EEE 158: Electrical and Electronics Engineering Laboratory V

Module 5: Serial Communications & USART

Introduction

So far, we have performed exercises where the microcontroller board was the only processor in town; in other words, our systems up to now have been *stand-alone*. In almost all practical applications, however, our microcontroller does not completely work on its own, but is *integrated* as part of a larger system. As such, our microcontroller needs to be able to communicate with other subsystems, like protocol-converter modules and other microcontrollers.

The PIC32CMLE00 / LS00 / LS60 microcontrollers include up to six serial-communications peripherals, hereafter referred to as “SERCOM”, whose signals are multiplexed as alternate functions on one or more pins. The peripheral is so-named because depending on its configuration, it can operate in one of the following modes:

- USART, internally-clocked or externally-clocked
- SPI in Host or Client mode
- I^{2}C in Host or Client mode

This module focuses on the internally-clocked USART mode; more commonly known as UART. The SPI and I^{2}C interfaces are discussed in their own modules.

IMPORTANT: Use of microcontroller peripherals usually precludes use of certain pins as GPIOs, but it does NOT preclude setting the appropriate bit/s for port direction/type on the precluded pins. It is the programmer’s responsibility to set the appropriate port-control bits, the exact bit/s of which must be referred to in the *device-specific* datasheet.

Resources

Here are some additional resources that may be of help for using this module:

- Arm v8-M architecture reference manual
- ASCII table <https://man7.org/linux/man-pages/man7/ascii.7.html>
  - The first 32 characters are used for terminal control, among them tab (`‘\t’`, `0x09`); carriage-return (`‘\r’`, `0x0D`); linefeed (`‘\n’`, `0x0A`), and escape (`‘\e’`, `0x1B`). The last is special in that it forms the basis of the ANSI escape codes.
- ANSI escape codes <https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797> <https://en.wikipedia.org/wiki/ANSI_escape_code>
  - These codes underpin the modern terminals in macOS and Linux, among other things.
- VT1xx Reference Manual <https://vt100.net/docs/vt102-ug/contents.html>
- Xterm Reference Manual <https://invisible-island.net/xterm/xterm-function-keys.html>

Configuring the Peripheral

NOTE: You are expected to read the datasheets of not only the microcontroller chip itself, but also that of the Curiosity Nano development board, as well as of the Arm architecture.

Accompanying this document is a ZIP archive containing a sample MPLAB X project with the following files:

- `platform/*.c` – Hardware should be initialized in a file within this folder. These files also serve to abstract low-level details from your application logic, although you are expected to edit at least one file in accordance with the required specification/s.
- `platform.h` – Declares functions intended to be called from `main()`. Any source code wishing to gain access to the declarations needs to include this file.
- `main.c` – Main sample application logic is implemented in this file.

Configuration of the `SERCOM` peripheral in internally-clocked `USART` mode is done by performing the following steps. Note that like the `TCx` peripheral, `SERCOM` provides different views depending on operating mode; use the `USART_INT` view for the classic 16550-compatible UART.

1. Enable the APB clock/s for the appropriate SERCOM instance/s. For this module, use the SERCOM instance that is connected to the debugger on-board the Curiosity Nano.
2. Configure and enable the corresponding `GCLK` peripheral channel/s.
3. Trigger a reset of the peripheral by setting the `CTRLA.SWRST` bit to one.
4. Configure the peripheral for the following settings. Consult the datasheet to determine which register/s to write; the first register to be written post-reset must be the one containing operating-mode selection. Do NOT enable the transmitter nor receiver yet.

Operating Mode	# of stop bits	Character size	Parity Mode	Flow Control	TX/RX Pad
Internally-clocked USART	2	8 bit	None	None	`[0]/[1]`
Data Order	**FIFO**	**Baud Mode**	**Baud Rate**	**Samples / Bit**	
LSB first	Disabled	Arithmetic	38400	16	

5. Program the baud rate by writing the value to `SERCOM_BAUD` corresponding to the target baud rate. See the next section for more details.
6. Enable both the transmitter and receiver by setting the `CTRLB.RXEN` and `CTRLB.TXEN` bits.
7. Enable the peripheral itself by setting `CTRLA.ENABLE` to 1.

At this stage, we can already send data by writing to the data register (`DATA`); once the transmit buffer is empty (`INTEN.DRE=1`), we can load the next character into `DATA`, which clears the `INTEN.DRE` bit.

