TRANSDUCERS & INSTRUMENTATION LAB (AUGUST 2026)

Lab 02
Serial Communication

AIM

To implement a serial communication protocol between a microcontroller (Teensy 4.1) and a computer.

LEARNING OUTCOMES

By the end of this lab, you should be able to:

1. Describe the principles of serial communication and the framing by which data is transmitted.
2. Implement a protocol decoder that reconstructs messages from a stream of received bytes.
3. Establish bidirectional communication between a microcontroller and a computer.
4. Demonstrate the real-time display and control of microcontroller data from the Python program on the computer.

OUTLINE OF THE SESSION

The stages are to be performed in the order listed. Each extends the system built in the preceding one, so that a fault may be traced to the stage in which it appeared rather than to the system as a whole.

1 Background

In the previous laboratory the values produced by the microcontroller were either observed directly with an instrument or sent as text and displayed on the serial monitor for a person to read. A microcontroller typically reports its measurements to other devices and accepts

instructions in return. Microcontrollers exchange data by several protocols; this section treats only the serial link between the board and the computer. Both ends of that link must agree in advance on how the information is to be represented, since the link itself conveys nothing but voltages. The remainder of this section describes that agreement, from the transmission of a single byte to the structure of a complete message, together with the software used on the computer.

Two pieces of software are written in this laboratory, one at each end of the link, and they are named throughout as follows. **The sketch** is the program that runs upon the Teensy, written in the Arduino IDE and uploaded to the board. **The Python program** is the program that runs upon the computer. Where a statement applies to both, both are named.

1.1 Serial communication

A digital line carries only one of two voltages, and can therefore convey only one bit at any instant. A number is transmitted by sending its bits one after another in time, and the receiver reassembles them. Communication of this kind is termed *serial* communication.

The receiver must know where each bit begins and ends. A line held high for 5ms could be two bits held high for 2.5ms each or a single bit held high for 5ms. Without an agreement on the duration of a bit, its not possible to communicate serially. Two methods are used to establish this. In the first, the transmitter supplies the timing, and the communication is termed *synchronous*. In the second, it does not, and the two ends agree upon the bit period in advance; the communication is then termed *asynchronous*. The two are compared in Figure 1.

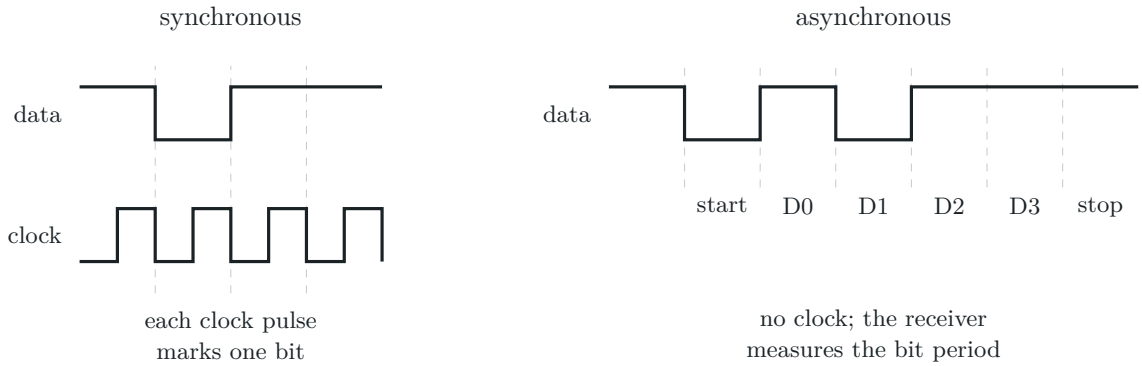

Figure 1: The same four data bits sent by each method. In synchronous communication a clock line accompanies the data and marks every bit. In asynchronous communication no clock is sent, and the byte is instead framed by a start bit and a stop bit.

Asynchronous communication

The scheme described here is the one implemented by the hardware known as the *UART*, the universal asynchronous receiver/transmitter. It is the peripheral within the microcontroller that converts a byte held in memory into the sequence of voltage levels placed upon the transmit line, and that reconstructs a byte from the levels observed upon the receive line. It is called universal because its parameters – chiefly the rate and the framing – are configurable rather than fixed. When a datasheet speaks of a *serial port*, or an Arduino sketch of `Serial1`, it is a UART that is meant. Two devices are connected by joining the transmit pin of each to the receive pin of the other and by giving them a common ground; no clock line is present.

