Lab: strings on an industrial bus

1 Objective

Continue the project of *Lab pointers*. Until now, the program read the level sensors by calling `sensor_read_level()`. In a real plant the sensors are far from the PC, and they are read through a **bus**: the PC sends a **command** as a string of text, and an input module answers with another string. In this lab you will write the two functions that talk to that module. At the end of this lab you will have:

- built a command with `snprintf()`, using the `<inttypes.h>` macros and checking that it fits in its array;
- analysed a response with `sscanf()`, checking its return value and rejecting every wrong response;
- seen why the control characters (`'\r'`) matter in a protocol, and why they must not be printed as they are.

> Estimated time: 60–90 minutes. Keep following the Barr-C style guide, and fix every warning as if it were an error.

2 The plant

The four level sensors are now connected to the channels 0 to 3 of an **analog input module** of the NuDAM family, at address `01` (hexadecimal). The modules of a plant share an **RS-485 bus**. The PC reaches the bus through its serial port and an RS-232 to RS-485 converter, such as the ND-6520 of the picture.

```
+------+  RS-232  +---------+  RS-485 bus  +------------------------+
|  PC  |----------| ND-6520 |------+-------| analog input module 01 |
+------+          +---------+      |       |  ch 0..3: tanks 0..3   |
                                   |       +------------------------+
                                   +------- (other modules)
```



RS-232 to RS-485 converter ND-6520 (NuDAM family)

The modules understand ASCII commands. To read one channel, the PC sends `#AAN<CR>`, where `AA`

is the address of the module (2 hexadecimal digits), `N` is the channel and `<CR>` is the carriage return, `'\r'` in C. In this plant the module returns the reading in hexadecimal format:

Message	Example	Meaning
Command	`#012<CR>`	Read channel 2 of module 01
Valid response	`>0BFF<CR>`	The reading is 0x0BFF (3071)
Error response	`?01<CR>`	Module 01 did not understand the command
(nothing)		No module answered: wrong address, or no `<CR>`

The manuals write `<CR>` because `'\r'` is invisible. This lab shows it the same way.

> You do not need any hardware: the module `nudam_sim`, given by your instructor, simulates the bus and the module. Its readings come from the `sensor` module, so the results will be those of *Lab pointers*.

3 Requirements

- *Lab pointers* finished: the folder `lab_pointers`, which builds without warnings and prints the expected output of that lab. If your version does not work, ask your instructor for the reference version.
- The file `lab_strings_given.zip`, given by your instructor.

4 Starting point

1. Copy the whole *Lab pointers* project into a new folder named `lab_strings`:

```
C:\...> cd %USERPROFILE%\c_projects
C:\...> xcopy lab_pointers lab_strings /E /I
C:\...> cd lab_strings
C:\...\lab_strings> mingw32-make clean
C:\...\lab_strings> del answers.txt
```

2. If you did the optional parts of *Lab pointers*, it does not matter: the new `main.c` replaces yours.
3. Extract `lab_strings_given.zip` into the project folder, replacing `src\main.c`. It contains:
   - `include\nudam_sim.h` and `src\nudam_sim.c`: the simulated bus. **Do not modify them.**
   - `include\nudam.h` and `src\nudam.c`: the module you will complete. `nudam.c` has the two functions with an empty body that always returns -1.
   - `src\main.c`: the new main program. Before the cycles it runs two tests of your functions, *Bus test* and *Parser test*. In each cycle it reads every tank through the bus, with the private

function `read_tank_level()`: if anything fails, the reading is `SENSOR_FAULT_VALUE`.

4. Build and run with `mingw32-make run`, and type `6`. The program works, but the module `nudam` is still empty, so it cannot read anything: the bus test shows `build error`, the parser test shows `ERROR` in every row, and every tank is `SENSOR FAULT` in every cycle. Read `read_tank_level()` in `main.c` before you go on.

