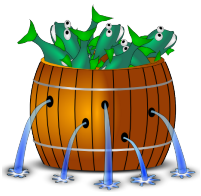

ADVANCED OPERATING SYSTEMS
Milestone 7: Memory copy offloading
Fall Term 2016

Assigned on: **25.11.2016** Due by: **16.12.2016**

1 Overview

In this last milestone you will implement a device driver for the sDMA (System Direct Memory Access) engine on the OMAP4460 to use and evaluate its capabilities. A DMA engine allows you to offload various memory operations such as transferring memory between devices, but also copying and writing to and from memory directly without any involvement of a CPU. The goal of this project is to use the sDMA engine to offload regular `memcpy` operations in an application to the engine.

The work consists of:

- Implement a driver for the SDMA engine.
- Design a safe interface for applications to use the driver.
- Implement a version of `memcpy` that uses your driver as part of your OS.
- Benchmarking the performance of your implementation to find heuristics on when it makes sense to use the DMA engine.

2 Overview of the sDMA Device

In order to write a device driver for the sDMA engine, you will have to understand how to program it. Luckily, the device is described in detail in Chapter 17 of the OMAP4470 Technical Reference Manual revision T. The most relevant sections for you are Section 17.4 “sDMA Functional Description”, Section 17.5 “sDMA Basic Programming Model” together with Section 17.6 “sDMA Register Manual”. You should study those chapters briefly before you start working on the driver and get a high-level understanding of the device. However, you have been warned that even this (arguably simple) device has a lot of functionality which you will not need to implement (such as working with multimedia, endian conversion, channel linking, packed port access, burst transfer, auto-restore etc.). It’s important not to get lost too much while reading the documentation and rather try to find the relevant information to achieve your goal (copying a buffer from A to B).

The internals of the sDMA module are abstracted by 32 logical channels for the configuration. A channel is responsible to execute a memory transfer and can be configured with various configuration parameters, including the (physical) source, destination address and transfer size. Furthermore, a channel can be software synchronized, meaning the transfer is initiated by software, or hardware synchronized. The latter means that another hardware device on the chip uses the sDMA engine to do a memory transfer on its behalf. For the scope of the project, you can ignore the hardware synchronized mode.

The device distinguishes between two units: (a) The element which is the smallest unit of transfer (8, 16 or 32 bits) and (b) the frame which is a memory region containing a fixed number of elements. A single DMA transaction may work on multiple frames, therefore the SDMA device typically expects three configuration parameters (element size, elements per frame, number of frames). Furthermore, the device supports several addressing modes which define how the address of the next element is computed after the previous element has been copied. The simplest addressing modes (Constant addressing and Post-increment addressing) should be sufficient for your use-case.

The sDMA module has support for up-to four interrupt lines which can be used to send an interrupt to the CPU. Every channel can be assigned individually to one of the four interrupt lines. Your driver must support interrupts and avoid polling. However, it is sufficient to just assign all channels to one interrupt line. The device can be configured to send an interrupt at various stages during a transfer. In your case, you will most likely be interested in the "End of Block" notification which is sent once the transaction of a channel is complete. For debugging purposes, it is also a good idea to enable the various interrupts that signal an error condition, encountered by the device while executing your transactions.

2.1 Writing a device driver

Writing a device driver can be a daunting task for two reasons: There is a lot of documentation that has to be processed and understood. Sometimes the documentation may be incomplete or inaccurate. Secondly, if something is not working, it can be very difficult to debug the problem. Therefore, you have been warned that this project will require you to do precisely what the specification says and in addition, requires you to be extra defensive when programming to safe-guard against any bugs. On the plus side, programming devices can be fun (sometimes). The following sections try to guide you through what is necessary for writing a sDMA driver. However, they are by no means complete tutorials. In addition, you will have to get familiar with various sections in the TRM in order to produce a working driver for your device.

2.1.1 Access to the device registers

Your first goal is to gain access to the various device registers that are described in Section 17.6 of the TRM. The simplest option is to just request (from your OS) a capability for the memory region where the sDMA device registers are located. The TRM Section 17.6.1 tells us that the memory region we are interested in is a single 4 KiB page at address 0x4A056000.

