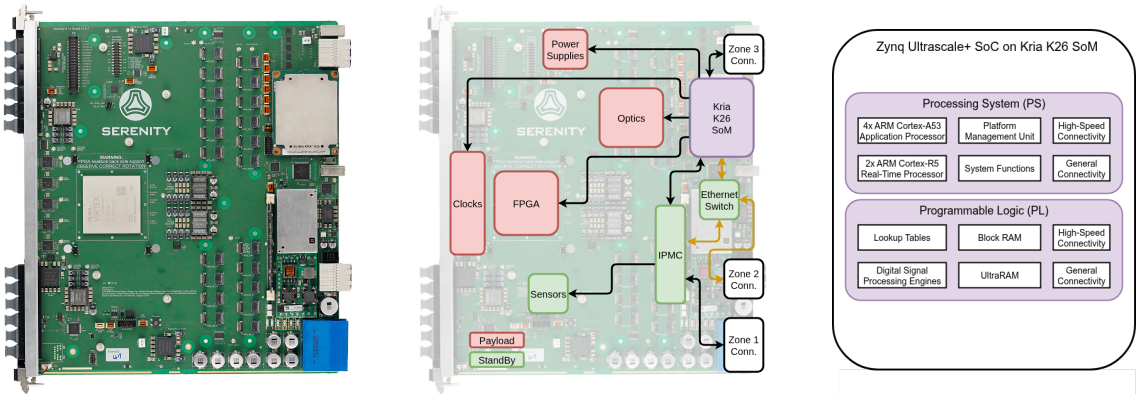


Figure 1. Left: Serenity-S v1.4 ATCA board. Middle: Simplified schematic showing the Kria K26 SoM connectivity within the Serenity-S board. Right: Simplified diagram showing the internal components of the AMD Zynq Ultrascale+ device used in the Kria K26 SoM.

an AMD Virtex UltraScale+ VU13P FPGA for high-throughput data processing and an AMD Zynq UltraScale+ SoC for slow control and board management, see figure 1 (Middle). The Zynq SoC combines FPGA flexibility with multi-core ARM processors and various interfaces, allowing direct integration with the board infrastructure, see figure 1 (Right). This contribution will highlight how SoCks has been successfully used to generate network-bootable Zynq UltraScale+ images for Serenity-S in a reproducible way while simplifying firmware and software integration across the distributed nature of the various CMS subsystems.

2 Related work

Existing SoC build frameworks, particularly Yocto and Buildroot, offer powerful customization options but are difficult to manage [3, 4]. They rely on deeply nested configurations and a large number of interdependent recipes that make the build process opaque and error-prone. Their primary design goal is to support minimal embedded systems with custom root file systems and no package management, which is ill-suited for high-performance SoCs that can easily host full Linux distributions.

Vendor-specific wrappers around standard tools, such as the Yocto-based PetaLinux from AMD (which is discontinued) provide limited automation and simplification but lack portability [5, 6]. Consequently, there is a need for a framework that combines transparency, modularity, and compatibility with modern software development methodologies such as containerization and CI/CD.

3 SoCks concept and implementation

SoCks reduces complexity in building heterogeneous SoC images by partitioning them into independent blocks that can be developed, built, and tested separately. Each block represents a logical part of the image — such as the Linux kernel, device tree, FPGA firmware, bootloader, or root file system. The number of blocks is chosen to minimize dependencies while maintaining flexibility. Communication between blocks is handled through well-defined interfaces that allow information to flow in a controlled and predictable manner.

Every block is processed by a dedicated *builder*, which defines how the block is created. Builders encapsulate the logic required to build software or firmware components and ensure that dependencies are satisfied. Switching between different implementations of a block, such as Debian or AlmaLinux root file systems, becomes straightforward thanks to these standardized interfaces. SoCks stores all project configuration in a single human-readable YAML file that includes both global and block-specific parameters. The inclusion mechanism allows default configuration files to be imported and customized for specific projects, ensuring consistency across development teams. Each block can be built inside a container, providing a controlled environment with all required tools and dependencies. This approach ensures reproducibility, portability, and isolation from the host system. The same container infrastructure integrates naturally with CI/CD pipelines, where each block can be automatically built and tested.