Receiving data is done by reading the data register (`DATA`) when the receive buffer is non-empty (`INTEN.RXC=1`); however, doing so as-is is bad because error information in the status register (`STATUS`) will be erased when `DATA` is read. Thus, when receiving data `STATUS` *must* be read and stored in a variable *before* reading the data in `DATA`. Checking for status is then done by inspecting the `STATUS` copy saved in the variable.

In practice, plainly using the technique above for reception is awkward; this is particularly troublesome at higher speeds, where data could be lost in transit. Solving these issues require the use of a FIFO queue, paired with SysTick/interrupt-based handling. At still higher data rates, additional specific features of the implementation (eg. DMA[1] capability) need to be utilized to maintain reliable transfer. These techniques are already at a more-advanced level; as this module is intended to be an introduction to USART, we will stick to the naïve implementation provided in the sample.

Programming the Baud Rate

For the PIC32CM LE00/LS00/LS60, the baud rate in asynchronous arithmetic mode is given by Equation (1); the f_{gclk} term represents the peripheral clock for the USART, as configured via the generic-clock (GCLK) peripheral, while the S term represents the number of samples for one bit.

$$f_{\mathrm{baud}} = \frac{f_{\mathrm{gclk}}}{S}\left(1 - \frac{\mathrm{SERCOM\_BAUD}}{65536}\right) \qquad \text{(Equation 1)}$$

More often than not, it is SERCOM_BAUD that needs to be computed when all other parameters are known. Equation (1) can thus be rearranged as such:

$$\mathrm{SERCOM\_BAUD} = 65536\left(1 - \frac{S \cdot f_{\mathrm{baud}}}{f_{\mathrm{gclk}}}\right) \qquad \text{(Equation 2)}$$

Because it is likely that the resulting value for SERCOM_BAUD will not be an integer value, there will exist a percentage error between the desired and actual baud rates; in such cases, the value to be programmed into `SERCOM_BAUD` should be the closest integer following standard rounding rules. Above a certain percentage (~1%), the USART begins to be unable to lock onto incoming data, resulting in corruption; as a result, asynchronous transmission can only work up to a certain extent.

Event-driven Programming

In our previous modules, we have been content with busy-looping on status values while a resource is still not ready. However, while in such a loop the processor is unable to do *anything*; in practical systems the end result could be an unresponsive system. While acceptable for the simplest of implementations, real-world scenarios place hard deadlines on system responsiveness which cannot be violated under *any* circumstances; such violations are treated as a catastrophic system failure. Examples include:

- Delayed parsing and/or generation of packets, causing dropped connections
- Delayed waveform generation for sinusoidal-PWM drives, causing inductor saturation and excess current flows

To avoid such scenarios, it is useful to “interleave” tasks within the main loop, as a form of cooperative multitasking. For example, rather than doing something similar to the following to toggle an LED:

```
    // … other code …
    while (!condition);
    toggle_led();
    // condition = false;    // Might be needed to prevent immediate retrigger
    // … other code …
```

1 DMA = Direct Memory Access; allows access to/from/between peripherals and memory without CPU involvement

we could do something similar to the following:

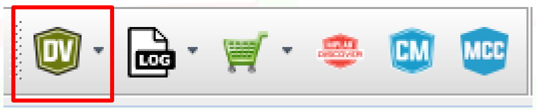
```
// … other code …
if (condition) {
     toggle_led();
     condition = false; // Required to prevent immediate retrigger
}
// … other code …
```

However, there is still a potential for missed events to occur – if, for example, `toggle_led()` takes too long to be called. One way to resolve this is to use SysTick, a facility provided by the Arm Cortex-M23 family to allow short periodic tasks to be done via interrupts, or for timing based on CPU clock frequencies; more information on this facility can be had by consulting both `platform/systick.c` and the Arm reference document. For longer tasks, a form of preemptive multitasking may be employed in addition to SysTick.[2] Finally, in practical systems some form of watchdog timer is usually implemented – forget to properly "ping" it (including proper pings done too early or too late), and it triggers a system reset. Most processors – PIC32CM LE00/LS00/LS60 included – are capable of indicating how a reset occurred (eg. power-on, external pin, watchdog timeout); the application code can use this to its advantage, like initiating some form of crash recovery.

Trying Out the Peripheral

In order to actually test the code on the board, we need a second device to communicate with the board over its serial line. Fortunately for us, the debugger on-board the Curiosity Nano board can also perform a dual role as a serial port that is already connected to certain pins of the on-board microcontroller.

