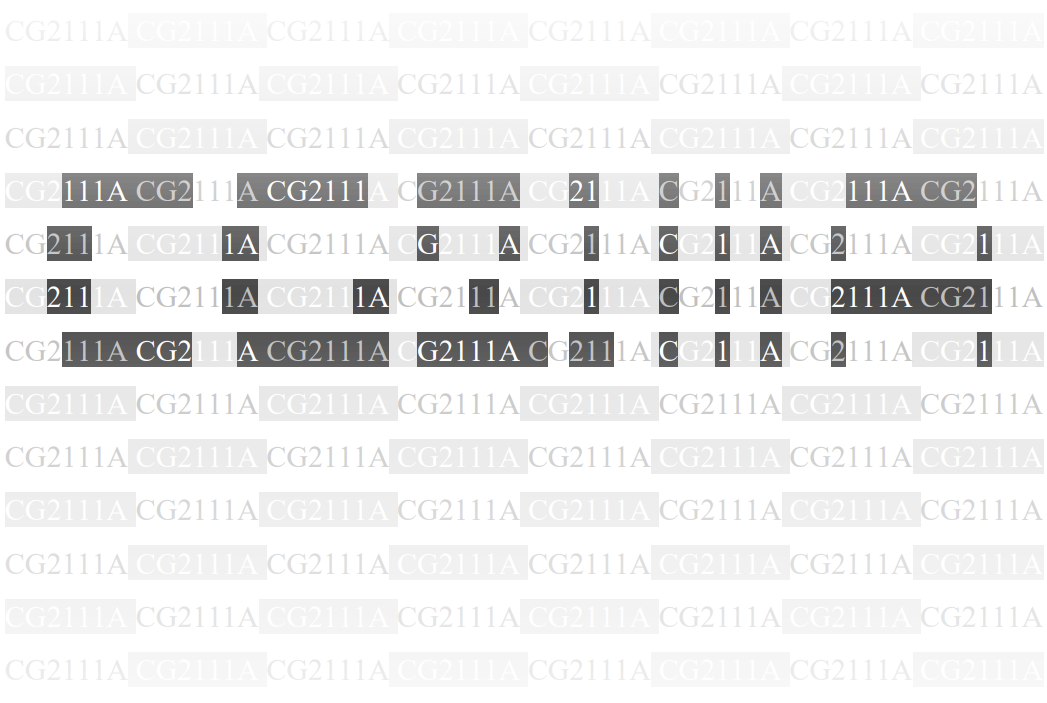
CG2111A Notes 2 —— GYN and friends



**ADC**

A **successive approximation ADC(may explain according to tut3 qn2)** works by sampling the input signal at a specific rate, which is determined by the ADC’s clock frequency. This clock frequency is often set by a prescaler.

The result is generated 1 bit at a time, starting from the Most Significant Bit.

Channel and reference selection are continuously updated before the conversion, but are locked when the conversion is in progress.

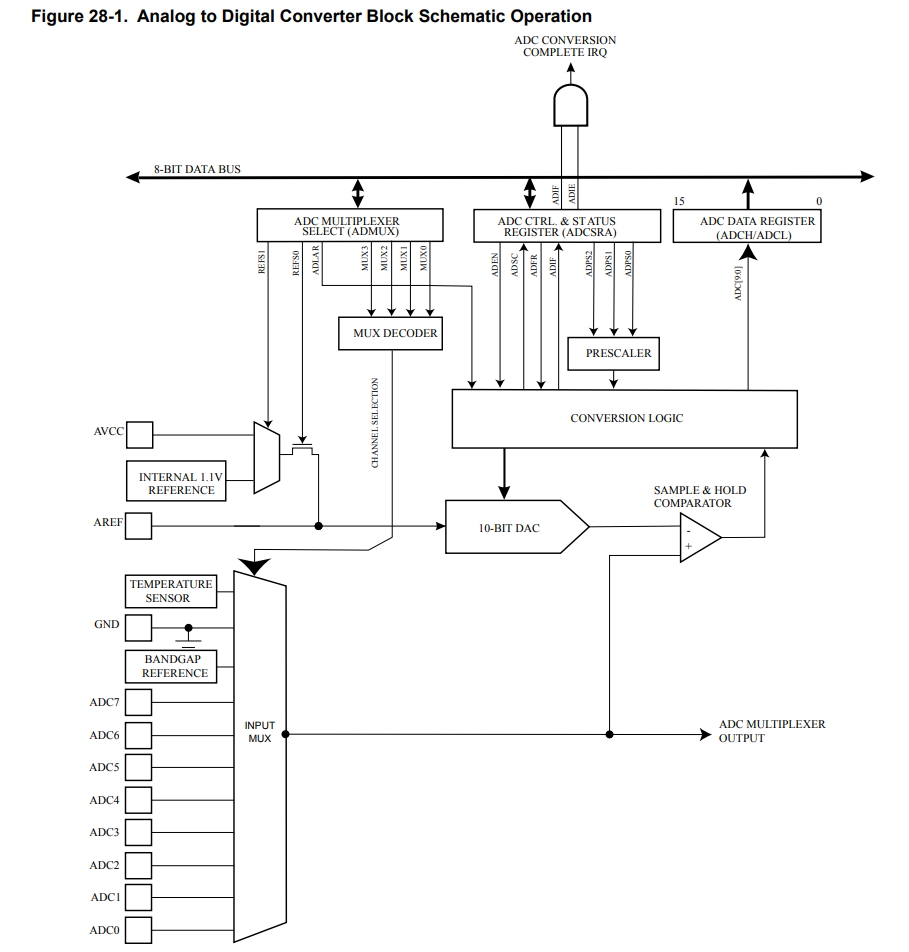
The **Nyquist Sampling Rate:** at least twice as fast as the highest frequency content of the signal so as to accurately represent the signal.

**Resolution:** voltage distance between two adjacent quantization levels.

A normal conversion takes 13 ADC clock cycles.

Basic Abbreviation we use:

MSB and LSB——Most and Least Significant Bit



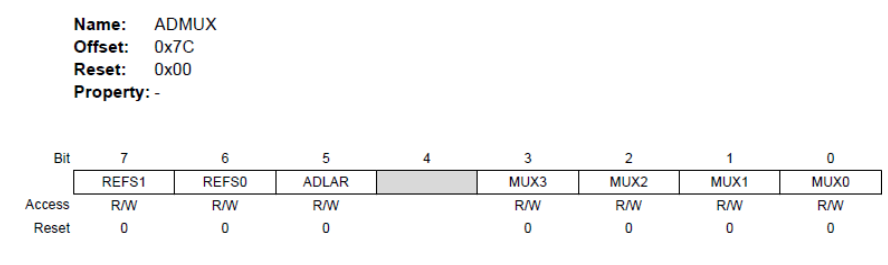
Input channel selected by writing to the MUX bits in ADC Multiplexer Selection Register, ADMUX.MUX[3:0]. Any ADC input pins, as well as GND and a fixed bandgap voltage reference, can be selected as single ended input.

ADC is enabled by writing ‘1’ to the ADC Enable bit in **ADC Control and Status Register A (ADCSRA.ADEN)**.

Start Conversion: by writing ‘0’ to power reduction ADC bit in Power Reduction Register (PRR.PRADC) and ‘1’ to ADC start conversion bit in ADCSRA (SDCSRA.ADSC). It will stay high as conversion is in progress and cleared by hardware afterwards.

ADC then Generate 10-bit result presented in ADC Data Register ADCH and ADCL.

**Programming Procedure**

1. Activate power, write ‘0’ to PRR.PRADC
   * 1. PRR &= 0b11111110
2. Switch on ADC, ‘1’ to pin 7 of ADCSRA (enable)
   1. Configure Prescaler, pin[0:2] of ADCSRA
      1. ADCSRA = 0b10000111;
3. Choose input and ref voltage
   1. 
      1. eg. ADMUX = 0b01000010;
4. Start Conversion (ADSC)
   * 1. ADCSRA |= 0b01000000;
5. Poll, loop till ADSC cleared by hardware
   * 1. while(ADSRA & 0b010000000);
6. Read Result (if right adjusted, always read ADSL first; If the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH.)
   * 1. loval = ADSL; // introduce variable to store value, as ADCL and ADSH may be overwritten
     2. hival = ADSH; // This is because ADC’s access to ADC data register is blocked after reading ADCL and only re-enabled when ADCH is read. This way we can read the full data before re-enabling.
     3. adval = (hival <<8) + loval; // combine together to get 10 bit.

