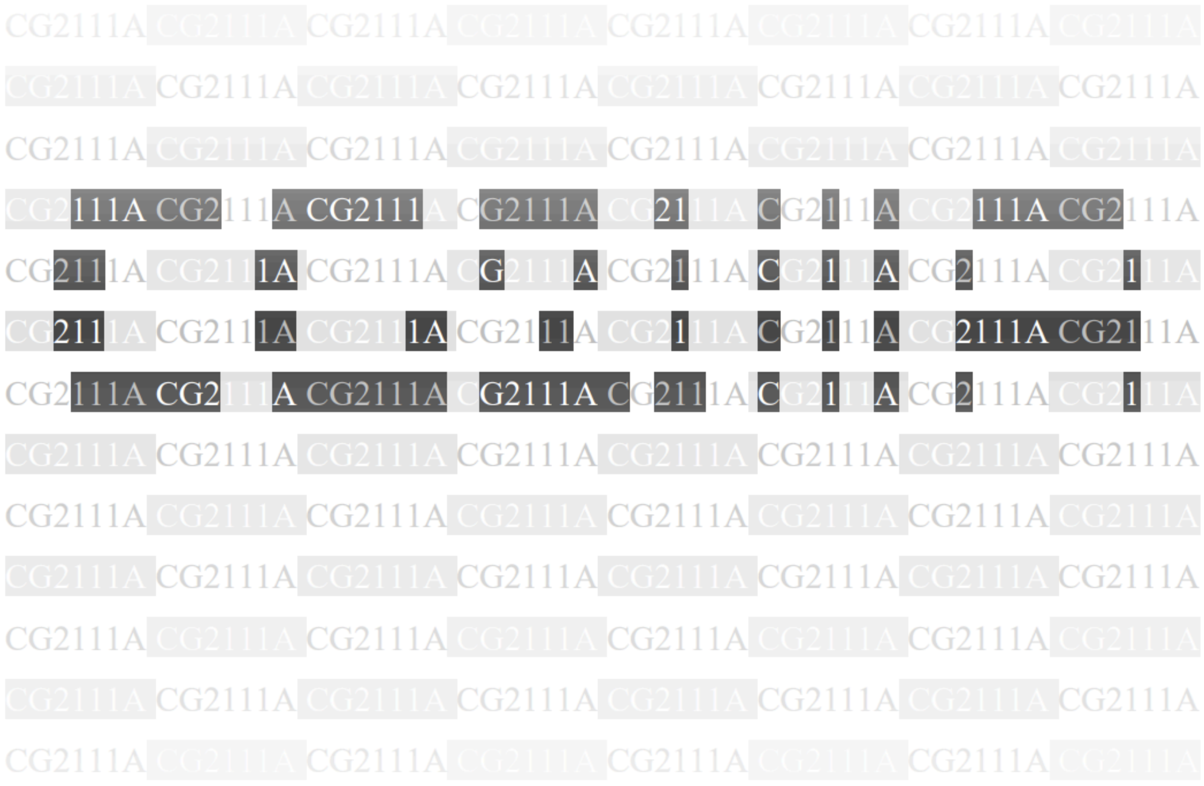
CG2111A Notes 2 —— GYN and friends



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.

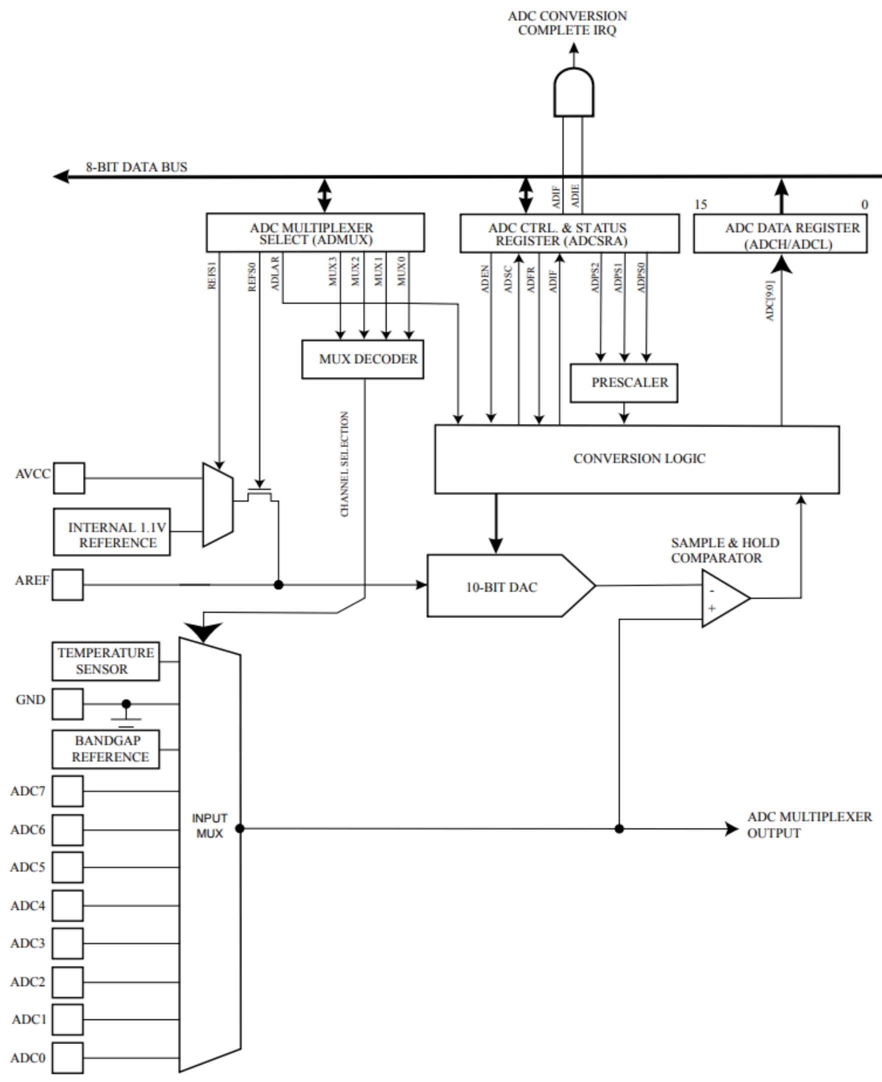
The **Nyquist Sampling Rate:** at least twice as fast as the highest frequency content of the signal so as to accurately represent the signal.
Resolution: voltage distance between two adjacent quantization levels.

A normal conversion takes 13 ADC clock cycles.

