THOS

January 2013
Two Hour Operating System (Version v2013-01)

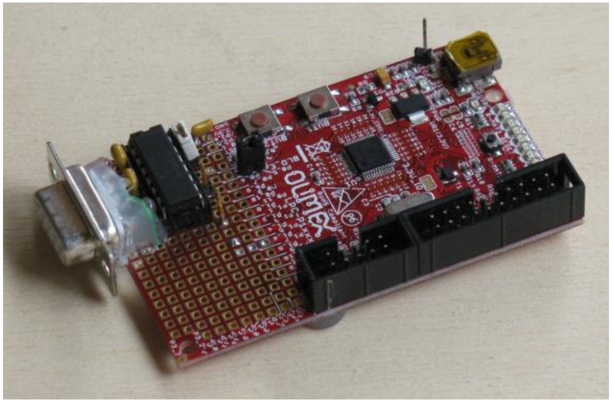
Alessandro Rubini (`rubini@gnudd.com`)

Introduction

This document accompanies source code for a simple operating system for micro-controllers. The OS uses cooperative multitasking and handles a few different tasks (but you can easily add more). The target CPU is the LPC1343 (Cortex-M3). The board I used is the one sold by Olimex for 15 EUR. but I had to add a level converter for the serial port. Actually, any board will work, but you, as a reader, are not actually expected to run the code because of lack of hardware, but please be assured that it works fine on my hardware.

In the future I may port the system to the *Versatile* board, in order to be able to run it on *Qemu*, without real hardware. No other changes to Thos are planned, because it must fit in two hours. (Once, in August 2012, I made an error with time slots and shrank my talk into one hour and 15 minutes, but it's not fun, that fast).

This project was born as a simplified spin-off of another OS that I am writing; Thos has been presented a few times in the form of a lesson. Initially I wrote it for ARM7 (using the *Christmas Tree* device), then I've redone it on a Cortex-M3, since the *tree* is not for sale any more and the cheap ARM7 boards are now more expensive than the cheap Cortex ones, so the current hardware I'm using is the board depicted below.



This document describes the whole writing of the OS, starting from an empty directory. As a prerequisite, you are expected to know C language and have some experience with computer programming in general. Master or PhD students are usually able to understand and enjoy the Thos lesson.

The released package is a *git* tree, so you can get back to older releases in the published history and follow the code described in the text as it is being written – however, the document is committed after the code in the published history, so you need to keep a copy of the documentation before checking out individual chapters.

Please note that sometimes the choices being made are suboptimal, and I'm well aware of it. This material is really being shown in two hours, so many details have been ignored because of time constraints. I'm somehow shy of the suboptimal choices now that I put it all in writing, but I still want this material to be explained while being typed live in a single lesson.

1 Laying out the Makefile

As a first step in setting up a new package, we'll set up a *Makefile*, taking care of compilation of our sources, which are both C and assembly files. We'll use a simple *Makefile* (as simple as possible, as usual), so there is no support for dependencies or such stuff.

1.1 Cross-Compilation

The code will be cross-compiled: the target is ARM while the build system is a PC. So let's start by defining the tools used for cross-compilation: the established practice is defining `CC` and other variables based on an environment variable called `CROSS_COMPILE`. Thus, let's copy the usual stanza from the *Makefile* of any kernel version:

```
AS              = $(CROSS_COMPILE)as
LD              = $(CROSS_COMPILE)ld
CC              = $(CROSS_COMPILE)gcc
CPP             = $(CC) -E
AR              = $(CROSS_COMPILE)ar
NM              = $(CROSS_COMPILE)nm
STRIP           = $(CROSS_COMPILE)strip
OBJCOPY         = $(CROSS_COMPILE)objcopy
OBJDUMP         = $(CROSS_COMPILE)objdump
```

The cross-compiler we need to use is any *gcc* starting from version 4.3. If your compiler is older than that, it won't support the *armv7*, which is our target CPU. The most common choice nowadays is using the *lite* version of `codesourcery.com`. For example, you can download this file:

```
http://www.codesourcery.com/sgpp/lite/arm/portal/package8734/public/
    arm-none-eabi/arm-2011.03-42-arm-none-eabi-i686-pc-linux-gnu.tar.bz2
```

(you may look for a more recent version, but all of them are a dozen clicks away from the home page, so the direct link here may be useful).

The compiler can be uncompressed to any directory of your choice. For example, if you unpack in `/opt`, you'll need to run this command

```
export CROSS_COMPILE=/opt/arm-2011.03/bin/arm-none-eabi-
```

in order for the *Makefile* to build all names for your commands.

1.2 CFLAGS

The Cortex-M3 is only one of the possible flavors of the ARM processor, so you'll need to always pass `-march=armv7-m -mthumb` to both the compiler and the assembler. Thus, our *Makefile* will need the following two lines as well. The extra `-g` forces debug information to be generated, and `-Wall` forces all warnings to be reported; both are things you really can't live without:

```
CFLAGS  = -march=armv7-m -mthumb -g -Wall
ASFLAGS = -march=armv7-m -mthumb -g -Wall
```

With these variables in place, *make* is able to automatically create *file.o* if either *file.c* or *file.S* exist. We only need to declare our list of object files and a rule to link them; let's use the handy *make* shortcuts that name the target and the dependents, to avoid repetition of file names:

```
thos: boot.o io.o main.o
        $(LD) $(LDFLAGS) $^ -o $
```

1.3 A First Build

We'll now "`touch boot.S io.c main.c`" to at least be able to start a compilation. The choice of the file names reflects the need to have at least some operation running, with the necessary boot steps and minimal output operations in place.

