Development of an embedded operating system based on the Linux kernel for SoC FPGA

Yurii Herman[1], Oleh Krulikovskyi[1,2], Serhii Haliuk[1] and Sergey Subbotin[3]

[1]Yuriy Fedkovych Chernivtsi National University, Department of radioengineering and information security, St. Storozhynetska, 101, Chernivtsi, 58012, Ukraine
[2]Ştefan cel Mare University of Suceava, Str. Universităţii 13, Suceava, 720229, România
[3]National University "Zaporizhzhia Polytechnic", St. Zhukovsky, 64, building 3, Zaporizhzhia, 69063, Ukraine

Abstract

In this paper, we create a distribution of the embedded operating system for Cyclone V SoC FPGA platforms based on the Linux kernel. A comparison of well-known open-source tools for creating embedded operating systems is demonstrated. A step-by-step example of embedded OS synthesis using custom scripts and a simplified pipeline is given, which increases the adaptability of the target system. The possibility of adding hardware-oriented tools for interaction between SoC and FPGA is demonstrated. This makes it possible to create a wide range of hardware applications with remote access. The proposed approach is also vendor-independent and can be applied to other FPGA SoCs.

The final system, although not significantly different in resource requirements from Yocto, is more adaptable and can be ported to the Yocto base if necessary. This allows us to make the most of the Full Custom approach, ensuring an optimal balance between development efficiency, responsiveness to changes, and system resource requirements.

Keywords

FPGA SoC, embedded operating systems, Linux kernel, system flexibility, extensibility, drivers, user space, dynamic development

1. Introduction

The compliance of modern computing platforms with Moore's Law requires developers to use new architectural approaches during development. One of these is a variety of systems on a chip (SoC). Among the wide variety of SoCs for real-time systems, SoC FPGAs (Field-Programmable Gate Array) have become widely used [1]. SoC FPGAs are integrated circuits that combine the functionality of a microprocessor core (CPU) with the capabilities of a programmable logic gate array (FPGA) on a single chip. This is an important class of microelectronics characterized by high programming flexibility, high-performance computing, and efficient use of resources [2-4]. In real-time embedded systems, FPGAs play a key role due to their versatility and ability to solve tasks in signal processing, controllers, image processing, and more [5]. Even though FPGAs provide outstanding characteristics (Flexible programming, Integration, Performance, Scalability), using them in their pure form requires high costs and qualifications in the development of modern applications.

Accordingly, to reduce the complexity of development, all real-time embedded applications must interact with the embedded operating system [6]. This is because it is difficult to manually create an optimized system without an OS. After all, when tested, such a system will not work better than similar solutions that use an OS [7]. Also, this approach allows you to better focus on system-level optimization when developing applications based on SoC FPGA. Because the operating system will be maximally optimized for the architecture of the microprocessor. The hardware is often fully optimized for the FPGA architecture. At the same time, communication between the software and hardware parts will be carried out through optimized bridges [8].

CMIS-2024: Seventh International Workshop on Computer Modeling and Intelligent Systems, May 3, 2024, Zaporizhzhia, Ukraine

yurii.herman@chnu.edu.ua (Yu. Herman); o.krulikovskyi@chnu.edu.ua (O. Krulikovskyi); s.haliuk@chnu.edu.ua (S. Haliuk); subbotin@zntu.edu.ua (S. Subbotin)

0009-0003-2473-7365 (Yu. Herman); 0000-0001-5995-6857 (O. Krulikovskyi); 0000-0003-3836-2675 (S. Haliuk); 0000-0001-5814-8268 (S. Subbotin)

An embedded OS is a specialized operating system for computer systems embedded in other systems. These OS are computer systems designed for a specific purpose, to increase functionality and reliability for a specific task [9].

An important difference between most embedded operating systems and desktop operating systems is that the application, including the operating system, is usually statically compiled into a single package. Speaking of the Linux kernel, it is worth noting that embedded OSes based on it have features that positively distinguish them from specialized OSes. Resource efficiency is achieved at the cost of losing some functionality or granularity that larger computer operating systems provide, including features that may not be used by the specialized systems on which they run. Depending on the multitasking method, this type of OS is often thought of as a real-time operating system, or RTOS. Embedded systems most often use real-time operating systems. QNX, FreeRTOS, and VxWorks are the most widely used embedded operating systems today [10-12].

Most embedded operating systems are proprietary and closed to modification. The field of embedded development is very complex and highly demanding for engineers. Open Source Software (OSS) allows the creation of task and platform-oriented OSes without the need to implement low-level layers from scratch. In the field of SoC FPGA and embedded OSes, two systems for assembly and construction are attracting the most attention Yocto Project + Poky [13] and Buildroot [14]. These tools are representatives of the FPGA supported and used by large companies (Intel, AMD). Using them allows the use of the entire FPGA library and the experience of professionals from open-source communities [15].