Basic Abbreviation we use:

MSB and LSB——Most and Least Significant Bit

Figure 28-1. Analog to Digital Converter Block Schematic Operation



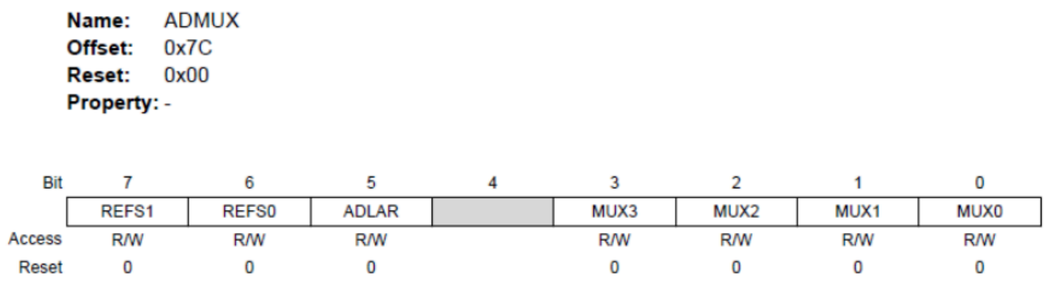
Input channel selected by writing to the MUX bits in ADC Multiplexer Selection Register, ADMUX.MUX[3:0]. Any ADC input pins, as well as GND and a fixed bandgap voltage reference, can be selected as single ended input.

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. 
       Name: ADMUX
       Offset: 0x7C
       Reset: 0x00
       Property: -

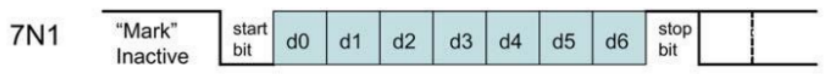
Bit	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR		MUX3	MUX2	MUX1	MUX0
Access	R/W	R/W	R/W		R/W	R/W	R/W	R/W
Reset	0	0	0		0	0	0	0

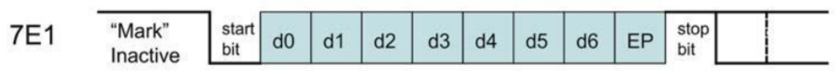
        1. eg. ADMUX = 0b01000010;
4. Start Conversion (ADSC)
    1. ADCSRA |= 0b01000000;
5. Poll, loop till ADSC cleared by hardware
    1. while(ADSRA & 0b010000000);
6. Read Result (if right adjusted, always read ADSL first; If the result is left adjusted and no more than 8-bit precision is required, it 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:

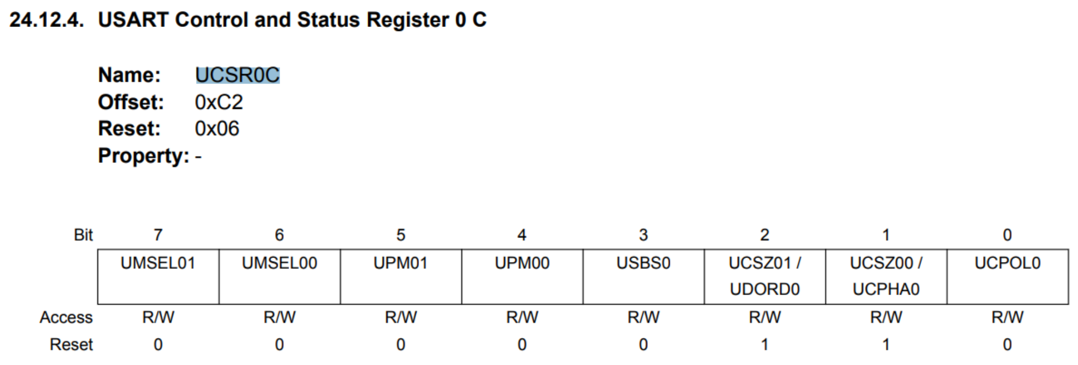
        - Receive (RX): Incoming data bits come here.
        - Transmit (TX): Outgoing data bits go out from here.
        - Ground (GND): Return path for RX and TX. Both receiver and transmitter must share a common ground.
    - Voltage Level
        - 5v for standard devices (e.g. Arduino UNO)
        - 3.3v 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



24.12.4. USART Control and Status Register 0 C

Name: UCSR0C
Offset: 0xC2
Reset: 0x06
Property: -

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	1	1	0

Bit 7:6 Mode

UMSEL0[1:0]	Mode
00	Asynchronous USART
01	Synchronous USART
10	Reserved
11	Master SPI (MSPIM)(1)

Bit 5:4 Parity

UPM0[1:0]	ParityMode
00	Disabled
01	Reserved
10	Enabled, Even Parity
11	Enabled, Odd Parity

Bit 3 Stop Bit

USBS0	Stop Bit(s)
0	1-bit
1	2-bit

Bit 2:1 Character Size and bit 2 of UCSR0B

UCSZ0[2:0]	Character Size
000	5-bit
001	6-bit
010	7-bit
011	8-bit
100	Reserved
101	Reserved
110	Reserved
111	9-bit

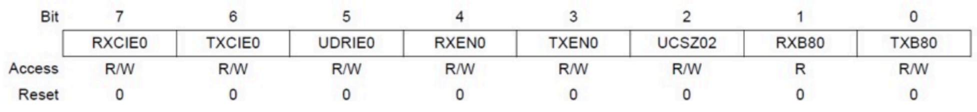
Bit 0 Clock Polarity Setting: 0 for Asynchronous mode

Setting Baud Rate : $B = \frac{fosc}{16 \times baud} - 1$ round to nearest int before subtracting 1

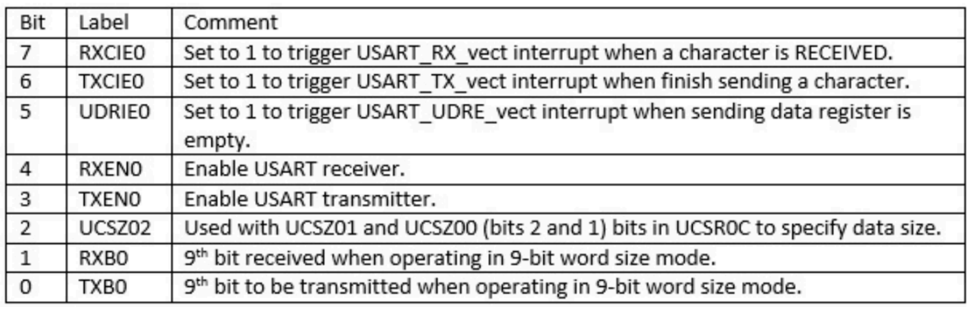
```
void setBaud(unsigned long baudRate)
{
      unsigned int b;
      b = (unsigned int) round(F_CPU / (16.0 *
            baudRate))- 1;
      UBRR0H = (unsigned char) (b >> 8);
      UBRR0L = (unsigned char) b;
}
```

