MAKING A KENREL

Camilo Talero

Making a kernel

Contents

Part I

The tools

Introduction and Setup

1 Introduction

This text is intended as both a log for the PiOS project and as a reference guide for anyone trying a similar project in the future. All text here is pertinent to the Raspberry Pi model 3 B, as this is the model we are using, past and different models may have different architectures, and as such, certain sections fo this guide may be useless for those working on those machines. This is not a documentation manual, for the PiOS documentation, please refer to the PiOS documentation manual.

2 Hardware and Physical tools

To start creating a kernel for the Raspberry Pi, the following is needed:

- Raspberry Pi board (model 3 B is used in this document)
- Micro SD card
- Micro USB adapter and cable to power the PI (5V @ 2A recommended)
- An independent machine with an operating system to compile the source code (aka PC)
- A micro SD to USB adapter or any other method to store the compiled binaries into the micro SD card

The following is optional:

- Raspberry Pi case
- JTAG system for debugging
- A monitor compatible with the Pi's video core

3 Software

The following software is needed to create the kernel:

- Cross compiler to generate the binaries. Options include the GNU ARM Embedded Toolchain (used in this document)
  WARNING: old `gcc` versions can generate problems with function attributes, it's recommended to use the latest `gcc` version.

- Raspberry Pi firmware binaries. In order for the boot process to be executed correctly it is needed to have the bootcode.bin and start.elf binaries inside of the micro SD card.
- A text editor or IDE (no help will be provided as to how to setup an IDE)

It is also recommended that the reader has enough experience with C and ARM assembly, as it is assumed that the reader understands the syntax of these languages. An understanding of linker scripts is also helpful.

4 Compilation

The following are the bare minimum commands needed to build a working kernel image.

To compile C files:

```
arm-none-eabi-\notengl{gcc} -<opt> -<arch> <src> -o -c <o_file>
```

Where **<opt>** refers to the level of optimization, **<arch>** refers to the target architecture, **<src>** is the source file(s) and **<o_file>** is the generated object file. The -c argument tells the compiler to generate the .o file without linking, this argument is VERY important, and things won't work without it.

Example:

```
arm-none-eabi-\notengl{gcc} -O0 -march=armv8-a source/MainFiles/PiTest.c
-nostartfiles -c -o objects/MainFiles/PiTest.o
```

To compile ARM assembly files:

```
arm-none-eabi-as -<arch> <src> -c -o <o_file>
```

The flags here are the same as above, except that **<src>** should be an ARM assembly file instead of a C file.

Example:

```
arm-none-eabi-as -march=armv8-a source/boot/boot.s -c -o
objects/boot/boot.o
```

To link all object files:

```
arm-none-eabi-ld <o_file(s)> -o <elf> -T <linker(s)>
```

Where **<o_files>** refers to all object binaries that will makeup the final kernel image, **<elf>** refers to the output .elf file and **<linker(s)>** refers to all linker scripts (normally just 1), needed to link the objects.

Example:

```
arm-none-eabi-ld ./objects/MainFiles/PiTest.o ./objects/boot/boot.o  -o
build/kernel.elf -T ./source/kernel.ld
```

To extract the raw image binary:

```
arm-none-eabi-objcopy <elf> -O binary <image>
```

Where **<elf>** is the .elf file created on the above step and <image> is the final kernel image binary.

Example:

```
arm-none-eabi-objcopy build/kernel.elf -O binary kernel.img
```

To disassemble the final .img binary for debugging:

```
arm-none-eabi-objdump -D <elf>
```

Although not necessary for compilation, it is recommended to execute and store the output of this command in a file, as it is very helpful for debugging and verifying the correctness of the final binary (.img).

Example:

```
arm-none-eabi-objdump -D build/kernel.elf > logs/kernel.list
```

For this project we also made use of C++, mainly to have the benefit of function overloading. However g++, the GNU C++ compiler, which is used in this document, enables exceptions by default, which may cause some odd errors about certain functions not being found. In order to prevent this error from happening an additional argument must be given to the g++ compiler to disable exceptions. The additional argument is:

```
-fno-exceptions
```

We will go into more detail in a future section.

For more information refer to the official GNU documentation for your current version (or to the documentation of your toolchain).

5 First Program

After all prerequisites are met, it is necessary to verify that things work properly. This section simply provides a minimal example to test that all requirements were installed correctly. A more in depth explanation is provided on the Mailbox section.

To test whether everything is working fine, we just want to turn on the Pi's ACT LED on, however on the Pi 3, the led is not connected to the GPIO lines, so we have to communicate with the video core through the mailbox to turn it on. To be extra-sure everything is fine, we want it to blink. The following C code is used to send the appropriate message to the mailbox to turn the LED on.

```c
#include <stdint.h>

#define REGISTERS_BASE 0x3F000000
#define MAIL_BASE 0xB880 // Base address for the mailbox registers
// This bit is set in the status register if there is no space to write
   into the mailbox
#define MAIL_FULL 0x80000000
// This bit is set in the status register if there is nothing to read
   from the mailbox
#define MAIL_EMPTY 0x40000000

struct Message
{
   uint32_t messageSize;
   uint32_t requestCode;
   uint32_t tagID;
   uint32_t bufferSize;
   uint32_t requestSize;
   uint32_t pinNum;
   uint32_t on_off_switch;
   uint32_t end;
};

volatile struct Message m =
{
   .messageSize = sizeof(struct Message),
   .requestCode =0,
   .tagID = 0x00038041,
   .bufferSize = 8,
   .requestSize =0,
   .pinNum = 130,
   .on_off_switch = 1,
   .end = 0,
};
```

```c
/** Main function - we'll never return from here */
int kernel_main(void)
{
   uint32_t mailbox = MAIL_BASE + REGISTERS_BASE + 0x18;
   volatile uint32_t status;

   while(1)
   {
      do
         status = *(volatile uint32_t *)(mailbox);
      while((status & 0x80000000));

      *(volatile uint32_t *)(MAIL_BASE + REGISTERS_BASE + 0x20) =
         ((uint32_t)(&m) & 0xfffffff0) | (uint32_t)(8);

      int i=0;

      while(i<0xF0000)
         i++;

      if(m.on_off_switch == 0)
         m.on_off_switch = 1;

      else
         m.on_off_switch = 0;

      m.requestCode = 0;
      m.requestSize = 0;
      m.pinNum = 130;
      m.end = 0;

      do
         status = *(volatile uint32_t *)(mailbox);
      while((status & 0x40000000));

      uint32_t temp;

      do
         temp = *(uint32_t *)(MAIL_BASE + REGISTERS_BASE);
         temp = temp & 0xF;
      while(temp != 8);
   }
}
```

As big as this seems for a "hello world!" example, it is the smallest C code we could come up with that light the LED. And we are not even done yet. This code is using memory, and if one where to disassemble the binaries, one could see that the sp register (stack pointer). Is used, as such we need to initialize it properly.
The following assembly initializes the stack pointer and branches to our main loop:

```
.section .init
.global _start

_start:
   ldr sp, =8000
   b kernel_main
```

However we still are not done. The Raspberry Pi always begins execution at address `0x8000`, so whatever instruction is at that address will be the first to run. We need to ensure this instruction is also the first instruction in the above code. For this we use the following linker file:

```
SECTIONS
{
   .init 0x8000 :
   {
   *(.init)
   }

   .text :
   {
   *(.text)
   }

   .data ALIGN(0x20) :
   {
   *(.data)
   }
}
```

WARNING: Depending on the compiler, sections of the code could be optimized out, be wary of compiler optimization options.

If the reader doesn't fully understand what the C code is doing, it will be further explained in the following sections. However the functioning of linker scripts, or assembly language will not be explained.

I — Understanding Compilation

1 Programming Languages

Here we will discuss some important details of the different languages we will use in order to understand certain design choices we will later make.

1.1 ARM

Although the least used of the 3 languages, ARM assembly is the most important of the 3 languages used, since all of our source code will eventually become arm assembly code before being turned into machine code (a compiler, such as gcc, first turns the source code into assembly code, and it then calls an assembler to translate this assembly code into binary code).

Specifically we care about the following things:

First there are hardware specific things that are not available at the C level and so we must use assembly to interact with them. Some examples of things that are not exposed to C are registers, special instructions like the "change program state" `cps` instruction, special registers like the stack pointer, link register, current program status register... and others.

