Pale INF422 – Composants d'un système informatique
INF422 Final Exam – Components of a Computing System

2009–2010

- L'examen dure 3 heures.
  The exam lasts 3 hours.
- Tous documents autorisés.
  All documents are authorized.
- Il est impératif de commenter les programmes et de justifier vos réponses.
  Programs must be commented and every answer must be justified.

English Text

Let us study the design of the computing component of a LED display like the one presented on Figure 1.

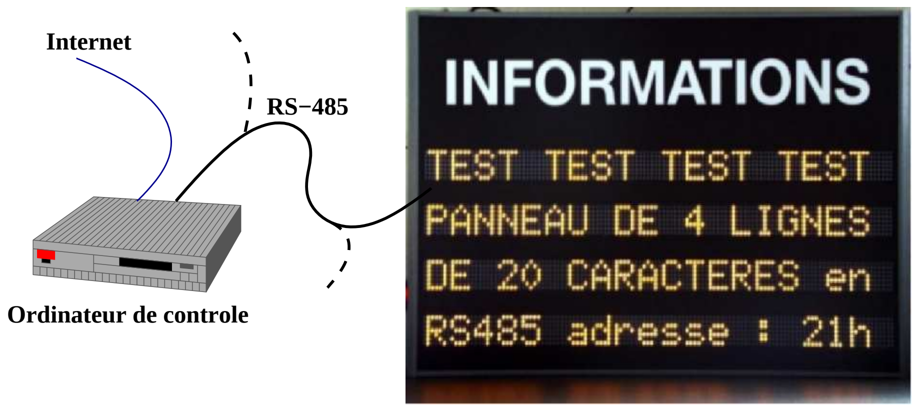

Figure 1: LED display

This display shows 4 lines of 20 characters. It is driven from an control computer embedded into a box and connected to the Internet. The interface between the control computer and the display uses a serial link following the RS-485 standard, to be detailed later. This link allows to connect multiple electronic devices on the same link: in our case, it is a set of LED displays (for example, in a train station). The control computer driving all these displays is based on an ARM processor and runs the Linux kernel.

1 Character encoding

For a long time, ASCII was the default character encoding, adapted to the most frequent characters in English. ASCII characters are encoded on the 7 least significant bits of a byte — hence their value lies between 0 and 127.

The UTF-8 standard, much more recent, is the default to represent international characters (accents, special symbols, other alphabets, non-alphabetical characters). ASCII characters retain their code in UTF-8, to make the extended code compatible with older programs. The *other* characters are coded on *multiple* bytes, up to 4.

We will study the coding principles involved in the UTF-8 standard. Non-ASCII characters are partitioned among 3 classes: the most frequent (e.g., accented characters) represented over 2 bytes, frequent ideograms over 3 bytes, and rare symbols over 4 bytes.

Question 1.1

Propose an encoding scheme for non-ASCII characters over 2 to 4 bytes following this partitioning. Some bits will be reserved to the identification of the character class (2, 3 or 4 bytes) and others to number the characters themselves; within each class, indicate the number of bits available for character numbering.

Justify informally the non-ambiguity (unicity) of the proposed coding principle.

Question 1.2

To enhance the robustness of UTF-8 decoding, its designers had the idea to strengthen the properties of the encoding to make it *auto-synchronizing*: reading any byte in the text allows immediately to decide if this byte is the first one coding a character or not. This way, any position in a UTF-8 string defines a valid sub-string (possibly skipping up to the first 3 bytes). Modify your encoding principle to make it auto-synchronizing.

Question 1.3

When representing integers, the control computer sorts bytes according to the *little-endian* ordering (the default for LINUX on the ARM processor), but the special-purpose circuit of the display uses the *big-endian* ordering. Is it necessary to convert multi-byte characters, and if so, how?

2 Serial link

The RS-485 serial link is a physical layer to communicate among remote devices. It is inexpensive, involving only two wires in half-duplex mode — no simultaneous two-way communications. The signal is defined by the voltage difference between the two wires. It may connect up to 32 electronic devices across significant distances.

In our setting, all devices have been configured to communicate bytes (bit-per-bit) at a rate compatible with the electrical features of the serial link (a few kilobytes per second). The RS-485 standard does not define a specific protocol to arbitrate among the devices attempting to transmit a signal.

Our protocol uses a unique "token" circulating among the devices. This token may be implemented as a boolean variable on each device: it indicates which device owns the right to emitting data on the serial link. The device owning the token — computer or display — emits continuously. If there is nothing specific to emit, it emits the 0-valued byte. The control computer initially owns the emission token.

Each display has a unique identifier between 1 and 30 (maximum 30 displays). The main computer has identifier 0.

Question 2.1

To indicate that it is going to send a message, the device owning the token starts by sending the sequence of bytes corresponding to the string `BEGIN`, coded in UTF-8 (equivalent to ASCII for these characters). This sequence is followed by the sequence corresponding to the message itself, and ends with `END`. Propose a message format to implement the following commands:

- transfer the token to another device — computer or display;
- send a sequence of n bytes to another device.

Question 2.2

Extend the previous format to add a command allowing device k owning the token to test the connection with an (other) device k' and the liveness of this device.

Show the list of messages communicated over the serial link through such a test.