On the PC side, Microchip provides the MPLAB Data Visualizer tool, which can be accessed either as a plugin within MPLAB X (install via *Tools* > *Plugins*, access via *Window* > *Debugging* or toolbar), or as a stand-alone program via a separate installer. When installed within MPLAB X, a toolbar icon like this should appear:

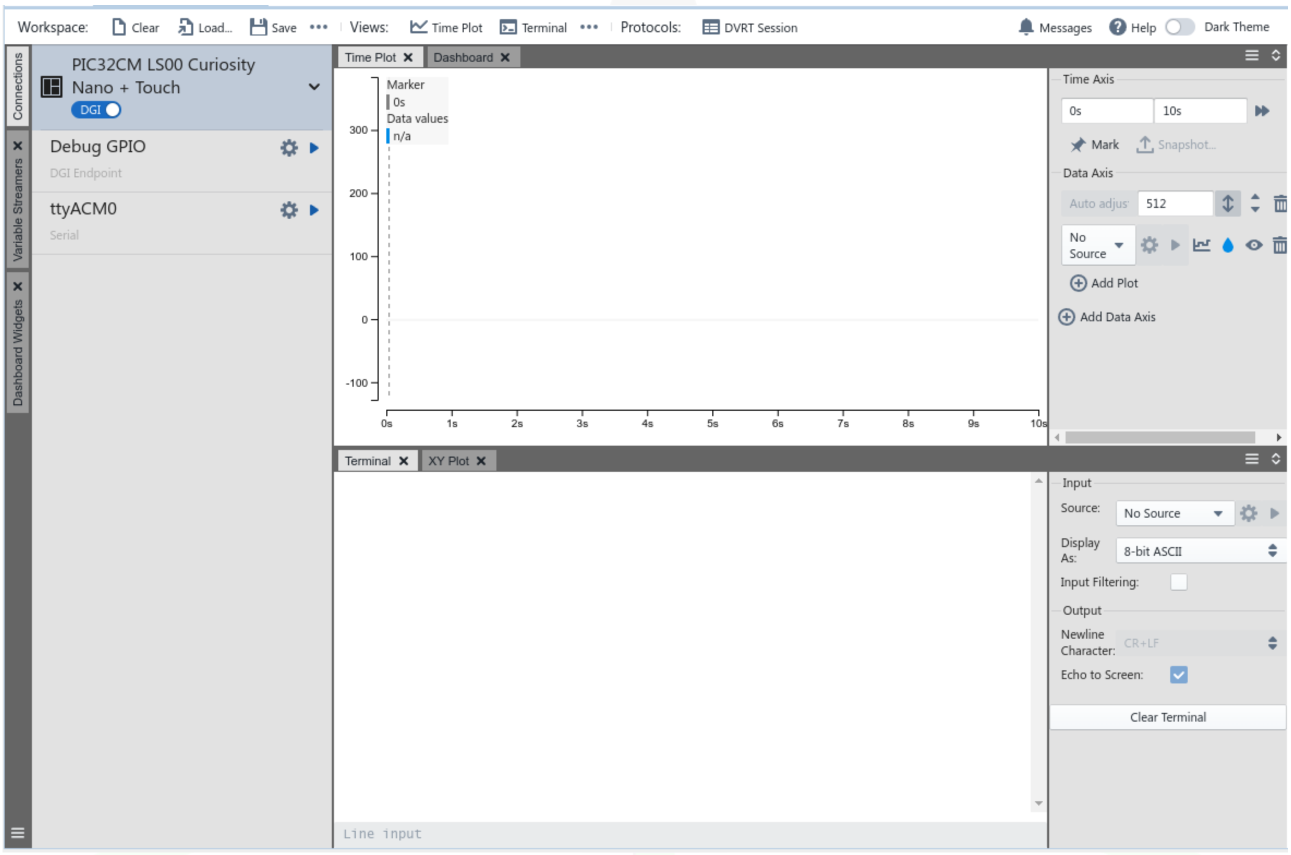


Alternatively, an external terminal program like PuTTY by Simon Tatham et al. (graphical), CoolTerm by Roger Meier (graphical) and GNU's `screen` (command-line) can be employed.

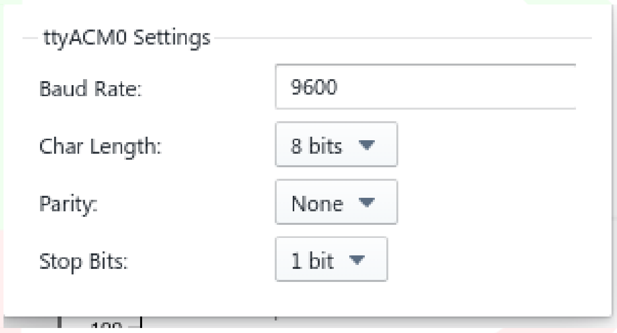
MPLAB Data Visualizer

1. Bring up the *Data Visualizer* pane, either via MPLAB or by launching the stand-alone version. You should get a window similar to the following figure; note that the board and its ports will only appear while plugged into your computer.

