Lab 4: **Serial Communication**

SED 1115 – Embedded Programming and Scripting

Faculty of Engineering – University of Ottawa

Lab Objectives:

The purpose of this lab is to introduce you to serial communication and how to use it to communicate with other devices. You will be using the UART (Universal Asynchronous Receiver-Transmitter) to communicate with another Raspberry Pi Pico. At the end of this lab, you will implement a simple traffic light system using two Raspberry Pi Picos.

Lab Instructions:

Parts and Equipment

You will need: a Raspberry Pi Pico, a laptop or other computer with USB interfaces and internet access, a USB micro-B cable that works with that computer and also an SED1115 custom designed “expansion” board.

You will also need a second Raspberry Pi Pico from your teammate, but don't worry, the code will be symmetrical so you can use your code on their board and vice versa.

Warnings

Note that a Pico is a 3.3V device that should not be connected to 5V devices or to other potential 5V power sources, other than via the onboard USB port (which is a supported 5V standard by the Pico). Connecting the Pico directly to other 5V devices can damage it. Please do not use the Pico on a conductive surface (e.g. metal table) to avoid shorting the exposed pins on the bottom of the device. Even when mounted on the “accessory” board, which has included mechanical standoff posts, shorting pins might still be possible, if the surface is very uneven.

It is recommended that you place the Pico (and/or the expansion board) on a flat, non-conductive, anti-static surface, such as a wooden table or a notebook. Electrostatic discharge (ESD) can damage the Pico (or most other electronic devices). To avoid such damage, when working on the Pico, ground yourself using an approved grounded ESD wrist strap (properly connected to ground) or touch a grounded metal part of a grounded object (e.g. your computer case), constantly, while working with any ESD-sensitive semiconductor devices. Try to touch only the metal standoffs of the accessory board.

Expansion Board

In this lab, you will be using the UART socket (H4) on the expansion board for serial communication.

Pin	GPIO	Function
3V3	-	3.3V Power Pin
RTS	GP7	UART request to send (w/ level shifter)
GND	-	Ground
TX	GP8	UART channel 1 TX (w/ level shifter)
RX	GP9	UART channel 1 RX (w/ level shifter)
GND	-	Ground
CTS	GP6	UART clear to send (w/ level shifter)
VUART		Power for UART device (3.3V needed)

When connecting to another UART device, connect the GND pin to the ground pin on the other device, RX to TX on the other device, and TX to RX on the other device and connect the 3.3V signals together and the ground signals together. This can all be accomplished by connecting with an 8-pin (**BUT FLIPPED AROUND**) connector, where pin 1 on one accessory board is connected to pin 8 of the other, etc.

If you want to use flow control, connect CTS to RTS on the other device and RTS to CTS on the other device.

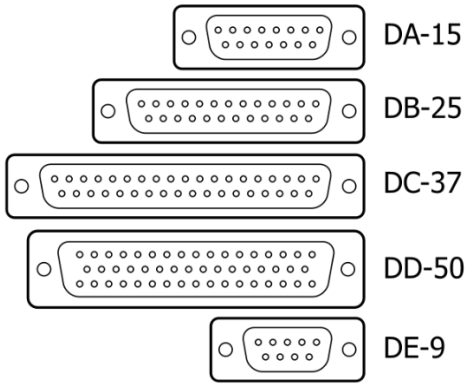
VUART is the voltage for the UART channel. When connecting to another UART device, it is important to make sure that the voltages are compatible. As already warned, the Raspberry Pi Pico uses 3.3V logic, so it is important to make sure that the other device also uses 3.3V logic. If the other device uses 5V logic, for example, you should connect 5V from the other device to VUART on the expansion board so the onboard level shifters (see appendix, for more details) can convert the voltage between the two devices. It is also important to note that you can only use voltage higher than or equal to 3.3V on VUART, not lower. If you connect a 1.8V device to VUART, for example, you may damage the other device.

Serial Communication

Serial communication is a method of sending data between two devices. It is called serial communication because the data is sent one bit at a time, in a series of electrical pulses on a single wire. In parallel communication, there are multiple connections, allowing multiple bits to be sent concurrently, rather than one after another.