The rate agreed upon by the two ends is called the *baud rate*, the number of signalling events per second on the line; since each event on a two-level line carries a single bit, this is here also the number of bits per second. At 9600 baud rate each bit occupies 104 µs. Nothing on the line conveys this value, and it must therefore be configured identically at both ends – the transmitter and the receiver.

Remember serial communcation is used to transmit single bytes. Because no timing is transmitted in asynchronous communcation, the beginning of each byte must be marked on the line itself. The line is held high when idle, and is pulled low for one bit period, which indicates the start of a byte. This is the *start bit*, and its falling edge marks the beginning of the transmission. The receiver waits half a bit period, which places it at the centre of the start bit, and thereafter samples once every bit period. Each sample then falls at the centre of a data bit rather than at a transition. The data bits are sent least significant first. They are followed by a *stop bit*, for which the line returns high, so that a falling edge is available to mark the byte that follows. Figure 2 shows the complete sequence.

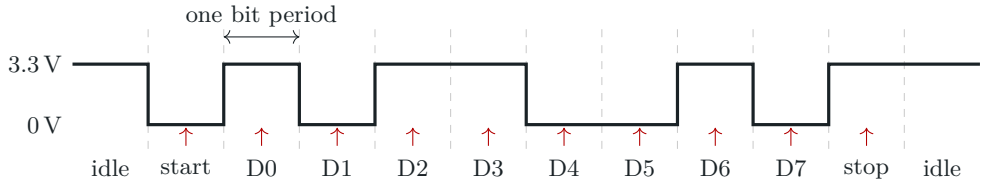

Figure 2: Transmission of one byte. The arrows mark the instants at which the receiver samples, each at the centre of a bit period.

Each frame therefore comprises a start bit, eight data bits and a stop bit. Such a frame is described by the notation 8N1, which is used to configure a serial port and gives its three adjustable parameters in order: the number of data bits, the parity setting, and the number of stop bits. Here they are eight, none, and one; two stop bits may also be used, giving 8N2. The start bit does not appear in the notation because a frame always begins with exactly one, whatever the other settings. The same notation is used in the documentation of serial devices, which commonly specify a default configuration such as 9600 8N1, and both ends of a link must be set to agree.

A byte thus occupies ten bit periods rather than eight, and a link at 9600 baud conveys 960 bytes per second. The start and stop bits are inserted and removed by the serial hardware, and are never seen by the software at either end.

Synchronous communication

In SPI (Serial Peripheral Interface) and I^2C (Inter-Integrated Circuit) the controlling device drives a separate clock line alongside the data. Each transition of that clock defines the instant at which the data line is to be read, so that the receiver is required to measure no interval of time. The rate is determined by the device generating the clock, and is derived from that device's own internal clock, commonly several megahertz for SPI and 100 kHz or 400 kHz for I^2C.

USB (Universal Serial Bus) achieves the same end without a clock line. Each packet begins with a synchronisation pattern from which the receiver determines the transmitter's rate, and the encoding guarantees frequent transitions thereafter, so that the receiver remains aligned for the length of the packet.

In none of the three cases is a start bit required before every byte, and the boundaries of a

transfer are marked instead by other means: in SPI by a chip-select line held active for its duration, in I^2C by the start and stop conditions upon the data line, and in USB by the packet itself, which is delimited by its synchronisation pattern and an end-of-packet signal, and carries a cyclic redundancy check on its contents. The overhead of two framing bits in every ten is thereby avoided, and a far greater proportion of the capacity of the link is available to carry data.

We will, however, be concerned only with the asynchronous link between the Teensy 4.1 and the computer, and the remainder of this section treats only that.

1.2 Serial communication over USB – Serial USB is not Serial

The Teensy 4.1 is joined to the computer by USB, not by an asynchronous serial link. The operating system nevertheless presents this connection to software as though it were an ordinary serial port, and the Python program is written for it in the same way. Three differences remain, and each of them matters later in this experiment.

First, **the baud rate has no effect**. The number given to `Serial.begin()` is ignored by the Teensy. The Teensy contains its USB hardware within the microcontroller itself, so no such link exists and there is nothing to set. This is true of `Serial`, the USB connection, alone. The transmit and receive pins of the board, used as `Serial1` to `Serial8`, are real UARTs, and the rate given to those does set the bit period on the line and must match the device at the other end.