Next we need to be aware that at the end of the day an assembly instruction in arm is nothing but a bit string, a sequence of 1's and 0's. As such there is no difference between an instruction and data as far as the CPU is concerned. For this we must be very careful to ensure that the program counter never holds a value that does not correspond to an instruction.
For example the value `0x00000000` corresponds to the arm instruction `andeq r0, r0, r0` and the value `0xe2833001` corresponds to the instruction `add r3, r3, #1`.
In addition to this, some values have no CPU instruction associated with them, they are undefined instructions and they will trigger an exception if loaded into the program counter.
The program counter always loads the instruction at the next `word` (i.e pc+4) so to prevent unexpected values from being loaded into the program counter we purposely add infinite loops at critical sections of the code.

1.2 C

The first of the 2 high level languages that we use. There just a few things that we need to understand about the use of C in the project.

First, C is very close to assembly. In fact, someone with enough experience could probably predict how a given C function will be compiled. This is important because we need to understand how C will be compiled into assembly to prevent errors and maximize our use of C. A common example is the fact that one can use pointers and structure definitions to abstract a given memory mapped peripheral such as the system timer. This is mostly self evident in the syntax so we will explain it further on the relevant sections.

Second we must be aware that all of our code lives in ram memory. As such the only thing differentiating a structure from a function is the code itself. A struct should never be used as a function, although it is perfectly possible to trick the C compiler to do it. So, as stated before, it is important to be aware that this high level abstractions, although convenient, don't exist once we are actually running our code. As an example in this comparison A function and an array get compiled identically to the same binary executable.

Function

```
void example()
{
   int i = 0;

   i = 500;
}
```

Array

```
unsigned int example[] =
{
   0xe52db004,
   0xe28db000,
   0xe24dd00c,
   0xe3a03000,
   0xe50b3008,
   0xe3a03f7d,
   0xe50b3008,
   0xe1a00000,
   0xe28bd000,
   0xe49db004,
   0xe12fff1e,
};
```

```
00000000 <example>:
0:    e52db004       push    {fp}
4:    e28db000       add     fp, sp, #0
8:    e24dd00c       sub     sp, sp, #12
c:    e3a03000       mov     r3, #0
10:   e50b3008       str     r3, [fp, #-8]
14:   e3a03f7d       mov     r3, #500
18:   e50b3008       str     r3, [fp, #-8]
1c:   e1a00000       nop
20:   e28bd000       add     sp, fp, #0
24:   e49db004       pop     {fp}
28:   e12fff1e       bx      lr
```

```
00000000 <example>:
0:    e52db004       push    {fp}
4:    e28db000       add     fp, sp, #0
8:    e24dd00c       sub     sp, sp, #12
c:    e3a03000       mov     r3, #0
10:    e50b3008       str     r3, [fp, #-8]
14:    e3a03f7d       mov     r3, #500
18:    e50b3008       str     r3, [fp, #-8]
1c:    e1a00000       nop
20:    e28bd000       add     sp, fp, #0
24:    e49db004       pop     {fp}
28:    e12fff1e       bx      lr
```

1.3 C++

This is the final language that is used in this project. Although it is almost identical to C there are a few differences that need to be addressed to prevent errors due to incompatibility or missing libraries.

First of all, unlike gcc, g++ enables exception handling by default. The most important aspect for us of this exceptions is the fact that the compiler assume s the existence of libraries and functions to handle an exception. It is possible that the code compiles just fine, but it's also natural that at some point during the compilation problem one sees at least one of the following errors:

```
undefined reference to '__aeabi_unwind_cpp_pr0'
```

```
undefined reference to '__aeabi_unwind_cpp_pr1'
```

Stack unwinding is used by exceptions, debuggers (e.g GDB) and by any program trying to display the call chain. In ARM this is done with the use of the sections **.ARM.exidx** and **.ARM.extab** . Although understanding this process is certainly important for the OS development it is not relevant to this section, but if one wishes to learn more there are external resources.

Using compilation options such as **-nostdlib** , **-nostartfiles** and/or **-nodefaultlibs** won't get rid of the error. However we cna prevent the error from happening very easily by including the option:

```
-fno-exceptions
```

Into our g++ command line parameters, as previously discussed.

The other important aspect of C++ is name mangling. Unlike C, C++ has object orientation and function overloading and overwriting. In C only one function definition with a given label may exist (such as printf()), but C++ allows us to define multiple functions with the same name as long as their signatures are different. In example one may define:

```
void function(int num)
{
   /* code */
}
void function(char character)
{
   /* code */
}
```

In the same C++ source code with no problems. This is all due to name mangling.

In reality 2 functions cannot share the same name, but C++ goes around this restriction by changing the names of functions and adding extra information such as the number and type of parameters to a functions name, thus guaranteeing that functions with the same name but different signature will be considered as different labels by the assembler.

Take for example the following function, which is defined in a C file (i.e .c extension). We can see how differently the gcc and g++ have compiled the funtion label:

Source

```
void function()
{

}
```

gcc

```
00000000 <function>:
0:   e52db004        push
{fp}
4:   e28db000        add
fp, sp, #0
8:   e1a00000        nop
c:   e28bd000        add
sp, fp, #0
10:   e49db004        pop
{fp}
14:   e12fff1e        bx
lr
```

g++

```
00000000 <_Z8functionv>:
0:   e52db004        push
{fp}
4:   e28db000        add
fp, sp, #0
8:   e1a00000        nop
c:   e28bd000        add
sp, fp, #0
10:   e49db004        pop
{fp}
14:   e12fff1e        bx
lr
```

2 Language integration

The reason why we care about this things is because we are integrating all 3 languages and we are using C and Assembly, which do not name mangle. This means that these languages would be unaware of the existence of C++ functions, mainly because if one writes:

```
void main(void)
{
   cppFunc();
}
```

In a C program, where **cppFunc()** is a C++ function, the C program expects exactly the label **cppFunc**, but due to name mangling the function will have a different name as we have already seen. There are 2 things we will do to make the

3 languages work together.

First, although it is possible to use a combination of gcc and g++ to compile source code, for simplicity we decided to use g++ as a compiler for both C and C++ source files, this fully integrates C and C++, since the g++ compiler understands name mangling, and as such, even if a C++ function is called in a C file, it will compile properly, which is easier than working with 2 compilers at a time.

Secondly, the assembly code however doesn't need to be compiled, only assembled. This is done by invoking the arm assembler, so the problem of name mangling appears once more. An instruction such as **b func** won't be assembled correctly if **func** is defined in a file compiled by the g++ compiler, as this label will be changed by the compiler. Fortunately there is a way around this. the use of the **extern "C"** key **word** allows us to tell the g++ compiler to avoid name mangling a specific function or set of functions. There are 2 ways of using it:

```
extern "C" void function()
{
   /* code */
}
```

```
extern "C" {
void function()
{
   /* code */
}
}
```

For the right option, any function declared inside of the curly brackets will not be named mangled, so it is very suitable to use this in header files. No matter which of the 2 we use however, the function will be labeled as is, so in both cases the generated assembly code would have this function defined as **function** . There is an alternative which is to see how functions get mangled by g++ and then use those labels in the assembly code, but this is tedious and has the issue that if we change a function's signature, the assembled label will change as well and we would have to change the assembly code as well, which is inefficient.

3 Compilation

Although briefly, we need to discuss compilation especially because we will be making our own compilation script.
The compilation process is fairly straightforward, and can be summarized as follows:

1. The macroprocessor replaces all macros: At this point all **#define** , **#includes** and other macros get replaced. The only thing we are worried about here is to make sure the proper include guards are defined in all header files to prevent redefinitions of symbols.

2. The compiler generates assembly code: At this stage the compiler will take all

macro replaced temporal files and will generate assembly code for the target architecture.

3. The compiler assembles the assembly code: At this stage the compiler invokes the assembler, which takes the temporal assembly files and generates a permanent, final binary file (usually object files with the .o extension).
4. The linker links everything together: At the final stage, the linker is invoked and takes all object files and combines them together to create a final binary executable.

