CS 452 Assignment 4

Victor Scurtu (20678028) and Alex Danila (20516806)

February 24, 2020

Operating Notes

The executable file is found in the student environment under:

```
/u/cs452/tftp/ARM/vscurtu/k4/kernel
```

The code can be found in the following repository:

```
https://git.uwaterloo.ca/vscurtu/cs452_kernel
```

To build, use the following commands at the root of the repository on a student environment computer:

```
make
```

To run the precompiled kernel, use the following command in redboot:

```
load ARM/vscurtu/k4/kernel; go
```

After entering the above command, the program will display state information about the kernel and track and begins an initialization procedure to set the track to a known state. All commands sent during the initialization period are ignored and an error message is displayed accordingly. This is because the track is not yet in a known state while initializing. In order to enter a command, begin typing and press ENTER once you've formulated your command.

The commands are exactly as outlined in the assignment 0 description. For a reminder they are:

- tr <train number> <train speed> – Set any train in motion at the desired speed (0 for stop).
- rv <train number> – Reverse a trains direction.
- sw <switch number> <switch direction> – Throw the given switch to straight (S) or curved (C). Note that switch_direction must be a capital S or C.
- q – halt the system and return to RedBoot. Pressing ESC is a quick alternative to this command.

Ensure that the console window is wide enough to display all the information comfortably. If things look messy, try quitting, widening the window, then rerunning. No features were left unimplemented. Minimal error checking is done on the commands entered that will check

whether the command prefix is valid, and if the arguments are a valid integer or an expected string. It is largely assumed that the integer values of arguments provided such as train and switch ids are valid.

Table of Contents

Initialization

The kernel starts at main and begins by initializing kernel memory by calling `kinit`. All initialization is done within `kinit`. It first configures COM2 to 115200 baud, FIFO disabled. After this, it calls `init_task_list` and `init_pqueue` to initialize the task and `pqueue` modules as follows:

Task List Initialization

`init_task_list`: Initializes the values of an array of `task` structures by setting each structure's `t_id` to its array index, and its `p_id` and `state` to `TASK_INVALID`. The meaning of the fields and states is described in the Tasks section. The size of this array is defined in `constants.h` and can be reconfigured easily. The current running task is initialized as -1.

Priority Queue Initialization

`init_pqueue`: Initializes a bank of `pqueue` nodes to all point to their following neighbor. These are all considered free nodes at initialization and as such a pointer to the list of `free` nodes is initialized to the first node in this array. The priority queues are stored as pointers to heads and tails in this node bank. Their heads and tails are initialized to null pointers and as such we begin with zero tasks. This means the kernel is not a real task.

After these two initialization procedures, the kernel initializes the interrupt vector table to point to our enter kernel function for software interrupts and an unhandled exception handler for all other interrupts for the purpose of debugging.

Caching is enabled via a call into an assembly routine, enable_cache. The routine reads r1 of coprocessor 15 into r0. This is the control register of the system control coprocessor. It uses orr to set the required bits to enable L2 cache, L1I cache, L1D cache, and the MMU - bits 26, 12, 2 and 0 respectively - and writes that value back from r0 to CP15's r1.

After initializing the task list and priority queue in kinit(), the message queue is initialized by calling the init_message_queue function. This works similarly to the init_pqueue function, but works on the global message queue instead. It initializes a bank of id nodes to all point to their following neighbor. These are all considered free nodes at initialization and as such a pointer to the list of "free" nodes is initialized to the first node in this array. The message queues are stored as pointers to heads and tails in this node bank. Their heads and tails are initialized to null pointers and as such we begin with zero messages in any tasks message queue. There is one message queue for every task.

During `kinit` the scrollback region is set to start from line 2 onwards using VT-100 escape sequences and `IDLE: 0%` is printed on the first line. This reserves the first line for the idle task to print the idle time percentage.

Also during `kinit`, the kernel writes `0xAA` to `SYS_SW_LOCK_ADDR` and enables bit 0 of `SW_HALT_ENABLE_ADDR` to enable transitions into Halt mode, as described in the EP93xx guide.

The kernel now creates both `umain` and `idle_task`. The idle task is created at the lowest priority, with the expectation that no other task will be given that priority, so that it will only run when all other tasks are blocked.

Printing

Logging Macros

A set of macros are defined in `logging.h` for various verbosity levels - debug, log, warn, error, fatal; from most to least verbose - which also print the line number and filename from which they were called. A call to fatal if enabled will cause a kernel panic. An assert is also defined, which behaves similarly to a fatal message; it prints the failed assertion and its location and then panics the kernel. Calling `panic` prints a message and returns to redboot using an experimentally determined hardcoded address. These are considered incompatible with the interrupt based IO in K4 and are not used except for debugging.

Printing without BWIO

The functions `format_string` and `_format_string` have been added to allow formatting strings and placing them into a buffer rather than printing them. They return non-zero if there is not enough room in the buffer provided for the formatted string which has been truncated to fit.

`print.h` has been added to provide convenience functions for printing using interrupt-mediated I/O. `Print` does the same thing as `print` from `logging.h`, printing a formatted string, but does so using a UART server's `Putc` and `Getc` rather than bwputc and bwgetc under the hood. `UPrint` does the same thing as `Print` but without formatting, which means it essentially just calls `Putc` in a loop. This is useful as it doesn't treat '%' characters specially for strings that have already been formatted. `MoveCursor` and `ClearScreen` are wrappers for terminal control commands that print using a UART server.

Tasks

The maximum number of tasks is set to 128. It is assumed that no individual task nor the kernel will consume more than 128KiB of memory. Therefore user task stack bases are allocated every 128KiB starting at 0x0100000 growing up towards a maximum address 0x1FE0000. This is in reverse of the direction of growth for individual stacks. Allocating user stack bases like this minimizes the probability that the tasks and kernel collide if the kernel stack grows too far.

The starting and maximum addresses, as well as the size and number of tasks, are defined in constants.h and can be easily changed if necessary. As configured, the addresses allow for about 31MiB of workable memory, but the system only has enough task descriptors for about 16MiB of tasks. It should be noted that we cannot use the full 31MiB of workable memory as the kernel stack grows from the end of it. Regardless, since we only use 16MiB of memory for the task descriptors, we keep well away from the kernel stack.

User tasks are tracked using the task module, which contains a task struct and several functions for interfacing with the array of tasks. The task struct has fields for the task ID (t_id), the parent ID, (`p_id`), priority, state, exit code, the stack pointer, and stack base, which point to the top and bottom of the stack respectively. The array is declared in `task.c` to protect it from direct access. This ensures that the array cannot be corrupted by writing invalid data, since the interface functions validate data using asserts before saving.

