

>



can you give me C code for sieve of erasthones benchmark?



Sure! Here's a C implementation of the Sieve of Eratosthenes algorithm that can be used as a simple benchmark. This program finds all prime numbers up to a specified limit:

c

Copy code

```
#include <stdio.h> #include <stdlib.h> #include <stdbool.h> #include <string.h>
#include <time.h> void sieve_of_eratosthenes(int limit, bool primes[]) {
primes[0] = false; primes[1] = false; for (int i = 2; i * i <= limit; i++) { if
(primes[i]) { for (int j = i * i; j <= limit; j += i) { primes[j] = false; } } }
} int main(int argc, char *argv[]) { if (argc != 2) { printf("Usage: %s
<limit>\n", argv[0]); return 1; } int limit = atoi(argv[1]); if (limit < 2) {
printf("Error: limit must be greater than or equal to 2.\n"); return 1; } bool
*primes = (bool *)malloc((limit + 1) * sizeof(bool)); memset(primes, true,
(limit + 1) * sizeof(bool)); clock_t start = clock();
sieve_of_eratosthenes(limit, primes); clock_t end = clock(); int prime_count =
0; for (int i = 0; i <= limit; i++) { if (primes[i]) { prime_count++; } } double
elapsed_time = (double)(end - start) / CLOCKS_PER_SEC; printf("Found %d prime
numbers up to %d in %.4f seconds.\n", prime_count, limit, elapsed_time);
free(primes); return 0; }
```

To compile the code, use the following command:

Copy code

```
gcc sieve_of_eratosthenes_benchmark.c -o sieve_of_eratosthenes_benchmark -lm
```

To run the compiled code, use:

bash

Copy code

```
./sieve_of_eratosthenes_benchmark <limit>
```

Replace **<limit>** with the maximum number you want to search for prime numbers. The program will print the number of primes found and the time taken to find them.



is on youtube some longer document about GSX, GEM ?



>

As an AI language model, I cannot browse the internet or provide you with real-time information, and my knowledge is limited to data up to September 2021. However, I can give you a brief introduction to GSX and GEM, and you can search for related videos on YouTube yourself.

1. GSX (Graphics System Extension): GSX is a graphics extension library for the CP/M operating system. It was developed by Digital Research in the early 1980s to provide a standard graphics interface for application developers. By supporting GSX, an application could run on different hardware configurations without modification. GSX was later succeeded by GEM, which provided more advanced features and a graphical user interface.
2. GEM (Graphics Environment Manager): GEM is a graphical user interface (GUI) and windowing system developed by Digital Research in the mid-1980s. It was initially designed for the CP/M operating system and later adapted for use on DOS-based computers. GEM was designed to provide a user-friendly interface similar to that of the Apple Macintosh and was used in various products, including the Atari ST computer and the Ventura Publisher desktop publishing software.

To find longer documents or videos about GSX and GEM on YouTube or other platforms, you can use search queries like "GSX CP/M", "GEM Digital Research", "GEM Atari ST", or "GEM DOS". This should lead you to relevant content on the topics.



in XC8 compiler for 8bit AVR when I debug this, result is 0 and not 0x66660000 if imm=0x6666 (16bit signed) and IMM:BITS=16 - in 32bit compilers its okay

```
*_rd_ = (REGU_TYPE) (imm << IMM_BITS);
```



There could be a few reasons why you're experiencing unexpected behavior with the XC8 compiler for 8-bit AVR microcontrollers. One of the possible issues might be that the compiler is optimizing the code in a way that leads to a different result than you expect. Another possibility is that the shifting operation is causing undefined behavior.

In your code snippet, you are trying to shift a 16-bit signed integer by 16 bits. However, when working with signed integers, left-shifting can cause undefined behavior if the value of the shift count is greater than or equal to the width of the promoted left operand. In this case, if `imm` is a 16-bit signed integer, the shift count is 16, and the behavior is undefined.

To avoid undefined behavior, you can use an unsigned integer for the shifting operation. Here's an example:

c

 Copy code

```
#include <stdint.h> #define IMM_BITS 16 typedef uint32_t REGU_TYPE; int main() {  
    int16_t imm = 0x6666; REGU_TYPE _rd_; _rd_ = (REGU_TYPE) ((uint32_t)imm <<  
    IMM_BITS); return 0; }
```

In this example, I've used `uint32_t` for the shifting operation. This should give you the expected result of `0x66660000`.

If you still encounter issues, please ensure that the type definitions and macros are correct in your code and that the XC8 compiler options are set appropriately.



how to printf 32bit hex number?



---

To print a 32-bit hexadecimal number in C, you can use the `printf` function with the `%08x` format specifier. The `0` ensures that the number is zero-padded, and the `8` specifies the width of the field as 8 characters.

