Contents

Please note that this document only describes a changelog for `K4` along with a description of our reimplementation of `A0`. For the full kernel documentation, please refer to `kernel.pdf`.

New in K4

UART IRQs

The UART combined interrupts are supported in the kernel as events `52` for UART1 and `54` for UART2. UART interrupts are enabled in user space by writting to the UART's CTLR (control register). For ease of use, a the following struct is defined in `ts7200.h`:

```
union UARTCtrl {
    uint32_t raw;
    struct {
        bool uart_enable : 1;
        bool sir_enable : 1;
        bool sir_low_power : 1;
        bool enable_int_modem : 1;
        bool enable_int_rx : 1;
        bool enable_int_tx : 1;
        bool enable_int_rx_timeout : 1;
        bool loopback_enable : 1;
    } _;
};
```

User tasks may manipulate the control register by creating a `UARTCtrl` struct, setting `.raw` to the register's data, modifying the bitfields in `._`, and writing `.raw` back to the register. By default, only the `uart_enable` bit is set. Interrupts are enabled via the `enable_int_*` bits.

When UART interrupts are enabled, `AwaitEvent(52|54)` will block the calling task until the combined UART interrupt is raised. These calls to `AwaitEvent` will return the UART's INTR (interrupt register) as an `int`. Again we define a struct for ease of use:

```
union UARTIntIDIntClr {
    uint32_t raw;
    struct {
        bool modem : 1;
        bool rx : 1;
        bool tx : 1;
        bool rx_timeout : 1;
    } _;
};
```

The fields in `._` correspond to each of the possible combined interrupts. Whenever an interrupt is raised, the kernel will automatically disable that interrupt on the UART's control register to prevent unwanted reentry. Therefore, is is up the user task to re-enable each interrupt as required.

UART Server

The UART server is implemented as a single server task for both UARTs, and one notifier task for each UART. The notifier tasks simply `AwaitEvent(<irq_no>)` in a loop, `Send()`ing the interrupt data to the UART server. The server follows the conventional pattern of a `Receive()` loop that never blocks. The exposed interface contains the required `Putc` and `Getc` functions, along with a few other helpful routines:

```
namespace Uart {
extern const char* SERVER_ID;
void Server();
int Getc(int tid, int channel);
int Putc(int tid, int channel, char c);

// Getn blocks until n bytes are received from the UART, writing the bytes to
// buf.
int Getn(int tid, int channel, size_t n, char* buf);

// Putstr and Printf can atomically write up to 4096 bytes to the UART in a
// single call.
int Putstr(int tid, int channel, const char* msg);
int Printf(int tid, int channel, const char* format, ...)
```

```
    __attribute__((format(printf, 3, 4)));

// Drain empties the RX FIFO for the given channel.
void Drain(int tid, int channel);

// Getline reads a line of input from the uart into `line`, treating the
// backspace and enter keys appropriately.
void Getline(int tid, int channel, char* line, size_t len);
}  // namespace Uart
```

Notably, `Putstr` and `Printf` atomically write an entire string to the UART (up to a maximum length of 4096 bytes), and `Getn` blocks until `n` bytes are received from the UART. Each of the output routines are written in terms of `Putstr`, and similarily, each of the input routinges are written in terms of `Getn`.

The UART server has two key pieces of state: output buffers (for TX) and blocked tasks (for RX). The output buffers are a 4096-byte ring buffer per UART.

When handling a `Putstr`, data is written to the UART until the FIFO fills up, at which point further data is pushed onto the output ring buffer and the `tx` interrupt is enabled. Once the tx interrupt is received, data is pulled off the ring buffer and written to the UART's data register until either the buffer is emptied or the FIFO is filled (in which case `tx` interrupts will be enabled again). `Putstr` replies to the sender immediately, ensuring only that the string has been buffered, not flushed.

`Getn` is handled by putting storing a "blocked task" per UART. When a task calls `Getn` (or `Getc`, which is just a special case of `Getn`), a blocked task entry is created for the UART, which contains a buffer to store bytes, along with a counter of how many bytes have been received and the task's Tid. The UART server then reads bytes off the UART's RX FIFO until either the FIFO is empty (in which case the `rx` and `rx_timeout` interrupts are enabled) or the desired `n` bytes have been received (in which case a reply is sent and the "blocked task" is cleared).

