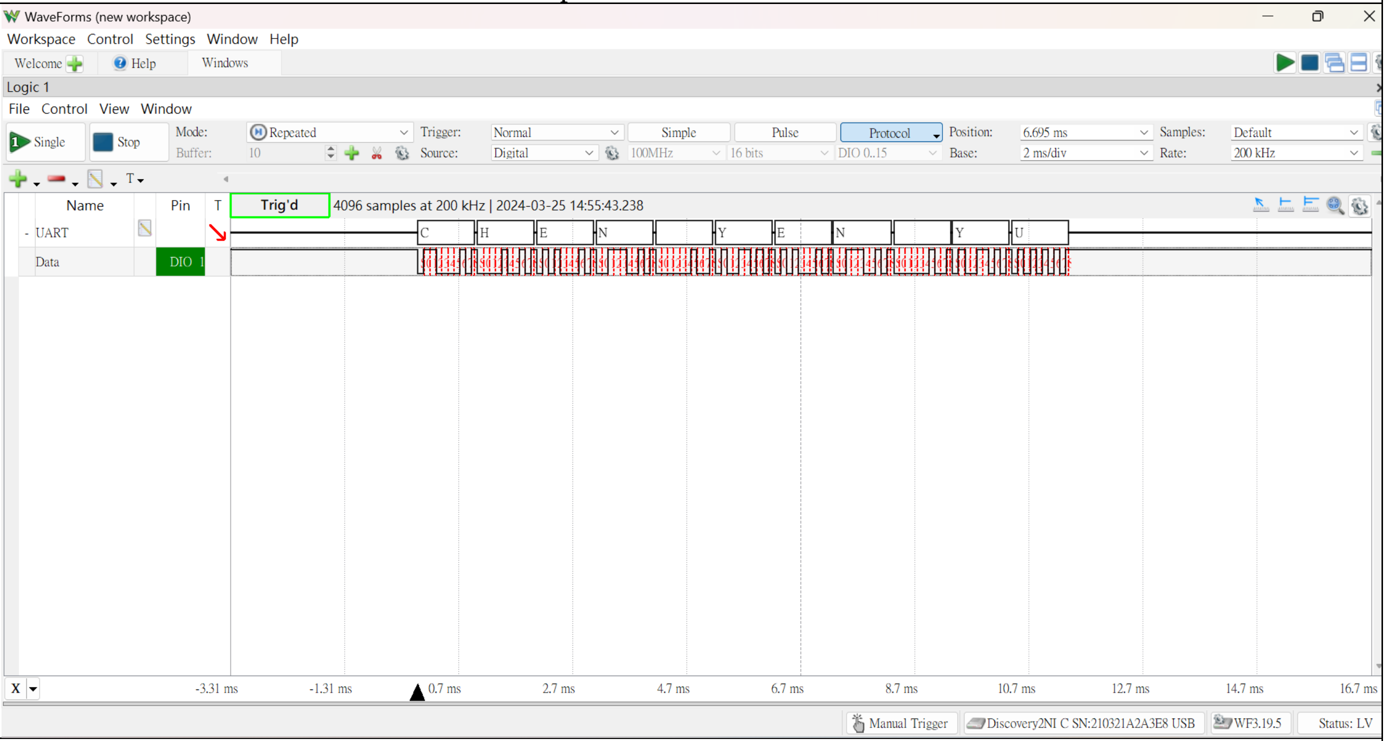
REPORT

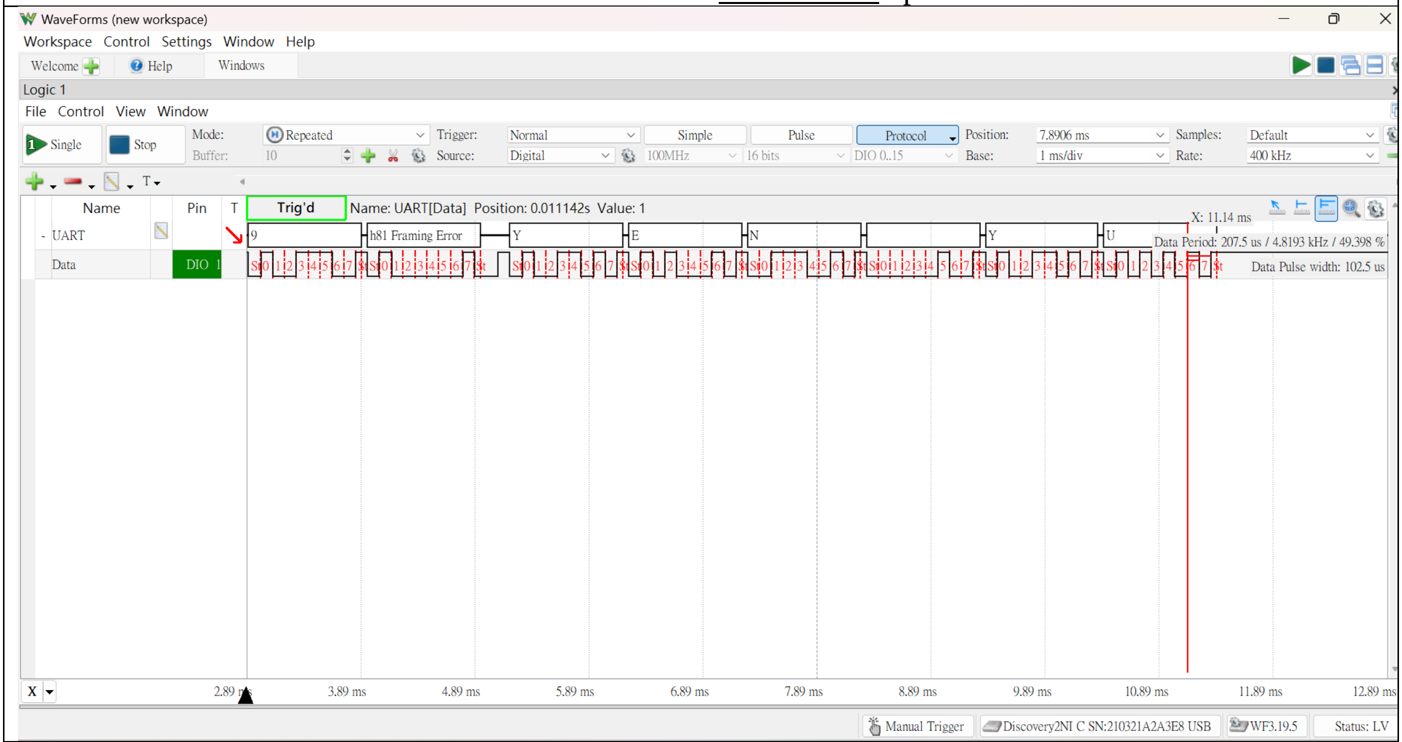
Experiment 1: Serial / UART communication