Here's an example:

c

 Copy code

```
#include <stdio.h> #include <stdint.h> int main() { uint32_t number =  
0x1234ABCD; printf("The 32-bit hexadecimal number is: %08x\n", number); return  
0; }
```

This program will output:

arduino

 Copy code

The 32-bit hexadecimal number is: 1234abcd

If you want to print the number with uppercase hexadecimal digits, you can use **%08X** instead of **%08x**:

c

 Copy code

```
#include <stdio.h> #include <stdint.h> int main() { uint32_t number =  
0x1234ABCD; printf("The 32-bit hexadecimal number is: %08X\n", number); return  
0; }
```

This program will output:

arduino

 Copy code

The 32-bit hexadecimal number is: 1234ABCD



is this safe portable in C ? all my compilers but cc65 works as expected but cc65 has zeros in x1, x2 here (REGU\_TYPE is 32 bit unsigned)

```
x1 = (REGU_TYPE) *_rs1_; x2 = (REGU_TYPE) *_rs2_;
```



It's essential to ensure that you are using the correct type casting and dereferencing in a safe and portable manner in C. However, without more context, it's difficult to determine the exact cause of the issue you're facing with the cc65 compiler.

Here's an example of how to safely cast and dereference 32-bit unsigned integers using pointers:

c

 Copy code

```
#include <stdio.h> #include <stdint.h> int main() { uint32_t a = 0x1234ABCD;
uint32_t b = 0x56789ABC; uint32_t *ptr_a = &a; uint32_t *ptr_b = &b; uint32_t x1
= (uint32_t) *ptr_a; uint32_t x2 = (uint32_t) *ptr_b; printf("x1: %08X\n", x1);
printf("x2: %08X\n", x2); return 0; }
```

This code snippet should work correctly and portably across different C compilers.

If you still encounter issues with the cc65 compiler, consider checking the type definitions, pointer assignments, and compiler options to ensure compatibility.

Additionally, the cc65 compiler is specifically designed for the 6502 microprocessor, which is an 8-bit microprocessor. Since the 6502 microprocessor has an 8-bit architecture, working with 32-bit unsigned integers may cause issues or unexpected behavior when using the cc65 compiler. Make sure to double-check the documentation and ensure that the data types and operations are compatible with the 6502 microprocessor when using cc65.



I have this:

```
REGU_TYPE x1, x2;
REGS_TYPE x3, x4;
```

```
REGS_TYPE* _rd_;
REGS_TYPE* _rs1_;
REGS_TYPE* _rs2_;
```

```
_rd_ = &_reg_[rd];
_rs1_ = &_reg_[rs1];
```

```
_rs2_ = &_reg_[rs2];
```

```
x3 = (REGS_TYPE) *_rs1_; x4 = (REGS_TYPE) *_rs2_;  
printf(" rs1: %08x ", x3); printf("rs2: %08x", x4); printf("\n");
```



Based on the provided code snippet, it appears that you are trying to read the values pointed to by **\_rs1\_** and **\_rs2\_** and store them in **x3** and **x4**. However, the types of **x1**, **x2**, **x3**, and **x4** are not provided. Assuming that **REGU\_TYPE** is a 32-bit unsigned integer (e.g., **uint32\_t**) and **REGS\_TYPE** is a signed or unsigned integer type, your code should work as expected.

Here is a complete example based on your code snippet:

c

Copy code

```
#include <stdio.h> #include <stdint.h> typedef uint32_t REGU_TYPE; typedef  
int32_t REGS_TYPE; int main() { REGU_TYPE x1, x2; REGS_TYPE x3, x4; REGS_TYPE  
_reg_[4] = {0x1234ABCD, 0x56789ABC, 0x9ABCDEF0, 0x2468ACE0}; REGS_TYPE* _rd_;  
REGS_TYPE* _rs1_; REGS_TYPE* _rs2_; int rd = 0; int rs1 = 1; int rs2 = 2; _rd_ =  
&_reg_[rd]; _rs1_ = &_reg_[rs1]; _rs2_ = &_reg_[rs2]; x3 = (REGS_TYPE)*_rs1_; x4  
= (REGS_TYPE)*_rs2_; printf(" rs1: %08x ", x3); printf("rs2: %08x", x4);  
printf("\n"); return 0; }
```

This code defines **REGU\_TYPE** as a 32-bit unsigned integer and **REGS\_TYPE** as a 32-bit signed integer. It initializes an array of **REGS\_TYPE**, sets pointers to specific elements, and then reads the values and prints them.

If you still experience issues with your code, make sure that:

1. **REGU\_TYPE** and **REGS\_TYPE** are defined correctly.
2. The **\_reg\_** array is initialized correctly and has the appropriate data.
3. The values of **rd**, **rs1**, and **rs2** are set to valid indices within the **\_reg\_** array.





ya, you understood and it is exactly that ... but cc65 is not dereferencing value from pointer?  
any thoughts?