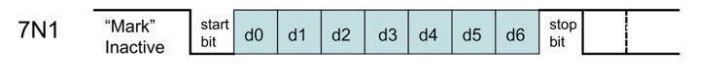
**Serial Communications and USART, Universal Synchronous Asynchronous Receiver Transceiver**

In asynchronous mode: data transmitted independently of any clk signal (it uses internal clock). The receiver derives a clocking signal from the received data.

In synchronous mode: Data transmit and received are coordinated by a separate clock.

We focus on **UART.**

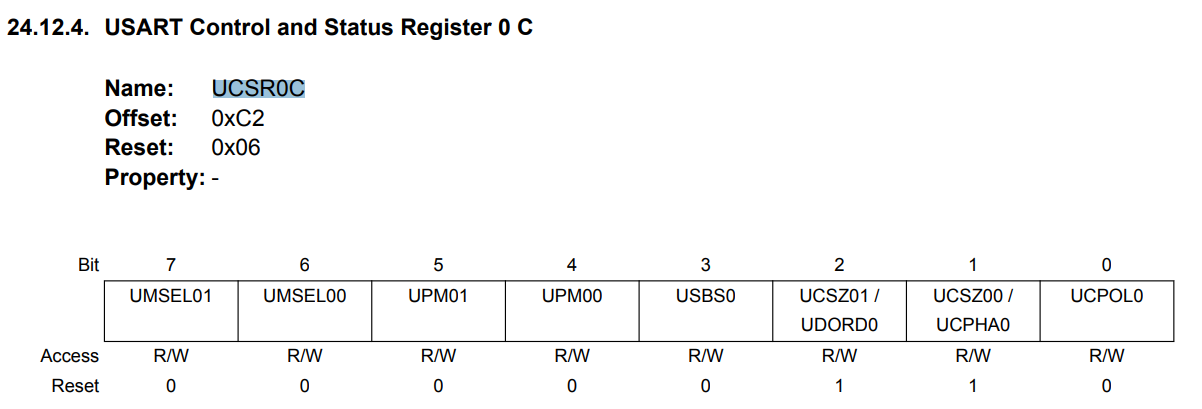
We are concerned with layer 1 to 3 of ISO OSI Model

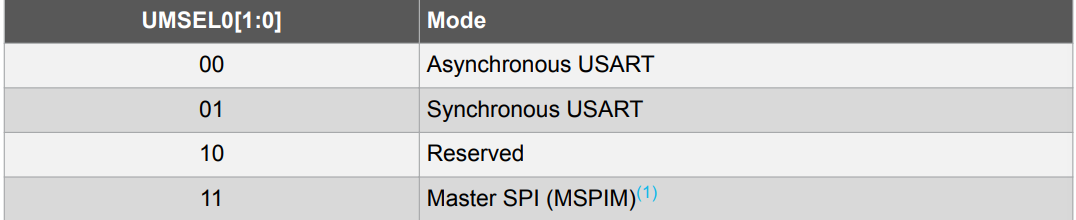
* Physical Layer
  + Total of three wires:
    - Receive (RX): Incoming data bits come here.
    - Transmit (TX): Outgoing data bits go out from here.
    - Ground (GND): Return path for RX and TX. Both receiver and transmitter must share a common ground.
  + Voltage Level
    - 5v for standard devices (e.g. Arduino UNO)
    - 3.3v for low-power devices (e.g. Raspberry Pi)
    - When connecting a 5v and 3.3v device together, you SHOULD use a level converter
* Data Link Layer
  + USART session between two computers, both sides agree on
    - Number of **data bits – 5, 6, 7, 8 or 9. Standard is 8**.
    - Type of **parity bit: None (N), Even (E) or Odd (O)**.
    - Number of **stop bits: 1 or 2.**
    - Bit rate: 1200, 2400, …, 115200 (not the same as baud rate)
  + on parity
    - The extra bit is set to 1 to make the total number of 1 bits odd/even.
    - For error checking, the receiving side will count the number fo 1 bits to ensure it is odd/even as previously agreed.
    - 
    - 