With the *Makefile* and the empty files in place, we can issue *make*, which finally builds a binary with no actual contents, as all input object files are empty. It also spits a warning, explained in the next chapter:

```
ld: warning: cannot find entry symbol _start; defaulting to 00008000
```

The current code situation is committed in the repository, like I always do for every chapter and section of this document. The commit message is a one liner, naming the chapter (or section) where the code is introduced.

2 The Linker Script

To perform the final link step, we actually need a *linker script*. In other words, we must tell the linker how to lay out the memory map for the program.

2.1 Spelling the Object Format

When compiling your `hello.c` or equivalent program, you'll end up using the default linker script (usually in */usr/lib/ldscripts*), which is designed to take care of programs that are hosted within an operating system. When writing Thos we need to provide our own linker script, because our program must reside directly in the memory of our micro-controller and can't count on an host operating system.

The first three lines of our linker script are standard stuff, that I refuse to learn by memory and I simply copy over from another linker script; the only customized line here is the entry point, which we want to be `_thos_start`, a *symbol* that we'll define in our small assembly source file.

```
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_thos_start)
```

2.2 Placing Sections in Memory

After the header part, the script must list the ELF output sections, and the address in memory where they are placed. To this aim, we need to know the memory map for the target processor.

In the specific case (LPC1343), the processor has internal RAM and internal Flash; a user binary can be programmed to Flash memory using the USB port, or can be programmed to either RAM or Flash using a serial port. Since programming to RAM is easier (both the associated *linker script* and the startup code are easier), my choice here is programming to RAM, even if the serial port is not directly wired out in the evaluation board as sold.

This means you'll have to connect a serial port in some way to test the code shown here: the UART is really the easiest way to print some diagnostic message. This is unfortunate, but doing it differently would require a huge lot of time (remember, this is meant to happen in a lesson of two hours in total). Actually, the board where I initially wrote THOS had a usb-serial interface, so there it was not an issue.

If you get a copy of the LPC1343 manual, you'll find that RAM starts from address 0x1000.0000, and the first bytes of the RAM itself are used by the internal ROM during the loading procedure, so I chose to load the program 1kB within RAM – later, I'm going to use that spare kB for the stack.

The resulting linker script is as shown below. Note that *dot* (`"."`) is the point where output data is being written; thus, the initial assignment to *dot* forces the output to be placed at address 0x1000.0400. Starting at that address, the usual ELF sections are placed, in the usual order. The sections we need are the old known `.code`, `.data` and `.bss`, with the more recent addition of `.rodata`, where the compiler places constant string and similar material.

```
SECTIONS
{
        . = 0x10000400;
        .text : {
                *(.boot)
                *(.text)
        }
        .rodata : { *(.rodata) }
        .data : { *(.data) }

        .bss : {
                . = ALIGN(16);
                __bss_start = .;
                *(.bss);
                . = ALIGN(16);
                __bss_end = .;
        }
}
```

The only unusual detail in the script just shown, is the `*(.boot)` line. It says that the `.text` output section is built to include the contents of any `.boot` section appearing in input files followed by the contents of any `.text` input sections. This is how I force the contents of *boot.S* to be at the head of the output file.

The leading `*` in the definition of input sections means "all input files", but you could also write `start.o(.text)`. This is a matter of personal preference: I prefer to define initial code by putting it in a specially-named section instead of hardwiring the input file name in the linker script.

2.3 Using the Script

To put our linker script in action we need to set `LDFLAGS` in the *Makefile*:

```
LDFLAGS = -T thos.lds
```

Now compilation of the empty input ends with the following message, which confirms the linker script is being used:

```
ld: warning: cannot find entry symbol _thos_start; defaulting to 10000400
```

3 The Assembly Code

When the system begins running at power-on, there is no pre-built context at all. The only thing you can count on is the initial execution address, and you can place your assembly instructions there.

3.1 Naming an ELF Section

Before you are able to run code in a higher level language at system boot, you need to build your so-called *run-time environment*. Since C is a very low-level language (they say it's powerful like assembly and simple like assembly) you need two very simple things: a stack pointer and a zeroed BSS. Remember that the `.bss` section is allocated but not stored in the ELF binary: it must be zeroed at run-time.

Our first source code will be the minimal amount of assembly needed to perform those initial steps. We place the stack pointer before code (so the stack can grow in the kilobyte of space we left free at the beginning of RAM), and we zero the *.bss* ELF section by means of the symbols we defined in the linker script: *__bss_start* and *__bss_end*.

The code in *boot.S* is going to live in the section called *.boot*, so the linker script will place it before the default code section. This ensures that the first instructions in the binary file come from *boot.S*.

The first line of the file is thus as follows:

```
        .section .boot, "ax"
```

While ELF section names are arbitrary, it's an established convention to use names with a leading dot. Assembly directives have always been starting with dot, like *.section* above, so a casual reader can see at a glance whether a word unknown to her is an instruction or a directive. In the old ages, section names were fixed, and the directives in assembly files were just *.text*, *.data* and *.bss*; this convention of using dot-prefixed section names remained when arbitrary names were introduced.

The `"ax"` above means *allocate* and *executable*. The former flag is now implicit, but usually retained for compatibility with older tools; the latter flag is needed so ELF-reading tools know what the content is. In this case, we'll need to disassemble with `objdump -d`, and the tool refuses to disassemble data sections. In other execution environments (e.g., when the code is hosted in another OS) the flag may be used to set permissions on memory pages, so we may say that marking executable sections with `"x"` is mandatory, even if sometimes things may work nonetheless.

3.2 Actual ASM code

Within the *.boot* section we'll now place our code, labelling the first instruction as `_thos_start`, which is the *entry point* declared in the linker script. Even though the entry point is not actually needed in this project, by defining it we avoid the ugly warning message; again, in other execution environments defining the entry point is mandatory, especially when it doesn't mark the first byte of the binary file.