Enable/Disable Features

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

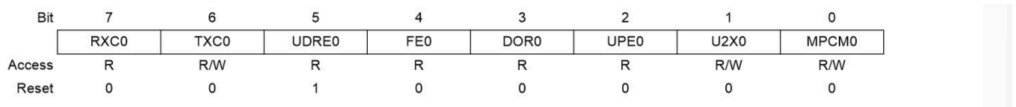


Bit	7	6	5	4	3	2	1	0
	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
Access	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Reset	0	0	0	0	0	0	0	0

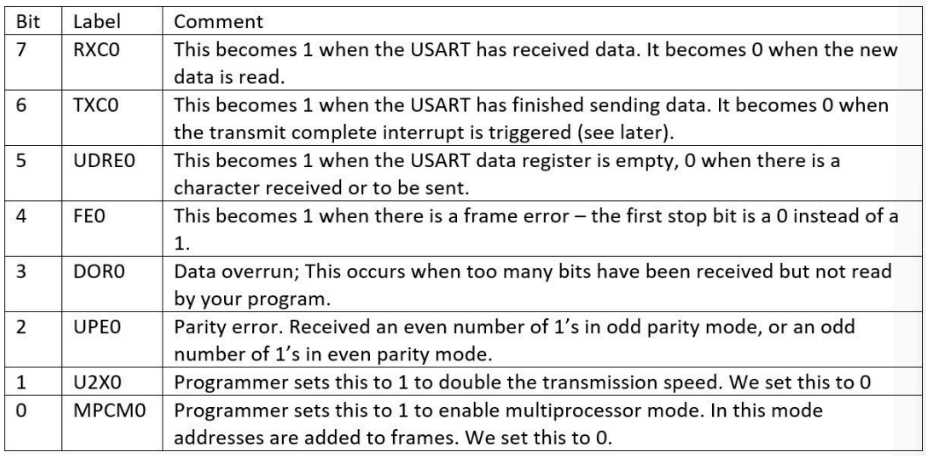


Bit	Label	Comment
7	RXCIE0	Set to 1 to trigger USART_RX_vect interrupt when a character is RECEIVED.
6	TXCIE0	Set to 1 to trigger USART_TX_vect interrupt when finish sending a character.
5	UDRIE0	Set to 1 to trigger USART_UDRE_vect interrupt when sending data register is empty.
4	RXEN0	Enable USART receiver.
3	TXEN0	Enable USART transmitter.
2	UCSZ02	Used with UCSZ01 and UCSZ00 (bits 2 and 1) bits in UCSR0C to specify data size.
1	RXB0	9th bit received when operating in 9-bit word size mode.
0	TXB0	9th bit to be transmitted when operating in 9-bit word size mode.

UCSR0A



Bit	7	6	5	4	3	2	1	0
	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Access	R	R/W	R	R	R	R	R/W	R/W
Reset	0	0	1	0	0	0	0	0



Bit	Label	Comment
7	RXC0	This becomes 1 when the USART has received data. It becomes 0 when the new data is read.
6	TXC0	This becomes 1 when the USART has finished sending data. It becomes 0 when the transmit complete interrupt is triggered (see later).
5	UDRE0	This becomes 1 when the USART data register is empty, 0 when there is a character received or to be sent.
4	FE0	This becomes 1 when there is a frame error – the first stop bit is a 0 instead of a 1.
3	DOR0	Data overrun; This occurs when too many bits have been received but not read by your program.
2	UPE0	Parity error. Received an even number of 1's in odd parity mode, or an odd number of 1's in even parity mode.
1	U2X0	Programmer sets this to 1 to double the transmission speed. We set this to 0
0	MPCM0	Programmer sets this to 1 to enable multiprocessor mode. In this mode addresses are added to frames. We set this to 0.

Sending/ Receiving in Polling Mode

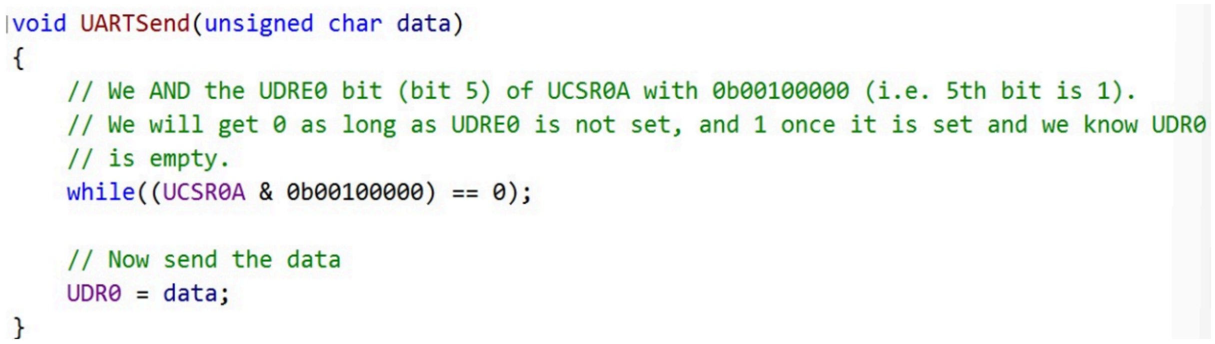
```
// Set up UCSR0C and UBRR0H and UBRR0L…and UCSR0A = 0;
UCSR0B = 0b00011000; // disable all INT, enable receiver and transmitter
```

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).

