Introduction to the legOS kernel.

Stig Nielsson

September 27, 2000
## Contents

1 **Hardware and the GCC cross compiler** ........................................ 3
   1.1 CPU .................................................................................. 3
   1.2 ROM and firmware .............................................................. 3
   1.3 External RAM ................................................................. 3
   1.4 The RCX subsystems .......................................................... 3
   1.5 Stack frame layout ............................................................. 3

2 **The legOS kernel 0.2.4** ............................................................... 4
   2.1 Starting the kernel ............................................................. 4
   2.2 Timing .............................................................................. 4
   2.3 Task management ............................................................. 5
      2.3.1 Dynamically linked user programs ............................... 5
      2.3.2 The execi function ..................................................... 6
      2.3.3 The task structures ..................................................... 6
      2.3.4 Summary of the notions of kernel image, user programs, and tasks. 7
      2.3.5 The scheduler .......................................................... 7
   2.4 Memory management ....................................................... 8
      2.4.1 Allocating memory ..................................................... 8
      2.4.2 Freeing memory ......................................................... 9
   2.5 Interprocess communication ............................................... 9
   2.6 Semaphore facility .......................................................... 9
   2.7 IR networking ............................................................... 10

3 **Peripheral devices** .................................................................. 10
   3.1 Motor control ................................................................. 10
      3.1.1 The motor hardware .................................................. 10
      3.1.2 Motor handler function .............................................. 10
      3.1.3 User interface to motor control .................................. 10
   3.2 Sensor control ............................................................... 10
      3.2.1 The on-chip A/D converter ....................................... 11
      3.2.2 Initialisation ............................................................. 11
      3.2.3 The A/D interrupt handler ......................................... 11
      3.2.4 Rotation sensors ...................................................... 11
      3.2.5 User interface to sensors .......................................... 12
   3.3 Sound control ................................................................. 12
      3.3.1 The sound handler ..................................................... 12
      3.3.2 Hardware issues ........................................................ 12
      3.3.3 User interface for the sound driver ............................... 12
   3.4 LCD display control .......................................................... 12
      3.4.1 The LCD handler ...................................................... 12
      3.4.2 User interface to the LCD display ............................... 13
   3.5 Button control ............................................................... 13
      3.5.1 The button handler .................................................... 13
      3.5.2 Hardware issues ........................................................ 13
      3.5.3 The button user interface .......................................... 13

References ................................................................................. 13
Abstract

This article presents an introduction to the legOS kernel with emphasis on documentation of the kernel source version 0.2.4, and less emphasis on the low level hardware issues.

LegOS is an open source embedded operating system, featuring preemptive multitasking, dynamic loading of programs, dynamic memory management and IR networking. LegOS is designed to run on a Lego Mindstorm RCX brick based on the Hitachi H8/3292 microcontroller. LegOS was started by Markus Noga in October 1998, continuously evolving as an open source project at http://legOS.sourceforge.net

1 Hardware and the GCC cross compiler

This section briefly describes the RCX hardware as well as the conventions used by the GCC cross compiler are discussed.

1.1 CPU

The RCX brick uses the Hitachi H8/3292 microcontroller extended with 32k external RAM. The microcontroller supports a 16 bit address space, and has 16 8 bit registers (R0H,R0L,...,R7H,R7L), which also may be accessed as 8 16 bit registers (R0,...,R7) for addressing purposes. There are 2 control registers: a 16 bit program counter register (PC) and an 8 bit conditions code register (CCR). The R7 register is used as the stack pointer register, and points to the top of the stack (lowest address). In the assembly language R7 is synonymous with R7. All stack operations access the stack in 2-byte words.

1.2 ROM and firmware

The CPU has 16kb of ROM, containing software provided by the manufacturer. The ROM contains a driver which is run when the RCX is powered up. The ROM provides low level routines for driving the RCX and its subsystems. The ROM interrupt handlers call addresses in RAM, which allows the firmware to customise interrupt handling, further details of the ROM can be found in [3].

The ROM driver is responsible for starting the firmware, which is the software stored in the external RAM. By factory default this contains a bytecode interpreter, leaving only 6k of RAM for user programs. When legOS is downloaded, the firmware is replaced by the kernel image.

LegOS only uses a few of the ROM functions, which are the functions dealing with power on/off, refreshing of the display and sound playing.