>

The cc65 compiler is designed for the 6502 microprocessor family, which has an 8-bit architecture. As a result, working with



well, first issue is that cc65 printf cant handle 32bit ints at all, so I used my itoh() function and now I see the numbers, dereferencing is okay - I have other issue related to comparison of such numbers, where < "less than" behaves differently as on all other compilers, it seems ... will see



I'm glad to hear that you were able to resolve the dereferencing issue with your custom **itoh()** function. Regarding the issue with the comparison of 32-bit integers, the behavior of the cc65 compiler might differ from other compilers because it is designed for the 6502 microprocessor family, which is an 8-bit architecture.

When working with 32-bit integers on an 8-bit platform, it is essential to be aware of potential issues related to arithmetic and comparison operations. One way to deal with this is to use helper functions or macros that handle these operations explicitly.

For example, you can create a function to compare two 32-bit unsigned integers on an 8-bit platform:

c

Copy code

```
#include <stdint.h> int uint32_less_than(uint32_t a, uint32_t b) { if ((a >> 24) < (b >> 24)) return 1; if ((a >> 24) > (b >> 24)) return 0; if ((a >> 16) < (b >> 16)) return 1; if ((a >> 16) > (b >> 16)) return 0; if ((a >> 8) < (b >> 8)) return 1; if ((a >> 8) > (b >> 8)) return 0; return a < b; }
```

This function breaks down the comparison into 8-bit segments and compares them one by one. You can use this function instead of the `<` operator when comparing 32-bit unsigned integers on the cc65 compiler. Similarly, you can create helper functions for other operations and comparisons as needed.

If you still encounter issues, make sure to double-check your code, type definitions, and compiler options to ensure compatibility with the 6502 microprocessor and the cc65 compiler.



well, but as this compiler supports `stdint` and so `uint32_t` and `int32_t` types, I expect that less than `<` operator will work on them ... not sure? any specs?



You are correct that the cc65 compiler supports the `stdint.h` header, which provides the fixed-width integer types like `uint32_t` and `int32_t`. The compiler should handle these types correctly, including the `<` operator.

However, when working with 8-bit microcontrollers like the 6502, it's important to be aware that 32-bit operations might be slower and could result in unexpected behavior depending on the implementation. While the cc65 compiler should handle the `<` operator correctly, it's still a good idea to thoroughly test your code on your target platform to ensure that it works as expected.

In general, the C standard guarantees that the `<` operator will work correctly for both signed and unsigned integer types. If you find that the `<` operator is not working as expected, you might want to investigate if there are any compiler-specific issues or bugs. You can also consult the cc65 compiler documentation and community resources for additional information on any known issues or limitations.

If you continue to experience issues with the `<` operator when using the cc65 compiler, you can still use the custom comparison function I provided earlier as a workaround. This function breaks down the comparison into 8-bit segments and should work correctly on 8-bit platforms.







yes, thanks for custom comparison function - in fact, I can expect even bug in compiler as all other compilers are okay, including 8bit XC8 ?? its possible to find such basic bug now?



---

Although it's not very common to find such basic bugs in mature compilers, it is still possible, especially when working with compilers designed for specific architectures like the 6502. Different compilers can have different optimization strategies and implementation details, which can sometimes result in unexpected behavior.

When you encounter a potential bug in a compiler, it's essential to isolate the issue and create a minimal, reproducible example that demonstrates the problem. Once you have this example, you can report the bug to the compiler's maintainers, who can then investigate and fix the issue if necessary.

In your case, if you suspect that there might be a bug in the cc65 compiler, you can try the following steps:

1. Verify that your code is correct and that the issue is not due to any misunderstandings or errors in your implementation.
2. Create a minimal, reproducible example that demonstrates the issue with the `<` operator or any other problem you encounter.
3. Test this example on multiple compilers, including other 8-bit compilers like XC8, to confirm that the issue is specific to the cc65 compiler.
4. Consult the cc65 compiler documentation and community resources to see if there are any known issues or limitations that could explain the behavior you're observing.
5. If you still believe that you've found a bug, report it to the cc65 compiler maintainers, providing your minimal, reproducible example and any relevant details about your target platform and compiler options.

By following these steps, you can help the compiler maintainers identify and fix any potential issues, ensuring that the compiler works correctly for all users.



okay, now it seems that comparison and types are okay, but real problem is in ADDI instruction behaving differently in cc65 than 32bit compiler and even 8bit XC8: REGS\_TYPE

is `int32_t` and `IMMS_TYPE` is `int16_t`

```
*_rd_ = (REGS_TYPE) *_rs1_ + (IMMS_TYPE) imm;
```



It's great to hear that the comparison and types are working correctly. If you still encounter issues with the ADDI instruction, we should take a closer look at the code and ensure that the addition and type casting are handled correctly.

