Porting RIOT OS to the ARM Cortex™-A9

Jakob Lennart Dührsen and Leon Martin George
Institute of Computer Science
Freie Universität Berlin
Takustrasse 9, 14195 Berlin, Germany
Email: {lennart.duehrsen, leon.george}@fu-berlin.de

***Abstract*—This is a report on the process of adding basic support for the RIoTboard to RIOT-OS as part of the software-project 2014 of the working-group "Computer Systems & Telematics" at Freie Universität Berlin. The port is based on the i.MX6 Platform SDK 1.1 and at the end of the project it is in a state where LEDS, the UART2, timers and interrupts have interfaces to RIoT-OS.**

I. Introduction

All the files we used while porting that are not used for building, such as documents, source code for working software, instructions and overviews, were gathered in a gitlab-repository on spline called "swp-report" the owner ist "backenklee".

II. Preparing For The Port

In order to complete the project successfully, we agreed on these goals with the teaching-staff:

- The RIoT-OS-wiki has pages on working with the hardware
- RIoT-OS compiles for the RIoTboard
- Interfaces to timers, UART and interrupts is implemented

As we had no idea how hard it was going to be, we were very generous with the curfews for the different steps.

A. Hardware and Information

The RIoTboard is a rather unusual board for the RIoT-OS to run on. It is based on the "i.MX6Solo" freescale-architecture, has an ARM cortex-A9, many different interfaces and is supposed to be used mainly by developers. It is shipped without a power-supply and with a USB-cable.

The manufacturer supplies two relevant documents: A Quick-User-Guide/User-Manual/ Technical-Data-Sheet and the board schematics. The board-manual consists mainly of a list of interfaces with picture of where to find them on the board. If you have physical access to the board, the only thing useful you will learned from there is which mini-USB-port is for OpenSDA and which is for the serial-connection. At the end there is a step-by-step instruction on "flashing" linux or android onto the board, by using a Windows-only tool, arranging the eight DIP-switches on the board in three different ways. The schematics file yields a bit more information, such as which pin is connected to what. Other pieces of information, like which pin of UART2 is GND/TXD/RXD, is missing. You can find that information in the repository mentioned in the introduction ($briefings/connect\_debug$). In order to understand the way the board works the i.MX6SDL processor reference manual is the most helpful. All the interfaces and the processor, including the boot-process, are described in there.

B. Software Running on the RIoTboard

On the boards website the manufacturer - embest-tech - offers binary-images and instructions for running android or ubuntu, of which android is installed by default. Both operating systems rely on u-boot as the boot-loader. The source-code of those ports is available in moderately-hard-to-find repositories on the internet and the code itself may somehow be re-useable.

Due to their structure, the linux-code and the u-boot-fork were useful in different ways: u-boot requires the seperation of syscalls, the "flash-header" - which helped understand the way the i.MX6 boots - and linker script. So with u-boot-imx embest-tech provides three files for those and one giant source-file for all the rest. The linux-port seems better structured but due our lack of knowledge about the kernel-nuild-process it was hard to use any parts of the source. Also, as far as initialisation goes, linux seems to repeat some of the steps that have already been done by u-boot and does some differently.

The focus on developers is somehow restricted to those two operating systems. There is no official support apart from discussions on a channel of an IoT/embedded-focused developer forum.

C. The Attempt of using u-boot

We really lost a lot of time on our attempt at basing the port on u-boot-code. During most of that time we did not produce anything that was really relevant because we had problems figuring out, how u-boot works during the build-process. After three weeks the instructors gave us the advice to look for an alternative and that even though we had not been able to run anything with it, the SDK looked more promising.

III. Booting the i.MX6

While trying to find a simple way of flashing the ROM of the cortex-A9 we learned that freescale does not want anybody to flash the CPU directly. Instead there is a program on the ROM that checks hardware-signals called **FUSE**s to decide how the CPU boots. Essentially, they allow to boot from one of four *USDHC*-block-devices or a serial-download-mode. In theory, **FUSE**s can be used to change other parameters for booting.

In case of the RIoTboard the first three *USDHC*-lines are 4 GB of internal eMMC, the SD- and the μSD-card-slot. The serial interface is wired to the mini-USB-port next to the ethernet-port and UART2-pins. Slightly more-detailed information can be found in the gitlab-repository mentioned in the introduction ($briefings/boot-switches\_and\_-modes$)

At startup, the program on the ROM loads a chunk from the beginning of the chosen boot device and checks the existence of an IVT (Image-Vector-Table). This table contains the start address of the program-code, a checksum and a pointer to DCD (Device-Configuration-Data). DCD is used to initialise register-memory and consists of pairs of addresses and 32-bit values which will be written to the registers at the preceding address. Details on this can be found in the i.MX6 processor reference manual in the chapter "Image Vector Table and Boot Data". The offset of the IVT for the RIoTboard - as all devices are connected via UDSHC - is 1KB - which allows a MBR to be placed on the device also - and the size of the initially loaded chunk is 4KB, so the first partition should not start before $1400. Too quickly get an image to run on the RIoTboard follow the steps in $build\_and\_run\_with\_uboot\_from\_sd-card$.

IV. Running our own Software

The simplest thing we could get to run on the RIoTboard was u-boot: Either via the serial connection, where the freshly transferred u-boot and an initramfs could be used to receive files or binary that could then be stored on the board, or by writing the binary onto a SD-card. From within the running u-boot it was possible to execute a method from a cross-compiled object-file that simply returned an integer value. The brief step-through for building u-boot and running it on the RIoTboard in our repository also contains information on the object-file-test ($briefings/build\_and\_run\_with\_uboot\_from\_sd-card$).