Tasks can have either `TASK_INVALID`, `TASK_READY`, `TASK_RUNNING`, or `TASK_ZOMBIE` for state. `TASK_INVALID` is used to designate unallocated task descriptors. `TASK_READY` is for tasks that can be scheduled to run. `TASK_RUNNING` is for the current running task. `TASK_ZOMBIE` is used by the `Exit` syscall; it is set for tasks that have exited but have not been deallocated.

There is a series of getters and setters for the `state`, `exit_code`, and `stack_pointer` task fields. There is also `get_task_by_id` which returns a copy of the task struct for a given task ID. The `t_id`, `p_id`, `priority`, and `stack_base` fields do not have getters or setters as they should not be set after a task is created.

All tasks have their `t_id` set when `init_task_list` is called. Individual tasks have their `p_id`, `priority`, and `stack_base` set when `allocate_task` is called. Available task descriptors are found by searching through the `task_list` until a descriptor with the state `TASK_INVALID` is found. This allows for reuse of tasks descriptors after they have been

destroyed. Importantly, `allocate_task` does not initialize the task stack. It is expected that the caller, `kcreate`, will initialize the stack after a newly allocated task is returned to it.

Additionally, the task descriptor module also has several record keeping and validation functions. It keeps track of the currently running task in a variable which is set to -1 on initialization. This is treated as a "virtual" ID for the kernel. Setting the running task can be done by either supplying a task ID to `set_running_task` or calling `set_task_state` with an ID and the `TASK_RUNNING` state as arguments, which then calls `set_running_task`. If `set_task_state` is called with the ID of the current running task and a state other than `TASK_RUNNING`, the current running task is set to -1.

In order to ensure that a task is not left in the `TASK_RUNNING` state when a context switch occurs, `set_running_task` asserts that the current task is set to -1 before attempting to set it to anything else. In order to do a context switch, `set_task_state` must first be called on the running task to set it in a state other than `TASK_RUNNING`, before `set_running_task` can be called on another task ID.

If a task state is set to `TASK_ZOMBIE`, `set_task_sate` searches through the task array for that task's children. Any children have their `p_id` set to `PARENT_ZOMBIE`.

For validation of parameters, `__is_defined_task_state` and `is_valid_task` check if the task state is one of the defined states, and check if the task state is not `TASK_INVALID` respectively. These are used in asserts to ensure logical consistency.

Lastly, a `free_task` function is defined for later use with the `Destroy` syscall. It sets the state of the task with the provided ID to `TASK_INVALID` so `__get_next_available_id` can find it.

Scheduling

Scheduling is done using a priority queue structure, the implementation of which is outlined in the Structures section of this report. Our priority queue structure contains 8 priority levels (0-7), each represented by a FIFO queue. 0 represents the "highest" priority and vice versa. Aside from the initialization function, described in the Structures section of this report, the `pqueue` structure provides the functions `push_task`, `pop_task`, and `peek_task` in its interface. `push_task` pushes a task onto the back of the queue corresponding to the priority level of the task as defined inside its task structure. `pop_task` removes the front task from the highest priority level queue and returns it. `peek_task` returns the front task on the highest priority level queue, but does not remove it. Both `pop_task` and `peek_task` return -1 if all priority

queues are empty. A current running task variable is stored in task.c that is accessed with the `set_running_task` and `get_running_task` functions.

Reschedules are only triggered by tasks since there is no pre-emption. A reschedule is always triggered every time a task calls a kernel primitive. For all kernel primitives except `Exit`, this causes the current running task to be pushed to the back of the priority queue and the next task to be popped from the queue. For `Exit`, it is simply removed from the front of its priority queue.

In all cases a new task will now be at the front of the highest priority queue (if there are 2 or more tasks). This is due to exposing a new element in the same queue, or making a new queue the highest priority non empty queue. This new task will be the next running task by calling pop_task after the previous manipulations have been done.

Message Passing

Message sending is built using a queue for every task where senders are processed on a first come first serve basis. Tasks that are waiting to send to another task are put on that task's message queue.

Copying is done inside the kernel using `kcopymessage` and `kcopyreply` to determine parameters for `copy`, which does the actual copying, and to set return values and task states.

Additional details are provided in the Kernel Primitives section under Send, Receive, and Reply.

Name Server

The name server is a user task that provides name resolution for other tasks. The interface and implementation are defined in `name_server.h` and `name_server.c` respectively. Tasks communicate with the server using the `RegisterAs` and `WhoIs` functions. These are wrappers for message sends to the `name_server` which combine the name provided with a prefix that encodes the operation type (`RegisterAs` or `WhoIs`) before sending it to the server. They also return -1 if the name server task id is invalid. The name server is created within the `umain` task in `user.c`, the entry point for all user tasks.

As discussed in the assignment, the task id of the name server must be known to the `RegisterAs` and `WhoIs` routines, which are executed by other tasks. This task id is stored in the `name_server_id` variable in `name_server.h`. It is populated at the beginning of the `umain` task after the `name_server` task is created.

The _register_as and _who_is functions are the implementations for RegisterAs and WhoIs respectively and are called in the name_server task after a new message is received and parsed.

The name server is implemented using an array of strings, indexed by the id of each task. Every time the name for a task is needed, the array is indexed using the task id. Every time a task id is needed for a name, the entire list of names is searched until a matching string is found.

The name_server function is the task's entry point. It initializes the list of names to all empty strings and then begins an infinite loop which works as follows. At the start of the loop a Receive call is made for a message. The message is then parsed, using the value of the first character to pick what the operation is to be done on the name, and the remainder of the string as the name. The first character of the message will have been set appropriately by one of the two wrappers. The corresponding _who_is or _register_as function is called. Once these return, their result is sent back to the task that made the request using a Reply call. This marks the end of the iteration.

_who_is works by iterating through the list of names, comparing each to the name provided, then returning the index of the name that matched which represents the task id associated with that name. If the name is not in the list, indicating that the name has not been registered in the name server, TASK_DNE is returned.

_register_as works by taking first accepting a name that is guaranteed to not be too long. The max size of a name is defined as MAX_NAME_LENGTH. The name provided will not be too long since it has been automatically truncated by the receive function only accepting a message of size MAX_NAME_LENGTH. This has to be done since the array of strings is a fixed size two-dimensional array. Inside _register_as, _who_is is first called on the name to find out if it is already in the name server. If it is, the name for that task is set to an empty string. The name is then copied into the name array at the requesting task id's index and 0 is returned to indicate success.