exp01-01 LA waveform

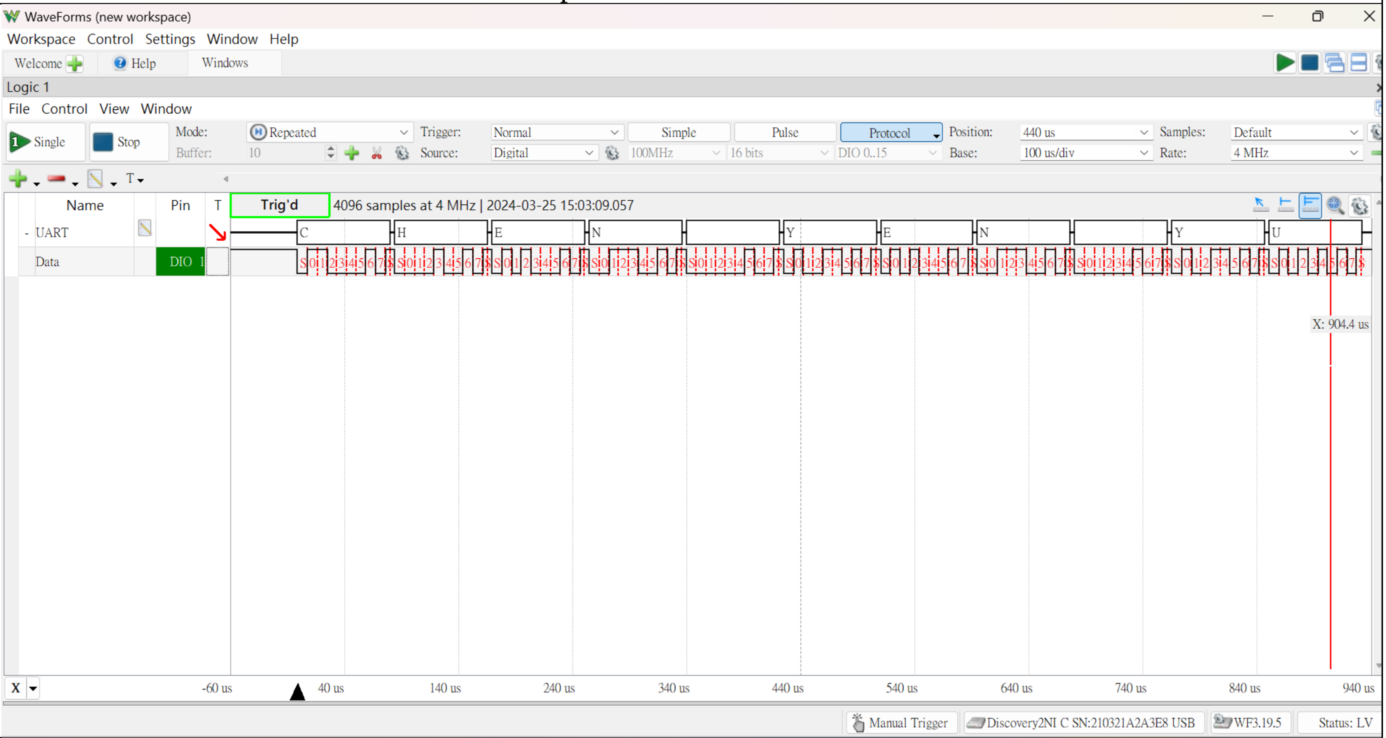


exp01-01 data transfer rate evaluation plot

data transfer rate ≈ 9638.6 bps

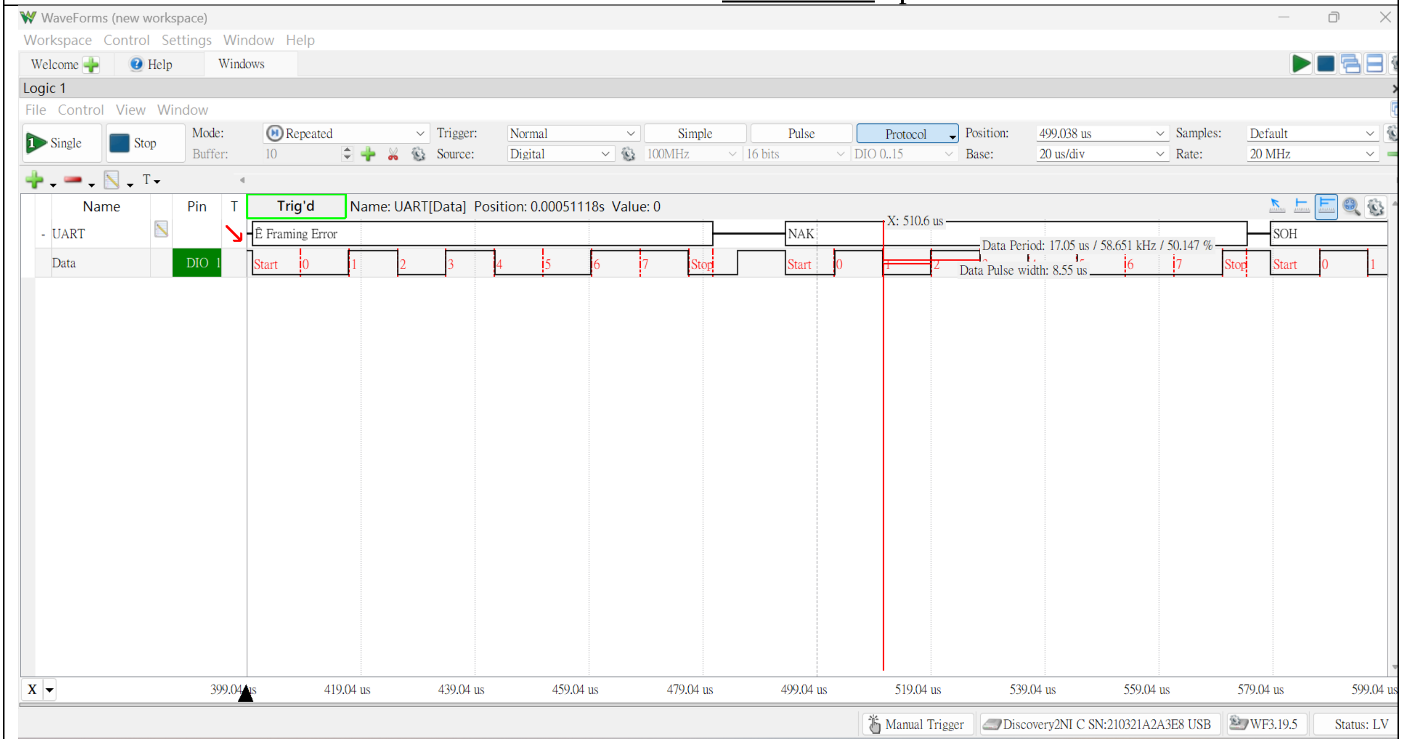


exp01-02 LA waveform

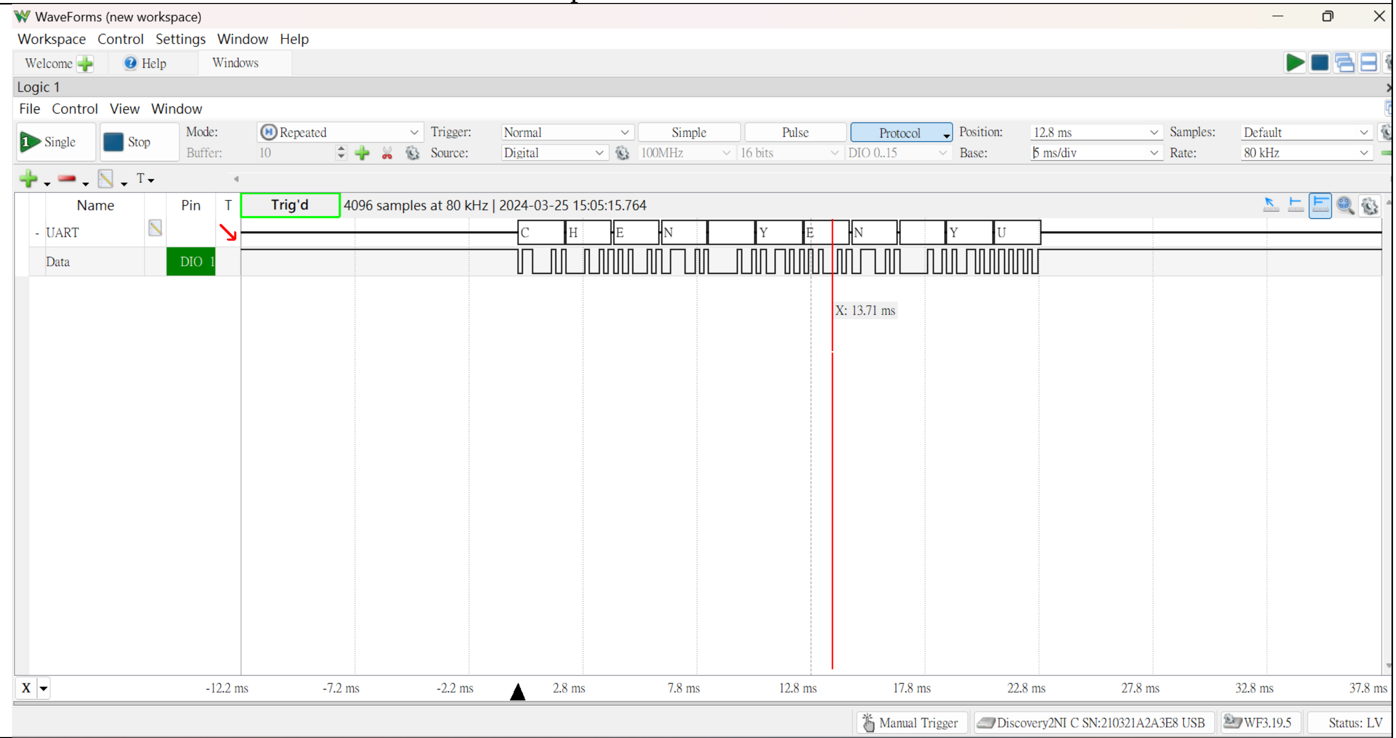


exp01-02 data transfer rate evaluation plot

data transfer rate ≈ 117302 bps

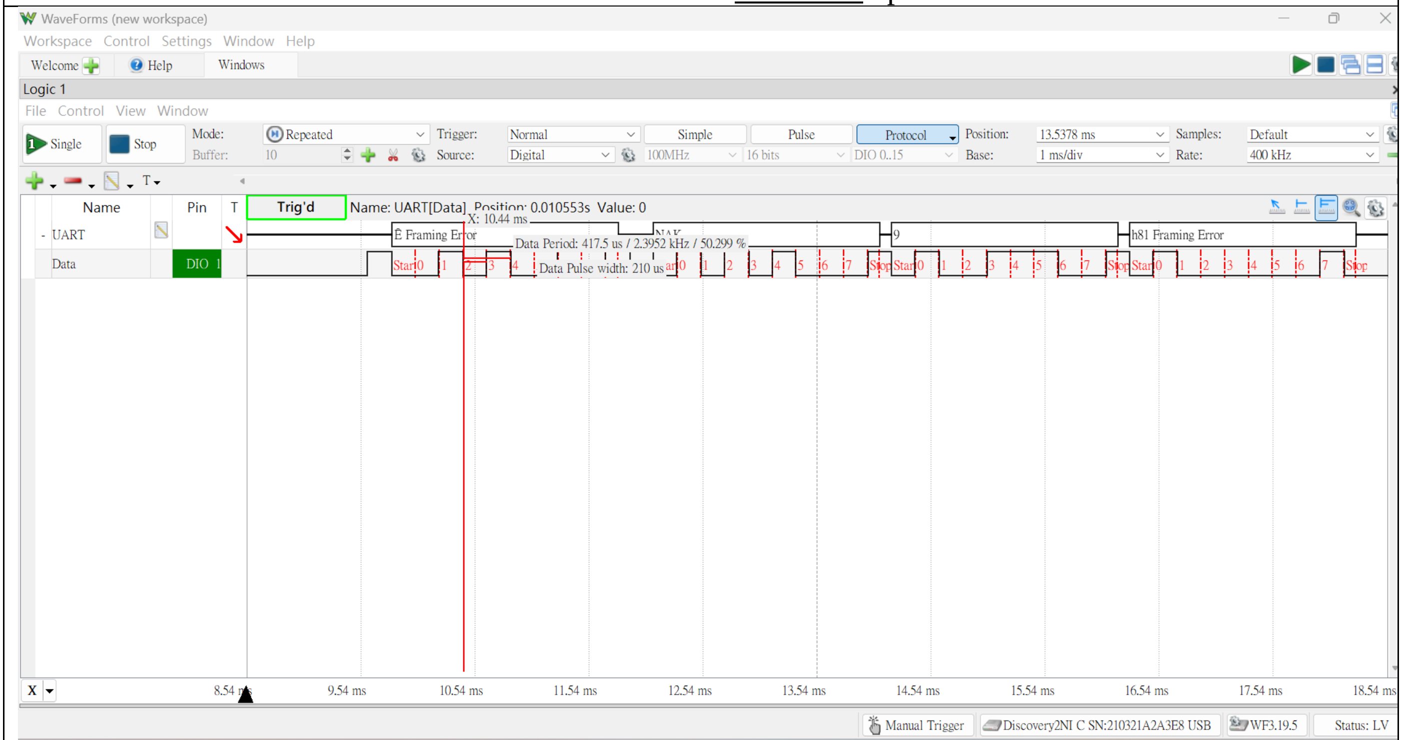


exp01-03 LA waveform

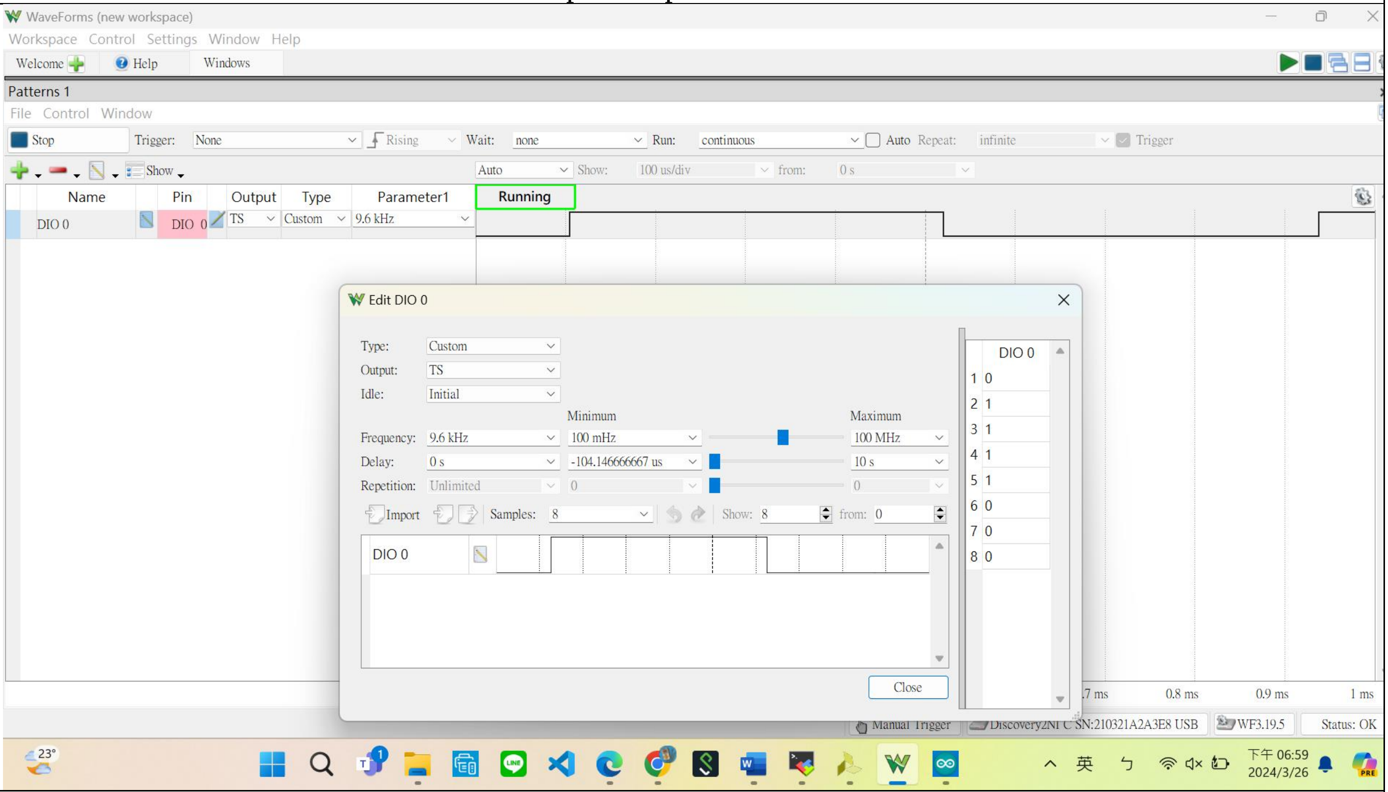