The reason why we must understand the compilation process properly, is because we will not be using the default linking script that g++ uses, but rather we will make our own. This is because we need to create certain labels, define some custom, special sections, and discard unused sections that may be created by default but that we are not using to reduce the final size of our executable. This will become apparent in future sections.
The final linking script looks like:

```
SECTIONS
{
   .init 0x8000 :
   {
      KEEP(*(.init))
   }

   .text :
   {
      . = ALIGN(4);
      __text_start__ = .;
      *(.text .text.* .gnu.linkonce.t.*)
      . = ALIGN(4);
      __text_end__ = .;
   }

   .data :
   {
      . = ALIGN(4);
      __data_start__ = .;
      *(.data .data.* .gnu.linkonce.d.*)
      . = ALIGN(4);
      __data_end__ = .;
   }

   .bss :
   {
      . = ALIGN(4);
      __bss_start__ = .;
      KEEP(*(.bss))
      . = ALIGN(4);
      __bss_end__ = .;
   }
```

```
    /* Made by LDB */
    .stack :
    {
       . = ALIGN(8);
       __stack_start__ = .;
       . = . + 512;   /* fiq stack size */
       __fiq_stack = .;
       . = . + 16384; /* usr & sys stack size (common) */
       __usrsys_stack = .;
       . = . + 16384; /* svc stack size (start-up) */
       __svc_stack = .;
       . = . + 4096;  /* irq stack size */
       __irq_stack = .;
       . = . + 512;   /* mon stack size */
       __mon_stack = .;
       . = . + 512;   /* hyp stack size */
       __hyp_stack = .;
       . = . + 512;   /* und stack size */
       __und_stack = .;
       . = ALIGN(8);
       __stack_end__ = .;
    }
    /* end of LDB contribution */

    .Heap :
    {
       . = ALIGN(4);
       Kernel_End = .;    /* Label to mark end of kernel code*/;
    }
 }
```

Co-Edited with Leon de Boer

Which simply defines an **init** section for second stage booting the standard **data**, **text** and **bss** sections, allocates some RAM space for the different stacks of the exception modes since each uses a different stack pointer and then finally creates a label to know where the end of the kernel space is and where the start of the heap space begins.

Part II
Making the kernel

II — Booting

1 Raspberry Pi firmware

As mentioned in a previous previous section, the first steps of booting are done with the help of some firmware. Basically the Video Core is responsible to do the first stage booting, which will initialize one of the cores and load the kernel binary into the core to begin execution.

We could customize this process with he help of a configuration file called config.txt which would do help configure the hardware; for example we could select a custom file as the selected kernel instead of using the default names. However we will leave it aside for now. More information about the config.txt file it can be found in the offical site of the Pi organization.

What we care about mostly, is that the name of our kernel binary affects how it is loaded into RAM. If we don't use the configuration file then, for the Raspberry Pi 3, the firmware will look for a list of file names in a priority order to load into RAM, the order, from first to last is: kernel8.img (boots into 64 bit mode, all other boot into 32 bit mode), kernel8-32.img, kernel7.img, kernel.img. This is simply so that one may load all 3 different kernel versions into the same SD card and all models of the pi would find their corresponding kernel image and load that one, for simplicity purposes however and since we are only working on a pi 3, we named our kernel image kernel.img.
No matter the kernel version, without a configuration file they all start executing whatever instruction is loaded at RAM address `0x8000`, which is why we made sure in our linker script to load the `init` section at that address.

2 Setting up the hardware

At this point the firmware has already finished loading the kernel image executable into ram and has initialized the program counter to the appropriate value (`0x8000` in our case). However the Video Core has done the minimum work to set up the hardware, only one core has been properly initialized, the floating point unit has not been set up... So we need to make sure to initialize everything properly ourselves.

2.1 Installing the Interrupt Vector Table (IVT)

The first thing we want to do is initialize the Interrupt Vector Table. In the Raspberry Pi 3, when an exception is triggered, a specific address's content gets loaded into the program counter, although it is actually possible to specify the location of the IVT, by default this is it's configuration:

RAM address	Exception Type	Execution Mode
0x00	Reset	Supervisor
0x04	Undefined Instruction	Undefined
0x08	Software Interrupt	Supervisor
0x0C	Prefetch Abort	Abort
0x10	Data abort	Abort
0x14	Reserved	(Reserved for future expansion)
0x18	Interrupt (IRQ)	IRQ
0x1C	Fast Interrupt (FIQ)	FIQ

Due to the fact that there is no space in between each table entry, the table consists of branch instructions that jump to a subroutine in a different section of memory, and this subroutine handles the exception and returns if and when appropriate.

We will talk more about this in the Interrupts section, for the moment all that we want to do is to make sure the correct values are installed in these first 8 **word**s in RAM.
The assembler considers labels to be relative to the PC position, thus if we simply try to load something like **ldr pc, =exception_handler** into the vector table, there will be a problem as this instruction will look like **ldr pc, [pc, #offset]**.
The way around it is to create a set of labels containing the actual RAM positions of the exception handling routines immediately after the branch instructions and load their values into low memory as well to keep the same relative offset. In other **word**s, in our data section we define this structure.
And then we simply load both the instructions and the subroutine addresses together into low memory as follows:

```
_start:
_reset_:
    ldr     r0, =_v_table
    mov     r1, #0x0000
    ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
    stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
    ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
    stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
```

Code taken from the Valvers Ttorial

```
_v_table:
   ldr    pc, _reset_h
   ldr    pc, _undefined_instruction_vector_h
   ldr    pc, _software_interrupt_vector_h
   ldr    pc, _prefetch_abort_vector_h
   ldr    pc, _data_abort_vector_h
   ldr    pc, _unused_handler_h
   ldr    pc, _interrupt_vector_h
   ldr    pc, _fast_interrupt_vector_h

_reset_h:                           .\compl{word}  _reset_
_undefined_instruction_vector_h:  .\compl{word}
   undefined_instruction_vector
_software_interrupt_vector_h:     .\compl{word}  software_interrupt_vector
_prefetch_abort_vector_h:         .\compl{word}  prefetch_abort_vector
_data_abort_vector_h:             .\compl{word}  _reset_
_unused_handler_h:                .\compl{word}  _reset_
_interrupt_vector_h:              .\compl{word}  interrupt_vector
_fast_interrupt_vector_h:         .\compl{word}  fast_interrupt_vector
```

Code taken from the Valvers Ttorial

Once this is done, the next step is to setup the execution mode. The Pi boots into Hypervisor mode, however although this mode has the highest privilege level, it cannot do certain things. Normally when we want to change modes we would use the **CPS** (Change Program State) instruction, but this will trigger exception on the Pi 3 when trying to switch out of Hypervisor mode, the way to actually switch to Supervisor mode was generously provided provided by Leon de Boer as the following subroutine:

```
RPi_CheckAndExitHypModeToSvcMode:
mrs r0, cpsr              ;@ Fetch the cpsr register
and r1, r0, #0x1F         ;@ Mask off the arm mode bits in register
cmp r1, #0x1A             ;@ check we are in HYP_MODE AKA register reads 1A
beq .WeHaveHyperMode
mov r0, #0;               ;@ return false
bx lr                     ;@ Return we are not in hypermode
.WeHaveHyperMode:
bic r0, r0, #0x1F         ;@ Clear the mode bits
orr r0, r0, #0xD3         ;@ We want SRV_MODE with IRQ/FIQ disabled
mov r1, #0                ;@ Make sure CNTVOFF to 0 before exit HYP mode
mcrr p15, #4, r1, r1, cr14 ;@ We do not want our clocks going fwd or bwd
orr r0, r0, #0x100        ;@ Set data abort mask
msr spsr_cxsf, r0         ;@ Load our request into return status register
mov r0, #1;               ;@ return true

.long 0xE12EF30E              ;@ "msr ELR_hyp, lr"
.long 0xE160006E              ;@ "eret"
```

Code provided by Leon de Boer

It can be summarized as, checking if the CPU is running in Hypervisor mode, if it is then we switch to supervisor mode. The exact details as to why this must be done in this specific way are not discussed here.

2.2 Initializing the stacks

Now that we are in Supervisor mode, we can properly set up the stacks of the different execution modes (On the Pi 3 this is not possible in Hypervisor mode).

Each of the execution modes has a different stack pointer, so we need to ensure that each of these points to a reserved location in RAM, that is big enough to hold the stack for that execution mode. In other **word** s things like exceptions, which should execute fast and then exit, don't need big stacks, but the supervisor mode however will need a relatively large stack, as it's where the kernel executes.
Thanks to our linker script, we have labels to sections of memory we know are good enough to store the different stacks, so all we need is to properly initialize the stack pointers to these labels, which is very straightforward:

```
msr CPSR_c, #0xD1                    ;@ Switch to FIQ_MODE
ldr sp, =__fiq_stack                 ;@ Set the stack pointer for that mode
   to 0x7000
msr CPSR_c, #0xD2                    ;@ Switch to IRQ_MODE
ldr sp, =__irq_stack                 ;@ Set the stack pointer for that mode
   to 0x6000
msr CPSR_c, #0xD3                    ;@ Switch back to SRV_MODE
ldr sp, =__svc_stack                 ;@ Set the stack pointer for that mode
   to 0x5000
```

Code provided by Leon de Boer

2.3 Initializing hardware

Finally, there are some things in the hardware that are not initialized by the Video Core that we need to initialize ourselves. Most of theses are not mandatory, but they will make code execution much faster, so it is worth the trouble.
The things we want to do ar basically, initializing the Floating Point Unit so that we can use floating point numbers, enabling branch prediction to accelerate the flow of the program and enable the caches for faster memory access for commonly used variables.
Here we would also initialize the other cores but we didn't reach that point.