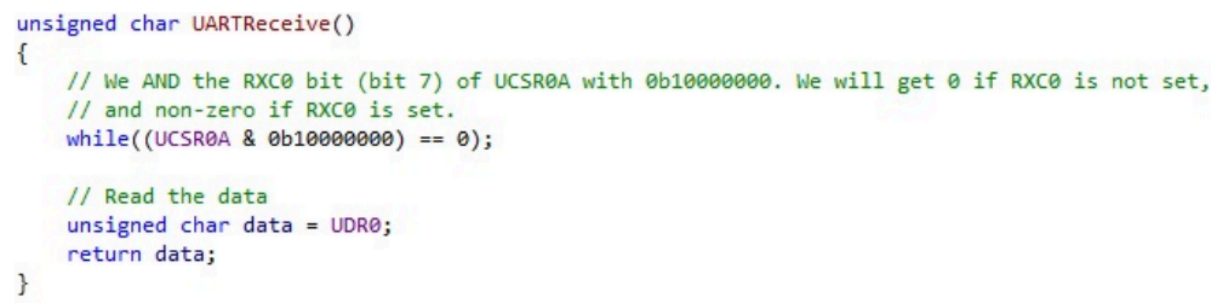


```
void UARTSend(unsigned char data)
{
    // We AND the UDRE0 bit (bit 5) of UCSR0A with 0b00100000 (i.e. 5th bit is 1).
    // We will get 0 as long as UDRE0 is not set, and 1 once it is set and we know UDR0
    // is empty.
    while((UCSR0A & 0b00100000) == 0);

    // Now send the data
    UDR0 = data;
}
```

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

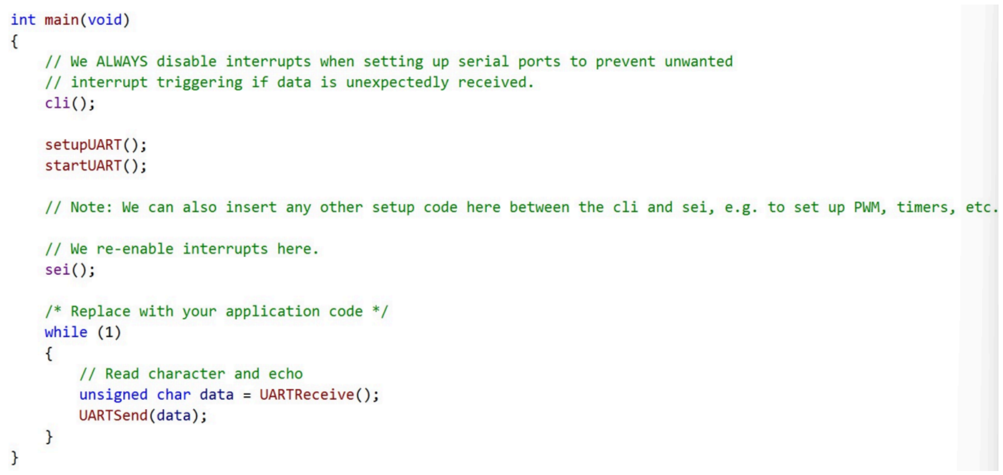
In reality we only need to wait till UDR0 is empty. The USART can continue sending the previous byte as we write to UDR0.

Receive:



```
unsigned char UARTReceive()
{
    // We AND the RXC0 bit (bit 7) of UCSR0A with 0b10000000. We will get 0 if RXC0 is not set,
    // and non-zero if RXC0 is set.
    while((UCSR0A & 0b10000000) == 0);

    // Read the data
    unsigned char data = UDR0;
    return data;
}
```

Main:



```
int main(void)
{
    // We ALWAYS disable interrupts when setting up serial ports to prevent unwanted
    // interrupt triggering if data is unexpectedly received.
    cli();

    setupUART();
    startUART();

    // Note: We can also insert any other setup code here between the cli and sei, e.g. to set up PWM, timers, etc.

    // We re-enable interrupts here.
    sei();

    /* Replace with your application code */
    while (1)
    {
        // Read character and echo
        unsigned char data = UARTReceive();
        UARTSend(data);
    }
}
```

Sending/ Receiving in Interrupt Mode

Similar to the polling mode except

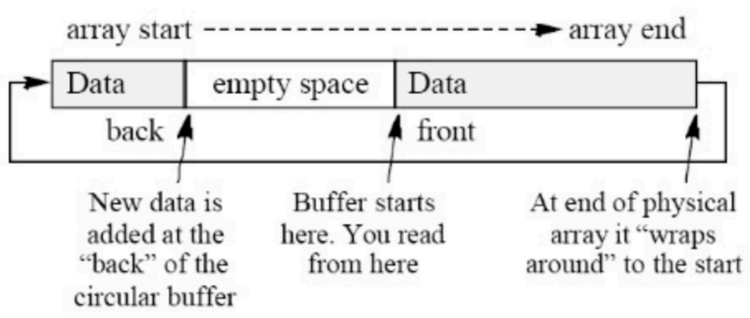
RXCIE0 (Receive Complete Interrupt Enable) bit **(bit 7) in UCSR0B to 1** and capture the USART_RX_vect interrupt

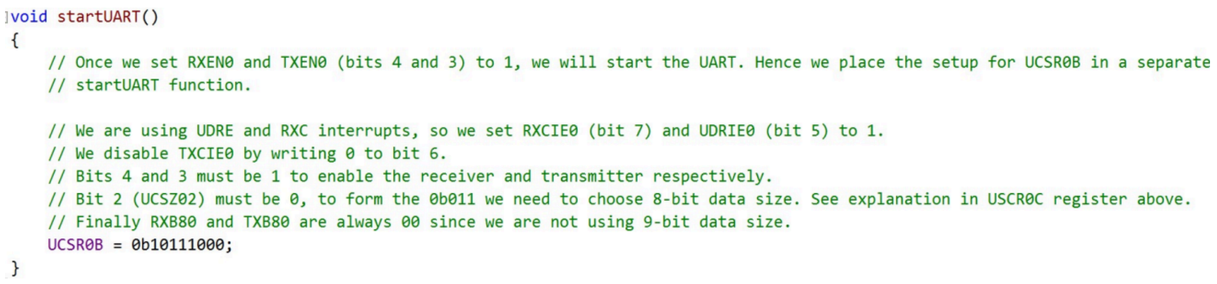
Set either:

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

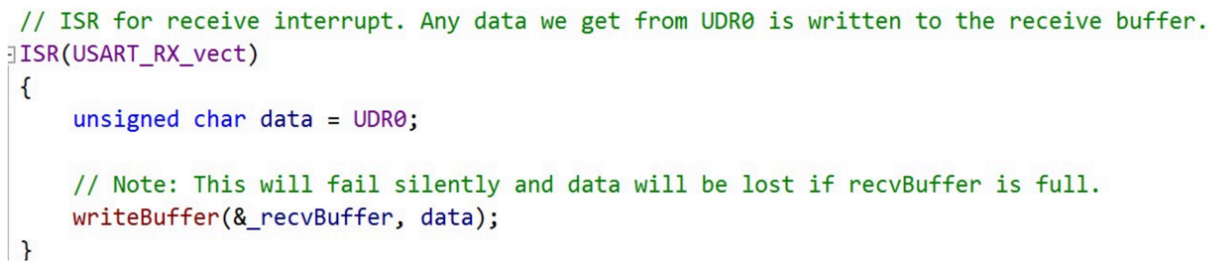
TXCIE0 (Transmit Complete Interrupt Enable) **bit (bit 6) in UCSR0B** and capture the **USART_TX_vect interrupt**.