The leading underscore in the name of the entry point is another convention: symbols that are used in assembly files or other low-level implementations should not be referenced by higher-level code: the underscore is a signal that the symbol is somehow special and it's only relevant to people who hack the internals.

The code then will set the stack pointer and zero bss. The stack address is calculated by subtracting from the program counter, to save an explicit address in the code, the bss-zeroing loop is then trivial. You are not expected to be able to write assembly code for your target CPU, but you are always expected to be able to read it; fortunately all assemblers are similar, so this is rarely a problem.

```
        .global _thos_start
        .extern __bss_start
        .extern __bss_end
        .extern thos_setup
        .extern thos_main

        _thos_start:
        /* set the stack pointer */
                mov r0, pc
                sub r0, #12
                mov sp, r0

        /* clear BSS */
                mov r0, #0
                ldr r1, =__bss_start
                ldr r2, =__bss_end
        0:
                cmp r1, r2
                bge 1f
                str r0, [r1]
                add r1, #4
                b 0b

        1:      bl thos_setup
                bl thos_main
                b 1b
```

There are a few details worth explaining here, which are independent of the target architecture and assembly language:

.global

Any symbols defined in the assembly file are local by default. Here, we want *_thos_start* to be externally visible, because it is the entry point looked for by the linker script.

.extern

Any undefined symbol is considered *extern* by default, so all these *.extern* directives are actually optional. However, stating that we know these symbols are known to be missing from this very file makes it more readable.

local labels

In Unix assemblers, and numeric-only label is considered local, and is referenced like shown above: `0b` means "0 backward" and `1f` means "1 forward" – the nearest local label forward or backward from the instruction naming it. This means that you can use the same label name several times in the same source file, which is very useful when you write assembly code in C macros.

Finally, the magic `#12` constant is there for a strange reason, as any constant will actually work (even 0). In ARM, when an instruction reads the program counter, the value returned points two instructions after the current one. Here, the PC is read in the first machine instruction ever, so it points to the third instruction. If we want the stack to live before the code, we want to subtract three instruction lengths. In *thumb* (our case), instructions are 2-bytes long, but full ARM has 4-byte instructions, so 12 works in every case. Just copying the PC to SP will work as well, but you'll overwrite the first two instructions of your program with your stack.

3.3 Compiling boot.S

With the assembly file in place, compilation now fails with `undefined reference to 'thos_main'` and `undefined reference to 'thos_setup'`. That's expected: the two `.extern` symbols that are marked as *undefined* in `start.o` are not provided by other object files. The error message we get back reports the exact source line where the undefined reference is found, this only happens because we used `-g` in compilation flags.

To quickly fix two undefined symbols, let's simply define two empty functions, *thos_setup* in `io.c` and *thos_main* in `main.c`, as that's the place they'll finally live:

```
int thos_main(void)
{ return 0; }
```

The function returns an integer value even if at this point we know the caller is ignoring such return value. In general, it's good practice to always return an error code (instead of `void`) even in simple functions, because otherwise you risk to change it later, when complexity increases as development goes on.

At the end of this chapter we have a complete binary, which does nothing but at least compiles and has some real content (here and elsewhere, `morgana%` is the command prompt of my host):

```
morgana% nm thos | sort
10000400 T _thos_start
10000428 T thos_setup
10000438 T thos_main
10000450 B __bss_end
10000450 B __bss_start
```

4 Loading the Binary

With a “working” binary in our hands, it’s now high time to write it to the final hardware. The preferred way to do that is using the serial protocol offered by the internal ROM in the CPU.

4.1 Pin-strapping

When power is applied to the LPC1343, pin P0_1 selects whether the CPU executes the program stored in internal flash or it runs code stored in internal ROM. Such ROM can handle re-programming flash memory using the USB interface, or interact through the serial port. Such behavior is selected with pin P0_3: if it is low, the serial port is used.

Thus, in my board I added a jumper on P0_3 to force serial communication (the jumper on P0_1 was already provided by the manufacturer). With both P0_1 and P0_3 pulled low, you can write to RAM or flash using the serial port. The subdirectory *tools* of this package includes two programs: `program` and `progrom`, they write a binary to either RAM or flash (here called ROM for symmetry).

4.2 Creating a Binary File

The binary that must be programmed to the CPU, however, is not the ELF file, but a pure binary file, with no header or other extra information. To turn our ELF file into a binary we can use *objcopy*, with the following rule in *Makefile*:

```
thos.bin: thos
        $(OBJCOPY) -O binary $^ $
```

Note that it is possible for a loading program to directly use the ELF file: writing an ELF loader from scratch is a matter of a few hours, if you have previous experience half an hour may be more than enough. Most other tools, though, load a bare binary file, so I chose to do the same here; moreover, by having a binary file on disk you can easily compare it with a dump of your flash memory. For disassembly we’ll still need the ELF file, to see symbolic names in the code.

4.3 UART Programming

The tools can be compiled by running `make -C tools`, as I lazily won’t be changing our *Makefile* to build them. Finally, programming with the serial port and *tools/program* looks like the following (where `ostro%` is my shell prompt):

```
ostro% ./tools/program thos.bin
Opening serial port /dev/ttyUSB0
Forcing boot loader mode
Syncronizing... done
Identifying... done
part number: 3d00002b
LPC1343, 32kB Flash, 8kB RAM
size is 900
W 268436480 900
0
```

```
....................OK
```

After this, the program mirrors data from *stdin* to the serial port and from the serial port to *stdout*. This will be useful to see the output messages of *thos*, but at this point there is nothing to look at.

5 Serial Output

One advantage we achieve as a side effect of using the UART port for programming, is that the serial port has already been configured for 115200 baud. Thus, the first snippet of real code in the OS will be some output function. Let's write a simple *putc* and *puts*, both in the `io.c` file.