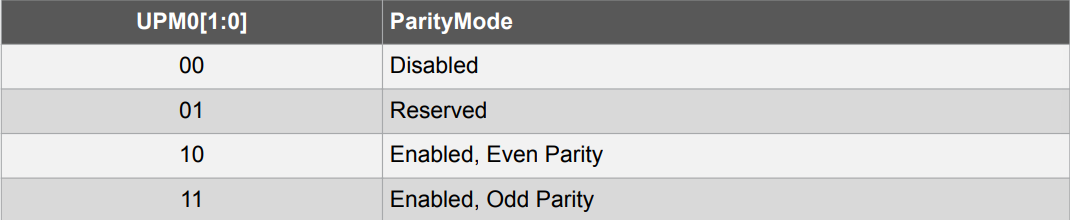
**Program UART Port**

1. Set up
2. Sending/ Receiving in Polling Mode
3. Sending/ Receiving in Interrupt Mode

**Setting up**



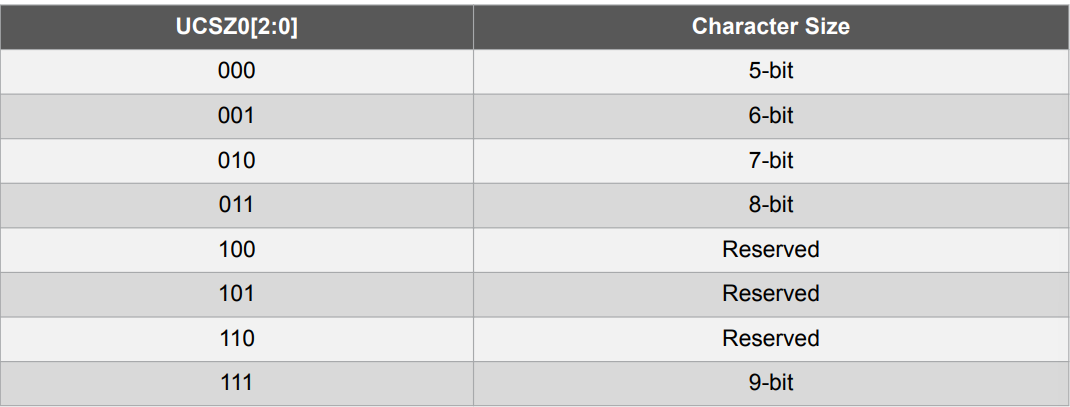
Bit 7:6 Mode 

Bit 5:4 Parity

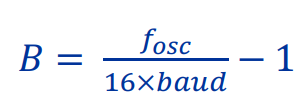
Bit 3 Stop Bit

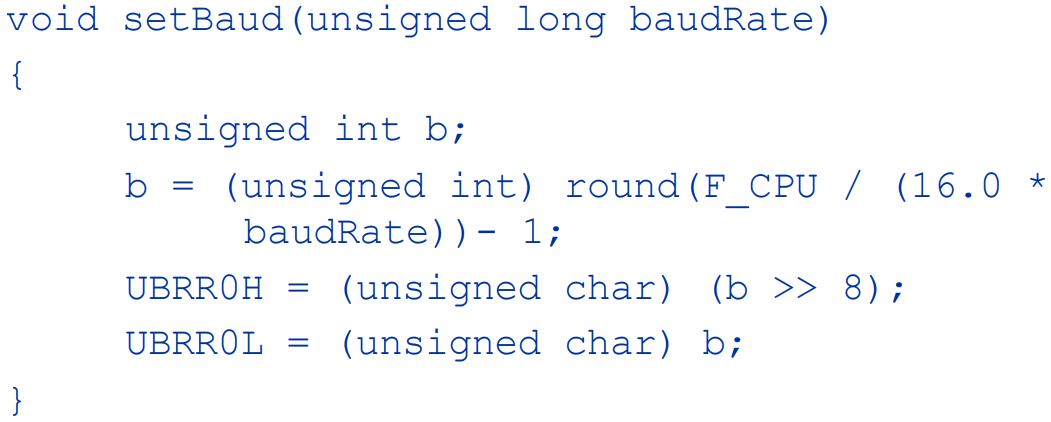


Bit 2:1 Character Size and bit 2 of UCSR0B



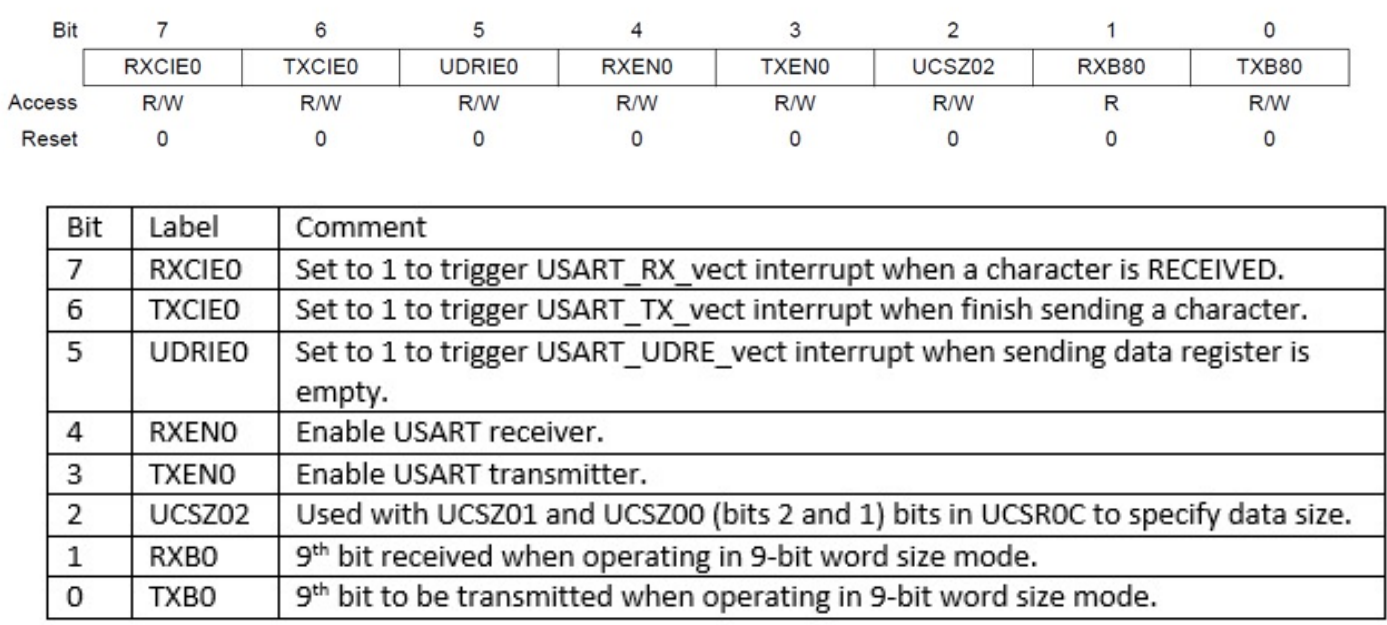
Bit 0 Clock Polarity Setting: 0 for Asynchronous mode

**Setting Baud Rate** : ****round to nearest int before subtracting 1

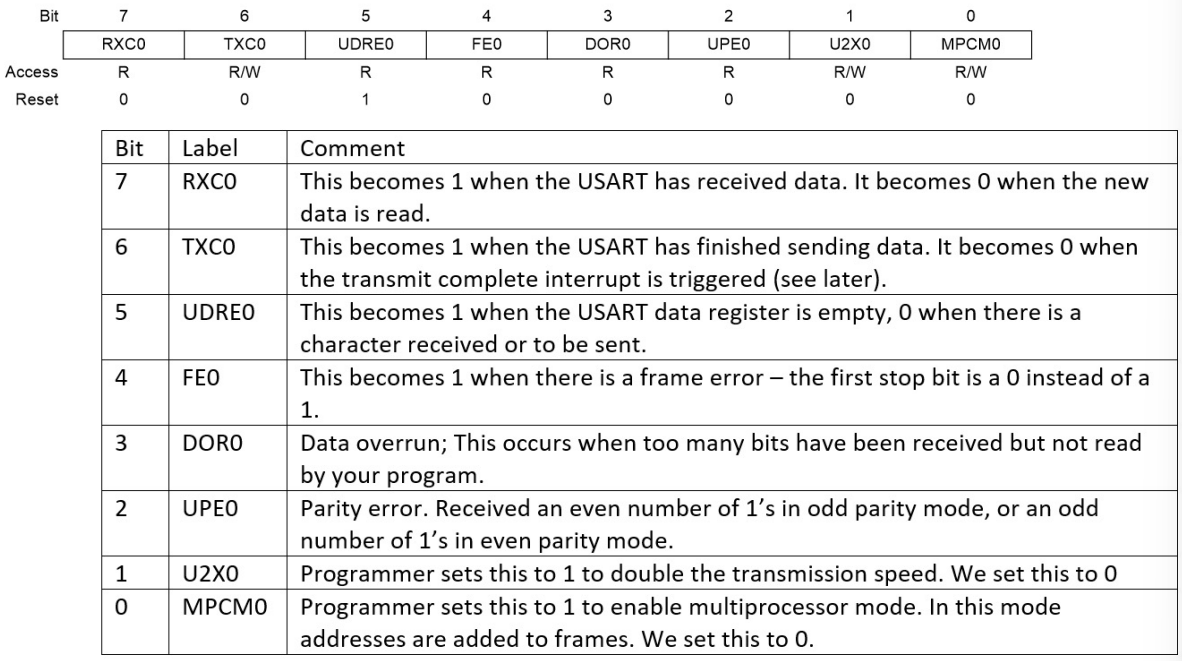


**Enable/Disable Features**

The **UCSR0B** register is used to enable/disable various interrupts, and to enable/disable the transmitter and receiver.



**UCSR0A**



**Sending/ Receiving in Polling Mode**

**//** Set up UCSR0C and UBRR0H and UBRR0L…and UCSR0A = 0;

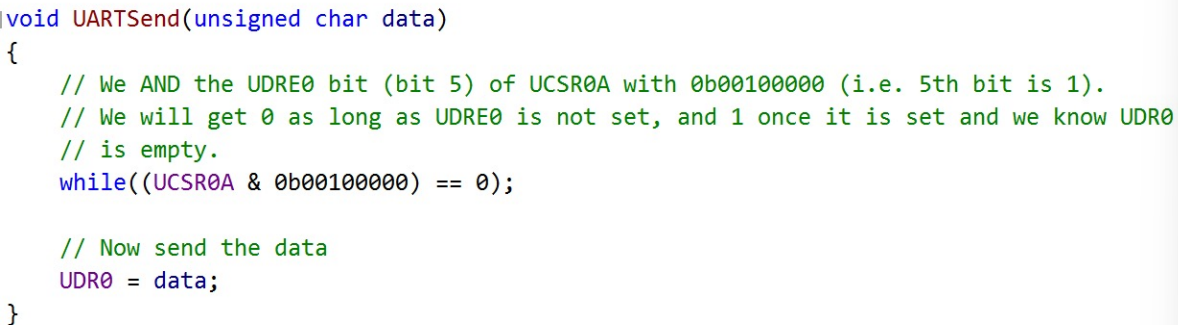
UCSR0B = 0b00011000; // disable all INT, enable receiver and transmitter

**USART Data Register**

Data can be read from UDR0 when bit 7 (RXC0) of UCSR0A is set or write to it when bit 6 (TXC0) in UCSR0A is set

**Send:**

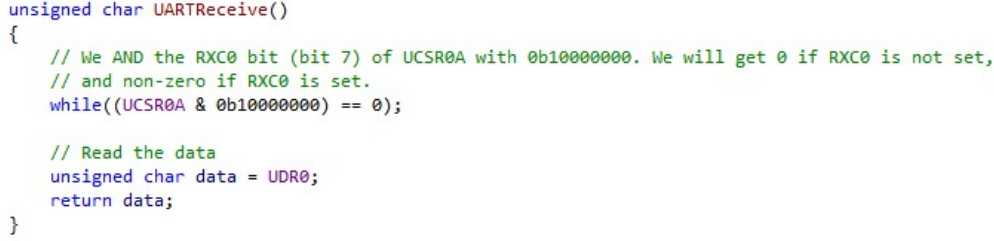
In the code below, UDRE0 (data register empty is polled instead of TXC0).