Accordingly, the work aims to compare the existing approaches to building an embedded OS using open-source tools using the example of the DE10-Nano board [16] with SoC FPGA Cyclone V. Since the process of synthesizing an embedded OS, except for some points, is vendor-independent and will be similar for other SoC FPGAs.

The paper is organized as follows. A comparison of different OS development approaches and their impact on the efficiency of the FPGA SoC system is presented in section 2. The step-by-step process of building an embedded OS with hardware-specific extensions for FPGA SoC is presented in section 3. The development of a user space system and conclusions are presented in sections 4 and 5 respectively.

2. Comparison of different OS development approaches and their impact on the efficiency of the FPGA SoC system

As already mentioned above, the two most widely used tools for developing an embedded OS distribution are Yocto Project + Poky and Buildroot. A third alternative is development using custom scripting.

2.1. Yocto Project

A Linux distribution is a set of software packages and the ways they interact. There are hundreds of Linux distributions available. Most of them are not designed for embedded systems, lack the flexibility needed to achieve target sizes and change functionality, and are not suitable for systems with limited resources.

The Yocto project is a Linux distribution factory that uses several other open-source projects [15]. Yocto Project, on the other hand, is a distribution creation tool in its own right. It allows you to create a Linux distribution designed for a specific system.

2.2. BuildRoot

Unlike Yocto, BuildRoot uses more general technologies, which negatively affects its flexibility. BuildRoot is a set of make files and patch aggregation that simplifies and automates the process of creating a complete and bootable Linux environment for an embedded system while using cross-compilation to create multiple target platforms on a single Linux-based host machine. Buildroot can automatically assemble the necessary cross-compilation tools, create the root file system, compile the Linux kernel image, and generate the bootloader for the target system, or

perform any independent combination of these actions. For example, an already installed cross-compilation tool can be used independently, and Buildroot only creates the root file system[17].

Buildroot is primarily designed for use in small or embedded systems based on a variety of computer and instruction set architectures (ISAs), including x86, ARM, MIPS, and PowerPC.

2.3. Full Custom

Unlike the options described above, the Full Custom approach allows for a multifunctional system that can be updated and modified on the fly and is much more flexible in the development and prototyping process [18]. When building the OS itself, we can make changes to the software package itself, which will be standard in it, and can choose the type of Linux wrapper depending on the needs of the project (Arch Linux, Debian, etc.) [19]. This option is also capable of providing access to external resources and packages for on-the-fly installation, which greatly simplifies the process of iterating in development. Full Custom, unlike the previous approaches, is not characterized by the convenience of initial development and requires the entire image generation process. We need to assemble all parts of the OS ourselves, starting with the kernel and ending with the file system configuration.

It is a more comprehensive option that gives us the ability to control every element of the resulting OS and assemble the software and tool packages needed to get the system up and running, and if necessary, easily adapt to updating these packages from a remote source. Another significant positive factor is the ability to control what exactly will be updated in the next iteration of the distribution.

However, it should be kept in mind that that approach's adoption has higher requirements for the engineer, as the process itself needs to be fully controlled.

2.4. Comparison of Yocto, BuildRoot, and Full Custom

Overall, both frameworks can produce the same resulting distribution: a root file system image for an embedded device.

Buildroot trying to be as simple to use as possible. Its main tool is made to be as compact and user-friendly as possible, so engineers can catch up with all the tools and start developing as fast as possible. Specific features are built by extensions, allowing to create products using standard tools that are included in Buildroot distribution. The OS image created by Buildroot is usually oriented to be minimalistic, making builds quick and allowing you to create a generic system rather than one tailored to a specific use case.

In contrast, Yocto is much more robust and supports a wide range of embedded systems. To make that possible, configuration is defined in scripts (called ‘recipes’) that declare what software to build and how to build it, while allowing to gather recipes into layers, which can be used are collections of recipes written and maintained by the development community or by the distribution author himself [20].

Full custom is based on the step-by-step creation of a distribution, with full control of each element. Initially, this is not a fully automated process that relies heavily on the knowledge of an engineer. However, this process can be easily adapted to automation and allows you to build a process that is necessary specifically for a particular task.

The result of Buildroot is an image of the root file system, nothing more. Every time you need to update the system, the whole product will be rebuilt from scratch to avoid discrepancies.

The result of Yocto is a "distribution". More precisely a fully done OS with specific tools, configuration and packages. It is worth mentioning that integration with external package providers so the system can be modified while booted, requires development and is not pre-built. Yocto supports development on the board by providing SDK, to avoid continuous rebuilds of distribution for each modification.

Builds created with Buildroot and Yocto are also different in configuration. Buildroot stores configuration in a main file that can be edited using the kernel's kconfig tool (e.g. xconfig or menuconfig). That file basically contains all the configuration for built OS, including user space part.

In the case of the Full Custom approach, the result is both the final image and all its components, which can be easily modified depending on the needs of the project.

3. Building an embedded OS for FPGA SoC

3.1. Building Boot-Loader

As was mentioned above, the process of building a Linux-based distribution consists of a couple of steps (see Fig. 1).