SoCks is implemented in Python using the facade design pattern, which enables a unified command-line interface over a set of specialized subsystems [7]. Each builder is represented as a Python class that communicates through standardized interfaces. Builders interact with external tools such as compilers and FPGA synthesis suites through POSIX-compatible shells, either directly on the host or within containers. The command-line interface of SoCks enables straightforward operation

2026 JINST 21 C03011

— a single instruction, such as `socks kernel build`, is sufficient to build an individual block or a complete image. Each block supports a defined set of commands for preparation, building, cleaning, and debugging. The execution order follows dependency logic, and incremental building is supported through timestamps, checksums, and configuration comparison. Once built, blocks are packaged as compressed archives that can be reused across projects or pipelines.

Designed for collaborative environments, SoCks integrates naturally with Git-based CI/CD workflows that automatically build complete SoC images and publish block packages on accessible servers. Developers can download pre-built blocks and rebuild only the components relevant to their work, significantly reducing local computational requirements. For instance, firmware developers can focus exclusively on the FPGA block, while software engineers modify the root file system, both reusing shared block packages for the rest of the image. This modular approach, combined with the ability to use full Linux distributions such as Debian or AlmaLinux, simplifies system maintenance, ensures compatibility with established package management and security update mechanisms, and enables reproducible development workflows. The build time for individual blocks is comparable to creating these components with Yocto and, in many cases, considerably shorter — up to 28 times faster when rebuilding the device tree [8, 9]. The SoCks project is open-source software and available on GitHub [10].

4 Building the Zynq UltraScale+ image for the Serenity-S board with SoCks

Integrating the Serenity-S into the network infrastructure of the CMS experiment poses a number of requirements for the central operating system used on the SoM. A fundamental requirement is the usage of a standard Linux distribution, more specifically the Red Hat Enterprise Linux (RHEL) clone AlmaLinux [11]. Using this distribution is also a requirement for general-purpose computers in this network, which enables the use of the same mechanisms and tools to administer both classes of devices, providing a familiar environment for system administrators and to users that log in to and use a Serenity-S board manually.

The Kria K26 image for the Serenity-S board uses the AlmaLinux root file system together with a custom Linux kernel based on AMD's official fork for Zynq UltraScale+ SoC devices. This kernel includes additional configuration and patches to enable features such as containerization. This custom kernel is necessary because Red Hat removed support for these devices in the RHEL 9.2 kernel release [12]. Eventually, such changes in the support policy will also be reflected in clones that aim to be a plug in replacement for RHEL, such as the AlmaLinux distribution. As is common practice for standard Linux distributions, the connecting link between the AlmaLinux root file system and the Linux kernel is an initramfs (initial RAM file system). While embedded systems frequently omit an initramfs to save storage space and simplify the system, having one enables the implementation of custom routines for preparing the root file system mounts, which is especially relevant in a network boot scenario. A busybox initramfs is frequently used in embedded systems, because it is very small, simple, and easy to manage [13]. The bootloaders and infrastructure software used in the Kria K26 image are customized versions of the respective templates provided by AMD. Figure 2 provides an overview of the complete Kria K26 image for the Serenity-S.

While this Kria K26 image can operate entirely on its own, in the CMS context it is used with a second, supporting image. This second image is used to implement the Split Boot v2 process, which enables fully network-based booting for AMD UltraScale+ SoCs by extending PXE with a

2026 JINST 21 C03011

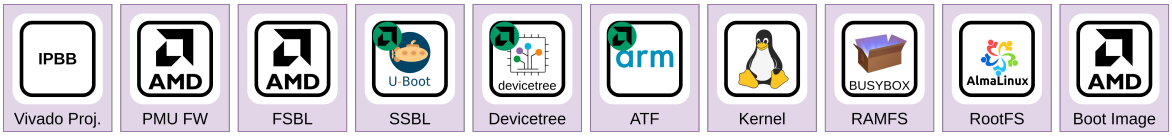


Figure 2. SoCks Components of the Main Serenity-S Image.

method to synchronize all low-level software, including boot loaders, with a network server [14]. This simplifies administration and automates the distribution of updates to a large number of SoCs in a distributed and physically difficult-to-access system such as the CMS back-end electronics. The Split Boot v2 mechanism is entirely implemented in the second-stage boot loader, U-Boot, which is the first component in the boot process of these SoCs that allows convenient network access. Although not strictly necessary for functionality, the image still contains a full Linux operating system, which serves as a fallback OS and debugging environment in case the Split Boot mechanism fails. Using an AlmaLinux distribution as with the main image is not possible, because the Split Boot v2 image needs to fit into the 64 MB QSPI memory available on the Kria K26 SoM.