Second, **the data travels in packets**. A packet may carry up to 512 bytes, and the computer, not the board, decides when packets are transferred. Bytes therefore arrive in bursts rather than at even spacing, and the moment at which a reading arrives is not a reliable measure of the moment at which it was taken. Where timing matters, the board must send a time of its own alongside each reading.

These packets are of no help in finding where one reading ends and the next begins, because the driver removes them and hands the Python program a single continuous stream of bytes, as Figure 3 shows. Nothing in that stream marks the start or the end of a reading. Any such structure must be added by the sketch and the Python program themselves, which is the subject of the rest of this experiment.

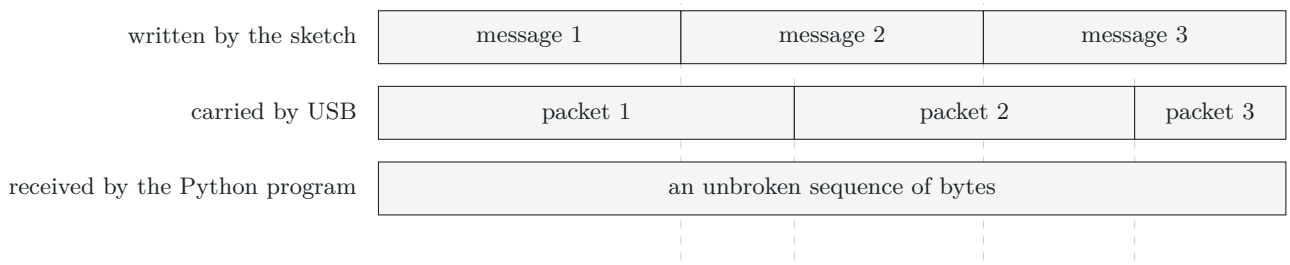

Figure 3: Boundaries at each stage of the link. The packets into which USB divides the data do not coincide with the messages written by the sketch, and neither division is presented to the Python program, which is given only a continuous sequence of bytes.

Third, **USB checks every transfer and repeats any that fails**, so corruption on the cable need not be guarded against. This is no protection, however, against a receiver that has lost track of where a message begins, which is the problem the sections that follow address.

1.3 Interfaces of the Teensy 4.1

The Teensy 4.1 provides both kinds – syncrhonous and asynchronous – as summarised in Table 1. Its three SPI interfaces, its three I^2C interfaces and the USB connection to the computer are synchronous. Its eight serial ports (UARTs) and its three CAN (Controller Area Network) interfaces are asynchronous, the former appearing as pairs of transmit and receive pins used with devices such as Bluetooth modules and GPS receivers, and the latter as differential pairs used to connect several controllers upon a common bus.

Table 1: Serial interfaces of the Teensy 4.1.

Interface	Timing	A transfer is delimited by	Typical rate	Number
SPI	separate clock line	a chip-select line held active	several MHz	3
I^2C	separate clock line	start and stop conditions on the data line	100 kHz or 400 kHz	3
USB	recovered from the data	the packet structure	480 Mbit/s	1
Asynchronous serial (UART)	agreed in advance	start and stop bits, one byte at a time	9600 to 115200 baud	8
CAN	agreed in advance	start-of-frame and end-of-frame fields, one message at a time	up to 1 Mbit/s	3

1.4 The serial monitor

Serial communication has already been employed in the previous laboratory. Every call to `Serial.print` transmitted data over this same USB connection, and the serial monitor displayed it.

That arrangement follows a protocol, albeit a simple one. `Serial.print` transmits a number as the characters that represent it, so that the value 512 is sent as the three bytes `5`, `1` and `2`. `Serial.println` appends a carriage return and a newline character, and these mark the end of one reading and the beginning of the next, as shown in Figure 4. The serial monitor interprets every byte it receives as a character and displays it, beginning a new line wherever the newline character occurs. The serial plotter applies the same rule and, in addition, interprets each line as one or more numbers separated by spaces or commas, which it then draws.

