• Remember to register your attendance using the UoB Check-In app. Either

1. download, install, and use the native app[a] available for Android and iOS, or
2. directly use the web-based app available at

https://check-in.bristol.ac.uk

noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.

• The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

https://www.bristol.ac.uk/it-support

• The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

https://cs-uob.github.io/COMS30048/vm

• The purpose of the worksheet is to provide a) a tutorial-style introduction to selected technologies or concepts, and/or b) a means to explore them via hands-on tasks and challenges. Note that the worksheet is not assessed *at all*: if you are confident that you already understand the content, there is no problem with nor penalty for totally ignoring it.

• Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

COMS30048 lab. worksheet #1.1

Before you start work, download (and, if need be, unarchive[a]) the file

https://assets.phoo.org/COMS30048_2025_TB-2/csdsp/sheet/lab-01-1.tar.gz

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a] For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-01-1.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive→Open` menu item to open `lab-01-1.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-01-1.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b] For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

1. Introduction

This unit adopts an overtly applied, hands-on approach throughout. Doing so is rationalised by considering that, for topics in applied cryptography[1] and cryptographic engineering[2], it both a) represents the most effective way to learn, and b) mirrors the challenges, and style of work you might face in an industrial setting. As part of the ongoing (open source) SCALE[3] project, we have developed a hardware platform to support this approach: within the context of this unit, you use a packaged, supported instance of said platform.

A boxed SCALE kit[a] constitutes

- 1 SCALE host board,
- 2 SCALE target boards (supporting LPC1114FN28 and LPC1313FBD48 target devices respectively),
- 1 PicoScope 2206B oscilloscope,
- 2 oscilloscope probes,
- 2 type-B USB to type-A USB cables,
- 2 male SMA-A to male BNC coaxial cables.

Although we cannot let you take the kit away, there should be enough kits for one per student; at the start (resp. end) of each lab. slot that uses it, collect a kit from (resp. return the kit to) a lab. demonstrator. Try to pack the content neatly before return: this will help whoever uses it next, in the sense that they can start work as quickly as possible. There is a chance the kit you collect is either incomplete or defective. Such an event *should* be rare of course, but *if* you identify a problem then let a lab. demonstrator know (e.g., versus just returning it as is): this allows us to solve the problem, or just provide a replacement.

[a] We are very interested in both positive and negative feedback about the kit. If, for instance, you do something interesting or "off piste" with it, we want to hear: take a photograph and drop us an email, for example, or tweet and mention `@BristolCS` so we can pick it up!

The goal of this lab. worksheet is to a) introduce the SCALE development board, then b) explore how to make use of it, i.e., write and execute software for it.

Note that although the lab. worksheet does conclude with a set of hands-on tasks and challenges, at this point they are intentionally biased towards reading and understanding the material (vs. more active alternatives, e.g., programming). It is *crucial* not to view this as optional effort: carefully working through the admittedly detailed content should allow you to more easily and rapidly engage with longer-term challenges (e.g., those relating to a given coursework assignment).

[1] http://en.wikipedia.org/wiki/Cryptography
[2] http://en.wikipedia.org/wiki/Cryptographic_engineering
[3] http://www.github.com/danpage/scale

2. An overview: the SCALE development board

2.1. *Physical* hardware

The SCALE development board[4] is, in a sense, fairly typical: the idea is to offer a minimal platform for developing and executing software on a given micro-controller[5]. It is formed by combining a host board (or motherboard[6]), which houses all the generic functionality, with a target board (or daughterboard[7]), which houses the specific functionality for a micro-controller and can be switched between as need be.

By default, the SCALE kit should include a development board formed from a) the host board, and b) the LPC1313FBD48 target board, plus c) a protective acrylic case: Figure 1 presents an image of it. Note that the board has been correctly pre-configured (i.e., all jumper settings are correct): you need not, and therefore *must not* dismantle it or alter this default form.

At a high level, and with reference to Figure 2, a set of pertinent features and functionality are summarised by the following:

- The micro-controller is an NXP LPC1313FBD48 [4], housing a 32-bit ARM Cortex-M3 [2] processor core; this core conforms to the ARMv7-M [1] architecture, supports the full Thumb-1 and Thumb-2 instruction sets, and, unlike some other M-class cores, includes hardware support for single-cycle (32×32)-bit multiplication.
- A UART[8] on the micro-controller is connected, via a bridge, to the type-B USB connector. Having attached it to a workstation, this acts as a) a power supply, b) a means of performing general-purpose communication between the micro-controller and workstation, *and* c) a way to program (i.e., transfer an executable image from the workstation to) the micro-controller.
- The instantaneous power consumption of the micro-controller can be sampled via a pipeline of on-board components; this includes a) a shunt resistor, b) an amplifier, and c) a low-pass filter. The resulting signal can be fed, via the associated SMA connector, to an oscilloscope.