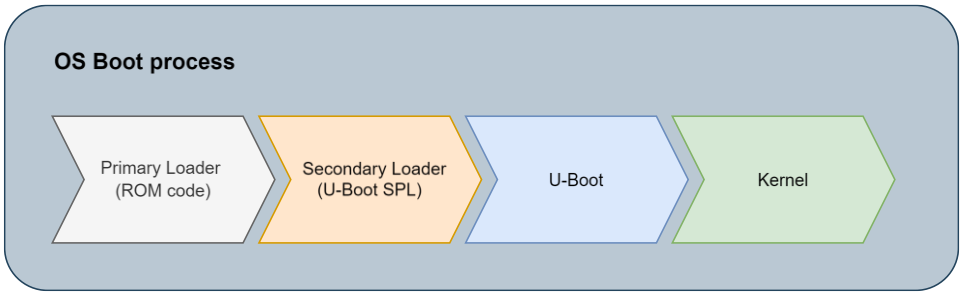

Figure 1: OS start steps diagram

The boot-loader is a crucial one because all the next parts of the OS are started by it and U-Boot handles all basic system configuration. We are using U-Boot because it is the most used and proven one, and so it usually is familiar to engineers while having an extensive knowledge base available. To start with, we can get publicly available source code and choose a specific build version, as shown in Fig. 2.

```
git clone https://github.com/u-boot/u-boot.git
cd u-boot
# We need to get list of all available u-boot versions
git tag
# And select desired one
git checkout $TAG
```

Figure 2: Getting U-Boot source code

Generally, the default configuration is enough to start with, but if we do need to customize something deeply, all configuration headers are open. For example, we can modify data and variables that are stored in the U-Boot environment or add new ones. When U-Boot starts running the kernel, it reads most of the configuration from the environment and runs scripts/code based on it.

For example, Fig. 3 demonstrates how we can anchor our OS to only a specific MAC address by modifying the header file.

```
// configs/socfpga_common.h
#define CONFIG_EXTRA_ENV_SETTINGS \
        "fdtfile=" CONFIG_DEFAULT_FDT_FILE "\0" \
        "bootm_size=0xa000000\0" \
        "kernel_addr_r="__stringify(CONFIG_SYS_LOAD_ADDR)"\0" \
        "fdt_addr_r=0x02000000\0" \
        "scriptaddr=0x02100000\0" \
        "pxefile_addr_r=0x02200000\0" \
        "ramdisk_addr_r=0x02300000\0" \
        "socfpga_legacy_reset_compat=1\0" \
        BOOTENV
#endif
// We are able to add variables into block above, for e.g
        // "ethaddr=static:mac:address\0" \
// Where \0 serves as mark for end of the string line
```

Figure 3: Environment configuration for boot

Most of the settings for U-Boot are stored in those header files, and they are used in compilation, which means that specific releases are hard-linked to those header files. We can

change some of those variables (e.g. we can set the ethernet address while U-Boot is starting). Compilation for U-Boot is straightforward (see Fig. 4) after the modification of headers is done.

```
# We are globally selecting architecture to build for
export ARCH=arm
# And setting compiller
export CROSS_COMPILE=/usr/bin/arm-linux-gnueabihf-
# We need to compile default configuration into binary first
# socfpga_de10_nano_defconfig - is preset for it
# Plenty of them are stored in configs folder
# ls -l configs/socfpga*
make socfpga_de10_nano_defconfig
# We can change default platform configuration using
menuconfig
make menuconfig
# To get resulting boot-loader we are running make
# where -j is used to run compilation for in parallel
make -j 24
```

Figure 4: Script for generation of binary file

Once compilation completes we can find out the U-Boot distribution by searching for file `u-boot-with-spl.sfp`. We will use that binary file when all the OS will be compressed in one image, as the launching point for the whole system.

3.2. Selecting the configuration and setting the Linux kernel parameters.

When building an OS, we need to go through several steps:

1. Get Preloader is a binary file provided by the manufacturer, in our case Intel/Altera. It performs some basic configuration before passing it to the bootloader.
2. Compile Bootloader - software that performs hardware initialization before passing it to the kernel for OS initialization.
3. Configure Kernel - The kernel is the heart of the OS and contains all the information about the hardware on the board.
4. Build RootFS - Create a root file system that contains the basic packages and dependencies.

When developing embedded systems, one of the key steps is to select the configuration and configure the Linux kernel options. This process includes choosing a basic RootFS configuration and selecting the kernel that best suits our needs [21].