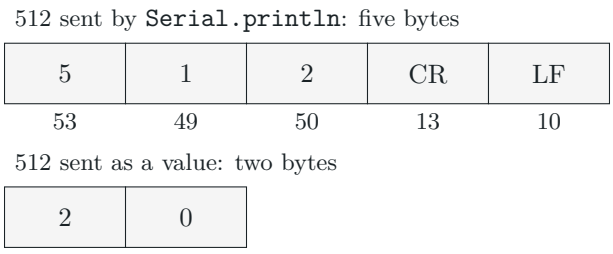

Figure 4: The value 512 transmitted as text and as a pair of bytes. The numbers beneath the upper row are the byte values actually sent, the characters above them being what the serial monitor displays.

This is adequate for confirming that a sketch is executing. It is not adequate for the present laboratory, for three reasons. Neither the monitor nor the plotter records what it receives, so no

measurement may be retained for later analysis. Neither provides any means of control beyond the typing of a line of text, so an interface containing a dial or a switch cannot be constructed from them. And both interpret every byte as a character, so that data sent in any other form is rendered as meaningless characters.

A program written for the purpose is therefore required on the computer, together with a protocol defined for that purpose rather than one imposed by an existing display. Both are the subject of the sections that follow.

1.5 Protocols

A *protocol* is the agreement between two communicating devices as to how information is to be represented and organised, and it is what allows a receiver to interpret what it is sent. The serial hardware delivers a sequence of correct bytes, but nothing more.

The UART serial communication works perfectly as it is if we are only interested in sending a single byte at a time. The difficulty arises when we wish to send more than one byte, as is the case when we wish to send readings from the analogue-to-digital converter, which may be two or more bytes long. What about a float? What about a value contained in a struct, which can have different types of variables with varying sizes? The receiver must know how many bytes to expect, and how to interpret them. It must also know where each reading begins and ends, so that it can recover the original values correctly. This cannot be done by the standard serial hardware or protocol alone, which provides no means of conveying that information. A protocol must therefore be designed to provide that information about the individual packets that contain more than one byte.

A message or packet must therefore be given a structure that allows its beginning to be identified. A byte of an agreed value may be placed at the start of every message, and the receiver may search for it. This is not sufficient by itself, since the same value may occur within the data and be mistaken for the start of a message. Using two such bytes in succession makes the coincidence less frequent but cannot prevent it, as any pattern that may be chosen may also occur in the data. The structure is therefore designed so that a false start is *detected* rather than prevented: a field giving the number of data bytes allows the receiver to determine where the message ends, and a checksum computed over the message allows it to establish what it has assembled is consistent. A message that fails this test is discarded, and the receiver resumes its search from the following byte, so that a false start costs a single message rather than corrupting everything that follows.

Several structures satisfying these requirements are in common use. Readings may be sent as text terminated by a newline character, which is simple to inspect but inefficient; a cyclic redundancy check may replace the checksum where stronger detection is required; and byte stuffing may be used to guarantee that the marker never occurs within the data. One such protocol is shown below, in which a message consists of a header of two bytes, a size byte, a variable number of data bytes, and a checksum byte.

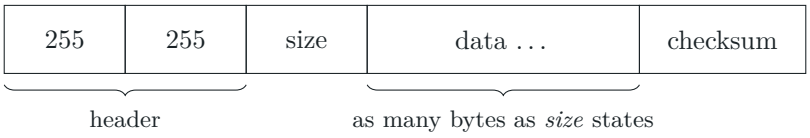

Figure 5: Structure of a message. We refer to this structure or format for trasmitting data over the UART as the JEDI format. Why? Because we are nerds.

The two header bytes each take the value 255. The size byte gives the number of data bytes that

follow, and the checksum is the sum of the size and data bytes taken modulo 256.

1.6 Software on the computer

The computer requires a program that opens the serial port, decodes the messages received, displays them, and transmits commands in return. In this laboratory that program is written in Python – the Python program named above – using three libraries: `pyserial`, which provides access to the serial port; `PyQt`, which provides the graphical interface; and `pyqtgraph`, which draws plots efficiently enough to be updated continuously. They are installed with

```
pip install pyserial pyqt6 pyqtgraph
```

The port must be identified before it can be opened. It appears as `COM3`; the exact name may be obtained from the Arduino IDE, which lists it under Tools → Port or from the device manager in ports.