2.2. *Emulated* hardware

Although the SCALE kit cannot be taken away, we recognise the preference to sometimes work *outside* the lab. environment. The SCALE project includes some infrastructure to support this mode of work: using Unicorn[9] and Capstone[10] in combination, it *emulates* the physical hardware using software alone. On one hand, the emulator is limited[11] in a number of respects; for example, it focuses exclusively on the execution of programs by the processor core, meaning it lacks various features (e.g., some peripherals, the power consumption acquisition pipeline) of the physical hardware. On the other hand, however, it *is* compatible *enough* with the physical hardware (e.g., it can emulate the same compiled binary) to be of use for out-of-lab. development of software.

2.3. Software

It is common for development board hardware to be complimented by some software, namely a so-called Board Support Package (BSP)[12]. In line with the general remit of a BSP, two software elements are provided to support use of the SCALE development board: it includes

- a generic build system, supporting the compilation, linkage, and execution of bare-metal C programs, and
- a library, which acts as an abstraction layer for any low-level functionality (e.g., peripherals) of the development board.

The following function prototypes and associated descriptions summarise *pertinent* (versus *all*) features in the latter:

[4] http://en.wikipedia.org/wiki/Microprocessor_development_board
[5] http://en.wikipedia.org/wiki/Microcontroller
[6] http://en.wikipedia.org/wiki/Motherboard
[7] http://en.wikipedia.org/wiki/Expansion_card
[8] http://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
[9] http://www.unicorn-engine.org
[10] http://www.capstone-engine.org
[11] It is important to recognise the limitations of this emulator, either inherent (e.g., those stemming from the fundamental difference between physical and emulated hardware) or extrinsic (e.g., those stemming from deficiencies in the implementation); for example, it currently does not model the status LEDs or GPIO more generally, nor acquisition of power consumption traces (although this is at least plausible; see, e.g., [5]). Provision of this emulator is somewhat experimental: we actively seek feedback on and improvements to it, which, in part, will only be viable with your help as users of it.
[12] http://en.wikipedia.org/wiki/Board_support_package

- **`bool scale_init( scale_conf_t* conf )`**
  Initialise the development board, including pertinent peripheral devices, using the configuration specified by `conf`; returns **true** if configuration succeeded, or **false** otherwise.
- **`void scale_delay_us( int us )`**
  Delay (i.e., wait or idle) for a period of ∼ `us` micro-seconds. Note that the concrete delay is calibrated to suit the clock frequency, so as to match the period of time specified.
- **`void scale_delay_ms( int ms )`**
  Delay (i.e., wait or idle) for a period of ∼ `ms` milli-seconds. Note that the concrete delay is calibrated to suit the clock frequency, so as to match the period of time specified.
- **`void scale_gpio_wr( scale_gpio_pin_t id, bool x )`**
  Write (or set) the state of a GPIO output pin specified by `id` ∈ {`SCALE_GPIO_PIN_GPO`, `SCALE_GPIO_PIN_TRG`}, updating it to `x` ∈ {**false**, **true**}.
- **`bool scale_gpio_rd( scale_gpio_pin_t id )`**
  Read (or sample) the state of a GPIO input pin specified by `id` ∈ {`SCALE_GPIO_PIN_GPI`} and return said state, say `r` ∈ {**false**, **true**}, as the result.
- **`bool scale_uart_wr_avail()`**
  Check if UART is available for write (i.e., would doing so block or not).
- **`bool scale_uart_rd_avail()`**
  Check if UART is available for read (i.e., would doing so block or not).
- **`void scale_uart_wr( scale_uart_mode_t mode, uint8_t x )`**
  Perform a blocking *or* non-blocking (per `mode`) write of an 8-bit byte `x` to the UART.
- **`uint8_t scale_uart_rd( scale_uart_mode_t mode )`**
  Perform a blocking *or* non-blocking (per `mode`) read of an 8-bit byte from the UART, returning said byte as the result.

3. Some hands-on tasks and challenges

The following Sections provide a tutorial-style introduction to use of the SCALE development board. Using a terminal[13] to execute the commands, you should be able to progress step-by-step to at *least* the point where you have executed an example program on the physical and emulated hardware.

Keep in mind that although you can simply copy-and-paste the commands, it remains important to understand their purpose. The short-term goal is to equip you with experience of use in the long(er)-term: if a given step, concept, or detail is unclear, *now* is the time to ask questions.