Since we do not reuse task descriptors currently, tasks that exit do not deregister from the name server. Since the RPS server uses a single signed char in the reply message as the returned task ID and uses the negative range of the char for errors, it's only able to return 0-127 for valid IDs. This means MAX_TASKS_ALLOWED must be at most 128 and there is an assertion that checks this.

String Utilities

In order to work with the strings in the nameserver, a few string utility functions were added in `string_utility.h` and `string_utility.c`. These closely mirror their counterparts in the standard libraries `string.h`.

We were unable to coerce the compiler to find and include string.h, however the names strcpy, strlen, and strcmp could not be used since the compiler specifically checks for those function names (even if string.h is not included) and throws a redefinition error. As a work around, the standard library function names prepended with a '_' were used.

Kernel Primitives

The following is the list of kernel primitives required for this assignment and their implementations. Every kernel primitive directly has a syscall associated with it. The following implementations occur in a syscall handler function handle_swi and occur while in kernel.

Create

Calls `kcreate` with the arguments in `r1` and `r2` which represent the priority and function pointer of the task respectively. `kcreate` works by first calling `get_running_task` which accesses the global currently running task ID, and `allocate_task` which will allocate a new task structure in the `task_list` array. It then obtains the actual task structure using the ID returned by `allocate_task` and initializes a trap frame using the `stack_base` pointer with an offset that provides enough space to store all the main registers and the `cspr`. This initial trap frame is needed since `enter_user` will attempt to pop the saved registers from the stack as though they are already there. Using the frame, important registers `r13`, `r14`, `r15`, and `cspr` are initialized on the task's stack to be restored when the task is entered. This is the `stack_base` address in `r13`, the `exit_handler` in `r14`, which calls `Exit` when a task returns, the function pointer in `r15`, designating the entry point, and the user processor mode in `cspr`. The `stack_pointer` is then set to point to the top of the frame and the task is added to the priority queue with `push_task`, placing it on the appropriate internal queue for its priority level. Finally, `kcreate` prints the created task ID and returns the ID to the caller, `handle_swi`. If the priority provided was not a valid value or `allocate_task` ran out of task descriptors, -1 or -2 is returned respectively.

MyTid

Calls `get_running_task` which accesses the global currently running task id.

MyParentTid

Gets the parent from the task structure associated with the current running task obtained the same as in `MyTid`.

Yield

Does nothing other than reschedule the task. As described before, this moves it to the back of its current priority queue.

Exit

Does nothing but set the state in the task structure to `TASK_ZOMBIE`. As described in Scheduling, the task is later popped off its priority queue by default.

Send

Checks if the receiver is in state `TASK_RECV_WAIT`. If it is, immediately copy the message, then put the sender into `TASK_RPLY_WAIT` and the receiver into `TASK_READY`, and put the receiver back into the schedule queue. Otherwise, put the sender on the receiver's wait queue in state `TASK_SEND_WAIT`.

Receive

Pops the first sender from the task's message queue. If there is none, blocks the task in `TASK_RECV_WAIT` until woken up by a send.

Reply

Checks that the receiving task is in `TASK_RPLY_WAIT`. If it is, copy the message immediately, unblock both tasks, and return both tasks to the schedule queue. Otherwise, return an error code.

AwaitEvent

Maintains an array of event wait spots. There is only one spot per event, so only a single task may wait on an event at a time. In the case of UART also enables relevant interrupts as it is not considered valid for UART interrupts to be enabled without a task waiting for the event. Also checks if UART CTS and returns immediately if they have, instead of enabling the interrupt and blocking the task. AwaitEvent is described further in the AwaitEvent implementation section.

Shutdown

Causes the kernel to break out of it's main loop, cleanup, and return to redboot. Cleanup is handled by the `kcleanup` function in `kernel.c`. This function clears the VIC, disables interrupts that were enabled, and clears any RX interrupts by reading bytes.

Context Switch

Originally, switching context was done through two assembly defined functions, `enter_kernel` and `enter_user`. These functions have been modified to accommodate hardware interrupts.

Enter Kernel

In essence, the process of saving the user state and restoring the kernel state remains the same. However, to support hardware interrupts, `enter_kernel` has been split into two handlers, `hardware_enter_kernel` and `software_enter_kernel`, and `finish_enter_kernel`, which is branched to by both handlers.

Both handlers switch to system mode to retrieve the user's stack pointer. In addition, `hardware_enter_kernel` switches back to IRQ mode and subtracts 4 from the LR, while `software_enter_kernel` switches back to supervisor mode.

Finally, `finish_enter_kernel` handles saving the user's registers, loading the kernel's and returning to the main C-language loop where the interrupt or system call is handled and another task is scheduled and run.

A minor change was made to the order in which user registers are saved. The user's PC is now the last register saved so it is at the top of the stack. This enables the atomic restore of user mode CPSR and PC using `movs`, described in the `enter_user` section.

A deliberate decision was made to execute kernel code in IRQ mode after `hardware_enter_kernel` and in supervisor mode after `software_enter_kernel`. This is because the kernel uses the current processor mode to determine if it is handling a hardware interrupt or a system call.

Enter User

`enter_user` has also been split into several parts. When `enter_user` is called from the kernel, it saves the stack pointer into `r3`, checks the current processor mode, and branches to that mode's version of `enter_user`. Both `supervisor_mode_enter_user` and `irq_mode_enter_user` ensure that the stack pointer for supervisor mode and IRQ mode match by switching to their respective counterpart mode and copying the address from `r3` into the stack pointer. This is done so that entering the kernel can work correctly from either mode. Finally, both jump to `finish_enter_user`, which saves the kernel registers and restores the user registers from the stack.
Finally, `finish_enter_user` restores the user's saved PC into LR and the saved CPSR into SPSR, since they are now both at the top of the stack. After restoring the first 15 registers, the `movs pc, lr` instruction atomically restores the user's PC and switches the processor to user mode by loading SPSR into CPSR.

Software Interrupts

The software interrupt handler function, `handle_swi`, uses a trap frame to inspect the stack of the calling task and extract the syscall type and arguments. It also uses the trap frame to set the return value of the call, if any, in `r0` on the caller's stack. The `Frame` structure used for trap frames is discussed further in the Structures section of this report

Hardware Interrupts

`interrupt.h` and `interrupt.c` provide an interface for enabling and disabling interrupt sources in the VIC and for handling interrupts when they are generated. Currently only `TC1UI`, `TC2UI`, and `TC3UI` are available to be enabled.