In our case, the basic configuration of RootFS is based on Arch Linux, and the kernel chosen is altera-opensource (as shown in Fig.5). The reason for choosing the Altera-opensource kernel was the ability to program the FPGA at startup or directly from the operating system, which is supported only by this Altera kernel modification.

```
# Here we are using an Altera fork of Linux Kernel
git clone https://github.com/altera-opensource/linux-socfpga.git
# github.com/torvalds/linux can be used too
cd linux-socfpga
# We need to get a full list of versions that are available
git branch -a

# Use the branch you prefer.
git checkout $BRANCH
```

Figure 5: Kernel selection script

Therefore, we will use this particular fork in the future, after compiling it. Configuring Linux kernel parameters is an important step in the process of developing embedded systems. For this purpose, special tools, such as *menuconfig*, as it specified in Fig. 6, are used to configure various

parameters that determine the behavior and functionality of the kernel. Here are some basic steps to configure Linux kernel settings

```
# We should specify target architecture and compiller before
# compilation is started
export CROSS_COMPILE=/usr/bin/arm-linux-gnueabihf-
export ARCH=arm
# First and foremost we need to generate default configuration
make socfpga_defconfig
# That configuration can be tailored using menuconfig tool
make menuconfig
# When settings are made and finished we are going to compile
# kernel image
make LOCALVERSION=zImage -j 24
# LOCALVERSION allows us to specify how image should be named
```

Figure 6: Build process of kernel configuration tool

To do this, we need to enable the `Overlay filesystem support` and `Userspace-driven configuration filesystem` in the kernel during configuration (see Fig. 7).

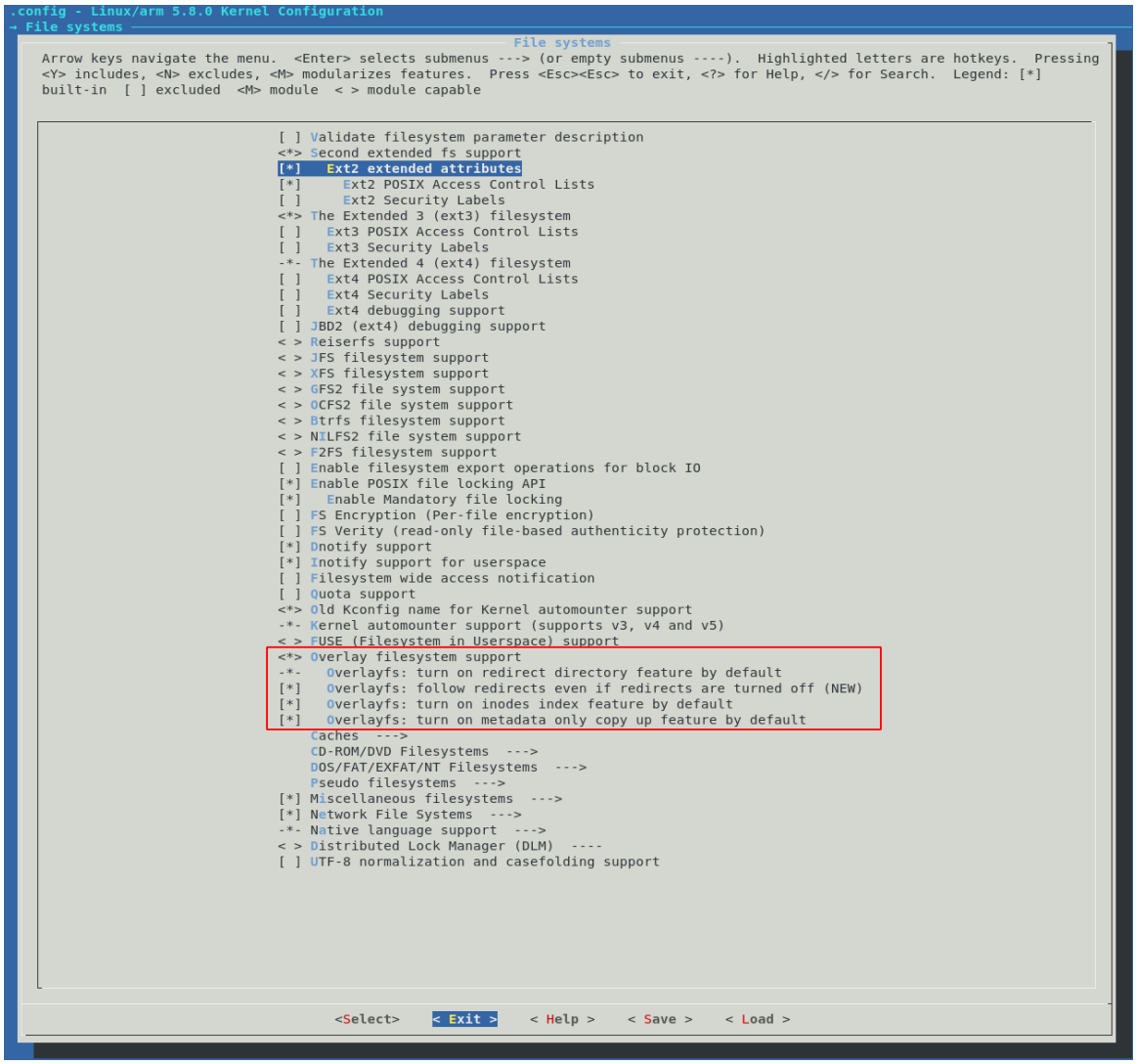

Figure 7: Configuration menu

In the future, we can change the configuration depending on our needs, but as an initial configuration, our version is sufficient.