In order to request the capability and map it in your address space, the driver can use the function `map_device_register` provided in the driverkit library (`lib/driverkit`, make sure to add the library to your Hakefile):

Mapping the device memory

```
errval_t err;
lvaddr_t dev_base;
```

```
// OMAP44XX_* macros are in include/omap4xx_map.h
err = map_device_register(OMAP44XX_MAP_L4_CFG_SDMA,
                          OMAP44XX_MAP_L4_CFG_SDMA_SIZE, &dev_base);
if (err_is_fail(err)) {
    USER_PANIC_ERR(err, "Failed to map sDMA registers");
}
```

If you inspect the code of that function, you will see that it uses the `aos_rpc_get_device_cap` call to request a device capability from your OS. This call is not implemented and you will have to implement it yourself (note: in case one person in your group is doing the file-system project you can share the effort):

include/aos/aos_rpc.h

```
errval_t aos_rpc_get_device_cap(struct aos_rpc *chan, lpaddr_t paddr,
                                size_t bytes, struct capref *frame);
```

2.1.2 Reading and writing device registers

While reading the "sDMA Registers Section" (TRM, 17.6.2) you will find that writing the device configuration will require a lot of bit manipulation to write to registers and set the appropriate values. While this can be done manually, it tends to be error-prone and tedious work. Your OS uses a domain-specific language (DSL) called Mackerel to program such registers more conveniently. We have already given you a Mackerel file for the sDMA device (in `devices/omap/omap44xx_sdma.dev`) which contains a complete machine-readable description of the available sDMA registers. The Mackerel DSL compiler can generate C functions for you to access the registers more conveniently. In order to use the Mackerel file, make sure to declare it in the Hakefile of your sDMA driver (`mackerelDevices = [ "omap/omap44xx_sdma" ]`).

Afterwards, you can include the generated Mackerel header file in your C code and use it to access the sDMA registers. To verify whether everything worked so far, you can use the following snippets to read the revision ID of your device (which should return 0x10900).

```
#include <dev/omap/omap44xx_sdma_dev.h>
void check_revision(mackerel_addr_t dev_base) {
    omap44xx_sdma_t devsdma;
    // Initialize 'devsdma' by handing it the virtual address of the device frame
    omap44xx_sdma_initialize(&devsdma, dev_base);
    // Now you can read the dma4_revision register like this:
    assert(omap44xx_sdma_dma4_revision_rd(&devsdma) == 0x10900);
}
```

Of course, you are not required to use Mackerel for writing your driver but we highly recommend it. For more information on how to use Mackerel you can read the Barrelfish Technote No. 2 [1]. In addition, note that the Mackerel file is really just a (generated) header file called `omap44xx_sdma_dev.h` and is located in your build folder (after the build is complete), you can inspect it to find the function names etc. by searching through the file.

2.1.3 Registering for Interrupts

You will also have to register your application with the kernel to receive interrupts from the sDMA device. The functionality is already implemented in the `inthandler_setup_arm` function, located in

[1] http://www.barrelfish.org/publications/TN-002-Mackerel.pdf

the `lib/aos/inthandler.c` source file. The function expects a handler function that is called when receiving an interrupt as well as the interrupt vector (source) you are interested in. On the OMAP4460, the interrupt vectors are hard-coded. The line 0 interrupt of the sDMA device has the number 44 (line 1 has number 45 etc.).

Register for Interrupts

```
void irq_handler(void* arg) {
    printf("Got sDMA interrupt!\n");
}

// Taken from OMAP TRM, Table 18-2
#define SDMA_IRQ_LINE_0 (32+12)

void enable_interrupt() {
    err = inthandler_setup_arm(irq_handler, NULL, SDMA_IRQ_LINE_0);
}
```

Note that in order for this function to work, your domain needs to own the IRQ capability. You can pass it in the initial CSpace when you create your process, or retrieve it using an RPC. The init process already has the IRQ capability (in the task CNode in slot `TASKCN_SLOT_IRQ`). Once you add this capability to your drivers CSpace as well, the function `inthandler_setup_arm` will be able to successfully register the for the interrupt.

Behind the scenes, `inthandler_setup_arm` will make a system call in the CPU driver, which in turn tells your CPU to accept interrupts from the sDMA device. If in the future, the sDMA device ever signals an interrupt to the CPU, the CPU driver will forward it to your application using the existing up-call mechanism. This implies that in order to receive any interrupts, your driver eventually has to listen for events on the default waitset (the library OS then ensures that the `irq_handler` callback is called as part of the event handler routine):

```
while(1) {
    event_dispatch(get_default_waitset());
}
```

2.1.4 Initializing the device

Typically, the first task of any device driver is to initialize the device by powering it on and putting it in a well-known initial state. Since the sDMA device on the Pandaboard is already powered on by default, initializing it will involve mostly configuring which interrupt should be triggered for a channel (by programming the CICR register, see Table 17-48). A more detailed description of what is required for the device setup is given in Section 17.5.1 “Setup Configuration” of the TRM. Your first task is to make sure you properly initialize the device according to the specification.