2 Not done here in EEE 158; may already be in the CoE 16x series



2. Configure the serial-port instance by selecting the “Gear” icon; you should get a pop-up window similar to the one to the right. The settings here must match those you set up within your board code.

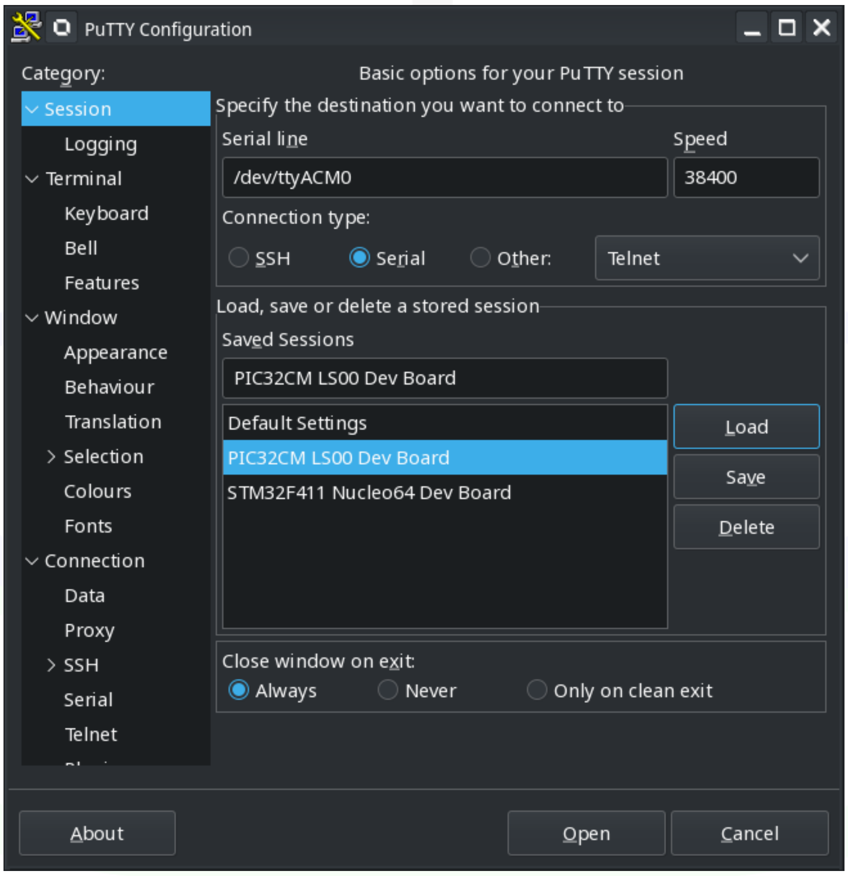

ttyACM0 Settings

Baud Rate:	9600
Char Length:	8 bits
Parity:	None
Stop Bits:	1 bit

3. Configure the terminal for the following:
   - Unicode display
   - No input filtering
   - Echo to screen disabled
   - Newline is CR+LF
4. Press the “Play” button within the serial-port instance to connect to the board. If successful, the button changes to a “Stop” button, which will disconnect from the board when pressed (and thence revert back to “Play”).