3.1. Prepare the hardware

1. Take the (grey) USB cable, and connect the development board to the workstation you are using; you should see the power status LED lit.
2. The kernel *should* recognise a new device[14] as having been connected, and then create a device node in order to interact with it. You can check this via

   ```
   ls -l /dev/scale-board
   ```

   with any error (e.g., `cannot access '/dev/scale-board': No such file or directory`) suggesting that the device has not been recognised: ask for help!

3.2. Prepare the software

1. Update your `${PATH}`[15] environment variable by executing

   ```
   export PATH="${PATH}:/opt/arm-gnu-toolchain/13.2.rel1/bin"
   export PATH="${PATH}:/opt/lpc21isp/1.97"
   export PATH="${PATH}:/opt/picoscope/bin"
   ```

[13] That is, within a BASH shell (or prompt, e.g., a terminal window) or similar.
[14] `lsusb` lists the device as a FTDI FT232R [3], which is a component tasked with supporting UART-based communication (by the target board) over a USB-based connection (on the host board).
[15] http://en.wikipedia.org/wiki/PATH_(variable)

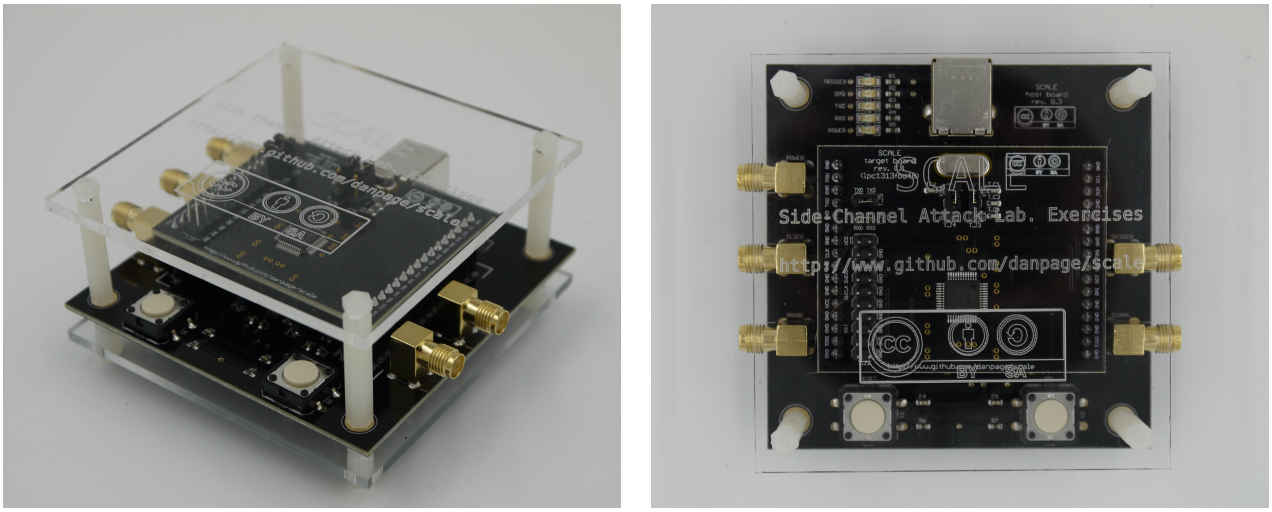


Figure 1: *A physical illustration of the SCALE development board.*

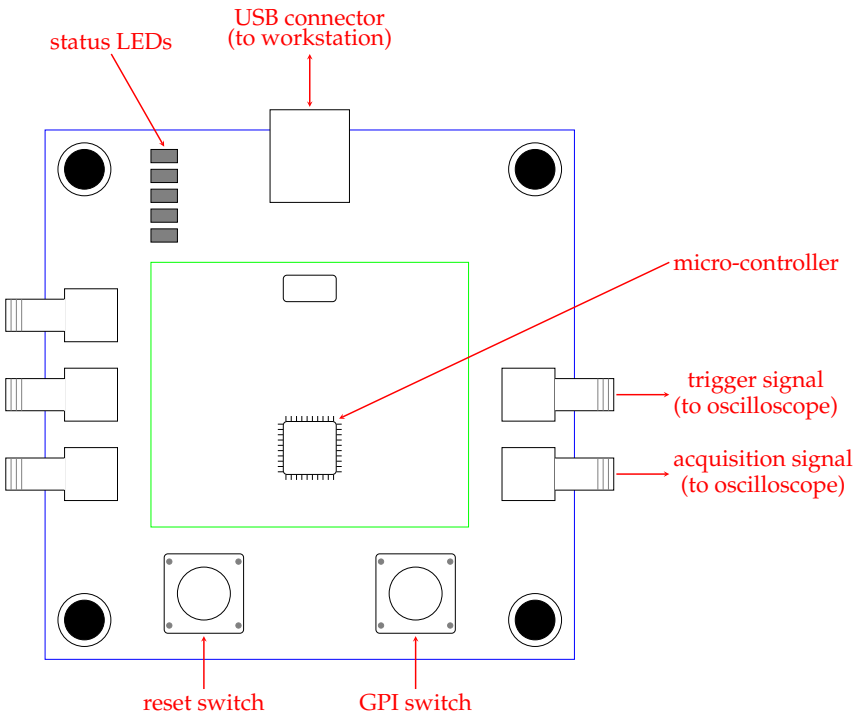


Figure 2: *A logical illustration of the SCALE development board; the blue, external PCB is the SCALE host board, whereas the green, internal PCB is a SCALE target board.*