Note that several of the sDMA device registers, like the `DMA4_CICR` register exist for every channel. Since we have a total of 32 channels, this means that the sDMA device typically exposes 32 such registers (aligned and in consecutive order in memory). Mackerel has a type for declaring arrays of registers called regarray. This allows you to conveniently program the `DMA4_CICR` register of any given channel (as is done in the following snippet for channel number one):

Enabling some interrupts on a channel

```
size_t channel = 1;
```

```
omap44xx_sdma_dma4_cicr_t cicr = omap44xx_sdma_dma4_cicr_rd(&devsdma, channel);
cicr = omap44xx_sdma_dma4_cicr_misaligned_err_ie_insert(cicr, 0x1);
cicr = omap44xx_sdma_dma4_cicr_supervisor_err_ie_insert(cicr, 0x1);
cicr = omap44xx_sdma_dma4_cicr_trans_err_ie_insert(cicr, 0x1);
cicr = omap44xx_sdma_dma4_cicr_block_ie_insert(cicr, 0x1);
omap44xx_sdma_dma4_cicr_wr(&devsdma, channel, cicr);
```

2.1.5 Copying Memory

After you have initialized your device you can now start to implement the memory copy operation for your device. The steps you have to implement for this are explained in detail in Section 17.5.2 ("Software-Triggered (Nonsynchronized) Transfer") of the TRM.

Some hints that may help you during the implementation:

- Use `invoke_frame_identify` to find the physical address of a memory region.
- You can use `omap44xx_sdma_PORT_PRIORITY_LOW` (the constant is defined in the mackerel file) for the `DMA4_CCR` read and write priority (the higher priorities are for peripheral to peripheral transfers).
- You can use `omap44xx_sdma_WRITE_MODE_LAST_NON_POSTED` (the constant is defined in the mackerel file) for the write mode in the `DMA4_CSDP`.
- If you receive an interrupt, you will have to manually mark it as handled by clearing the corresponding bit (using the `DMA4_IRQSTATUS_LINE` register) before the device will signal a new interrupt of the same type.
- It may be easiest to come up with some abstraction over channels as you will likely have to maintain some state about them (for example: is there a transfer in-progress etc.).

Make sure to test that your implementation works before you go to the next section: You should be able to copy a region of memory to another region within the driver domain itself and you should correctly receive interrupts once a transfer has been completed (and no errors are signaled).

EXTRA CHALLENGE

The sDMA device is not just able to perform memory to memory copy operations but can also be used to implement other operations like filling a memory region with a constant value or rotating a memory buffer by 90 degrees (for offloading image operations). As an additional challenge, you can implement sDMA functionality to support `memset` or `transpose` (for matrices) operations. Note that in order to receive additional points, you will also have to expose these features to applications and evaluate their performance as described in the next two sections.

3 Device Interface

In the last part of the implementation you will have to design a safe interface to expose your driver to applications. By safe, we mean that it should not be possible for applications to make use of the sDMA

engine to read from, or write to memory that they do not have the necessary permissions for! You can use your existing messaging infrastructure in order to communicate with the driver. While we do not require you to write an asynchronous interface, it is preferred as it allows you to do additional work in your application while copying data. Make sure that your driver can serve multiple requests concurrently (from several clients), this will involve programming multiple channels and keeping track of their current state.

4 Performance Evaluation

As a last step, you'll have to evaluate performance of your sDMA driver. Your evaluation should at least contain the maximum bandwidth you achieved and compare it to the performance you would get when using the CPU instead. In addition, you should also measure the latency overhead introduced from communicating with your driver.

You need to be able to explain your results, where the potential bottlenecks are in your implementation and how you could improve them.

We provide a very simple driver for the Cortex-A9 global timer (see Cortex-A9 MPCore TRM, p4-8ff). To use this driver you'll need to add the `omap_timer` library to the program (`addLibraries` in the `Hakefile`). Here's a very simple example:

```
#include <omap_timer/timer.h>

int main(void)
{
  // initialize the timer
  omap_timer_init();
  // enable the timer
  omap_timer_ctrl(true);
  uint64_t start = omap_timer_read();
  // do work
  uint64_t end = omap_timer_read();
  uint64_t time = end - start;
  // do analytics on elapsed time.
}
```

EXTRA CHALLENGE

There may several opportunities to improve the performance of your system. This can be done by analyzing current bottlenecks in your code and improving the affected functionality. Possible examples are your messaging system or the sDMA engine itself.

If you do any work to improve the performance of your system, make sure to document it through benchmarking. Also, explain to us what you have done in the submission and how it affected the performance. Bonus points will be awarded on a case by case basis depending on your analysis and the optimizations that you did.

5 Lab demonstration

You are expected to demonstrate the following functionality during the lab-session:

- Explain the workings of your sDMA driver and demonstrate that is able to handle requests from applications to copy memory.
- Explain the API that exposes the sDMA device to applications and the design decisions you took.
- Show any benchmark programs and measured data in form of plots or tables and be able to explain the results.

Once your code is demonstrated, you should submit it over Online submission system (accessible only from ETHZ network) on Friday 16.12.2016 midnight (23:59h).