After we`ve done with configuration we can build the image of said kernel using the steps from Fig. 8.

```
# When kernel properties are built and configured we are going
# to compile  kernel image
make LOCALVERSION=zImage -j 24
# LOCALVERSION allows us to specify how image should be named
```

Figure 8: Building the kernel image

As a result of compilation, we will get a binary zImage file, which will be used as a main Kernel executable. That binary file can be replaced separately from all other elements of system, allowing engineers to freely modify only the needed part of the distribution.

3.3. Flexibility and extensibility of the system based on the embedded operating system.

The flexibility and extensibility of a system based on an embedded operating system are important aspects, as they determine the ability to adapt and expand the system's functionality in the future. Let's take a closer look at these aspects:

- System flexibility:
    1. Configuration:

  Embedded operating systems typically have a flexible architecture that allows for customization of parameters and options to meet the requirements of a particular application.

b. Modularity:

  The use of a modular architecture makes it easy to add and remove functional modules, which makes the system more flexible in changing requirements and project development.

- System extensibility:
    1. Support for new devices:

  Embedded operating systems have mechanisms to dynamically add and support new devices without the need to recompile or flash the kernel.

b. Extensibility of functionality:

  The system can easily add new features and services without significant changes to the source code or architecture.

- Means for expansion:
    1. Dynamic libraries and modules:

  Using dynamic libraries and modules allows you to add new features without rebooting your system or recompiling your applications.

b. Network services:

  Implementation of network services that allow dynamic application loading and system configuration over the network.

3.4. The impact of the embedded OS on the flexibility and extensibility of FPGA SoC designs.

The use of an embedded operating system (OS) such as Linux in FPGA system-on-chips (SoC FPGA) significantly increases the flexibility and extensibility of projects. Due to the configurability of Linux, developers can customize the system to meet specific project requirements, including the choice of supported devices and functionality. The modular architecture of Linux allows you to dynamically add and remove features through the kernel or load modules while the system is running, which allows you to quickly expand functionality without a complete system reboot. In addition, Linux supports FPGA software upgrades without the need to shut down the system, which makes it easy to adapt to changes and implement new features and capabilities without much effort.

In the case of Full Custom Linux, we have access to many external modules, which means we can quickly reconfigure the entire system or its specific parts.

Since the basic distribution includes a set of tools for working with FPGA, you can make changes to the firmware directly from the OS, as well as read the FPGA configuration and perform basic control over it. Similarly, we can read and write from the HPS-to-FPGA registers.

To increase the flexibility of the system and simplify the development process, a separate set of packages has been built in. Using these tools allows you to automate interaction, create interfaces, and reduce the time required to update the FPGA circuitry and structure.

Table. 1

List of installed utilities [22]

Name	Functionality
FPGA-status	Reading the status of an FPGA structure
FPGA-readMSEL	Reading FPGA configuration mode via MSEL-Bit
FPGA-reset	Reset the FPGA structure and delete the current configuration
FPGA-writeConfig	Writing a new FPGA configuration
FPGA-readBridge	Read addresses (32-bit register) from HPS-to-FPGA Bridge, Lightweight-HPS-to-FPGA Bridge, or MPU (HPS) memory
FPGA-writeBridge	Writing addresses (32-bit register) to HPS-to-FPGA Bridge, Lightweight-HPS-to-FPGA Bridge, or MPU (HPS) memory
FPGA-gpiRead	Reading FPGA general purpose registers (GPI)
FPGA-gpoWrite	Writing FPGA general purpose registers (GPOs)

In addition to the above package, it is also worth adding the ability of the OS to read .rbf files when the kernel is running and update the FPGA part with a new version of the schematic. This set of functionality has a positive impact on the flexibility of the entire system, simplifies development, and allows you to implement dynamic processes while iterating on a project.

Access to external resources and repositories allows you to expand the system with the help of specific packages, both those provided by vendors and those published by the system developers. Using the embedded OS allows you to flexibly respond to changes in the environment, tasks, or development vector. It also makes it possible to implement more complex systems and combine them, expand their functionality, and provide access to external resources.

3.5. Tools and strategies to maximize system flexibility.

To achieve maximum flexibility in an embedded system, various tools and strategies can be used:

Modular architecture: Developing a system based on a modular architecture allows you to divide the system into small modules that can be independently developed, tested, and updated. This provides flexibility to make changes and extend functionality.

Use of configuration files: configuration files allow you to change the FPGA configuration, including changing logic, I/O, and other parameters without the need to completely recompile the code.

Use of IP cores: FPGA IP cores are off-the-shelf modules that can be used to implement various functional blocks. The use of IP cores allows you to quickly and efficiently integrate the desired functionality without the need for development.