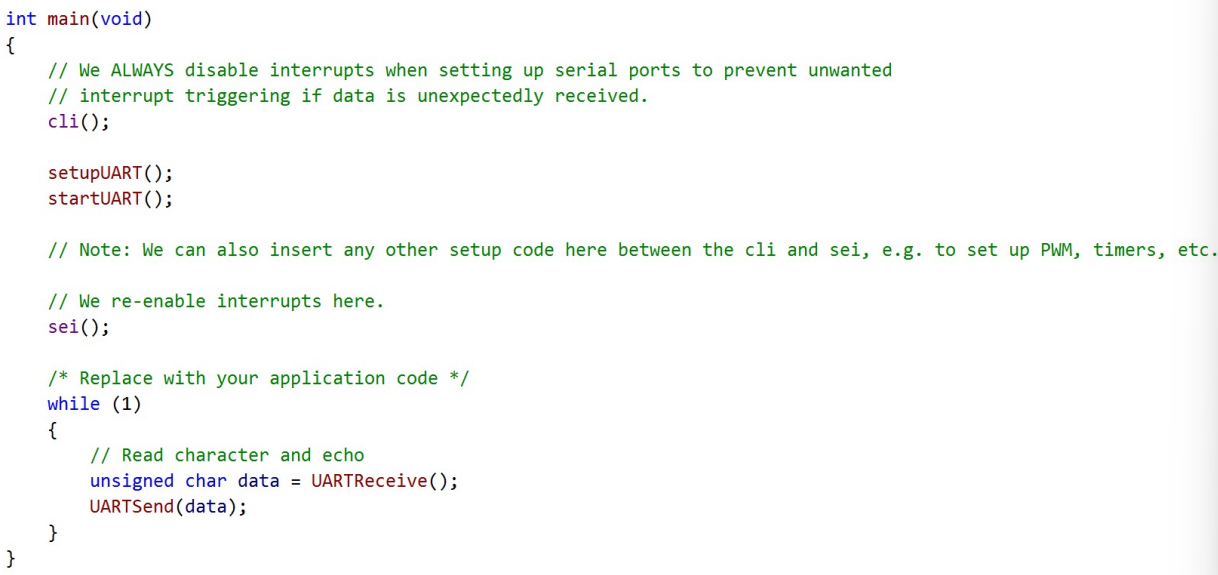
**Note**: can also poll till TXC0 is set, but this is less efficient because TXC0 is not set until every bit of the previous byte has already been sent out.

In reality we only need to wait till UDR0 is empty. The USART can continue sending the previous byte as we write to UDR0.

**Receive:**



**Main:**

****

**Sending/ Receiving in Interrupt Mode**

Similar to the polling mode except

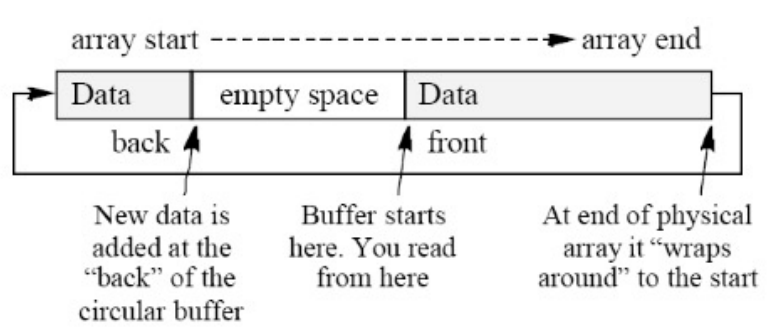
**RXCIE0** (Receive Complete Interrupt Enable) bit **(bit 7) in UCSR0B to 1** and capture the USART\_RX\_vect interrupt

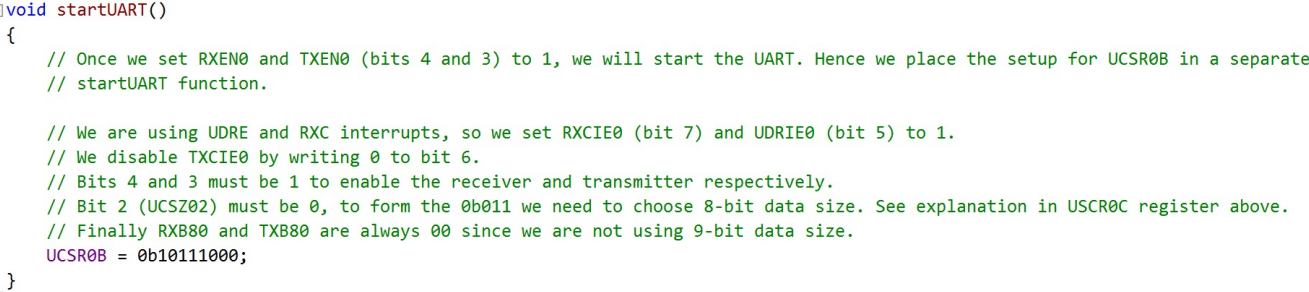
Set either:

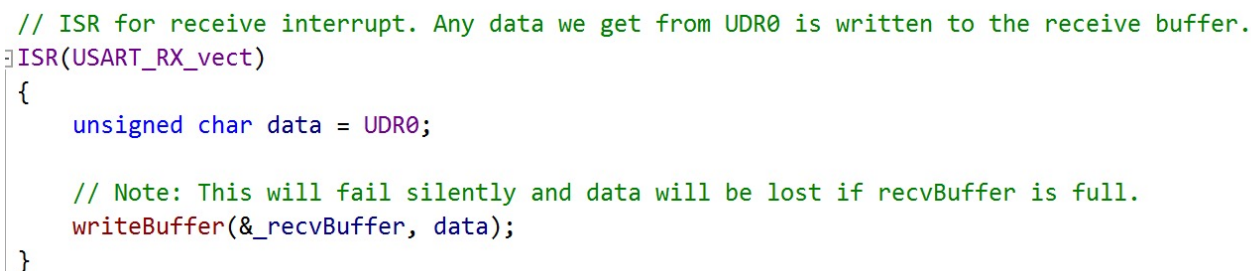
UDRIE0 (USART Data Register Interrupt Enable) **bit (bit 5) in UCSR0B** and capture the **USART\_UDRE\_vect interrupt. (slightly more efficient)**

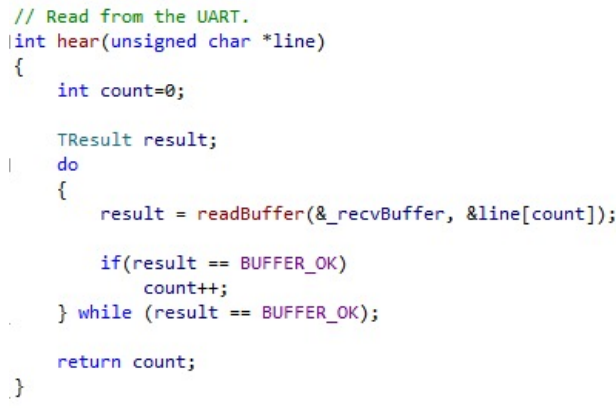
TXCIE0 (Transmit Complete Interrupt Enable) **bit (bit 6) in UCSR0B** and capture the **USART\_TX\_vect interrupt**.

