Layerscape Kernel Manual

System and Networks

University Augsburg

Dominik Walter

October 10, 2017

Contents

1 Introduction

This document describes the installation (section 2), concept (section 3) and usage (section 4, section 5) of the `layerscape_kernel`. For a deeper insight please refer to the given references. The kernel is written in modern C++ for ARMv8 and the `freescale LS2085ARDB`-board including 8 ARM-Cortex-A57 cores. An U-Boot installation is required. Currently 3 separated code-projects exist (see 1.1), which serve as templates for future extensions.

1.1 Projects

- `layerscape_kernel`: The Kernel and all of its modules.
- `layerscape_shell`: A simple Command-Shell.
- `layerscape_helloworld`: A Hello-World-Program.

1.2 License

The kernel uses a small amount of code from the U-Boot-project ([5]). Therefore the GPL-Licence is applied to this project as well.

2 First Steps

2.1 Installation

Executing the script `install.sh` will first download and install all required binaries and then start `configure.sh` to set up the build configuration. Afterwards every project should have a build folder and a symbolic link `tftp` to the image directory `/var/tftproot/`. On non-ubuntu systems, you may need to adopt these scripts.

2.1.1 Required Packages/Binaries

- *aarch64-elf/gcc-linaro-5.4.1-2017*
- *xinetd*
- *tftpd*
- *tftp*
- *u-boot-tools*
- *make*
- *cmake*
- *screen*

2.2 Building Process

Since `cmake` is already configured, all projects can be build directly by calling `make` in their respective build directories. After compiling, the executables will be converted to static images (`.img`), where the kernel gets additional header information for u-boot (`.uimg`).

2.3 Starting U-Boot

Plug the serial cable into the `UART`-Port 2, type `'screen /dev/ttyUSB0 115200'` to start a new session on the host system and finally press the Power-Button to boot into U-Boot. Once finished, you need to cancel the autoboot by pressing any key.

2.4 Deployment

2.4.1 Flash

You can use the following commands in U-Boot to install an image called `layerscape_kernel.uimg` to the internal flash memory. Thereby the installed kernel can be started with the `boot` command or the autoboot functionality.

```
dhcp;
setenv serverip 137.250.172.39;
tftpboot layerscape_kernel.uimg;
erase $kernel_start +$filesize;
```

```
cp.b $loadaddr $kernel_start $filesize;
setenv kernel_size $filesize;
setenv bootcmd "cp.b '\$kernel_start' '\$kernel_load' '\$kernel_size'
&& bootm '\$kernel_load'";
saveenv;
```

2.4.2 RAM

If no persistant storage is neccessary, the image should be loaded directly from the RAM to prevent frequent flash erases. This can be done with the following code sequence:

```
dhcp;
setenv serverip 137.250.172.39;
tftpboot layerscape_kernel.uimg;
cp.b $loadaddr 0xa0000000 $filesize;
```

Without the flash you need to specify the start address to boot the kernel.

```
bootm 0xa0000000;
```

2.5 Additional Applications

Any other application should be stored in the RAM using the following command-structure. Currently the kernel exspects an application at `0xB0000000` with no entry-point offset.

```
tftpboot <FILENAME>.img;
cp.b $loadaddr <DEST> $filesize;
```

The kernel reserves the address space `[0xB0000000, 0xBFFFFFFF]` for applications.

3 Overview

3.1 Boot Process

3.1.1 Early Initialization

The kernel starts execution in `startup.S`. While the primary core runs in `Exception Level 3 (EL3)`, all secondary cores are already in `EL2`. Therefore the entry point differs. Before the program can branch to a C/C++ function, some pre-initialization needs to be performed. First of all, the MMU will be disabled and the caches invalidated. Afterwards the kernel defines a new exception vector, branches to `EL1` and initializes the stackpointer. Finally it can set up some system registers (e.g. traps, masks) and leave the assembly-level by entering `entry.cpp`.

3.1.2 System Initialization

Before any module can be loaded, the system class, which provides the fundamental module interface, needs to be initialized. All cores share the same system, hence no operation from the secondary cores is required. Then the kernel continues with `module_list()`.

3.1.3 Module Initialization

In `module_list()` all modules are loaded in a sequential order. Therefore every module calls their `load()` member function. However, secondary cores will ignore all modules with `multi_load = false`. If any call returns false, an error will be 'thrown' and the kernel terminates. Otherwise the boot was successfull.

3.1.4 Application Start

Once all cores completed their boot phase, the system class calls a main function, which was registered during the module initialization. Generally this should be a task scheduler and not an usermode application.

3.2 Process Scheduler

The scheduling module provides a simple preemptive `Round-Robin-Scheduler` with a fix timespan for each process. However the module supports static scheduling, which disables timer interrupts for the specific core. Before switching to a new task, all pending signals (e.g. SIGTERM) will be executed. Every core shares the same task queue.

3.3 Virtual Memory

The address-size is 40 Bit, with the leading 24 Bit required to be either 0 (kernelmode) or 1 (usermode). Hence the Virtual Address Space for `EL0` starts at 0xFFFFFF0000000000, while the Virtual Address Space for `EL1` ends at 0x000000FFFFFFFFFF. For translation, a four level lookup table with 4KB granularity is used.

3.4 Kernel Interrupts

Every IRQ and FIQ exception is masked at `EL1`. Therefore no interrupt will be taken.

