UART

- **U**niversal **A**synchronous **R**eceiver/**T**ransmitter
- A common peripheral that is integrated in practically all modern microcontrollers
- Takes bytes of data and transmits the bits in a sequential fashion. At the receiving end (a second UART) re-assembles the bits into bytes
- Serial transmission requires only one signal wire (+ground) which makes it more cost effective than parallel transmission. Serial transmission allows much longer wire than parallel transmission.
- Asynchronous transmission allows data to be transmitted without a dedicated clock line from sender to receiver
  - Sender and receiver must agree on timing parameters (bits/second) and transmission format (number of bits, number of syncronization bits)

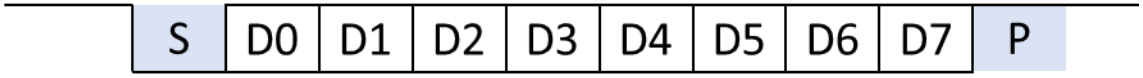
UART protocol

- In the absence of a clock signal the timing is based on bit length which must be known by both of the communicating parties
- Common data rates are 9600 bps and 115200 bps
- Timing is based on start and stop bits → some overhead in the transmission
- Transmission
  - Sender keeps the line at logical high until transmission starts
  - Transmission starts with a start bit (logical low)
  - Then sender sends a byte of data, one bit a time
  - After one byte the sender sends a stop bit (logical high)
  - Repeat for each byte of data to transmit

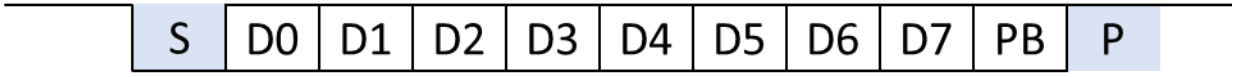
UART timing diagram

- Receiver synchronizes to start bit of every byte that is sent
- Short syncronization interval allows for some jitter in timing
  - Resynchronize on every start bit
- Transmission is essentially a state machine with states: Idle, Start, D0, D1, D2, D3, D4, D5, D6, D7, Stop
- The number of data bits can be 5 to 8 and there can also be a parity bit and the number of stop bits can be 1 or 2
  - Typically the number of data bits is 8
  - The most common setting is 8,N,1 (8 data bits, no parity bit, 1 stop bit) for general data transfer

8,N,1 data frame



8,E,1 data frame



N = no parity
E = even parity
O = odd parity
(odd parity is very rarely used with UARTs)

UART timing

- UART timing is derived from the system clock with clock dividers
- To set up the timing you need to know:
  - System clock rate (some processors have a separate peripheral clock)
  - Transmission bit rate
- Internally most UARTs use 16x clock for accurate start bit detection and centering of sampling points
- The formulas to calculate the dividers can be found in the data sheet

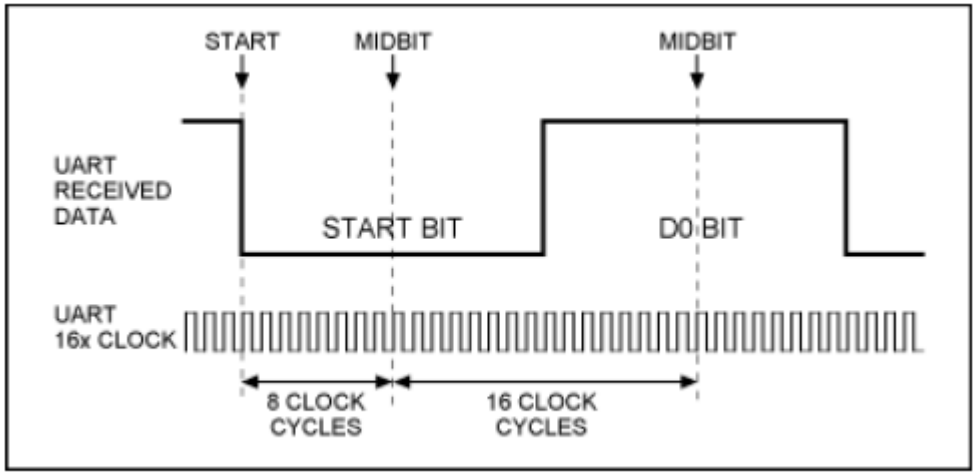


Figure 2. UART receive frame synchronization and data sampling points.