exp01-03 data transfer rate evaluation plot

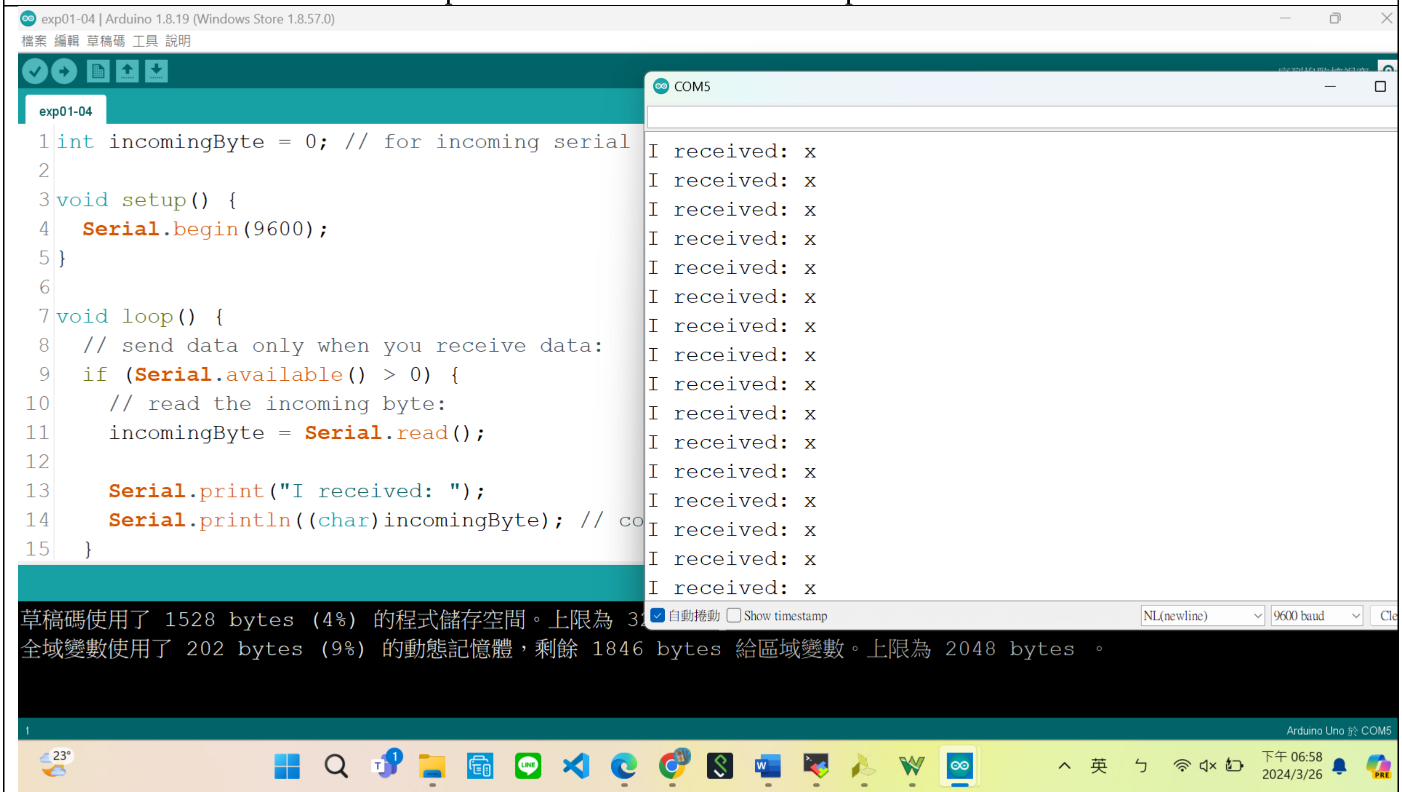
data transfer rate ≈ 4790.4 bps



exp01-04 pattern editor



exp01-04 Arduino serial monitor printscreen



What is the UART protocol?

UART (**Universal Asynchronous Receiver/Transmitter**) is a hardware communication protocol used for **asynchronous serial communication**. It is commonly used for **low-speed communication** between devices, such as between a computer and a peripheral device or between two microcontrollers. The UART protocol defines the rules for transmitting data serially, one bit at a time, over a single communication channel (usually a pair of wires). The protocol specifies the following key elements:

- **Baud Rate**: The speed at which data is transmitted, measured in **bits per second (bps)**. Common baud rates are **9600**, 19200, 38400, 57600, and 115200 bps.
- Data Bits: The number of bits used to represent a single character or piece of data. Common values are 5, 6, 7, or 8 bits.
- Parity Bit: An optional bit used for error checking. It can be set to even, odd, or none.
- Start Bit: A single bit used to indicate the beginning of a data packet.
- Stop Bit(s): One or more bits used to indicate the end of a data packet. Commonly, one or two stop bits are used.