4 Coding Guide

4.1 General Structure

The kernel project is divided into 3 namespaces, with corresponding directories in `src`. The `kernel` namespace contains the code necessary for module loading. This includes the startup files for booting and the system class. (see 3.1.2). Each module is placed in the `module` directory with an optional nested namespace. All helper functions, which do not belong to any of those 2 groups, are located in the `util` namespace. Constructors and Destructors should be avoided, since they require C++ exceptions for error handling and `new/delete` for non-stack allocation. Both are not available in the current lowlevel environment. Finally the usage as a member of an other class without a constructor and global definition would be restricted as well. Instead `init/destroy` member functions should be used.

4.2 Code Examples

4.2.1 Basic Module

All modules are pure static classes with no instantiation. The following example shows the basic code structure and serves as a style reference. For clarity the whitespace was reduced from the original 4 to 2.

src/modules/example.h:

```
#ifndef modules_example_h
#define modules_example_h

namespace module {
  class example {
  public:
    struct info {
      constexpr static const char* title = "Example";
      constexpr static const bool  multi_load = false;
    }
    static bool load();
    static void functions_with_underscore();
  private:
    static int m_exampleMember;
  };
}

#endif
```

src/modules/example.cpp:

```
#include "src/modules/example.h"

namespace module {
```

```
  bool example::load() {
    return true;
  }

  void example::functions_with_underscore() {}

  int example::m_exampleMember;
}
```

src/module_list.cpp:

```
kernel::system::load<module::example>();
```

4.2.2 Syscalls and Interrupts

A syscall is always part of a wrapper module in the `syscall` namespace and can be registered using the `syscall::controller`. This example maps the syscall 45 to a simple addition and shows the usage in application code. The registration of interrupts (e.g. timers) is analogous.

src/modules/syscall/example.h:

```
struct example_syscall {
  constexpr static const int id = 45;
  static void handle(kernel::register_maps::general_registers& r);
};
```

src/modules/syscall/example.cpp:

```
bool example::load() {
  controller::register_syscall<example_syscall>();
  return true;
}

void example::example_syscall::handle(kernel::register_maps::general_registers& r){
  r.x0 = r.x0 + r.x1;
}
```

application.c:

```
// returns x + y
extern "C" uint64_t add(const uint64_t x, const uint64_t y);
```

application.S:

```
.globl add
add:
  mov   x8, #45
  svc   #0
  ret
```

5 Application Interface

5.1 Run Executable

Since the kernel does not support a filesystem, every application needs to be loaded directly from a memory address. Besides that, the first instruction may not be on the first memory location. Hence you need to specify the entry point, which can be obtained from the build messages. Since the kernel reallocates the application to a new virtual address, the application size is required as well (see 2.5). You can either create new processes directly from the kernel using `create_process()` from the `Scheduling module` or use the `layerscape_shell` project.

5.2 Implemented Syscalls

Currently these syscalls are implemented. See `syscall.h` and `syscall.S` from the application projects for more details.

Name	Description
`write`	Writes to the serial console
`read`	Reads from the serial console
`malloc`	Allocates memory in 4096 byte blocks
`free`	Frees an allocated memory region
`pid`	Returns the current process id
`thread_create`	Creates a thread
`thread_join`	Blocks until the thread has terminated
`process_create`	Creates a process
`EXIT`	Exits the current process/thread

Table 1: List of implemented syscalls

5.3 Implemented Shell

`layerscape_shell` provides a simple command-shell for starting applications directly from memory. Each command represents an image and can be registered with the following code sample. Currently no arguments are supported. The kernel exspects the shell at `0xB0000000`.

app_list.cpp:

```
struct example {
  constexpr static const uint64_t address      = 0xb1000000;
  constexpr static const uint64_t entryAddress = 0xb10000c0;
  constexpr static const uint64_t size         = 4096;
  constexpr static const char*    command      = "example";
};

void app_list {
  app::register_app<example>();
}
```

6 Troubleshooting

As a rule of thumb, you should always try to avoid errors in the kernel, since any exception in `EL1` is taken to `EL2`. The MMU is only enabled for `EL0` and `EL1`, hence any access to memory from `EL2` will ignore written cache entries. Thereby error messages may contain wrong information.
The kernel distinguishes 3 different error types. More details are provided by the ARMv8-documentation for `ESR_ELx` on page 2254 [1].

Type	Reason	Action
Error	Invalid syscall or interrupt	Continue process
Fatal Error	Execution fault	Kill process
Kernel Panic	Sync, FIQ, SError	Halt core

Table 2: Error types

7 References

[1] *ARM Architecture Reference Manual ARMv8 — for ARMv8-A architecture profile*. URL https://static.docs.arm.com/ddi0487/b/DDI0487B_a_armv8_arm.pdf.

[2] *ARM Bare-metal Boot Code for ARMv8-A Processors*. URL http://infocenter.arm.com/help/topic/com.arm.doc.dai0527a/DAI0527A_baremetal_boot_code_for_ARMv8_A_processors.pdf.

[3] *ARM Generic Interrupt Controller Architecture Specification — GIC architecture version 3.0 and version 4.0*. URL https://static.docs.arm.com/ihi0069/c/IHI0069C_gic_architecture_specification.pdf.

[4] *Serial Programming/8250 UART Programming*. URL https://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming.

[5] DENX Software Engineering. *U-Boot Code-Repository*. URL http://github.com/u-boot/u-boot.

[6] Linux Project. *Device Tree for LS208xa*. URL https://github.com/torvalds/linux/blob/master/arch/arm64/boot/dts/freescale/fsl-ls208xa.dtsi.