5.1 puts

The implementation of *puts* is trivial: it just calls *putc* for each byte until end of string:

```
void puts(char *s)
{
        while (*s)
                putc (*s++);
}
```

5.2 An I/O Model

As for *putc*, the function must write the next character to the TX register in the hardware, but only when there is no outstanding transmission, or the byte will be lost. By looking at the CPU manual, we'll easily find the `U0THR` register (UART 0 Transmit Holding Register) and the `U0LSR` register, with its `THRE` bit (Line Status Register, Transmit Holding Register Empty).

Once the register values are known, the code is pretty simple, however, we need to define our I/O primitives: a standardised way to access hardware registers. Here a few approaches are possible. Let's list a few possibilities

Defining register names like they were variables

This approach is pretty common in the small microcontroller-class operating system. The resulting code to write the Transmit Holding Register is somewhat like "`U0THR = c`". I personally dislike it because the casual reader should know in advance that `U0THR` is an hardware register, and hardware accesses are not generally visible in the code.

Using `readl` *and* `writel` *helpers*

This is the Linux approach. Every access to registers is performed by pointer and is consistently encapsulated in a special function. It is a very good option, and one worth following; it has the minor disadvantage that readers are expected to know the convention.

Using a `regs` *array to address register*

It is an alternative way to mark register access. While less widespread than the *readl/writel* helpers, I enjoy the fact it's immediately understood by the casual reader; moreover it helps perusing the linker script, which I find useful for teaching purposes.

This project, therefore, uses `regs` as an array of registers. The array is defined extern in a new header, `hw.h`, and exploits the fact that all registers are 32 bits wide; finally, is uses the standard sized type `uint32_t` from `<stdint.h>`:

```
#include <stdint.h>
extern volatile uint32_t regs[];
```

Here, the `volatile` attribute is needed to kill compiler optimisations over read/write operations concerning the array: every read or write operation that appears in source code will actually perform a read or write instruction in generated machine code. Clearly, also the other I/O approaches listed above hide a `volatile` keyword somewhere in their implementation.

5.3 Defining Registers

With `regs` properly declared, we need to define register names for our needs. Since the `U0THR` register lives at address 0x40008000 we'll want that exact number to appear in our `hw.h`, to ease people grepping for the symbolic name in the source code. The easiest approach in this case is arranging for `regs` to be 0; the actual registers and bits we need are then defined in this way:

```
#define REG_U0THR              (0x40008000 / 4)
#define REG_U0LSR              (0x40008014 / 4)
#define REG_U0LSR_THRE         0x20
```

Actually, an extra-abstract implementation will use `sizeof(regs[0])` in place of the explicit 4, but here I prefer to assume people knows 32-bit registers are 4-byte long.

The chosen implementation is only one of a number of options, but it has a few good points: it preserves the hex number that you find in the CPU documentation, re-stresses that such things are registers by using `REG_` as a prefix, and finally enumerates bits in the name space of the hosting register by using a second underscore.

5.4 putc

With the three definitions in place, our *putc* turns out to be like this:

```
void putc(int c)
{
        if (c == '\n')
                putc('\r');
        while ( !(regs[REG_U0LSR] & REG_U0LSR_THRE) )
                ;
        regs[REG_U0THR] = c;
}
```

The function as shown takes also care of newlines: whereas C code uses *newline* alone to mark newlines, the UART conventions want *return-newline*, so the fix is performed at output time, to save source code from this boring detail.

5.5 The regs Array

The last missing point here is the definition of `regs`. This is simply done in the linker script, by means of the following line:

```
regs = 0;
```

This is no different from the definition of `__bss_start` and `__bss_end`; the linker script is actually there exactly for this reason: defining symbols – in addition to placing code in memory.

5.6 Header and more CFLAGS

To close up this chapter with a working binary, we need a few more details, that deserve no discussion, being pretty trivial:

- Create `thos.h` with prototype for all functions.

- Include `thos.h` in all C sources, including `io.c` even if it doesn't call any external functions, to ensure the prototype matches actual code.
- Call *puts* from `main.c`, to exercise the code.
- Add `-ffreestanding -O2` to `CFLAGS` in *Makefile*.

Optimisations are generally needed to make the code better; with the current code the overall binary size is reduced from 240 to 176 bytes (although actual figures may differ according to the compiler being used).

The *freestanding* flag is need to avoid the following warnings:

```
thos.h:5: warning: conflicting types for built-in function 'putc'
thos.h:6: warning: conflicting types for built-in function 'puts'
```

The compiler knows what are the prototypes for a number of standard-compliant functions, so we must clearly state that we are not hosted in a Posix environment, thus the `-ffreestanding`.

5.7 Testing puts

With the code for this chapter, after programming we get an endless stream of:

```
The mighty Thos is alive
```

The *endless* part depends on the fact that *thos_main* returns, and our assembly code restarts the system after calling *thos_main*.

6 The Jiffies Variable

The system is now equipped with serial output, which is a great first step. We now need a way to handle timing.

6.1 The Timer Tick

The easiest way to handle time is having an integer variable that counts timer interrupts. For example once every 10ms. As a matter of facts, the Cortex family of processors has a specific timer-tick interrupt line designed for this specific aim.

Unfortunately, handling interrupts is not feasible in a single lesson, so we must look for alternatives. While reading the CPU manual, a programmer might notice that the timers of the LPC1343 have a 32-bit-wide prescaler; this means that the counter register itself can count as slowly as needed. For example it can count at 100HZ even if the input quartz we are using clocks the system as 12MHz.

6.2 Power-on the Timer

In this specific SoC, when the device is turned on most peripherals are powered off until we gate their clock (the UART was already turned on only because we used the serial port for programming: the internal ROM turned it on and configured it for us).