Initialization

Hardware interrupts are enabled in the VIC during the `kinit` phase using the `enable_interrupt` interface function which enables the appropriate bits in the VIC. For the implementation of `AwaitEvent`, Timer 1 is enabled in 2kHz rundown mode, with an initial value of 20. This will cause it to underflow and generate an interrupt after 10 milliseconds.

Handling

After returning from `finish_enter_kernel`, the kernel updates the stack pointer in the structure of the running task, as it did in K2, and then checks the current processor mode. If it is

in supervisor mode, it is assumed that a system call occurred and `handle_swi` is called as usual. Otherwise, if the processor is in IRQ mode, `handle_interrupt` is called.

Currently, `handle_interrupt` assumes that only one interrupt will be generated at once. This is a valid assumption since interrupts will only be enabled on one clock for this assignment, with no other interrupt sources enabled.

Before handling the interrupt that occured, `handle_interrupt` first sets the current running task to `TASK_READY` and pushes it onto the schedule queue.

`handle_interrupt` checks both VICs for the interrupt source. Once found, the interrupt is handled by waking the task that was waiting on that event, and clearing the interrupt source. Here, waking a task means setting its return value, setting its state to `TASK_READY` and pushing it back onto the schedule queue.

AwaitEvent Implementation

The implementation of `AwaitEvent` is provided in `await.c` and `await.h`. The functioning is divided into two parts: the system call side and the interrupt side.

Implementation

Inside `await.c` is an array that holds the task ID of the singular task waiting on a particular event. This implementation assumes that only a server listener will ever wait on an event.

The array is initialized to hold all `TASK_INVALID` IDs by a call to `init_event_wait_tid_list` during `kinit`. The array indices correspond to the event id, which are defined, along with the number of events, in `await.h`.

The functions `event_await` and `event_wake` are used by the system call and interrupt sides to complete a call to `AwaitEvent`, respectively. `event_await` returns `-1` if the event ID is not valid and `0` if it is. `event_wake` returns the ID of the task waiting on the specified event, or `TASK_INVALID` if no task was waiting on that event.

System Call Side

A task that calls `AwaitEvent` is put into state `TASK_AWAIT` and has its ID placed on the wait slot for the event ID it requested via a call to `event_await`. If the event ID was not valid, `handle_swi` sets the task back to `TASK_READY`, sets its return value to `-1`, and pushes it back onto the schedule queue. Otherwise, it will remain blocked on the wait list until an interrupt

occurs that wakes it up. The interrupt handler is expected to set the return values for all other results.

Interrupt Side

When an interrupt occurs, `handle_interrupt` calls `event_wake` for that interrupt event. If a valid task ID is returned, the handler sets the task's return value, sets it to `TASK_READY`, and pushes it back on the schedule queue before clearing the interrupt.

Structures

PQueue

The interface of this function is described in the Scheduling section. This structure contains an array of `pqueue_nodes` called "data" that's the size of the maximum number of tasks (since no more can be scheduled). This works as a fixed size allocator, providing nodes for all priority level queues. All nodes in this list are initially considered to be free, and have their next pointers pointing to their neighbor. Thus, the first node represents the head of a linked list of free nodes. The field "free" holds this head. Finally we have the queues field which holds an array the size of the number of priority levels we have (8). The elements in the array are pqueue_queue structs which simply hold a head and a tail for a linked list representing one priority level queue. The indices of this array correspond to the priority levels.

`pop_task` and `push_task` work by moving nodes between queue lists and the free list. All nodes sit in the same "data" array, but the lists they represent change. When we `push_task`, a node is taken from the free list and becomes the new tail of the desired priority level queue. When we `pop_task` a node is taken from the highest non empty priority level queue and becomes the new head of the free list. These two processes represent allocating and deallocating a `pqueue_node` respectively.

Task

An array of "task" structures which each hold information about a task is maintained. Each task structure holds the task id, parent id, its priority (which is crucial when providing a task id to pqueue which reads this value), its state (running, ready, invalid, or zombie), an exit code is currently unused, a pointer to its stack and its stack base. Currently zombie tasks are not invalided so a task structure and ID cannot be reclaimed. The task states work as follows:

`TASK_READY`: the task is ready to execute
`TASK_RUNNING`: the task is currently running

`TASK_ZOMBIE`: the task has exited but not been deallocated
`TASK_INVALID`: the task is unallocated
`TASK_RECV_WAIT`: the task is blocked by a call to Receive
`TASK_SEND_WAIT`: the task is blocked by a call to Send, waiting to send
`TASK_RPLY_WAIT`: the task is blocked by a call to Send, waiting for a reply

Frame

A structure used to cleanly access a trap frame in memory. Mainly used when creating a new task and setting up an initial trap frame in C. This structure has the same layout in memory as the trap frames that we push and pop in `enter_user` and `enter_kernel`, so a stack pointer with a trap frame at the top can simply be cast to a `Frame` struct then all saved registers, the `cspr`, and the first two stack elements can be accessed using its fields.

MessageQueue

This structure is very similar to the `pqueue`. It keeps track of all message queues. There is one message queue for each task, which stores the id's of all the tasks which have sent a message to the task in a FIFO order. The actual messages sent are stored on the sending task's stacks. Unlike the `pqueue` struct, there is no interaction between the message queues for different tasks in the functions provided in the interface.

The interface for the global message queue includes the following:

`init_message_queue`: Initializes the global `MessageQueue` and must be called before calling any of the following functions.

`push_message`: Takes an id and messaging id and pushes the messaging task onto the back of the first id's message queue.

`pop_message`: Takes an id, removes the first messaging task on it's message queue, and returns it.

`peek_message`: Takes an id and returns the first messaging task on it's message queue.

This structure contains an array of `id_nodes` called "data" that's the size of the maximum number of tasks, since no more than one message per task can be sending at one time. This works as a fixed size allocator, providing nodes for all message queues. All nodes in this list are initially considered to be free, and have their next pointers pointing to their neighbor. Thus, the first node represents the head of a linked list of free nodes. The field "free" holds this head.

Finally we have the queues field which holds an array the size of the maximum number of tasks. The elements in the array are `message_queue` structs which simply hold a head and a tail for a linked list representing one priority level queue. The indices of this array correspond to task ids.

`pop_message` and `push_message` work by moving nodes between queue lists and the free list. All nodes sit in the same "data" array, but the lists they represent change. When we `push_message`, a node is taken from the free list and becomes the new tail of the desired tasks queue. When we `pop_message` a node is taken from the desired task's queue and becomes the new head of the free list. These two processes represent allocating and deallocating an id_node respectively.