1.3 External RAM

The external RAM is referenced in the 0x8000-0xFFFF address range, although parts of this address space are reserved for memory mapped I/O. Especially the on-chip registers are mapped to memory in the 0xFF88-0xFFFF range. Furthermore, addresses are reserved for LCD display memory, memory mapped motor control, shadow registers for I/O ports, and interrupt vectors. The RAM holds the kernel image and user programs and is thus divided into a kernel part and a user program part. The kernel part is located in the lower part of memory, and user programs are located at higher addresses, see section 2.4. Besides the external RAM, the RCX does not provide any storage devices, so consequently there is no notion of a filesystem in legOS.

1.4 The RCX subsystems

The Hitachi H8/3292 microcontroller is equipped with a number of on-chip subsystems. This include a 16 bit timer, 8 bit timer module with 2 channels, an A/D converter, I/O ports. The 16 bit timer is used to generate the timer interrupts which drives the OS. The 2 8 bit timer channels are used for generating signals to the speaker and for timing infrared communication. The A/D converter is used to sample the analog signals from the sensors. The I/O ports are used for monitoring button presses and controlling motors etc. The interaction between the subsystems and the OS will be touched upon in 3.

1.5 Stack frame layout

This is a description of the default stack frame layout used by the GCC cross compiler when compiling C programs for the H8/3292, being useful when the task manager is explained.
A function call in C is translated to the `j r` (jump subroutine) instruction, which implicitly pushes the PC register onto the stack. GCC uses R6 as a base pointer register and by convention R6 is callee saved. Actually R6 does not point to the base of the stack frame, but to the element following the base element.

The callee starts by pushing R6 on the stack, and loads the address of this element in R6. R6 will now point to the location which holds the just saved R6. The first 3 arguments are passed in registers R0, R1 and R2, and are copied to the next 3 elements in the stack frame by the callee. Remaining parameters are referenced in the callers stack frame. Local variables are allocated on the stack after the first 3 arguments, see figure 1. The stack pointer points to the last element pushed onto the stack.

Figure 1: GCC's stack frame layout

The function returns by executing `rts` (return from subroutine), but just before that the stack pointer, and the callee saved R6 are restored. The PC register is implicitly popped by the `rts` instruction.

2 The legOS kernel 0.2.4

The legOS kernel is monolithic, as all of the source are compiled together in the same binary image. The interface for user programs are given in a dynamic linker script, which contains all exported kernel symbols.

2.1 Starting the kernel

The kernel starts when `main` is called by the ROM. This function initialises the kernel, before it is started in either single tasking or multitasking mode. If the task manager is not included in the kernel, only singletasking mode is possible, and dynamic program loading will not be possible. In the following I assume that all features of legOS are used.

During kernel initialisation, 3 tasks are started. The first is the idle task, which is a dummy task with the lowest priority. The idle task executes when no other tasks need CPU time, and it puts the CPU in power down mode by indefinitely executing the `sleep` instruction. The next 2 tasks implement the dynamic program loading capabilities. They are called `packet_consumer` and `key_handler` and are run with the highest possible priority. The `packet_consumer` handles activity on the IR-port, and the `key_handler` handles activity on the buttons of the RCX brick. The task manager is then started, and the kernel starts switching the execution among the 3 tasks. New tasks are started by downloading and starting userprograms, as explained in section 2.3.1.

2.2 Timing

LegOS is driven by interrupts from the 16 bit timer, which is configured to make an interrupt every millisecond. The timer interrupt is handled by a ROM function, which in turn calls the function pointed to by `ocia_vector`. This vector points to the `systime_handler` function, defined in `kernel/systime.c`. The `systime_handler` function polls all the various subsystems in turn, by calling their handlers. This means that legOS uses polling instead of interrupts, to communicate with the environment.

The subsystems are handled in the following order: system timer is incremented, motor handler is called, sound handler is called, LNP (legOS Network Protocol) is checked for timeout, the button handler is called, the battery indicator is updated, the legOS state visualisation is updated and the LCD display is updated.

The last thing which is done, is to check whether a task switch is needed by inspecting the timeslice counter of the current task. The default configuration of timeslices is 20 milliseconds. The practical timeslice limits are about 6 milliseconds at minimum and 255 at maximum [6]. If the timeslice of the current slice has elapsed, the `tm_switcher`
function is called, which will perform a context-switch in cooperation with the scheduler.