Image source: Application Note 2141: http://www.maximintegrated.com/an2141

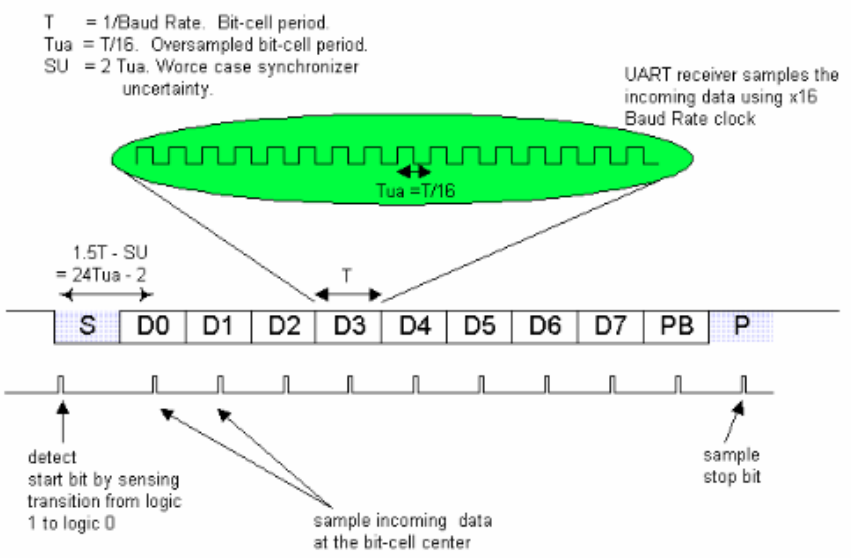
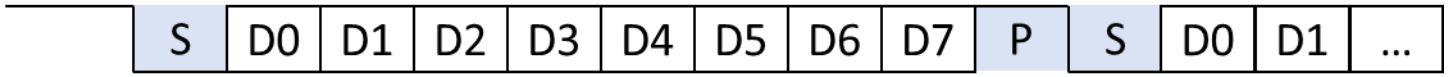


Image source: https://tutorial.cytron.io/2012/02/16/uart-universal-asynchronous-receiver-and-transmitter/

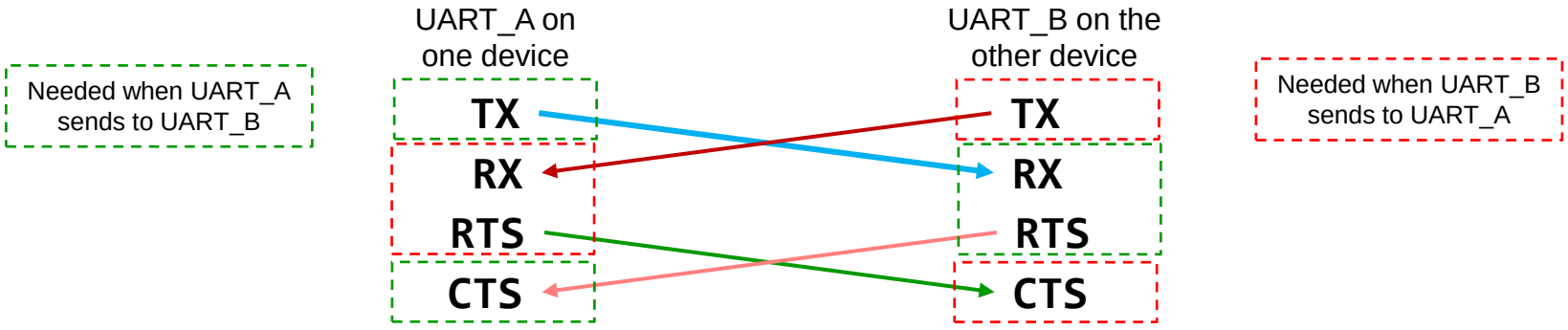
Transmission overhead and net data rate

- The minimum overhead per transmitted byte is 2 bits
- Typical 8,N,1 framing sends 10 bits per byte
  - The (maximum) net data rate = UART bit rate / 10
    - 9600 bps → 960 bytes/s
    - 115200 bps → 11520 bytes/s



