Embedded systems engineering
Distributed real-time systems

David Kendall

Introduction

- Serial communication
- UART
- mbed Serial libraries
  - Serial
  - RawSerial
  - UARTSerial
- Application protocols

Serial communication

Serial communication involves sending data over a communication link one bit at a time

Contrast with parallel communication in which multiple bits are sent simultaneously

Serial communication may be *synchronous*, i.e. the communication includes a separate clock signal

Serial communication may be *asynchronous*, i.e. the communication does not include a separate clock signal, instead timing is determined by the presence of additional bits in the data signal, e.g. start and stop bits

Focus in this lecture is on asynchronous serial communication using UARTs

This is one of the simplest methods of communication between computing nodes

Universal Asynchronous Receiver-Transmitter (UART)

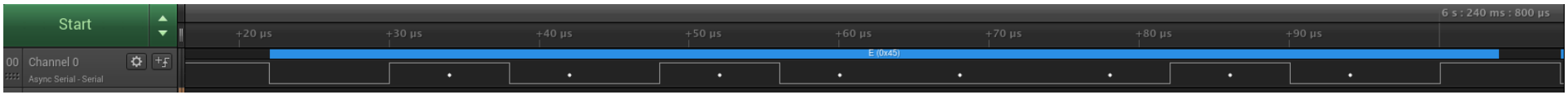
UART can receive a sequence of bits on a single wire (RX) and assemble them into a byte for presentation to a microprocessor

UART can take a byte from a microprocessor and transmit it as a sequence of bits on a single wire (TX)

Conversion from bits to bytes (and vice versa) is performed by a *shift register*

The number of data bits per byte is often configurable, e.g. 5, 6, 7 or 8

Usually, the least significant bit (LSB) is transmitted first, e.g.



Transmission of a byte begins with a *start* bit (low) and ends with one or two *stop* bits (high). The transmission may also contain a *parity* bit (odd/even)

Parity

- Use of a parity bit is optional
- Parity can be configured as *odd* (O), *even* (E) or *none* (N)
- Odd
  - The value of the parity bit is set so that there are an *odd* number of 1-bits in the transmitted byte (including the parity bit).
- Even
  - The value of the parity bit is set so that there are an *even* number of 1-bits in the transmitted byte (including the parity bit).
- None
  - A parity bit is not transmitted
- Example - consider the character **E (0x45)**
  - Bit representation is 01000101
  - Odd parity – value of parity bit is 0 – 0 01000101
  - Even parity – value of parity bit is 1 – 1 01000101

Baud

Baud is a measure of the transmission rate of the communication

Baud – the number of *symbols* transmitted per second

When there are only 2 symbols (e.g. 0 and 1) then baud is equivalent to the bit rate, i.e. number of bits per second

If the medium is capable of representing more than 2 symbols then the bit rate is greater than baud

For example, assume a line has 4 signal levels, 0V, 3V, 6V, and 9V. Then each level is capable of representing one of four different symbols. In this case, the bit rate would be twice the transmission rate given in baud, e.g. 1000 baud would be 2000 bps

In general, for a system with M distinct symbols and a rate f_s of symbols per second, the bit rate R in bits per second is given by

$$R = f_s \cdot N$$

where $N = \log_2(M)$

Full duplex, half duplex, simplex

Full duplex

- Ability of a UART to send and receive data simultaneously
- Requires separate transmit and receive circuitry

Half duplex

- Transmission is possible in both directions *but not at the same time*

Simplex

- Transmission is possible in one direction only

Most UARTs today have full duplex capability

UART configuration

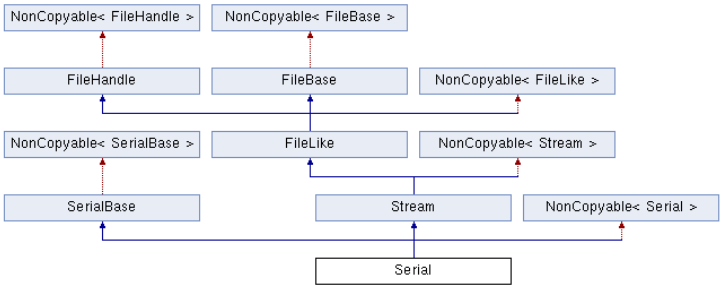
A typical format for indicatinga UART's configuration is
<baud> <data bits> <parity> <stop bits>

Examples

- 9600 8N1
  - 9600 baud, 8 data bits, no parity, 1 stop bit
- 115200 7E2
  - 115200 baud, 7 data bits, even parity, 2 stop bits
- 38400 5O1
  - 38400 baud, 5 data bits, odd parity, 1 stop bit

Note it is important that both communicating nodes are configured with the same parameters. If they are not, communication may not occur at all or, if it does, the data may not be received correctly

