Week 2

Bare-Metal Development
M2M Lectures

Grenoble University

Quentin Cartier

January 12, 2021

1.Preface

1.1 Preface by Pr. Olivier Gruber

This document is your work log for the first step in the M2M course, master-level, at the University of Grenoble, France. You will have such a document for each step of our course together.

This document has two parts. One part is about diverse sections, each with a bunch of questions that you have to answers. The other part is really a laboratory log, keeping track of what you do, as you do it.

The questions provide a guideline for your learning. They are not about getting a good grade if you answer them correctly, they are about giving your pointers on what to learn about.

The goal of the questions is therefore not to be answered in three lines of text and be forgotten about. The questions must be researched and thoroughly understood. Ask questions around you if things are unclear, to your fellow students and to me, your professor.

Writing down the answers to the questions is a tool for helping your learn and remember. Also, it keeps track of what you know, the URLs you visited, the open questions that you are trouble with, etc. The tools you used. It is intended to be a living document, written as you go.

Ultimately, the goal of the document is to be kept for your personal records. If ever you will work on embedded systems, trust me, you will be glad to have a written trace about all this.

REMEMBER: plaggia is a crime that can get you evicted forever from french universities... The solution is simple, write using your own words or quote, giving the source of the quoted text. Also, remember that you do not learn through cut&paste. You also do not learn much by watching somebody else doing.

1.2 Methodology by Quentin Cartier

In order to get a better understanding of the concepts related to the TP, I'll build along with my answers a glossary of the terms used.

2.QEMU

What is QEMU?

Qemu is a type-2 hypervisor. It runs on an OS and can emulate different bare machines.

Why is it necessary here?

In our case, Qemu is necessary, as it allows us to simulate an arm architecture, without having additional hardware, and no cables.

`***Qemu***` : is an open source type-II hypervisor. It can emulate different `*Bare Machine*`s.
It simulates a real physical machine in software. Many `Targets machine`s can be emulated , with different `*Processor*`s and devices.

`***Hypervisor***` : A computer software / firmware or hardware that creates and runs virtual machines. A hypervisor is a variant of supervisor, which is a traditional term for the kernel of the OS. There are two types of hypervisor : `*Type-1 Hypervisor*`, and `*Type-2 Hypervisor*`.

`***Type-1 Hypervisor***` : (Native or Bare-metal hypervisor) Run directly on the host's hardware.

`T***ype-2 Hypervisor***` : (Hosted hypervisor) Run on a conventional OS.

`***Bare Machine***` : A computer device executing instructions directly on logic hardware, without the help of an Operating System.

It allows us to isolate execution, without being aware of the host machine hardware.

`***Target Machine***` : Specific `*Machine*` into which a program is loaded and run.

- In `*Qemu*`.we can take a look at which target machines are supported with the command qemu-system-{architecture} -machine help:

`***Machine***` : Essentially about a preset hardware configuration. In `Qemu`, it is about what is the configuration of the hardware simulated:

- `Processor` type and number of cores
- Memory size
- Peritherics (disks, ide,scsi disks ...)
- Serial lines //TODO
- Display ?
- Buses (ISA / PCI bus)
- etc...

`***Processor***`: The logic circuitry responding to and processing the basic instructions defined. Some of its basic elements are: `ALU` , `FPU`, `Register`s.

`***ALU***` : Arithmetic logic unit. This unit processes arithmetic and logic operations on instructions and operands.

`***FPU***` : (Floating Point Unit) Part of the `Processor` that performs floating point calculations.

`***ARM***` : (*Advanced RISC Machines*) A `RISC` architecture for computer `*Processor*`s. It is a desirable architecture for light, portacle and battery powered devices due to its low costs and , relative lower heat generation, and minimal power consumption.

`***RISC***` (Reduced Instruction Set Computing) : A type of processor architecture that uses fewer and simpler instructions that a complex instruction set computing (CISC) processor. RISC processors perform complex instructions by combining several simpler ones (*source : https://techterms.com/definition/risc*)

`***Register***` : Storage location within the circuitry of the CPU. Very fast on-chip memory storing binary values (32 bits or 64 bits). They are different types of registers:

- General Purpose Registers
- SP (Stack Pointer) : Stores the return address and parameters when a function is called. Stores the return address, when an interrupt occurs, in order to resume the execution after the interrupt.
- PC (Program Counter) : Stores address of the next instruction to be executed.