5 Part 1: build the command

Complete `nudam_build_read()` in `src/nudam.c`:

```
int nudam_build_read(char *p_command, size_t command_size,
                     uint8_t address, uint8_t channel);
```

It writes in `p_command` the command that reads `channel` of the module at `address`, for example `"#012\r"`. It returns 0 on success, and -1 if `channel` is greater than `NUDAM_MAX_CHANNEL`, or if the command does not fit in `command_size` characters.

1. Write the command with **one call to** `snprintf()`. The address goes with 2 hexadecimal digits in upper case, and the command ends with `'\r'`, not with `'\n'`.
2. Check the value returned by `snprintf()` with the pattern of the slides: if it is negative, or greater than or equal to `command_size`, the command did not fit.

> `address` and `channel` are `uint8_t`: use the macros of `<inttypes.h>`. The address is printed with `"%02" PRIX8`, and the channel with `"%" PRIu8`. The compiler joins the adjacent string literals.

3. Build and run. The bus test must show the dialogue with the module:

```
Bus test (module 01, cycle 0)
  tank  command    response    raw
     0  #010<CR>   >04B0<CR>   parse error
     1  #011<CR>   >07D0<CR>   parse error
     2  #012<CR>   >0800<CR>   parse error
     3  #013<CR>   >0DAC<CR>   parse error
```

   The module answers, but `nudam_parse_read()` is still empty: that is Part 2.

4. **Forget the `<CR>`.** Remove `\r` from your format temporarily, build and run. Keep what the bus test shows for question 1, then restore the `\r`.
5. **A command that does not fit.** In `test_bus()` (`main.c`), replace temporarily `sizeof(command)` with `5U` in the call to `nudam_build_read()`, build and run. Keep the result for question 2, then restore `sizeof(command)`.

6 Part 2: analyse the response

Complete `nudam_parse_read()`:

```
int nudam_parse_read(const char *p_response, uint16_t *p_raw);
```

It analyses a response such as `">0BFF\r"`. If the response is a valid reading, it writes the value in `*p_raw` and returns 0. Otherwise it returns -1 and **does not modify** `*p_raw`.

1. Analyse the response with **one call to** `sscanf()`, with this format: the literal character `>`, a hexadecimal number of at most 4 digits read into a `uint16_t`, and then one more character with `%c`, which must be the `'\r'`.
2. The response is valid only if `sscanf()` assigned **both** items and the last character is `'\r'`. Read the value into a **local** variable, and copy it to `*p_raw` only when you know the response is valid.

> To read a `uint16_t` in hexadecimal with at most 4 digits, use `"%4" SCNx16`. A literal character in the format of `sscanf()` must appear in the input: if the first character is not `>`, `sscanf()` stops and returns 0.

3. Build and run. The tests must show:

```
Bus test (module 01, cycle 0)
  tank  command    response    raw
     0  #010<CR>   >04B0<CR>   1200
     1  #011<CR>   >07D0<CR>   2000
     2  #012<CR>   >0800<CR>   2048
     3  #013<CR>   >0DAC<Cr>   3500

Parser test
  response    result
  >0BFF<CR>   OK, raw = 3071
  >04B0<CR>   OK, raw = 1200
  ?01<CR>     ERROR
  >0BFF       ERROR
  >XYZ<CR>    ERROR
  (empty)     ERROR
```

Type `6`: from *Cycle 0* to the end, the output must be identical to the one of *Lab pointers*. In cycle 2, the module returns `>FFFF<CR>` for tank 2: the response is valid, and it is `tank_update()` that detects the faulty sensor.

7 Part 3: questions

Answer these questions in a plain text file named `answers.txt`, in the project folder. Short answers are fine, but justify them.