Memory Management Unit (MMU): The MMU allows you to dynamically manage memory access, which provides flexibility in the use of memory resources and allows you to use memory efficiently for different tasks.

Use of virtual devices: Virtual devices provide an interface between the FPGA and external components or systems, allowing for easier integration with other systems.

The use of tools included in the OS package allows for faster and easier interaction with the FPGA. For example, the FPGA-status and FPGA-reset utilities allow you to read the status and delete an existing configuration, making the system flexible in management and debugging. Also, FPGA-writeConfig allows you to change the configuration, which allows you to quickly adapt the system to new requirements or work scenarios. In addition, the FPGA-readBridge and FPGA-writeBridge utilities extend the capabilities of interaction between the software and hardware environment, and FPGA-gpiRead and FPGA-gpoWrite provide the ability to interact with the system through general-purpose input register signals.

This package makes the system more flexible and adaptable to changes in requirements and use cases.

Similarly, access to external repositories allows you to make changes externally, using both an "over-the-air" update and a rolling release approach, which reduces the time it takes to deliver an update to the system. The support for configuring the FPGA from the FAT partition of the embedded OS allows us to have a stable startup configuration and, if necessary, return to it.

Since the system can update the configuration during operation, we can use a modular approach and targeted development of certain modules for functionality or to solve certain tasks facing the project.

Adding to this the ability to cross-communicate between FPGA and OS parts, we can freely operate memory and utilize configurable approaches based on data exchange between system elements. Taken together, the use of the tools described above allows you to build your strategies for expanding and maintaining the designed system.

4. Development of a user space system

Developing the user space in a Linux-based embedded operating system (OS) for an FPGA SoC involves creating software that runs directly on the FPGA SoC. This user space can include various components such as applications, services, drivers, and other software modules that provide the required system functionality [23]. Here are some key aspects of the third point:

1. Application development:

Create applications that interact with the FPGA SoC through various interfaces such as UART, SPI, I2C, Ethernet, etc. These applications can perform data processing, control devices, and provide a user interface to interact with the system.

2. Service development:

Create services that run in the background and provide certain functions or event processing. For example, services can manage network connections, collect and analyze data, and interact with other system components.

3. Driver development:

Create drivers to interact with FPGA SoC hardware components such as peripherals, storage, sensors, etc. These drivers allow the user space to access and interact with hardware resources.

4. Integration with FPGA resources:

The use of FPGA resources to implement specific functions or data processing. This can include the design and integration of logic blocks, signal processing, hardware accelerators, and other elements into the system's user space.

5. Testing and debugging:

Once the software is developed, it is important to test and debug it to ensure that it works correctly, to identify and eliminate errors, and to optimize system performance and reliability.

4.1. Create and configure custom components.

To create a user space, we need to configure and enable RootFS properly. Rootfs is a special instance of ramfs always present on Linux 2.6 systems. It is used as a location inside the Linux kernel, as a list with directories of the file system the place to search. RamFs is also a pointer to the points for starting the operating system, but it is focused more on working with RAM. Most systems simply mount another file system on top of it and ignore it, because the ramfs instance takes up very little space. Rootfs is a file system. In Linux, all file systems have a mount point, which is the directory where the mounted file system connects to the root file system. This is the difference between them, RamFs are the older ancestor of RootFs, but with a focus on placing a small amount of RAM.

The peculiarity of our approach is the use of symlinks (a virtual link to a specific directory that can be located in any other directory), which allows us to split the elements of the distribution while keeping it unified for virtual access.

One of the main differences in creating RootFS for Archlinux ARM is that we need to first create an SD card image, mount it, and then create our RootFS [24]. The build process can be summarized as a step-by-step script (see Fig.9)

```
# We need to start by getting core fs files and turn them into binaries
git clone https://aur.archlinux.org/qemu-arm-static.git

cd qemu-arm-static


makepkg -si

cd..

# Here we are getting filestructure
wget http://sg.mirror.archlinuxarm.org/os/ArchLinuxARM-armv7-latest.tar.gz

cd sdcard/ext4

# For decompression we need to utilize root users permissions
sudo su
tar xf ../ArchLinuxARM-armv7-latest.tar.gz .


cd ..

# To work with emulated rootfs we need to inject core binaries there
sudo mkdir -p ext4/usr/bin
sudo cp /usr/bin/qemu-arm-static ext4/usr/bin

# After preparations are done we can initiate session in newly created FS and begin
# customization
sudo arch-chroot ext4 /bin/bash
```

Figure 9: RootFS creation script

Following the scenario above, we will get a functional file system that can be modified depending on the needs and configured to meet the project's requirements. Such file systems do not take up many resources and do not put a significant burden on the entire scheme, since they contain minimal components and basic libraries for their extension. However, if necessary, they can be converted into a user-level system with all its pros and cons.