A serial port may be held open by only one program at a time. The serial monitor of the Arduino IDE must therefore be closed before the Python program is run, and the Python program must be stopped before a sketch is uploaded. Failure to observe this is the most frequent cause of a port that cannot be opened.

The Arduino IDE provides two facilities for observing data sent from the board. The serial monitor displays received bytes as text, and permits text to be typed and sent in return. The serial plotter draws a graph of numbers printed as text and separated by spaces or commas. Both are useful for checking that a sketch is running, and both are limited in ways that matter here. Neither interprets data that is not text, so neither can display the messages used in this laboratory. Neither offers any control beyond typing a line, so an interface with a dial or a switch cannot be built with them. Neither records what it receives, so no measurement can be kept for later analysis. A separate program is therefore required, and Python is used for that purpose in this laboratory.

2 Apparatus and components

APPARATUS AND COMPONENTS

Qty	Item	Specification
1	Teensy 4.1 development board	with micro-USB cable
1	Solderless breadboard	—
1	LED	–
1	Resistor	–
1	Potentiometer	10 kΩ, linear taper
1	Push-button switch	tactile, SPST, breadboard mount
1	Digital multimeter	—

3 Experiment 01: Understanding serial communication

3.1 Objective

To observe the bit duration, framing and baud rate of an asynchronous serial transmission on an oscilloscope.

3.2 Circuit

Connect the oscilloscope probe to digital pin 1, which is the transmit line of `Serial1`, and its ground lead to `GND`.

3.3 Observations

3.3.1 Bit duration and framing

Write a sketch that calls `Serial1.begin(9600)` and transmits the character `U` repeatedly, with a short delay between transmissions. This character has the value 85.

Set the oscilloscope to trigger on a falling edge and adjust the timebase until a single transmission fills the screen. Record the waveform.

Measure the duration of one bit and the duration of the complete frame. Repeat for each baud rate in Table 2, altering the value passed to `Serial1.begin` and adjusting the timebase accordingly.

Table 2: Bit period and frame duration against baud rate.

Baud rate	Expected bit period (µs)	Measured bit period (µs)	Measured frame duration (µs)
9600			
19200			
57600			
115200			

Questions to be answered

Q3.1 State the relation between the measured bit duration and the baud rate.

3.3.2 Decoding a character

Alter the sketch to transmit the character `A` at 9600 baud, and capture a single transmission as before.

From the recorded waveform, identify the start bit, the eight data bits and the stop bit. Read the data bits in the order in which they were transmitted and enter them in Table 3. Assemble them into a byte and compare the result with the value of the character according to the ASCII table.

Table 3: Decoding of a transmitted character.

Character	D0	D1	D2	D3	D4	D5	D6	D7	Value decoded
`A`									

Repeat for a character of your own choosing, and confirm that the value decoded from the waveform agrees with its ASCII value in each case.

4 Experiment 02: Reading a potentiometer

4.1 Objective

To sample the voltage produced by a potentiometer.

4.2 Circuit

A potentiometer is a resistive track with a contact, the *wiper*, that may be moved along it. Connecting the two ends of the track to 3.3 V and `GND` forms a voltage divider, and the voltage at the wiper varies continuously between those two values according to its setting.

Connect one end of the 10 kΩ potentiometer to `3V3`, the other end to `GND`, and the wiper to analog pin `A0`.

4.3 Observations

Write a sketch that reads `A0` and prints the value to the serial monitor at intervals of 100 ms.

Turn the potentiometer to each of eight positions distributed randomly between its two extremes. At each, record the value reported by the converter and the voltage measured at the wiper with the multimeter, and convert the reported value to a voltage as in the previous laboratory.

Table 4: ADC values against measured wiper voltage.

No.	Measured wiper voltage (V)	ADC value	Calculated voltage (V)
1			
2			
3			
4			
5			
6			
7			
8			

Plot the reported value against the measured voltage.

Questions to be answered

Q4.1 What is the slope of the line obtained, and how does it compare with the slope expected from the range of the converter and the reference voltage?

Q4.2 Do the two extremes of the potentiometer correspond to 0 V and 3.3 V? If they do not, account for the difference.

5 Experiment 03: Controlling LED brightness from the potentiometer

5.1 Objective

To vary the brightness of an LED continuously according to the voltage sampled from the potentiometer.

5.2 Circuit

The potentiometer of the previous stage is retained. Connect an LED in series with a current limiting resistor between digital pin 2 and GND, as in the previous laboratory.