Circular buffer for receiving and transmitting for efficiency

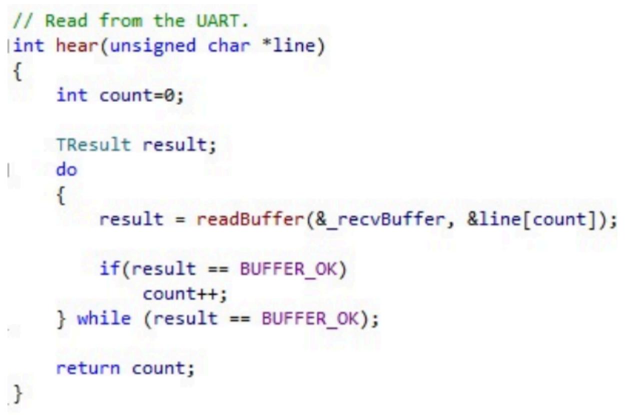

array start
array end
Data
empty space
Data
back
front
New data is added at the "back" of the circular buffer
Buffer starts here. You read from here
At end of physical array it "wraps around" to the start



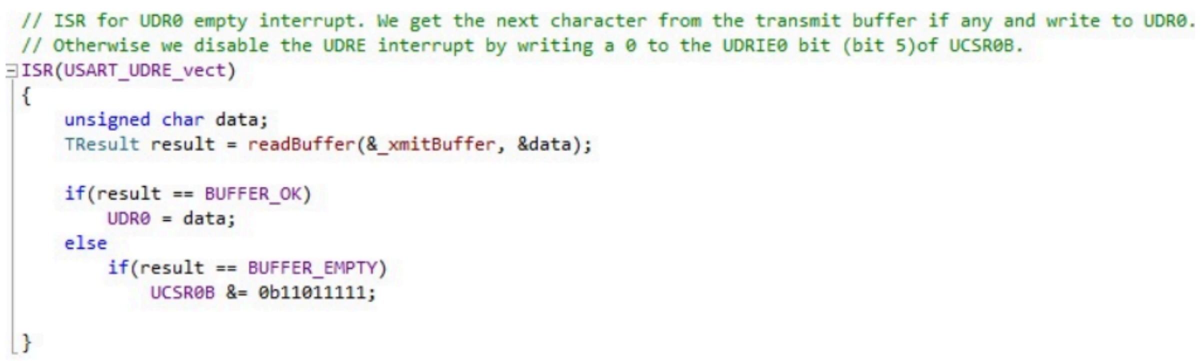
```
void startUART()
{
    // Once we set RXEN0 and TXEN0 (bits 4 and 3) to 1, we will start the UART. Hence we place the setup for UCSR0B in a separate
    // startUART function.

    // We are using UDRE and RXC interrupts, so we set RXCIE0 (bit 7) and UDRIE0 (bit 5) to 1.
    // We disable TXCIE0 by writing 0 to bit 6.
    // Bits 4 and 3 must be 1 to enable the receiver and transmitter respectively.
    // Bit 2 (UCSZ02) must be 0, to form the 0b011 we need to choose 8-bit data size. See explanation in USCR0C register above.
    // Finally RXB80 and TXB80 are always 00 since we are not using 9-bit data size.
    UCSR0B = 0b10111000;
}
```



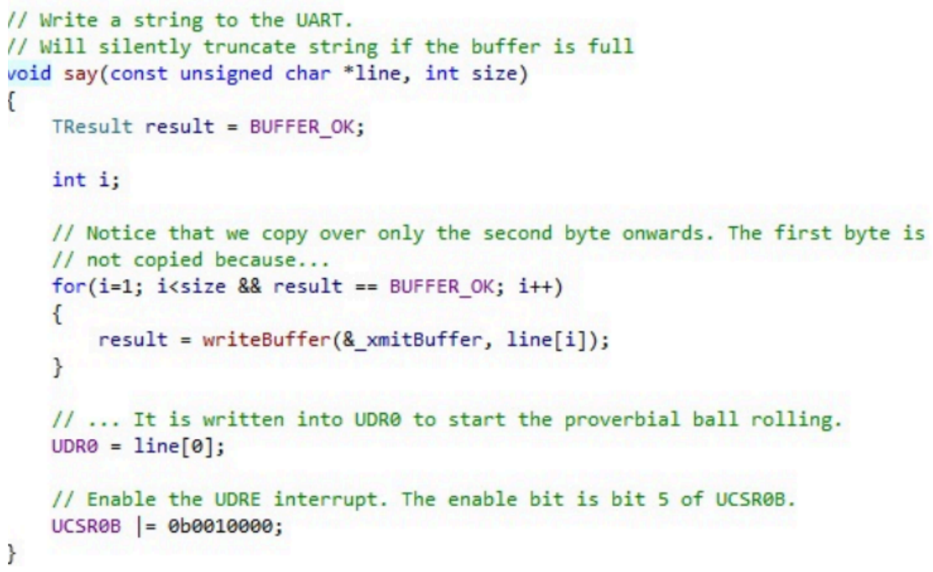
```
// ISR for receive interrupt. Any data we get from UDR0 is written to the receive buffer.
ISR(USART_RX_vect)
{
    unsigned char data = UDR0;

    // Note: This will fail silently and data will be lost if recvBuffer is full.
    writeBuffer(&_recvBuffer, data);
}
```



```
// Read from the UART.
int hear(unsigned char *line)
{
    int count=0;

    TResult result;
    do
    {
        result = readBuffer(&_recvBuffer, &line[count]);

        if(result == BUFFER_OK)
            count++;
    } while (result == BUFFER_OK);

    return count;
}
```



```
// ISR for UDR0 empty interrupt. We get the next character from the transmit buffer if any and write to UDR0.
// Otherwise we disable the UDRE interrupt by writing a 0 to the UDRIE0 bit (bit 5)of UCSR0B.
ISR(USART_UDRE_vect)
{
    unsigned char data;
    TResult result = readBuffer(&_xmitBuffer, &data);

    if(result == BUFFER_OK)
        UDR0 = data;
    else
        if(result == BUFFER_EMPTY)
            UCSR0B &= 0b11011111;

}
```



```
// Write a string to the UART.
// Will silently truncate string if the buffer is full
void say(const unsigned char *line, int size)
{
    TResult result = BUFFER_OK;

    int i;

    // Notice that we copy over only the second byte onwards. The first byte is
    // not copied because...
    for(i=1; i<size && result == BUFFER_OK; i++)
    {
        result = writeBuffer(&_xmitBuffer, line[i]);
    }

    // ... It is written into UDR0 to start the proverbial ball rolling.
    UDR0 = line[0];

    // Enable the UDRE interrupt. The enable bit is bit 5 of UCSR0B.
    UCSR0B |= 0b0010000;
}
```

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.