- SR (Status Register) : Contains bits that are set and cleared based on the results of an instruction. For example, store the information of the occurrence of an overflow.

Getting started

For the project, we will use an ARM Versatile/PB (ARM926EJ-S) (v***ersatilepb)***.

Qemu commands	
Choose the machine	qemu-system-arm ***-machine versatilepb***
administrative console (“monitor console”) on the stdio serial line	*qemu-system-arm -serial mon:stdio*
Memory of the virtual board	qemu-system-arm -m 64M
Keyboard layout	*qemu-system-arm -k en-us*
Access the monitor console in qemu (while qemu is running)	*Ctrl-a c*

We’ll first install the necessary tools:

sudo apt-get install qemu-system-arm qemu-system-x86

The directory arm.boot contains:

```
daoliangshu@zenbook11:~/Documents/Cours_M2GI/IoT/NewCourse/Step0/workspace/arm.b
oot$ ls
kernel.bin  kernel.ld  kprintf.o  main.h  Makefile  README-GDB       startup.o
kernel.elf  kprintf.c  main.c     main.o  README    README-QEMU-ARM  startup.s
```

When we launch do “make run”, the following commands are executed:

```
arm-none-eabi-as -mcpu=arm926ej-s -g  startup.s -o startup.o
arm-none-eabi-ld -T kernel.ld  startup.o main.o -o kernel.elf
arm-none-eabi-objcopy -O binary kernel.elf kernel.bin
qemu-system-arm -M versatileab -m 1M -nographic -kernel kernel.bin -serial mon:s
tdio
```

The steps are:

- Compiling objects
- Linking
- Elf to bin: we obtain the binary file kernel.bin
- Run qemu
  - While running, we can to “ctrl-a c” to access the (qemu) interactive console. Inside the consol, the following commands are accessible:
    - “quit”

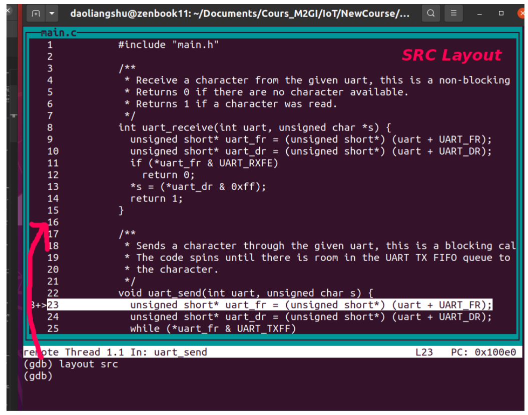
3.GNU Debugger

To use gdb, we need 2 terminals:

Terminal 1	***Terminal 2***
$make debug	$make debug-cli (I made a rule in the makefile containing the command below)
(***command executed by the makefile***) qemu-system-arm -M versatileab -m 1M -nographic -kernel kernel.bin -serial mon:stdio ***-gdb tcp::1234 -S***	`gdb-multiarch -q --nh -ex 'set architecture arm' -ex 'file kernel.elf' -ex 'target remote localhost:1234' -ex 'layout split' -ex 'layout regs'`
	Explanation: Launching gdb-multiarch and set architecture to arm. load file kernel.elf, and connect to the server at localhost at port

	1234. Display a window in the terminal to see the registers and source file.

Basic Gdb commands	
`(gdb)ctrl-C`	Stop the execution
`(gdb)where`	Give current call stack on current thread
`(gdb)list`	With no argument, lists 10 more lines after or around previous listing
`(gdb)thread`	Output current thread
`(gdb)Info threads`	
`(gdb)kill`	Kill the program being debugged
`(gdb)make`	
`(gdb)run`	
Breakpoints	
`(gdb)br *0x7c00`	Set breakpoint giving memory addr
`(gdb)br boot.S:136`	Set breakpoint in assembly source file at line number
`(gdb)br loader.c:122`	Set breakpoint in C source file at line number
`(gdb)br my_function`	Set breakpoint at function name
`(gdb)info br`	Show the breakpoints
`(gdb)d 1`	Remove the breakpoint number 1
Layouts	***Layouts are src, asm, split, regs***
`Layout src`	Display source and command windows