The advantage of serial communication is that it only requires one wire to send data. This makes it easier to connect devices together and helps keep cables and connectors small. If you have seen the old parallel printer cables, you will know that they have a lot of wires and can

have large connectors. The following image shows a type of common parallel connectors called D-subminiatures. You might find the DB-25 connector on printers and the DE-9 connector on VGA devices like monitors and projectors.



Source: Wikimedia Commons

Some common parallel electrical communication protocols are PCI, PCI Express, VGA, Twisted-Pair Ethernet, and HDMI. Some common serial electrical communication protocols are UART, SATA, SPI, I2C, and USB. Note that most optical communications is done serially, although parallel technologies have also been defined.

In theory, parallel data rates should be higher than serial data rates. In practice, serial data rates can still be very high (e.g. measured in Gb/s). This is because the lines can be controlled electrically, more easily, without increasing the size or cost of the required connectors and cables. With parallel cables, it is necessary to control the skew or variation in length between the different parallel conductors.

UART

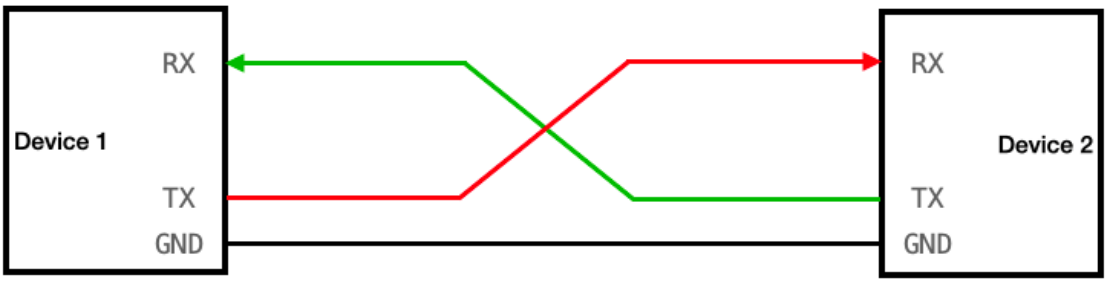
In this lab, you will be using the UART (Universal Asynchronous Receiver-Transmitter) protocol to communicate between two Raspberry Pi Picos. This is one of the earliest and simplest serial communication protocols. It is used to communicate between devices that are close together (e.g. like the two Raspberry Pi Picos in this lab!).

As the name implies, a UART uses asynchronous transmission, which means that the data is sent without a separate clock signal, unlike SPI and I2C. Instead, the clock is extracted at the receiver, based on the transitions of start/stop bits, sent with each transmission. In this sense, an asynchronous communications channel is said to be “self-timed”, making it easier to implement than SPI and I2C, in terms of wiring, but limiting the overall speed.

Since there is no clock signal, the two devices need to agree on the baud rate, where baud rate measures the number of symbols that are sent per second. For a UART, these symbols are

voltage levels that represent the bits. For example, if the baud rate is 9600, then 9600 symbols can be sent per second. For a typical UART, one symbol represents one bit, so the baud rate is the same as the bit rate. For example, if the baud rate is 9600, then 9600 bits are sent per second, therefore the transmission rate is 9600 bits per second or 1200 bytes (i.e. there are 8 bits in a byte) per second. Some common baud rates are 9600, 19200, 38400, 57600, and 115200.

A UART uses two wires to send data, one for sending data and one for receiving data. The sending wire is called TX (**T**ransmit) and the receiving wire is called RX (**R**eceive). The TX wire on one device is connected to the RX wire on the other device and vice versa, to allow bidirectional communications. This is shown in the following image.



Source: https://vanhunteradams.com/

Note that you also need to connect the ground pins of the two communicating devices together. This is because the voltage on the TX and RX wires are relative to the ground voltage. If the ground voltages are different, then the voltage on the TX and RX wires will be different, which will cause problems, when interpreting the signals.

A UART is a full-duplex protocol, which means that data can be sent in both directions at the same time. Half-duplex protocols, only allow data to be sent in one direction at a time. Although slower, a half duplex connection can be implemented with only a single wire, whose data transmission direction is reversed, periodically.

When implementing UART, flow control can be used to throttle data transmission. This is useful when one device might be sending data faster than the other device can receive it. To implement flow control, a mechanism is needed for the receiver to tell the transmitter to start or stop transmission.