### 2.3 Task management

In the following text, I refer to the task manager, by which I mean the collection of functions and related functions, implemented in `kernel/tm.c`, which deal with the task managing issues. The task manager is not a separate process or thread.

Before discussing the multitasking properties of legOS, it is important to make a clear distinction between the notion of programs and the notion of tasks/processes (task and process are used interchangeably), for a brief summary, see 2.3.4. A program in legOS is the actual binary image stored in RAM. A process is a thread of execution, and a running program will have at least one process, and it can start new processes by the `execi` function, see section 2.3.2. Actually, processes are more like threads, as they share the memory space of the program which started them.

![pdata_t structure](image)

Figure 2: The `pdata_t` structure.

The task manager implements the preemptive multitasking feature of legOS, by preempting and scheduling processes, according to some scheduling scheme. A process is described by the process data structure, defined as `pdata_t` in `include/tm.h`. This structure holds the information necessary for switching among different tasks, and for organising tasks in priority queues. The `pdata_t` structure includes a saved stack pointer, a process state, process flag, the process priority, pointers to neighbours in the process queue, a pointer to the parent process, a pointer to the beginning of the stack and a pointer to the function to be called when the function returns from a wait state (the wake-up function). The process state has 5 different values:

- **Dead** The process has terminated and its stack has been deallocated.
- **Zombie** The process has terminated, but its stack has not yet been freed.
- **Waiting** The process is idle and waiting for an event.
- **Sleeping** The process is idle but ready to run.
- **Running** The process is running.

The state information is used by the scheduler, and it is the responsibility of the scheduler to free the stack of a terminated process. A process can voluntarily stop executing and wait for an event, which is done by calling `wait_event`. The parameters for `wait_event` specify a wake-up function, which is used to check whether the wait condition is fulfilled. The wait condition check is performed by the scheduler, when it encounters waiting processes during the search for a new process to execute. Wake-up functions execute atomically even though they do not disable interrupts, because they are called by the `systime_handler`, which has disabled interrupts. If the wake-up function returns a non-null value, the wait condition is no longer satisfied, and the process is ready to run. The problem with this scheme is that wait conditions only are checked when a new process is scheduled. Thus the wait condition could have changed value many times between the check, which could have serious consequences if the process waits for an external event which must be reacted to rapidly. [6] discusses this problem.

#### 2.3.1 Dynamically linked user programs

In the early versions of legOS, the user programs had to be statically linked with the kernel. The kernel and user program were contained in a single binary image, and the entire kernel had to be recompiled when changes was made to the user program.

The latest versions of the kernel provide dynamic program loading, which makes it possible to load up to 8 programs into the RAM. User programs are compiled into a relocatable format with extension `.lr`. A dynamic linker and loader program, called `dll` is used to download programs
and link them with the kernel. The boot directory contains the 2 files legOS.src and legOS.lds, the first being the kernel image, and the second a linker script generated during compilation of the kernel. The linker script associates all exported kernel symbols, with their addresses relative to 0x8000, i.e. the start of RAM. When compiling user programs legOS.lds is used in the linking step to create the .lx file. During the downloading of programs, the final linking relative to the memory allocated for the program is performed.

When downloading a program, the packet_consumer task responds to the communication requests from the IR tower, takes care of allocating memory for the program, and stores the program in the programs array. After the downloading is completed, the program can be started by pressing the run button on the RCX brick.

Information about the user programs are stored in an array of 8 program_t structures. This structure holds information about addresses and sizes of the .text, .data, and .bss segments of the programs. Furthermore the structure contains information about stack size, the start address relative to the start of the text segment, priority of the program and a count of the number of downloaded bytes, which is used for detecting errors in the download process. In kernel version 0.2.4, it is not possible for one user program to start another user program, because they do not know the address of each others program_t structures. By exporting the programs array and the program_run function to legOS.lds (by making them non-static), it is possible for one program to start a program in another program slot by indexing into the programs array. Otherwise, a program has to be started by selecting it with the play button on the RCX, and then pressing the run button. When a program is started, a new task is created with a call to the execi function.