	
`Layout asm`	Display assembler and command windows
`Layout split`	Display src + asm + commande windows
`Layout regs`	Display register window
`Layout next`	
`Continuing and stepping`	
`Continue, c, fg`	Resume program execution, at the address where the programme last stopped.
`step`	Continue running until control reaches a different source line. Only stops at the first instruction of a source line.
`Step <count>`	As step, but do it <count> times.
`Next , n`	Similar to step, but function class that appear within the line of code are executed without stopping.
`finish`	Continue running until just after function in the selected stack frame returns.
`Stepi, stepi arg, si`	Execute one machine instruction, then stop and return to the debugger.
`Printing / Displaying`	
`(gdb)print <variable_name>`	Print the variable.
`(gdb)display <variable_name>`	Print the variable whenever it stops.
`(gdb)x/4xb 0x1000`	Print 4 bytes in hexadecimal at 0x1000

`(gdb)x/2uw 0x1000`	Print 2 words (32 bits) as unsigned decimal.

4. Makefile

You need to read and fully understand the provided makefile. Please find a few questions below highlighting important points of that makefile. These questions are there only to guide your reading of the makefile. Make sure they are addressed in your overall writing about the makefile and the corresponding challenge of building bare-metal software.

1. ***What is the TOOLCHAIN?***

A toolchain is a set of tools (compiler, linker, libraries, debugger, etc) that are used to produce an executable for the target (the computer on which we want to run the code).

> `***Toolchain***`: A set of programming tools usually to create a software program. The tools are in general executed consecutively, such that the output of one environment state becomes the input or the following one. However the term is also used to refer to related tools not necessarily executed consecutively.
> ***Eg****: A simple toolchain may consist of a compiler and a linker, libraries, and a debugger.*
>
> `***Cross Compiler***`*: Compiler capable of creating executable code for a platform other than the one on which the compiler is running (source : https://en.wikipedia.org/wiki/Cross_compiler)*

2. ***What are VersatilePB and VersatileAB***?

`***ARM Versatile Family***`: Base on the ARM926EJ-S CPU core, on a special tailored test chip. Series of machines that have been the basis of ARMv5TE development, and was the second ARM reference design after the ARM integrator to receive full support for linux kernel from ARM. Was popular around 2003-2010. This family has two main boards : `*VersatileAB*` and `*VersatilePB*`.

`**<u>VersatileAB</u>**` (ARM Versatile Application Baseboard) : Board that comes with daughterboards named `EB1` and `EB2`.


(versatileab-eb1 with external display, source : wikipedia)


(versatileab-eb2, source wikipedia)

`**<u>VersatilePB</u>**` (ARM Versatile Platform Baseboard): Board that has a `PCI` expansion tile and is good for development of embedded systems with `PCI`.

`**PCI**` (Peripheral Component Interconnect) : A local computer bus for attaching hardware devices in a computer, and is a part of the PCI Local Bus Standard.

`**EB1**` (Extension Board 1) : Can connect the `VersatileAB` to an external LCD color display using a special cable. Can be used for prototyping interactive display systems.

`**EB2**` (Extension Board 2) : Has a mobile phone form factor, and comes with a small display, candy bar phone keypad and a GSM modem. Can be used for prototyping mobile handsets.

3. ***What is a linker script? Look at the linker option "-T"***

`***Linker Script***`: A text file made up of linker directives telling the linker where the available memory is and how it should be used. Generally and file *.ld for linux. We can supply our own Linker Script by using the '-T' command line option. This is done in the Makefile through the LDFLAGS variable :

```
LDFLAGS= -T kernel.ld
```

Kernel.ld is our Linker Script.

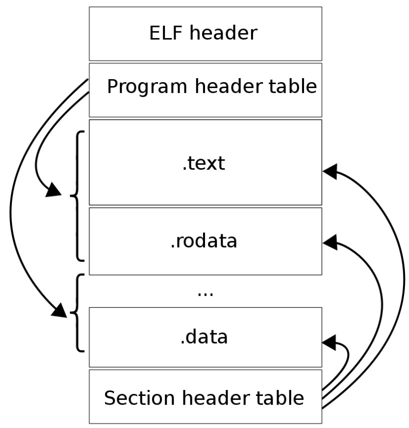
4. ***Read and understand the linker script that we use***

The linker script is separated into two parts : The entry point definition and the definition of the sections.