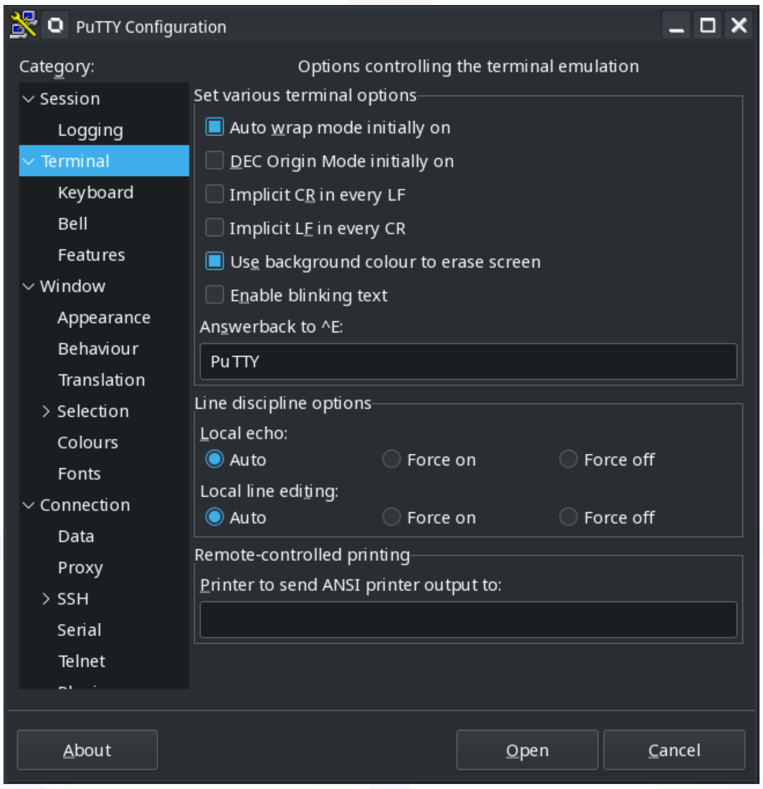
PuTTY Terminal

1. If not yet done, launch the program using its shortcut. You should get a window similar to the following figure; the correct line may be found using your OS tools (eg. *Ports (COM & LPT)* in Windows' *Device Manager*, `dmesg` on Linux).



Specify the destination you want to connect to
Serial line: /dev/ttyACM0
Speed: 38400
Connection type: SSH [✓ Serial] Other: Telnet
Load, save or delete a stored session
Saved Sessions: PIC32CM LS00 Dev Board
Default Settings
PIC32CM LS00 Dev Board
STM32F411 Nucleo64 Dev Board
Close window on exit: [✓ Always] Never Only on clean exit

2. Because PuTTY is intended to connect to a multitude of terminals past and present, in addition to the serial configuration (which must match those you configured in your board code) configure the terminal settings as follows:



Options controlling the terminal emulation

Set various terminal options

- [✓] Auto wrap mode initially on
- [ ] DEC Origin Mode initially on
- [ ] Implicit CR in every LF
- [ ] Implicit LF in every CR
- [✓] Use background colour to erase screen
- [ ] Enable blinking text

Answerback to ^E: PuTTY

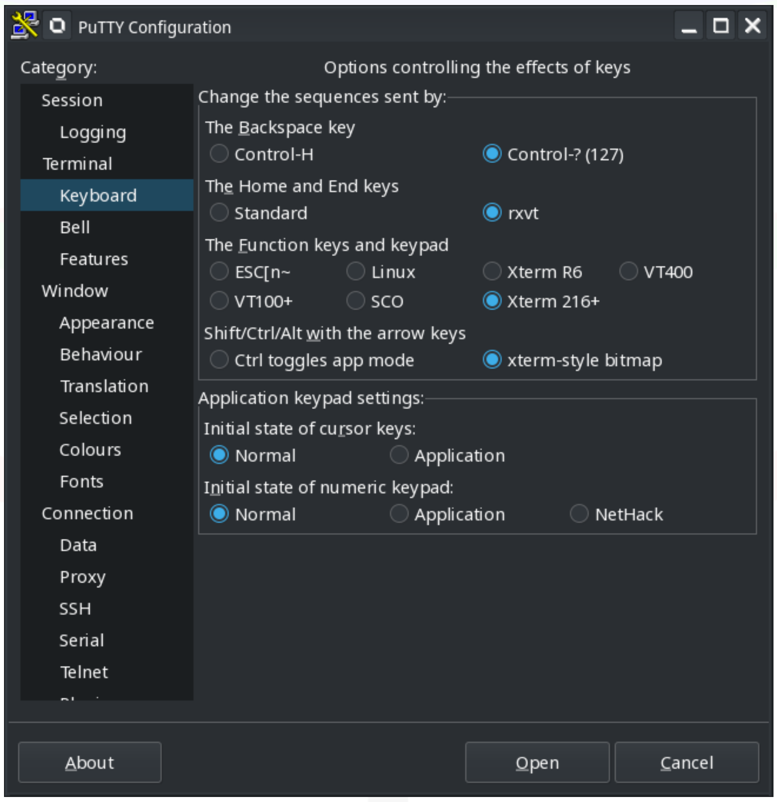
Line discipline options

Local echo: [✓ Auto] Force on Force off

Local line editing: [✓ Auto] Force on Force off

Remote-controlled printing

Printer to send ANSI printer output to:



Options controlling the effects of keys

Change the sequences sent by:

The Backspace key: Control-H [✓ Control-? (127)]

The Home and End keys: Standard [✓ rxvt]

The Function keys and keypad: ESC[n~ Linux Xterm R6 VT400 VT100+ SCO [✓ Xterm 216+]

Shift/Ctrl/Alt with the arrow keys: Ctrl toggles app mode [✓ xterm-style bitmap]

Application keypad settings:

Initial state of cursor keys: [✓ Normal] Application

Initial state of numeric keypad: [✓ Normal] Application NetHack

3. You may go to *Session* to save your settings. When done, click on “Open” to launch the session and connect to the board.