2.3.2 The execi function

The execi function is used for creating new tasks. Execi takes as arguments, a pointer to a function to execute, the number of arguments, the actual arguments to the function, the priority of the new task, and a stack size.

Execi will try to allocate memory for a new pdata_t entry in the priority queues, and allocate memory for the task’s stack. The new task is entered into the task structure as a sleeping process, as if it had been preempted by the scheduler and it is therefore necessary to simulate the new task’s stack as that of a sleeping task, in order for the task manager to execute it.

<table>
<thead>
<tr>
<th>Address of interrupted task’s next instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM callee saved data</td>
</tr>
<tr>
<td>&amp;rom_cci_return</td>
</tr>
<tr>
<td>&amp;systime_tms_return</td>
</tr>
<tr>
<td>saved context of current task, R1...R5</td>
</tr>
</tbody>
</table>

Figure 3: The general stack frame of a sleeping task

The address of the exit function is pushed on the stack. This will be the return address of the task, and this way tasks will always finish by a calling exit. Exit will free the memory that the user program allocated.

<table>
<thead>
<tr>
<th>&amp;exit</th>
<th>New task’s frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;task code start</td>
<td>ROM timer interrupt</td>
</tr>
<tr>
<td>ROM saved ccr(-0)</td>
<td>Systime_handler</td>
</tr>
<tr>
<td>ROM saved r6(-0)</td>
<td></td>
</tr>
<tr>
<td>&amp;ROM return address</td>
<td></td>
</tr>
<tr>
<td>callee saved a0</td>
<td></td>
</tr>
<tr>
<td>&amp;systime return</td>
<td></td>
</tr>
<tr>
<td>saved rl(-argv)</td>
<td></td>
</tr>
<tr>
<td>saved r2(-0)</td>
<td></td>
</tr>
<tr>
<td>saved r3(-0)</td>
<td></td>
</tr>
<tr>
<td>saved r4(-0)</td>
<td></td>
</tr>
<tr>
<td>saved r5(-0)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: The generated stack frame of a new task

The following 3 generated stack frames are: a stack frame of the ROM timer interrupt function, a stack frame of the systime_handler, and a stack frame of the task manager switcher. The details can be studied in figure 4.

2.3.3 The task structures

Each task is associated with a priority. In the current version there are 20 priority levels with
1 as the lowest priority, and 20 as the highest. The number of tasks which can be associated with a given priority level is unbounded. The tasks are organised in a task structure, which is an ordered list of priority levels, the priority chain, which each has a non empty double linked list of pdata_t structures, called the task queue. Each priority level has a designated current task, which is the task that is currently executing, or most recently has executed. Figure 5 illustrates the task structure. Note that not all pointers have been included. Each pdata_t structure also contains a pointer to the pchain_t element which points to its priority chain, and a pointer to pdata_t structure of the parent process.

**Figure 5: The task structures**

Tasks are added to this structure by the execi function. Execi accesses the task structure in mutual exclusion with other processes, by checking a semaphore, which also ensures that the scheduler is not scheduling while the task structure is being updated.

An insertion of a new task starts by traversing the priority chain from the top towards the lower prioritized levels. If a chain exists with a priority level equal to the priority of the new task, the task is inserted in the back of the associated task queue, which will be just after the priority level’s designated task. Otherwise, a new entry in the priority chain is created, and the task is inserted in the associated task queue.

### 2.3.4 Summary of the notions of kernel image, user programs, and tasks.

To summarise, the kernel is a separately compiled binary image, which exports a set of symbols (the symbol table) in a linker script called legos.lds. The kernel image is located in the beginning of RAM.

User programs are compiled separately, and linked with the kernel symbol table, creating the relocatable format .lo. The final linking to absolute addresses are done when downloading the program with dll, after the absolute address of the text segment of the program is known. User programs are downloaded to the part of RAM following the kernel image part.

Tasks are started by a user program as threads in the address space of the user program. As user programs are compiled in separate address spaces, user programs cannot start threads in other user programs.

### 2.3.5 The scheduler

When a the timeslice of a process has elapsed, or when a process voluntarily gives up the CPU, the scheduler is invoked to decide which process to execute for the next timeslice. A trywait is made on the task semaphore. If it is blocked, some other process is updating the task structures, so the scheduler returns and no new process will be scheduled. If the task semaphore is not blocked, the scheduler will block it, and examine the state of the current process. If it is a zombie, the scheduler deallocates its memory, and removes its entry from the task structure. If it was the last process in the priority level, the level is removed from the priority level list.