5. ***Why do we translate the "kernel.elf" into a "kernel.bin" via "objcopy"***
   a. What is a *.elf file

      `***ELF file***` (Executable and Linkable Format): A common standard file format for executable files, object code, shared libraries, and core dumps. ELF file is by design flexible, extensible and cross-platform. It supports different endiannesses and addresses sizes so it does not excludes any particular CPU or ASI.
      An ELF file represents the structure expected of a binary file.

   b. Structure of a elf file



(Composition of a elf file, source : wikipedia)

The elf file has metadatas associated with it. For our kernel.elf file:

```
                                                        [...]/Step0/wor
kspace/arm.boot$ arm-none-eabi-readelf -h kernel.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x10000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          68528 (bytes into file)
  Flags:                             0x5000200, Version5 EABI,
soft-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         16
  Section header string table index: 15
```

c. For elf to bin

Why do we need to translate the elf file into a bin file?

The ELF file has metadata associated with it. It is a product of building and needs to be translated into a raw binary file *.bin that does not contain memory fix-ups or relocations and has explicit instructions to be loaded at a specific memory address, which is necessary for the device that expects instructions at a given address.

6. ***What ensures that we can debug?***

The *.elf ensures that we can debug. The elf file contains the location of the symbols in memory so addresses can be mapped into readable symbols.

7. ***What is the meaning of the "-nostdlib" option? Why is it necessary?***

   a. What is -nostdlib?
   It means that it does not use the standard system startup files or libraries when linking, that is there won't be startup files passed to the linker, and only the library explicitly specified will be passed to the linker.

   b. Why is it necessary?
   An embedded device generally has a very limited place in memory, so we should optimize this place and avoid to load libraries that are not necessary.

8. Try MEMORY=32K, it fails, why? Look at the linker script.

```
arm-none-eabi-as -mcpu=arm926ej-s -g  startup.s -o startup.o
arm-none-eabi-ld -T kernel.ld  startup.o main.o -o kernel.elf
arm-none-eabi-objcopy -O binary kernel.elf kernel.bin
qemu-system-arm -M versatileab -m 32K -nographic -kernel
kernel.bin -serial mon:stdio
```

The program compiles, but when I try to debug, I obtain a core dump.

`Core dump` : Generated when a process receives certain signals, such as SIGSEGV, which the kernels sends it when it accesses memory outside its address space.

With MEMORY=1M

```
0x10000 <_start>          ldr      sp, [pc, #64]   ; 0x10048 <_halt+4>
>0x10004 <.relocate>      ldr      r3, [pc, #64]   ; 0x1004c <_halt+8>
0x10008 <.relocate+4>    ldr      r4, [pc, #64]   ; 0x10050 <_halt+12>
0x1000c <.relocate+8>    cmp      r3, r4
0x10010 <.relocate+12>   beq      0x10028 <.clear>
0x10014 <.relocate+16>   ldr      r9, [pc, #56]   ; 0x10054 <_halt+16>
0x10018 <.relocate+20>   ldm      r3!, {r5, r6, r7, r8}
0x1001c <.relocate+24>   stmia   r4!, {r5, r6, r7, r8}
0x10020 <.relocate+28>   cmp      r4, r9
0x10024 <.relocate+32>   bcc      0x10018 <.relocate+20>
0x10028 <.clear>          ldr      r4, [pc, #36]   ; 0x10054 <_halt+16>
0x1002c <.clear+4>        ldr      r9, [pc, #36]   ; 0x10058 <_halt+20>
0x10030 <.clear+8>        mov      r5, #0
```

With MEMORY=32K

```
>0x10000 <_start>      andeq   r0, r0, r0
0x10004 <.relocate>    andeq   r0, r0, r0
0x10008 <.relocate+4>   andeq   r0, r0, r0
0x1000c <.relocate+8>   andeq   r0, r0, r0
0x10010 <.relocate+12>  andeq   r0, r0, r0
0x10014 <.relocate+16>  andeq   r0, r0, r0
0x10018 <.relocate+20>  andeq   r0, r0, r0
0x1001c <.relocate+24>  andeq   r0, r0, r0
0x10020 <.relocate+28>  andeq   r0, r0, r0
0x10024 <.relocate+32>  andeq   r0, r0, r0
0x10028 <.clear>       andeq   r0, r0, r0
0x1002c <.clear+4>     andeq   r0, r0, r0
0x10030 <.clear+8>     andeq   r0, r0, r0
```