SortedList

This structure is similar to the priority queue implemented in K1 and the message queue implemented in K2 in that it uses a similar linked list internally. It's purpose is to keep track of a list of task ID and time pairs that is sorted in ascending order by the time values (smallest times first). These are meant to represent tasks being delayed, and the time which they are supposed to be delayed until. An instance of this list is kept in `clock_server.c`. The interface and it's implementation for this structure can be found in `clock_server.h` and `clock_server.c` respectively.

The interface for the `SortedList` structures includes the following:

`init_sorted_list`: Initializes a `SortedList` passed to it by pointer and must be called before calling any of the following functions on a `SortedList` instance.

`add_item`: Takes a `SortedList`, `tid`, and `time` in ticks and inserts the `tid` and `time` pair into the `SortedList` in the correct place such that the `SortedList` remains sorted. To do this `add_item` first checks the cases where the `SortedList` is empty or the `time` provided is less than the current time. In these cases the `tid` and `time` pair would go at the start of the list. If these are not the case, then it scans through the list until it finds the least element with a time less than or equal to the time provided and places the new element after this one.

`peek_front_tid`: Returns the `tid` at the front of a sorted list. Does not change the list in any way.

`peek_front_time`: Returns the `time` associated with the `tid` at the front of a sorted list. Does not change the list in any way.

`remove_front`: Removes the `tid` and time pair at the front of the sorted list.

This structure contains an array of `list_nodes` called `nodes` that's the size of the maximum number of tasks, since at most all tasks could be delayed. This works as a fixed size allocator, providing nodes the sorted list. All nodes in this list are initially considered to be free, and have their next pointers pointing to their neighbor. Thus, the first node represents the head of a linked list of free nodes. The field `free` holds this head. Finally we have the `list` field which holds a pointer to the head of the sorted list. The sorted list is a singly linked list.

`add_item` and and `remove_front` work by moving nodes between the sorted list and the free list. All nodes sit in the same `nodes` array, but the lists that they represent change. When we `add_item`, a node is taken from the free list and is inserted somewhere in the sorted list. When we `remove_front` the node at the front of the sorted list is removed and becomes the new head of the free list. These two processes represent allocating and deallocating a `list_node` respectively.

Conventions and Assumptions

The user task created must be called `umain` and declared in `user.h`. The kernel must create the first task and the user should not be editing the kernel code.

When a task calls `Exit`, all of its children have their parent ID set to `PARENT_ZOMBIE`, which is defined to be -11 in `task.h`.

There is also a `PARENT_DEAD` defined as -10 in `task.h`, which will become the parent ID of a task's children if `Destroy` is implemented.

Currently there is no checking for collision of task stacks. It is assumed that 128KiB will be sufficient for tasks. It is further assumed that the kernel stack will not grow the ~15MiB from the end of memory into the currently configured lowest user task stack.

Idle Task and Idle Printer

The idle and idle_printer tasks are somewhat special. The idle task is responsible for halting the processor when all tasks are blocked, and the time it spends running is used to calculate the idle time. The idle printer task is responsible for printing idle statistics, the kernel expects that it declares three global unsigned integers: `idle_time`, `start_time`, and `end_time`. These are declared in `idle_printer.h`. The kernel will update these as it measures task run times.

Running

Since it has the lowest priority, the idle task will only be scheduled to run when all other tasks are blocked. It runs an infinite loop where it puts the processor into Halt mode, where the processor stays until an interrupt occurs.

Measuring Idle Time

Task run times are measured by saving the debug lower 32 bits of the debug timer before and after a call to `enter_user`. If the idle task was the last running task, the task run time is added to `idle_time`. The way timing is done also implies that `end_time` is the value of the debug timer at measurement time, which is also approximately the total system runtime. The idle task percentage is thus `idle_time/(end_time/100)`.

Timer

The timer interface has been expanded to work with timers 1, 2, and 3. It provides new functions for enabling and disabling the timers, changing their settings, reading, and clearing interrupts via reading and writing to the relevant memory registers.

Clock Server

The clock server is implemented in much the same way as the name, and RPS servers when it comes to how it parses requests, and how the interface functions wrap message sends. The clock server is communicated with by users by using the `Time`, `Delay`, and `DelayUntil` functions in `clock_server.h`. These are wrappers for 5 byte message `Send` calls to the `clock_server` which store the operation done in the first byte and the integer argument in the last four bytes. Currently, the argument is only used by the `Delay` and `DelayUntil` functions to communicate the time to delay for or until respectively. The four byte integer arguments are stored in and read from the character array messages using the `pack_int` and `unpack_int` functions respectively. The server replies to these `Send` calls with a four byte message in the same manner; it stores an integer representing either the intended return value of the operation or an error code. The task ID of the clock server is obtained by making a `WhoIs` call to the `nameserver` for "`clock_server`". The `clock_server` is created within the `clock_server_test` task in user.c.

A notable difference from past server implementations is that the `clock_server` creates a new task called `clock_notifer`. This task's job is to infinitely loop, notifying the `clock_server` every 10 milliseconds that a tick has occurred. At the start of every iteration it calls `AwaitEvent` on `EVENT_TIMER1_INTERRUPT` which happens every 10 milliseconds

(20 timer cycles). When the event occurs, it then sends a `TICK_OCCURED` message to the clock server, ending the iteration and repeating.

Unlike the name and RPS servers, the implementations of `Time`, `Delay`, and `DelayUntil`, are implemented directly within a switch in the clock server that parses a received message for the command type and argument. The switch also includes a case for a `TICK_OCCURED` command which does not make use of the integer argument and is received via a `Send` from the `clock_notifier` when a new 10 millisecond tick has occurred (the `clock_notifer` has been interrupted by timer 1).

The clock server is implemented using variables that store the state of who is being delayed and the time elapsed since its inception. The `SortedList waiting` stores all the currently delayed tasks, each element being a node storing the ID of the task being delayed and the time that it is to be delayed until. It stores these nodes sorted in ascending order by the time they are to be delayed until.