This can all be done sequentially and each thing is relatively small, so we will simply provide the code snippets necessary for these processes.

- Initializing the FPU

```
    mrc p15, 0, r0, c1, c1, 2        ;@ Read NSACR into R0
    cmp r0, #0x00000C00              ;@ Access turned on or in AARCH32
                                     ;@ mode and can not touch
                                        register or EL3 fault

    beq .free_to_enable_fpu
    orr r0, r0, #0x3<<10             ;@ Set access to both secure and
                                     ;@ non secure modes
    mcr p15, 0, r0, c1, c1, 2        ;@ Write NSACR

  .free_to_enable_fpu:
    mrc p15, 0, r0, c1, c0, #2        ;@ R0 = Access Control Register
    orr r0, #(0x300000 + 0xC00000)    ;@ Enable Single & Double
       Precision
    mcr p15,0,r0,c1,c0, #2            ;@ Access Control Register = R0
    mov r0, #0x40000000               ;@ R0 = Enable VFP
    vmsr fpexc, r0                    ;@ FPEXC = R0
```

Code provided by Leon de Boer

- Enable the caches and Branch Prediction

```
    mrc p15,0,r0,c1,c0,0             ;@ R0 = System Control Register

  #define SCTLR_ENABLE_DATA_CACHE     0x4
  #define SCTLR_ENABLE_BRANCH_PREDICTION 0x800
  #define SCTLR_ENABLE_INSTRUCTION_CACHE 0x1000

    // Enable caches and branch prediction
    orr r0, #SCTLR_ENABLE_BRANCH_PREDICTION
    orr r0, #SCTLR_ENABLE_DATA_CACHE
    orr r0, #SCTLR_ENABLE_INSTRUCTION_CACHE

    mcr p15, 0, r0, c1, c0,0            ;@ System Control Register = R0
```

Code provided by Leon de Boer

To see a final, working implementation of all of these features, you can refer to the Boot.s file in the PiOS source code.

3 Software initialization, the _cstartup routine

In a fully implemented Operating System, any global variables that need to be initialized before they are used, and any library initialization code needs to be called before these libraries cna be used. In our modest project, there are only a couple of libraries, but they need to be initialized nonetheless. Also, the `bss` section must be cleared (i.e we must write a 0 to all addresses in the section). All of this is done in the `_cstartup` function.

Although very humble in it's current state, if the project is extended so that more functionality is added, this is where we would call any library initialization code, in order to ensure that everything is setup before any function in a library gets called.

The very first thing we want to do is clear the `bss` section. Thanks to our linker script, we have a label to indicate exactly where this section begins and another label to indicate exactly where it ends, the script also ensures that all `bss` sections of all files get contiguously put together in the final kernel image, so we know that, once we clear this section, all variables that need to be initialized to 0 will contain their correct values. Since the addresses form a continuous block of memory, we can easily do his with a loop:

```
volatile uint32_t* current_address = &__bss_start__;
uint32_t* end = &__bss_end__;

while(current_address < end)
{
   *current_address = 0;
   current_address++;
}
```

The next step is to sequentially call any initialization code from all libraries, and then go to the main kernel loop. So the final code would look like:

```
void _cstartup()
{
   /* clear bss section */

   /* initialize all libraries */
   init_lib1();
   init_lib2();
   init_lib3();
   [...]

   kernel_main();

   while(1){}
}
```

We add an infinite loop at the end as a precaution. The main kernel routine should never return, but if for any reason it did, this prevents the program counter from being loaded with random values (which could even damage the hardware).

The final implementation for the PiOS project can be seen in the cstartup.c file.

III — Peripherals

Now that we have initialized the hardware and our libraries we can start doing some I/O.

1 Mailbox

The most important peripheral is probably the Mailbox. Most communication with non default peripherals such as monitors and USB peripherals is done through this device. Although officially there are 2 mailboxes on the Pi 3, it is unclear what the purpose of the second mailbox is, so for the sake of this document, it can be assumed that whenever we talk about the mailbox we are referring to the first one.

1.1 Mailbox Property Interface

Communicating with the mailbox is a complex task and may be confusing. First there are 10 channels defined for communication with this device:

Channel	Name
0	Power management
1	Framebuffer
2	Virtual UART
3	VCHIQ
4	LEDs
5	Buttons
6	Touch screen
7	Undefined
8	Property tags (ARM to VC)
9	Property tags (VC to ARM)

But we will only focus on channel 8, since the other channels tend to behave strangely. We also must be aware of how the device is mapped to memory, which for all 3 models can be expressed as:

Address	Register
BASE + 0x00	read
BASE + 0x04	unused
BASE + 0x08	unused
BASE + 0x0C	unused
BASE + 0x10	Poll
BASE + 0x14	Sender
BASE + 0x18	Status
BASE + 0x1c	Configuration
BASE + 0x20	Write

For all 3 models the mailbox base address is **0xB880** and the peripheral address for the Pi 3 is **0x3F000000**, so the BASE variable in the above table should be **0x3F00B880**.

Reading and writing to the mailbox are always done in the same fashion:

- Reading
  1. Read status register until the Mailbox isn't empty.
  2. Red the data from the read register
  3. Check the lower four bits to see if they match the desired channel, if not go to 1.
  4. Clear the lower 4 bits, the data is in the 28 most significant bits.
- Writing
  1. Read status register until the Mailbox isn't full.
  2. Write the data along with the channel, which is stored in the 4 less significant bits.

Note that, because the channel is stored in the lowest 4 bits, the data must be 16 byte aligned to ensure the lower 4 bits are all 0. A more in depth explanation can be found on a Github Repository but it should be taken with a grain of salt as there are mistakes on the documentation, and it is outdated for the Pi 3.

Reading

```
uint32_t read_from_mailbox(Channel channel)
{
   uint32_t status;

   do
      status = *(volatile uint32_t *)(MAIL_BASE + IO_BASE + 0x18);
   while((status & MAIL_EMPTY));

   uint32_t response;

   do
      response = (*(uint32_t *)(MAIL_BASE + IO_BASE));
   while((response & 0xF) != channel);

   return response & ~0xF;
}
```

Writing

```
void write_to_mailbox(uint32_t message, Channel channel)
{
   uint32_t status;

   do
      status = *(volatile uint32_t *)(MAIL_BASE + IO_BASE + 0x18);
      while((status & MAIL_FULL));

      *(volatile uint32_t *)(MAIL_BASE + IO_BASE + 0x20) =
         (((uint32_t)(message) & ~0xF) | (uint32_t)(channel));
}
```

1.2 Mailbox Messages

When we wish to write to the mailbox, we must create a contiguous message in memory that specifies the information we are sending to the mailbox, and what we write to the Mailbox is the address of this message, combined with the channel, as seen before.
There is a general pattern for all messages as follows:

- Message size in bytes.
- Request/Response code.
- Sequence of concatenated tags.
- 0x0 (End tag).

The request code is 0, and we should always write this value when sending a message, the 2 possible response codes are **0x80000000** for a successful response and **0x80000001** for an error.

The general pattern for a single tag is:

- Tag identifier.
- Length of the value buffer in bytes.
- Length of the response buffer (overlaps value buffer) in bytes.
- Value/Response buffer.