**Circular buffer** for receiving and transmitting for efficiency

****

****

****

****

****

**Communication Protocols**

**Building a protocol**

Assign ID to each device

Create Packet Types (Hello, ACK,NAK,etc)

TCPIP Triple handshake

A Hello→B

B ACK→A

A ACK → B

Getting data from Arduino

Arduino send back data in “heartbeat packet” (regular interval, common)

RPi poll for data (send a request for data, wait for reply, easy)

Periodic push by arduino

Arduino sends data whenever available, RPi monitors and buffers data

Periodic Poll by RPi

Arduino waits for poll packets from RPi (if wait for too long, data may lose due to Arduino small RAM)

Commanding the Arduino

RPi can either poll the arduino continuously to check if finished, OR

The Arduino send back a packet that it is done.

Finding Checksum（attached to end of packet）

used to check is data is received (almost) correctly. If multiple char are sent, there may be corruption due to noise.

*Sender side:*

*Compute checksum: checksum = b1 XOR b2 XOR b3 XOR …*

*Attach to end of packet*

*Receiver side:*

*compute checksum using data other than the checksum*

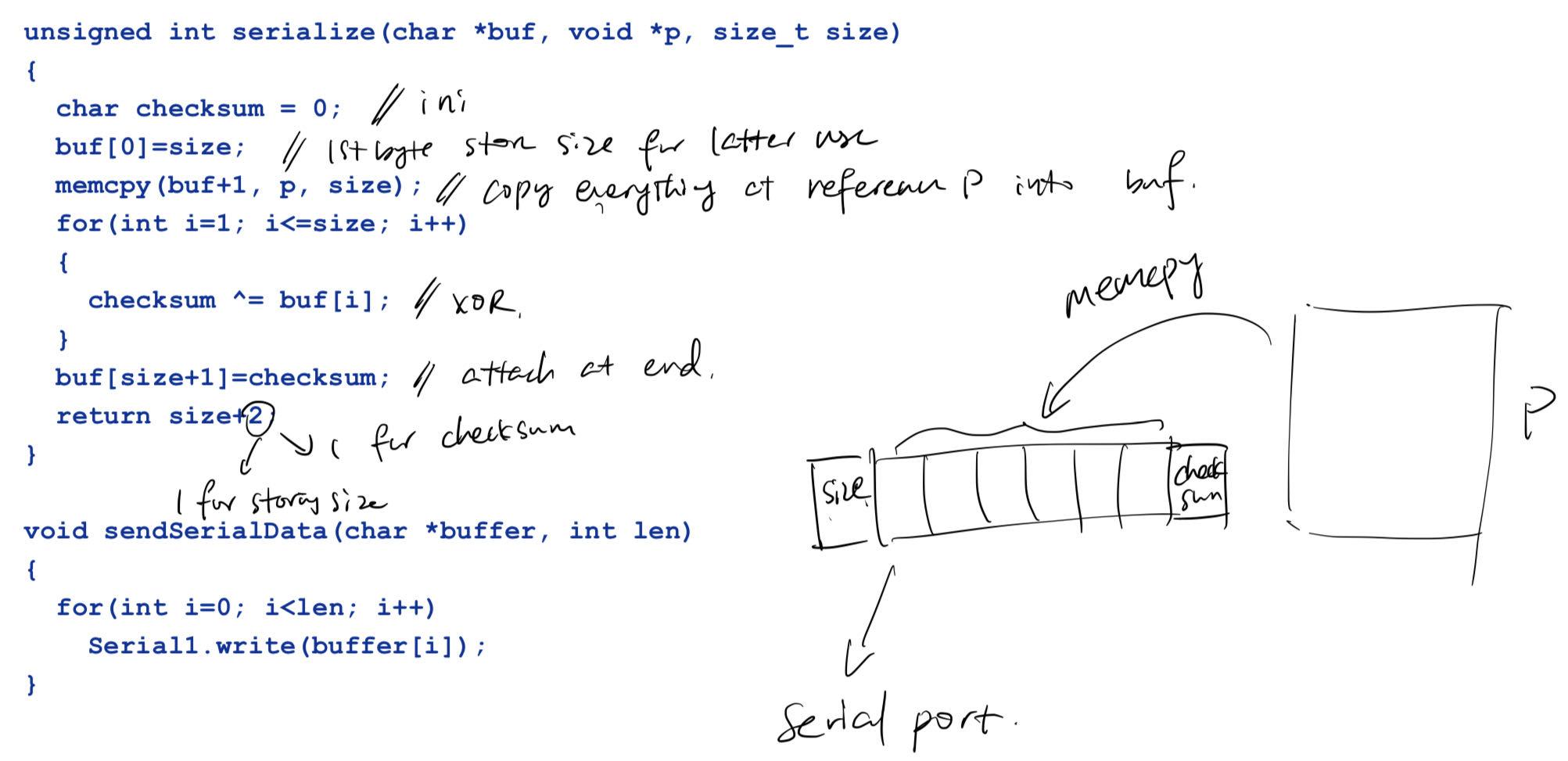
*compare*

*if equal, ACK, else, NAK*

Note: checksum’ != checksum → Error Exists, but the equal checksums does not imply correctness. When there are even number of error on the same bit, the error, the indication cancels out each other.

Serializing structure

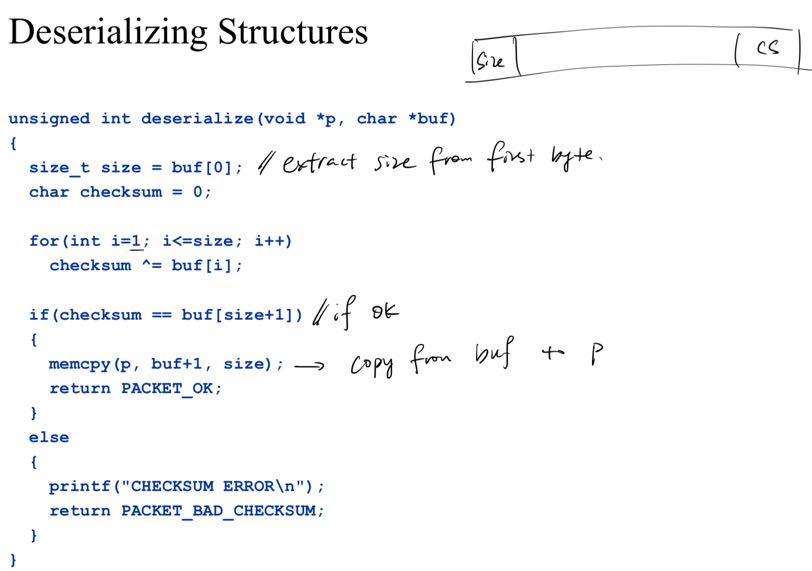
Convert a struct (which is the packet) into a stream of bytes since serial devices can only deal with bytes. In the code below \*p is a ptr to the structure.



Deserializing Structure

Get a ptr to the structure

copy buffer of bytes to that ptr



**Endianness**

Eg. 0x87654321 a four-byte int

Big Endian: 87 65 43 21

Little Endian 21 43 65 87

If different, convert to a standard endianness and convert back to the native endianness at destination.

Both Arduino and RPi are LITTLE Endian

Endianness only affects the way a single variable is stored. A struct is made up of multiple members, each one is subjected to endianness setting (except char), but the struct itself must maintain its members order.

**Different Data Sizes**

use standard integer types

#include<stdint.h>

replace int with int32\_t

unsigned int with uint32\_t

long with int64\_t

unsigned long with unint64\_t

\*Float is not affected

**Bytes may not all arrive in time**

Test number of bytes received

**Codes compile differently on different Architectures**

eg. RPi moves data by 4 bytes (32-bit, since ARM architecture is 32 bit). It handles 4 byte chunk per instruction.

Arduino uses an 8-bit AVR microcontroller so it handles data by 1 byte chunk.