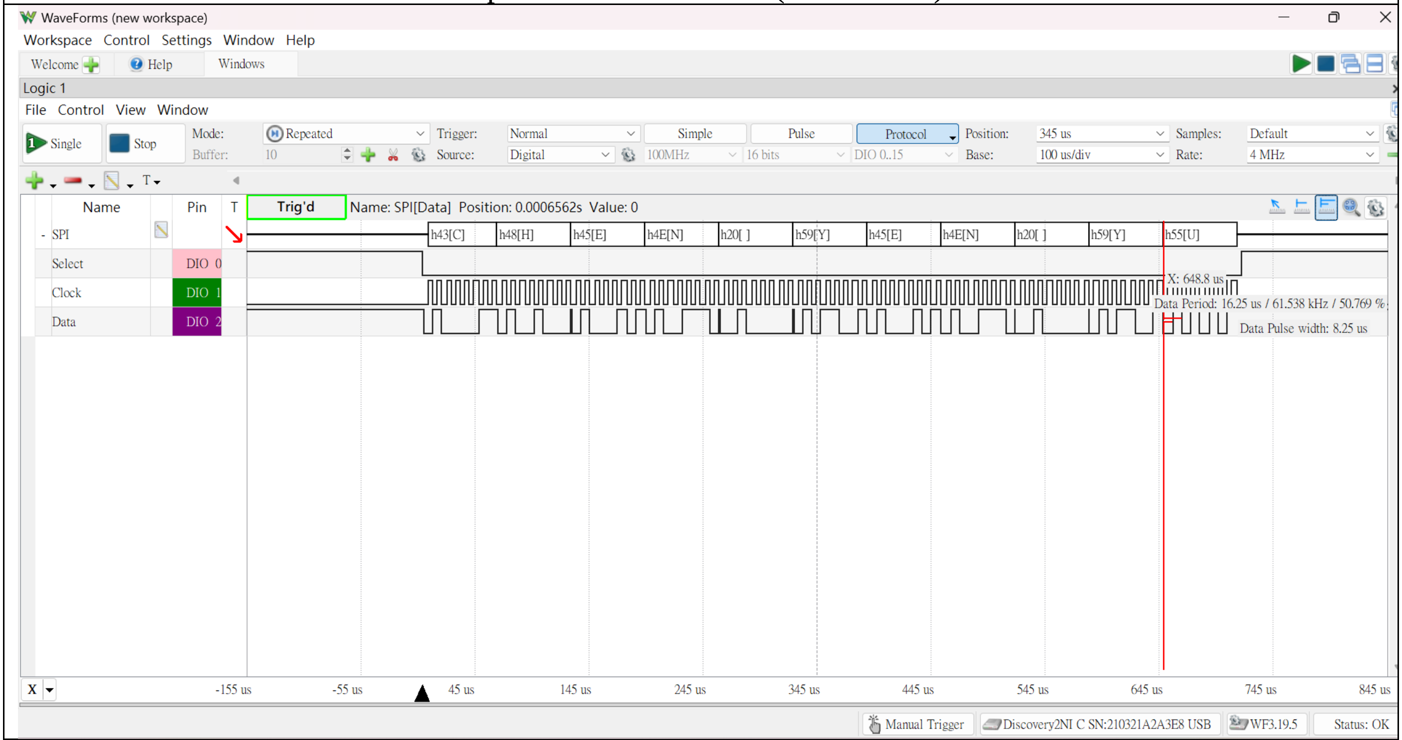
The UART protocol is asynchronous, meaning that there is **no dedicated clock signal** shared between the transmitter and receiver. Instead, **the devices must be configured with the same baud rate and other settings to ensure proper communication**.

Experiment 2: SPI communication

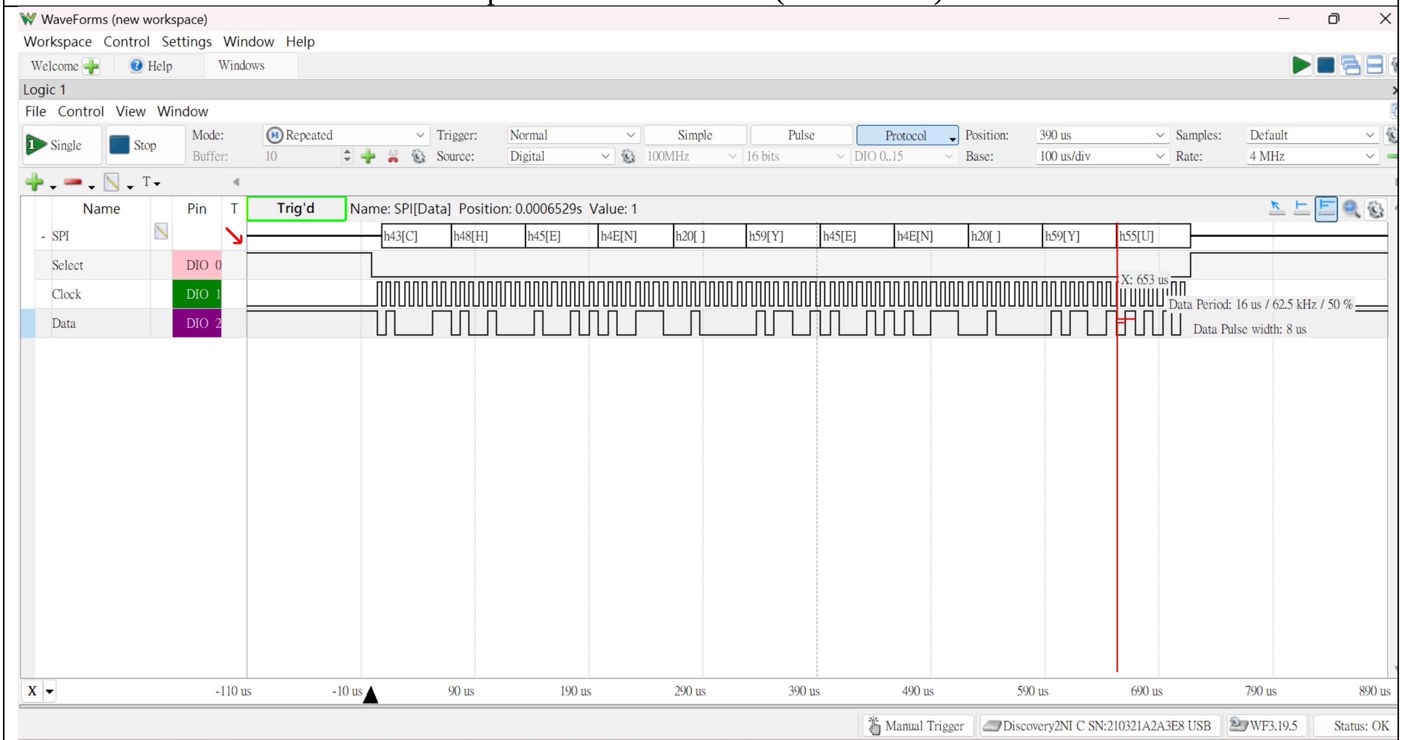
Clock Speed (Hz)	Data Transfer Rate (bps)
125k	125k