2. Check said update worked correctly by executing

```
which arm-none-eabi-gcc
which lpc21isp
which picoscope
```

   noting that any reported error (e.g., `no arm-none-eabi-gcc in ...` or similar) suggests it did not: ask for help!
3. Clone the repository:

```
git clone --branch COMS30048_2025 http://www.github.com/danpage/scale-hw.git ./scale-hw
```

4. Fix the working directory:

```
cd ./scale-hw
```

5. Execute some steps to configure any material in the repository:

   (a) Execute the repository configuration script:

```
source ./bin/conf.sh
```

   (b) Store the repository path in an environment variable, so it can be referenced easily later:

```
export REPO="${PWD}"
```

   (c) Store the target device in an environment variable, so it can be referenced easily later:

```
export TARGET="lpc1313fbd48"
```

6. Execute some steps to build material in the repository:

   (a) Build any material related to the Python virtual environment

```
make venv
```

   then activate it

```
source ./build/venv/bin/activate
```

   (b) Build any material related to the BSP:

```
make --directory="./src/scale/target/${TARGET}" build
```

   Note that in doing so you may observe compilation *warnings*, but, provided there are (terminal) compilation *errors*, you can safely ignore them. Once successfully built, you can retain and so reuse material related to both the Python virtual environment and BSP: both are useful, for example, in lab. worksheet #1.2.

3.3. Prepare then execute an example program

Figure 3 captures an example[16] program for the development board. The source code is heavily annotated to describe each major feature or step; it realises largely meaningless, "hello world"[17] style functionality.

3.3.1. Using *physical* hardware

1. Fix the working directory:

```
cd ${ARCHIVE}/board
```

2. Build any material related to the example:

```
make PROJECT="helloworld" build
```

3. The processor core has a bootloader[18] which supports In-System Programming (ISP) via the USB connection. Initiate the programming process by executing

```
make PROJECT="helloworld" program
```

   then, having done so, perform the following manual steps

[16] For reference, this example duplicates `${REPO}/src/scale/example/example.[ch]` within the repository.
[17] http://en.wikipedia.org/wiki/"Hello,_World!"_program
[18] http://en.wikipedia.org/wiki/Booting, or see [4, Section 21.2]

```
8  #ifndef __HELLOWORLD_H
9  #define __HELLOWORLD_H
10
11 #include <scale/scale.h>
12
13 #endif
```

(a) *${ARCHIVE}/board/helloworld.h*

```
8  #include "helloworld.h"
9
10 /** Initialise the SCALE development board, then loop indefinitely.  Each loop
11   * iteration will
12   *
13   * 1. flash the trigger status LED on and off,
14   * 2. flash the GPO     status LED on or  off (depending on whether or not the
15   *    GPI switch is pressed), and
16   * 3. write the string "hello world" to the UART.
17   */
18
19 int main( int argc, char* argv[] ) {
20   // select a configuration st. the external 16 MHz oscillator is used
21   scale_conf_t scale_conf = {
22     .clock_type        = SCALE_CLOCK_TYPE_EXT,
23     .clock_freq_source = SCALE_CLOCK_FREQ_16MHZ,
24     .clock_freq_target = SCALE_CLOCK_FREQ_16MHZ,
25
26     .tsc               = false
27   };
28
29   // initialise the development board
30   if( !scale_init( &scale_conf ) ) {
31     return -1;
32   }
33
34   char x[] = "hello world";
35
36   while( true ) {
37     // read  the GPI     pin, and hence switch : t   <- GPI
38     bool t = scale_gpio_rd( SCALE_GPIO_PIN_GPI        );
39     // write the GPO     pin, and hence LED    : GPO <- t
40               scale_gpio_wr( SCALE_GPIO_PIN_GPO, t     );
41
42     // write the trigger pin, and hence LED    : TRG <- 1 (positive edge)
43               scale_gpio_wr( SCALE_GPIO_PIN_TRG, true  );
44     // delay for 500 ms = 1/2 s
45     scale_delay_ms( 500 );
46     // write the trigger pin, and hence LED    : TRG <- 0 (negative edge)
47               scale_gpio_wr( SCALE_GPIO_PIN_TRG, false );
48     // delay for 500 ms = 1/2 s
49     scale_delay_ms( 500 );
50
51     int n = strlen( x );
52
53     // write x = "hello world" to the UART
54     for( int i = 0; i < n; i++ ) {
55       scale_uart_wr( SCALE_UART_MODE_BLOCKING, x[ i ] );
56     }
57   }
58
59   return 0;
60 }
```