The first step before doing anything with the code is thus defining the registers and bits to configure the timer; let's turn on timer 1 (I plan to use timer 0 for an external buzzer I connected to pin 0_11):

This is the definitions for the needed registers, preserving the names found in the CPU manual:

```
/* clock control */
#define REG_AHBCLKCTRL         (0x40048080 / 4)
#define REG_AHBCLKCTRL_CT32B0  (1 << 9)
#define REG_AHBCLKCTRL_CT32B1  (1 << 10)
```

```
/* counter 1 */
#define REG_TMR32B1TCR          (0x40018004 / 4)
#define REG_TMR32B1TC           (0x40018008 / 4)
#define REG_TMR32B1PR           (0x4001800c / 4)
```

6.3 The HZ Macro and thos_setup

To make thing a little abstract, let's also define our frequencies in `hw.h`, so they can be readily changed when porting to a different board using the same CPU:

```
#define THOS_QUARTZ (12 * 1000 * 1000)
#define HZ 100
```

The explicit multiplication has been chosen for readability: the calculation is performed at compile time and you see at first sight that it is 12 millions – on the other hand, a row of 6 zeroes is confusing and you'll find yourself checking it over and over if you encounter a bug related to timing.

The choice of `HZ` as a short name for our ticking frequency is mimicking Linux, which is always a good choice given the huge amount of programmers already used to its conventions.

With those additions to our headers, we are ready to fill `thos_setup`, the function we already have, though empty, in `io.c`:

```
int thos_setup(void)
{
        regs[REG_AHBCLKCTRL] |= REG_AHBCLKCTRL_CT32B1;

        /* enable timer 1, and count at HZ Hz (currently 100) */
        regs[REG_TMR32B1TCR] = 1;
        regs[REG_TMR32B1PR] = (THOS_QUARTZ / HZ) - 1;
        return 0;
}
```

6.4 Defining jiffies

jiffies is now counting ad 100Hz. We only need to see it from C code: following Linux tradition, let's declare it in `thos.h` and define it in the linker script, with the following two lines, one per file:

```
extern volatile unsigned long jiffies;

jiffies = 0x40018008;
```

Note that we might have defined *jiffies* in terms of `regs`, without using the linker script again, or we could have used the name `REG_TMR32B1TC` by using the C preprocessor over the linker script. There are always several ways to achieve the same result, and you are free to change to code to test different implementations, this is only my personal choice for the sake of simplicity.

6.5 Using Our Time Facility

As a final step in this chapter , let's now exercise the *jiffies* variable: our main function can print its self-promotion string once per second:

```
int thos_main(void)
{
        unsigned long j = jiffies;
        while (1) {
                puts("The mighty Thos is alive\n");
```

```
                        j += HZ;
                        while (jiffies < j)
                                ;
                }
        }
```

The code, simple as it is, works perfectly once programmed to the target CPU.

7 The Task Model

The OS is now in good shape. It has messaging and a time source; it only needs a task model. To avoid introducing interrupts, which wouldn't fit in the available time, let's use cooperative multi-asking: each *task* is implemented by time-based *jobs* (the names being used here are common the real-time world). A job is simply a function that returns when done.

7.1 Defining a Task Structure

Considering that most tasks you need to accomplish can be reasonably modelled as a state machine, the job function can receive its current state as an argument and pass the next state as return value to the caller. The most generic state is a `void *`, because it can either be casted as a simple number or used to point to a more complex structure.

The data structure we need is therefore as follows, in `thos.h`:

```
        struct thos_task {
                char *name;
                void *(*job)(void *);
                int (*init)(void *);
                void *arg;
                unsigned long period;
                unsigned long release;
        };
```

The `name` field is there for informational purposes, `job` is the state-machine and `init` allows to decouple any setting up of the hardware.

The `init` function receives the same argument as the job itself so it can behave differently according to its context, the same effect can be achieved by passing a pointer to `struct task` (i.e. to itself). I discourage from being lazy in parameter passing, especially for functions that are not in the critical execution path. For the same reason, `init` returns `int`, so the system may be extended to handle initialization failures, should the need arise.

The `release` field is the release time for the job: it is initialised to the first activation time and then modified by the scheduler according to `period`, each time the job runs.

7.2 Stuffing Tasks in an ELF Section

The *thos* operating system is one of the conventional real-time operating systems that are built as a single binary images, which includes all compile-time defined tasks in a single blob.

The easy way to arrange compile-time static sets is by building arrays. Instead of making an array for a predefined maximum number of tasks, we'd rather make an array of just the right size. The best way to do this is by juxtaposing data structures and bless the boundaries of the data area with two symbolic names.

By building a `.task` ELF section we can easily achieve that. The *gcc* compiler offers the `__attribute__` construct to assign code or data to specific sections, overriding the default one (`.text` for functions and `.data` or `.bss` for variables);

To hide the hairy `__attribute__` syntax from the user, let's add this definition in `thos.h`:

```
#define __task __attribute__((section(".task"),__used__))
```

We need `__used__` in order to be able to declare the structure as `static` within the file where it is defined. As a side effect, if we forget to spell out our `__task` macro the compiler will warn about a static structure being defined but not used.

The linker script must collate all input `.task` sections into an output section, adding symbolic names in the usual form at the boundaries of such sections:

```
__task_begin = .;
.task : {*(.task) }
__task_end = .;
```

Finally, the two extern symbols are defined in `thos.h`:

```
extern struct thos_task __task_begin[], __task_end[];
```

7.3 Using Wildcards in Makefile

I am a lazy typist, and I get upset whenever i need to edit a file to activate a feature; so, I want my users to be able to just drop-in new tasks in the *thos* directory and have them run, without manually editing a task list. The ELF section helped, but there is still the `Makefile` to be edited.