Only one reading task per UART is supported, as concurrent readers leads to confusing behaviour - should received bytes be sent to both tasks concurrently? In a round robin? Or in sequential order? The semantics of multi-reader is confusing, and likely unnecessary for any applications that will be built on the kernel. So "bocked tasks" are implemented as simple `std::optional`s per UART, and concurrent reads to the same UART will panic.

`Drain` is a notable addition to the specification - it clears any bytes that may be sitting in the RX FIFO and any blocked task's buffer. This is useful when we know that certain bytes are expected to be received after a certain event. Calling `Drain` ensures that all received bytes must have been received after the `Drain` call, which is useful when doing request-response style calls over the UART.

A note about CTS

Due to time constraints stemming from social and familial obligations during reading week, we were unable to get the CTS functionality on UART1 working correctly. We currently delay 250ms between Marklin commands.

We will continue to work on implementing proper CTS functionality, making sure that it is working correctly by the next deliverable.

Re-Execute Kernel without resetting TS-7200

Part of our feedback for K3 was that our kernel could not be re-executed without first reseting the TS-7200. This was a silly oversight on our part, as while we had all the infrastructure in place to support this (i.e: a shutdown routine that disabled and asserted interrupts), we had neglected to clear the Timer2 interrupt when shutting down.

Our kernel is now fully re-executable, and should work without having to reset the TS-7200 between runs.

A Proper Low-Power Mode Idle Task

As mentioned in our K3 docs, we knew full-well that our empty-looping idle task wasn't the best approach. As such, we've gone back and implemented a proper low-power mode idle "task" directly in the kernel.

As it turns out, this was surprisingly simple to implement:

During kernel initialization, we enable the Halt/Standby magic addresses in the System Controller. This is done in two simple lines of code:

- `*(volatile uint32_t*)(SYSCON_SWLOCK) = 0xaa;`
- `*(volatile uint32_t*)(SYSCON_DEVICECFG) |= 1;`

With this setup out of the way, the entire system can be put into the low power Halt mode by reading from the System Controller's Halt register (i.e: `*(volatile uint32_t*)(SYSCON_HALT);`).

As it turns out, using the System Controller to halt the CPU has a very useful side effect: the CPU does *not* need to have it's IRQs enabled for the System Controller to wake it up! This means that we can leave the CPU in the regular IRQ-disabled Kernel Mode before entering the Halt state, saving us from having to do any mode-switching assembly shenanigans. When an IRQ eventually arrives, the System Controller simply "unhalts" the CPU, which resumes executing whatever instructions come after the `SYSCON_HALT` read. Critically, the CPU does *not* jump to the global IRQ handler, which would be quite dangerous, given that the IRQ handler is typically only entered when a user task is preempted.

Thus, our entire "idle task" logic is as simple as:

```
if (/* there is nothing to schedule */) {
    // read the current time before going idle
    idle_timer = *TIMER3_VAL;
    // go idle
    *(volatile uint32_t*)(SYSCON_HALT);
    // update idle time counter after waking up
    idle_time::counter += idle_timer - *TIMER3_VAL;
    // handle the IRQ that just woke the CPU up
    driver::handle_interrupt();
}
```

Splitting `kernel.cc` into multiple files

The kernel is no longer written as a `class` with a single instance, but instead a collection of namespaces with a few static state variables. This has allowed us to finally split up the massive `kernel.cc` file by slices of functionality, instead of keeping everything in one class, fighting with unnecessary encapsulation.

The namespaces are as follows:

- `kernel` defines the kernel's state (`tasks`, `current_task`, `ready_queue`, `event_queue`) and the `run()` function that executes the main scheduling loop.
- `kernel::driver` implements the functions required by `kernel::run()` (`activate_task`, `schedule`, `initialize`, `shutdown`) along with the `swi` and `irq` handlers.
- `kernel::handlers` implements the functions to handle each syscall (`Create`, `MyTid`, `Send`, etc).
- `kernel::perf` keeps the state to track our idle time measurment.
- `kernel::helpers` contains functions used accross multiple namespaces.