(b) *${ARCHIVE}/board/helloworld.c*

Figure 3: *An example program for the SCALE development board.*

(a) the `Makefile` executes `lpc21isp`, which then waits, trying to connect to the bootloader,
(b) press and hold the (right-hand) GPI switch,
(c) press and hold the (left-hand) reset switch,
(d) release the (left-hand) reset switch,
(e) transfer via `lpc21isp` starts,
(f) release the (right-hand) GPI switch,
(g) transfer via `lpc21isp` finishes,

and, finally, reset the board:

(a) press and hold the (left-hand) reset switch,
(b) release the (left-hand) reset switch.

Note that the sequenced use of switches on the development board once the transfer is initiated from the workstation: this can be awkward at first, so demands some practice to master.

Once the process is complete, the example should start to execute: as the source code in Figure 3 suggests, you should find that

- the trigger status LED will flash on and off,
- the GPO status LED is on or off depending on whether or not the GPI switch is pressed, and
- the string "hello world" is repeatedly written to the UART.

4. Via another terminal, execute PuTTY to interact with the physical hardware and hence executing program, e.g., observe the string being written to the UART:

```
make PROJECT="helloworld" putty-physical
```

Note that any advanced configuration must be done via the PuTTY GUI (rather than CLI).

3.3.2. Using *emulated* hardware

1. Fix the working directory:

```
cd ${ARCHIVE}/board
```

2. Build any material related to the example:

```
make PROJECT="helloworld" build
```

3. Execute the example via the emulator:

```
make PROJECT="helloworld" emulate
```

Note that the instruction throughput of the emulator depends on the host workstation, but, either way, is likely to be *much* lower than the physical hardware. In addition, the 500ms delays in Lines #36 and #40 of Figure 3 are no longer calibrated correctly.

4. Via another terminal, execute PuTTY to interact with the emulated hardware and hence executing program, e.g., observe the string being written to the UART:

```
make PROJECT="helloworld" putty-emulated
```

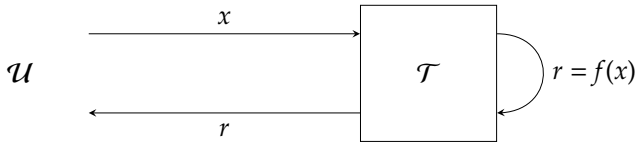
Note that any advanced configuration must be done via the PuTTY GUI (rather than CLI), and that you may have to wait longer to see any output versus use of the physical hardware: be patient!

3.4. Develop some infrastructure for computational off-load

Concept. Computational off-load[19] is often couched in the language of high-performance computing. There, it relates to off-loading a resource intensive (computational) task from some less-capable to a more-capable device (e.g., a co-processor) or platform (e.g., the cloud). However, the same concept applies where the task relates to additional, perhaps niche functionality. Use of a special-purpose dongle[20] to supplement the capabilities of a general-purpose computer is a common example involving a local form of interaction (e.g., via USB); use of a Hardware Security Module (HSM)[21] is another, more security-specific example, typically involving a more remote form of interaction (e.g., across a network).

[19] http://en.wikipedia.org/wiki/Computation_offloading
[20] http://en.wikipedia.org/wiki/Dongle
[21] http://en.wikipedia.org/wiki/Hardware_security_module

Imagine the SCALE development board is used as a special-purpose dongle in order to provide some security-specific functionality. In a generic sense, interaction between a user $\mathcal{U}$ and the dongle $\mathcal{T}$ can be illustrated as follows:



That is the user sends a request (e.g., input data x) to the dongle via a UART (and hence across the USB) connection, the device computes the result $r = f(x)$ of applying some function f, then the dongle sends a response (e.g., output data r) to the user via a UART (and hence across the USB) connection. One can obviously consider various embellishments to this protocol (e.g., multiple items of input and/or output data, a command i to select which f is applied, and so on), but, even so, it acts as a useful starting point.

Example. Consider an example scenario where the function f is somewhat trivial, e.g., it reverses bytes in x to produce r, and all communicated values are represented using (a particular type of) octet strings; this is further expanded upon in Appendix B.