To find the next process to be scheduled the priority levels are examined, in prioritized order. In each level the state of the task following the designated task in the queue is tested. If it is sleeping, the process is ready to be switched in, and it is scheduled for execution, by setting its state to running. The scheduler posts the task semaphore, and returns the saved stack pointer of the new task.

If the task is in a waiting state, its wake-up function is tested. If the wake-up function returns a non-null value, the process is selected for
running, and the scheduler returns as described above. If the process is not finished waiting, the next process in the level is examined. If no ready processes are found in the level, the scheduler proceeds to the next priority level. The idle task is never in the waiting state, meaning that this task will execute if no other tasks are eligible.

The scheduler thus uses a prioritized round robin scheme. The combination of shared protected resources (semaphores) and prioritized scheduling, introduces the problem of priority inversion, where a lower prioritized process blocks a higher prioritized by holding a resource needed by the higher prioritized process. No attempts have been made to avoid this problem, but [4] discusses this problem and provides a solution.

### 2.4 Memory management

LegOS uses a straightforward continuous allocation scheme for managing the memory. There is no support in the RCX hardware for advanced memory management schemes such as paging or segmentation. Furthermore, the memory is a scarce resource, so an advanced memory management scheme would probably imply too much overhead. In the following, the memory manager denotes the collection of functions defined in kernel/mm.c which implements the memory management.

The memory is divided into a kernel part and a user program part. The kernel code and static kernel data are located in the lower part from address 0x8000 up to address mm.start, which is a global variable. The memory manager has the responsibility of all the memory from address mm.start to address 0xFFFF.

<table>
<thead>
<tr>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
</tr>
<tr>
<td>Data field</td>
</tr>
</tbody>
</table>

Figure 6: The basic memory block

The memory is organised in memory blocks which consist of a 4 byte header and an even number of data bytes, see figure 6. The first 2 bytes of the header contain the process id of the process which has allocated the block. If the block is unallocated, the value is MM_FREE which is defined to be zero. The 2 last bytes of the header hold the size of the data block following the header. A global pointer variable mm_first_free is used to indicate the first free memory block. The unit of memory used by the memory manager internally is 2 byte words.

During kernel startup, the memory is initialised to contain the blocks indicated in 7, and mm_first_free it set to point to the first free block. The initialisation is done with 2 macros, MM_BLOCK_FREE(addr) and MM_BLOCK_RESERVED(addr).

![Figure 7: The initial memory allocation](image)

As memory is freed, adjacent free memory blocks may appear, and they have to be merged to reduce the amount of external fragmentation. The function mm_try_join takes the address of a free memory block as parameter, and merges it with all the following free memory blocks, until a non-free block is found. The merging requires locking of the memory manager semaphore, mm_semaphore, which makes all other processes trying to allocate memory, wait for the merging operation. Merging is done at different times, depending on the way the kernel runs. In singletasking mode the merging will be done when memory is freed, but in multitasking mode it is important to keep the locking of the mm_semaphore at a minimum, why it is done at allocation time, where the semaphore is already locked.

#### 2.4.1 Allocating memory

Kernel processes and user programs request memory allocation by calling malloc. Malloc first waits for the memory manager semaphore mm_semaphore to ensure that no other processes are manipulating the memory. When granted access
to the critical region, malloc will start searching the memory from address mm_first_free, until a free memory block of suitable size is found (or end of memory is reached in which case null is returned).

If legOS is running in multitasking mode, an attempt to merge the free memory block with the following memory block is made. If the difference between the size of the free memory block and the requested size exceeds a certain threshold (currently defined to headersize + 8 words), the memory block will be split. When this splitting occurs, the allocated block will get exactly the requested size, and the remainings of the block will form a new free block. This reduces the amount of internal memory fragmentation. The threshold value must be chosen with care. If it is too small, there will to much overhead in keeping track of the memory blocks, and if it is too large, there will be too much internal fragmentation. If successful, malloc returns a pointer to the datablock of the allocated memory block.

### 2.4.2 Freeing memory

Freeing dynamically allocated memory is done by calling free. This function simply marks the designated block as free. Only non-null even addresses will be considered. As mentioned above, an attempt to merge the freed block with the preceding block is made, if the kernel is running in single tasking mode. Finally the mm_first_free pointer is updated.