Serial Class



Basic Examples

```
Serial pc(USBTX, USBRX, 115200);
Serial sp(D1, D0);
sp.baud(38400);
sp.format(7, SerialBase::Even, 2);
```

Blocking Examples

```
c = sp.putc('H');
c = sp.getc();
```

Non-blocking example

```
uint8_t clientBuffer[128];
volatile bool messageReceived;

void serialCbHandler(int events);

int main() {
  event_callback_t serialCb = serialCbHandler;
  int result;

  ...
  result = client.read(&clientBuffer[0], 20,
                       serialCb);
  ...
}

void serialCbHandler(int events) {
  messageReceived = true;
}
```

Complete Documentation

Serial Events

An `event_callback_t` function is passed an integer argument describing the events that triggered the interrupt, so it can take an appropriate action, depending on the events

TX events

SERIAL_EVENT_TX_COMPLETE
SERIAL_EVENT_TX_ALL

RX events

SERIAL_EVENT_RX_COMPLETE
SERIAL_EVENT_RX_OVERRUN_ERROR
SERIAL_EVENT_RX_FRAMING_ERROR
SERIAL_EVENT_RX_PARITY_ERROR
SERIAL_EVENT_RX_OVERFLOW
SERIAL_EVENT_RX_CHARACTER_MATCH
SERIAL_EVENT_RX_ALL

More serial classes

May be best to use one of the following classes when communicating with a device for which supporting a stream API (`printf`, `scanf`, etc) is difficult

RawSerial

- Variation of the Serial class
- Doesn't use streams
- Safe for use in interrupt handlers with the RTOS

UARTSerial

- Provides a buffered UART communication facility
- Separate circular buffers for send and receive channels

Application protocols

The mbed libraries provide mechanisms for achieving communication between two devices connected serially but . . .

Each application needs a suitable application protocol

An application protocol determines

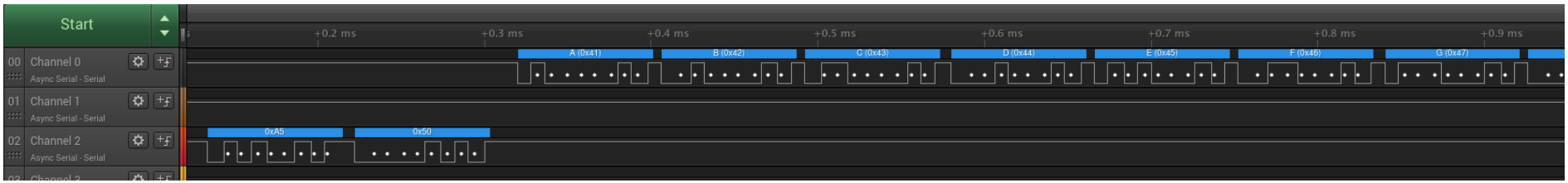
- how data is represented
  - strings
    - ★ simple strings
    - ★ structured, e.g. JSON
  - binary
    - ★ little endian
    - ★ big endian
- how data is delineated (where a ‘unit’ of data begins and ends)
  - fixed length
  - length encoded
  - null terminated
  - eof terminated

Simple application protocol – call and response

‘Client’ sends fixed length sequence, e.g.

- 0xA5 – the next byte is a command
- 0x50 – send a string

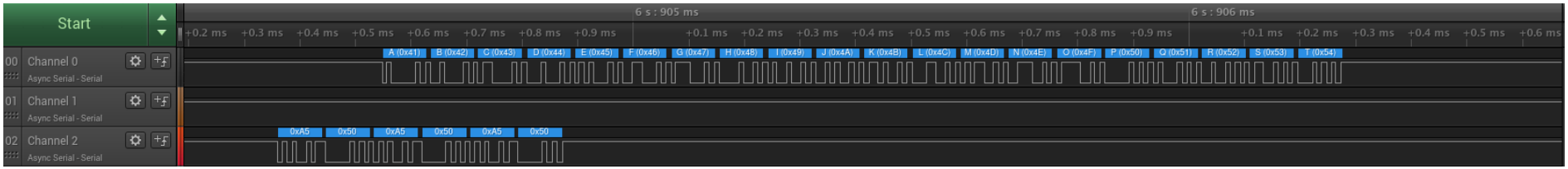
‘Server’ responds with a fixed length string



What might go wrong here?

A more robust application protocol

‘Client’ repeatedly sends command sequence until the ‘server’ responds



If blocking calls are used, it can be difficult to detect the server response in time to stop the sending of further (unnecessary) commands

Non-blocking calls can help to solve this problem

Summary

A UART provides a simple mechanism to allow two devices to communicate serially

The configuration of the UARTs must match each other for successful communication

- Baud
- Number of data bits
- Parity – None, Even, Odd
- Number of stops bits – 1 or 2

mbed libraries provide both blocking and non-blocking reads and writes

It is important that the devices agree on an *application protocol*