embeddedworld2020
Exhibition&Conference
... it's a smarter world

From design to run-time: A practical approach to reconfiguration

Mike Looijmans
System expert TOPIC Embedded Products
Best, The Netherlands

***Abstract*—They have been around for a while: Devices combining multiple CPUs, GPUs, co-processors and FPGA fabric on a single chip or module. All these components need programming. And all these are reconfigurable: Their function can be altered in various ways at runtime. This presentation will explain how to do that in practical examples.**

Free yourself from the limits of the classic boot sequence where the bootloader configures the pins and programs the FPGA and drowns the system configuration in virtual concrete. In small steps, make your system adapt itself, allowing access to a broader range of applications while also reducing the maintenance on board and device support.

This paper will focus on the tools and infrastructure available today to actually put this technology to work. We'll discuss some practical applications of how we can use Linux' Devicetree overlay support to make the system adapt itself at runtime and how one could deal with add-on boards.

Keywords—reconfiguration; FPGA; boot sequence; devicetree overlay

I. PREFACE

I think it was Socrates who said that the first step in gaining knowledge on a subject was realizing that one doesn't have that knowledge. In other words, knowing that one does not know. Further, in order to create something new, one first has to be convinced, or at least aware, that such a thing can actually be accomplished. In this paper, I aim to create some of that awareness and conviction. Although the presentation will provide more details and examples of "how" things can be done, this paper restricts itself to explaining "what" is possible and "why" that would be useful.

Enough ancient Greek philosophy. The topic at hand is reconfiguration in embedded systems. After several projects involving embedded Linux and FPGA logic, it occurred to me that many parts of the system offer a lot more flexibility than I had been using so far. Whereas I/O pins used to have fixed functionality and FPGAs had to be programmed

in flash memories, nowadays pin assignments are programmable and FPGAs are completely RAM based and can be reprogrammed in part or in whole many times per second. But I was still using them in the classic way of just programming them once at startup. The first step was made – I knew I didn't know. And it turned out that this wasn't some rocky back-road I stepped on, but a well-paved pathway.

II. INTRODUCTION

The boot sequence of embedded systems running Linux or a similar OS is roughly the same for many devices. A quick overview:

- ROM – The built-in boot code of the CPU picks a storage device to run from. It loads the first stage into on-chip RAM and jumps to an entry point in that code.
- First stage – Sometimes vendor-provided, but more often a part of U-boot that runs from on-chip RAM. Its main purpose is to set up pin multiplexing, configure boot peripherals, and initialize the external memory (usually DDR SDRAM). It then loads the second stage into external RAM and proceeds from there.
- Second stage – Usually a U-boot instance that further configures hardware, loads the Linux kernel, devicetree [2] and optional extra data into memory, and then passes control to the kernel.
- Operating system – Usually a Linux kernel and a root filesystem on a storage device or in RAM. The kernel will wake up extra processor cores on multi-core systems, and finally load and execute user applications.

This list is by no means exhaustive, some platforms need additional steps (e.g. ARM64 requires extra firmware to be loaded and executed after the first stage) and some need less, some bare metal systems even combine all steps in a single package.

In this paper we'll assume that the system uses the boot sequence described above.

III. Use cases

Run-time configuration has many use cases. We'll list a few in this chapter.

Reuse. Once we built a good product, we want to build an even better one after that. So we want to expand on what we have. But we'll also want to backport things from the new products into existing ones. If the bootloader is tightly coupled to the application, we won't be able to do so. So we design the bootloader such that it does the absolute minimum to get the system up and running. That allows us to share more code between platforms using similar chips, reducing the time to market for new products, and improving the maintainability of existing ones.

Calibration. Measurement systems often have complex calibration procedures that take up considerable resources that are not actively used during measurement. If the calibration routine requires a lot of gates in the FPGA, these resources would be wasted in a static system that just programs the FPGA only once at boot. If we re-program the FPGA for either calibration or measurement use, we can use a smaller one.

Performance. Postponing configuration steps allows us to display a user interface while the hardware is still running high-speed link training and tests for example. This yields a much more responsive system.

Add-ons: Many evaluation boards offer expansion capabilities. Run-time configuration allows us to detect what is connected and dynamically load and initialize modules to control add-on boards.

Development. Being able to create a new or partial FPGA component and deploy and test it immediately without even having to reboot the board provides much shorter development cycles than having to build some boot image first, writing that to flash memory and then rebooting. There is also much to gain from a development setup where we only have to add things to an existing setup rather than having to start from scratch.

IV. Practical: Pin multiplexing

Pin multiplexing, in short "pinmux", is found on many systems and on different levels. It is common for CPU and SoC pins to implement multiple functions, so a set of pins can be used for either SPI, I2C, or plain GPIO. This usually extends to add-on connectors, and even to standard connectors like a USB-C port that can also provide HDMI output on its pins.