5.3 Observations

Write a sketch that samples A0 continuously and sets the brightness of the LED from the value obtained, using `analogWrite` on pin 2.

The two ranges do not match. The converter reports values from 0 to 1023, ten bits, whereas `analogWrite` accepts values from 0 to 255, eight bits, so the sampled value must be divided by four before it is used. (Neither range is a limit of the hardware – the Teensy 4.1 permits up to twelve bits in either direction, through `analogReadResolution()` and `analogWriteResolution()` – but use the default resolutions here, so that the scaling is required.)

Set the potentiometer to four positions spread across its range. At each, record the value read from the converter and the value passed to `analogWrite`, and calculate the duty cycle of the resulting pulse-width modulated output, which is the commanded value divided by 255.

Table 5: Scaling of the ADC value to the brightness command.

ADC value	PWM value	Duty cycle (%)

Questions to be answered

Q5.1 How many distinct values can the converter report?

Q5.2 How many distinct brightness levels can be commanded?

Q5.3 The first number is the larger, so several readings must produce the same brightness. How many readings share each brightness level?

Q5.4 What change in the wiper voltage is needed to alter the commanded value by one step?

6 Experiment 04: Transmitting readings to the computer

6.1 Objective

To transmit the sampled values to the computer.

6.2 Circuit

No change is made to the circuit.

6.3 Observations

6.3.1 Transmitting data

Two functions send data over the serial connection, and the distinction between them is essential. `Serial.print(512)` transmits the characters `5`, `1` and `2`, that is three bytes of text. `Serial.write(2)` transmits a single byte whose value is 2. The protocol of this laboratory is defined in terms of byte values, and therefore requires `Serial.write`.

A sampled value may exceed 255 and cannot be carried in one byte. It is therefore sent as two: the upper byte, obtained as `value » 8`, followed by the lower byte, obtained as `value & 0xFF`. The receiver reconstructs the value as `(upper « 8) | lower`. The order in which the two bytes are sent is part of the agreement and must be the same at both ends.

Modify the sketch so that each sampled value is transmitted as a message of the form given in Section 1.5, with two data bytes. Open the serial monitor and observe what is displayed.

Questions to be answered

Q6.1 Attach a screenshot of what is displayed on the serial monitor. Can you explain what is shown?

6.3.2 Decoding

Write the Python program. It is to open the serial port, read whatever bytes have arrived, and reconstruct the messages from them.

Bytes do not arrive in complete messages, and a read may well end in the middle of one. The decoder must therefore remember, between one read and the next, how far through a message it has progressed. It proceeds as follows.

1. Search the bytes received for the two header bytes in succession.
2. Take the byte that follows as the size, and collect that many data bytes.
3. Take the byte after those as the checksum, and compare it with the sum of the size and data bytes, modulo 256.

4. If the two agree, accept the message. If they do not, discard what has been assembled and resume the search from the byte following the false header, since what was taken for a header was evidently something else.

Once a message has been accepted, the Python program is to reconstruct the reading from its two data bytes as `(upper « 8) | lower`, convert that reading to the corresponding potentiometer voltage as in Section 4, and print both on the screen.

Set the potentiometer to three positions in turn. At each, record the value printed by the Python program, the voltage it reports, and the voltage measured at the wiper with the multimeter.

Table 6: Readings recovered by the Python program against the measured wiper voltage.

No.	Value printed	Voltage reported (V)	Measured wiper voltage (V)
1			
2			
3			

Questions to be answered

Q6.2 Do the values recovered by the Python program agree with the measured wiper voltages, as they did in Section 4?

Q6.3 Why must the decoder remember its progress between one read and the next, rather than treating the bytes returned by each read as one message?

7 Experiment 05: Displaying received data in a graphical interface

7.1 Objective

To display the received values as a continuously updated plot, together with the brightness of LED shown as a number.

7.2 Circuit

No change is made to the circuit.

7.3 Observations

7.3.1 Polling the serial port

A graphical program is organised around an event loop, which redraws the window and responds to the mouse and keyboard. Any operation that waits within that loop prevents it from doing either, and the window ceases to respond. Reading from the serial port must therefore never wait for data to arrive.