As a result, we will obtain the generalized OS which consists of several parts that are interconnected with each other (see Fig.10)

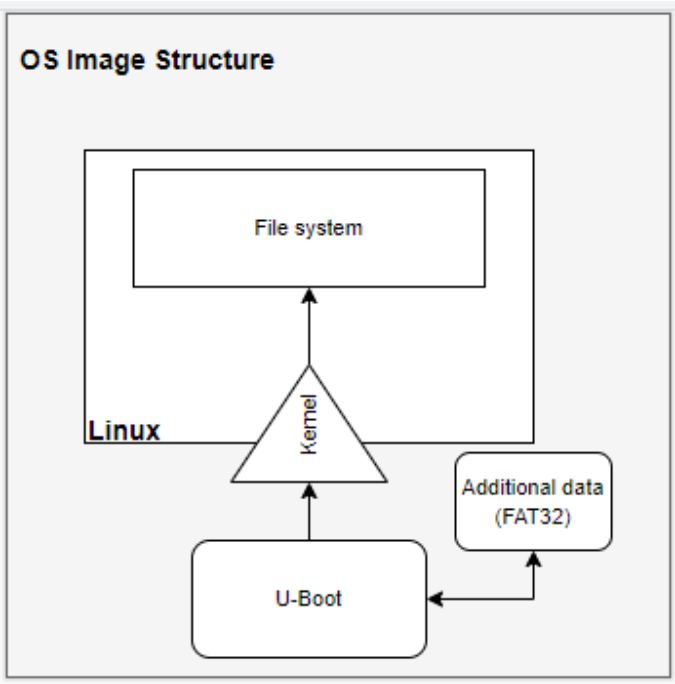

Figure 10: Embedded OS Image Structure

That composition means that we can update the specific part of the Image and the existence of the FAT32 additional component allows us to hook the default configuration that will be used as

a starter, or even a recovery point. Also, we can interact with additional data part from U-Boot, if required.

4.2. Interaction with FPGA SoC system resources from the user space level.

Interacting with FPGA SoC system resources from the user space level of the embedded operating system (OS) can be done using various tools and utilities, including table-based utilities such as FPGA-status, FPGA-reset, FPGA-writeConfig, FPGA-readBridge, FPGA-writeBridge, FPGA-gpiRead, FPGA-gpoWrite, and the devmem tool.

FPGA-reset: Allows you to clear the FPGA configuration to the basic level

FPGA-writeConfig: This utility allows you to write the FPGA configuration from the user space level.

FPGA-readBridge and FPGA-writeBridge: These tools will allow you to interact with bridges and interfaces between FPGA and other system components from the user space level.

FPGA-gpiRead and FPGA-gpoWrite: These utilities allow you to read input signals and write FPGA output signals from the user space level.

devmem: Allows you to read and write data to the device memory from the user space level. This can be useful for accessing various FPGA SoC resources, such as configuration registers, control registers, and others.

In general, the tools already in place allow integration and interaction with various aspects of the FPGA SoC from the OS user space level, making them important tools for simplifying development and interaction with FPGA. In addition to the above capabilities, the user can also use the direct memory access (DMA) option to quickly and efficiently read and write data to the FPGA SoC device memory from the user space level. This enables efficient use of memory resources and increases system performance. In addition, using the embedded Linux-based operating system, the user can easily use packages from external repositories. This extends the development capabilities as it allows the use of off-the-shelf solutions and libraries, which greatly facilitates the development, support, and expansion of the functionality of embedded systems based on FPGA SoC with a Linux environment [25]. In general, the available tools allow you to build interaction through HPS-to-FPGA and direct memory access, which simplifies the development of systems that can process and exchange data with FPGA. Knowing the addresses of the registers or the bus communication protocol, we can directly read and write binary data to this register using most programming languages or scripting tools (shell, as an example).

4.3. Using libraries to facilitate code development and optimization.

There are many advantages to using libraries for development on the embedded Linux for FPGA SoC with access to external repositories. Firstly, it allows us to significantly reduce development time, as we can use ready-made solutions to interact with hardware and software resources such as GPIO, SPI, UART, etc. Libraries also contribute to code optimization, as they provide an efficient interface to interact with these resources without the need to write your own code from scratch.

Furthermore, the use of external resources allows you to access a wide range of libraries and modules that extend the system's functionality. For example, we can include libraries for networking, data processing, creating a graphical interface, and much more. This helps to make the project more flexible and adaptive to changes in requirements or new features. In addition, by knowing which libraries and modules are required for your project, it is possible to create a standard package that can be included in a new release of the embedded OS. This simplifies the deployment and support of the system, as the standard package contains all the necessary components for the new version to work on the circuit.

Prototyping directly on the FPGA significantly reduces development time and makes it easier to understand how a system might behave in real-world conditions. Instead of using simulations or emulation environments, we can work directly with the FPGA, which allows us to identify and fix bugs faster and test different system scenarios more efficiently.

Thanks to our architecture, we can directly obtain test versions of the software code and check and modify them dynamically. This ensures a fast development cycle as we can work on new

features and modules efficiently. We can also update the FPGA circuit to suit our project without rebuilding the entire system using the tools described above. This allows us to respond to changes in requirements or identified issues and quickly make the necessary changes to the system hardware without stopping the software.