For simplicity and convenience we can make all values are the size of a **word**. So as to consider all of them 32 bit long numbers. The most general example for a mailbox message would be:

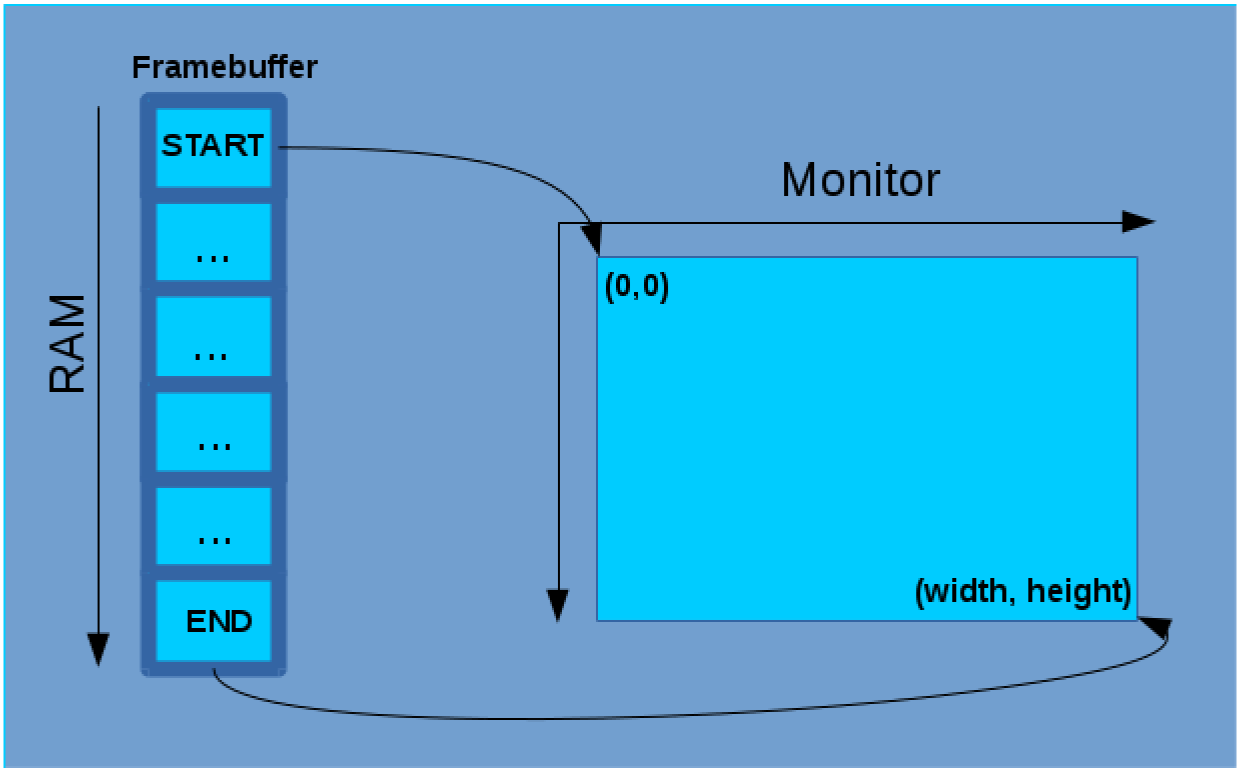
u32:	Size	
u32:	0x0	
	u32:	Tag 1
	u32:	Request size
	u32:	Response size
	u32:	Start of Value/Response buffer
	⋮	
	u32:	End of Value/Response buffer
	u32:	Tag 2
	u32:	Request size
	u32:	Response size
	u32:	Start of Value/Response buffer
	⋮	
	u32:	End of Value/Response buffer
⋮		
u32:	0x0	

A detailed explanation of most tags can be found on the official Github Page But once again, there are mistakes so one should be weary.
The complete list of all tags defined on the pi 3 is given as an appendix.
A good way to test the Mailbox is to set the ACT LED , which can be done with the following example code:

```c
uint32_t mailbox_message[8] __attribute__ ((aligned (16)));
uint32_t index;

void set_LED(int value)
{
   // Set the default message values
   index = 1;
   mailbox_message[index++] =0, //request
   mailbox_message[index++] = (uint32_t) SET_GPIO_STATE, //tag
   mailbox_message[index++] = 8, //request size
   mailbox_message[index++] = 0, // response size
   mailbox_message[index++] = 130, //pin num
   mailbox_message[index++] = value, //state ON or OFF
   mailbox_message[index++] = 0, //end tag

   mailbox_message[0] = index*sizeof(uint32_t); //Total message size

   write_to_mailbox(((uint32_t) &mailbox_message | BUS_MASK),
      PTAG_ARM_TO_VC);
   read_from_mailbox(PTAG_ARM_TO_VC);
}
```

1.3 Framebuffer

The most important use of the Mailbox for us, is it's ability to communicate to an HDMI connected monitor. A framebuffer is a contiguous section of memory that the Video Core reads information from and writes this information to the monitor. In other **word** s, it's a 1 dimensional array that is mapped to a 2 dimensional space, where each entry in the array corresponds to a pixel in the screen.



For things to work properly we must know beforehand the dimensions of the monitor we will be working with. Then we can set a virtual display whose dimensions are independent from the actual resolution of the monitor, for example we could make the virtual display 4 times smaller than the monitor, and as such each virtual pixel would correspond to 4 real pixels.

However, for simplicity, we will initialize the framebuffer such that the dimensions match the monitor's and we will also make it so that the color depth is 32 bits, such that each **word** in the framebuffer corresponds to a pixel. There are 4 tags relevant to the framebuffer initialization process, set the physical dimensions, set the virtual dimensions, set the color depth and allocate the framebuffer in memory. This last one is important, as, unlike a desktop computer, the Raspberry Pi shares RAM with the Video Core, so we want to avoid accidentally overwriting the data. Fortunately the Video Core returns not only the address of the framebuffer, but its size in bytes, so we can easily calculate it's boundaries.

For both the physical dimensions, virtual dimensions and color tags, the Video Core May not set the same values as the message depending on a variety of factors, the actual set values are returned at the same addresses as where they were sent (e.g the set physical width is returned on the same address as where the requested physical width was). The framebuffer tag also allows to specify the allignment of the allocated framebuffer on the same address as where the framebuffer pointer will be returned.

A possible way to create this message is as follows.

```
uint32_t mailbox_message[22] __attribute__ ((aligned (16)));
uint32_t index;

void set_init_display_message()
{
   index = 1;
   mailbox_message[index++] = 0;//request code

   mailbox_message[index++] = (uint32_t) SET_PHYSICAL_WIDTH_HEIGHT; //tag
   mailbox_message[index++] = 8; //request size
   mailbox_message[index++] = 8; //response size
   mailbox_message[index++] = physical_width; //horizontal resolution of
      the monitor
   mailbox_message[index++] = physical_height; //vertical resolution of
      the monitor

   mailbox_message[index++] = (uint32_t) SET_VIRTUAL_WIDTH_HEIGHT; //tag
   mailbox_message[index++] = 8; //rquest size
   mailbox_message[index++] = 8; // response size
   mailbox_message[index++] = virtual_width; //horizontal resolution of
      virtual screen
   mailbox_message[index++] = virtual_height; //vertical resolution of
      virtual screen

   mailbox_message[index++] = (uint32_t) SET_DEPTH; //tag
   mailbox_message[index++] = 4; //request size
   mailbox_message[index++] = 4; //response size
   mailbox_message[index++] = color_depth; //color depth of the frame
      buffer

   mailbox_message[index++] = (uint32_t) ALLOCATE; //tag
   mailbox_message[index++] = 8; //request size
   mailbox_message[index++] = 8; //response size
   mailbox_message[index++] = 16; //alignment fb ptr returned here
   mailbox_message[index++] = 0; //fb size returned here

   mailbox_message[index++] = END;//end tag

   mailbox_message[0] = index*sizeof(uint32_t); //size of message
}
```

After this we simply write the message's address to the mailbox as previously explained and everything should be set. A fully working implementation for setting and using a framebuffer can be found on the mailbox.cpp file in the PiOS source code.

2 System timer

The next peripheral we want to focus on is the system timer. There are actually multiple clocks on the Raspberry Pi located at different address, some are used by the ARM CPU, others by the Video Core. The one we are using is supposed to be very stable and it's what Embedded Xinu uses to keep track of time.

The system timer runs by default at 1.2 GHz, so for convenience and simplicity we can assume that 1 million cycles are 1 second (this is quite obviously wrong and would cause issues in a real time OS, but for our humble project it's good enough), the actual relation is that 1.2 million cycles are 1 second, so we are somewhat close.

On the Pi 3 the peripheral is located at the peripheral address plus **0x3000** so at address **3F003000**. The register layout for this peripheral is:

Address	Register
BASE+0x00	Control Status
BASE+0x04	Low Counter
BASE+0x08	High Counter
BASE+0x0C	Compare 0
BASE+0x10	Compare 1
BASE+0x14	Compare 2
BASE+0x18	Compare 3

The **Control Status** register allows us to control and check the status of the timer. The **Low Counter** and **High Counter** register from a 64 bit counter, the first being the least 32 significant bits and the other being the 32 most significant bits. The remainder registers are used for interrupt setup.
Getting the current cycle can be done by reading the contents of both registers, or we can ignore the high counter and simply use the low one.

To enable a timer interrupt we write the cycle number at which the interrupt must be triggered to any of the Compare registers. However the **Compare 0** and **Compare 2** registers are used by the Video Core, so it is best to not touch them.