1. What did the bus test show in Part 1, step 4, and why? Look for the reason in `nudam_sim_transaction()`. Would the module answer if the command ended with `'\n'`?
2. In Part 1, step 5, what did the bus test show? Which value did `snprintf()` return, and what was left in `command`? What is the minimum size of an array for the command, and why? In `main.c`, what is the value of `sizeof(command)`, and what is the value of `strlen(command)` after a successful call? Why must the function receive `sizeof(command)` and not `strlen(command)`?
3. For the response `">0BFF"` (the `'\r'` did not arrive), which value does `sscanf()` return, and what has it already written? Why does `nudam_parse_read()` use a local variable instead of writing directly into `*p_raw`?
4. In `test_bus()`, replace temporarily the two calls to `print_padded()` with `printf("%s%s\n", command, response);` and run. What do you see on the screen for each tank, and why? Restore the code afterwards. Why does a program that shows a protocol need a function such as `nudam_sim_print()`?
5. Which command reads channel 3 of a module at address 10 (decimal)? Replace temporarily `"%02" PRIX8` with `"%" PRIu8` in your format, and run: which commands are sent, and why does no module answer? Restore the format afterwards.
6. In Java, you could build the command with `String.format()` and read the value with `Integer.parseInt(text, 16)`. What does Java do when the text is not a valid number? What does C do, and what happens if the program does not check the value returned by `sscanf()`?

8 Going further (optional)

1. **Robust keyboard input.** Replace the `scanf()` of the number of cycles with a private function in `main.c`:

```
static int read_cycles(void);
```

   It asks for the number of cycles until the user types a valid value (1 to 6), reading each line with `fgets()` and analysing it with `sscanf()`. It returns -1 at the end of the input, and then

`main()` prints `No more input` and ends with `EXIT_FAILURE`. Run the program and type `abc`, `9` and `6`: the question must appear three times, and then the cycles must be the expected ones.

2. **Checksum.** The protocol can protect each message with a checksum: 2 hexadecimal digits with the sum of the codes of all the previous characters, modulo 256, placed just before the `<CR>`. For example, the checksum of `#010` is `B4`, so the command becomes `#010B4<CR>`, and the response becomes `>04B014<CR>`. When the checksum is enabled, the module ignores a command whose checksum is wrong.
   - Add to `nudam` the function `void nudam_set_checksum(bool enabled);`, and a private function that computes the checksum of a number of characters.
   - With the checksum enabled, `nudam_build_read()` appends it, and `nudam_parse_read()` reads exactly 4 data digits and 2 checksum digits, and checks them.
   - At the beginning of `main()`, call `nudam_sim_set_checksum(true)` and `nudam_set_checksum(true)`. The bus test must show `#010B4<CR>` and `>04B014<CR>` for tank 0, and the cycles must not change. Why does the parser test now show `ERROR` in every row?

9 Checklist and submission

Your lab is complete if:

- ☐ `mingw32-make clean` followed by `mingw32-make` builds the project without any warning.
- ☐ `nudam_sim.h`, `nudam_sim.c`, `sensor`, `area`, `volume`, the `Makefile` and the files in `.vscode` have not been modified.
- ☐ `nudam_build_read()` uses one `snprintf()` with the `<inttypes.h>` macros, and checks its return value.
- ☐ `nudam_parse_read()` uses one `sscanf()`, checks its return value and the final `'\r'`, and does not modify `*p_raw` on error.
- ☐ `answers.txt` answers the six questions.
- ☐ With 6 cycles, the program prints the expected output below.