`screen` Command-line Tool

NOTE: This has been tested only on Linux. Your version of `screen` may be different.

Type the following command at a terminal to launch a session:

```
screen <device path> <settings>
```

Example:

```
screen /dev/ttyACM0 38400,-parenb,cstopb,cs8,raw
```

To determine the correct device, look through the output of `dmesg | less` or similar.

The settings string is described in detail in the `stty(1)` man page. In summary:

- `38400` is the baud rate.
- `-parenb` disables parity generation (transmit) and checking (receive). Removing the “-” prefix (aka. “`parenb`”) enables said functionality.
- `parodd` selects odd parity. If disabled (aka. “`-parodd`”), even parity is selected.
- `cstopb` indicates 2 stop bits. If disabled (aka. “`-cstopb`”), 1 stop bit is used.
- `cs8` selects 8-bit character transmission.
- `raw` treats the port as a true serial port – no “fancy” terminal features like line-editing.

Exercise: GPIO Monitoring and Control via PC

NOTE: You are expected to undertake some tests to determine the appropriate key sequence/s for your computer. If a specified key sequence is in conflict with OS-provided functionality, you may use a different combination; but this must be documented accordingly.

Hardware Configuration

- **G/P output:** `PA15`, active-HI (on-board LED)
- **G/P input:** `PA23`, active-LO (on-board pushbutton)
- **Serial port:** 57.6 kbaud, 8-bit character size, 16 samples/bit, even parity, no flow control, 1 stop bit

Application-code Specifications

- Pressing `<Home>` or `<Ctrl>+E` shall perform a full refresh of the banner shown on the PC-side terminal, showing at least the state of the on-board button (pressed or not pressed). Upon power-on/reset, the system must act as if this keypress was performed.
- Whenever the button state has changed, the PC-side terminal shall reflect the change. The terminal display must not appear garbled.
- There shall be five blinking settings for the LED, as shown in the following table. At system power-on/reset, the setting must be `#0`.

Setting #	`0`	`1`	`2`	`3`	`4`
ON-time	Zero	100 ms	100 ms	100 ms	Positive, indefinite
OFF-time	Positive, Indefinite	900 ms	400 ms	200 ms	Zero

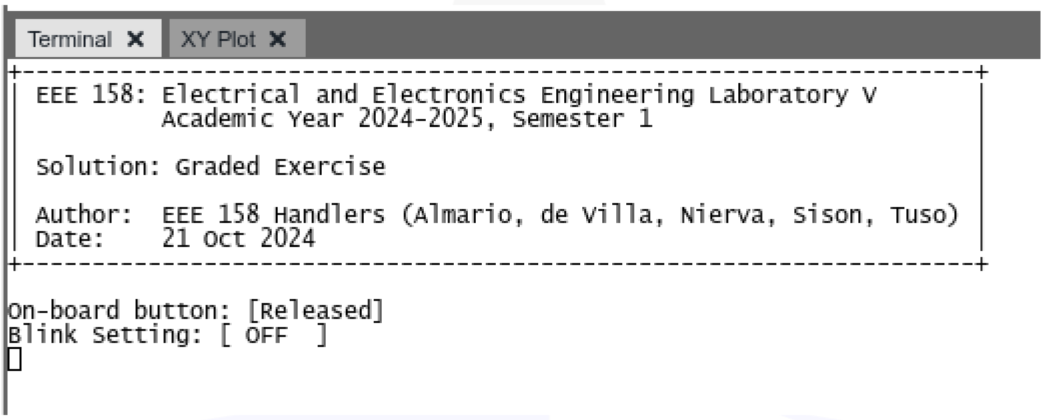
- Pressing `<LEFT>`, `<A>` or `<a>` shall cycle through the settings in descending order. On the other hand, `<RIGHT>`, `<D>` or `<d>` shall cycle in ascending order. In either case, wrap-around must not occur.
- **Blinking must NOT be done via busy-looping.** Use interleaved programming and/or TC/TCC peripherals to implement said functionality.