In a burst of extra-laziness, let's use wildcards in the `Makefile`:

```
 TSRC = $(wildcard task-*.c)
 TOBJ = $(TSRC:.c=.o)

 thos: boot.o io.o main.o $(TOBJ)
```

This is a bad idea in many situations, because whenever you make a backup copy of a source file in the same directory, you'll find that both files get compiled and the linker complains about duplicate symbols. However, I find it handy in small projects, especially because I can drop-in a task that plays music without any edit.

The commit for this chapter also includes a `make clean` target, which is useful in a system with no support for real dependencies.

7.4 Three Simple Tasks

Let's write a few tasks that print strings on a timely basis. We can use a single job function, with a different argument. The source file is called `task-uart.c`:

```
#include "thos.h"
#include "hw.h"

static void *uart_out(void *arg)
{
 char *s = arg;
 puts(s);
 return arg;
}

static struct thos_task __task t_quarter = {
 .name = "quarter", .period = HZ/4,
 .job = uart_out, .arg = "."
};

static struct thos_task __task t_second = {
```

```
         .name = "second", .period = HZ,
         .job = uart_out, .arg = "S",
         .release = 1,
        };
```

The actual file includes two more tasks, one running every 10s and one running every minute. There's nothing strange in these few lines, with the exception of the `release` field.

The `release` field states when the task's job must be first invoked; if several jobs are released at the same time, the scheduler will have to choose the order of activation. By forcing a different activation time for each task we can release the scheduler from working on priorities. Once again, the choice is dictated by the lack of time – the two hours are almost over, so we must hurry to meet the deadline.

7.5 Writing the Scheduler

The scheduler is really simple: it must select which task is next to run, wait for its activation time to arrive, and call it. We put it inside `main.c`, after the *mighty* hello message:

```
        while (1) {
                struct thos_task *t, *p;

                for (t = p = __task_begin; p < __task_end; p++)
                        if (p->release < t->release)
                                t = p;
                while ((signed)(t->release - jiffies) > 0)
                        ;
                t->arg = t->job(t->arg);
                t->release += t->period;
        }
```

In the trivial code above, `t` is the selected task, and `p` is a pointer that scans the array.

The *jiffies* comparison needs a cast to `signed` because the values are unsigned: without the cast any value different from 0 would be considered greater than zero. Note that in Linux this detail is hidden in a `time_before()` macro, that also makes the code more readable. Had we used `jiffies < t->release` the code would have failed after 497 days; not a real problem here, but it's a good habit to always deal with overflows.

7.6 Task-set Initialisation

Something is missing here in the code just shown: we have no idea about what the initial value of *jiffies* is, and we are not allowed to write it (even if it was a software counter, modifying a variable that is being read by several users is bad practice and should be avoided as much as possible).

The solution is simple: just add "`jiffies + 2`" to all activation times before using them. Actually, we take a snapshot of *jiffies* in a new variable, called `now`, and then we add "`now + 2`", because *jiffies* may have been incremented between two reads:

```
        now = jiffies;
        for (p = __task_begin; p < __task_end; p++)
                p->release += now + 2;
```

We could add "`now`", but then the first activation time will already have expired, and we'd have wrong timing at each boot; not a real issue, but it is easily avoided. We could add "`now + 1`", which is expected to be in the future, but is may be in the past by the time we completed the loop. Using `now + 2` is surely in the future, even if *jiffies* is incremented immediately after we took our snapshot into `now`: the loop takes less than 10ms in a 12MHz CPU, however long the task list may be.

Finally, let's use the `name` field in the task list, this is a good diagnostic aid to verify that everything went in place despite use of an obscure ELF section. In this same loop we'll also call the *init* method of each task, if assigned:

```
for (p = __task_begin; p < __task_end; p++) {
        puts("Task: "); puts(p->name); putc('\n');
        if (p->init) p->init(p->arg);
}
```

This loop must run before the `release` times are updated, because UART output is slow: at 115200 baud we have output only 11 bytes per millisecond.

7.7 Running the Task Set

We are really done: let's just run `./tools/program thos.bin` to see that things do really work:

```
[...]
....................OK
The mighty Thos is alive
Task: minute
Task: 10second
Task: second
Task: quarter
.S
minute!
....S....S....S....S....S....S....S....S....S....S
....S....S....S....S....S....S....S....S....S....S
....S....S....S....S....S....S....S....S....S....S
....S....S....S....S....S....S....S....S....S....S
....S....S....S....S....S....S....S....S....S....S
....S....S....S....S....S....S....S....S....S....S
minute!
....S....S....S....S....S....S....S....S....S....S
....S....S....S....S....S....S....S....S....S....S
....S....S....S....S....S....S....S....S....S....S
```

8 GPIO Output

My major disappointment with the current code is that the OS provides for tasks as state machines, but the feature is not being used. There is also support for an *init* method, but nobody is using it.

This chapter adds a new task, one that flips 4 bits, each connected to a LED. Bits 0,1,2,3 of GPIO port 3 are connected to leds, so they are perfect for us.

By checking the CPU manual we find we need two new registers in `hw.h`, shown below, and one bit in the control register (`REG_AHBCLKCTRL_GPIO`), not shown here:

```
#define REG_GPIO3DAT             (0x50033ffc / 4)
#define REG_GPIO3DIR             (0x50038000 / 4)
```

The source file is called `task-led.c` and no other file is changed, as we have wildcard support in `Makefile`. The job runs 5 times per second and features a running light with a period of 1 second: 4 states turn on one led, in sequence, and the last state has all leds turned off.