(if receive fragments, Test number of bytes received. If it is == size of data structure, accept. Otherwise buffer and add in subsequent bytes that arrive)

Without any action, a structure containing 1 char, 2 int32\_t in the form cxxxxyyyy sent by Arduino to Pi will be handled as cxxx for char (only c will be interpreted since it is a char, xxx will be skipped, since the movement is by 4 byte chunk), xyyy for the first int32\_t and y for the second int32\_t (taken as LSB, said by Copilot).

Compilers **may/ may not** pad dummies.

To handle this, we pad with dummy bytes to make sure the size of structure is divisible by 4 bytes.

char dummy[3]; // an example

**Magic Number**

Include a magic number to test if corrupted and also to make sure it is the correct file (magic number can have meaning).longer magic number eliminates randomness better because there may be some random data stream exactly the same as the magic number

**Secure Network**

**TCP/IP:** Transport Control Protocol/ Internet Protocol

Physical Layer connects computers to the Internet Service Provider, the ISP to other routers, the routers to each other, etc.

**Physical Layer:** Hardware specification and link layer specification

**Network Layer (IP):**

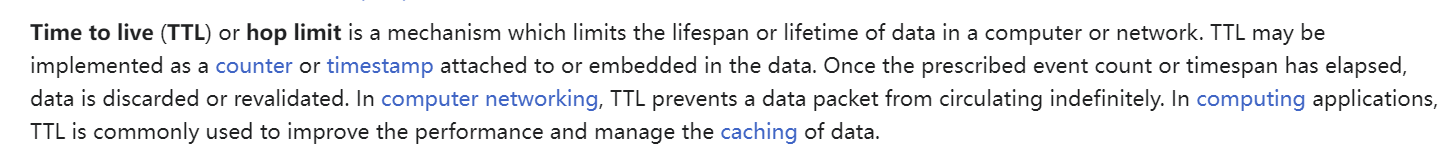
The IP protocol is a standard specification of algorithms and packet format on how to get data from one point to another.

Routers look at the destination address field, and consult internal routing tables to decide which router to forward to.

**The Transport Layer (TCP/UDP)**

Transport Layer deals with the following problem

The IP layer does not guarantee delivery. Packets may be dropped when: Time to Live (TTL) exceeded OR Lack of space in router’s buffers.



(Basic Definition and Use of TTL)

The IP layer does not deal with ports (which are identifiers to which application on the host is to receive the packet).

User Datagram Protocol(UDP)

The simplest TCP protocol:

Best-effort delivery:

Don’t know if a packet is dropped

Quietly drop packets with bad checksum

User data put into data section of UDP, which is in turn put into Data part of IP packet

TCP add in Acknowledgement Request to ensure packet are correctly sent.

Computes checksums and requests for the packet to be re-sent if the packets

corrupted.

Detects dropped packets through time-outs

**TCP/IP: Lack of Security**

Data is not secure:

using packet sniffers, what is being transmitted can be seen.

DNS poisoning or hijacking IP address, can launch man-in-the-middle attacks

**Symmetric Cryptosystems**

same key for encryption and decryption

**Asymmetric Cryptosystems**

public key for encryption

private key for decryption

**Signing messages**

If msg is encrypted using private key, we can also decrypt using the public key.

A can encrypt a verification message (m,c) where m is the plaintext, c is the ciphertext to B.

B manage to decipher c using A’s public key and get m’. If m’ == m, it is definitely true that m was encrypted using A’s private key. The sender of message is confirmed to be A. This is **digital signature.**

For digital signature, since m is also large, h(m) is often sent, where m cannot be derived from h(m). If the receiver manage to get a h’(m), where h’(m) == h(m), it is also sufficient to prove the identity of the sender.

Certificate: the message (pub\_key, h(pub\_key), c) where c is h(pub\_key) encrypted using private key. Let h’(pub\_key) = decrypt(c, pub\_key); if h’(pub\_key) == h(pub\_key), identity confirmed.

**Certificate Authorities**

A and B have a trusted CA.

B send B.cert to CA.

CA sign B.cert using CA.private\_key and send to A.

A decipher using CA.pub\_key and verify B.cert.

**Key Exchange**

**the use of PKC generates large amount of data, not efficient for transmission, key exchange is a hybrid mode so PKC is done once, then both parties communicate using symmetric key**

Diffie-Hellman Key Exchange

**Transport Layer Security (TLS, formerly secure socket layer)**

TLS Handshake

B —— Hello + all symmetric ciphers B understands ——> A

A —— Hello + symmetric cipher to use, SC ——> B

A —— A.cert ——> CA —— CA.private(A.cert) + CA.pub\_key ——> B, B uses CA’s public key to check validity

B —— A.pub(random string) ——> A

Both generate a common session key S, using algo such as Diffie-Hellman. B uses S and selected cipher

B use S and SC ——FINISH——> A

A use S and SC ——FINISH——> B

Handshake completes.

**Additional Info**

Protocol Design:

As shown above, devices can communicate by populating message into a packet structure, serialise into array and send out. However, when communicating with another host on Internet, we populate an array instead of a structure.

There is a wide range of configurations of different host. We **cannot** reliably pad any data structure to ensure Pi understand what client send. However, and **array of char has no ambiguity.**

There are libs containing byte-ordering functions to handle the endianness.

Using Structure is easier to re-assemble the data provided we know how to deal with endianness and padding.

Array works reliably regardless of machine word size.

Representation **structure idea**:

sender side:

memcpy(buffer, &packet, sizeof(TPacket)); // Copy the whole structure into buffer

receiver side:

memcpy(&data[1], packet->data, sizeof(packet->data)); // for data part

Representation of **Array idea**:

sender side

int\_32t temp = htonl(data.x); // change to network layer order

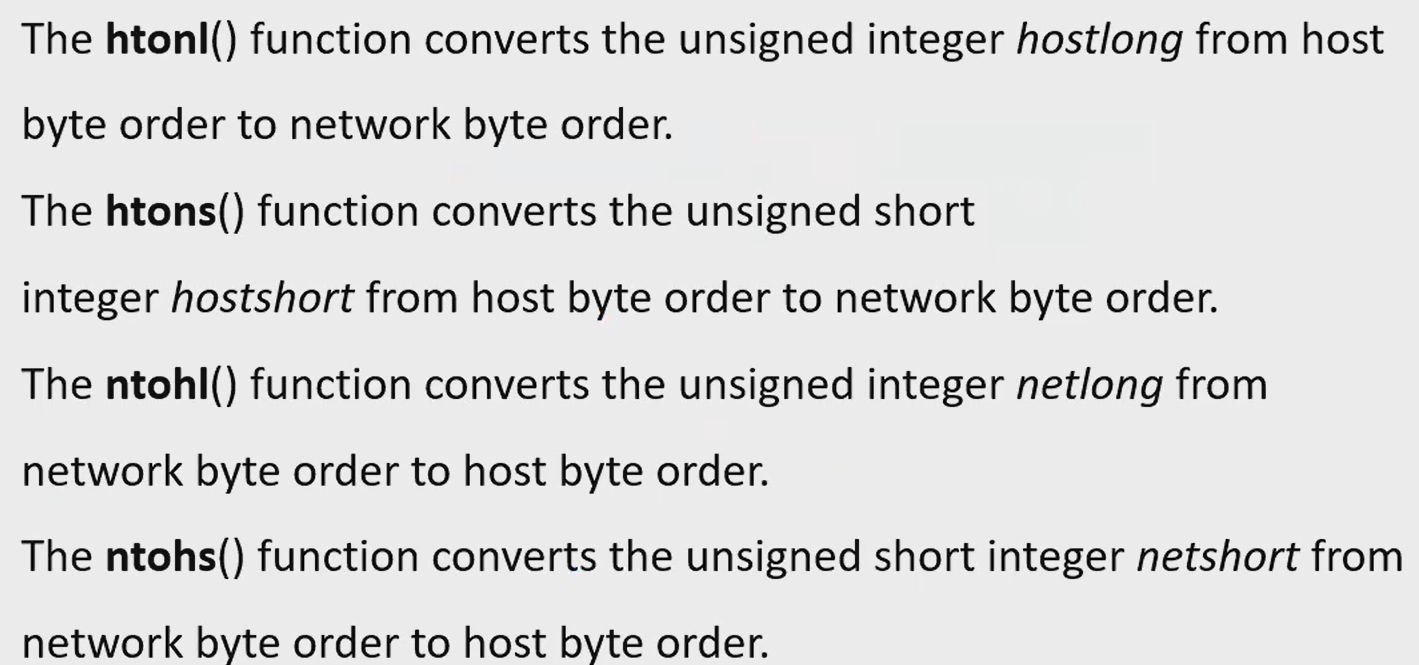
memcpy(buffer, &temp, sizeof(temp)); // data copied one by one

ndx += sizeof(temp); // here ndx is a ptr to keep track of where we are.

receiver side；

int\_32t temp; // assume we want an int\_32t data

memcpy(&temp, buffer(ndx), sizeof(temp)); // copy one by one.