To clear an interrupt we must write a 1 to the bit in the **Control Status** corresponding to the system timer Compare register that triggered the interrupt, so we write **0x2** to the status control register to clear an interrupt triggered by **Compare 1** and **0x8** to clear an interrupt triggered by **Compare 3** (Beware the Xinu documentation is wrong on this regard).

3 Interrupts

As we saw before the IVT must be installed into low memory and consists of branch instructions that call the actual exception handlers. Although there are 7 defined exceptions, the **Reset** exception we already discussed, and of the rest the most important one asides form the **IRQ/FIQ** is the **Software Interrupt** which can be used to implement system calls. Unfortunately we didn't get that far and as such we will only discuss **IRQ's** on this section.

3.1 Defining the Exception Handlers

The exception handlers behave somewhat differently to normal subroutines and as such need especial entry and return code. We already discussed that they all have their own stack pointer, which we also set already. however they also have other shadowed registers, and depending on teh exception the execution mode will automatically change according to this table.

We will thus make use of the GNU compiler's features, which can automatically generate appropriate entry and exit points for our exception handlers so that we don't have to manually set them through assembly. The declaration of these handlers will look as follows:

```
void _reset_() __attribute__((interrupt("RESET")));
void undefined_instruction_vector() __attribute__((interrupt("UNDEF")));
void software_interrupt_vector() __attribute__((interrupt("SWI")));
void prefetch_abort_vector() __attribute__((interrupt("ABORT")));
void interrupt_vector() __attribute__((interrupt("IRQ")));
void fast_interrupt_vector() __attribute__((interrupt("FIQ")));
```

WARNING: This is not C syntax, it's a unique feature of the GNU compiler.

We shall also define stubs for all of the unimplemented exception handlers. Which will all look identically.

```
void prefetch_abort_vector()
{
    __asm__
    (

      "cpsid if\n"
    );
    set_LED(ON);
    while(1){}
}
```

3.2 The Interrupt Controller

The Raspberry Pi 3 has a memory mapped peripheral to manage exceptions. The Interrupt Controller is located on the peripheral section (`0x3F000000`) of RAM at offset `0xB200` so actual address `0x3F00B200` . And it has the following register layout:

Address	Register
BASE+0x00	IRQ basic pending
BASE+0x04	IRQ pending 1
BASE+0x08	IRQ pending 2
BASE+0x0C	FIQ control
BASE+0x10	IRQ enable 1
BASE+0x14	IRQ enable 2
BASE+0x18	IRQ enable basic
BASE+0X20	IRQ disable 1
BASE+0X24	IRQ disable 2
BASE+0X20	IRQ disable basic

The first set of registers is used to check for the source(s) of an interrupt. The second set is used to enable interrupt sources and the last set is used to disable interrupt sources. Checking for an interrupt source is done by reading the corresponding bit in any of the pending registers, a set bit indicates the source has triggered an interrupt and we must handle it; enabling or disabling an interrupt source is done by setting the corresponding bit in the relevant register. The full bit and register association list can be found in Appendix A.

3.3 IRQ Handler

The interrupt source we truly care about is the system timer. The true purpose of this interrupt source was to implement preventive multi tasking, however in it's current state it's only a proof of concept.

The way the system timer triggers an interrupt is by comparing the value of the 4 system compare registers if any is less than or equal to the current value of the `Low Counter` register, the interrupt will fire. However all interrupt sources are disabled by default when we first boot, to enable them we must first change the program state (`cpsr`) register to enable IRQ's, then enable the system timer as an interrupt source in the interrupt controller and finally write the cycle after which we want the interrupt to trigger.

```
void kernel_main()
{
    __asm__ volatile
    (
    "cpsie i\n"
    );

    IRQ_controller->enable_irqs_1 = 2;

    System_Timer->Compare1 = trigger_cycle;
}
```

After the setup is ready, we need to ensure that we handle things properly once the interrupt handler gets called. Interrupts can still be triggered inside the handler, so ideally the first thing we would like to do is to disable them as soon as we enter. However on the Pi 3 this seems to clear the pending registers, so instead we must store the pending registers information before disabling interrupts, this is a risk, since interrupts could fire while we do this, but the author didn't have enough time to design a better approach.
We can disable interrupts in multiple ways, an option is to disable all of them by writing a 1 to every single bit in every register for example, but since we know what the interrupt source is going to be we can simply write the pending register's contents to the disable register, thus disabling all and only those interrupts that got fired.
Once disabled we want to check which interrupt sources need to be taken care off and call the appropriate code for each source. Disabling an interrupt will NOT clear the corresponding bit in the pending register, this has to be done in a device specific way, so each source is cleared differently. However we already know that for the timer this is simply writing a 1 to the appropriate bit in the **Status Control** register.
Finally, once all interrupt sources have been taken care of, we enable those that we want to enable and return to wherever execution stopped when the interrupt was called.

IRQ Handler example

```
void interrupt_vector()
{
    uint32_t pending_1_status = irq_controller->IRQ_pending_1;
    // Disable all interrupt sources with pending interrupts
    irq_controller->Disable_IRQs_1 = irq_controller->IRQ_pending_1;

    // Check if interrupt source is the system timer
    if(pending_1_status & 0x2)
    {
        // Check if the system timer compare 1 register is less than the
           current time
        if(system_timer->compare_1 <= system_timer->counter_low)
        {
            /*
            * To clear this irq we must write a 1 to the bit in the control
               status register
            * that has the same index as the system compare register (see
               docuemntation)
            */
            system_timer->control_status = 0b10;
            // Show interrupts are getting called
            print("\nIrq's called:");
            print(example++);
            // Schedule an interrupt in 3 seconds
            system_timer->compare_1 = system_timer->counter_low + 3000000;
        }

        // Re-enable the system timer interrupt
        irq_controller->Enable_IRQs_1 = 0x2;
    }
}
```

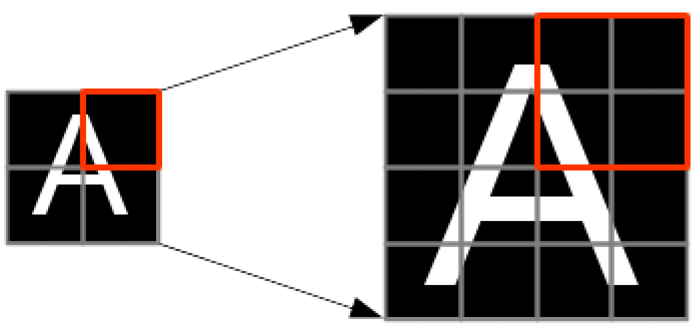
The PiOS implements this code in between the kernel_main.cpp and the interrupts.c files.

IV — Kernel libraries

1 Basic I/O

There is no currently implemented way to get input, and there are only 2 ways to get output in the current state of the PiOS. The first output way is the ACT led, which we set through the mailbox, as the Pi 3 does not directly expose the LED's bus addresses to the ARM CPU due to space constraints. Since we have already talked about how to do this in 2 previous sections we shall omit it here.

The other implemented method is a simplistic **print()** method. It begins by borrowing an 8×8 bit font that defines all basic **ascii** characters for us. then, with the help of the memory allocation routine (described in the next section), we create a buffer as a **word** array that will contain the expanded version of the character as an image that will be drawn to the framebuffer. Then we simply map the original character image to the expanded one bit by bit, making each bit become a **word** representing a color (in our case white). In other **word**s if the final character is to be 4 times bigger than the bit font, then we create an array buffer with $(8*4)^2$ **words**. And then we simply map this final image to a position in the framebuffer and copy the data.



Character expansion algorithm

```
void init_char_image(const char* charMap, uint32_t size, uint32_t*
    drawnChar)
{
   for(uint32_t i=0; i<CHAR_BITS*size; i++)
   {
      uint32_t line = charMap[i/size];
      // iterate through every column in teh final char image
      for(uint32_t j=0; j<CHAR_BITS*size; j++)
      {
         drawnChar[i*CHAR_BITS*size+j] = (BIT(line, 0))*WHITE_32; //
            BIT(val,bit) is a bit mask
         if(j%size == 0 && j>0)
            // Every size iterations we must shift the current line to
               select the next bit
            line = line >> 1;
      }
   }
}
```

Image Positioning algorithm

```
void drawChar(uint32_t *characterImage, uint32_t size,
uint32_t x_offset, uint32_t y_offset)
{
   uint32_t scaling = CHAR_BITS*size;
   // transform the given coordinates from text coordinates to screen/fb
      coordinates
   x_offset *= scaling;
   y_offset *= scaling;

   // iterate through every pixel in the buffer
   for(uint32_t i=0; i<scaling; i++)
   {
      for(uint32_t j=0; j<scaling; j++)
      {
         // The current framebuffer word (a pixel)
         *(volatile uint32_t *)((main_monitor.fb_ptr & ~BUS_MASK)
         + ((i+y_offset)*main_monitor.virtual_width + (j+x_offset)) //map
            the char buffer value
         *main_monitor.color_depth/CHAR_BITS) = //to the fb coordinates
         // The char buffer value
         characterImage[i*scaling+j];
      }
   }
}
```

The fully implemented print function and it's overwritten versions are defined in the string.c file in the PiOS.