The `clock_server` function is the task's entry point. It first registers itself as `"clock_server"` on the name server and then begins the `clock_notifier` task. Afterwards, it initializes the `time_elapsed` variable to `0` and an empty `SortedList waiting` that stores all currently delayed tasks. It then begins an infinite loop which works as follows. At the start of the loop a `Receive` call is made for a message. The message is then parsed, using the value of the first character to pick what the operation is to be done, and the value of the last four characters as an integer argument if the operation requires it. The first character of the message will have been set appropriately by one of the three wrappers or the `clock_notifier`. This parsing is done by a switch statement and the code in each case is the implementation of each operation. If `TICK_OCCURED` or `TIME_ELAPSED` was sent then the tasks are replied to immediately. In the case of `TIME_ELAPSED`, the reply contains the ticks elapsed since the clock servers creation in the last 4 bytes. If the `DELAY` or `DELAY_UNTIL` commands are sent and their argument is non negative, then the sending task's ID is placed on the `waiting` sorted list, and they are not replied to. Additionally, `DELAY` is internally converted into a `DELAY_UNTIL` for timekeeping. If the argument was negative, then they are replied to immediately with an error code in the last 4 bytes of the message. After this switch statement, a while loop replies to all messages at the front of the waiting `SortedList` whose `DELAY_UNTIL` times have passed.

UART Interface

Interface functions for UART devices are declared in `uart.h` and defined in `uart.h`. There are functions for enabling, disabling, or clearing TX interrupt, RX interrupt, and combined interrupt (modem status, used to detect changes in the status register) on the UART hardware itself, which are used to manage interrupts as necessary for events. All relevant UART-related interrupts are always enabled in the VIC.

There are also functions to read and write bytes to the UART devices and to read the flags and statuses, also used in events.

UART Events

Events for UART enable the relevant interrupt when a task is put on the wait queue for that event and disable it when the interrupt fires and the task is woken up. This is based on the reasoning that a task will wait on a UART event only when it needs the UART device to be in a specific state, and the kernel should not be responsible for maintaining state for tasks. Thus, it is not valid for a UART device to interrupt the system if there is no task currently waiting for the UART state to change.

In order to account for the fact that the UART 1 CTS flag might change while no task is waiting on either `EVENT_UART1_CTS_LOW` or `EVENT_UART1_CTS_HIGH`, but it is still important to catch these events as they occur, there is some extra functionality in `AwaitEvent`. The importance of why the UART1 TX notifier must see `EVENT_UART1_CTS_LOW` before finding `EVENT_UART1_CTS_HIGH`, and how it is ensured, is discussed in the UART Servers and Notifiers section.

When a task waits on either of these, the kernel side implementation, `event_await`, first checks if the UART 1 device is already in the relevant state and returns to the caller immediately if it is. Otherwise, it enables the UART 1 combined interrupt and puts the task into the wait slot for the event as usual.

When the combined interrupt fires, the interrupt handler checks if the CTS flag is in the correct state, and if it is, it clears and disables the combined interrupt and wakes the task. Otherwise, it clears the interrupt but leaves the task waiting.

CTS is not relevant to UART 2 so the combined interrupt is not implemented for UART 2.

For both UART 1 and UART 2, when a TX interrupt occurs, the interrupt handler unblocks the waiting task and disables the interrupt on the UART device. In addition to the reasoning given above, the TX interrupt is disabled since there is no method to clear it except by writing a byte to the UART device.

When an RX interrupt occurs, the interrupt handler again unblocks the waiting task and disables the interrupt on the UART device. It also reads the byte and returns it to the waiting task as the return value of `AwaitEvent`.

UART Servers and Notifiers

The UART servers provide interrupt driven input/output. To handle UART communication two servers with two separate notifiers each are created, so in total there are four notifiers. `Getc` and `Putc` are implemented as described in the kernel description. One caveat is that the channel parameter is ignored since the channel is determined based on the UART server tid passed in as the tid argument. The UART1 and UART2 servers are registered in the name server as “com1” and “com2” respectively.

UART1 Server

This server task is found as `uart1_server` in `uart_server.c`. The task begins by creating a `uart1_getc_notifer` and a `uart1_putc_notifer` at a higher priority than itself.

The getc notifier works by first sending a `NOTIFIER_STARTUP` message to `uart1_server`. This message does nothing in the server, and its purpose is to ensure the getc notifier begins blocked without telling the server that a character was received. When it’s woken up, it begins an infinite loop where it calls `AwaitEvent` on `EVENT_UART1_RX_INTERRUPT`, then sends a `CHAR_RECEIVED` message to the `uart1_server` along with the character returned by `AwaitEvent`.

The putc notifier works by sending a `PUT_READY` message to the server, since we begin in a state where we are ready to transmit bytes. After being replied to, it immediately calls `AwaitEvent` three times on `EVENT_UART1_CTS_LOW`, `EVENT_UART1_CTS_HIGH`, and `EVENT_UART1_TX_INTERRUPT` in that order. The order is important since we first need to ensure CTS has gone low indicating that the byte was in the process of being sent, before we know that were clear to send again.

The `uart1_server` has a circular buffer queue for getc and and another for putc. It runs an infinite loop where it begins by receiving a message, then using a switch statement to parse the message type. `GET_CHAR` and `PUT_CHAR` are commands sent by the `Getc` and `Putc` wrappers respectively. The `PUT_CHAR` case immediately places the character received onto the putc

queue then replies to unblock the process. The GET_CHAR case checks if the get buffer has any characters in it, in which case it replies with the next character in the queue. Otherwise, the queued_getc_task variable is set to the tasks tid, the task is not replied to and will be blocked until a character is received. If the queued_getc_task already has a tid in it, we return an error code stating that another task is currently blocked on Getc. This means that we currently only allow one task to wait for a Getc at a time. Technically, we do not need a queue to hold getc characters and it would suffice to use a single char because of this. This may be implemented in the future, but it currently has a negligible performance hit. The PUT_COMMAND case is the same as the PUT_CHAR case, but places two characters on the put buffer instead of one.

The PUT_READY case is triggered by the putc notifier and is used to set a put_ready flag to true. This flag is used to trigger code at the end of the loop to write characters from the put buffer to the UART when it is ready to transmit a character.

The uart1_server only ever blocks one of the two notifiers at once. This is done because we can only ever be receiving or transmitting at once since UART1 is half-duplex. Whenever a byte is written from the putc buffer signifying a request for sensor data that is between 128 and 133 inclusive or 192 and 197 inclusive, the server wakes up the getc notifier and then writes the byte rather than waking the putc notifier. The getc notifier is then the only one awake. A receiving variable is set to the number of bytes expected based on what value was put to trigger a sensor dump.

The getc notifier will continue to send CHAR_RECEIVIED messages to the server when it is unblocked until all characters of the sensor dump have been received. In the CHAR_RECEIVED case, the receiving variable is decremented and the byte is added to the getc buffer. If receiving is now zero, all bytes have been received. Since UART1 is half-duplex, the put_ready flag is set to indicate that we are ready to put. Importantly, the getc notifier is not unblocked. If receiving is not yet zero, the byte is added to the getc buffer and the getc notifier is unblocked.