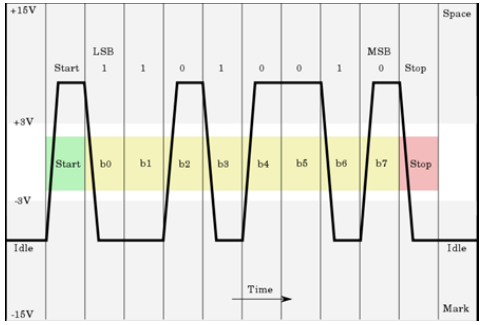
Handshake signals and wiring

- Handshake signals are used for flow control when there is a need to pause the transmission of data for example, because of processing delays
- The handshake signals are:
  - RTS (Request To Send)
    - When this signal is set it is OK to transmit data
  - CTS (Clear To Send)
    - Receiver sets this signal to tell that it is ready to receive data
- When two UARTs are communicating:
  - TX (output) of sender is wired to RX (input) of the receiver
  - CTS (input) of sender is wired to RTS (output) the receiver



RS-232 voltage levels

- RS-232 (Recommended Standard 232) is a standard for serial binary data signals connecting between a Data Terminal Equipment (DTE) and a Data Communication Equipment (DCE).
  - It is commonly used in computer serial ports.
  - One of the significant differences between TTL level UART and RS-232 is the voltage level
- Modern laptops don't have a built-in RS-232 serial port – USB-converters are typically used
  - Converters either use RS-232 voltage levels or TTL levels
    - TTL level converter can be connected directly to MCU pins
    - RS-232 converter requires a voltage converter on the MCU side



Logic	Voltage
Low	+3 – +15V
High	-3 – -15V

Typical USB UART converters

RS-232 → D9 connector

TTL → wires/pin header

Source: https://tutorial.cytron.io/2012/02/16/uart-universal-asynchronous-receiver-and-transmitter/

RP2040 UART

- Programmable baud rate generator
- 32 byte transmi FIFO
- 32 byte receive FIFO
- Polled, interrupt driven or DMA driven operation
- Based on ARM PL011 UART
- Supports automatic hardware handshakes (RTS, CTS)
  - Handshakes are seldom used in modern applications. Modern MCUs are fast enough to keep up with high data rates with IRQ/DMA. Most Pico pinout diagrams don't even show where the handshakes can be routed to because RX/TX is enough for most use cases.