We were trying to figure out how to use u-boot for booting and supplying the standard library but all the files were spread all over the u-boot-source-directory. As we were realising how difficult it was going to be to not only have a functional binary based on our own code but also integrate the existing structure into the make-system of RIoT-OS, we didn't think we would be able to meet any goal beyond supplying a rudimentary framework so that coding can start in the RIoT-OS-codebase.

By then we were behind schedule as we could not get anything to work that was not built on top of and run from within u-boot.

V. Integrating the i.MX6-Platform-SDK into RIoT-OS

Luckily, freescale supplies a SDK for the i.MX6-platform that is able to run on their reference-boards. We agreed with the teaching-staff to rather use the latest version of this SDK (1.1) because it is closer to developing applications in a C-manner, a bit more minimalistic and the source-tree is well-structured without making it too hard to find a particular piece of code and also, there are hardly any source-files that are irrelevant to this particular port.
Then again, sadly, none of the targets for the SDK seem to produce code that runs on the RIoTboard and was not clear whether we would be able to produce anything using the SDK but it seemed like the right way to go because of its rather clean interfaces.

The i.MX6-platform-SDK brings a *make*-structure that has applications as targets and allows a target-board to be specified via parameters. After the board has been chosen, CPU- and board-specific code will be included based on defines. These defines were passed as parameters to **make**. We did not see any sense in adopting this pattern in RIoT-OS as it would have made changes to the *make*-system necessary. In order to avoid this problem, we picked the defines relevant to the i.MX6Solo and the RIoTboard and changed the code manually as if the preprocessor had run.

This has major implications for future ports to i.MX6-variants other than the Solo/DualLite: Every source- or header-file of the un-modified SDK containing *#ifdef*s or *#ifndef* potentially contains code important for a successful port. Because of our concerns about completing the project in time we decided not to develop a proper concept to deal with this issue. We do however have a proposal which might turn out to work quiet nice without putting to much effort into it:

- Add prefixes to SDK-macro names (e.g. **IMX6_SDK_**)
- Define the necessary macros in the *board-sdk.h*
- Include the header file in every SDK-file.

We could implement this relatively simple technique but we dont have the means to test it and expect a better way to exist.

The next step was to correctly build the SDK when **make** was run from within RIoT. As even the *Makefile*s of the SDK worked with defines, we decided to scrap and re-write them. Understanding the *make*-structure of RIoT-OS and creating *Makefile*s for the SDK took us about a week.

Even though practically everything apart from the low-level initialisation-code needed minor to moderate adjustments we were able to skip the step of building a minimal program and go directly to builing the hello-world-program of RIoT-OS.

VI. Enabling Interfaces of the RIoTboard within RIoT-OS

As all the components of the RIoTboard have a driver in the platform-SDK we did not expect these steps to be too hard. However, we heavily underestimated the importance of the IOMuxer on the board. This component has no driver in the SDK, as it is not supposed to be manipulated while the program is running. Instead you are supposed to use a windows-only tool called "IOMux-Tool" which allows muxing pins by clicking on the desired functions. It automatically checks for conflicts and can generate source-files containing macros that will work on the right registers without you having to worry about muxer-config after you have used the tool. We added a way to correctly alter and copy the generated source files into the right directories. See the corresponding briefing in our repository for instructions on using it ($briefings/muxer\_config$). If you are not using this script, please make sure the IoMuxDesign.xml always matches the set of macros currently in the riotboard-directory of RIoT-OS.

A. LED

During our preparation for running a minimalistic C-program on the RIoTboard we had looked up how the connection of the CPU to the LEDs work. With this knowledge is was very easy to write code for flashing the LEDs using the SDKs GPIO-driver. The program we were running then needed only the GPIO-pins to be configured in the muxer.

B. UART

We also spent a large amount of time on enabling UART2 for I/O-support: Although the UART-initialisation ran on the right registers and nothing seemed to be wrong with with the code itself, the UART did not do anything. Neither could we receive any output nor did the board process any of the signals we sent. Without the UART our only means for testing were the LEDs. When we thought it could be only the output-channel that was not working, we tried toggling the LED upon reception.

We tried replacing the default SDK-code for initialisation with u-boot-code that was altered to use the SDK-macros. This version could be fixed by rearranging the instructions in a different order. We did not understand why this worked.

In the end, we had - by accident - enabled a mechanism that would forbid using the UART from being used while its buffer is overloaded. This mechanism stopped the UART from working altogether because the other side was not configured to use it. For some reason this was also activated in the default SDK-code.

C. Timers

Having missed our last deadline we decided to get help understanding the timer-interface of RIoT-OS before we started working on this subsystem. We were told there was a clean implementation using another layer of timer-abstraction in the code for another processor. But as soon as we started implementing this new lower-level interface we noticed that the SDK already offered a higher-level abstraction and scrapped the code we had so far. There were only slight conceptual problems with mapping the SDK-driver to the RIoT-OS-interface because of the SDK offering slightly differenct functions than RIoT-OS expects.

D. Interrupts

The SDK already offers a way to map interrupt-callback-routines to interrupt-IDs. We do not think this mechanism should be used directly. The drivers probably are the right way to use interrupts. This practically gave us a milestone for free. Apart from that, we would not have had the time to do anything else before cleaning the code. And the code being useable in the future is more useful than another interface.