Alternatively, there are Linux distributions like Alpine Linux that are optimized to be small and resource efficient, while at the same time providing the convenience of a tested environment, public repositories with binary packages, and regular updates. In combination with a custom Linux kernel based on the official fork from AMD but trimmed to the minimum, it is possible to create an image that fits well within the limits of the QSPI memory on the Kria K26 SoM. Therefore, this is the approach we propose for Serenity-S. The Split Boot image for Serenity-S is depicted in figure 3. As can be seen by comparing this to figure 2, the Split Boot image does not have a nonvolatile root file system. Instead, it relies on a RAM file system, which eliminates the need for a file system on the QSPI memory, reducing both storage requirements and write cycles.

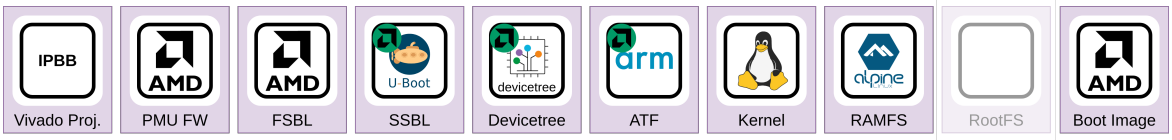


Figure 3. SoCks Components of the Split Boot v2 Serenity-S Image.

Together, the main image and the split boot image for the Kria K26 on Serenity-S enable a fully network-based boot process for the AMD Zynq UltraScale+ SoC. However, to use this feature, a network server must provide all components of the main image. The complete setup is visualized in figure 4. Except for the user home directories, all software and firmware components stored on the server are golden files that are used on all Serenity-S boards of a given CMS subsystem. This ensures that all boards operate with the same image and stay synchronized. In detail, the Split Boot v2 mechanism is used to synchronize the low-level software and firmware components, whereas the lower layer of the root file system is mounted via NFS, which eliminates the need for a local copy on the Serenity-S. The initramfs — which together with the Linux kernel is part of the low-level software and firmware components — is used to mount the lower layer of the root file system in read-only mode and combine it with the read- and write-enabled upper root file system layer on the local SSD to create the complete root file system. While the lower layer on the NFS server contains the complete AlmaLinux

2026 JINST 21 C03011

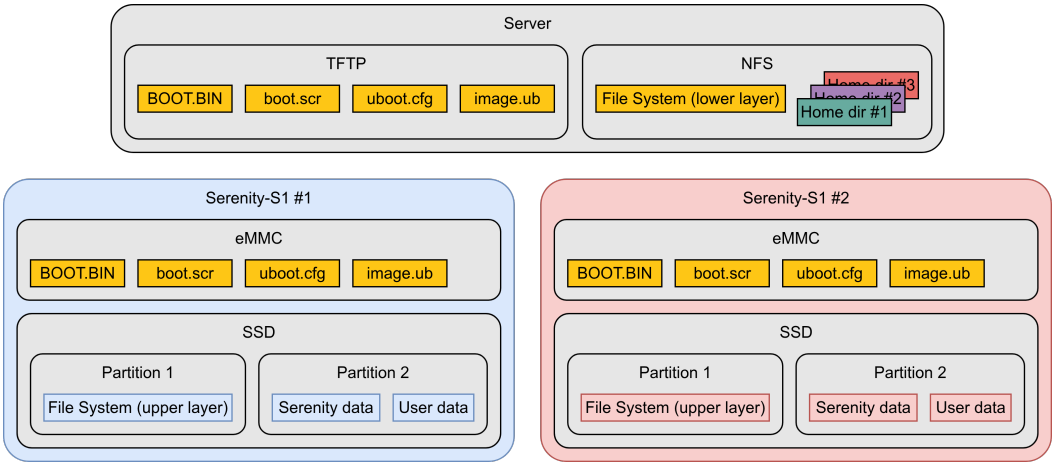


Figure 4. Serenity-S Kria storage infrastructure. User home directories are not tied to individual boards but are instead linked to the user's CERN account, allowing access across multiple boards.

installation, the upper layer on the SSD is used to store files created or modified at runtime. Once the Kria K26 SoM boots into the AlmaLinux root file system, it automatically mounts the second partition on the SSD, which can be used by tools and users to store temporary data. The appropriate user directory is finally mounted as soon as a user logs in.