At the end of the loop, after the switch statements, is a condition that checks if the UART is ready to put, and if the put buffer has characters in it to put. If these are the case, it checks if the put is requesting a sensor dump, in which case it sets the receiving variable, wakes up only the getc notifier, and then writes the byte to the UART and previously described. Otherwise, if the character does not request a sensor dump, the putc notifier is unblocked then the char is written to the UART. It's important that the putc notifier gets unblocked first. Since it is of higher priority it will cause it to wait on the CTS low event before we write the byte, ensuring that it will detect the CTS low. We must detect a CTS low first, because we are interested in detecting

the state change from CTS low to high, not that CTS is high. Waiting on CTS high alone is erroneous because it may have never gone to low, and the interrupt may be generated from a change to a different status flag. We have no method of detecting change in state in the kernel. At the end of this condition, the `put_ready` variable is set to false. It can then only be reactivated by a putc notifier message or the last getc occuring in a sensor dump.

At the end of the loop is one more condition that checks if there is a task waiting for a getc, and a character in the get buffer. If this is the case, the character at the front of the getc queue is removed and sent to the task, unblocking it.

UART2 Server

This server task is found as `uart2_server` in `uart_server.c`. The task begins by creating a `uart2_getc_notifer` and a `uart2_putc_notifer` at a higher priority than itself. Note that these are separate notifiers from the UART1 server.

This server and its notifiers work similarly to the UART1 server, with the main difference being that no restriction is placed on which notifiers are unblocked at a time. This means that both notifiers may be unblocked at once. Additionally, CTS is not used for transmitting and only the `EVENT_UART2_TX_INTERRUPT` event is waited for.

The putc notifier immediately begins an infinite loop, first sending a `PUT_READY` command to the `uart2_server` then calling `AwaitEvent` on `EVENT_UART2_TX_INTERRUPT` afterwards before repeating.

The getc notifier immediately begins an infinite loop, first calling `AwaitEvent` on `EVENT_UART2_RX_INTERRUPT`, then sending a `CHAR_RECEIVED` message to the server along with the character returned by the `AwaitEvent` call.

Like the `uart1_server`, after creating its notifiers, and initializing its putc and getc buffers and other variables, the `uart2_server` enters an infinite loop where it begins by receiving a message, then using a switch statement to parse the message type. The `GET_CHAR`, `PUT_CHAR`, and `PUT_READY` cases function the same as in `uart1_server`. The `CHAR_RECEIVED` case is simpler in that it only adds the received byte to the get buffer, and immediately unblocks the getc notifier.

After the switch statement, there is a similar case that checks if `put_ready` is true and the put buffer has characters, although it is simpler because UART is not half-duplex. It removes the byte at the front of the putc buffer, writes in to UART2, then unblocks the putc notifier. This is

notably different from the uart1_server which must reply before writing to ensure the CTS low interrupt is caught. In this case we must write first to clear the interrupt, and then reply to the putc notifier. If the putc notifier were replied to first, it would instantly send another message because its higher priority and its AwaitEvent would return due to the same interrupt which hasn't been cleared yet.

At the end of the loop is the same condition in uart1_server that checks if there is a task waiting for a getc, and a character in the get buffer. If this is the case, the character at the front of the getc queue is removed and sent to the task, unblocking it.

Track Control Server

The track control server organizes all commands to be sent and received over UART1 to control the track. It provides several interface functions outlined in tc_server.h to send commands and read sensor data. The task itself can be found as tc_server in tc_server.c. The tc_server provides atomicity to train commands, while also allowing unblocking behaviour for commands that take time such as reversing and switching turnouts. It also organizes all track commands that may be sent from multiple user tasks.

The interface provided by the tc_server is expected to change come TC1 and is currently setup to fulfill the needs of K4. Due to the setup of tc_server's internals, it can offer either blocking or non blocking commands. Currently, commands are chosen to block or return immediately based on what is needed for K4.

The tc_server works by maintaining separate command queues for each train, and one for switch commands. Each command may be associated with a delay, which will be waited for until the next command in its queue is executed. Within each queue commands are run in serial with possible delays between them, and all queues are run in parallel to each other. This allows multiple delayed commands such as reverses to be run in serial for the same train, while allowing other commands to run on other trains or switches at the same time. Additionally, each command may be associated with a task id to unblock, this allows commands to keep a task blocked until they are sent to the UART1 server.

The command queues are represented by the circular buffers called CommandRingBuffers. These have appropriate helpers to initialize them and add, remove, and peek commands as well as process all commands at the beginning given some delta time with process_command_queue. The collection of all train queues and switch queue is called the CommandQueue and has helpers to initialize it, check if all of its queues are empty, and process

all commands in its its queues given some delta time by calling process_command_queue on each queue (process_commands).

The workhorse of the tc_server is the process_command_queue function. It works by continuously subtracting time and popping elements from the front of the queue until the time subtracted is equivalent to the delta time provided. What's important to note is that a command is first run, then a flag is set in the queue that says its first command has run, then its delay is decremented until its zero and popped, in which case the flag is set back to say the new front command has not run. This is done to ensure the delays happen after the commands, and not before.

As with all other servers created, all of the functions defined in tc_server.h besides the tc_server task itself, are wrappers for Send calls to the tc_server. Whether or not each function blocks or returns immediately is outlined with inline comments. Before a task sends any command to the tc_server, it must first call InitComplete which will unblock once the server is finished its initialization procedure and the state of the track can be determined.

The tc_server currently keeps track of the state of trains as determined by the last command added to the queue. This may be augmented for TC1 to also keep track of the state of trains based on which command was last sent to the uart1_server, which is different from which command was last queued. Functions may also be added to get train speed.

The tc_server creates two notifiers. One is called the tick_notifier, which continuously delays for one tick then sends the time to the tc_server. It is used to wake up every tick and process commands that are ready. The other is called the sensor_dump_notifier which begins blocked by sending a useless NOTIFIER_STARTED message to the server, then makes the appropriate Putc call to begin a sensor dump, calls Getc 10 times, and sends the sensor dump to the tc_server with the SENSOR_DUMP message.

After initializing variables, the tc_server begins by sending a go command to the uart1_server, and adding commands the CommandQueue to set all train speeds to 0 and switch all turnouts to straight. After this it begins an initialization loop that receives TIME_CHANGED messages from the tick notifier to process the initial commands, as well as INIT_COMPLETE_COMMAND and NOTIFIER_STARTED messages. NOTIFIER_STARTED does nothing and is used to start a notifier on a blocked state. INIT_COMPLETE_COMMAND adds the sending task to a list of tasks waiting for initialization to be completed.