The reading is instead performed by a timer.[1] A `QTimer` is configured to call a function at a fixed

[1] `https://doc.qt.io/qt-6/qtimer.html`

interval; that function reads whatever bytes are already waiting, passes them to the decoder written in the previous stage, and returns immediately. This is the same arrangement used in the previous laboratory, where the state of a switch was examined on each pass through the loop rather than waited upon; only the peripheral has changed.

Bytes that arrive between one call and the next are not lost. They accumulate in a buffer maintained by the operating system, and the number waiting may be obtained from `in_waiting`.[2] All of them should be read on each call and passed to the decoder, which will assemble whatever complete messages they contain and retain the remainder. Reading a fixed number of bytes instead, on the assumption that it constitutes one message, causes the Python program to fall progressively further behind. The arrangement is shown in Figure 6.

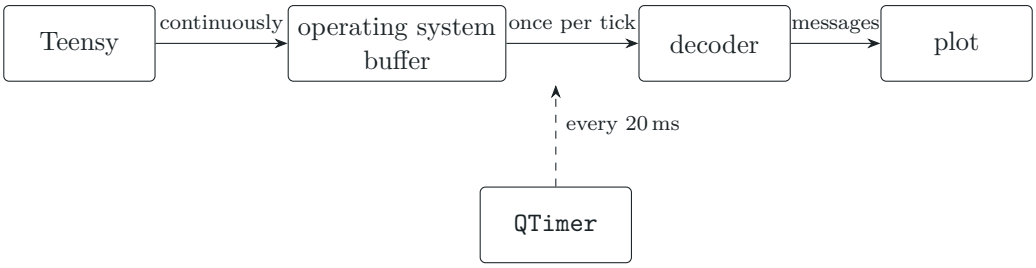

Figure 6: Data pipeline. Bytes arrive from the sketch continuously and accumulate in the buffer, from which every byte waiting is removed at each tick of the timer and passed to the decoder. The rate of arrival and the rate of reading are therefore independent, and the buffer holds the difference.

To confirm the necessity of this arrangement, replace the timer with a loop that waits for the next message and returns only when one has arrived. Attempt to move the window while the Python program runs, and record the result. Restore the timer afterwards.

Questions to be answered

Q7.1 What happens to the window while the Python program waits for the next message, and why?

8 Experiment 06: Sending commands from the computer to the board

The direction of travel is now reversed. In this stage the board only receives: the LED is set solely by the dial in the graphical interface, and the sketch sends nothing at all. The sampling of `A0` is removed from the sketch altogether, so that the plot built in Section 7 remains empty throughout this stage; both directions are used together in Section 9.

8.1 Objective

To set the brightness of the LED from a dial in the graphical interface.

8.2 Circuit

No change is made to the circuit. The potentiometer remains wired to `A0`, but is not used in this stage and is read again in Section 9.

[2] `https://pyserial.readthedocs.io`

8.3 Observations

8.3.1 Transmitting commands

The JEDI format of Section 1.5 serves in either direction, and the same two header bytes, size byte and checksum are used for a command as for a reading. Only the data bytes differ.

The brightness is commanded here as a twelve-bit number, in the range 0 to 4095. Such a value does not fit in one byte, and is therefore carried in two, exactly as a reading was in Section 6: the upper byte, obtained as `value » 8`, followed by the lower byte, obtained as `value & 0xFF`. A command thus has two data bytes, and the upper of them never exceeds 15.

Twelve bits are also more than `analogWrite` accepts by default. The sketch must therefore call `analogWriteResolution(12)`, as Section 5 anticipated, after which `analogWrite` accepts values from 0 to 4095 and no scaling of the commanded value is required.

Add a `QDial` to the interface, with its range set to 0 to 4095. Whenever its value changes, the Python program is to construct such a command and write it to the serial port.

8.3.2 Receiving commands on the board

The sketch must read these messages without waiting for them. Commands arrive only when the dial is moved, and a sketch that waited for one would be unable to do anything else in the meantime – the same difficulty met on the computer in Section 7, and avoided there by a timer. `Serial.available()` reports the number of bytes already received; if it is zero the loop proceeds, and if it is not, those bytes are passed to a decoder identical in structure to the one written in Python.

Once a complete message has been decoded and its checksum found to agree, the sketch is to reconstruct the twelve-bit value as `(upper « 8) | lower` and set the brightness to it. A message whose checksum does not agree is discarded, as on the computer.