My initial example was shorter (used 4 states, so no `if` nor `switch` was needed) but this is more realistic as a state machine:

```
static int led_init(void *unused)
{
        regs[REG_AHBCLKCTRL] |= REG_AHBCLKCTRL_GPIO;
        regs[REG_GPIO3DIR] |= 0xf;
        return 0;
}

static void *led(void *arg)
{
        int value, state = (int)arg;

        if (state > 4)
                state = 0;
        switch (state) {
        case 4:
                value = 0; /* all off */
                break;
        default:
                value = 1 << state;
                break;
        }
        regs[REG_GPIO3DAT] =  0xf & ~value;
        return (void *)(state + 1);
}

static struct thos_task __task t_led = {
        .name = "leds", .period = HZ / 5,
        .init = led_init, .job = led,
        .release = 10
};
```

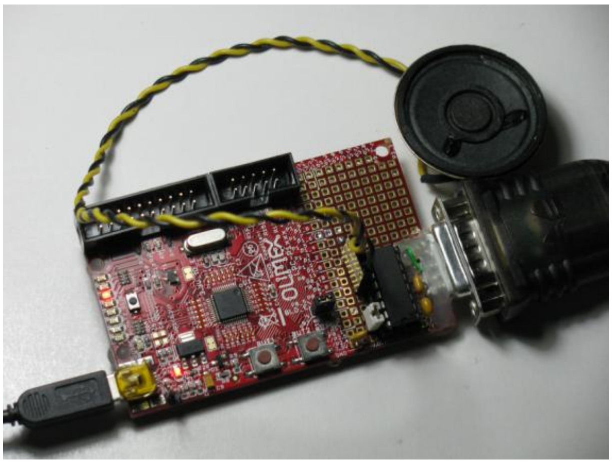
Needless to say, it works as expected, concurrently with the tasks that print messages to the serial port.

9 The Buzzer

The two hours of my lesson are now over, but there is no strict time limit in the written document. Thus, I'd like to introduce a new task, one that plays a tune on a buzzer. The new task is not adding anything to the OS core, it's just a new task in the set.

9.1 Buzzer Hardware

The hardware being used for playing music is a common buzzer or loudspeaker (one of those you can steal from a dead PC). It is connected to ground and the P0_1, where one of the PWM signals of timer 0 can be output.



9.2 Defining the Tune

The patch called “The Buzzer” in the repository adds a new file, `task-pwm.c` and augments `hw.h` with the register names we need to setup the counter 0. No other changes are needed.

The task is designed as a state machine referencing a global `tune` variable. The variable is a text string naming the notes and pauses; the current state of the state machine is a pointer to the next character in the string. When at end-of-string the pointer is reset to the beginning of the `tune` string. Note that this is suboptimal: the task code itself can run two state machines concurrently because `tune` is global. The correct fix would be in defining a structure with two fields as task status: one field should be the tune strings for the task and the other a running pointer to the current note.

The tune being played is the following string, which I suspect is a blatant violation of copyright on my side. Thus, sharing and distributing Thos in its current version is a crime, you are warned.

```
static char tune[] =
        "f f f a c c ccc d d d b ccc aaa "
        "f f f f a a a a g g g g fffff   "
        "                                ";
```

9.3 Defining the Notes

The notes themselves are identified by their frequency: knowing that a semitone is the twelfth root of two and A is exactly 440Hz, this is the definition of our notes:

```
#define HALF 1.059463094359295264555 /* exp(2, 1/12) */
#define TONE (HALF*HALF)

#define F    (G/TONE)
#define G    (A/TONE)
#define A    440.0
#define B    (A*HALF) /* moll */
#define C    (B*TONE)
#define D    (C*TONE)
```

Each note is output as a square wave. The counter is programmed to run a 4-cycles loop with a 50% duty cycle; the actual frequency is set by changing the prescaler register to reach the desired output frequency. Values to be written in the prescaler are stored in a table, built at compile time from the constant frequencies we defined earlier:

```
struct note {
        char name;
        int period;
};
```

```
#define PWM_FREQ (THOS_QUARTZ / 4) /* we make a 4-cycles-long pwm */

struct note table[] = {
        {'f', (PWM_FREQ/F) + 0.5},
        {'g', (PWM_FREQ/G) + 0.5},
        {'a', (PWM_FREQ/A) + 0.5},
        {'b', (PWM_FREQ/B) + 0.5},
        {'c', (PWM_FREQ/C) + 0.5},
        {'d', (PWM_FREQ/D) + 0.5},
        {0, ~0}, /* pause */
};
```

Our processor has no floating point unit, but the compiler converts the frequency values to integer when assigning the `period` field at compile time.

9.4 The PWM Task

Based on the table just shown and the tune string, this is the code of the task that plays the tune (I won't show the boring `pwm_init` function, which is part of the released code anyways):

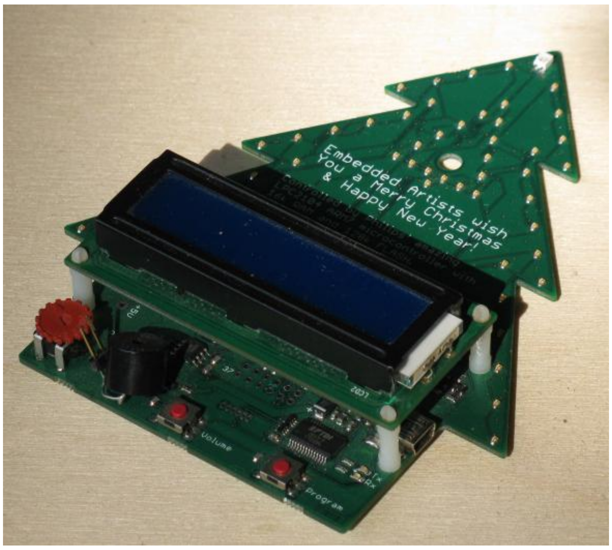
```
static void *pwm(void *arg)
{
        char *s = arg;
        struct note *n = table;

        if (!s || !*s) s = tune; /* lazy */

        /* look for freq */
        for (n = table; n->name && n->name != *s; n++)
                ;

        /* activate it by writing the prescaler and resetting the timer */
        regs[REG_TMR32B0PR] = n->period;
        regs[REG_TMR32B0TCR] = 3;
        regs[REG_TMR32B0TCR] = 1;

        return s + 1;

}
```

At the beginning of operation (when `arg` is null) and at end of string the `s` variable is reset to the beginning of our tune; then the table is looked up, and the last entry is used for every unkwnown note (the blank character thus count as pause, like any other non-note ASCII value).