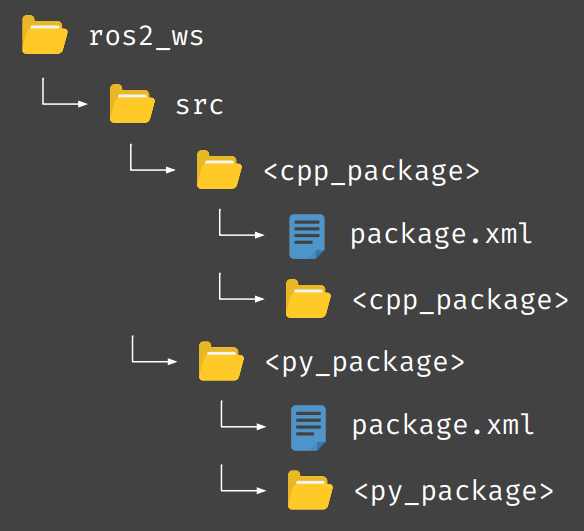


**ROS**

**Node** - Each node is responsible for a single, modular purpose; Each node can send & receive data from other nodes via **topics,** services, actions or parameters.

**Topics** - Publishers publish onto a specific topic. Subscribers listen to a specific topic. All subscribers of the topic can see every message published to it.

**ROS workspace （应该不考）**



**SLAM**

1. **Spike Extraction**
   1. **One simple way to find "interesting" points is to look for abrupt change in the data points. Perform the following in excel:**

**Calculate the 2D distance with the previous point**

**Ignore those points with low quality (anything <= 128) as they usually means bad / error data**

**Find out points with a large enough distance with previous point. This is known as a "spike"**

**Use the X-Y scatter plot and the identified spike points to discuss their significance**

* 1. **The spike points usually indicate the starting / ending point of a landmark. e.g. the edge of a table, the starting point of a wall segment etc. If there are spike points close by, the group of data may indicate a small object in the real world, e.g. a chair, a person, etc.**

1. **RANSAC**
2. **Basic Steps**

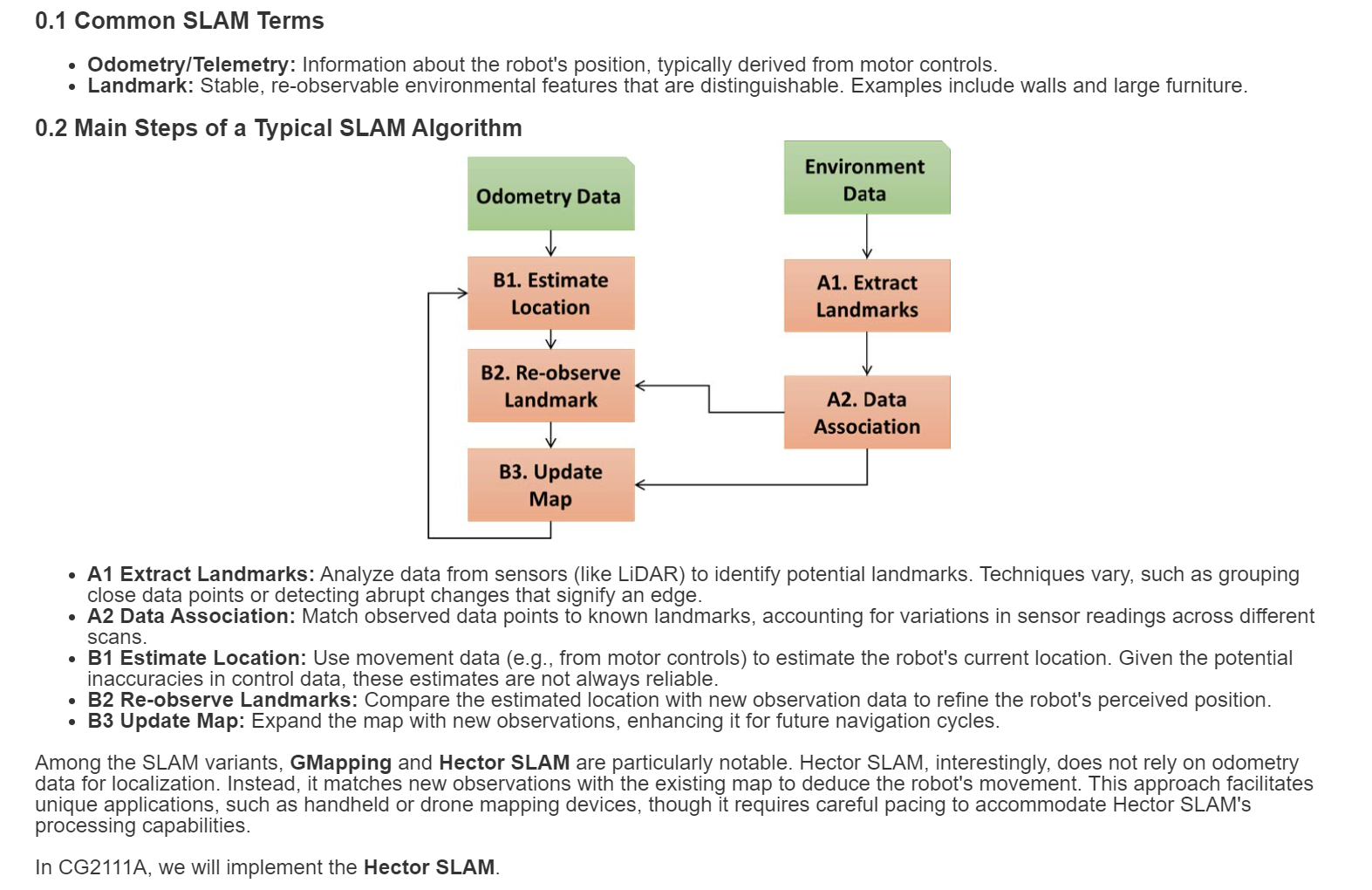
* **Random Sampling: Select two random data points to estimate the parameters of a line.**
* **Consensus Measurement: Calculate the number of data points that lie within a predefined distance from the estimated line.**
* **Iteration and Best Line Selection: Repeat the first two steps for a predefined number of iterations and select the line with the highest consensus as the final line**

1. **RANSAC applied to LIdar Data**

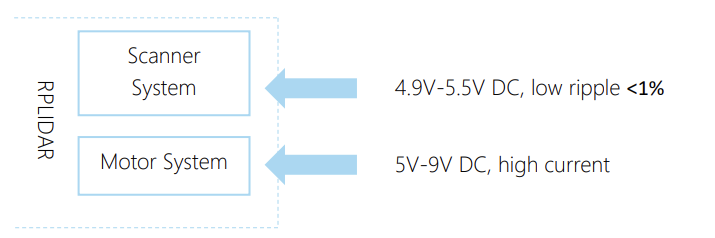
* **While**
* **There are still unassociated LiDAR readings, and the number of readings is larger than the consensus, and we have done less than N trials.**
* **Do:**
* **Select a random laser data reading.**
* **Randomly sample S data readings within D degrees of this LiDar data reading (for example, choose 5 sample readings that lie within 10 degrees of the randomly selected laser data reading).**
* **Using these S samples and the original reading calculate a least squares best fit line.**
* **Determine how many laser data readings lie within X centimetres of this best fit line.**
* **Iteration**

