1. EPROM separate programs using read (), write (), get (), put (), update ()

Read ()

#include < EEPROM.h>

int a = 0;

int value;

void setup()

{

Serial.begin(9600);

}

void loop()

{

value = EEPROM.read(a);

Serial.print(a);

Serial.print("\t");

Serial.print(value);

Serial.println();

a = a + 1;

if (a == 512)

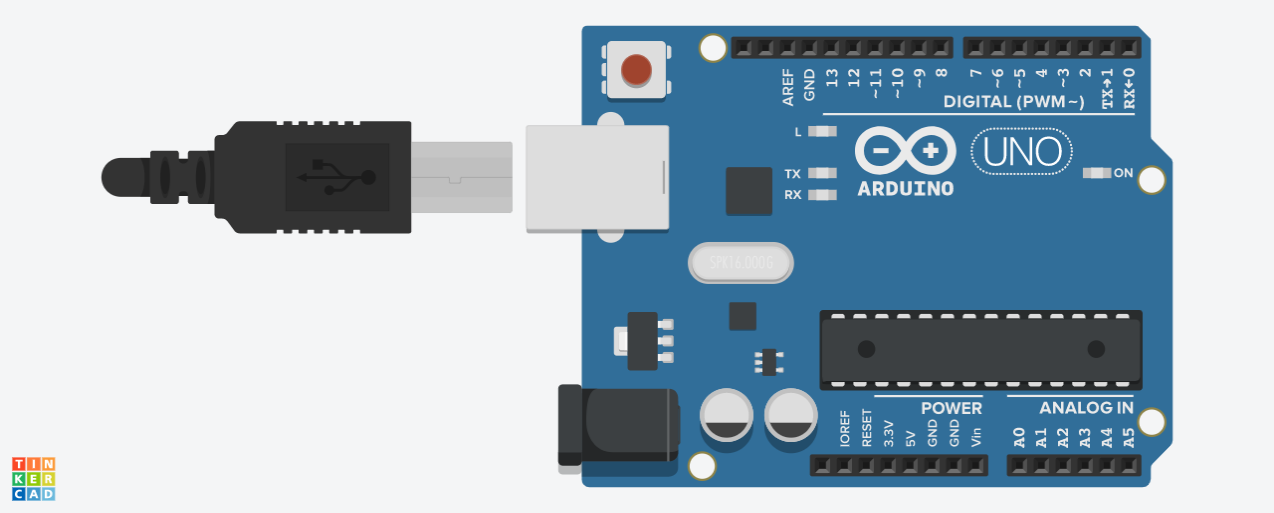
{

a = 0;

}

delay(500);

}



Write ()

#include <EEPROM.h>

void setup()

{

for (int i = 0; i < 255; i++)

{

EEPROM.write(i, i);

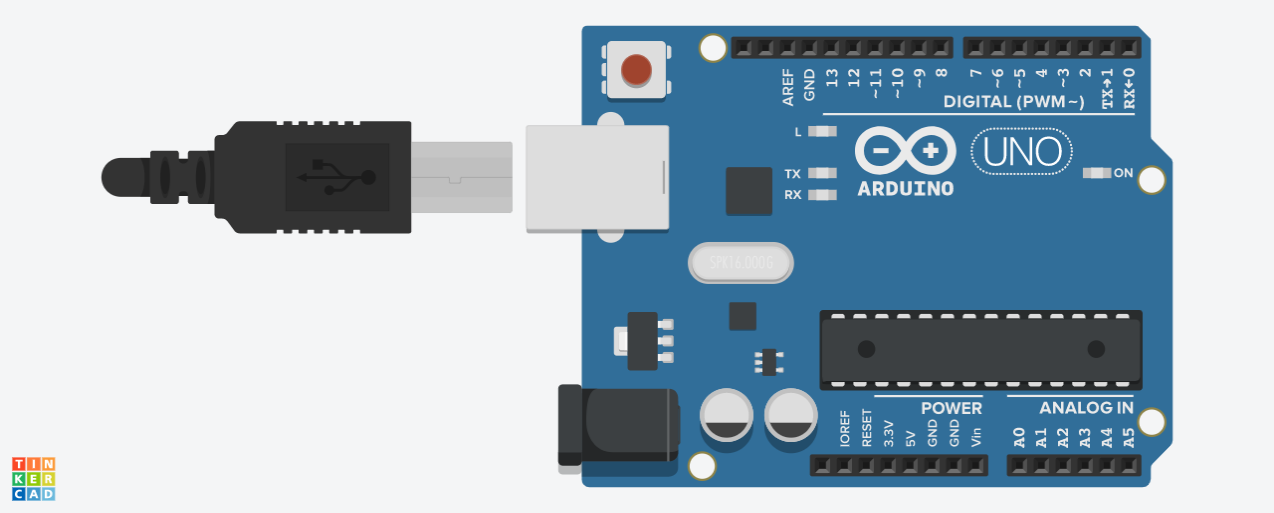
}

}

void loop()

{

}



Update ()

#include <EEPROM.h>

void setup()

{

for (int i = 0; i < 255; i++)

{

EEPROM.update(i, i);

}

for (int i = 0; i < 255; i++)

{

EEPROM.update(3, 12);

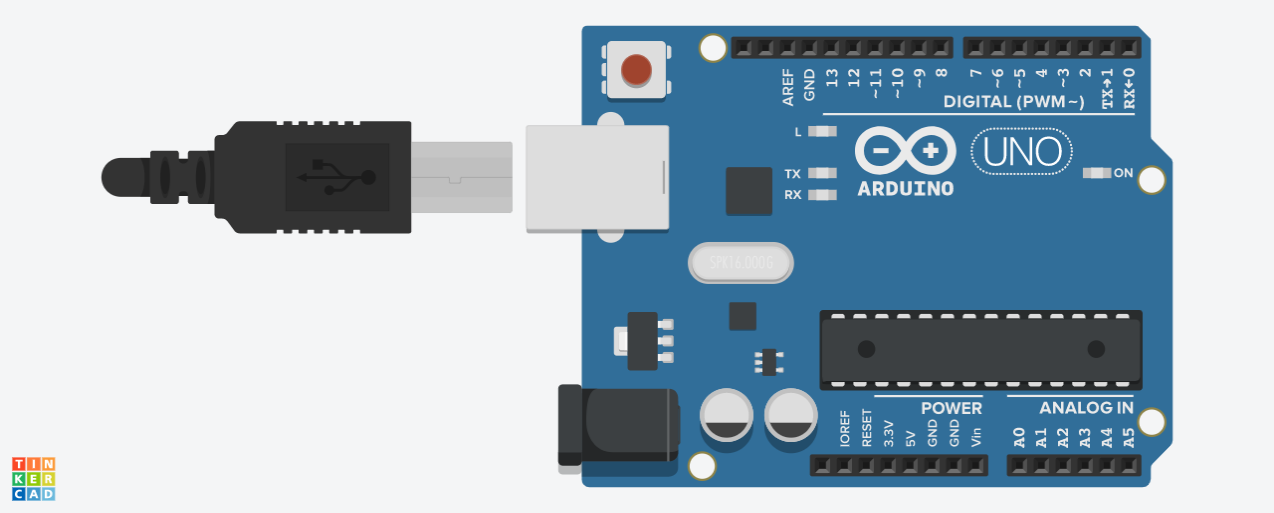
}

}

void loop()

{

}



Put()

#include <EEPROM.h>

struct MyObject

{

float field1;

byte field2;

char name[10];

};

void setup()

{

Serial.begin(9600);

while (!Serial)

{

;

}

float f = 123.456f;

int eeAddress = 0;

EEPROM.put(eeAddress, f);

Serial.println("Written float data type!");

MyObject customVar =

{

3.14f,

65,

"Working!"

};

eeAddress += sizeof(float);

EEPROM.put(eeAddress, customVar);

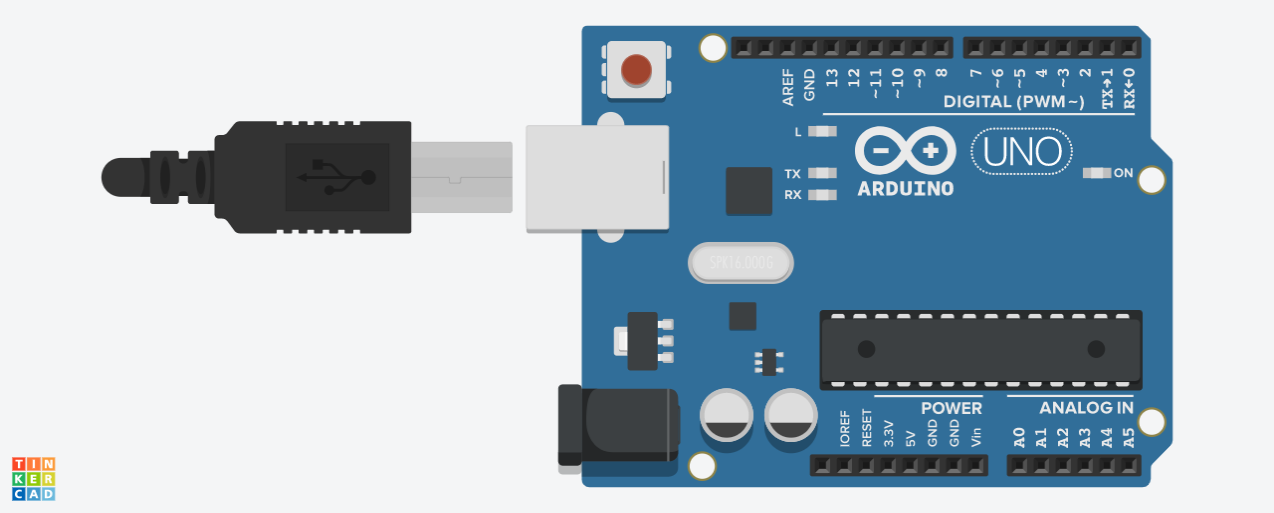
Serial.print("Written custom data type! \n\nView the example sketch eeprom\_get to see how you can retrieve the values!");

}

void loop()

{

}



get()

#include <EEPROM.h>

struct MyObject

{

float field1;

byte field2;

char name[10];

};

void setup()

{

float f = 0.00f;

int eeAddress = 0;

Serial.begin( 9600 );

while (!Serial)

{

;

}

Serial.print( "Read float from EEPROM: " );

EEPROM.get( eeAddress, f );

Serial.println( f, 3 );

eeAddress = sizeof(float);

MyObject customVar;

EEPROM.get( eeAddress, customVar );

Serial.println("Read custom object from EEPROM: " );

Serial.println( customVar.field1 );

Serial.println( customVar.field2 );

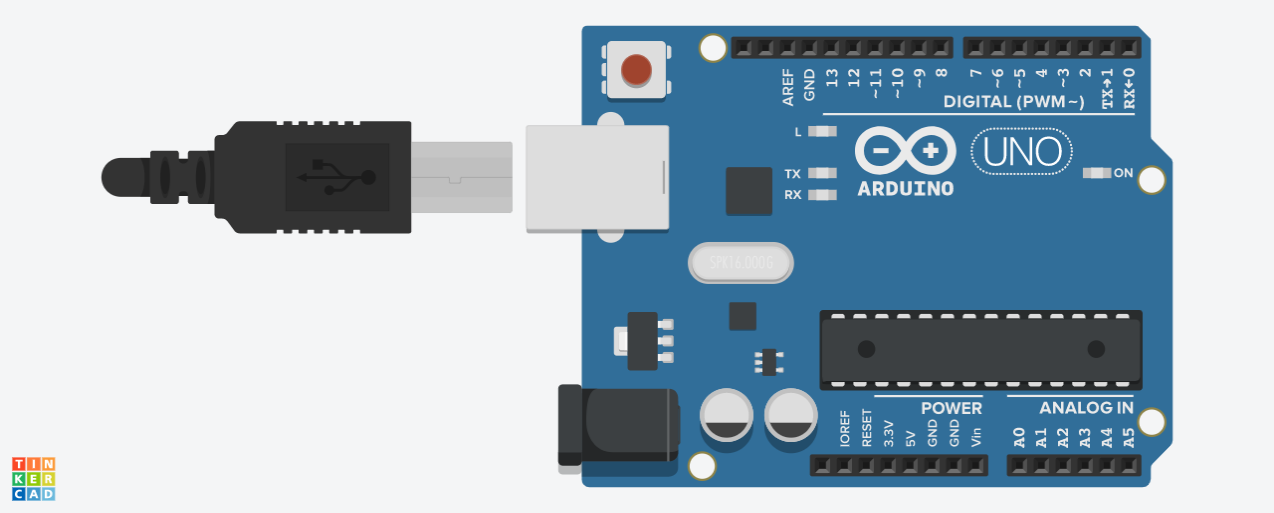
Serial.println( customVar.name );

}

void loop()

{

}



1. UART - sending and receiving messages

Or

UART study the protocol and list inbuilt functions for sending and receiving data

**Protocol**: The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end. UART transmitted data is organized into packets. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional parity bit, and 1 or 2 stop bits:

**Start bit:** To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

**Data frame:** The data frame contains the actual data being transferred. It can be 5 bits up to 8 bits long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long.

**Parity:** Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

**Stop bits:** To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

The inbuilt functions for UART are as follows:

1. To use UART serial in the Arduino IDE, we will have to initialize the serial module.

Serial.begin(9600); // Start the serial module with a baud rate of 9600 bps and the default configuration

1. To write using the UART serial we use the write command. It sends only one byte of data at a time. A single value of 0-255.

Serial.write(70); // transmits the value of 70

1. To print the strings line after line we can use print command.

char array1[5] = {4, 8, 16, 23, 42};

Serial.print(array1); // Will transmit the values 4, 8, 16, 23, and 42

1. To get first byte of the serial buffer

int receivedByte;

receivedByte = Serial.read(); // Returns the first byte from the serial buffer

1. To get return the number of bytes that are currently available in the serial buffer.

if (Serial.available() > 0)

{

receivedByte = Serial.read();

}

1. To clear all bytes from the Serial buffer

Serial.flush();

1. To return the first byte in the serial buffer but does not remove it.

Serial.peak();

1. I2C - sending and receiving messages

Or

I2C study the protocol and list inbuilt functions for sending and receiving data

**Protocol**: With I2C, data is transferred in messages. Messages are broken up into frames of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:

**Start Condition**: The SDA line switches from a high voltage level to a low voltage level before the SCL line switches from high to low.

**Stop Condition**: The SDA line switches from a low voltage level to a high voltage level after the SCL line switches from low to high.

**Address Frame**: A 7- or 10-bits sequence unique to each slave that identifies the slave when the master wants to talk to it.

**Read/Write Bit**: A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

**ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

The inbuilt functions used for I2C are.

1. It is used to write (transmit) data to the master or slave device.

Wire.write('hello'); //sends the string to the slave device

This function is used by a master or slave to check the requested data is available or not. It returns the no. of bytes available

Wire.available();

1. It is used to read the requested data by master from slave or read the data transmitted from a master to a slave.

Wire.read();

1. It initiates the Wire library and joins the bus as a master.

Wire.begin();

This function begins a transmission with the I2C slave device having specified slave address.

Wire.beginTransmission (slave address);

1. It initiates the Wire library and joins the I2C bus as a slave with specified address

Wire.begin(address);

1. The handler functions to be called when a slave device receives a transmitted data from a master.

Wire.onReceive(handler)