When processes terminate, their dynamically allocated memory has to be freed. Exit is called when processes terminate, and exit calls mm_reaper. MM_reaper makes 2 passes through all memory blocks. The first pass marks all blocks belonging to the terminating process. Note that the stack memory is not owned by the process, but by its parent process, and thus the stack will not be freed by mm_reaper. If the process is flagged as a kernel process, the memory blocks are not marked. This is because all dynamically allocated memory is deleted when the RCX is turned off. The text and static data segments of the

---

1Note that there was a bug in mm_reaper in the official 0.2.4 release. The fix to this bug added the notion of process flags to the pdata_t structure, to be able to differentiate kernel processes from user processes.
wake-up function checked is woken up. As the wake-up function decrements the semaphore, only one process will be woken. The scheduler always examines the higher prioritized processes first, so the higher prioritized processes are woken before lower prioritized waiting for the same semaphore. As mentioned earlier, the problem of priority inversion exists in legOS.

2.7 IR networking

This section is based on the information provided from the legOS newsgroup [2].

LNP is the LegOS Network Protocol which is used by user programs to communicate over the IR port. In the current version 2 kinds of LNP packets can be used. The first one is called integrity packets, which contain no addressing information, and is used for broadcasting packets. The second one is called the addressing packet, and is like an integrity packet augmented with addressing information, including a source and a destination address. The addressing packet provides an UDP like service, which does not guarantee arrival of sent packets, but guarantees errorfree packets. For further information please consult the above mentioned news group.

To communicate with a PC running Linux, a daemon called lnpd is used. Lnpd allows multiple clients to connect to an RCX running legOS. Furthermore, a library called libnp exists, so applications can connect with remote lnpd daemons.

3 Peripheral devices

LegOS provides interfaces to the various devices supplied with RCX brick, which includes motor control, sensor control, sound control, LCD display control, and button control.

3.1 Motor control

This section explains the way motors/output ports are handled. The nature of the motors are also briefly discussed.

3.1.1 The motor hardware

The motors are controlled by memory mapped I/O. All 3 motors are controlled by writing in the byte at address 0xF000.

In legOS the state of a motor is characterised by a direction and a speed, and the direction setting determines the motor behaviour when it receives a pulse. Direction forward corresponds to the bit pattern 01, reverse to 10, stop to 00 and brake to 11. When in brake mode, the motor will hinder rotation, whereas stop mode allows the motor to turn freely. Furthermore, an 8 bit speed and an 8 sum variable are used to control the frequency by which the direction bits are written to the memory which controls the motor.

3.1.2 Motor handler function

The motor handler, dm_handler, is called by the system timer interrupt handler, and it implements Bresenham’s linear drawing algorithm. The 3 motors are handled in turn. In the motor state, the sum variable is incremented with the motors speed setting. If the sum overflows, the direction bits for the motor will be written to the 2 bits controlling the motor, otherwise zeroes are written. Thus, if a motor has a speed setting of 255, it will overflow every time, and the motor will be driven every millisecond.

3.1.3 User interface to motor control

The motors are named A,B, and C corresponding to output port A,B and C on the RCX brick. Each motor (or output port) is controlled by the 2 functions motor_X_dir and motor_X_speed, where X is a,b or c. Motor_X_dir takes an enumerated type as argument, which has the values off, fwd, rev and brake. Motor_X_speed takes an integer argument between MIN_SPEED and MAX_SPEED, which is 0 and 255 respectively. Using these functions updates the variables, used by the motor handler. Motors are initialised by calling the motor shutdown function, which just set all motor directions to off.

3.2 Sensor control

This section explains how the sensor values are sampled by the on-chip A/D converter, to provide values on the RCX input ports.
3.2.1 The on-chip A/D converter

The H8/3292 includes a 10-bit successive-approximation A/D converter with a selection of up to 8 analog input channels. The 8 analog input channels are named AN0-AN7. The A/D converter has a set of memory mapped registers, mapped to the addresses 0xFFE0-0xFFE7. 4 16 bit data registers named ADDRA-ADDRD contain the result of A/D conversion of up to 4 input channels. The data registers are read only and are split in a high and a low part when mapped to memory. The control status register (ADCSR) is an 8 bit read/writeable register used for controlling the A/D converter. ADCSR is mapped to address 0xFFE8. When a conversion of an analog signal has completed, an interrupt (ADI) can be requested [1].