`kernel.h` defines the public interface of each namespace, but each namespace is implemented accross one or more separate files. Syscall handlers each get their own file (i.e. `src/kernel/handlers/send.cc`), as do the `irq` and `swi` handlers. Data structures such as `Tid` and `TaskDescriptor` are also separated.

`kernel.cc` is now substantially smaller and more navigable: it just defines the global state variables, `kernel::run()` and the `kernel::driver` routines used by `kernel::run()`. Everything else is separate.

FirstUserTask with Priority 0

Prior to K4, the FirstUserTask would always be spawned with a priority of 4. While 4 is a certainty a fun and perfectly respectable number, it's not *really* the first number to spring that springs to mind when asking the question "What should the FirstUserTask priority be?" As such, starting from K4, we've given the FirstUserTask a much saner priority of 0.

Note: Although it wasn't strictly required, we did go back and fix up some of our old assignments, as some of them relied on the FirstUserTask having a priority of 4.

Spawning the Nameserver Alongside the FirstUserTask

The Nameserver is quite a *special* user task, as it's interface implicitly relies on the Nameserver task having a *fixed* Tid. In our kernel, that fixed Tid happens to be 1.

In previous iterations of our kernel, we would ensure that the Nameserver task had a Tid of 1 by having a implicit requirement that that FirstUserTask must spawn the Nameserver task *before* spawning any other tasks.

While this approach *worked*, it wasn't very robust, as we would often inadvertently spawn a task prior to the Nameserver, and spend a minute or two wondering why the Nameserver calls were failing. To avoid these sorts of mistakes, and to cut down on some boilerplate, we've opted to spawn the Nameserver *alongside* the FirstUserTask as part of the kernel setup, thereby ensuring that the Nameserver task both exists, *and* had the correct Tid.

New Syscalls

As part of our cleanup efforts, we've introduced two new kernel syscalls:

`Perf`

In K3, we had a somewhat... unpolished approach to outputting idle time measurements. Namely, we would simply `bwprintf` the idle percentage each time the idle task was preempted.

This approach had two serious issues:

- It used busy-wait IO, which seriously hurt system performance.
- It would output idle time diagnostics to the same place on screen, without any regard for the UI of the running application.

To tackle both of these issues, we decided that we needed a mechanism whereby user tasks could query the kernel for system performance information. As such, we decided to introduce a new syscall to our kernel: `Perf`.

`Perf` uses syscall number `9` and has the following C signature: `void Perf(struct perf_t*)`, where `struct perf_t` contains all sorts of useful system performance information. At the moment, `struct perf_t` only contains a single field: `uint32_t idle_time_pct`, which as the name implies, is the current idle time percentage of the system.

With this new Perf syscall, individual userland programs to are able to query and display system performance metrics whenever and however they like!

One final note: Prior to K4, our idle time measurement would be show the percentage idle time *from startup*. While this provided some indication of system performance, it made it difficult to spot any sudden changes in system performance. To get a more granular notion of system performance, we've opted to have each call to `Perf` reset the idle time measurement, providing a form of "windowed" idle time measurement, where the window size depends on the frequency at which `Perf` is called.

`Shutdown`

When working on userland code, we make liberal use of `assert` and `panic(const char* fmt, ...)` to notify us if any of our invariants are broken. Prior to K3, these macros used the regular old `Exit` syscall, which would terminate an individual task if something went wrong. This was *okay*, but it typically resulted in us having to manually restart the TS-7200, as our program would most likely hang once an assertion / panic fired.

At the same time, we found that as our programs became more complex, and spawned more and more long-lived tasks, it became quite difficult to trigger a kernel exit (i.e: by exiting all running tasks).

A clean solution to both these problems was to introduce a new syscall: `Shutdown`.

`Shutdown` uses syscall number `10` and has the following C signature: `void Shutdown(void) __attribute__((noreturn))`.

The kernel's Shutdown handler routine is incredibly simple, simply invoking `kexit`, which triggers the kernel's shutdown sequence, and eventually returns execution back to the Redboot.