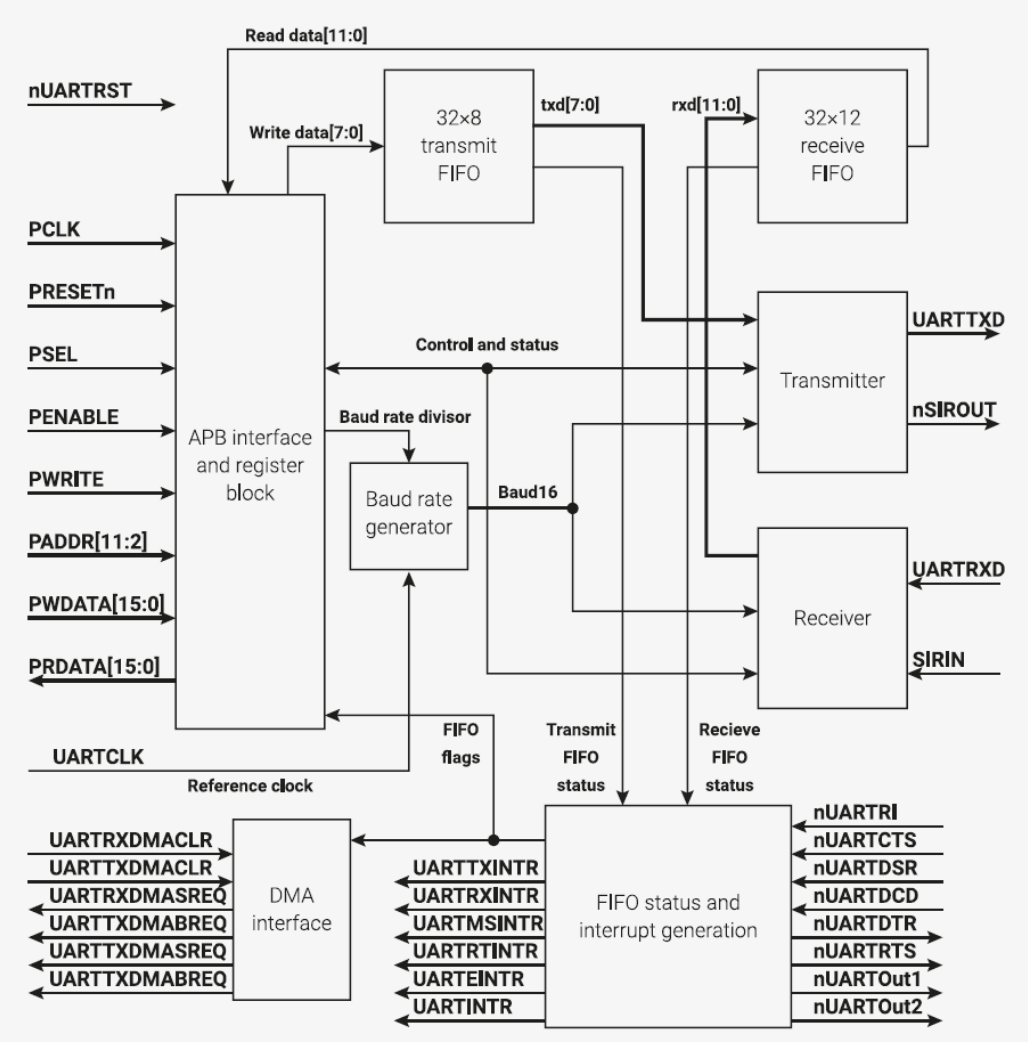


Image source: RP2040 datasheet p.418

UART operating principle

- UARTDR (Data Register) is the interface to transmit and receive FIFOs
  - Data that is written to UARTDR is placed in transmit FIFO
  - Reading UARTDR returns data from receive FIFO
    - Receive FIFO contains the received byte and receive status of the byte (framing error, parity error, break, overrun)
- UARTRSR (Receive Status register) holds the receive status
  - Overrun occurs if new value is ready in Receive Shift Register but FIFO is full – the newly received character is discarded and overrun bit is set in the status register
- UARTFR (Flag Register) status must be checked before reading or writing
  - Flag register contains transmit and receive FIFO status (empty, full), handshake status and transmit busy flags
  - Read from empty FIFO and write to full FIFO has undefined behaviour
    - Data may be read only when FIFO is not empty
    - Write is allowed only when there is space in the FIFO (not full)