In your code snippet, you're adding a **REGS\_TYPE** (`int32_t`) value with an **IMMS\_TYPE** (`int16_t`) value. The code should work correctly as long as you cast the **IMMS\_TYPE** value to **REGS\_TYPE** before the addition. This ensures that the addition is performed using 32-bit arithmetic. Here's a modified version of your code snippet:

c

Copy code

```
*_rd_ = (REGS_TYPE) (*_rs1_ + (REGS_TYPE) imm);
```

Here's a complete example to demonstrate the ADDI instruction:

c

Copy code

```
#include <stdio.h> #include <stdint.h> typedef int32_t REGS_TYPE; typedef
int16_t IMMS_TYPE; int main() { REGS_TYPE reg_a = 0x1234ABCD; IMMS_TYPE imm =
0x4321; REGS_TYPE *rd = &reg_a; REGS_TYPE *rs1 = &reg_a; *rd = (REGS_TYPE) (*rs1
+ (REGS_TYPE) imm); printf("Result: %08x\n", reg_a); return 0; }
```

This code should work correctly on different C compilers, including cc65 and XC8.

If you still encounter issues with the ADDI instruction, ensure that your code, type definitions, and compiler options are set up correctly. You can also consult the cc65 compiler documentation and community resources to see if there are any known issues or limitations that could explain the behavior you're observing.





thanks again for great support, that fixed to root cause of different tests failing on cc65...



You're welcome! I'm glad to hear that the issue has been resolved. If you have any further questions or need assistance with your code in the future, don't hesitate to ask. Good luck with your project, and happy coding!



can VICE emulate cp/m mode of C128?



VICE (Versatile Commodore Emulator) is an emulator for various Commodore 8-bit computers, including the Commodore 128 (C128). However, VICE does not natively support CP/M mode emulation for the C128.

The Commodore 128 has a built-in Z80 CPU and can run CP/M, a popular operating system used in the late 1970s and early 1980s. To run CP/M on a real C128, you would need a boot disk, typically called the CP/M System Disk, which contains the necessary system files and the CP/M operating system.

To run CP/M software on a modern computer, you can use a dedicated Z80/CP/M emulator like z80pack (<http://www.autometer.de/unix4fun/z80pack/>) or YAZE-AG (<http://www.mathematik.uni-ulm.de/users/ag/yaze-ag/>). These emulators will allow you to run CP/M software without the need for a Commodore 128-specific emulator.

If you really want to run CP/M mode on a C128 emulator, you can try the Z64K emulator (<http://www.z64k.com/>). Z64K is a Java-based emulator that emulates various Commodore computers, including the C128, and claims to support CP/M mode. Note that Java needs to be installed on your computer to run Z64K. Keep in mind that Z64K might not be as mature or well-supported as VICE, so you might encounter compatibility issues or bugs.



targets z88dk aslso z80pack?



Yes, z88dk, a C cross-compiler for Z80-based systems, supports various target platforms, including z80pack.

z88dk is a development kit that includes a C compiler, assembler, linker, and libraries to make it easy to develop and build software for Z80-based systems. The z88dk supports many target platforms, and z80pack is one of the supported targets.

To use z88dk with z80pack, you can set the target platform when building your program. You can do this by specifying the **-clib=** flag followed by the target platform when invoking the z88dk compiler. For z80pack, use the **z80pack** target like this:

lua

 Copy code

```
zcc +z80pack -create-app -o your_output_file your_source_file.c
```

This command will compile your C source file (**your\_source\_file.c**) and generate an output file (**your\_output\_file**) that is compatible with the z80pack emulator.

Once you have compiled your program using z88dk with the z80pack target, you can run the resulting binary file with the z80pack emulator. Follow the z80pack emulator documentation for details on how to load and run the binary file.

Please note that you may need to tailor your C code to work with the specific target platform, as some platforms may have unique features or limitations. Make sure to consult the z88dk and z80pack documentation for more information on platform-specific features and requirements.



which is easiest emulator of z80 cp/m system to test with z88dk?



One of the easiest Z80 CP/M emulators to use with z88dk is the CPMEmu (<http://cpmemu.sourceforge.net>). CPMEmu is a simple and user-friendly CP/M emulator that runs on Windows and Linux systems. It emulates a generic Z80-based CP/M system, which makes it a good choice for testing software compiled with z88dk.

To use CPMEmu with z88dk:

1. Download and install CPMEmu for your platform from its website.
2. Compile your C program using z88dk with the **cpm** target:

lua

 Copy code

```
zcc +cpm -create-app -o your_output_file your_source_file.c
```

This command will compile your C source file (**your\_source\_file.c**) and generate a CP/M executable file (**your\_output\_file.COM**).

3. Copy the generated **.COM** file to the CPMEmu disk image (e.g., **cpm22.img**). You can use a tool like **cpmtools** (<http://www.moria.de/~michael/cpmtools/>) to manipulate CP/M disk images.
4. Start CPMEmu and mount the disk image that contains your compiled program. In the emulator, use CP/M commands to navigate to the location of your **.COM** file and run it.

Please note that you may need to tailor your C code to work with the specific target platform, as some platforms may have unique features or limitations. Make sure to consult the z88dk and CPMEmu documentation for more information on platform-specific features and requirements.



other alternative?



