Embedded systems – Exercise #4

1 Exercise purpose

In this exercise you will complete your simple cellular device by letting it store the received SMS messages on a FLASH device. You will use the same FLASH device for which you have written a driver in exercise 2.

Also – your system, and yourself, are now ready for **optimizations** (as Donald Knuth once said: "*We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**…*").

2 Device description

Your cellular device will be composed of the same elements as it was on the previous exercise, only this time, a NOR FLASH will be used to save the messages, so it will be available even if the device is turned off and on again.

3 Storing the data to flash

Now, whenever an SMS is received from the network, it should be saved to the FLASH device. Also, deleting an SMS should cause it to be deleted from the FLASH.

You are expected to write some kind of fail-proof file system, which will abstract the NOR FLASH in an easy to use API. The API should allow you to write, over-write, read and delete SMS messages. You might also want to add some kind of API to list existing SMS messages.

You are required to save at least the last 100 messages – sent and/or received – so they can be read later by the user.

Beware of data lost!

The device is expected to consider all possibilities, so that no data will be lost. Consider, **for example**, the case where the device fails (battery is over, or the user throws it on the wall and the battery falls out) *exactly* after it has written the first 27 characters of an SMS to the flash. In this case, it is expected that the SMS will not be added to the list of SMS messages which are present on the device, and that the file system will stay intact – all the previous messages case still be read from the FLASH.

Note that you cannot assume anything about the atomicity or the order of the FLASH actions; the device can even fail in the middle of updating a single byte, changing only some of the bits and leaving the rest as before of the crash. The same goes with block erase operation – the FLASH might fail leaving any subset of the bits not "turned up".

4 Optimizations

You should try to optimize the following parameters:

1. **Optimize** Space, in this priority (former is higher)
    a. **Minimize** used ROM size.
    b. **Minimize** used RAM size.
    c. **Minimize** used FLASH size.
2. **Optimize** power consumption.

You should consider these optimization goals when you design your device FW, and implement accordingly.

In the README file submitted with your solution, you must support the design decisions you made towards these goals.

You'll perform power consumption optimization by putting the ARC processor into sleep mode whenever possible. Read about the *SLEEP* instruction in the *ARC-600 programmer's guide*, and about the *_sleep()* compiler-intrinsic function in the *C/C++ programmer's guide*.

Look for the debugger's 'icnt' and 'sleep' simulator registers that keep track how many instructions were executed ('icnt' register) and how many "did not executed" when the processor was in sleep mode ('sleep' register).

Read in the *Debugger User's Guide* how to profile a program execution to find out where your program is spending its time.

5 Flash bootloader (10 points bonus)

An additional 10 bonus points will be given for implementing a "bootloader".

A flash bootloader is a mechanism in which the major part of the code is saved into the flash, and only a small portion of the code (the "bootloader") is programmed to the ROM, that on the device startup knows how to load the rest of the code from the flash into the RAM and continue the execution from the RAM.