We observe that the location for _start is not set for 32K.
We see in the linker script that _start is at memory address 0x10000, which is 65536, that is larger than 32K.

```
 * Since we are loaded by Qemu, as a kernel, we are loaded at 0x10000
 * For simplicity, we consider that we will be linked at the same addr
 * in order to avoid a relocation of the code at each boot.
 */
. = 0x10000;
_load = .;
. = 0x10000;
_start = .;
/*
```

(kernel.ld)

9. Could you use printf in the code? Why?
   No, we use “-nostdlib” in linking, so the standard libraries are not available.

4.1 Linker Script

Detail here your understanding of the linker script that we use.

1. Why do we translate the ”kernel.elf” into a ”kernel.bin” via ”objcopy”

   Objcopy copy or translate an object file into another.

   We give the parameter -O binary to indicate that we want a binary output.

```
# Notice that we link with our own linker script: test.ld
all: startup.o main.o
   $(TOOLCHAIN)-ld $(LDFLAGS) startup.o main.o -o kernel.elf
   $(TOOLCHAIN)-objcopy -O binary kernel.elf kernel.bin
```

“Objcopy” generates a raw binary file, which is basically a `Memory dump` of the contents of the input object file. All the symbols and relocation information are then discarded (no more metadata). The memory dump starts at the load address of the lowest section copied into the output file.

We translate the kernel.elf into kernel.bin so that

2. ***Why do we link our code to run at the 0x10000?***

   The address 0x10000 is the entry point.

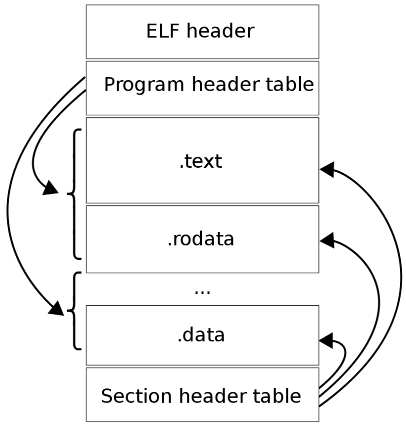
3. ***Why do we make sure the code for the object file ”startup.o” is first?***

   As 0x10000 is our entry point address, we need to make sure that what is read is the starting point of the program.

4.2 ELF Format

1. ***What is the ELF format?***

   `***ELF file***` (Executable and Linkable Format): A common standard file format for executable files, object code, shared libraries, and core dumps. ELF file is by design flexible, extensible and cross-platform. It supports different endiannesses and addresses sizes so it does not excludes any particular CPU or ASI.



(Composition of a elf file, source : wikipedia)

It is widely used for executable files, relocatable object files, shared libraries, and core dumps

2. ***Why is it used as an object file format and an executable file format.***

It is used as such because of the design of the ELF format : It is *flexible*, *extensible* and *cross-platform* by design, supporting different endians and address sizes. That is to say, the ELF's design is not limited to a specific processor, instruction set or hardware architecture.

3. ***How does the ELF executable contain debug information? Which option must be given to the compiler and linker? Why both?***

    a. ***How does the ELF executable contain debug information***

We can use *`objdump -W kernel.elf`* to print raw debug contents contained in ELF

```
kernel.elf:        file format elf32-little

Raw dump of debug contents of section .debug_line:

  Offset:                      0x0
  Length:                      81
  DWARF Version:               3
  Prologue Length:             32
  Minimum Instruction Length:  2
  Initial value of 'is_stmt':  1
  Line Base:                   -5
  Line Range:                  14
  Opcode Base:                 13

 Opcodes:
```

```
  Opcode 1 has 0 args
  Opcode 2 has 1 arg
  Opcode 3 has 1 arg
  Opcode 4 has 1 arg
  Opcode 5 has 1 arg
  Opcode 6 has 0 args
  Opcode 7 has 0 args
  Opcode 8 has 0 args
  Opcode 9 has 1 arg
  Opcode 10 has 0 args
  Opcode 11 has 0 args
  Opcode 12 has 1 arg

 The Directory Table is empty.

[...]
```

b. Which option must be given to the compiler and linker? Why both?