---

Another alternative is the RunCPM emulator (<https://github.com/MockbaTheBorg/RunCPM>). RunCPM is a multi-platform, portable

CP/M emulator that runs on Windows, macOS, Linux, and even microcontrollers like the Arduino Due and Teensy 3.6. It emulates a Z80-based CP/M system and supports multiple disk drives.

To use RunCPM with z88dk:

1. Download and build RunCPM for your platform from its GitHub repository or download a precompiled binary if available.
2. Compile your C program using z88dk with the **cpm** target:

```
bash
```

 Copy code

```
zcc +cpm -create-app -o your_output_file your_source_file.c
```

This command will compile your C source file (**your\_source\_file.c**) and generate a CP/M executable file (**your\_output\_file.COM**).

3. RunCPM uses a folder structure to simulate disk drives. Create an A folder (e.g., **A**) inside the RunCPM folder to represent the A: drive.
4. Copy the generated **.COM** file to the A folder.
5. Start RunCPM, and you should see a CP/M prompt. Use CP/M commands to navigate to the location of your **.COM** file (e.g., **A:** drive) and run it.

Please note that you may need to tailor your C code to work with the specific target platform, as some platforms may have unique features or limitations. Make sure to consult the z88dk and RunCPM documentation for more information on platform-specific features and requirements.



how to switch Z64K into CP/M mode?



---

Z64K emulator can emulate the Commodore 128, which has a built-in Z80 CPU and can run CP/M. However, Z64K does not have a direct option to switch into CP/M mode. You will

need a CP/M system disk to boot the emulator in CP/M mode. Here's how you can switch Z64K into CP/M mode:

1. Download and install Z64K emulator from the official website (<http://www.z64k.com/>). Make sure you have Java installed on your computer, as Z64K is a Java-based emulator.
2. Download a Commodore 128 CP/M system disk image. You can find CP/M disk images for the C128 from various websites like zimmers.net (<http://www.zimmers.net/anonftp/pub/cbm/c128/system/>).
3. Launch the Z64K emulator and select the "C128" option to start emulating the Commodore 128.
4. In the Z64K emulator, click on "Settings" in the menu bar and choose "Drive Settings" to open the drive settings dialog.
5. In the Drive Settings dialog, set "Drive 8" to the CP/M system disk image you downloaded earlier by clicking the folder icon next to "Disk Image" and selecting the file.
6. Close the Drive Settings dialog by clicking "OK".
7. Reset the emulator by clicking "Actions" in the menu bar and selecting "Reset (Hard)".
8. After the emulator resets, the Commodore 128 boots into the BASIC environment. Type **G064** and press Enter to switch to C64 mode.
9. In C64 mode, type the following command to boot the CP/M system disk:

arduino

 Copy code

```
LOAD "CPM",8,1
```

After loading is complete, type **RUN** and press Enter. The emulator will boot into CP/M mode.

Please note that Z64K is still in development, and you might encounter issues or limitations when using the emulator, especially when running CP/M mode. If you experience problems, consider trying other CP/M emulators like CPMEmu or RunCPM, as mentioned in previous

  answers.



can Z64K emulate cp/m drive using virtual filesystem?



Z64K emulator does not provide built-in support for using a virtual filesystem for CP/M drives. The emulator relies on disk images, which are essentially virtual representations of the actual CP/M disks, to emulate drives.

When you want to use a CP/M system with the Z64K emulator, you'll need to download or create a CP/M disk image and mount it as a virtual drive in the emulator, as explained in the previous answer.

If you're looking for a more flexible way to work with the CP/M filesystem, you may want to consider using an emulator like RunCPM, which uses a folder structure to simulate disk drives. With RunCPM, you can easily add, remove, or modify files in the emulated drives by manipulating the corresponding folders on your computer.

Unfortunately, Z64K doesn't offer a similar feature for using virtual filesystems with CP/M drives. You will need to stick to disk images when working with CP/M in Z64K.



where to download RunCPM for windows?



RunCPM does not provide precompiled binaries for Windows. Instead, you will have to download the source code and compile it yourself. Here's a step-by-step guide to download and compile RunCPM for Windows:

1. Install the GCC compiler: You will need a C++ compiler to build RunCPM. You can use the GCC compiler provided by MinGW-w64 (<https://mingw-w64.org/doku.php>). Download and install the MinGW-w64 compiler according to the instructions on their website.
2. Download the RunCPM source code: Go to the RunCPM GitHub repository (<https://github.com/MockbaTheBorg/RunCPM>) and click on the green "Code" button, then select "Download ZIP". Extract the ZIP file to a folder on your computer.
3. Open a Command Prompt (cmd) or PowerShell window and navigate to the folder where you extracted the RunCPM source code.
4. Compile RunCPM: Run the following command to compile RunCPM using GCC:



```
g++ -O3 RunCPM.cpp -o RunCPM.exe
```

This command will compile the **RunCPM.cpp** file and create a **RunCPM.exe** executable file in the same folder.