Submission

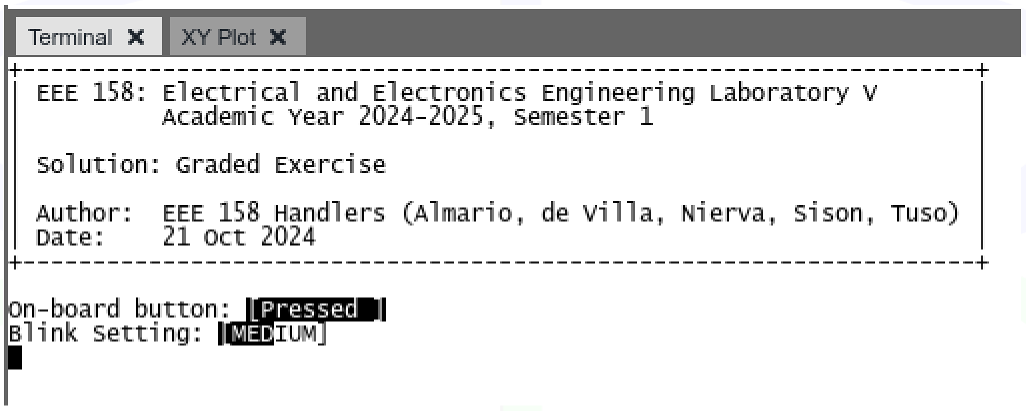
At the minimum, you are to submit the whole project as exported from the MPLAB X IDE, in the form of a ZIP file. The internal project name must have the following naming pattern: `EEE158_<Surname>-<First Letter/s of Given Name/s>_Mod05_Exer`. An example would be "`EEE158_DeVilla-A_Mod05_Exer`" (without the quotes). Errata, if any, must be included as a TXT file inside the ZIP submission.

Your instructor must be able to reproduce your work on their PIC32CM LS00 board instance.

Sample Screenshots

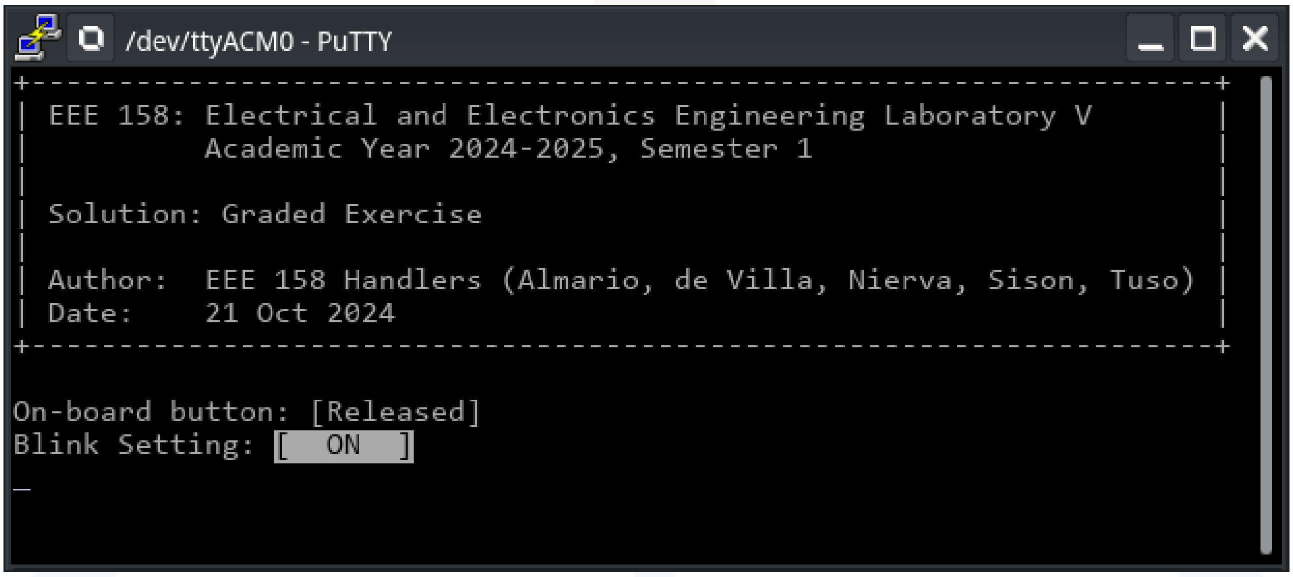


```
+----------------------------------------------------------------------+
| EEE 158: Electrical and Electronics Engineering Laboratory V         |
|          Academic Year 2024-2025, Semester 1                         |
|                                                                      |
| Solution: Graded Exercise                                            |
|                                                                      |
| Author:  EEE 158 Handlers (Almario, de Villa, Nierva, Sison, Tuso)   |
| Date:    21 Oct 2024                                                 |
+----------------------------------------------------------------------+

On-board button: [Released]
Blink Setting: [ OFF  ]
```