In our introduction we listed "pinmux" as a task for the first stage bootloader. This often makes sense, the first stage loader is usually generated from board information and pins don't suddenly change function. Most systems limit themselves to configuring the functionality of the chip's or board's pins at boot and then never change it afterwards. The pin multiplex configuration is actually runtime configurable and can be changed at any time.

A simple application is to load a bootloader from an SPI NOR flash chip, and then change the pin configuration and use some of the same pins for NAND storage. Such a system can even have both devices accessible by just changing the pin configuration on demand.

Something that has become a standard in Linux is to be able to change the SDA and SCL pins of an I2C controller to GPIO mode. This allows the system to recover a "stuck" I2C bus, or implement protocol parts that the I2C controller cannot handle itself.

Also common in Linux systems is to reduce power consumption by changing the pin configuration when a device is in low-power mode. This usually serves to reduce power consumption in I/O circuitry, but may also be used to prevent invalid I/O or glitches when a peripheral is put in low-power mode to propagate to other parts of the system.

V. Practical: FPGA firmware

Let's look at the boot sequence for the Xilinx Zynq series. The vendor-provided boot flow creates a first-stage loader that configures pins and clocks, and programs the FPGA, and then launches u-boot which in turn loads the kernel and boots into Linux.

This means that if we want to make changes to the FPGA, we'll need to re-program the first-stage bootloader, a 20MB package because of the FPGA content, onto the board. Since the kernel is configured statically, every component inside the FPGA must be ready and configured when the kernel starts. So let's try and make this a more flexible system.

First thing to do is to move the FPGA programming away from the first stage loader. Both u-boot and Linux kernel have the required support to do so. So we remove the FPGA image from the first stage image and place it in the root filesystem under /lib/firmware.

Now that the FPGA isn't programmed when the kernel boots, we have to make sure that drivers do not attempt to access the non-existing logic. So we need to move all FPGA-implemented devices from the 'static' device-tree that the kernel loads at boot into an overlay [1]. This overlay also triggers the FPGA "firmware" load.

The boot sequence is now that the system boots rapidly into Linux. When the root filesystem attaches, the kernel programs the FPGA, and then activates all peripherals that are being addressed through the FPGA. This provides the following advantages:

- The FPGA bitstream can be stored on any medium that the kernel can access, so the bitstream can be on a compressed medium, the network, or a fault tolerant filesystem for example.
- If the FPGA fails to configure, the system can still provide limited functionality to resolve the issue, instead of simply halting.
- It is possible to roll back the FPGA overlay, which also stops all related drivers. This allows the system to completely reprogram the FPGA without rebooting. Also an interesting option in dealing with single-event upsets, the system can quickly re-program the FPGA after a malfunction.
- Detect available hardware (e.g. expansion boards, system revision) and load alternative overlays based on the outcome.
- More efficient boot, the kernel continues to run other tasks while the FPGA is being programmed.

VI. PRACTICAL: FPGA PARTIALS

FPGAs these days also support partial reconfiguration, which allows the system to program only a part of the FPGA while the rest of the FPGA content remains untouched and continues operating without interruptions. The Linux kernel supports this using “bridge” connectors. The kernel de-activates a bridge, programs the partial and then re-activates the bridge. This allows one to insert and remove PCI peripherals or other bus-connected (e.g. AXI) devices while the system remains running. This opens up many more interesting possibilities, for example:

- Load and activate a ADC/DAC interface in the FPGA when an applicable expansion board is detected.
- Load and activate accelerators (or algorithms) on software demand. For example mathematical or encryption modules.
- Automate hardware emulation for system tests.

VII. CLOSING ARGUMENTS

I know now that I know more than I used to. I sincerely hope that this article has inspired awareness and helps in actually implementing some changes.

I thought it surprising that all the infrastructure for implementing run-time reconfiguration like pin multiplexing and FPGA configuration was already in place, but that even the vendors implementing it did not actually make good use of it. Many a reference design still sticks to a “one application one bootloader” approach, thus having to start from scratch in every new project. So I also want to make an appeal to the providers of board support packages and software design flows to consider using more run-time configuration.

REFERENCES

[1] “Device Tree Overlay Notes”, part of Linux kernel documentation, https://www.kernel.org/doc/Documentation/devicetree/overlay-notes.txt

[2] “Linux and the Device Tree”, part of Linux kernel documentation, https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt

About Mike Looijmans

Mike Looijmans has been working for TOPIC Products since its initiation in 2014. His daily activities as system expert consist of bringing up new hardware, developing board support packages, participating in new designs, writing driver and application software and educating and assisting his colleagues. Since his electrical engineering study he has worked in various high-tech companies as a consultant and developed in-depth knowledge in fields ranging from management information systems to highly integrated embedded systems.

Contact: Mike.looijmans@topic.nl

TOPIC, Materiaalweg 4, 5681 RJ Best, The Netherlands
Website : www.topic.nl
LinkedIn: www.linkedin.com/company/topic-embedded-systems