We must give the parameter -g. This parameter requests the compiler and the linker to generate and retain source-level debugging/symbol information in the executable itself.
We need to give this parameter to both the compiler and the linker because:

-

`***Debugging Symbol Table***` : Maps instructions in the compiled binary program to their corresponding variable, function, or line in the source code.

Things inferred :

- A symbol table works for a particular version of a program
- Debug builds are usually larger than retail builds.
- To debug a binary not compiled by yourself, we must get the symbol tables from the author.

`***Debug symbol***` : Special kind of symbol that attaches additional information to the symbol table of an object file.

`***Symbol table***`: A data structure used by a translator (compiler0) , in which each identifier (= symbol) in a program's source code is associated with information relating to its declaration or appearance in the source.

4. Confirm what ELF object files and the final ELF executable are with the shell command "file".

```
~/my/path/IoT/Step0/workspace/arm.boot$ file kernel.elf
kernel.elf: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, with debug_info, not stripped
```

5. Look at the ELF object files and the final ELF executable with the tool: Arm-none-eabi-objdump.

```
~/my/path/IoT/Step0/workspace/arm.boot$ arm-none-eabi-objdump -f kernel.elf

kernel.elf:     file format elf32-littlearm
architecture: armv5tej, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00010000
```

5. Startup Code

Read and understand the startup code in the file "startup.s".
Explain here what it does.

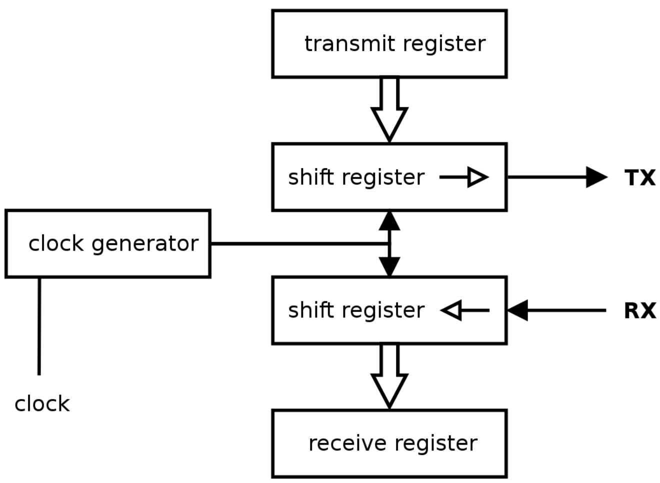
`.global _entry`
`_entry:`
` ldr sp, =stack_top`
	.global is an assembler directive that marks the symbol as global in the ELF file.

6.Main Code

Read and understand the main code in the file "main.c".
Explain here what it does. In particular, explains how the characters you type in the terminal window actually appear on the terminal window.
With a regular shell, the shell echoes the character as you type them. It only sends the characters once you hit the return key, as a complete line.
It is the behavior you notice here?

1. ***What is an UART and a serial line?***

`***UART***` : (Universal Asynchronous Receiver-Transmitter) A asynchronous serial communication. Takes bytes of data and transmits the individual bits in a sequential fashion.



(source : Wikipedia)

2. ***What is the purpose of a serial line here?***

The Serial line in embedded device has the following advantages :

- It's easy to set up. A few lines of codes are enough to configure it.
- It provides communication before the linux kernel is booted, so can be useful to display useful information.

3. ***What is the relationship between this serial line and the Terminal window running a shell on your laptop?***

The shell reads the standard input, and tries to interpret it.
The serial line reads the standard input without interpreting , and prints what is sent to the serial line.

4. ***What is the special testing of the value 13 as a special character and why do we send back '\r' and '\n' ?***

char(13) is carriage return or CR (move the carriage from right to left)
char (10) is called a New Line LN (or Line Feed LF) and written \n.

We are on a basic terminal, without shell interpreting what is sent through uart to the terminal. Traditionally, when we press the button enter, we don't go to a new line, but do a line carriage (moving the point from right to left). Thus, in the code provided, we catch the carriage return, and manually add a new line (\n).