Note: adjust SPI sample timing in the “Logic” setting

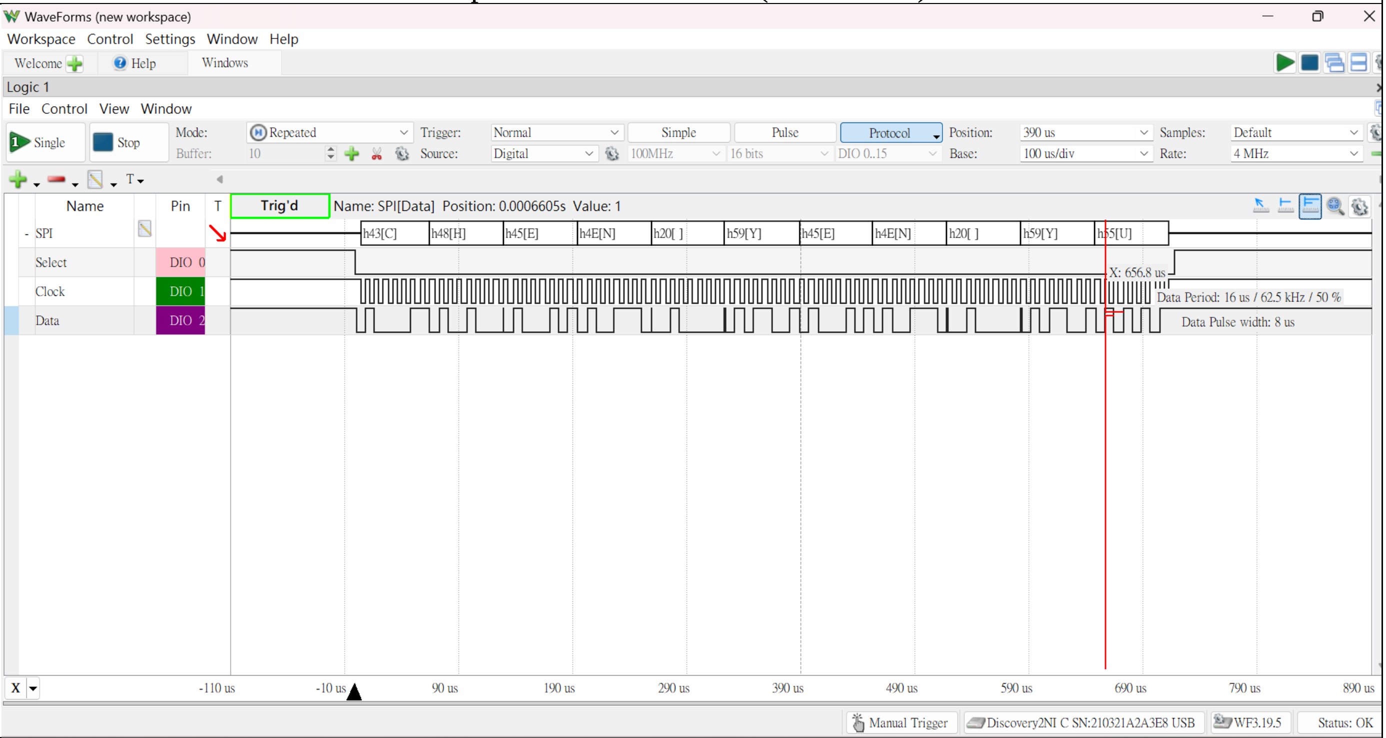
exp02-00 LA waveform (SPI mode 0)



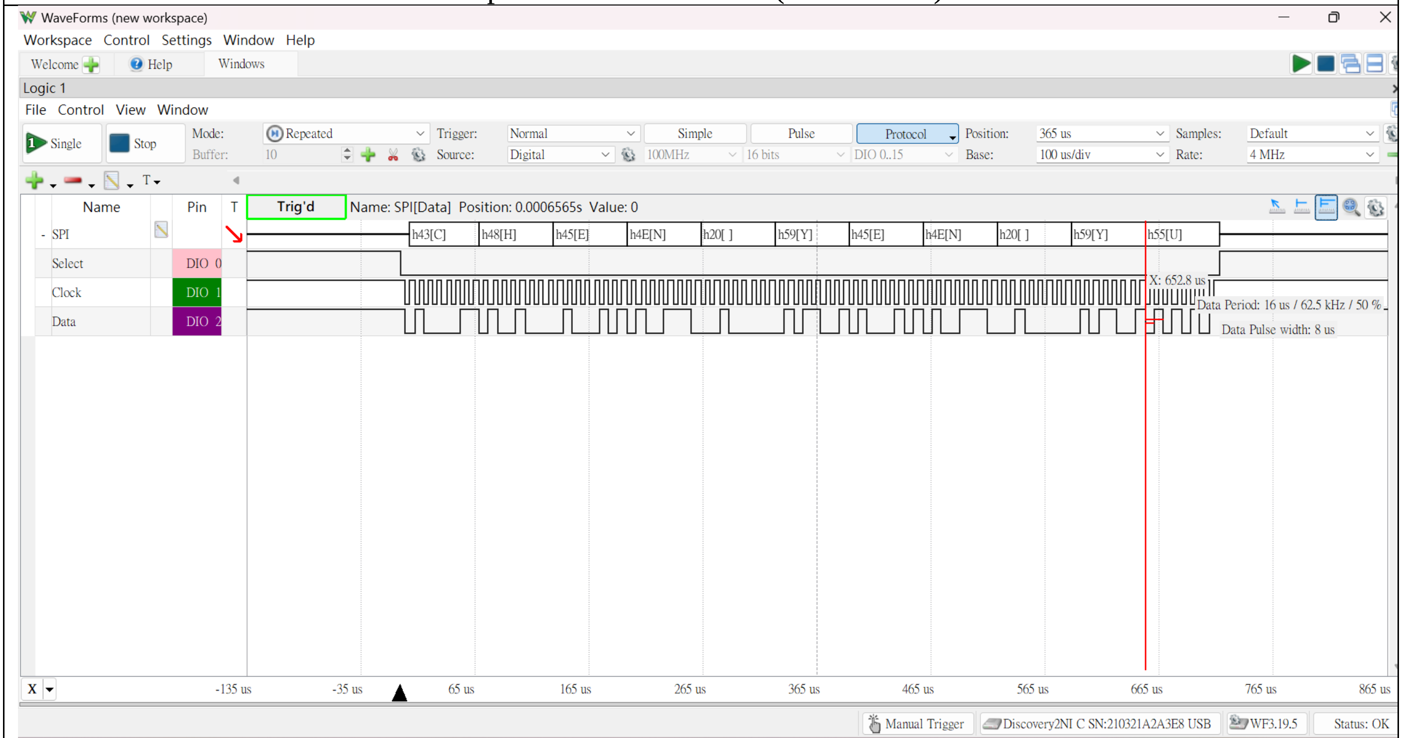
exp02-01 LA waveform (SPI mode 1)