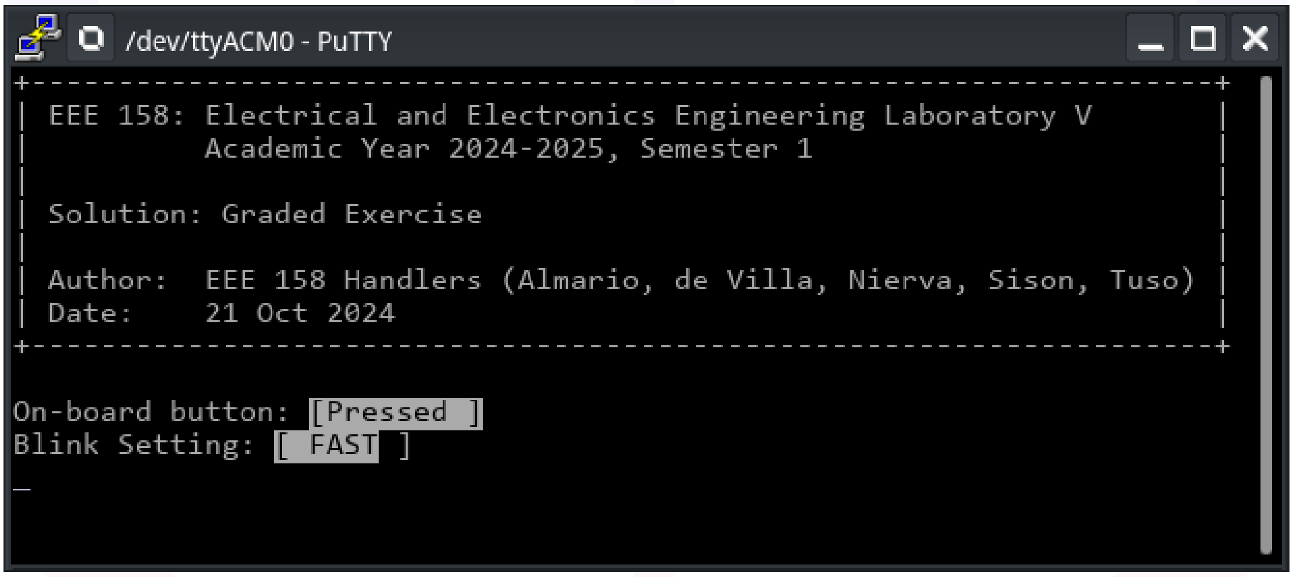


```
+----------------------------------------------------------------------+
| EEE 158: Electrical and Electronics Engineering Laboratory V         |
|          Academic Year 2024-2025, Semester 1                         |
|                                                                      |
| Solution: Graded Exercise                                            |
|                                                                      |
| Author:  EEE 158 Handlers (Almario, de Villa, Nierva, Sison, Tuso)   |
| Date:    21 Oct 2024                                                 |
+----------------------------------------------------------------------+

On-board button: [Pressed ]
Blink Setting: [MEDIUM]
```

Using MPLAB Data Visualizer via MPLAB X



```
+----------------------------------------------------------------------+
| EEE 158: Electrical and Electronics Engineering Laboratory V         |
|          Academic Year 2024-2025, Semester 1                         |
|                                                                      |
| Solution: Graded Exercise                                            |
|                                                                      |
| Author:   EEE 158 Handlers (Almario, de Villa, Nierva, Sison, Tuso)  |
| Date:     21 Oct 2024                                                |
+----------------------------------------------------------------------+

On-board button: [Released]
Blink Setting: [   ON   ]
_
```



```
+----------------------------------------------------------------------+
| EEE 158: Electrical and Electronics Engineering Laboratory V         |
|          Academic Year 2024-2025, Semester 1                         |
|                                                                      |
| Solution: Graded Exercise                                            |
|                                                                      |
| Author:   EEE 158 Handlers (Almario, de Villa, Nierva, Sison, Tuso)  |
| Date:     21 Oct 2024                                                |
+----------------------------------------------------------------------+

On-board button: [Pressed ]
Blink Setting: [  FAST  ]
_
```

Using the PuTTY terminal program