Once all initial commands in the CommandQueue are executed and it is empty, the initializing loop breaks, signalling that initialization is complete. Another loop is run to wake up all tasks waiting for initialization to be completed, and then the main loop begins infinitely looping.

The main loops works similarly to all other servers, calling Receive at the beginning, then parsing the command received with a switch statement. The same TIME_CHANGED case is used as in the initialization loop. INIT_COMPLETE_COMMAND now immediately replies to unblock the sending task. SWITCH_TRACK_ASYNC_COMMAND, SET_SPEED_COMMAND, and REVERSE_COMMAND all add commands with apprirate delays onto the tc_servers CommandQueue then reply to unblock the task immediately. GET_SENSORS_COMMAND unblocks the sensor_dump_notifier so it begins obtaining sensor data then stores the tid of the sender. Currently only one task is allowed to ask for a sensor dump at once, but this will likely change so that if a dump is occuring, all commands that send in that time will receive a copy of it when it's completed. SENSOR_DUMP is sent by the sensor_dump_notifier and is accompanied by a 10 byte sensor dump. This sensor dump is sent in a reply to the task waiting for a sensor dump. SWITCH_TRACK_COMMAND works similarly to SWITCH_TRACK_ASYNC_COMMAND except does not reply to the task, and instead creates an unblocking command instead of a normal command to turn the solenoid off that unblocks the task when the command to turn off the solenoid is eventually sent to the uart1_server.

Terminal Task

The terminal task works both to provide an interactive terminal for the user, and organize prints by other tasks with the TPrint function. The TPrint function uses a format_string function from string_utility.h that formats the string into a new buffer before sending it to the terminal task.

In terms of providing an interactive terminal, all of the features from a0 have been carried over as mentioned in the K4 assignment description. Various variables are used to keep track of the currently typed command, as well as the state of the sensors and turnouts. Most of the code to keep track of the state and print what's on the terminal was carried over from Victor Scurtu's A0 and it is not in the scope of this report to elaborate on it further.

The terminal uses four notifiers to function. The input notifier continuously calls Getc on UART2 to get user input, then sends it to the terminal. The terminal_tick_notifier continuously delays for 5 ticks then sends the current time to the terminal. This is used to print the time and handle a spinner animation. The track_initialized_notifier does not loop, and instead simply calls InitComplete once then sends a message to the server when InitComplete returns. This is used to block user commands sent to the server during the

initialization period as well as stop the terminal quitting until initialization is completed to avoid a hanging turnout switch command burning out a solenoid. The sensor state notifier continuously calls `GetSensors` to get a sensor dump from the `tc_server`, and sends the result to the terminal task. It's created only after the `tc_server` has finished initialization. The terminal task knows the id's of all of its notifiers since it creates them, and so uses the id of the sender to figure out what was received. If the input notifier sent a message, the input is pulled from the message and parsed accordingly. If the sensor state notifier sent a message the sensor states are parsed from the 10 bytes and the list of the last 16 sensor hits is updated if needed. If the terminal tick notifier sent a message, the time is extracted from the message using an `unpack_int` utility function, and that time is then printed to the console. A soothing spinner animation may also be updated depending on the tick.

If the task sending the message was not a notifier, it is assumed to be a `TPrint` message sent from another task. `TPrint` calls a function to format the output into a char array before sending it to the terminal task. When received, this currently only calls a `UPrint` function which simply calls `Putc` in a loop without formatting the string (ignoring %'s). In the future this will be used to maintain a separate output area on the terminal that any task may print to without disrupting the command line and state output.

When the "q" command is called, a `Shutdown` kernel primitive that calls a `SYSCALL_SHUTDOWN` syscall is called which causes the kernel to break out of it's main loop, cleanup, and return to redboot.

Known Bugs

- Since only the lower 32 bits of the debug timer are used for idle timing, the idle percentage will become inaccurate after about 72 minutes of run time when those bits overflow.
- The output is wide due to the sensor data being printed horizontally. Because of this, if the window is too narrow it will destroy the formatting of the window, particularly the switch state readout. To be safe, keep the window at least 110 characters wide.
- Some sensor hits may appear at the beginning despite nothing happening on the track. These are sensor activations left over from the previous run that were sent when no program was accepting them and should be ignored.

General Notes

- There is an implementation of memset added in memset.c/memset.h. It is used by gcc when the -O3 flag is enabled.
- The timer module communicates with the 40-bit timer. Reading the timer reads only the lower 32 bits. This is done for speed in the timing tests, as overflowing into the higher bits has not been a problem.

Inlining

Okay this is a fun one. There's a Python 3 script - `kernel/inline.py` - which handles combining all of the kernel source into a single monolithic file. It takes two arguments - a path to the source directory and a path to the build directory. This is passed in by the Makefile that calls it.

First, it uses ctags to find all the typedefs, structs, and functions defined in the source. It processes the typedefs and structs into a list of types it uses to identify function signatures, and stores the functions themselves for later. It then pairs off all the .c and .h files into objects. The one unpaired .c file is main.c, which cannot be inlined, and the rest of the .h files are header-only definitions which also get their own objects.

Next, it outputs code into a file called inline-all.c in the build directory. First it outputs the .h files starting with constants.h the rest of the unpaired .h files, followed by the rest of the .h files. During output, it comments out any function declarations that were found by ctags, as they will be declared later. It also comments out #include that have quoted headers, as all local headers will be copied in.

Then, it writes out static inline declarations for all the functions found by ctags in a declaration block in inline-all.c, with exceptions for panic and print_lr, as they are called from assembly and cannot be inlined.

Finally, it copies out all the paired .c files into inline-all.c, but when it sees a function signature, it outputs it with __attribute__((always_inline)) above and static inline before the original signature. The last file to be copied is main.c, which is a special case since main cannot be inlined.

This is obviously all super janky and is not considered stable enough to run as default. It may never be, but it sure was interesting to make. It exists just in case we run into performance issues later that can be fixed by inlining, but without requiring us to change our code significantly to inline.

If you wish to run it anyway, replace `make` with `make inline` in the build instructions. It will attempt to run the script to produce build/inline-all.c. If this succeeds, it will then attempt to compile the created inline-all.c file. If by some miracle that also succeeds, k4.elf will be generated as expected.

I hope you at least find this amusing. :) > K4 NOTE: this still seems to work!