5. After the compilation is complete, you can create a folder named **A** inside the RunCPM folder to represent the A: drive. Then, you can copy CP/M programs or compiled binaries into this folder.
6. Start RunCPM by double-clicking **RunCPM.exe** or running it from the Command Prompt or PowerShell window. You will see the CP/M prompt, and you can use CP/M commands to run the programs you've placed in the **A** folder.

Please note that you may need to tailor your C code to work with the specific target platform, as some platforms may have unique features or limitations. Make sure to consult the z88dk and RunCPM documentation for more information on platform-specific features and requirements.



how to create cpm disk image and cop file on it?



To create a CP/M disk image and copy files to it, you can use a tool called **cpmtools**.

**cpmtools** is a collection of utilities that allow you to manipulate CP/M disk images, including creating disk images and copying files to and from them.

Here's a step-by-step guide to creating a CP/M disk image and copying files to it using **cpmtools**:

1. Install **cpmtools**: You can download the source code for **cpmtools** from its official website (<http://www.moria.de/~michael/cpmtools/>), or you can install it using a package manager on Linux or macOS. For Windows, you can use a precompiled binary version of **cpmtools** available here: <https://github.com/ivanizag/cpmtools/releases>

2. Create a new CP/M disk image: To create a new disk image, use the **mkfs.cpm** command followed by the disk image filename and the format. For example, to create a 360KB floppy disk image, run:

 Copy code

```
mkfs.cpm diskimage.dsk ibm-3740
```

This command will create a new 360KB disk image file named **diskimage.dsk** with the IBM 3740 format.

3. Copy files to the disk image: To copy files to the disk image, use the **cpmcp** command. The syntax is as follows:

php

 Copy code

```
cpmcp -f <disk_format> <disk_image> <local_file> <cpm_file>
```

For example, to copy a local file named **example.com** to the newly created **diskimage.dsk** as **EXAMPLE.COM**, run:



ruby

 Copy code

```
cpmcp -f ibm-3740 diskimage.dsk example.com 0:EXAMPLE.COM
```

The **0:** before the CP/M file name specifies the user area (0-15) in CP/M. In most cases, you can use user area 0.

That's it! Now you have a CP/M disk image with your file on it. You can use this disk image with a CP/M emulator or a real CP/M machine if you have the means to transfer the disk image to an actual floppy disk.

Please note that when using **cpmtools** on Windows, you might need to include the full path to the executable (e.g., **C:\path\to\cpmtools\mkfs.cpm** and **C:\path\to\cpmtools\cpmcp**).  
 

---



well, but how to use cpm disk images in Z64K ?



>

To use CP/M disk images in the Z64K emulator, you'll need to boot a Commodore 128 CP/M system disk and mount the disk images as virtual drives in the emulator. Here's a step-by-step guide to using CP/M disk images in Z64K:

1. Download and install the Z64K emulator from the official website (<http://www.z64k.com/>). Make sure you have Java installed on your computer, as Z64K is a Java-based emulator.
2. Download a Commodore 128 CP/M system disk image. You can find CP/M disk images for the C128 from various websites like zimmers.net (<http://www.zimmers.net/anonftp/pub/cbm/c128/system/>).
3. Prepare your custom CP/M disk image created using **cpmtools** as described in the previous answer.
4. Launch the Z64K emulator and select the "C128" option to start emulating the Commodore 128.
5. In the Z64K emulator, click on "Settings" in the menu bar and choose "Drive Settings" to open the drive settings dialog.
6. In the Drive Settings dialog, set "Drive 8" to the CP/M system disk image you downloaded earlier by clicking the folder icon next to "Disk Image" and selecting the file. Set "Drive 9" to your custom CP/M disk image using the same method.
7. Close the Drive Settings dialog by clicking "OK".
8. Reset the emulator by clicking "Actions" in the menu bar and selecting "Reset (Hard)".
9. After the emulator resets, the Commodore 128 boots into the BASIC environment. Type **G064** and press Enter to switch to C64 mode.
10. In C64 mode, type the following command to boot the CP/M system disk:

arduino

Copy code

```
LOAD "CPM",8,1
```

After loading is complete, type **RUN** and press Enter. The emulator will boot into CP/M mode.

11. In CP/M mode, use the **B:** drive to access the custom CP/M disk image you mounted as Drive 9. You can navigate to the **B:** drive by typing **B:** and pressing Enter. Now you can use CP/M commands to run the programs on your custom disk image.

Please note that Z64K is still in development, and you might encounter issues or limitations when using the emulator, especially when running CP/M mode. If you experience problems, consider trying other CP/M emulators like CPMEmu or RunCPM, as mentioned in previous answers.



can z88dk while compiling to +cpm also create disk?



z88dk does not directly create CP/M disk images during compilation. However, you can easily create a disk image with your compiled program using **cpmtools**, as I mentioned in a previous answer. To do this, you'll need to follow these steps:

1. Compile your program using z88dk targeting CP/M. For example:

Copy code

```
zcc +cpm -o myprogram.com myprogram.c
```

This command will compile **myprogram.c** for the CP/M target and create a **myprogram.com** file.