A common way to do this is to use two additional wires. The wire for the transmitter to indicate that data is available is called “Request to Send” (RTS). On the receiver side, the wire that indicates that the receiver is ready to receive this data is “Clear to Send” (CTS). The CTS input on one device is connected to the RTS output of the other device. This is done for each communication direction (e.g. like the Rx and Tx wires).

See the appendix for the pinout of the UART socket on the expansion board.

UART Frame

A UART sends data in frames. You can think of a frame as a packet, or time-limited stream, of data. The frame consists of a start bit, the data bits, an optional parity bit, and stop bits and you need to make sure that the two devices agree (i.e. are set to be the same types of values) on the number of these data bits, parity, and stop bits. Otherwise, the receiving device will not be able to decode the data correctly and you will see weird characters displayed.

Start Bit

Recall that there is no synchronized clock signal in UART, so the receiving device needs to know when the data bits start. This is the purpose of the start bit. The start bit is always 0, so the receiving device knows when the data bits start.

Data Bits

The data bits are the actual data that is being sent. The number of data bits can be 5, 6, 7, or 8 (a byte is the most common data format).

Parity Bit

The parity bit is an optional bit, that is used for error checking. It works by making the number of 1s in the frame either even or odd. The exact choice for this depends on how parity has been configured (at both the transmitter and receiver). It needs to be configured to be the same at both ends of the connection.

If the parity is set for even, if there is a even number of 1s in the frame, then the parity bit is 0. If there is an odd number of 1s in the frame, then the parity bit is 1. If the parity is set for odd, if there is a even number of 1s in the frame, then the parity bit is 1. If there is an odd number of 1s in the frame, then the parity bit is 0.

If the parity is set for even, then the total number of 1s for the data bits and parity bit should be even. If the parity is set to be odd, then the total number of 1s in the data bits and parity bit should be odd. If there is an error in the frame resulting in a single bit flip, the receiving device can detect it by checking the parity bit. If the parity bit is incorrect, then the receiving device knows that there is an error in the frame.

However, if there are two or more even number of bit flips in the frame, then the parity bit will still be correct, so the receiving device will not know that there is an error. Therefore, parity should not be relied on for error checking, it is only useful for detecting single or odd numbers of bit errors. For 8 bits, what parity bit setting would give you parity errors when data was all 1’s or all 0’s (e.g. as might happen when a serial communications wire was not connected!)?

Odd paritiy would produce parity errors when the data is all 1's and 0's.

What happens if only the parity bit is corrupted and the data is uncorrupted? Will you still get a parity error?

We would still get a parity error. A single-bit error in either the data bits or the parity bit would trigger.

Stop Bit

The stop bits are always 1s, so the receiving device knows when the frame ends. The stop bits can be one or two bits long.

Example

The following example shows a UART frame with 8 data bits, 1 parity bit (even), and 2 stop bits.

```
001100001111
```

This frame can be broken down into the following parts:

Start bit	Data bits	Parity bit	Stop bits
0	01100001	1	**11**

Note that there are three 1s in the data bits, so the parity bit must be calculated and sent as a "1" to make the parity even. The data bits can be converted, using the ASCII character set, to get a lowercase "a" value.

What is the hexadecimal (base 16) value of binary 01100001_2. **Hint**: convert the value one nibble at a time (i.e. convert the 0110_2 nibble then convert the 0001_2 nibble)?

61

What should the parity bit be if the data bits are 01010011 (or 53_{16}) and the parity is set to be odd?

1

MicroPython UART

The machine library in MicroPython provides a class called UART for using the UART protocol. See https://docs.micropython.org/en/latest/library/machine.UART.html for the detailed documentation.

Raspberry Pi Pico has a built-in RX and TX buffer of 32 bytes for each UART channel. When the Pico receives data, it will store that data in the RX buffer. And you can use the `any()` method to check if there is any data in the RX buffer, and the `read()` method to read the data from the RX buffer. See Receiving Data section for more details. The data buffers make sending and receiving data much more convenient and efficient for the software.