- Using the starting point provided, i.e., by altering `dongle.c` and `dongle.h`, develop a program for the development board which models the dongle $\mathcal{T}$ in this scenario. For example, start by implementing the functions whose prototypes

  ```
  void octetstr_wr( const uint8_t* x, int n_x )
  ```

  and

  ```
  int octetstr_rd( uint8_t* r, int n_r )
  ```

  hint that they respectively a) read an octet string from the UART, decoding it into a byte sequence `r` of maximum length `n_r`, and b) write an octet string to the UART, encoding it from a byte sequence `x` of given length `n_x`. In doing so, keep in mind the challenge of correctly managing the associated EOL semantics; this is further expanded upon in Appendix C.
- At first, you should test your implementation manually: once it is executing on the development board, execute PuTTY to interact with it by providing the input x and then inspecting the output r. However, the supposed goal is to support computational off-load: this suggests it could be useful to write a client that models the user $\mathcal{U}$, or, put another way, acts on behalf of it.

  The idea then is to have this client program send x and receive r to and from the development board, which demands interacting directly and programatically with the serial device (versus indirectly and manually via PuTTY). The challenge of doing so is greatly reduced if/when a higher-level interface is used via a suitable library. For example, `pyserial`[22] for Python and `libserialport`[23] for C both offer very convenient solutions. Such a client, written in Python by using `pyserial`, is provided; as well as capturing an example of serial communication with the development board, it includes reference functions for conversion to and from octet strings.

References

[1] *ARM Architecture Reference Manual: ARMv7-M edition*. Tech. rep. DDI-0403E. ARM Ltd., 2014. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403e.b/index.html (see p. 3).

[2] *Cortex-M3 Technical Reference Manual*. Tech. rep. DDI-0337E. ARM Ltd., 2006. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/index.html (see p. 3).

[3] *FT232R USB UART IC*. Tech. rep. FT000053. Future Technology Devices International Ltd., 2015. URL: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf (see p. 4).

[4] *LPC1311/13/42/43 User manual*. Tech. rep. UM10375. NXP B.V., 2012. URL: http://www.nxp.com/docs/en/user-guide/UM10375.pdf (see pp. 3, 6).

[5] D. McCann, E. Oswald, and C. Whitnall. "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages". In: *USENIX Security Symposium*. 2017, pp. 199–216 (see p. 3).

[22] http://www.github.com/pyserial/pyserial
[23] http://sigrok.org/wiki/Libserialport

A Frequently Asked Questions (FAQs)

The `/dev/scale-board` device node doesn't seem to exist; the device is not recognised. This suggests either 1) a problem with the hardware itself, and/or 2) a problem with configuration of associated software (e.g., driver); even if you did not explicitly check the device node, you *may* have inferred the device is not recognised because use of it (e.g., via PuTTY) failed somehow. As some first steps,

1. check the USB cable is corrected plugged in,
2. check the power LED is illuminated,
3. power cycle[24] the device (i.e., "turn it off and on again" by removing and reinserting the USB cable).

If the above fail, the cause is something more subtle or significant: ask for help in the lab.

I tried to program the development board with a compiled executable, but the process fails. Although there are various *potential* causes for this, by far the most obvious relate to some misstep during the programming process (e.g., incorrect use of the switches): this *can* be awkward at first, although less so once you have gone through it a few times. Beyond that, it is worth keeping the following in mind:

- If the device node is already in use when the programming process is started, then `lpc21isp` will be unable to open it and thus abort. The solution is simple: before starting the programming process manually, terminate any processes (e.g., PuTTY) which are using the device node.
- If the development board gets into a state where no valid program is available, the programming process starts automatically (i.e., without a need to use the buttons); this *might* happen if a previous attempt to program it failed (an so was incomplete), for example. It is easy to not notice such a case, and thus manually start the programming process as normal. The solution is simple: before starting the programming process manually, check that it has not already started automatically.

The implementation task was a challenge for me: do you have any advice?! Although the task is relevant to the coursework assignment because your implementations of `octetstr_wr` and `octetstr_rd` are intended to be reusable, the first thing to say is don't worry: completing the task is far less important than engaging with it, and so, e.g., gaining experience with the equipment and associated development workflow. Beyond that, it is worth keeping the following in mind:

- Writing C programs for development board is not easy, but you can make doing so a *lot* easier if you 1) adopt a step-by-step versus all-or-nothing approach wherever possible, 2) separate board-specific from board-agnostic code; develop the latter using a "friendly" platform, e.g., a workstation, then port[25] it to the development board, and 3) keep things simple, e.g., by assuming valid input, initially at least.
- As an example of 1), consider that the task involves reading some input x, performing some computation $r = f(x)$, then writing some output r. The idea is that implementing the steps out-of-order can make sense. For example, imagine that you first implement the output step (with an "empty" computation meaning $r = x$, and a fixed hard-coded input), then implement the input step, then implement the computation step: doing so can help avoid the challenge of debugging the computation step *until* the input and/or output steps are reliable.
- As an example of 2), consider an implementation of `octetstr_rd` which is split into two parts:

```
int _octetstr_rd( uint8_t* r, int n_r, char* x ) {
  ...
}

int  octetstr_rd( uint8_t* r, int n_r          ) {
  char x[ 2 + 1 + 2 * ( n_r ) + 1 ]; // 2-char length, 1-char colon, 2*n_r-char data, 1-char terminator

  for( int i = 0; true; i++ ) {
    x[ i ] = scale_uart_rd( SCALE_UART_MODE_BLOCKING );

    if( x[ i ] == '\x0D' ) {
      x[ i ] = '\x00'; break;
    }
  }

  return _octetstr_rd( r, n_r, x );
}
```

The idea is that `octetstr_rd` is board-specific (it will use the UART to read a CR-terminated string), whereas, in contrast, `_octetstr_rd` is board-agnostic (it will parse an octet string into a byte array). You *could* develop the latter using a workstation then port it to the development board, therefore, rather than directly develop it on the development board; doing so potentially yields easier and quicker development cycles (due to the

[24] http://en.wikipedia.org/wiki/Power_cycling
[25] http://en.wikipedia.org/wiki/Porting

more familiar, better supported development environment of the workstation). You *could* even attempt the same thing with the former, by abstracting the means of reading and writing input and output: modulo the EOL semantics, for example, note that `scale_uart_rd` ≃ `getc`[26] via `stdin` and `scale_uart_wr` ≃ `putc`[27] via `stdout`.

- Iff. you opt to port your C program as suggested above, strictly adhering to standard best-practices can help: selected examples might include
  - use a strict set of compiler options (e.g., `-Wall`),
  - avoid undefined or implementation defined behaviour (e.g., any uninitalised variables),
  - keep in mind potential differences between architectures (e.g., endian'ness), and so avoid assumptions based on them,
  - be precise with the type system (e.g., by using `int32_t` versus `int`).

I want to include other source files in the build process, e.g., `Y.c` and `Y.h`, not just `X.c` and `X.h`: how can I do that? The build system is not very clever, but it *is* possible to do this.

By default, it uses `PROJECT` to compute

```
PROJECT_HEADERS += ${PROJECT}.h
PROJECT_SOURCES += ${PROJECT}.c
```

which are then used during compilation. It is possible to *preset* these variables: in `Makefile`, e.g., after defining `PROJECT`, add the lines

```
export PROJECT_HEADERS = Y.h
export PROJECT_SOURCES = Y.c
```

Although arguably not too elegant a solution, this results in the build system appending to these preset variables and thereby including `Y.c` and `Y.h` as required.

When building my implementation, I get an error similar to `No rule to make target 'X.map'`. Although there are other potential causes, it is likely that you 1) copied `helloworld.c` to `X.c`, then 2) changed `PROJECT` from `helloworld` to `X`] in an attempt to start developing your own implementation. The problem is, the build system is not very clever: it assumes there is *always* a `X.h` as well as `X.c`, which causes the build to fail. However the solution is simple: create `X.h`, e.g., by executing `touch X.h`, even if, as in this case, it is an empty file.

When building my implementation, I get an error similar to `undefined reference to _sbrk`. Although there are other potential causes, it is likely that you used either `malloc` (or similar) and/or `printf` (or similar) in your implementation: both require access to a heap, which, in turn, requires the standard C library and the linker (plus linker script) to cooperate. By default the BSP does not support a heap. Although it is possible possible to add such support yourself, doing so is quite challenging. Therefore, it is important to assess whether you actually *need* to do so: often, it is possible to avoid use of the function causing the error, e.g., via use of static rather than dynamic memory allocation.

[26] http://man7.org/linux/man-pages/man3/getc.3p.html
[27] http://man7.org/linux/man-pages/man3/putc.3p.html

B Representation of byte sequences using (hexadecimal) octet strings

The term octet[28] is normally used as a synonym for byte, most often within the context of communication (and computer networks). Using octet is arguably more precise than byte, in that the former is *always* 8 bits whereas the latter *can*[29] differ. A string is a sequence of characters, and so, by analogy, an octet string[30] is a sequence of octets: ignoring some corner cases, it is reasonable to use the term "octet string" as a synonym for "byte sequence".