2. Create a new CP/M disk image or use an existing one



okay but I need to copy my compiled file to C128 supported disk image



In that case, you can create a Commodore 128 CP/M disk image and copy your compiled file onto it using **cpmtools** with the appropriate disk format. Here's a step-by-step guide to create a C128 CP/M disk image and copy your compiled file to it:

1. Compile your program using z88dk targeting CP/M, as described in the previous answer.
2. Install **cpmtools** if you haven't already, following the instructions provided in an earlier answer.
3. Create a new CP/M disk image for the Commodore 128 using the **mkfs.cpm** command with the **c128-40** disk format. This format represents a 1541 disk drive with 40 tracks used by the Commodore 128 for CP/M:

 Copy code

```
mkfs.cpm c128_diskimage.d64 c128-40
```

This command will create a new disk image file named **c128\_diskimage.d64** with the Commodore 128 CP/M format.

4. Copy your compiled program to the disk image using the **cpmcp** command:

```
ruby
```

 Copy code

```
cpmcp -f c128-40 c128_diskimage.d64 myprogram.com 0:MYPROG.COM
```

This command copies the **myprogram.com** file to the **c128\_diskimage.d64** disk image as **MYPROG.COM** in user area 0.

5. Now you can use the created disk image with the Z64K emulator following the steps provided in a previous answer. However, instead of mounting an existing CP/M system disk image to Drive 8, use your newly created **c128\_diskimage.d64** disk image.

Please note that you might need to have a CP/M system and additional files like CCP and BDOS on the disk image for it to boot correctly on the emulator. You can copy the required files from an existing C128 CP/M system disk image using **cpmtools**.





I apologize for the confusion. It appears that the **c128-40** format is not a predefined format in **cpmtools**. In this case, you can create a custom disk definition file for the Commodore 128 CP/M disk format.

Follow these steps to create a custom disk definition file and then create a CP/M disk image for the Commodore 128:

1. Create a new text file and name it **c128diskdef**. Add the following lines to the file:

ruby

Copy code

```
# C128 CP/M format for 1541 disk drive with 40 tracks diskdef c128-40 seclen 256
tracks 40 sectrk 21 blocksize 1024 maxdir 64 skew 0 boottrk 0 os 2.2 end
```

This configuration defines a custom disk format **c128-40** for the Commodore 128 CP/M 1541 disk drive with 40 tracks.

2. Save the **c128diskdef** file in a location you can easily access.
3. Create a new CP/M disk image for the Commodore 128 using the **mkfs.cpm** command and reference the custom disk definition file you created:

mathematica

Copy code

```
mkfs.cpm -f c128-40 -D c128diskdef c128_diskimage.d64
```

This command will create a new disk image file named **c128\_diskimage.d64** with the custom Commodore 128 CP/M format.

4. Copy your compiled program to the disk image using the **cpmcp** command, referencing the custom disk definition file:

ruby

Copy code

```
cpmcp -f c128-40 -D c128diskdef c128_diskimage.d64 myprogram.com 0:MYPROG.COM
```

Now you have a Commodore 128 CP/M disk image with your compiled program on it. You can use this disk image with the Z64K emulator as described in my previous response.



have you also format / diskdef for d71 ?



> Yes, I can provide you with a custom disk definition for a D71 image (a 1571 disk drive with 70 tracks) for the Commodore 128 in CP/M mode. Create a new text file named **c128d71diskdef** and add the following



excuse me, is format for d71 really right? I have corrupted disk after cpmcp



I apologize for the confusion. It seems there was an error in the disk definition I provided. Let's create a new disk definition for a D71 image for the Commodore 128 in CP/M mode.

Create a new text file named **c128d71diskdef** and add the following lines to the file:

ruby

Copy code

```
# C128 CP/M format for 1571 disk drive with 70 tracks diskdef c128-d71 seclen  
256 tracks 70 sectrk 21 blocksize 1024 maxdir 64 skew 0 boottrk 0 os 2.2 end
```

Save the **c128d71diskdef** file in a location you can easily access.

Now, create a new CP/M disk image for the Commodore 128 using the **mkfs.cpm** command and reference the custom disk definition file you created:

mathematica

Copy code

```
mkfs.cpm -f c128-d71 -D c128d71diskdef c128_diskimage.d71
```

This command will create a new disk image file named **c128\_diskimage.d71** with the custom Commodore 128 CP/M format for a 1571 disk drive with 70 tracks.