2 Memory allocation

For anyone familiar with the Linux operating system, the **malloc()** function should ring a bell. We took inspiration from the Linux specifications and the **malloc()** function to write our own, simplistic memory allocation routine.

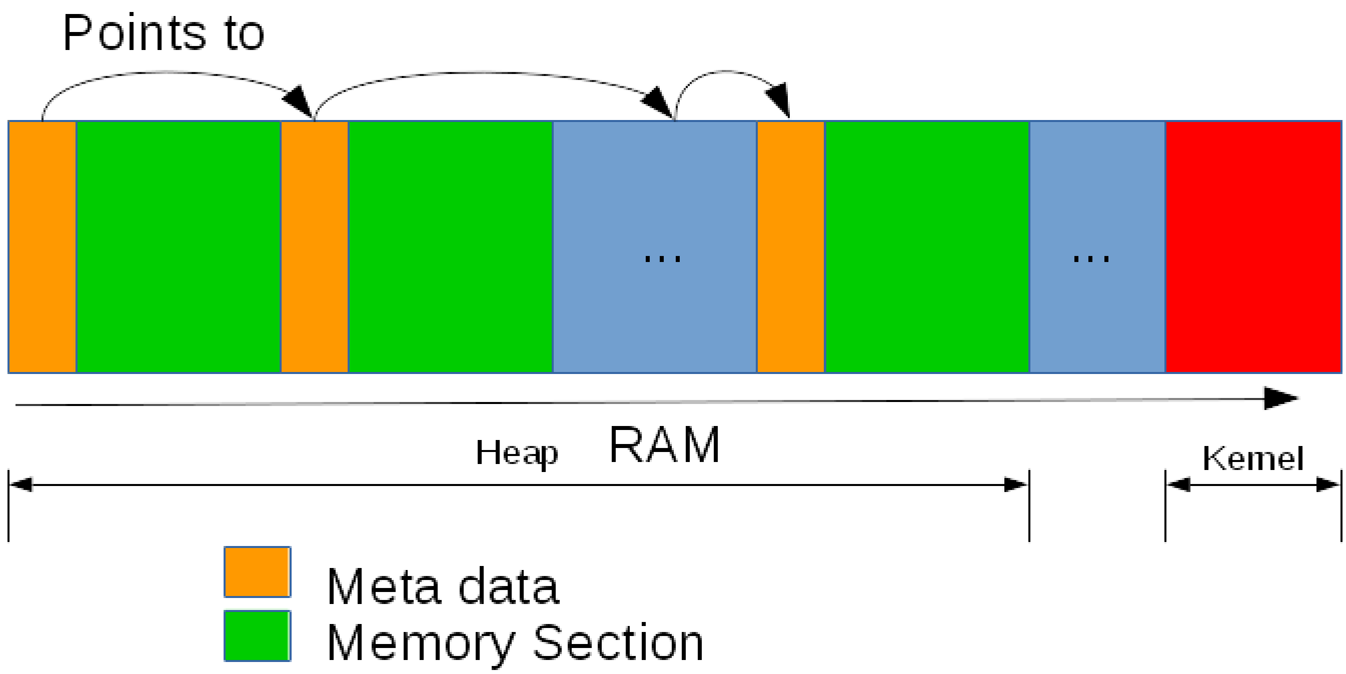
The first challenge is that the Video Core and the ARM processor share the same RAM memory, in addition to this there are multiple memory mapped peripherals, so trying to select a heap area is problematic, as one may accidentally end up writing to a protected section of memory. The memory mapped peripherals and framebuffer RAM, seem to start in high memory after **0x3F000000**, an our kernel lives in low memory at around **0x8000** the Salomonic solution is to take the space in between the kernel and this address as our heap space and sacrifice all other, potentially useable RAM for the sake of security.

After doing this we need a way to mark which sections of the memory are used and which are protected and then we need a way to allocate this memory safely. Routines compiled by C look for memory backwards (they subtract from the base address), which means that whatever reference it returns must be above a section of memory large enough to contain the data. Moreover we need an efficient way to find free memory sections in the heap. We implemented the most naive and simplistic algorithm that solves all of this problems. However the algorithm is a first fit solution, and due to it's nature it not only creates a lot of memory fragmentation, but it is also susceptible of breaking after a large, undefined number of calls. So it should eb taken as an example more than a serious solution.

Starting at the final address (**0x3F000000**) we start creating blocks from high memory into low memory whenever memory is requested. A memory block consists of 2 parts, a memory section, corresponding to the allocated memory, and a meta data section containing the size of the block, a marker that indicates the state of the block (e.g free or locked) and a pointer to the next memory block's metadata. So allocating memory is done by iterating through all blocks, trying to find one that is free and large enough to hold the requested memory, if one is found we simply return the address immediately below the metadata, which is the first address in the memory section. If no such block is found, we see if there is enough memory from the last block to the end of the kernel space to hold the requested memory block and it's metadata, if enough memory is available we increase the heap space and create a new block, then we return the address to the memory section.

It is obvious from this explanation why we experience memory fragmentation and why we will eventually be unable to find appropriate blocks despite the availability of free ram. But it is a correct and working example, good enough for illustration purposes.

The memory allocation implementation and related subroutine is found in the memory_management.cpp file.


Points to
...
...
Heap
RAM
Kernel
Meta data
Memory Section

Part III
Epilogue, Personal reflection

What I learnt

This project certainly taught me a lot. First it forced me to learn a lot about the compilation process, before I had very little experience with make file scripts and I think I am very competent with them now, I also learnt a lot about the compilation process, the difference between executables and object files, the different compilation stages...
I also learnt a lot about the C language it's power and it's limitations, and I think I now understand exactly why it is always said that C is a language for writing operating systems. It is certainly incredibly versatile, and it's one to one correspondence with assembly language makes it very suitable at imagining how the source code will be compiled and executed by the machine.

I became more proficient with pointers, I was somewhat competent before, but I now understand like the back of my hand, pointer arithmetic, void pointers, the difference between pointers and references, passing by value vs passing by reference and the differences between structs and functions. For example, a mistake I did was assuming that all pointers are treated the same by C, and then I discovered that void pointer arithmetic is forbidden by the C standard, because of how pointer arithmetic works (i.e `pointer_type++` adds to the address stored in the pointer a value equal to the size of the structure or primitive it points at, so a pointer to a byte is incremented by 1 but a pointer to an integer is incremented by 4). This also makes arrays make a lot more sense now, as all that they are is a base address and a lot of pointer arithmetic.

Off course I also learnt a lot about low level programming, embedded systems and the ARM architecture. I know see how much I have yet to understand about low level programming but I am definitely a step ahead on that process. I could go extend this document further by enumerating all of what I learnt, but I hope that all the explanations through the document reflect it well enough, and this project report is already larger than I expected so we shall end our PiOS adventure here... Until the next time.

Where to go from here

We have just scratched the surface, and there is much more to be done. Among the things that were planned there is, creating a file system, making the kernel be able to call other programs and schedule them, implement USB drivers for basic I/O through a mouse and a keyboard, making the system dynamic by figuring the monitor dimensions dynamically instead of hard coding them for a specific resolution. Implement multi-threading and initializing the other 3 cores, creating a shell, improving the existent code, specially the memory allocation... Alas time is short and one person can only do so much.

This document and code will hopefully help, however tries something like this in the future, to advance more quickly and avoid many headaches. Maybe one day the PiOS will be fully operational, but as of now, it and I both rest...