Initialization

https://docs.micropython.org/en/latest/library/machine.UART.html#machine.UART.read

To initialize the UART, you need to specify the id, the baud rate, the number of data bits, the parity, the number of stop bits, the TX pin, and the RX pin.

The UART id is the id of the UART channel. The Raspberry Pi Pico has two UART channels, so the id can be 0 or 1. The UART socket on the expansion board is connected to UART channel 1, so the id should be 1. The TX and RX pins on the expansion board are connected GPIO 8 and 9, respectively, so you should use GPIO 8 for TX and GPIO 9 for RX.

This is an example of initializing a UART object with 8 data bits, no parity, and 1 stop bit:

```
uart = UART(1, baudrate=9600, tx=Pin(8), rx=Pin(9))

uart.init(bits=8, parity=None, stop=1)
```

You can set parity to `None` for no parity, `1` for odd parity, or `0` for even parity.

Sending Data

https://docs.micropython.org/en/latest/library/machine.UART.html#machine.UART.write

To send data, you can use the "`write()`" method. The `write()` method takes a buffer of bytes as an argument.

The following example shows sending 5 bytes of data containing the ASCII characters `H`, `e`, `l`, `l`, and `o`:

```
uart.write('Hello')
```

What actual binary bit values would be sent on the serial link, for the previous code? **Hint**: Write out each frame, one after the other.

h: 00010010, e: 10100110, l: 00110110, o: 11110110

Receiving Data

https://docs.micropython.org/en/latest/library/machine.UART.html#machine.UART.read

To receive data, you can use the `read()` method. The `read()` method will read the specified number of bytes from the UART. If the number of bytes is not specified, then it will read all of the available bytes. It will return a bytes object that contains the data that was read. If no data was read, it will return `None`. **Hint**: Your code should check for this return code (e.g. "*handle all cases*").

The following example shows reading 5 bytes of data and storing it in a variable called `data`:

```
data = uart.read(5)
```

The following example shows reading all available bytes of data:

```
data = uart.read()
```

Why would you want to use one method or the other and what would you also need to do (in either case) in your code, when you know exactly how many bytes should be transmitted by a transmitter that is operating properly in your design (**hint**: "*handle all cases*")?

When I know excactly the number of bytes I have, I'd use uart.read(n). When I don't know the amount of bytes I have, I'd use uart.read(). I would always handle all cases safely so my program doesn't crash or miss data (check what was actually returned).

Any()

https://docs.micropython.org/en/latest/library/machine.UART.html#machine.UART.any

To check if there is any data available to read, you can use the `any()` method. The `any()` method will return the number of frames that are available to read. To check if there is any data available to read, you can simply use the following:

```
if uart.any():

    # There is data available to read
```

This will convert the number of frames to a boolean value. If the number of frames is `0`, then it will be converted to `False`, otherwise it will be converted to `True`.

Lab Assignment:

Send and receive messages between two Pico devices over a serial wire connection, with transmitted message content that is specified by the user connected to a particular Pico device and also displaying the messages that are being received from the other device. Essentially, this is a chat application. You need to write a program that, on a continuous basis, switches between these two activities (i.e. looks for incoming messages and requests/accepts local user input to create any outgoing messages). If there are no messages from a user, then you should send a periodic "keep alive" message with the local time only to the partner Pico.

Lab Submission:

Please submit the filled-in laboratory manual PDF file into BrightSpace, with a link to your GitHub repository, and another copy of your source file into BrightSpace. The GitHub repository should contain the same source file and should be set to "public", so that the TA can access it. You can provide a functional link to this GitHub repository in the space that is provided below:

https://github.com/Dxher/Lab4

The code should be well-formatted and well-commented. The code should be easy to read and understand.

Due Date:

See BrightSpace.