Data rate and polling

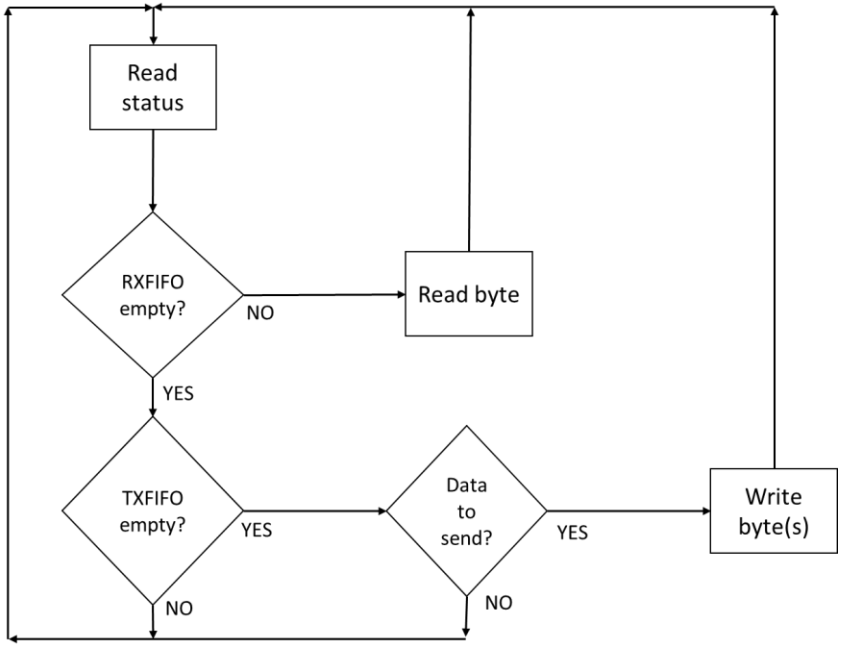
- Polling works fine as long as you poll often enough
  - FIFO makes polling easier by extending maximum poll interval to $32 \times character\ transmit\ time$
- Character transmit times at common data rates
  - 9600 bps – 1.04 ms → 33 ms poll interval with 32 byte FIFO
  - 115200 bps – interrupts at 87 μs intervals → 2.7 ms poll interval with 32 byte FIFO

Data rate and interrupt load

- UART can be configured to generate an interrupt when the status bits are set
  - Which bits generate interrupts is configurable
  - Buffered UART – for example 16 byte FIFO reduces interrupt load and allows longer ISR response time without losing data
- Character transmit times at common data rates
  - 9600 bps – 1.04 ms → 33 ms interrupt interval with 32 byte FIFO
  - 115200 bps – interrupts at 87 μs intervals → 2.7 ms interrupt interval with 32 byte FIFO
  - FIFO interrupt threshold is configurable. Usually interrupt is triggered before FIFO is completely full which relaxes interrupt latency requirements
- DMA based transfer – risk of losing data is minimal
  - Direct transfer from FIFO to RAM
  - DMA transfers are quite challenging to program compared to traditional ISR based transfers

UART

- The following priciples apply both to ISR driven and polled UART handling
- Prioritize reading over writing
  - Usually, you can buffer/delay your writing, but the only way to prevent overrun is to read character(s) before the next one is received
- Handshake signals can be used to to tell the sender if it is OK to send or not
  - Handshaking adds complexity to UART handling



UART example

- Note that you are dealing with raw data – there is no backspace or end of line processing
- Sender may send the data at full speed with pause between bytes which means that you need to poll often enough to get all bytes
- None of the UART receive functions make strings – you always need to add zero to the end yourself
- `uart_getc()` is a blocking function that does not return until a byte was received

```
// Set up our UART with the required speed.
uart_init(uart0, BAUD_RATE);

// Set the TX and RX pins by using the function select on the
GPIO
// See datasheet for more information on function select
gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);
gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);


const uint8_t send[] = "test\n";
char str[STRLEN];
int pos = 0;

while(true) {
    if(!gpio_get(button)) {
        while(!gpio_get(button)) {
            sleep_ms(50);
        }
        uart_write_blocking(uart0, send, strlen(send));
    }
    while(uart_is_readable(uart0)) {
        char c = uart_getc(uart0);
        if(c == '\r' || c == '\n') {
            str[pos] = '\0';
            printf("received: %s\n", str);
            pos = 0; // start over after line is printed
        }
        else {
            if(pos < STRLEN - 1) {
                str[pos++] = c;
            }
        }
    }
}
```

Read one character