Part IV
Additional information

Appendices

A Arm Execution Modes

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Figure A2-1 Register organization

B Interrupt Controller Register Map

Interrupt source	bit number
IRQ pending 1	
INTERRUPT_TIMER0	0
INTERRUPT_TIMER1	1
INTERRUPT_TIMER2	2
INTERRUPT_TIMER3	3
INTERRUPT_CODEC0	4
INTERRUPT_CODEC1	5
INTERRUPT_CODEC2	6
INTERRUPT_VC_JPEG	7
INTERRUPT_ISP	8
INTERRUPT_VC_USB	9
INTERRUPT_VC_3D	10
INTERRUPT_TRANSPOSER	11
INTERRUPT_MULTICORESYNC0	12
INTERRUPT_MULTICORESYNC1	13
INTERRUPT_MULTICORESYNC2	14
INTERRUPT_MULTICORESYNC3	15
INTERRUPT_DMA0	16
INTERRUPT_DMA1	17
INTERRUPT_VC_DMA2	18
INTERRUPT_VC_DMA3	19
INTERRUPT_DMA4	20
INTERRUPT_DMA5	21
INTERRUPT_DMA6	22
INTERRUPT_DMA7	23
INTERRUPT_DMA8	24
INTERRUPT_DMA9	25
INTERRUPT_DMA10	26
INTERRUPT_DMA11	27

INTERRUPT_DMA12	28
INTERRUPT_AUX	29
INTERRUPT_ARM	30
INTERRUPT_VPUDMA	31
IRQ pending 2	
INTERRUPT_HOSTPORT	0
INTERRUPT_VIDEOSCALER	1
INTERRUPT_CCP2TX	2
INTERRUPT_SDC	3
INTERRUPT_DSI0	4
INTERRUPT_AVE	5
INTERRUPT_CAM0	6
INTERRUPT_CAM1	7
INTERRUPT_HDMI0	8
INTERRUPT_HDMI1	9
INTERRUPT_PIXELVALVE1	10
INTERRUPT_I2CSPISLV	11
INTERRUPT_DSI1	12
INTERRUPT_PWA0	13
INTERRUPT_PWA1	14
INTERRUPT_CPR	15
INTERRUPT_SMI	16
INTERRUPT_GPIO0	17
INTERRUPT_GPIO1	18
INTERRUPT_GPIO2	19
INTERRUPT_GPIO3	20
INTERRUPT_VC_I2C	21
INTERRUPT_VC_SPI	22
INTERRUPT_VC_I2SPCM	23
INTERRUPT_VC_SDIO	24
INTERRUPT_VC_UART	25
INTERRUPT_SLIMBUS	26

INTERRUPT_VEC	27
INTERRUPT_CPG	28
INTERRUPT_RNG	29
INTERRUPT_VC_ARASANSDIO	30
INTERRUPT_AVSPMON	31
IRQ pending basic	
INTERRUPT_ARM_TIMER	0
INTERRUPT_ARM_MAILBOX	1
INTERRUPT_ARM_DOORBELL_0	2
INTERRUPT_ARM_DOORBELL_1	3
INTERRUPT_VPU0_HALTED	4
INTERRUPT_VPU1_HALTED	5
INTERRUPT_ILLEGAL_TYPE0	6
INTERRUPT_ILLEGAL_TYPE1	7
INTERRUPT_PENDING1	8
INTERRUPT_PENDING2	9
INTERRUPT_JPEG	10
INTERRUPT_USB	11
INTERRUPT_3D	12
INTERRUPT_DMA2	13
INTERRUPT_DMA3	14
INTERRUPT_I2C	15
INTERRUPT_SPI	16
INTERRUPT_I2SPCM	17
INTERRUPT_SDIO	18
INTERRUPT_UART	19
INTERRUPT_ARASANSDIO	20

C Mailbox Tags

END	0
GET_FIRMWARE_REVISION	0x00000001
SET_CURSOR_INFO	0x00008010
SET_CURSOR_STATE	0x00008011
GET_BOARD_MODEL	0x00010001
GET_BOARD_REVISION	0x00010002
GET_BOARD_MAC_ADDRESS	0x00010003
GET_BOARD_SERIAL	0x00010004
GET_ARM_MEMORY	0x00010005
GET_VC_MEMORY	0x00010006
GET_CLOCKS	0x00010007
GET_POWER_STATE	0x00020001
GET_TIMING	0x00020002
SET_POWER_STATE	0x00028001
GET_CLOCK_STATE	0x00030001
GET_CLOCK_RATE	0x00030002
GET_VOLTAGE	0x00030003
GET_MAX_CLOCK_RATE	0x00030004
GET_MAX_VOLTAGE	0x00030005
GET_TEMPERATURE	0x00030006
GET_MIN_CLOCK_RATE	0x00030007
GET_MIN_VOLTAGE	0x00030008
GET_TURBO	0x00030009
GET_MAX_TEMPERATURE	0x0003000a
GET_STC	0x0003000b
ALLOCATE_MEMORY	0x0003000c
LOCK_MEMORY	0x0003000d
UNLOCK_MEMORY	0x0003000e
RELEASE_MEMORY	0x0003000f
EXECUTE_CODE	0x00030010

EXECUTE_QPU	0x00030011
SET_ENABLE_QPU	0x00030012
GET_DISPMANX_RESOURCE_MEM_HANDLE	0x00030014
GET_EDID_BLOCK	0x00030020
GET_CUSTOMER_OTP	0x00030021
GET_DOMAIN_STATE	0x00030030
SET_CLOCK_STATE	0x00038001
SET_CLOCK_RATE	0x00038002
SET_VOLTAGE	0x00038003
SET_TURBO	0x00038009
SET_CUSTOMER_OTP	0x00038021
SET_DOMAIN_STATE	0x00038030
GET_GPIO_STATE	0x00030041
SET_GPIO_STATE	0x00038041
SET_SDHOST_CLOCK	0x00038042
GET_GPIO_CONFIG	0x00030043
SET_GPIO_CONFIG	0x00038043
ALLOCATE	0x00040001
BLANK_SCREEN	0x00040002
GET_PHYSICAL_WIDTH_HEIGHT	0x00040003
GET_VIRTUAL_WIDTH_HEIGHT	0x00040004
GET_DEPTH	0x00040005
GET_PIXEL_ORDER	0x00040006
GET_ALPHA_MODE	0x00040007
GET_PITCH	0x00040008
GET_VIRTUAL_OFFSET	0x00040009
GET_OVERSCAN	0x0004000a
GET_PALETTE	0x0004000b
GET_TOUCHBUF	0x0004000f
GET_GPIOVIRTBUF	0x00040010
RELEASE	0x00048001
TEST_PHYSICAL_WIDTH_HEIGHT	0x00044003

TEST_VIRTUAL_WIDTH_HEIGHT	0x00044004
TEST_DEPTH	0x00044005
TEST_PIXEL_ORDER	0x00044006
TEST_ALPHA_MODE	0x00044007
TEST_VIRTUAL_OFFSET	0x00044009
TEST_OVERSCAN	0x0004400a
TEST_PALETTE	0x0004400b
TEST_VSYNC	0x0004400e
SET_PHYSICAL_WIDTH_HEIGHT	0x00048003
SET_VIRTUAL_WIDTH_HEIGHT	0x00048004
SET_DEPTH	0x00048005
SET_PIXEL_ORDER	0x00048006
SET_ALPHA_MODE	0x00048007
SET_VIRTUAL_OFFSET	0x00048009
SET_OVERSCAN	0x0004800a
SET_PALETTE	0x0004800b
SET_TOUCHBUF	0x0004801f
SET_GPIOVIRTBUF	0x00048020
SET_VSYNC	0x0004800e
SET_BACKLIGHT	0x0004800f
VCHIQ_INIT	0x00048010
GET_COMMAND_LINE	0x00050001
GET_DMA_CHANNELS	0x00060001

References

[1] Leon de Boer, through multiple posts in the Raspberry Pi forums.

[2] Brian Sidebotham *Valvers Raspberry Pi tutorial*.
`http://www.valvers.com/open-software/raspberry-pi/step01-bare-metal-programming-in-cpt1/`

[3] Jalal Kawash, *Computer Machinery 2 [CPSC 359]* winter 2016, University of Calgary.

[4] The ARM info center
`http://infocenter.arm.com/help/index.jsp`

[5] Embedded Xinu documentation
`http://embedded-xinu.readthedocs.io/en/latest/arm/rpi/`

[6] Raspberry Pi Linux kernel source code
`https://github.com/raspberrypi/linux/tree/rpi-3.6.y/arch/arm/mach-bcm2708`

[7] The official Raspberry Pi Github Repository
`https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface`

[8] Marwan Burelle, *A malloc tutorial*, February 16, 2009

[9] Ken Werner, *Stack Frame Unwinding on ARM*, Budapest 2011
`https://wiki.linaro.org/KenWerner/Sandbox/libunwind?action=AttachFile&do=get&target=libunwind-LDS.pdf`

[10] Raspberry Pi forums
`https://www.raspberrypi.org/forums/`

—*"Live long and propser"*,

Spock