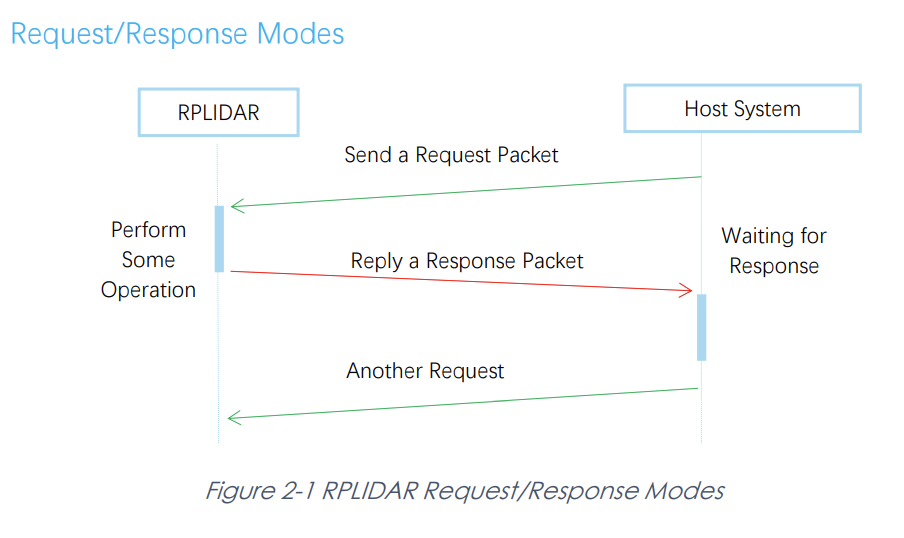
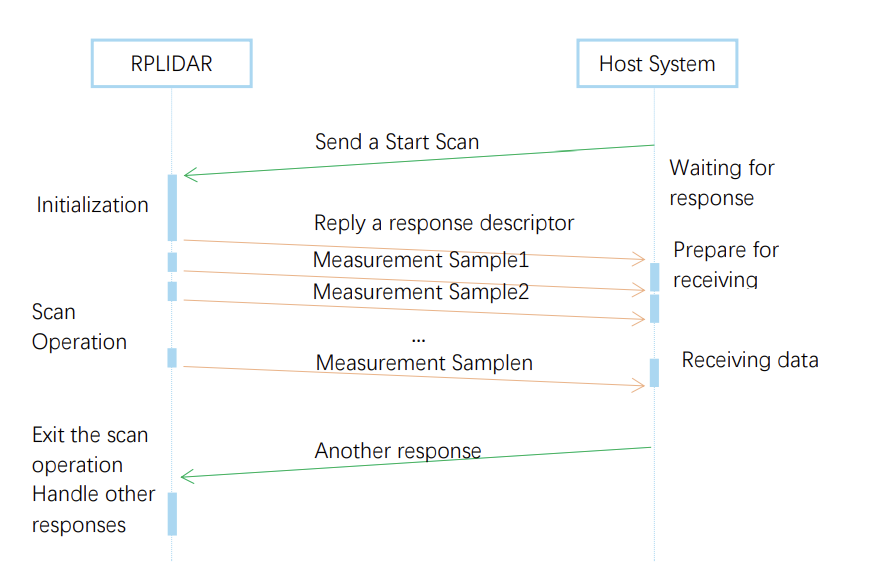
**If the number of laser data readings on the line is above some consensus C do the following:**

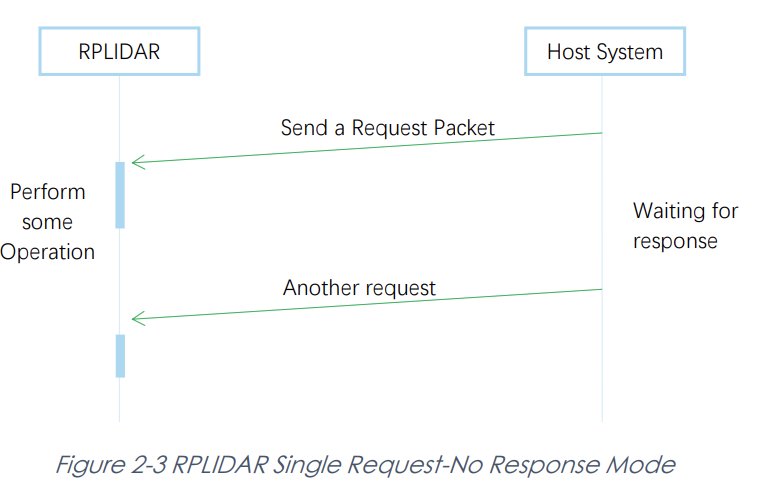
* **Calculate new least squares best fit line based on all the laser readings determined to lie on the old best fit line.**
* **Add this best fit line to the lines we have extracted.**
* **Remove the number of readings lying on the line from the total set of unassociated readings.**

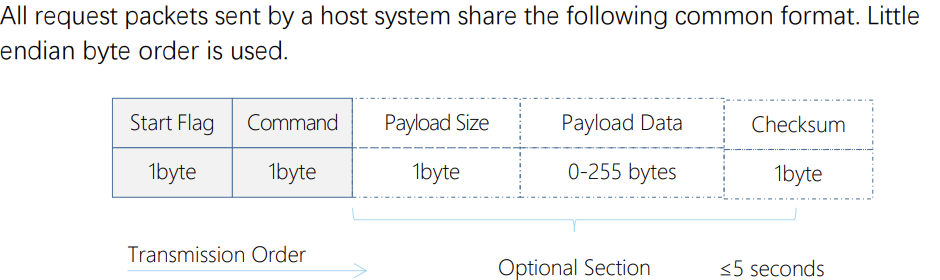


**LIDAR(I believe that most content will be given in the appendix)**

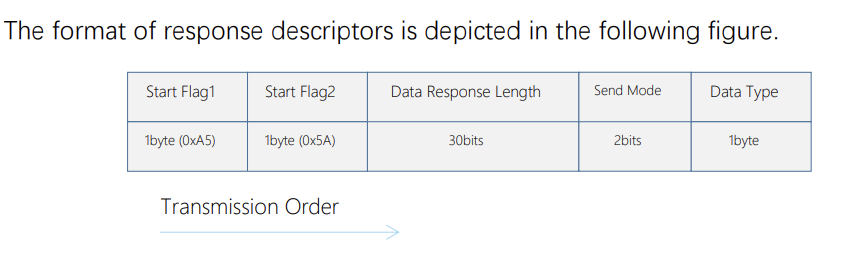
1. **RPLIDAR A1’s scanning frequency reached 5.5hz when sampling 1450 points each round. And it can be configured up to 10hz① maximum.**
2. **power**
3. **The host system communicates with RPLIDAR core system via the TTL UART serial interface**
4. **A communication session is always initialised by a host system, i.e. a MCU, a PC, etc. RPLIDAR itself won’t send any data out automatically after powering up. If a data packet is sent from host systems to RPLIDARs, such a packet is called a Request. Once an RPLIAR receives a request, it will reply the host system with a data packet called a Response. RPLIDAR will only start performing related operations required by a host system once after it receives a request. If RPLIDAR should reply to the host system, it will send one or more required response packets. In order to let an RPLIDAR start scanning operation and send out data, a host system is required to send a pre-defined Start Scan request packet to RPLIDAR. RPLIDAR will start scanning operation once after it receives the request and the scan result data is sent out to the host system continuously.**





1. **Request Packets’ Format**

**A fixed 0xA5 byte is used for each request packet, RPLIDAR uses this byte as the identification of a new request packet. An 8bit (1byte) command field must follow the start flag byte.**

1. **Response Packets’ Format**
2. **RPLIDAR always checks the motor rotation status when working in the scanning state. Only when the motor rotation speed becomes stable, RPLIDAR will start taking distance measurement and sending out the result data to the host system.**
3. **Request Overview(尽力了，困了，剩下的 data sheet会给的，不给拉倒)**