Move the dial slowly through its range and confirm that the LED responds.

Questions to be answered

Q8.1 Why is the commanded value carried in two bytes rather than one, and what is the largest value the upper byte can take?

Q8.2 What would the LED do were `analogWriteResolution(12)` omitted from the sketch, the dial still being moved through its whole range?

9 Experiment 07: Bidirectional communication

9.1 Objective

To transmit readings and receive commands simultaneously, to place the control logic on the computer rather than on the board, and to measure the latency this introduces.

9.2 Circuit

Add a push-button between digital pin `3` and `GND`, configured with the internal pull-up as in the previous laboratory. Add a second LED of a different colour, in series with a current-limiting resistor, between digital pin `4` and `GND`.

9.3 Observations

9.3.1 LED control

The original LED lightining logic remains the same as the one before. The potentiometer controls the original LED's brightness, and this potentiometer value is transmitted to the PC and dispalyed on the Python program's plotter.

The green LED's ON/OFF state is controlled by the push button. The LED is off when the button is not pressed. When the button is not pressed the dial on the graphical interface is disabled. When the button is pressed, the dial is enabled and its position determines the brightness of the green LED. The potentiometer does not control the green LED's brightness.

Table 7: Data bytes of the message sent from the board, and the stage at which each was introduced.

Byte	Contents	Introduced in
1–2	the reading, upper byte first	Section 6
3	the state of the button	

9.3.2 Simultaneous operation

Confirm that readings continue to be plotted while commands are being transmitted, and that neither direction impedes the other. Under board control, move the dial and record the state of the LEDs and the value displayed by the interface; repeat under computer control, turning the potentiometer instead. Then press the button while the dial is in continuous motion.

10 Deliverables

Submit the following:

1. The completed tables: Tables 2 and 3 from Section 3, Table 4 from Section 4, Table 5 from Section 5, and Table 6 from Section 6.
2. The plot of the reported value against the measured wiper voltage, from Section 4.
3. The screenshot of the serial monitor called for in Q6.1.
4. The sketch and the Python program in their final form, with comments. Where a stage altered either of them in a way worth showing, include that version also, labelled with the section it belongs to.
5. Written answers to every question posed in the Observations of each stage, each identified by its number. The number gives the section in which the question appears: **Q4.1**, for instance, is the first question of Section 4.

References

[1] PJRC, *Teensy 4.1 Development Board*. https://www.pjrc.com/store/teensy41.html

[2] *pySerial Documentation*. https://pyserial.readthedocs.io

[3] *QTimer Class*. https://doc.qt.io/qt-6/qtimer.html

[4] *PyQtGraph Documentation*. https://pyqtgraph.readthedocs.io

Marking sheet

Question	2 marks
Experiment 01 — Understanding serial communication	
Q3.1 State the relation between the measured bit duration and the baud rate.	
Experiment 02 — Reading a potentiometer	
Q4.1 What is the slope of the line obtained, and how does it compare with the slope expected from the range of the converter and the reference voltage?	
Q4.2 Do the two extremes of the potentiometer correspond to 0 V and 3.3 V? If they do not, account for the difference.	
Experiment 03 — Controlling LED brightness from the potentiometer	
Q5.1 How many distinct values can the converter report?	
Q5.2 How many distinct brightness levels can be commanded?	
Q5.3 The first number is the larger, so several readings must produce the same brightness. How many readings share each brightness level?	
Q5.4 What change in the wiper voltage is needed to alter the commanded value by one step?	
Experiment 04 — Transmitting readings to the computer	
Q6.1 Attach a screenshot of what is displayed on the serial monitor. Can you explain what is shown?	
Experiment 05 — Displaying received data in a graphical interface	
Q7.1 Do the values recovered by the Python program agree with the measured wiper voltages, as they did in Section 4?	
Q7.2 Why must the decoder remember its progress between one read and the next, rather than treating the bytes returned by each read as one message?	
Experiment 06 — Sending commands from the computer to the board	
Q8.1 What happens to the window while the Python program waits for the next message, and why?	
Experiment 07 — Bidirectional communication	
Q9.1 Why is the commanded value carried in two bytes rather than one, and what is the largest value the upper byte can take?	
Q9.2 What would the LED do were `analogWriteResolution(12)` omitted from the sketch, the dial still being moved through its whole range?	