Appendix – Pin Reference and Level Shifter

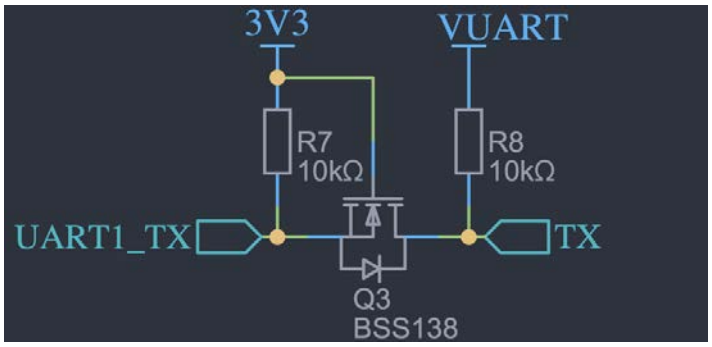
Pin	GPIO	Function
Q7 FB	GP4	Shift Register Most significant (earliest) bit
SHCP	GP18	Shift Register clock pulse
STCP	GP19	Shift Register Storage Register clock pulse
DS	GP20	Shift Register Serial Data input
OE	GP21	Output Enable for Shift Register Storage Register
LEDR1	-	Shift Register bit 0 (Red LED 1)
LEDY1	-	Shift Register bit 1 (Yellow LED 1)
LEDG1	-	Shift Register bit 2 (Green LED 1)
LEDW1	-	Shift Register bit 3 (White LED 1)
LEDR2	-	Shift Register bit 4 (Red LED 2)
LEDY2	-	Shift Register bit 5 (Yellow LED 2)
LEDG2	-	Shift Register bit 6 (Green LED 2)
LEDW2	-	Shift Register bit 7 (White LED 2)
LEDR3	-	Shift Register bit 8 (Red LED 3)
LEDY3	-	Shift Register bit 90 (Yellow LED 3)
LEDG3	-	Shift Register bit 10 (Green LED 3)
LEDW3	-	Shift Register bit 11 (White LED 3)
LEDR4	-	Shift Register bit 12 (Red LED 4)
LEDY4	-	Shift Register bit 13 (Yellow LED 4)
LEDG4	-	Shift Register bit 14 (Green LED 4)
LEDW4	-	Shift Register bit 15 (White LED 4)
SW1	GP10	SW 1 (needs a weak pull-down)
SW2	GP11	SW 2 (needs a weak pull-down)
SW3	GP12	SW 3 (needs a weak pull-down)
SW4	GP13	SW 4 (needs a weak pull-down)
SW5	GP22	SW 5
R1	A0 / GP26	Left potentiometers
R2	A1 / GP27	Right potentiometers
J4	A2 / GP28	External analog input socket
SDA	GP14	I2C channel 1 serial data
SCL	GP15	I2C channel 1 serial clock
CTS	GP6	UART clear to send (w/ level shifter)
RTS	GP7	UART request to send (w/ level shifter)
TX	GP8	UART channel 1 TX (w/ level shifter)
RX	GP9	UART channel 1 RX (w/ level shifter)
J1	GP0	PWM channel 0A
J2	GP1	PWM channel 0B
J3	GP2	PWM channel 1A
SQW	GP3	Square wave output from the DS3231 module

Level Shifters

The level shifters are circuits that convert the voltage between logic levels. The Raspberry Pi Pico uses 3.3V logic, but many other devices use 5V logic. The level shifters on the expansion board convert the voltage between 3.3V and 5V.

The level shifters on the expansion board are bidirectional, so they can convert the voltage in both directions. This means that you can send data from the Pico to a 5V device and receive data from a 5V device on the Pico.

The level shifters on the expansion board are based on the BSS138 (Datasheet: https://www.onsemi.com/pdf/datasheet/bss138-d.pdf) N-channel Metal Oxide Semiconductor Field Effect Transistor (MOSFET) chip. The circuit is shown below.



The circuit is based on the SparkFun Logic Level Converter.

We will not go into the details of how the circuit works, you will learn this in your electrical engineering courses. The important thing to know is that the circuit converts the voltage between 3.3V and 5V bidirectionally, so you can use it to communicate with other devices that use 5V logic.

Shift Register

A shift register (see the data sheet for the device; 74HC595) is a memory device with a serial data input. Data is “clocked into” the device using a separate control signal. To enter data, the serial data input should be set to a desired bit value and the clock signal driven from high to low. This process can be repeated until all 8 bits of the device have data stored in them.

On the accessory board, there are two such devices, themselves connected in series, so that a total of 16 bits can be stored. Each of these storage element outputs are connected to the traffic intersection LEDs on the board. Therefore, from software, it is possible to set the 16 bits of the shift register to any value to control the 16 LEDs on the board.