5. Conclusions

In the summary of the Linux kernel-based OS distribution for embedded systems based on FPGA SoC compared to Yocto and Buildroot, we concluded that using the Full Custom system with a set of built-in tools significantly speeds up development and simplifies prototyping. Based on this information, we built an OS distribution focused on rapid adaptation and simplification of project development on the FPGA SoC platform.

The resulting system has the necessary components built in to work on projects and the ability to dynamically add and remove libraries and modules. This approach greatly facilitates the process of code development and optimization. In addition, the availability of utilities that allow direct interaction with the FPGA system allows you to get test versions of the software code, check and modify them dynamically, and update the FPGA circuit according to the project requirements. This ensures efficient and fast system development based on the embedded Linux OS for FPGA SoC.

The final system, although not significantly different in resource requirements from Yocto, is more adaptable and can be ported to the Yocto base if necessary. This allows us to make the most of the Full Custom approach, ensuring an optimal balance between development efficiency, responsiveness to changes, and system resource requirements.

Acknowledgments

Oleh Krulikovskyi thanks the Intel company for the equipment provided within the Intel FPGA Academic Program.

References

[1] Ross K. Snider, Advanced Digital System Design using SoC FPGAs, Springer Cham, 2023. https://doi.org/10.1007/978-3-031-15416-4

[2] Arifur Rahman, FPGA Based Design and Applications (1st. ed.). Springer Publishing Company, Incorporated, 2008.

[3] Andres Upegui, Andrea Guerrieri and Laurent Gantel (Ed.), Special Issue Reprint Applications Enabled by FPGA-Based Technology, Electronics, MDPI, Basel, 2023. doi.org/10.3390/books978-3-0365-8785-1

[4] Dheeraj Punia (December 7, 2023) FPGA Design, Architecture and Applications URL:https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/

[5] Fuming Sun, Xiaoying Li, Qin Wang and Chunlin Tang, FPGA-based embedded system design, in: Proceedings of the 2008 IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2008, Macao, 2008. doi: 10.1109/APCCAS.2008.4746128.

[6] Mastura D. Marieska, Paul G. Hariyanto, M. Firda Fauzan, Achmad Imam Kistijantoro,and Afwarman Manaf, On Performance of Kernel Based and Embedded Real-Time Operating System: Benchmarking and Analysis, in: Proceedings of the International Conference on Advanced Computer Science and Information System. ICACSIS 2011, Mercure Convention Centre, Jakarta, Indonesia, 2011 .

[7] Bare-Metal, RTOS, or Linux? Optimize Real-Time Performance with Altera SoCs URL: https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/wp-01245-optimize-real-time-performance-with-altera-socs.pdf.

[8] Cyclone®V Hard Processor System Technical Reference Manual (cv_5v4 2023.08.28). URL: https://www.intel.com/programmable/technical-pdfs/683126.pdf.

[9] Building embedded Linux for the terasic de10-nano URL: https://bitlog.it/20170820_building_embedded_linux_for_the_terasic_de10-nano.html

[10] BlackBerry QNX: Real-Time OS and Software for Embedded Systems URL: https://blackberry.qnx.com/en
[11] FreeRTOS™ Real-time operating system for microcontrollers URL: https://www.freertos.org/
[12] VxWorks: The Leading RTOS for the Intelligent Edge. URL: https://www.windriver.com/products/vxworks
[13] The Yocto Project It's not an embedded Linux distribution, it creates a custom one for you. URL: https://www.yoctoproject.org/
[14] The Buildroot user manual URL: https://buildroot.org/downloads/manual/manual.html
[15] Welcome to OpenEmbedded URL: https://www.openembedded.org/wiki/Main_Page
[16] Terrasic DE10-Nano Manual 2018
[17] Yocto Project Documentation 2024 URL: https://www.yoctoproject.org/docs/.
[18] Embedded Linux System Design and Development / P.Raghavan Amol Lad Sriram Neelakandan Taylor & Francis 2019
[19] Linux From Scratch Gerard Beekmans URL: https://www.linuxfromscratch.org/lfs/
[20] Alex Gonzalez. Embedded Linux Development Using Yocto Project / Chris Simmonds. – BIRMINGHAM - MUMBAI: PACKT, 2018.
[21] What is the Linux kernel? (2019, February 27). RedHat. URL: https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel
[22] Robin Sebastian RSTools URL: https://github.com/robseb/rstools
[23] Abbott, D. Linux for Embedded and Real-Time Applications; Butterworth-Heinemann: Oxford, UK, 2003.
[24] Rocketboards WIKI URL: https://www.rocketboards.org/ foswiki/Documentation/ WebHome
[25] Li Y, Matsubara Y, Takada H (2018) A comparative analysis of RTOS and Linux scalability on an embedded many-core processor. J Inf Process 26:225–236