Copy your compiled program to the disk image using the **cpmcp** command, referencing the custom disk definition file:

ruby

 Copy code

```
cpmcp -f c128-d71 -D c128d71diskdef c128_diskimage.d71 myprogram.com  
0:MYPROG.COM
```

Again, I apologize for the confusion earlier. This disk definition should work correctly for a D71 image in the Commodore 128 CP/M mode.



well, the cpm version in c128 is 3.0



>

You're correct; the CP/M version for the



do you have please format also for d81?



Certainly! Here is a custom disk definition for a D81 image (a 1581 disk drive with 80 tracks) for the Commodore 128 in CP/M mode. Create a new text file named **c128d81diskdef** and add the following lines to the file:

ruby

 Copy code



```
# C128 CP/M format for 1581 disk drive with 80 tracks diskdef c128-d81 seclen
256 tracks 80 sectrk 40 blocksize 1024 maxdir 128 skew 0 boottrk 0 os 3.0 end
```

Save the **c128d81diskdef** file in a location you can easily access.

Now, create a new CP/M disk image for the Commodore 128 using the **mkfs.cpm** command and reference the custom disk definition file you created:

mathematica

 Copy code

```
mkfs.cpm -f c128-d81 -D c128d81diskdef c128_diskimage.d81
```

This command will create a new disk image file named **c128\_diskimage.d81** with the custom Commodore 128 CP/M format for a 1581 disk drive with 80 tracks.

Copy your compiled program to the disk image using the **cpmcp** command, referencing the custom disk definition file:

ruby

 Copy code

```
cpmcp -f c128-d81 -D c128d81diskdef c128_diskimage.d81 myprogram.com
0:MYPROG.COM
```

This disk definition should work correctly for a D81 image in the Commodore 128 CP/M mode.



have you some emulator of msx on windows?



Yes, there are several MSX emulators available for Windows. One popular and widely recommended emulator is openMSX. openMSX is an open-source MSX emulator that supports a wide range of MSX hardware and has a user-friendly interface.

You can download openMSX for Windows from the official website: <https://openmsx.org/>

After downloading the installer or the ZIP package, follow the installation instructions or extract the contents of the ZIP file to a folder. To run the emulator, launch the openMSX executable. Keep in mind that you may need to provide the appropriate MSX system ROMs for the emulator to function correctly. You can find more information about setting up openMSX in the documentation available on the website.

Another MSX emulator for Windows is BlueMSX. Although it is no longer actively developed, it remains a popular choice for MSX emulation. You can download BlueMSX from its official website: <http://www.bluemsx.com/>

Download the installer or the ZIP package and follow the installation instructions or extract the contents of the ZIP file to a folder. As with openMSX, you'll need to provide the appropriate MSX system ROMs for the emulator to work correctly. The BlueMSX website contains additional information about setting up and configuring the emulator.



allows they filesystem drive?



Both openMSX and blueMSX allow you to use file system drives or directory-based drives, which means you can access files on your computer's file system as if they were on a virtual MSX disk drive.

For openMSX, you can use the "dir-as-disk" feature to mount a directory on your computer as a virtual disk drive. To do this, open the openMSX console (press F10) and enter the following command:

arduino

 Copy code

```
diska insert_dir "/path/to/your/directory"
```

Replace **/path/to/your/directory** with the actual path of the directory you want to mount. This command will mount the specified directory as a virtual disk drive in drive A of the emulator.

For blueMSX, you can create a directory-based disk image. To do this, follow these steps:

1. Launch blueMSX and configure the emulator for the MSX system you want to use.
2. Go to "File" > "Cartridge Slot 1" > "Insert Directory as Disk" or "File" > "Disk Drive A" > "Insert Directory as Disk" depending on the system configuration.
3. Browse to the directory you want to use as a virtual disk drive and select it.

After completing these steps, the selected directory will be mounted as a virtual disk drive in the emulator, and you can access its contents as if they were on a real MSX disk drive.



have you also sord m5 emulator?



There are not many emulators specifically for the Sord M5 computer, but you can use MAME (Multiple Arcade Machine Emulator), which emulates a wide range of computer systems and arcade machines, including the Sord M5.

To use MAME as a Sord M5 emulator, follow these steps:

1. Download MAME for Windows from the official website: <https://www.mamedev.org/>
2. Extract the downloaded MAME package to a folder on your computer.
3. Obtain the Sord M5 ROMs. Due to legal reasons, I cannot provide direct links to ROM files. However, you can search for them online or try to obtain them from an original Sord M5 system.
4. Create a folder named "roms" in the same directory where you extracted MAME.
5. Place the Sord M5 ROM files in the "roms" folder.
6. Open a command prompt (CMD) in the MAME directory. You can do this by pressing Shift + Right-click in the MAME folder and choosing "Open command window here" or "Open PowerShell window here."
7. In the command prompt, type the following command and press Enter:

 Copy code

```
mame64 sordm5
```

MAME will launch and start the Sord M5 emulation. Make sure to consult the MAME documentation for information on configuring the emulator, such as setting up controllers, keyboard mappings, and display options.

---