The final output of the program, typing `6`, must be:

```
Plant configuration: 4 tanks
  tank  radius(m)  height(m)  low(%)  high(%)  capacity(L)
     0       0.50       2.00      20       90       1570.8
     1       0.75       3.00      15       85       5301.4
     2       0.40       1.50      25       90        754.0
     3       1.00       2.50      10       95       7854.0
  plant capacity: 15480.2 L

Bus test (module 01, cycle 0)
  tank  command    response    raw
     0  #010<CR>   >04B0<CR>   1200
     1  #011<CR>   >07D0<CR>   2000
     2  #012<CR>   >0800<CR>   2048
     3  #013<CR>   >0DAC<CR>   3500

Parser test
  response    result
  >0BFF<CR>   OK, raw = 3071
  >04B0<CR>   OK, raw = 1200
  ?01<CR>     ERROR
  >0BFF       ERROR
  >XYZ<CR>    ERROR
  (empty)     ERROR

Number of cycles (1-6): 6

Cycle 0
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   1200     0.586     29.3      460.3  NORMAL
     1   2000     1.465     48.8     2589.2  NORMAL
     2   2048     0.750     50.0      377.1  NORMAL
     3   3500     2.137     85.5     6712.8  NORMAL
  plant volume: 10139.4 L (65.5 % of capacity), 0 sensor fault(s)

Cycle 1
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   1800     0.879     44.0      690.5  NORMAL
     1   1500     1.099     36.6     1941.9  NORMAL
     2   2050     0.751     50.1      377.5  NORMAL
     3   3550     2.167     86.7     6808.7  NORMAL
  plant volume: 9818.5 L (63.4 % of capacity), 0 sensor fault(s)

Cycle 2
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   2400     1.172     58.6      920.6  NORMAL
     1   1000     0.733     24.4     1294.6  NORMAL
     2  65535         -        -          -  SENSOR FAULT
```

```
    3   3600     2.198     87.9     6904.6  NORMAL
  plant volume: 9119.8 L (58.9 % of capacity), 1 sensor fault(s)

Cycle 3
  tank    raw  level(m)  fill(%)  volume(L)  status
    0   3000     1.465     73.3     1150.8  NORMAL
    1    600     0.440     14.7      776.8  LOW
    2   2047     0.750     50.0      376.9  NORMAL
    3   3700     2.259     90.4     7096.4  NORMAL
  plant volume: 9400.8 L (60.7 % of capacity), 0 sensor fault(s)

Cycle 4
  tank    raw  level(m)  fill(%)  volume(L)  status
    0   3600     1.758     87.9     1380.9  NORMAL
    1    400     0.293      9.8      517.8  LOW
    2   2049     0.751     50.0      377.3  NORMAL
    3   3850     2.350     94.0     7384.1  NORMAL
  plant volume: 9660.1 L (62.4 % of capacity), 0 sensor fault(s)

Cycle 5
  tank    raw  level(m)  fill(%)  volume(L)  status
    0   3800     1.856     92.8     1457.6  HIGH
    1    300     0.220      7.3      388.4  LOW
    2   2048     0.750     50.0      377.1  NORMAL
    3   3950     2.411     96.5     7575.9  HIGH
  plant volume: 9799.0 L (63.3 % of capacity), 0 sensor fault(s)

Volume history (L), valid readings only
  tank  samples        min        max       mean
    0        6      460.3     1457.6     1010.1
    1        6      388.4     2589.2     1251.5
    2        5      376.9      377.5      377.2
    3        6     6712.8     7575.9     7080.4
```

> Submit the folders `src` and `include` and the file `answers.txt`, as indicated by your instructor. Do not submit the `build` folder: run `mingw32-make clean` first. Keep the project: the next labs start from it.

10 Troubleshooting

- `implicit declaration of function 'snprintf'` or `'sscanf'`**.** Include `<stdio.h>` in `nudam.c`.
- `expected ')' before 'PRIX8'` (or `'SCNx16'`). Include `<inttypes.h>` in `nudam.c`.
- `format '%x' expects argument of type 'unsigned int *'`**.** You read a `uint16_t` with

`%x`: use `"%4" SCNx16`.

- **The bus test shows `no answer`.** Check that the command ends with `\r` and that the address has 2 digits: `#010<CR>`.
- **The bus test shows the raw values, but the parser test accepts `>0BFF`.** You do not check that the last character is `'\r'`.