3.2.2 Initialisation

The sensors provide data for the 3 input ports on the RCX. The sensor values are obtained by converting analog values on the sensors to digital numbers with the on-chip A/D converter. Only the first 4 of the 8 possible input channels are used, and the setup is as follows: AN0 is used for input port 3 (on the RCX), AN1 for input port 2, AN2 for input port 3 and AN4 for reading the battery level. The A/D converter is initialised by the function ds_init during kernel initialisation.

First the power output to the 3 sensor ports is activated, and the A/D converter interrupt (ADI) vector is set to point to ds_handler. External triggering of the A/D is disabled, and the status control register ADCSR is set. This setting implies singlemode operation with a conversion time of 266 states. As the microcontroller is running with a system clock of 16 MHz, each of the 4 channels are estimated to be sampled 8 to 10 times per millisecond, when the interrupt handling overhead is taken into account.

3.2.3 The A/D interrupt handler

After initialisation, the A/D converter periodically samples the channels, and generates an interrupts handled by ds_handler. The job of ds_handler is to alternate between channels, handle rotation sensors and activate/deactivate output power to the sensors.

The value of the global variable ds_channel indicates the channel which number has been sampled, and the corresponding bit number in ds_activation indicates if this channel needs a power output to function properly. If this is the case, the power output to the sensor is activated. Some of the sensors are active sensors, needing power to operate, but the power must be cut off when sampling values. The power was cut off in the previous call of the handler and must therefore be turned on after the sampling. This is done by setting the bit corresponding to the channel number in I/O PORT 6. The bits in ds_activation are set with the inline functions ds_passive and ds_active defined in include/dsensor.h. The rotation sensors use the variable ds_rotation and the inline functions ds_rotation_on and ds_rotation_off.

If rotation sensor support is included in the kernel, and the current channel has a rotation sensor, the rotation sensor handler is next called. R0-r3 are saved before the call, and restored afterwards.

The next channel to be sampled is selected, and the power output to this channel is cut off. A busy wait for 32 states is done to allow the sensor to settle upon the power deactivation. The ADCSR is adjusted, so the new channel is to be sampled next and ds_handler returns.

3.2.4 Rotation sensors

Rotation sensors have their own handler function named ds_rotation, which is called by ds_handler. The variables mentioned here are actually arrays with 3 elements, where the element used corresponds to the current channel number.

The rotation sensor is calibrated by calling ds_rotation_set, which sets the currently sampled value as starting point, and afterwards a state machine is used for calculating rotations. The raw values sampled are mapped to the range 0x0-0xF, and this value corresponds to a state. Each of the 16 states is associated with a value between -1 and 3. These values are used to measure when the rotation counter should be increased by 1 or decreased by 1. I will not go into the details of the algorithm, and the source can be found in kernel/dsensor.c.
3.2.5 User interface to sensors

As the data registers of the A/D converter are memory mapped, their values are read directly using the macros SENSOR_1, SENSOR_2 and SENSOR_3 defined in include/dsensor.h. These are the raw digital values, and therefore wrapper macros are defined for the various sensor types.

3 light sensor macros, LIGHT_1-LIGHT_3 are defined in terms of the macro LIGHT. LIGHT_1 is used for reading the value of a light sensor on input port 1 and so forth. The LIGHT macro maps the raw light sensor sampling to a value from zero to light max/white, where zero is darkness/black.

In the same way, the touch sensors values are provided by the macros TOUCH_1-TOUCH_3. The values of the touch sensors are mapped to a binary value which is either 0 (not activated/touched) or 1 (activated/touched). The raw battery level value is provided by the macro BATTERY. The values of the rotation sensors are obtained by the macros ROTATION_1-ROTATION_3, which expand to indexing into the ds_rotations array.

3.3 Sound control

3.3.1 The sound handler

Control of the sound capabilities is handled by the sound handler. The sound handler works by playing notes and playing pauses, i.e. no sound. A note is a structure which specifies the node’s pitch and the node’s length. Valid node lengths are whole, half, quarter and eight. Where the length of 1/16th note is 200 milliseconds. The pitch specified will be translated to a frequency by indexing into an array called pitch2freq.