We've swapped our `assert` and `panic` routines to use `Shutdown` instead of `Exit`, and have used `Shutdown` directly when implementing the `q` command in our k4 user task.

A0 Redux - Now with 100% more Kernel!

Overview

It's been almost two months since A0, and we're now half way through the journey of self-discovery and sleep-deprivation that is CS 452. The time has finally come to be reunited with the venerable Märklin 6051 train set, and get those trains chuggin' along.

Unlike A0 though, this time we've got an actual OS to build on top of!

Using our newfound *OS and embedded programming skillz*, we've (re)implemented a simple program to control various aspects of the model trainset, including but not limited to: setting the speed of individual trains, reversing individual trains, and setting various switches on the track to be either straight or curved. This

program simultaneously displays the time since startup, and a list of recently triggered track sensors.

Operating Instruction

After a brief initialization procedure, the display should look something similar to the following:

```
[0:23:3]                                                    [Idle Time 97%]

Sensors: D11

Train |   1 |  24 |  58 |  74 |  78 |  79 |
Speed |   0 |   0 |   0 |   0 |   0 |   0 |

Switches
|  1 C|  2 C|  3 C|  4 C|  5 C|  6 C|  7 C|  8 C|
|  9 C| 10 S| 11 C| 12 C| 13 S| 14 C| 15 C| 16 S|
| 17 S| 18 C|153 C|154 C|155 C|156 C|




Initializing Track...
Ready.
>
```

The display consists of the following components:

- A live updating timer, displaying the amount of time the program has been running for
- A readout of the system's reported idle time
- A list of (valid) trains with their current speeds
- A list of (valid) switches with their current state (**C**urved or **S**traight)
- A readout of the track's recently triggered sensors
- A command prompt
- (**extra**) A feedback message regarding the last run / currently running command

The program accepts all the commands specified in the A0 assignment outline:

Command	Description
`tr <train number> <train speed>`	Sets the speed of a given train (from 1 - 14, with 0 to stop)
`rv <train number>`	Reverses the direction of a given train
`sw <switch number> <switch direction>`	Sets a given track switch to be either (s)traight or (c)urved

Command	Description
`q`	Quit the program and return back to RedBoot

In addition to the required commands, this program also implements a few additional "extra" commands:

Command	Description
`l <train number>`	Toggles the lights of a given train
`s`	Sends the "Emergency Stop" (i.e: STOP) signal to the train set
`g`	Sends the "Release" (i.e: GO) signal to the train set

As a quality-of-life feature, the program remembers the last-entered command, which is re-run if the return key is hit without entering a new command. This is most useful in conjunction with the `l` command, as it lets you blink a train's lights on-and-off! How Fun!

Architecture

Unlike A0, we actually have access to an OS for this assignment! This has made it a lot easier to implement much of the program's functionality, most notably in relation to task scheduling and timing.

Broadly speaking, for each UI element on screen (e.g: command prompt, train speed, sensors, etc...), there is a corresponding Task which performs any data processing and/or IO required to render the information to the screen.

In total, we use 6 application-specific long-running tasks to run this application:

- PerfTask
  - Uses the `Perf` syscall to render the idle time in the top right corner of the screen
- TimerTask
  - Uses the Clock Server to display a application uptime timer in the top right corner of the screen
- CmdTask
  - Renders the Command prompt (`> ...`) and command feedback UI
  - Reads lines from UART2, parses them as commands, and relays requests to the MarklinCommandTask
  - Train commands update the train UI, while the switch command updates the switch UI
- MarklinCommandTask
  - Encapsulates the nitty-gritty details of communicating with the Marklin controller

  - Accepts high-level requests (i.e: set train X to have speed Y), and translates them into specific byte-sequences to be output via UART1
- SensorPollerTask + SensorReporterTask
  - SensorPollerTask sends a Sensor Query request to the MarklinCommandTask every 300ms
  - SensorReporterTask reads incoming sensor data from UART1, displaying the data via the sensors display

Known Bugs

- User input doesn't work while train commands are being executed.
  - This is most noticeable with the `rv` command, as is implemented inline in the cmd task.
  - One potential fix would be to have a separate "ReverseManager" task which would accept reverse requests from the cmd task.