Question 2.3

What happens in case of failure of the device owning the token? Distinguish the case of failure depending on whether it is in the process of sending a command or continuously emitting 0.

Question 2.4

Assuming that the control computer does not fail (otherwise there is not much to do...), modify the protocol to preserve the communications with the displays remaining active and connected. You may use a predefined timeout constant to detect a failure.

In the following, we will make the simplifying assumption that displays do not fail.

3 Driver for the serial link

At the level of the control computer, one wishes to abstract this serial link and the associated communication protocol. This involves a dedicated device driver. It implements the functions of a character device, and in particular the following functions implementing the `read()` and `write()` system calls within the driver:

- `rs485read(`*`buf`*`, `*`n`*`)`: reads up to *`n`* bytes from the serial link and stores them in buffer *`buf`*, then returns the number of bytes effectively read;
- `rs485write(`*`buf`*`, `*`n`*`, `*`id`*`)`: writes up to *`n`* bytes from buffer *`buf`* to the serial link, targetting device *`id`*, then returns the number of bytes effectively written.

To read or write a byte on the serial link, these two functions read or write a byte at a specific memory address determined when booting the serial device (*memory-mapped I/O*). We will simulate this behavior in a Java program (absurd for privileged, kernel code, but necessary in the context of the INF422 course); we assume that any read or write access to the static field `rs485link` corresponds respectively to the reception or the transmission of a byte on through the serial link.

The read operation is "destructive": each read access removes a byte from the input buffer of the controller of the serial device (UART). Analogously, each write access adds a byte to the output buffer of the controller.

Question 3.1

Something is missing to implement reliable communications over the serial link; what is it? Indication: the input and output buffers of the serial link are bounded, and it is necessary to respect a precise rate in transmitted bytes per second to implement a reliable communication.

We will ignore this difficulty in the following, and we will assume that it is possible to read or write as fast as possible in the static field `rs485link` to handle every byte one after another.

Question 3.2

Reading from the serial link is a blocking operation: the call to `rs485read()` returns only when the message has been received. Write a Java program implementing this function. It waits for the reception of a message designated to the controlling computer, then stores the *`n`* first bytes of its contents (or the full message if it is shorter) in the byte array *`buf`* and returns the number of bytes effectively read.

Question 3.3

The writing function tests that the computer owns the token, and if not waits until it obtains it (do not forget to handle the reception of the corresponding message). Then it writes the *`n`* first bytes of array *`buf`* to the serial link. To simplify, we suppose that no error may occur during the transmission, hence the function always returns *`n`*. Implement this function in Java.

Question 3.4

Comment on the qualities of your programs, in terms of reactivity of the kernel (implementing this function) and in terms of energy consumption for the control computer. Without modifying the code itself, propose some directions to improve it, possibly relying on additional hardware support.

4 Driver for display management

We now define the format of the commands recognized by the displays. Each display holds 4 lines of 20 characters.

- The `CLEAR` command has size $n = 1$ and is coded by a single 0-valued byte. It clears the whole display.
- The `MOVE` command has size $n = 2$ and is coded by a 1-valued byte followed by the position of the character on which to position the cursor. Position 0 corresponds to the first character in the upper-left corner, position 79 to the last character in the lower-right corner.
- The `PRINT` command is coded by a 2-value byte followed by $n - 1$ octets coding for a UTF-8 string. It prints this string (continuing to the next line if necessary) starting from the current position, and without erasing preceding or subsequent characters.

Question 4.1

Message exchanges are always initiated by the control computer. The displays reacts to commands, in particular to indicate the end of some displaying operation. Define a single format for the end-of-operation message corresponding to the three previous commands.

Question 4.2

Implement the display of message `Troisième !` on the third line of display 17, using functions `rs485read()` and `rs485write()`.

Question 4.3

Indicate the list of bytes communicated over the serial link when displaying this message. You will represent ASCII characters with the letter itself and the non-ASCII character `è` with a code of your choice compatible with the representation defined in the first exercise.

Question 4.4

We suppose that the device driver providing the three display operations `CLEAR`, `MOVE` and `PRINT` has been implemented. Propose a reasonable matching system call for each one of these operations, among `read()`, `write()` and `ioctl()`.

Question 4.5

When using this driver, each display is designated with a special character device file `/dev/display`*`ID`* where *`ID`* is the identifier of the display (from 1 to 30). Why not create a unique special file shared by all displays?

What else needs to be specified for the driver in charge of the displays to interpret input-output (system calls) on one of these special files as commands to the corresponding display.

Question 4.6

We suppose that the UNIX command `ls -l /dev/cdrom` executed on the control computer prints:

```
brw-rw---- 1 root cdrom 11, 0 2009-11-06 09:04 /dev/cdrom
```

What realistic message would be printed by the command `ls -l /dev/display17`, explaining what this message means and why you selected these meta-data for device special file `/dev/display17`.

5 Application

We assume that all Java applications are run with the rights of a plain user.

Question 5.1

We wish to restrict display access to a certain group of users (among which the plain user executing the applications belongs). Leveraging in particular the features of the file system, explain how to enforce this access policy.

Question 5.2

Write a shell script to create the 30 device special files associated with the displays with the adequate meta-data.

Question 5.3

Write a Java program to cyclically scroll a string over the first line of *each* display. You will use a thread and a different string for each display (if you lack of inspiration, the thread number will do).