<u>Finding Checksum（attached to end of packet）</u>

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

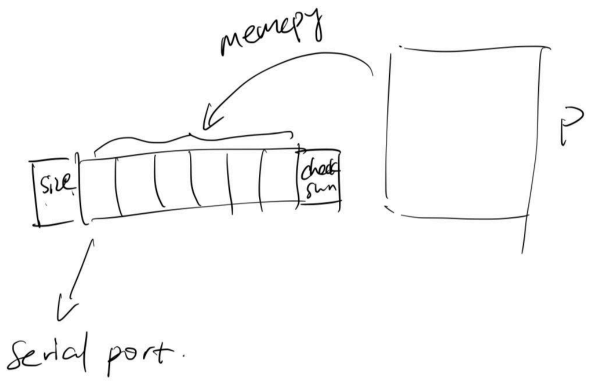
Sender side:
Compute checksum: checksum = b1 XOR b2 XOR b3 XOR …
Attach to end of packet
Receiver side:
compute checksum using data other than the checksum
compare
if equal, ACK, else, NAK

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.

<u>Serializing structure</u>

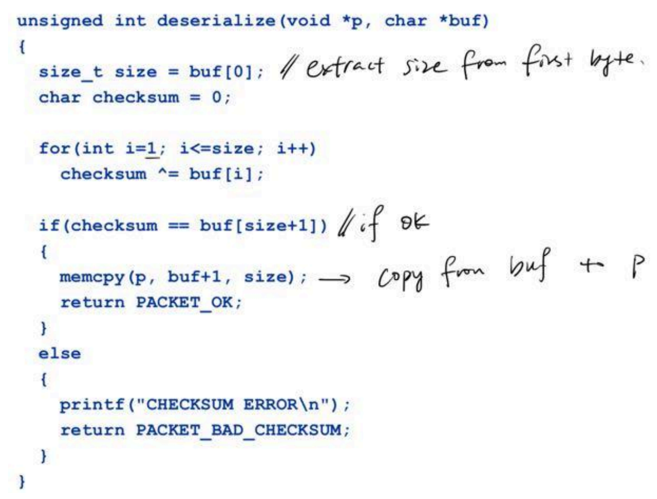
Convert a struct (which is the packet) into a stream of bytes since serial devices can only deal with bytes. In the code below *p is a ptr to the structure.

```
unsigned int serialize(char *buf, void *p, size_t size)
{
  char checksum = 0;  // ini
  buf[0]=size;  // 1st byte store size for latter use
  memcpy(buf+1, p, size); // copy everything at reference P into buf.
  for(int i=1; i<=size; i++)
  {
    checksum ^= buf[i];  // XOR.
  }
  buf[size+1]=checksum;  // attach at end.
  return size+2   ↘ 1 for checksum
          ↙
   1 for storing size
}

void sendSerialData(char *buffer, int len)
{
  for(int i=0; i<len; i++)
    Serial1.write(buffer[i]);
}
```



<u>Deserializing Structure</u>

Get a ptr to the structure
copy buffer of bytes to that ptr



```
unsigned int deserialize(void *p, char *buf)
{
  size_t size = buf[0]; // extract size from first byte.
  char checksum = 0;

  for(int i=1; i<=size; i++)
    checksum ^= buf[i];

  if(checksum == buf[size+1]) // if OK
  {
    memcpy(p, buf+1, size); → copy from buf to p
    return PACKET_OK;
  }
  else
  {
    printf("CHECKSUM ERROR\n");
    return PACKET_BAD_CHECKSUM;
  }
}
```

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

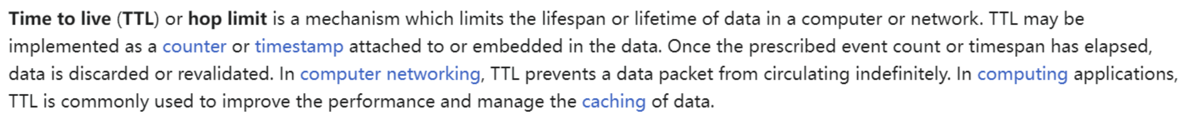
Magic Number
Include a magic number to test if corrupted and also to make sure it is the correct file (magic number can have meaning).longer magic number eliminates randomness better because there may be some random data stream exactly the same as the magic number

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.


Time to live (**TTL**) or **hop limit** is a mechanism which limits the lifespan or lifetime of data in a computer or network. TTL may be implemented as a counter or timestamp attached to or embedded in the data. Once the prescribed event count or timespan has elapsed, data is discarded or revalidated. In computer networking, TTL prevents a data packet from circulating indefinitely. In computing applications, TTL is commonly used to improve the performance and manage the caching of data.

(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

<u>TCP add in Acknowledgement Request</u> 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.

<u>Certificate:</u> 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)
<u>TLS Handshake</u>

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.**

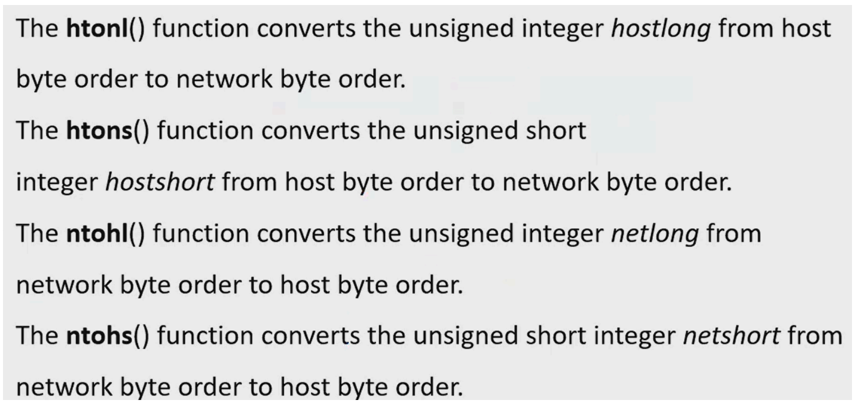
There are libs containing byte-ordering functions to handle the endianness.

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**:
<u>sender side:</u>
memcpy(buffer, &packet, sizeof(TPacket)); // Copy the whole structure into buffer
<u>receiver side:</u>
memcpy(&data[1], packet->data, sizeof(packet->data)); // for data part

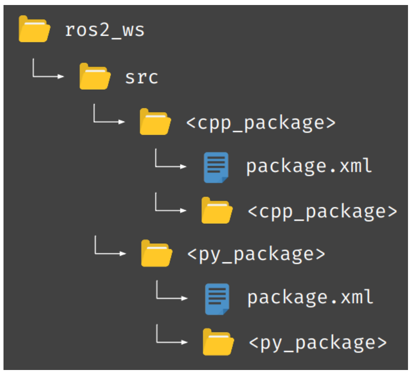
Representation of **Array idea**:
sender side
int_32t temp = htonl(data.x); // change to network layer order
memcpy(buffer, &temp, sizeof(temp)); // data copied one by one
ndx += sizeof(temp); // here ndx is a ptr to keep track of where we are.
receiver side:
int_32t temp; // assume we want an int_32t data
memcpy(&temp, buffer(ndx), sizeof(temp)); // copy one by one.


The **htonl**() function converts the unsigned integer *hostlong* from host byte order to network byte order.
The **htons**() function converts the unsigned short integer *hostshort* from host byte order to network byte order.
The **ntohl**() function converts the unsigned integer *netlong* from network byte order to host byte order.
The **ntohs**() function converts the unsigned short integer *netshort* from network byte order to host byte order.

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）


```
ros2_ws
└→ src
    └→ <cpp_package>
        └→ package.xml
        └→ <cpp_package>
    └→ <py_package>
        └→ package.xml
        └→ <py_package>
```

SLAM

1. **Spike Extraction**
    a. **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

b. 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.

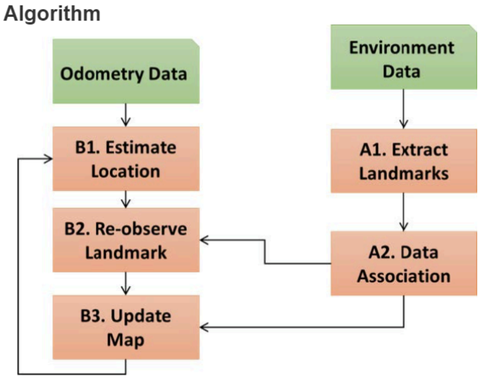
2. RANSAC
    a. Basic Steps
        - Random Sampling: Select two random data points to estimate the parameters of a line.
        - Consensus Measurement: Calculate the number of data points that lie within a predefined distance from the estimated line.
        - Iteration and Best Line Selection: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

0.1 Common SLAM Terms

- **Odometry/Telemetry:** Information about the robot's position, typically derived from motor controls.
- **Landmark:** Stable, re-observable environmental features that are distinguishable. Examples include walls and large furniture.

0.2 Main Steps of a Typical SLAM Algorithm



- **A1 Extract Landmarks:** Analyze data from sensors (like LiDAR) to identify potential landmarks. Techniques vary, such as grouping close data points or detecting abrupt changes that signify an edge.
- **A2 Data Association:** Match observed data points to known landmarks, accounting for variations in sensor readings across different scans.
- **B1 Estimate Location:** Use movement data (e.g., from motor controls) to estimate the robot's current location. Given the potential inaccuracies in control data, these estimates are not always reliable.
- **B2 Re-observe Landmarks:** Compare the estimated location with new observation data to refine the robot's perceived position.
- **B3 Update Map:** Expand the map with new observations, enhancing it for future navigation cycles.

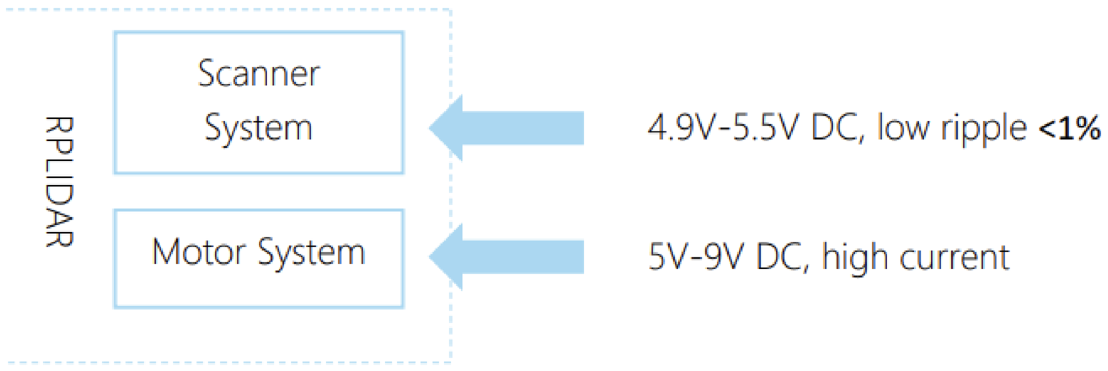
Among the SLAM variants, **GMapping** and **Hector SLAM** are particularly notable. Hector SLAM, interestingly, does not rely on odometry data for localization. Instead, it matches new observations with the existing map to deduce the robot's movement. This approach facilitates unique applications, such as handheld or drone mapping devices, though it requires careful pacing to accommodate Hector SLAM's processing capabilities.

In CG2111A, we will implement the **Hector SLAM**.

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.**

Request/Response Modes

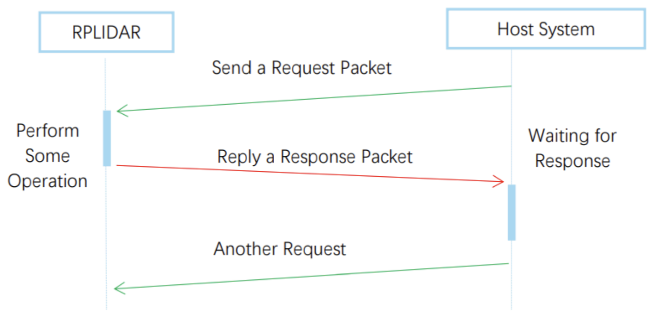
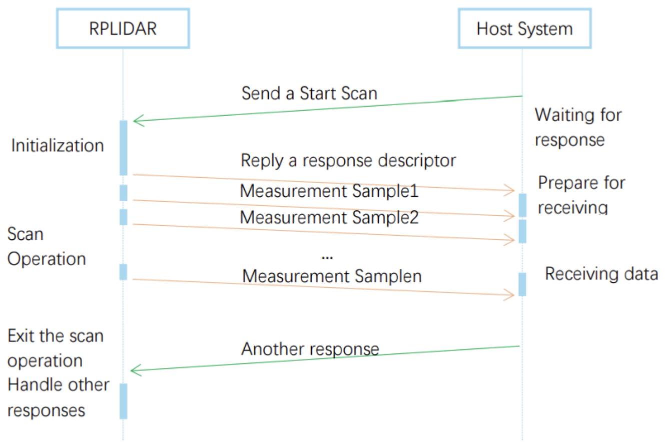


Figure 2-1 RPLIDAR Request/Response Modes



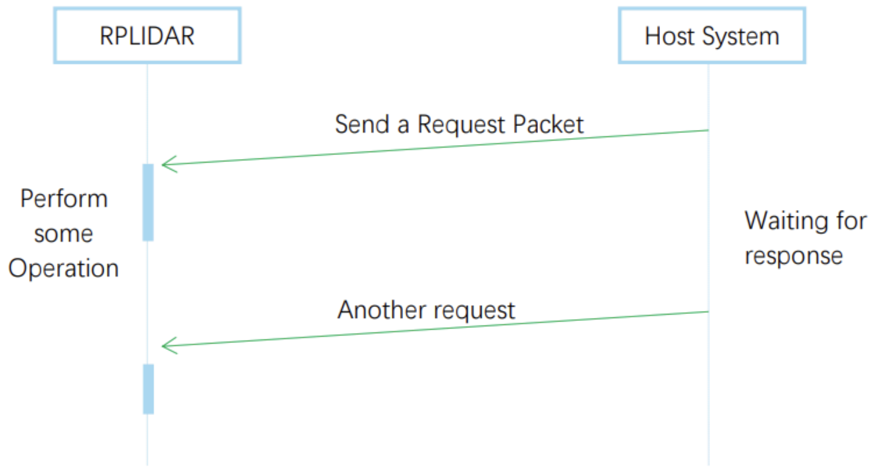
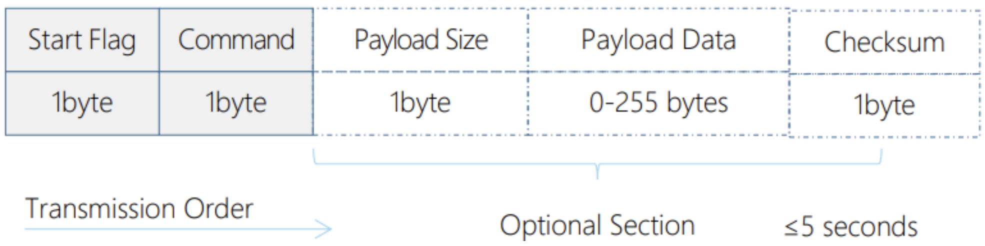


Figure 2-3 RPLIDAR Single Request-No Response Mode

5. **Request Packets’ Format**

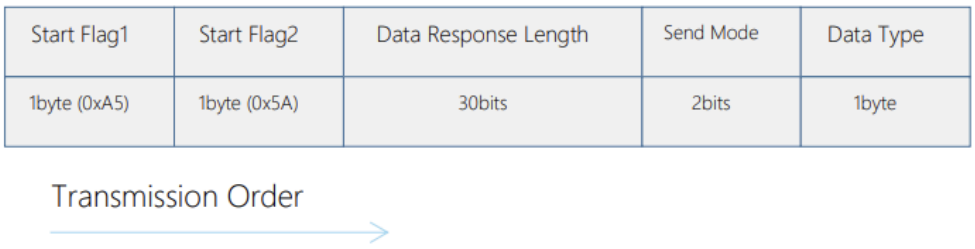
   All request packets sent by a host system share the following common format. Little endian byte order is used.



   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.

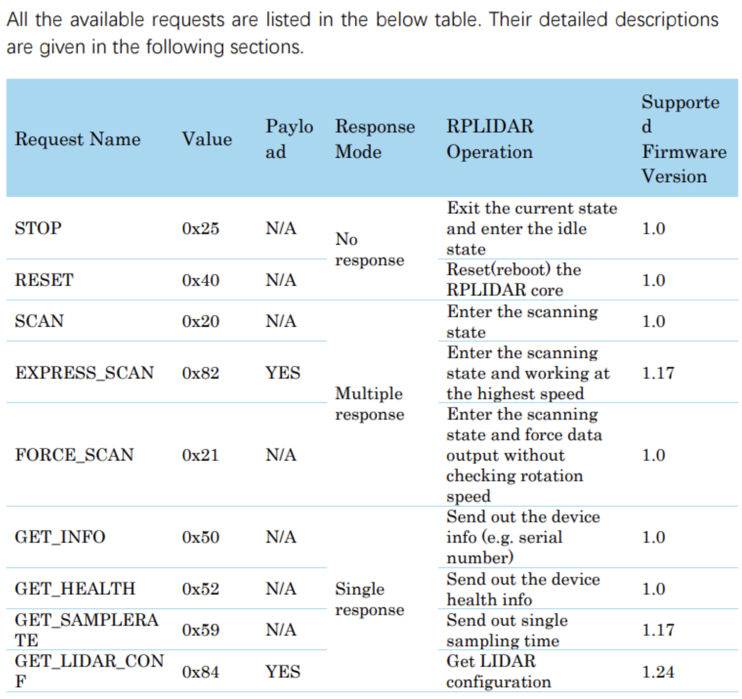
6. **Response Packets’ Format**

   The format of response descriptors is depicted in the following figure.



7. **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.**

8. **Request Overview(尽力了，困了，剩下的 data sheet会给的，不给拉倒)**



All the available requests are listed in the below table. Their detailed descriptions are given in the following sections.

Request Name	Value	Payload	Response Mode	RPLIDAR Operation	Supported Firmware Version
STOP	0x25	N/A	No response	Exit the current state and enter the idle state	1.0
RESET	0x40	N/A		Reset(reboot) the RPLIDAR core	1.0
SCAN	0x20	N/A	Multiple response	Enter the scanning state	1.0
EXPRESS_SCAN	0x82	YES		Enter the scanning state and working at the highest speed	1.17
FORCE_SCAN	0x21	N/A		Enter the scanning state and force data output without checking rotation speed	1.0
GET_INFO	0x50	N/A	Single response	Send out the device info (e.g. serial number)	1.0
GET_HEALTH	0x52	N/A		Send out the device health info	1.0
GET_SAMPLERATE	0x59	N/A		Send out single sampling time	1.17
GET_LIDAR_CONF	0x84	YES		Get LIDAR configuration	1.24