To represent a given byte sequence, we use what can be formally termed a (little-endian) length-prefixed, hexadecimal octet string. However, doing so requires some explanation: each element of that term relates to a property of the representation, where we define a) little-endian[31] to mean, if read left-to-right, the first octet represents the 0-th element of the source byte sequence and the last octet represents the $(n-1)$-st element of the source byte sequence, b) length-prefixed[32] to mean n, the length of the source byte sequence, is prepended to the octet string as a single 8-bit[33] length or "header" octet, and c) hexadecimal[34] to mean each octet is represented by using 2 hexadecimal digits. Note that, confusingly, hexadecimal digits within each pair will be big-endian: if read left-to-right, the most-significant is first. For convenience, we assume the term octet string is a catch-all implying all such properties from here on.

An example likely makes all of the above *much* clearer: certainly there is nothing complex involved. Concretely, consider a 16-element byte sequence

```
uint8_t x[ 16 ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }
```

defined using C. This would be represented as

$$\hat{x} = \texttt{10:000102030405060708090A0B0C0D0E0F}$$

using a colon to separate the length and value fields:

- the length (LHS of the colon) is the integer $n = 10_{(16)} = 16_{(10)}$, and
- the value (RHS of the colon) is the byte sequence $x = \langle 00_{(16)}, 01_{(16)}, \ldots, 0F_{(16)} \rangle = \langle 0_{(10)}, 1_{(10)}, \ldots, 15_{(10)} \rangle \equiv \mathbf{x}$.

Note that the special-case of an empty byte sequence *is* valid: now starting with the 0-element byte sequence

```
uint8_t x[ 0 ] = { }
```

defined using C, setting n to 0 and x to an empty byte sequence yields the representation

$$\hat{x} = \texttt{00:}$$

vs. say an empty or null string, which, in contrast, is an invalid octet string.

[28] http://en.wikipedia.org/wiki/Octet_(computing)

[29] http://en.wikipedia.org/wiki/Byte, for example, details the fact that the term "byte" can be and has been interpreted to mean a) a group of n bits for $n < w$ (i.e., smaller than the word size), b) the data type used to represent characters, or c) the (smallest) unit of addressable data in memory: although POSIX mandates 8-bit bytes, for example, each of these cases permits an alternative definition.

[30] Note the octet string terminology stems from ASN.1 encoding; see, e.g., http://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One.

[31] http://en.wikipedia.org/wiki/Endianness

[32] http://en.wikipedia.org/wiki/String_(computer_science)

[33] Although it simplifies the challenge associated with parsing such a representation, note that use of an 8-bit length implies an upper limit of 255 elements in the associated byte sequence.

[34] http://en.wikipedia.org/wiki/Hexadecimal

C UART communication and EOL semantics

The concept of the End Of Line (EOL) character (aka. newline[35]) *seems* trivial, and, in theory, is: in essence it is a control character we expect to be associated with pressing a return (or enter) key. In practice, however, the control character, or characters, used will differ based on various factors. The most obvious example is the use of Carriage Return (CR), i.e., the byte $0D_{(16)}$ (or C escape character `'\r'`), and/or Line Feed (LF), i.e., the byte $0A_{(16)}$ (or C escape character `'\n'`), characters. Note that much of the terminology[36] stems from (electronic) typewriters. For example, CR moves the type element (or cursor) to the start of the *same* line, whereas LF moves the type element to the same position on the *next* line; in combination (i.e., CR+LF) realises what we normally consider to be a new line (or express verbally as "start a new line").

As such, different EOL semantics are possible: for example a Linux will typically use LF, whereas Windows will typically use CR+LF. You may have already *observed* this difference, when extra control characters appear in a text file (e.g., C source code) first written on a Windows-based platform then transferred to a Linux-based alternative. The same difference is important when engaging in serial communication, e.g., with a given development board. Although not complicated, this *does* need some care. Arguably the easiest, and hence recommended approach is to use the same EOL semantics as PuTTY:

1. By default, PuTTY emulates a VT100 terminal[37]. This means pressing the return key will transmit CR.
2. Match those semantics in your implementation. For example, one might read a line of input by consuming characters until a CR is encountered; at this point, the CR is "eaten" (or discarded) and the line deemed complete.
3. When receiving, PuTTY can be configured so it injects an implicit LF and/or CR. This can be useful, since receiving CR without LF, for example, can induce (visually) odd behaviour in the terminal (per a typewriter, lack of LF produces "overwritten" text).
4. By default the `pyserial` function `readline` waits for a LF to mark the EOL, so a CR-based alternative means taking an alternative approach. Viable approaches include a) writing a bespoke `readline` replacement or b) using the `TextIOWrapper` wrapper, which allows an explicit selection of EOL semantics.

[35] http://en.wikipedia.org/wiki/Newline
[36] See, e.g., http://en.wikipedia.org/wiki/Carriage_return
[37] http://en.wikipedia.org/wiki/VT100