5. ***Why can we say this program polls the serial line? Although it works,why is it not a good idea?***

Polling means that the process loop and checks at each iteration. It works but is inefficient because the process is still working and consumes cpu instructions, and thus consumes power.

6. ***How could using hardware interrupts be a better solution?***

Hard interrupts are a better solution, because it allows the system to sleep (so low power consumption) , and to wake up only when an interrupt is triggered. The program can also do whatever it needs to do, instead of polling.

`***Interrupt***` : A signal to the processor emitted by hardware or software indicating an event that needs immediate attention. When an interrupt occurs, the controller completes the current instruction, and then starts the execution of an `ISR`.

`***ISR***` : (Interrupt Service Routine, or Interrupt Handler) Handler that tells the processor / controller what to do when an interrupt occurs.

`***Hardware Interrupt***` : An electronic alerting signal sent to the processor from an external device. For example: When we press a key on the keyboard, it trigger an hardware interrupt which causes the processor to read the keystroke.

7. ***Could we say that the function uart send may block? Why?***

The function uart_send may block. UART_TXFF is the bit that indicate wheter the UART_TX Fifo queue is full, and we test this flag in the function, so we may block until there is place in the UART_TX Fifo queue.

8. ***Could we say that the function uart receives is non-blocking? Why?***

The function uart_receive is not blocking, as we simply return if the UART_RX Fifo queue is empty.

9. ***Explain why uart send is blocking and uart receive is non-blocking.***

Uart_send is blocking because we need to ensure that we can place a character in the queue before placing it, but for uart_receive, we just check if there is a character.

7. Test Code

7.1 Blocking Uart-Receive

1. Change the code so that the function uart receives is blocking.

```
int uart_receive(int uart, unsigned char *s) {
 unsigned short* uart_fr = (unsigned short*) (uart + UART_FR);
 unsigned short* uart_dr = (unsigned short*) (uart + UART_DR);
 while (*uart_fr & UART_RXFE);
   //return 0;
 *s = (*uart_dr & 0xff);
 return 1;
}
```

2. ***Why does it work in this particular test code?***

It works because the program reacts to the key pressed. However, it never enters the loop:

```
while (0 == uart_receive(UART0, &c)) {
     count++;
     if (count > 50000000) {
       uart_send_string(UART0, "\n\rZzzz....\n\r");
       count = 0;
```

```
    }
  }
```

3. *Why would it be an interesting change in this particular setting?*

It is an interesting setting because we can then avoid executing useless instructions inside the loop.

7.2 Adding Printing

We provided you with the code of a kernel-version of printf, the function called ”kprintf” in the file ”kprintf.c”.
Add it to the makefile so that it is compiled and linked in.

```
all: startup.o main.o kprintf.o
   $(TOOLCHAIN)-ld $(LDFLAGS) startup.o main.o kprintf.o -o kernel.elf
   $(TOOLCHAIN)-objcopy -O binary kernel.elf kernel.bin
[...]
kprintf.o: kprintf.c
   $(TOOLCHAIN)-gcc $(CFLAGS) kprintf.c -o kprintf.o
```

Look at the function ”kprintf” and ”putchar” in the file ”kprintf.c”.
Why is the function ”putchar” calling the function ”uart send”?

Putchar is print chars through UART0, so it makes sense to use the function uart_send

Use the function kprintf to actually print the code of the characters you type and not the characters themselves.

```
   kprintf(&c);
   /*if (c == 13) {
```

```
    uart_send(UART0, '\r');
    uart_send(UART0, '\n');
  } else {
    uart_send(UART0, c);
  }*/
}
```

Hit the following special keys:

- ***left and right arrow.***
- ***backspace and delete key.***

Explain what you see.

When I press left and right arrow, the cursor on the terminal is moving to left and right. However the backspace and the delete key do not work.

7.3 Line editing

The idea is now to allow the editing of the current line:

• Using the left and right arrows
• Using the ”backspace” and ”delete” keys

First, experiment using the left/right arrows... and the backspace/delete keys...

• Explain what you see
• Explain what is happening?

Now that you understand, write the code

<u>References:</u>

https://www.tutorialspoint.com/gnu_debugger/gdb_debugging_symbols.htm

https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm