DAILY ASSESSMENT

Date:	12-6-2020	**Name:**	Kavyashree m
Course:	VLSI	**USN:**	4al15ec036
Topic:	CMOS Inverter Basics	**Semester & Section:**	8th A
Github Repository :	**kavya**		

FORENOON SESSION DETAILS

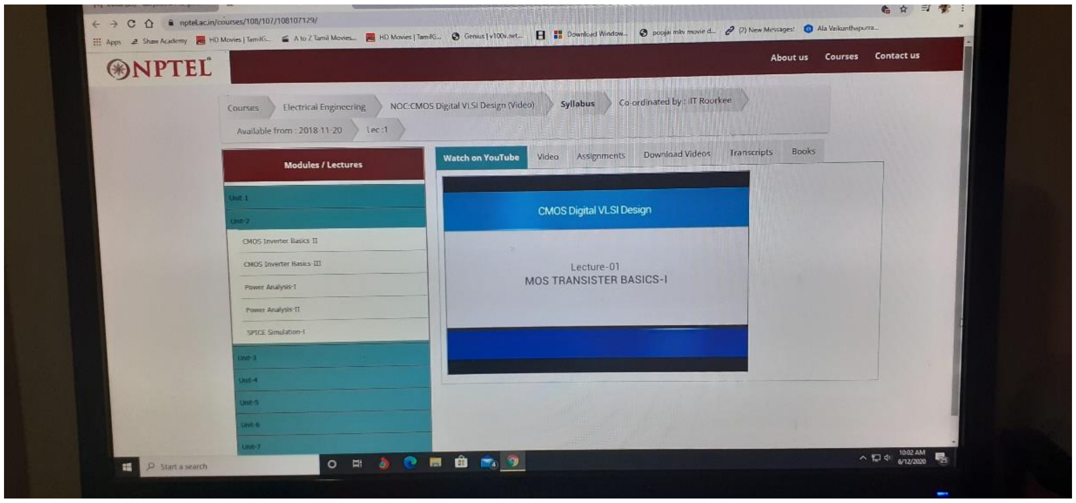
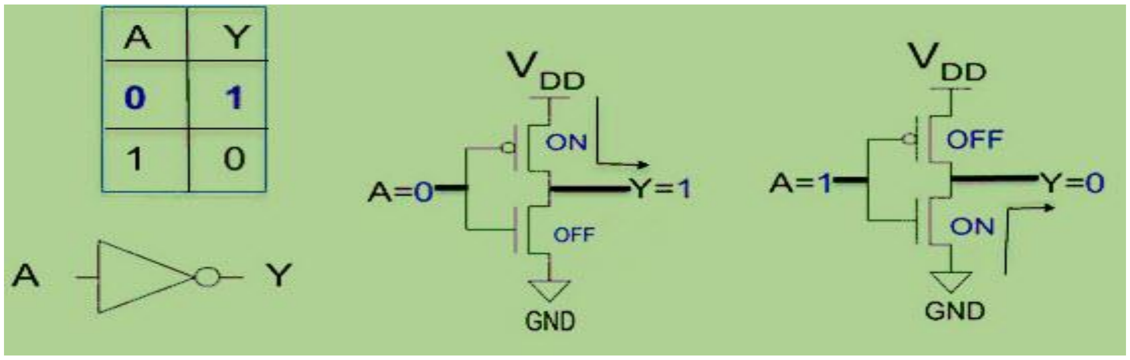


Fig : CMOS Inverter Basics

CMOS Inverter Basics

CMOS inverter

The inverter circuit as shown in the figure below. It consists of PMOS and NMOS FET. The input A serves as the gate voltage for both transistors.



CMOS Inverter

The NMOS transistor has an input from Vss (ground) and PMOS transistor has an input from Vdd. The terminal Y is output. When a high voltage (~ Vdd) is given at input terminal (A) of the inverter, the PMOS becomes open circuit and NMOS switched OFF so the output will be pulled down to Vss.

When a low-level voltage (<Vdd, ~0v) applied to the inverter, the NMOS switched OFF and PMOS switched ON. So the output becomes Vdd or the circuit is pulled up to Vdd.

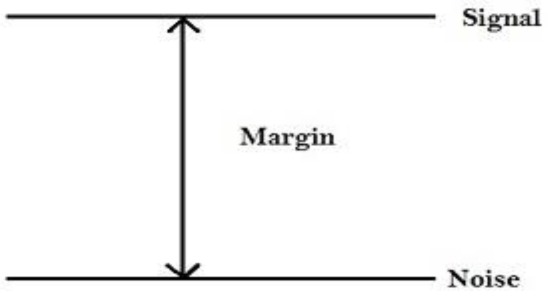
INPUT	LOGIC INPUT	OUTPUT	LOGIC OUTPUT
0 v	0	Vdd	1
Vdd	1	0 v	0

Switching Threshold

The switching threshold, VM, is defined as the point where Vin = Vout. Its value can be obtained graphically from the intersection of the VTC with the line given by Vin = Vout.In this region, both PMOS and NMOS are always saturated, since VDS = VGS. An analytical expression for VM is obtained by equating the currents through the transistors. We solve the case where the supply voltage is high so that the devices can be assumed to be velocity-saturated (or VDSAT < VM - VT).

Noise margin

Noise margin is the amount of noise that a CMOS circuit could withstand without compromising the operation of circuit. Noise margin does makes sure that any signal which is logic ‘1’ with finite noise added to it, is still recognized as logic ‘1’ and not logic ‘0’. It is basically the difference between signal value and the noise value. Refer to the diagram below.



Consider the following output characteristics of a CMOS inverter. Ideally, When input voltage is logic ‘0’, output voltage is supposed to logic ‘1’. Hence Vil (V input low) is ‘0’V and Voh (V output high) is ‘Vdd’V.

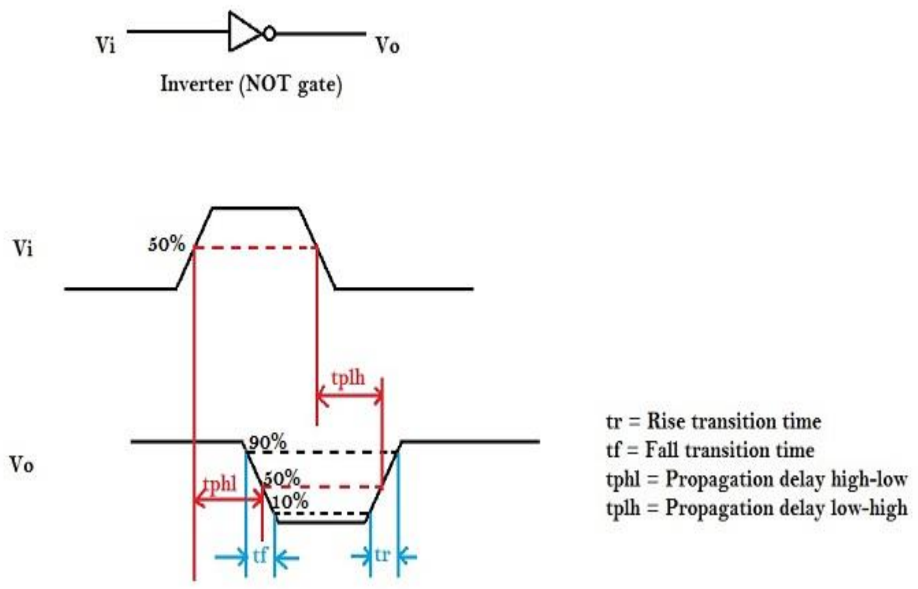
- Vil = 0
- Voh = Vdd

Ideally, when input voltage is logic ‘1’, output voltage is supposed to be at logic ‘0’. Hence, Vih (V input high) is ‘Vdd’, and Vol (V output low) is ‘0’V.

- Vih = Vdd
- Vol = 0

Propagation Delay of CMOS inverter

The propagation delay of a logic gate e.g. inverter is the difference in time, when output switches, after application of input.


Vi
Vo
Inverter (NOT gate)
Vi
50%
tplh
Vo
90%
50%
10%
tphl
tf
tr
tr = Rise transition time
tf = Fall transition time
tphl = Propagation delay high-low
tplh = Propagation delay low-high

In the above figure, there are 4 timing parameters. Rise time (t_r) is the time, during transition, when output switches from 10% to 90% of the maximum value. Fall time (t_f) is the time, during transition, when output switches from 90% to 10% of the maximum value. Many designs could also prefer 30% to 70% for rise time and 70% to 30% for fall time. It could vary upto different designs.

The propagation delay high to low (t_{pHL}) is the delay when output switches from high-to-low, after input switches from low-to-high. The delay is usually calculated at 50% point of input-output switching, as shown in above figure.

Now, in order to find the propagation delay, we need a model that matches the delay of inverter. As we have seen above, the switching behavior of CMOS inverter could be modeled as a resistance R_{on} with a capacitor C_L, a simple first order analysis of RC network will help us to model the propagation delay.

Inverter capacitance

Load capacitance in a CMOS circuit is a combination of input capacitance of the following circuit(s) and the capacitance of the interconnect. (For long interconnects things get more tricky as transmission line effects need to be taken into consideration)

The effect of load capacitance is that it causes a transient current demand on the inverter output, which causes a number of secondary effects, two of which are:

The output has a limited current capability, so this limits the maximum rate of change of the signal, slowing down the edges.

The transient output current is drawn from the power supply and hence causes spikes in the power supply (since the power supply and its interconnect are non-ideal and have series impedance). This is the reason why decoupling capacitors need to be connected between the power rails close to the output stage.

Optimum nmos –pmos ratio

Static CMOS gates are a "ratioless" circuit family, meaning that the gates will work correctly for any ratio of PMOS sizes to NMOS sizes. However, the ratios do influence switching threshold and delay, so it is important to optimize the P/N ratio for high speed designs. In this section, we will explore the DC transfer characteristics of various ratios, the question of sizing for equal rise/fall resistance or minimum delay, and skewing gates to favor critical edges.

Dynamic power dissipation

Dynamic power dissipation can be further subdivided into three mechanisms: switched, short-circuit, and glitch power dissipation. All of them more or less depend on the activity, timing, output capacitance, and supply voltage of the circuit. The repeated charging and discharging of the output capacitance is necessary to transmit information in CMOS circuits. This charging and discharging causes for the switched power dissipation. The power consumption of a CMOS digital circuit can be represented as

$P = fCV_{dd}^2 + f I_{short} V_{dd} + I_{leak} V_{dd}$ (3.1)

Where f is the clock frequency , C is the average switched Capacitance per clock cycle, Vdd is the supply voltage , I short is the short circuit current and I leak is the leakage current. In a well optimized low power VLSI circuits, the Ist term of equation (3.1) is by far the dominant. The stand by power consumption is accounted for by the 3rd term. Using a lower Vdd is an effective way to reduce the dynamic power consumption since Ist term is proportional to the square of Vdd.It should also be noted that the short circuit and leakage power dissipation are also strongly dependent on Vdd.However, using a lower Vdd degrades performance.

Static power dissipation

The static power components become important when the circuits are at rest, i.e. when there is no activity in the circuits and they are all biased to a specific state. The static power dissipation includes sub threshold and reversedbiased diode leakage currents. Due to the necessary but harmful down-scaling of threshold voltages, the sub threshold leakage is becoming more and more pronounced. Below the threshold voltage, in weak inversion, the transistors are not completely off. The sub threshold current has a strong dependence on the threshold voltage.

Power delay product and energy delay product

In digital electronics, the power–delay product (PDP) is a figure of merit correlated with the energy efficiency of a logic gate or logic family. Also known as switching energy, it is the product of power consumption P (averaged over a switching event) times the input–output delay or duration of the switching event D. It has the dimension of energy and measures the energy consumed per switching event.In a CMOS circuit the switching energy and thus the PDP for a 0-to-1-to-0 computation cycle is $C_L \cdot V_{DD}^2$. Therefore, lowering the supply voltage V_{DD} lowers the PDP.

Energy-efficient circuits with a low PDP may also be performing very slowly, thus energy–delay product (EDP), the product of E and D (or P and D^2), is sometimes a preferable metric.

- Energy-Delay Product (EDP) = quality metric of gate = E × t

Source of leakage current

There are five major sources of leakage currents in CMOS transistors, they are:

- Gate oxide tunnelling leakage (I_G)
- Subthreshold leakage (I_{SUB})
- Reverse-bias junction leakages (I_{REV})
- Gate Induced Drain Leakage (I_{GIDL})
- Gate current due to hot-carrier injection (I_H)

Gate oxide tunnelling leakage

The downscaling of the gate oxide thickness increases the field oxide across the gate resulting to electron tunnelling from gate to substrate or from substrate to gate. The resulting current is called gate oxide tunnelling current and it is the major leakage current in the nanometer CMOS. Two mechanisms are responsible for this phenomenon. The first is called Fowler-Nordheim (FN) tunnelling mechanism, which is electron tunnelling into the conduction band of the oxide layer. The other mechanism, direct tunnelling, is more dominant than the FN. In this case, electron tunnel directly to the gate through the forbidden energy gap of the silicon dioxide layer. The resulting current is called the gate direct-tunnelling leakage and it flows from the gate through the oxide insulation to the substrate.

Subthreshold leakage

The subthreshold leakage is the drain-source current of a transistor during operation in weak inversion (where transistors switch ON though the gate source voltage is below the threshold voltage, the voltage at which when exceeded the transistor is expected to be turned ON). Unlike the strong inversion region in which the drift current dominates, the subthreshold conduction is due to the diffusion current of the minority carriers in the channel for a metal oxide semiconductor (MOS) device. The magnitude of the subthreshold current is a function of the temperature, supply voltage, device size, and the process parameters.

Reverse-bias source/drain junction leakages

Though the p-n junctions between the source/drain and the substrate are reverse-biased, yet a small amount of current flows causing these junctions to leak. This current is called reverse biased junction leakage current. The magnitude of this current depends on the area of the source/drain diffusion and the current density, which is in turn determined by the doping concentration. The highly doped shallow junctions and halo doping necessary to control short channel effects (SCE) in the nanometer devices has escalated this leakage current. Under this situation, electrons tunnel across the p-n junction causing junction leakage.

Gate Induced Drain Leakage (GIDL)

This leakage current is caused by high electric field effect in the drain junction of MOS transistors. Over the years, transistor scaling has led to increasingly steep halo implants, where the substrate doping at the junction interfaces is increased, while the channel doping is low. Its purpose is to control punch-through and drain-induced barrier lowering with minimal impact on the mobility of the carrier in the channel. The steep doping profile that results at the drain edge increases the band-to-band tunnelling currents there, especially as drain-bulk voltage (V_{db}) is increased. Thinner oxide and higher supply voltage increase GIDL current. Controlling the doping concentration in the drain of the transistor is the best way to control GIDL.

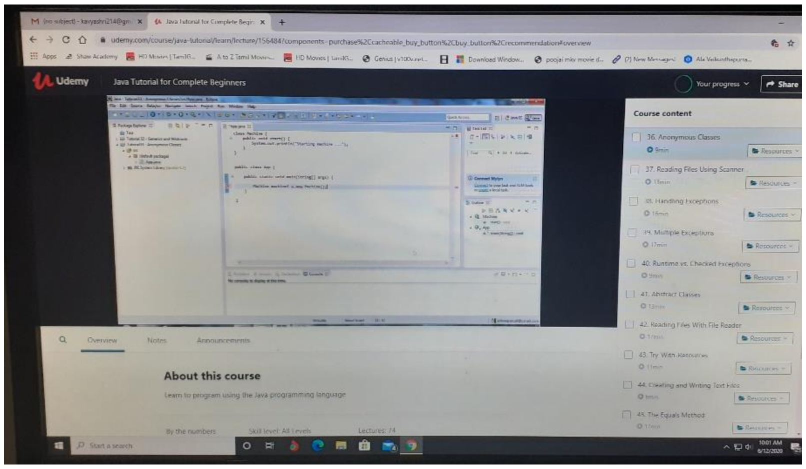
Gate current due to hot-carrier injection (I_H)

This leakage current is due to drift over time of the threshold voltage in short channel devices. The high electric field near the Si-SiO_2 interface can cause electrons or holes to gain sufficient energy to overcome the interface potential and enter into the oxide layer. In this phenomenon known as hot carrier effect, the electron injection is more likely to occur than the hole as electron has both lower effective mass and barrier height than hole. These carriers trapped in the oxide layer change the threshold voltage of the device and consequently the subthreshold current. Proportionate scaling down of the supply voltage with the device dimension is one possible way of controlling this leakage.

AFTERNOON SESSION DETAILS

Date:	**12-6-2020**	**Name:**	**Kavyashree m**
Course:	**Java**	**USN:**	**4al15ec036**
Topic:	**Generics and wildcard,anonymous classes,reading files using scanner,handling exceptions,multiple exceptions, Runtime vs. Checked Exceptions, Abstract Classes,Reading Files With File Reader, Creating and Writing Text Files**	**Semester & Section:**	**8th A**
Github Repository:	**kavya**		

Image of session


Udemy
Java Tutorial for Complete Beginners
Your progress
Share
Course content
36. Anonymous Classes
37. Reading Files Using Scanner
38. Handling Exceptions
39. Multiple Exceptions
40. Runtime vs. Checked Exceptions
41. Abstract Classes
42. Reading Files With File Reader
43. Try With Resources
44. Creating and Writing Text Files
45. The Equals Method
Overview
Notes
Announcements
About this course
Learn to program using the Java programming language

Generics and wildcard

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type. Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly.

Anonymous classes

Anonymous classes are inner classes with no name. Since they have no name, we can't use them in order to create instances of anonymous classes. As a result, we have to declare and instantiate anonymous classes in a single expression at the point of use.

An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class. Its syntax is:

```
class outerClass {


    // defining anonymous class
    object1 = new Type(parameterList) {
         // body of the anonymous class
    };
}
```

Reading files using scanner

In this Java program, we have used java.util.Scanner to read file line by line in Java. We have first created a File instance to represent a text file in Java and than we passed this File instance to java.util.Scanner for scanning. Scanner provides methods like hasNextLine() and readNextLine() which can be used to read file line by line. It's advised to check for next line before reading next line to avoid NoSuchElementException in Java. Here is complete code example of using Scanner to read text file in Java :

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 *
 * Java program to read file using Scanner class in Java.
 * java.util.Scanner is added on Java 5 and offer convenient method to read data
 *
 * @author
 */
public class ScannerExample {

    public static void main(String args[]) throws FileNotFoundException {

        //creating File instance to reference text file in Java
        File text = new File("C:/temp/test.txt");

        //Creating Scanner instnace to read File in Java
        Scanner scnr = new Scanner(text);

        //Reading each line of file using Scanner class
        int lineNumber = 1;
        while(scnr.hasNextLine()){
            String line = scnr.nextLine();
            System.out.println("line " + lineNumber + " :" + line);
            lineNumber++;
        }
    }

}
```

Output:

```
line 1 :-------------------- START--------------------------------------------------
line 2 :Java provides several way to read files.
line 3 :You can read file
using Scanner, FileReader, FileInputStream and BufferedReader.
line 4 :This Java program shows How to read file using java.util.Scanner class.
line 5 :-------------------- END------------------------------------------------------
```

This is the content of test.txt file exception line numbers. You see it doesn't require much coding to read file in Java using Scanner. You just need to create an instance of Scanner and you are ready to read file.

Handling exceptions

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

Example:

```
// Java program to demonstrate how exception is thrown.
class ThrowsExecp{

    public static void main(String args[]){

        String str = null;
        System.out.println(str.length());


    }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
    at ThrowsExecp.main(File.java:8)
```

Multiple exception

In the following code, we have to handle two different exceptions but take same action for both. So we needed to have two different catch blocks as of Java 6.0.

```
// A Java program to demonstrate that we needed
// multiple catch blocks for multiple exceptions
// prior to Java 7
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (99%n == 0)
                System.out.println(n + " is a factor of 99");
        }
        catch (ArithmeticException ex)
        {
           System.out.println("Arithmetic " + ex);
        }
        catch (NumberFormatException ex)
        {
           System.out.println("Number Format Exception " + ex);
        }
    }
}
```

Input 1:

```
GeeksforGeeks
```

Output 2:

```
Exception encountered java.lang.NumberFormatException:
For input string: "GeeksforGeeks"
```

Input 2:

```
0
```

Output 2:

```
Arithmetic Exception encountered java.lang.ArithmeticException: / by zero
```

Runtime Vs checked exceptions

Runtime Exceptions : Runtime exceptions are referring to as unchecked exceptions. All other exceptions are checked exceptions, and they don't derive from java.lang.RuntimeException.

Checked Exceptions : A checked exception must be caught somewhere in your code. If you invoke a method that throws a checked exception but you don't catch the checked exception somewhere, your code will not compile. That's why they're called checked exceptions : the compiler checks to make sure that they're handled or declared.

Abstract class in Java

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- ➢ An abstract class must be declared with an abstract keyword.
- ➢ It can have abstract and non-abstract methods.
- ➢ It cannot be instantiated.
- ➢ It can have constructors and static methods also.
- ➢ It can have final methods which will force the subclass not to change the body of the method.

Reading files with file reader

FileReader is an object with the sole purpose of reading data from Blob (and hence File too) objects.

It delivers the data using events, as reading from disk may take time.

The constructor:

let reader = new FileReader(); // no arguments

The main methods:

- readAsArrayBuffer(blob) – read the data in binary format ArrayBuffer.
- readAsText(blob, [encoding]) – read the data as a text string with the given encoding (utf-8 by default).
- readAsDataURL(blob) – read the binary data and encode it as base64 data url.
- abort() – cancel the operation.

The choice of read* method depends on which format we prefer, how we're going to use the data.

- readAsArrayBuffer – for binary files, to do low-level binary operations. For high-level operations, like slicing, File inherits from Blob, so we can call them directly, without reading.
- readAsText – for text files, when we'd like to get a string.
- readAsDataURL – when we'd like to use this data in src for img or another tag. There's an alternative to reading a file for that, as discussed in chapter Blob: URL.createObjectURL(file).

As the reading proceeds, there are events:

- loadstart – loading started.
- progress – occurs during reading.
- load – no errors, reading complete.
- abort – abort() called.
- error – error has occurred.
- loadend – reading finished with either success or failure.

When the reading is finished, we can access the result as:

- reader.result is the result (if successful)
- reader.error is the error (if failed).

Creating and Writing Text Files

Create a File

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

Example

```
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
File created: filename.txt
```

Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

Example

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Successfully wrote to the file.
```