The sound handler is driven by calls from the system time handler. The sound handler mainly works with 2 variables, dsound_next_note and dsound_next_time, and a data array of nodes. The handler function starts by checking if note should be played, by comparing the system time with dsound_next_time. If the system time is greater than dsound_next_note, either an internote (pause) or a note is played, depending on the value of the internote flag. An internote is played every second time, and has a default duration of 15 milliseconds. An internote does not use data from the node array. The dsound_next_time is updated, and the internote flag is reset.

After a node has been played dsound_next_time and dsound_next_note are updated, and the internote flag is set to indicate that the next note to play is an internote.

3.3.2 Hardware issues

I will not go into details of exactly how the sounds are created, but I will list the basic principles. The source can be found in kernel/dsound.c.

A note is played by calling the inline function play_freq, which instructs the hardware through memory mapped I/O. Channel zero of the 8 bit on-chip timer module is used for playing the different sound frequencies on the speaker.

The sounds are created by letting the timer generate signals to the speaker. The argument to play_freq is a 16 bit frequency, of which the MSB byte is used to decide the clear value of the timer, and the LSB byte is used to select internal clock input to the timer.

3.3.3 User interface for the sound driver

A process plays sounds by making an array of nodes, and calling dsound_play with the array as parameter. Thus the task which calls dsound_play can continue doing other things, while the system timer interrupts drive the sound driver.

The sound playing is stopped when a null is encountered in the node array. This makes the handler function fall through to a call to dsound_stop. Dsound_stop plays a pause, and then resets the sound variables. Only one process at a time can use the sound device.

An array, dsound_system_sounds of predefined systems sounds exists, but currently only one sound, the system beep, has been defined.

3.4 LCD display control

3.4.1 The LCD handler

The LCD display is controlled by the LCD handler, driven by timer interrupts. The handler function lcd_refresh_next_byte is called every 6th milliseconds. As the LCD data is 9 bytes long, this results in a complete refresh every 54 milliseconds, or 18-19 refreshes per second.
2 arrays are used for the LCD data, `display_memory` defined by the ROM and `lcd_shadow` which contains the last displayed bytes. The shadow array is used when refreshing the display, to check if a refresh is necessary. Only altered bytes will be refreshed.

3.4.2 User interface to the LCD display

A manual refresh can be done by calling `lcd_refresh`, which will re-display all 9 bytes in `display_memory`.

In the file `include/conio.h` more functions for user control of the LCD display are declared. This includes `cputs` for displaying strings and `cputw` for displaying hexadecimal words. These functions are defined in terms of `cputc_native`, which displays single hex numbers and `cputc` which displays single ASCII values. Furthermore, `cls` is provided for clearing the display.

3.5 Button control

3.5.1 The button handler

The `dkey_handler` is called from the system time handler. It handles debouncing which means that the button state only will be checked after a certain debounce timer has elapsed, to avoid reading a push on a button several times. The debounce timer is set to 100 ms. Every time the handler function is called the debounce timer is checked, and only if it equals zero, the button state will be handled. Otherwise the handler just returns (bounces back).

3.5.2 Hardware issues

The buttons are connected to the I/O ports in the following way: The `on/off` button is connected to bit 1 of I/O port 4, the `run` button is connected to bit 2 of port 4, the `view` button to bit 6 of port 7 and the `prgm` button to bit 7 of port 7. The values of these 4 bits are inverted and used to form a 4 bit bitmask. The variable `dkey_multi` holds the last used bit mask, and `dkey` holds the current bit mask identifying the button(s) pressed. If `dkey_multi` and `dkey` are equal, `dkey` is not updated. Otherwise, both the current bit mask is assigned to both variables, and the debounce timer is reset to 100. Thus the variable `dkey` holds the information about the buttons which have been pressed, and it is the value of this variable which is returned by `getchar`.

3.5.3 The button user interface

The user interface for buttons is the function `getchar`, which waits until a button is pressed, and returns the value of the button pressed. The button values are defined by the following macros: `KEY_ONOFF`, `KEY_RUN`, `KEY_VIEW`, `KEY_PRGM` and `KEY_ANY`. The values can be bitwise ORed to detect multiple button presses.
References