The task is then defined as usual, running at a frequency of `HZ / 10`.

10 Thos on ARM-7

I initially wrote Thos on the LPC-2104 processor, in the "Christmas Tree" evaluation board.



After writing the LPC-1343 version, described in this document, I ported the code back to the Christmas Tree, and put it in the `Thos-2104` subdirectory. The code is slightly simpler, as the older device had less details to deal with. Besides, it had a serial port already connected, so using the UART for programming and messaging was definitely easier than it is now.

The main differences are in register names and values (i.e., `hw.h`). Other dissimilarities are as follows:

- The peripheral clock is 1/4 of the quartz frequency.
- There is no need to turn on the peripherals, as the are all on by default.
- The `CFLAGS` have no arch-specific flags.
- The buzzer has slightly different code.

The last item depends on the different kind of device being used: the buzzer you find on modern PC cases is a replacement for a loudspeaker, so you drive it with a square wave of the desired frequency. The one mounted on the Christmas Tree is self-oscillating, and ticks at a few kilohertz when powered on – for this reason the code is making a short positive pulse in the PWM output, changing the overall frequency.

As for programming, I ported the `tools/` programs back to ARM-7 where they were originally born. The current version works with both chip families, and can write either to RAM or to Flash memory, even though the code shown here only works from RAM – the LPC-21xx family has no USB-storage capabilities.

11 Thos on AVR (Arduino)

The directory called *Thos-avr* hosts a port to the AVR CPU. My actual target is the initial Arduino board, so compilation is specifically tailored to the ATmega8 implementation, featuring 8kB of flash memory and 1kB or RAM. Such port only includes the serial tasks, as I have no user leds and the hardware is not capable enough to play a tune using PWM using the simple code shown earlier.

There are a number of differences between the code i described in previous chapters and the one ported to the AVR. The differences are mainly due to the separation between program memory and data memory featured in this 8-bit CPU: we can't execute a program from RAM, so we boot from flash and we need to copy the data segment from flash to RAM before calling the main function. The same kind of code is also needed on the various ARM microcontrollers if you want to save your program to non-volatile memory; the original Thos avoided that complexity by running directly from RAM.

The other main difference is in how *jiffies* is managed: the CPU has no 32-bit timers, so I had to configure a timely interrupt to update a real variable accessed by C code.

This is the complete list of the differences between the top-level code and the one in the *Thos-avr* subdirectory:

- The linker script is more complex, because it must deal with separate program memory (flash) and data memory (RAM).
- The boot assembly code is longer, because it must copy the data segment to RAM before calling C code.
- `vectors.S` is an additional file. It defines the initial instructions of the binary image, i.e. the interrupt vectors. In a burst of laziness, I just made all interrupt vectors point to the procedure that updates the *jiffies* value.
- `jiffies.S` is another new file: it increments *jiffies* at each invocation. Such operation is made up of several additions, with carry propagation, because the CPU in itself offers 8-bit registers, with only a few 16-bit operations on pairs of registers.
- The `regs` variable is an array of `uint8_t` locations, because all I/O registers are 8 bits wide. (It is an array of `uint32_t` items in the ARM Thos.
- HZ is 244 and the tick doesn't even run at a sharp 244Hz. This depends on hardware limits in how the prescaler can be configured.
- The serial port runs at 38400 bits per second, not 115200. Again, this depends on the internal clocks we can use, and the choice is documented in the source itself. Configuration of the UART is done in the code itself, as we can't rely on a vendor-provided loader to configure it for us.
- The Makefile includes flags for ATmega8, and a new `load` target, to call *avrdude* with proper parameters. This matches my own setup and you may need to change it to fir yours.

On the other hand, `main.c` and `task-uart.c` are unchanged, so the basic layout of the OS is still what is described in the initial pages of this document.

12 The Future of Thos

In 2011, after writing this document, I felt I was almost done with Thos, unless there were bugs to fix (e.g., I fixed a major typo in original code after release 2011-04, and a student fixed some other typos and submitted a patch).

Back then I planned to port tot the Arduino (something which finally happened) and maybe also to a device that is emulated by *Qemu*, so you don't need real hardware to see the code running. This will happen, sooner or later (or I can accept patches for that).

But what actually happened is that I added some features and functionalities, that don't fit in the 2 hours at all, much less than the PWM task described earlier. The material is libraries (e.g. printf), drivers (such as GPIO and SPI) and tasks (a shell on the serial port, real I/O and possibly more).

While adding such code, I noticed how the simple code layout chosen here doesn't fit any more, even though the basic design remains the same and the main function (the scheduler) is unchanged. Initially I tried to bring on the master branch of this repository, but after a few days I realized it really doesn't work because there I'm interested more in the final result than in having a bugless history to be described commit by commit. So the new project, called Bathos (Born Again Thos), has a new repository, that continues development (and documentation) starting from release 2012-12 of Thos.

The links to the repositories are as follows: they are host both on gitorious and on my own system.

- `git://gitorious.org/rubi/thos.git`
- `git://gitorious.org/rubi/bathos.git`

- `git://gnudd.com/thos.git`
- `git://gnudd.com/bathos.git`

Hope you enjoyed this, and you'll get the incentive to write your own OS sooner or later. In the 20th century real people didn't call themselves programmers until they wrote their own editor; in this millennium writing an OS is easier than writing an editor, so you have no excuse – even if writing an HTTP server is still easier, if you can count on a working TCP implementation.

Table of Contents