exp02-02 LA waveform (SPI mode 2)



exp02-03 LA waveform (SPI mode 3)



What is the SPI protocol?

The SPI (**Serial Peripheral Interface**) protocol is a synchronous serial communication interface standard used for short-distance communication between microcontrollers, sensors, and other peripheral devices. It is a **full-duplex communication protocol**, meaning that **data can be transmitted in both directions simultaneously**.

The SPI protocol operates in a master-slave configuration, where a single master device initiates and controls the data transfer, while one or more slave devices respond to the master's requests. The SPI interface typically consists of four signals:

- **SCLK** (Serial Clock): This is the clock signal generated by the master device to **synchronize data transmission**.
- **MOSI** (Master Output, Slave Input): This is the data line for **transmitting data from the master to the slave**(s).
- **MISO** (Master Input, Slave Output): This is the data line for **transmitting data from the slave(s) to the master**.
- SS (Slave Select): This is a separate line for each slave device, used by **the master to select which slave it wants to communicate with**.

Arduino no-longer uses this naming convention for political correctness reasons, but the function of each signal remains the same.

The key features and characteristics of the SPI protocol include:

- **Baud rate**: In the SPI protocol, the baud rate (or data transfer speed) is **determined by the clock signal (SCLK) generated by the master device**.
- **Full-duplex** communication: Data can be **transmitted in both directions simultaneously**.
- High data transfer rates: SPI can operate at relatively high clock speeds, up to several MHz or even tens of MHz, depending on the hardware capabilities.
- Simple and efficient protocol: SPI requires fewer signal lines compared to other protocols like I²C, making it more efficient for short-distance communication.
- Flexible configuration: SPI allows for configurable data frame sizes (e.g., 8-bit, 16-bit) and clock polarity/phase settings.

Comparisons between UART & SPI:

Characteristic	UART	SPI
Synchronization	Asynchronous	Synchronous
Communication Mode	Half-duplex (**Transmit or Receive one at a time**)	Full-duplex (**Transmit & Receive at the same time**)
Configuration	Master-slave (no strict relationship)	Master-slave
Signal Lines	2 (Tx, Rx) + optional flow control	4 (SCLK, MOSI, MISO, SS)
Distance	Longer distance	Short distance
Data Transfer Speed	Lower speed	Higher speed
Serial Communication	Yes	Yes
Embedded Systems/Microcontrollers	Yes	Yes
Configurable Settings	Baud rate, data bits, parity, stop bits	Data frame size, clock polarity/phase

Experiment 3: 2-digit 7-segment display

Table Number: 23

Video link
https://www.youtube.com/watch?v=Pd7bGntNI9E

(optional) Attach your code here.

```
int a = 2;
int b = 3;
int c = 4;
int d = 5;
int e = 12;
int f = 7;
int g = 8;
int dp = 9;
int left = 10;
int right = 11;
void setup() {
  // put your setup code here, to run once:
  pinMode(a,OUTPUT);
  pinMode(b,OUTPUT);
  pinMode(c,OUTPUT);
  pinMode(d,OUTPUT);
  pinMode(e,OUTPUT);
  pinMode(f,OUTPUT);
  pinMode(g,OUTPUT);
```

```
  pinMode(dp,OUTPUT);
  pinMode(left,OUTPUT);
  pinMode(right,OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(a,LOW);
  digitalWrite(b,LOW);
  digitalWrite(c,HIGH);
  digitalWrite(d,LOW);
  digitalWrite(e,LOW);
  digitalWrite(f,HIGH);
  digitalWrite(g,LOW);
  digitalWrite(dp,HIGH);
  digitalWrite(left,LOW);
  digitalWrite(right,HIGH);
  delay(5);
  digitalWrite(a,LOW);
  digitalWrite(b,LOW);
  digitalWrite(c,LOW);
  digitalWrite(d,LOW);
  digitalWrite(e,HIGH);
  digitalWrite(f,HIGH);
  digitalWrite(g,LOW);
  digitalWrite(dp,HIGH);
  digitalWrite(left,HIGH);
  digitalWrite(right,LOW);
  delay(5);
}
```

Code Explanation:

To control multiple seven-segment displays simultaneously using an Arduino, we used a technique called **multiplexing**. In this method, we switch between activating the two seven-segs so rapidly that the human eyes don't notice the lights blinking. **My table number is 23. The upper half of the code controls the left display, 2; the lower half controls the right display, 3. There's a 5ms delay in between the two.** This works because of the **persistence of vision effect(視覺暫留)**. This effect allows our eyes to perceive rapidly changing images as a continuous image if the refresh rate is high enough.