Since the local SSD is only used to store temporary and user-specific data, updating the Kria K26 image for all Serenity-S boards of a given CMS subsystem is performed in two steps, which are carried out remotely. First, the golden files on the server are updated. Subsequently, the Kria SoMs are restarted and they fetch automatically the golden files in the server as part of their boot process, replacing their local copies. This means that all Kria K26 SoMs in the system immediately run with the latest version of the image. Besides a reliable updating mechanism, a robust build process for Kria K26 images is also essential to guarantee system reliability and enable rapid updates when needed. To achieve this, both the main and Split Boot images are generated within a GitLab CI/CD pipeline, which automatically triggers builds whenever changes are pushed to their respective repositories.

5 Conclusion

SoCks introduces a practical and lightweight approach to managing the complexity of heterogeneous SoC systems. By dividing firmware and software into modular blocks with clearly defined interfaces, it achieves a high degree of flexibility and maintainability. The Python-based implementation, combined with containerized builders and a transparent configuration structure, provides a reproducible and CI/CD-compatible environment.

Compared with established frameworks like Yocto and Buildroot, SoCks offers faster build times, lower resource consumption, and easier integration into collaborative workflows. It allows developers to use standard Linux distributions as part of their SoC images, thereby bridging the gap between embedded and general-purpose computing environments. Within the CMS Serenity-S platform, SoCks has proven particularly effective for constructing complex, multi-layered images for the Zynq UltraScale+ SoC, integrating custom kernels, bootloaders, and layered file systems. This capability enables reproducible, maintainable, and centrally managed device images across a large number of distributed SoC-based boards.

2026 JINST 21 C03011

Acknowledgments

The authors acknowledge the support by the Doctoral School "Karlsruhe School of Elementary and Astroparticle Physics: Science and Technology".

References

[1] R. Spiwoks et al., *Summary of the system-on-module survey*, CERN-OPEN-2023-001 (2023).

[2] T. Mehner et al., *Lessons learned while developing the Serenity-S1 ATCA card*, 2024 *JINST* **19** C02018 [arXiv:2311.02222].

[3] *Yocto*, https://www.yoctoproject.org (2025).

[4] *Buildroot*, https://buildroot.org (2025).

[5] T. McDowell, *PetaLinux Relationship to Yocto Project*, https://xilinx-wiki.atlassian.net (2024).

[6] *AMD Embedded Development Framework*, https://www.amd.com/ (2025).

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994).

[8] M. Fuchs et al., *SoCks — Simplifying Firmware and Software Integration for Heterogeneous SoCs*, arXiv:2510.15910.

[9] M. Fuchs et al., *SoCks — A Build Framework for Heterogeneous High Performance System-on-Chips*, in the proceedings of the *4th CERN System-on-Chip Workshop*, CERN, 28–31 October 2025 [https://indico.cern.ch/event/1518977/contributions/6644638].

[10] *Socks*, https://github.com/kit-ipe/SoCks (2025).

[11] *Almalinux*, https://www.almalinux.org (2025).

[12] D. Scannicchio and M. Dobson, *System And Network Administration for SoCs*, in the proceedings of the *3rd CERN System-on-Chip Workshop*, CERN, 2–6 October 2023 [https://indico.cern.ch/event/1253805/contributions/5556269].

[13] *Busybox*, https://www.busybox.net (2025).

[14] M. Fuchs, *Split Boot v2 — Simple and Reliable Network-Based Booting for Serenity-S1 and other Boards with ZynqMPs Devices*, in the proceedings of the *3rd CERN System-on-Chip Workshop*, CERN, 2–6 October 2023 [https://indico.cern.ch/event/1253805/#29-split-boot-v2-simple-and-re].

2026 JINST 21 C03011